浙江大学

本科实验报告

课程名称：人工智能

姓　　名：王若鹏

学　　院：信息与电子工程学院

专　　业：电子科学与技术

学　　号：3170105582

指导教师：王东辉

2020 年 4 月 10 日

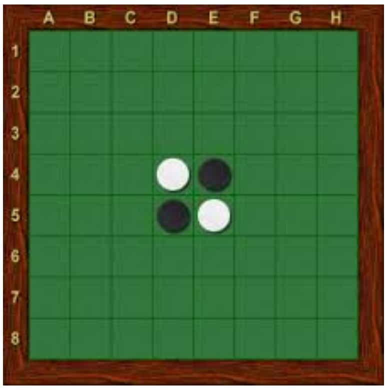
基于 MCTS 的黑白棋算法设计

1. 背景

黑白棋 (Reversi)，也叫苹果棋，翻转棋，是一个经典的策略性游戏。一般棋子双面为黑白两色，故称“黑白棋”。因为行棋之时将对方棋子翻转，则变为己方棋子，故又称“翻转棋”(Reversi) 。棋子双面为红、绿色的称为“苹果棋”。它使用 8x8 的棋盘，由两人执黑子和白子轮流下棋，最后子多方为胜方。

游戏规则：

棋局开始时黑棋位于 E4 和 D5 ，白棋位于 D4 和 E5，如图所示。



1.黑方先行，双方交替下棋。

2.一步合法的棋步包括：在一个空格新落下一个棋子，并且翻转对手一个或多个棋子；新落下的棋子必须落在可夹住对方棋子的位置上，对方被夹住的所有棋子都要翻转过来，可以是横着夹，竖着夹，或是斜着夹。夹住的位置上必须全部是对手的棋子，不能有空格；一步棋可以在数个（横向，纵向，对角线）方向上翻棋，任何被夹住的棋子都必须被翻转过来，棋手无权选择不去翻某个棋子。

3.如果一方没有合法棋步，也就是说不管他下到哪里，都不能至少翻转对手的一个棋子，那他这一轮只能弃权，而由他的对手继续落子直到他有合法棋步可下。

4.如果一方至少有一步合法棋步可下，他就必须落子，不得弃权。

5.棋局持续下去，直到棋盘填满或者双方都无合法棋步可下。

6.如果某一方落子时间超过 1 分钟或者连续落子 3 次不合法，则判该方失败。

2. 实验要求

- 使用“最小最大搜索”、“Alpha-Beta 剪枝搜索”或“蒙特卡洛树搜索算法”实现 miniAlphaGo for Reversi（三种算法择一即可）。
- 使用 Python 语言。
- 算法部分需要自己实现，不要使用现成的包、工具或者接口。

3. 开发环境

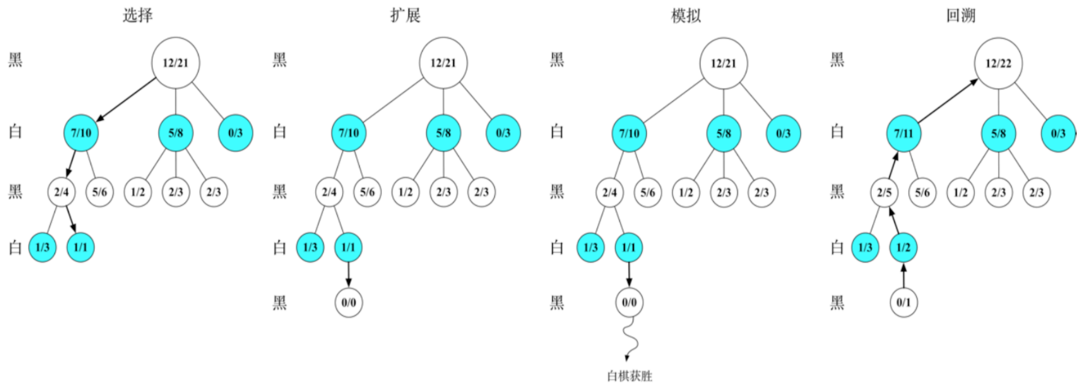
- 编程语言：Python 3
- 开发工具：Anaconda、Python shell
- 开发平台：Mo 人工智能训练平台
- 操作系统：MacOS Mojave 10.14.6

4. 算法原理与设计

4.1 蒙特卡洛树搜索

蒙特卡洛树搜索分为 4 个部分：选择、扩展、模拟、回溯。

- 选择：指从根节点开始，选择连续的子节点向下至叶子节点 L。
- 扩展：指除非任意一方的输赢导致游戏结束，否则 L 会创建一个或多个子节点。
- 模拟：从 L 的子节点中随机布局。
- 回溯：使用布局结果更新从 L 到根节点路径上的节点信息。



对于任一节点的信息，使用字典 dic 存储，key 为从根节点到当前状态的路径 route，格式为字符串，如"A2B4C6D8"表示行棋顺序为 A2-B4-C6-D8。每个 key 含有一个长度为 2 的列表[a,b]，a 为已获得的奖励 reward，b 为该状态的总访问次数。以下表格是算法伪代码的符号注解：

$\boldsymbol{S}$	状态集
$\boldsymbol{A(s)}$	在状态s能够采取的有效行动的集合
$\boldsymbol{s(v)}$	节点v所代表的状态
$\boldsymbol{a(v)}$	所采取的行动导致到达节点v
$\boldsymbol{f}: S \times \mathrm{A} \rightarrow S$	状态转移函数
$\boldsymbol{N(v)}$	节点v被访问的次数
$\boldsymbol{Q(V)}$	节点v所获得的奖赏值
$\boldsymbol{\Delta(v,p)}$	玩家p选择节点v所得到的奖赏值

4.2 搜索树算法 UCTSearch

首先是实现整个 MCTS 算法的主体算法 UCTSearch，此外我还增加了一些跳出循环的条件: ①选举出来的节点已经是叶节点，②已经遍历到预先设定的最大层级，③时间已消耗至 20s 以上(一局时间为 60s)。

伪代码如下：

function UCTSEARCH(s_0)
    create root node v_0 with state s_0
    while within computational budget **do**
        $v_l \leftarrow$ TREEPOLICY(v_0)
        $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
        BACKUP(v_l, Δ)
    return a(BESTCHILD($v_0, 0$))

MCTS 的核心代码结构如下：

```
while (datetime.datetime.now() - start_time).seconds < 20:
    moboard = deepcopy(board) #模拟棋盘
    route = "" #记录路径
    route = self.tree_policy(moboard, player, route, dic) #选择/扩展
    reward = self.default_policy(moboard, player, weight) #模拟
    dic = self.backup(route, dic, reward) #回溯

action = self.best_action(board, ready, dic, player, prior)

return action
```

4.3 选择算法 tree_policy

首先获取目标棋局的所有合法招式，如果某一个招式对应的棋局未被搜索过，则调用 expand()并返回该招式对应的棋局，否则选择所有招式中当前最优的招式对应的棋局，继续调用 tree_policy()，进行递归，直到达到预先设定的深度。选择算法伪代码如下所示：

function TREEPOLICY(v)
    while v is nonterminal **do**
        if v not fully expanded **then**
            return EXPAND(v)
        else
            $v \leftarrow$ BESTCHILD(v, Cp)
    return v

4.4 扩展算法 expand

扩展节点，先在子节点中选择一个未被尝试过的，将这一新的节点更新至搜索路径中，并为这一新的状态赋予初始值[0,0]，添加到字典 dic 中，伪代码如下：

$$
\begin{array}{l}
\textbf{function } \text{EXPAND}(v) \\
\quad \text{choose } a \in \text{untried actions from } A(s(v)) \\
\quad \text{add a new child } v' \text{ to } v \\
\quad\quad \text{with } s(v') = f(s(v), a) \\
\quad\quad \text{and } a(v') = a \\
\quad \textbf{return } v'
\end{array}
$$

4.5 最佳子节点算法 best_child

实现计算 UCB 得出估值最高的子节点的算法，伪代码如下。这里考虑了己方节点和对方节点对于估值选择的不同决策：己方节点将选择子节点中估值最大的(Max)，对方节点将选择子节点估值最小的(Min)。

$$
\begin{array}{l}
\textbf{function } \text{BESTCHILD}(v, c) \\
\quad \textbf{return } \underset{v' \in \text{children of } v}{\arg\max} \dfrac{Q(v')}{N(v')} + c\sqrt{\dfrac{2 \ln N(v)}{N(v')}}
\end{array}
$$

其中 UCB 的计算代码如下，需要调用 math 库：

```
def ucb(self,nu,qv,nv,moplayer):
    c = 2 #UCB算式的常数项
    if moplayer == self.color:
        result = 1.0*qv/nv + c * math.sqrt(2.0*math.log(nu)/nv)
    else:
        result = 1 - 1.0*qv/nv + c * math.sqrt(2.0*math.log(nu)/nv)
    return result
```

对于最终根节点选择行棋的策略，我编写了另一个函数 best_action。创建了一个优先度矩阵 prior，标记了棋盘上各个位置的优先程度，默认四个顶点为 9，其余为 1。在其中增加了对行棋后果的判断：若行这步棋之后能够促使对方占领四个顶点，则降低该位置的优先度至 0.1。最后计算子节点的 UCB 并乘上优先度，得到最适合的行棋策略。

```
prior = {'A1':9,'B1':1,'C1':1,'D1':1,'E1':1,'F1':1,'G1':1,'H1':9,
         'A2':1,'B2':1,'C2':1,'D2':1,'E2':1,'F2':1,'G2':1,'H2':1,
         'A3':1,'B3':1,'C3':1,'D3':1,'E3':1,'F3':1,'G3':1,'H3':1,
         'A4':1,'B4':1,'C4':1,'D4':1,'E4':1,'F4':1,'G4':1,'H4':1,
         'A5':1,'B5':1,'C5':1,'D5':1,'E5':1,'F5':1,'G5':1,'H5':1,
         'A6':1,'B6':1,'C6':1,'D6':1,'E6':1,'F6':1,'G6':1,'H6':1,
         'A7':1,'B7':1,'C7':1,'D7':1,'E7':1,'F7':1,'G7':1,'H7':1,
         'A8':9,'B8':1,'C8':1,'D8':1,'E8':1,'F8':1,'G8':1,'H8':9}
#记录实时权重信息
```

4.6 模拟算法 default_policy

实现模拟至终局的算法 default_policy，伪代码如下:

```
function DEFAULTPOLICY(s)
    while s is non-terminal do
        choose a ∈ A(s) uniformly at random
        s ← f(s, a)
    return reward for state s
```

在本算法中，模拟过程的每一步决策，抛弃了随机策略，而采用参考当前棋盘每个位置的权重，选择权重最大的有效棋步来下，如此能够获得更为准确的终局结果。将棋盘各个位置的权重设置如下：

```
weight = {'A1':10,'B1':-6,'C1':8,'D1':6,'E1':6,'F1':8,'G1':-6,'H1':10,
          'A2':-6,'B2':-8,'C2':-4,'D2':-3,'E2':-3,'F2':-4,'G2':-8,'H2':-6,
          'A3':8,'B3':-4,'C3':7,'D3':4,'E3':4,'F3':7,'G3':-4,'H3':8,
          'A4':6,'B4':-3,'C4':4,'D4':0,'E4':0,'F4':4,'G4':-3,'H4':6,
          'A5':6,'B5':-3,'C5':4,'D5':0,'E5':0,'F5':4,'G5':-3,'H5':6,
          'A6':8,'B6':-4,'C6':7,'D6':4,'E6':4,'F6':7,'G6':-4,'H6':8,
          'A7':-6,'B7':-8,'C7':-4,'D7':-3,'E7':-3,'F7':-4,'G7':-8,'H7':-6,
          'A8':10,'B8':-6,'C8':8,'D8':6,'E8':6,'F8':8,'G8':-6,'H8':10}
```

模拟过程中选择权重最大的策略来模拟执行。对于终局结果的估值，我们采用己方胜利则 reward 为 1，己方失败则 reward 为 0，打平则 reward 为 0.5 的策略。

4.7 回溯算法 backup

对于模拟终局的结果，作为 reward 需要回溯更新每一个祖先节点，遍历方法时每次把 route 减去一个位置作为 key 更新字典的值，代码如下:

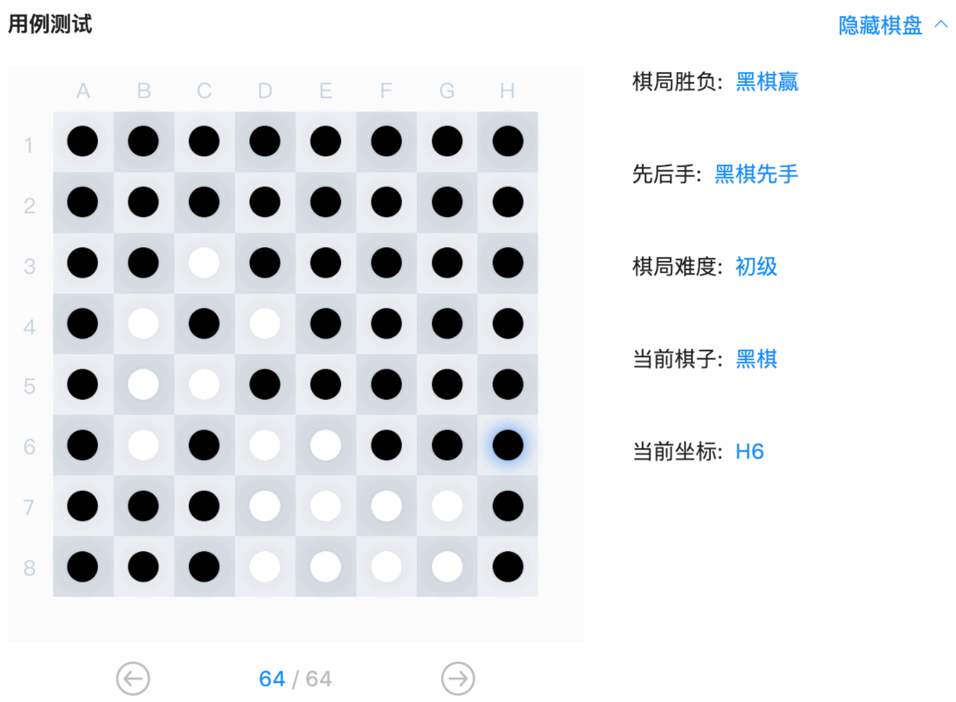
```
function BACKUP(v, Δ)
    while v is not null do
        N(v) ← N(v) + 1
        Q(v) ← Q(v) + Δ(v, p)
        v ← parent of v
```

```
def backup(self, route, dic, reward):
    #回溯路径上的每个节点，更新其[win,all]
    n = len(route)
    while n > 0:
        node = route[0:n]
        dic[node][0] += reward
        dic[node][1] += 1
        n -= 2
    return dic
```

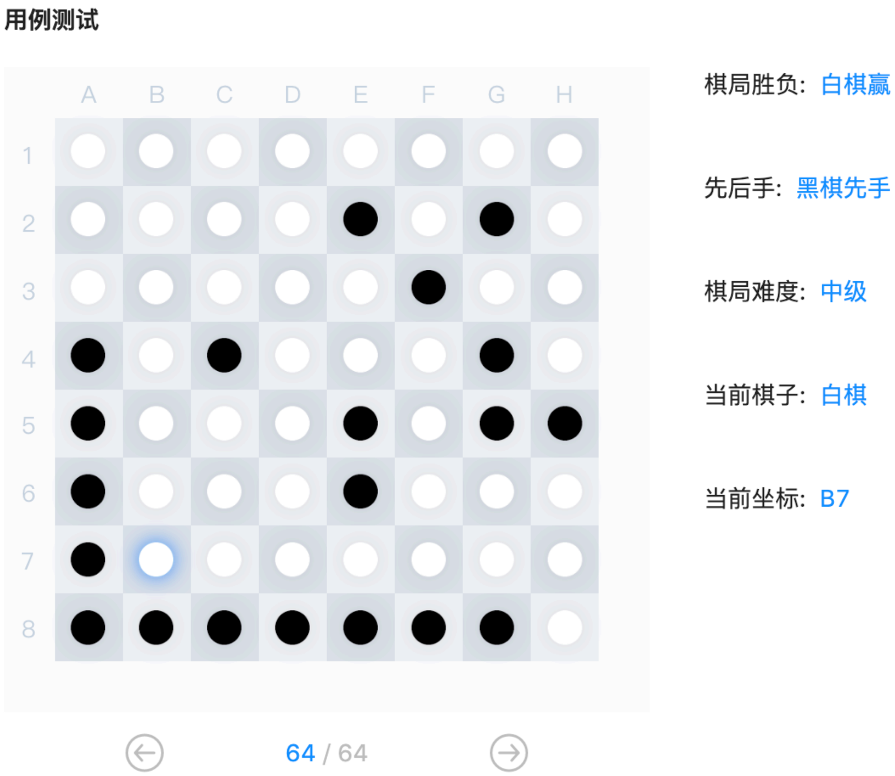
5. 对弈结果

接口测试通过后，分别与初级、中级、高级棋手对弈。对于初级棋手和高级棋手，基本都能下赢，但一直难以打败中级棋手。

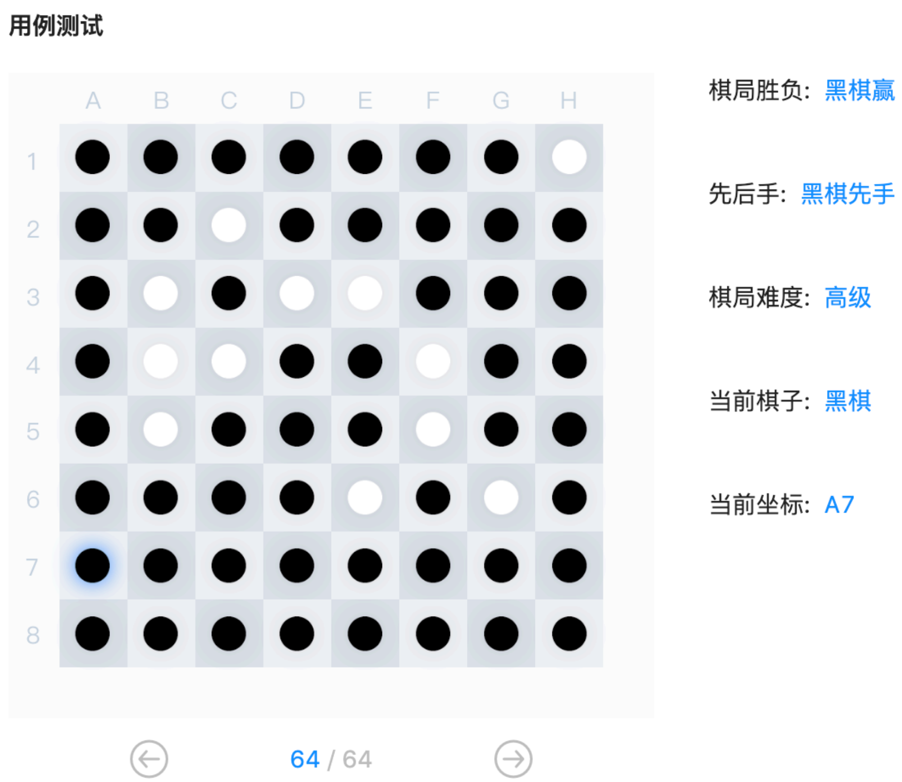
（1）对战初级棋手：执白棋赢 22 子，执黑棋赢 32 子



（2）对战中级棋手：执白棋输 18 子，执黑棋输 24 子



（3）对战高级棋手：执白棋赢 38 子，执黑棋赢 40 子



6. 评价

- 优点：蒙特卡洛树搜索算法比 alpha-beta 剪枝搜索和 minimax 更复杂，效果较好。
- 缺点：优化程度还有很大的改善空间，甚至可以利用强化学习来优化参数。

7. 心得体会

这是我第一次完成人工智能相关的项目，整个过程让我学到了很多。

首先，准备工作的工作量是非常大的。起初，我连黑白棋都不会下。为此在游戏网站上特意练习了许久，摸清了一些棋法和套路，这样会大大方便将来对算法的优化过程。同时，由于这是我第一次使用 python 完成项目，以前也没有学过 python 语言，我在老师推荐的“菜鸟教程”上系统的学习了一遍 python 语法和面向对象的思维。

其次，便是写代码和 debug 的漫长过程。我花了 3 个整天时间写好了整个项目，而且每写好一个模块还专门把它单独拿出来测试，这样会给最后系统测试的 debug 带来方便。通过接口测试、成功赢了初级棋手的那一刻，内心的自豪感是很强烈的。

最后便是算法的优化，我根据黑白棋的下棋要领，给算法添加了一些新的规则。比如优秀占据四个顶点位置，绝不能下倒数第二个顶点位置等等，胜率有了一定的提高，最好的情况下能赢高级棋手 38 个子。