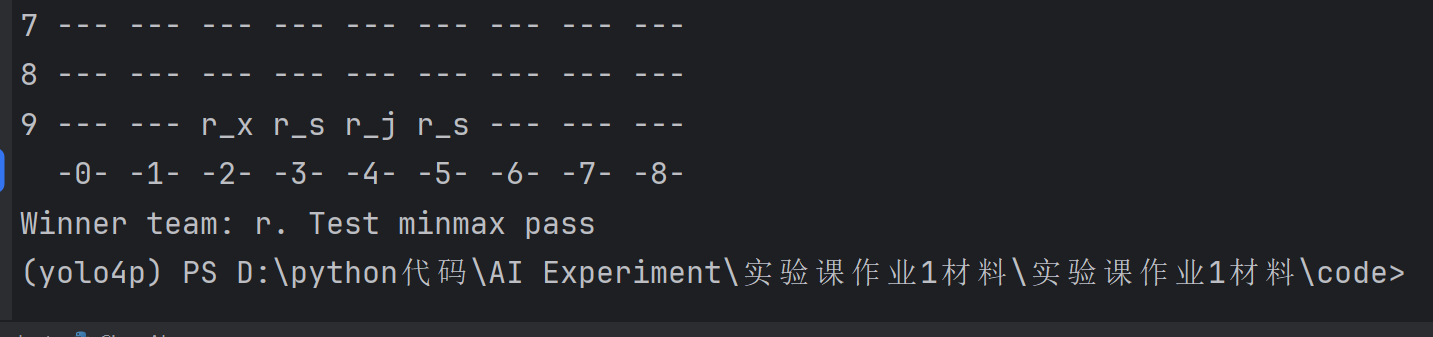
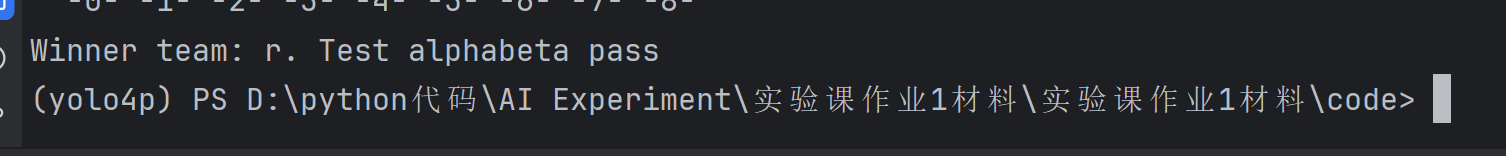
## 算法设计思路与实现细节说明





### Min\_max的核心思维：

min\_max本质是深度优先搜索，每个父节点会根据其是max回合还是min回合决定从子字节的中选取max，min的节点。在设计函数变量时，因为中国象棋分为红黑两方，所以需要变量记录当前玩家所属的阵营，因为深度搜索到终局的成本太高无法接受，所以需要设置最大搜索深度变量，然后我们需要一个变量棋盘记录当前棋盘的所有信息。

所以min\_max整体结构如下:

def min\_max(self, depth, chessboard: ChessBoard):

def max\_value(depth, current\_player, chessboard):

value = max(value, current\_value)

return value

def min\_value(depth, current\_player, chessboard):

value = min(value, current\_value)

return value

# 主程序

best\_score = -float('inf')

current\_player = self.team

current\_value = max(value,min\_value(depth - 1, next\_player , next\_chessboard))

return

考虑棋盘终止条件为depth == 0 or chessboard.judge\_win()，当条件满足则return

evaluate(chessboard)

对于一个父节点，他的子节点应列举己方阵营所有棋子的所有可达点：

for chess in chessboard.get\_chess():

if chess.team != current\_player : continue

for (row,col) in chessboard.get\_put\_down\_position():

将所有因素组合可得：

def min\_max(self, depth, chessboard: ChessBoard):

def max\_value(depth, current\_player, chessboard):

# 终止条件：深度耗尽或当前玩家已胜

if depth == 0 or chessboard.judge\_win(current\_player):

return self.evaluate\_class.evaluate(chessboard)

value = -float('inf')

for chess in chessboard.get\_chess():

if chess.team != current\_player:

continue # 仅处理当前玩家的棋子

positions = chessboard.get\_put\_down\_position(chess)

for (row, col) in positions:

next\_board = self.get\_tmp\_chessboard(chessboard, chess, row, col)

# 递归调用 min\_value，传递正确的玩家和深度

current\_value = min\_value(depth - 1, self.get\_nxt\_player(current\_player), next\_board)

value = max(value, current\_value)

return value

def min\_value(depth, current\_player, chessboard):

if depth == 0 or chessboard.judge\_win(current\_player):

return self.evaluate\_class.evaluate(chessboard)

value = float('inf')

for chess in chessboard.get\_chess():

if chess.team != current\_player:

continue

positions = chessboard.get\_put\_down\_position(chess)

for (row, col) in positions:

next\_board = self.get\_tmp\_chessboard(chessboard, chess, row, col)

# 递归调用 max\_value，传递正确的玩家和深度

current\_value = max\_value(depth - 1, self.get\_nxt\_player(current\_player), next\_board)

value = min(value, current\_value)

return value

# 主逻辑

best\_score = -float('inf')

current\_player = self.team

for chess in chessboard.get\_chess():

if chess.team != current\_player:

continue

positions = chessboard.get\_put\_down\_position(chess)

for (row, col) in positions:

next\_board = self.get\_tmp\_chessboard(chessboard, chess, row, col)

# 初始调用 min\_value，传递 depth-1 和对方玩家

score = min\_value(self.max\_depth - 1, self.get\_nxt\_player(current\_player), next\_board)

if score > best\_score:

best\_score = score

self.old\_pos = [chess.row, chess.col]

self.new\_pos = [row, col]

return

### alphabeta（）的核心逻辑：

alphabeta剪枝是在minmax的基础上剔除掉不可能被选择的节点，用alpha记录max节点能拿到的最大值，用beta记录min节点能拿到的最小值。所谓的剔除掉不可能被选择的节点既是：在两名玩家都是理性人的情况下，当在计算max节点的所有子节点时，如果有子节点的返回值大于max的父节点min存储的beta值，min节点不可能进入该max节点，对于该max节点的遍历可以直接退出。对于min节点也是同理。

以max\_value()为例：

def max\_value(current\_depth, current\_player, alpha, beta):

if current\_depth == 0 or chessboard.judge\_win(current\_player):

return self.evaluate\_class.evaluate(chessboard)

max\_value = -float('inf')

# 按棋子价值排序（车、马等高价值棋子优先）

chesses = sorted(

[chess for chess in chessboard.get\_chess() if chess.team == current\_player],

key=lambda x: self.evaluate\_class.single\_chess\_point[x.name],

reverse=True

)

for chess in chesses:

positions = chessboard.get\_put\_down\_position(chess)

# 启发式排序：优先吃子或靠近对方将

'''positions.sort(

key=lambda pos: self.\_move\_heuristic(chess, pos, chessboard),

reverse=True

)'''

old\_row,old\_col = chess.row,chess.col

for (new\_row, new\_col) in positions:

target\_chess = chessboard.chessboard\_map[new\_row][new\_col]

# 执行移动

chessboard.move\_chess\_silent(old\_row, old\_col, new\_row, new\_col)

# 评估当前局面

current\_value = min\_value(current\_depth - 1, self.get\_nxt\_player(current\_player), alpha,beta)

# 回溯

chessboard.move\_chess\_silent(new\_row, new\_col, old\_row, old\_col)

chessboard.chessboard\_map[new\_row][new\_col] = target\_chess

max\_value = max(max\_value,current\_value)

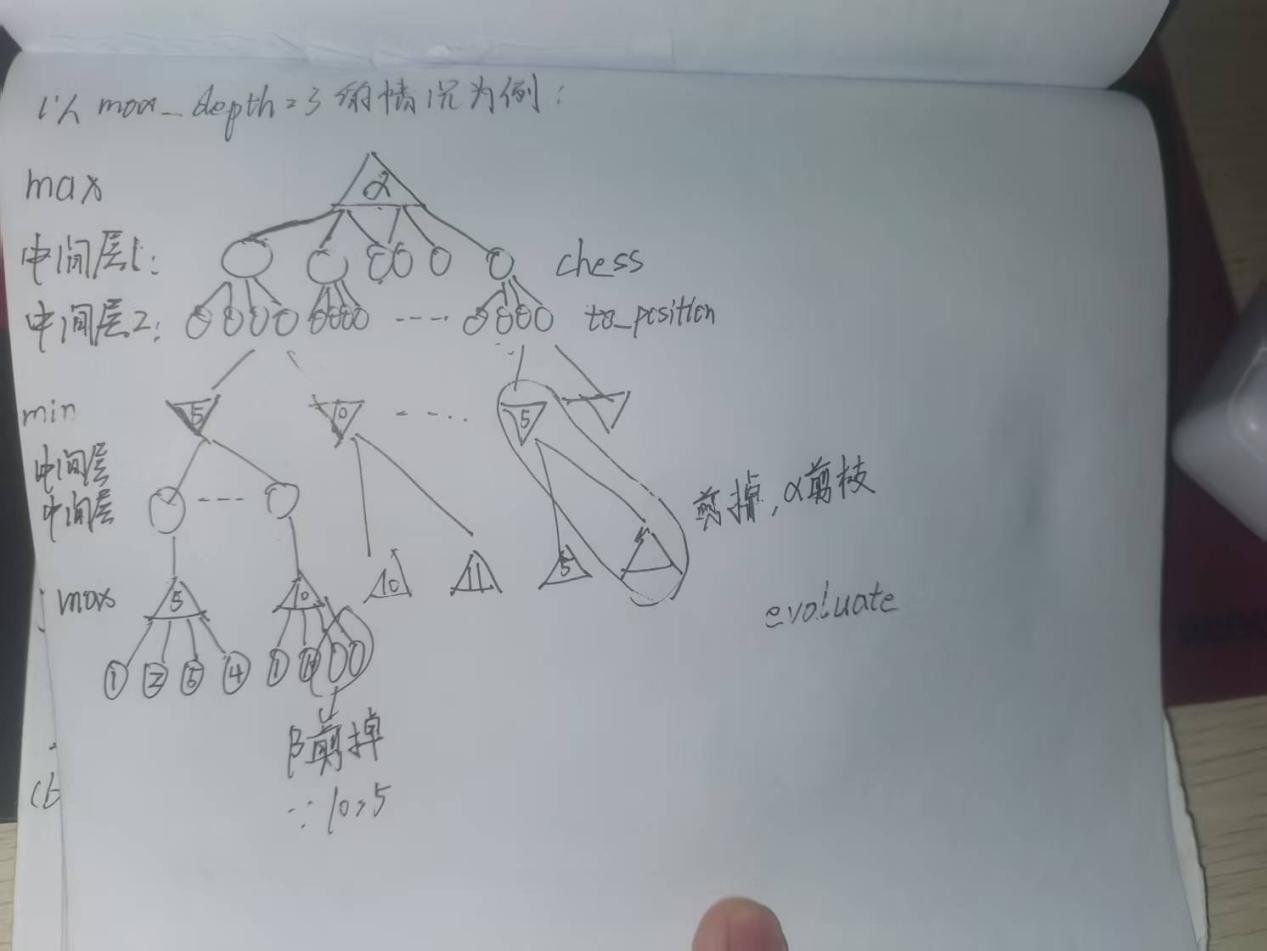
if max\_value >= beta:

return max\_value # Beta剪枝

alpha = max(alpha, max\_value)

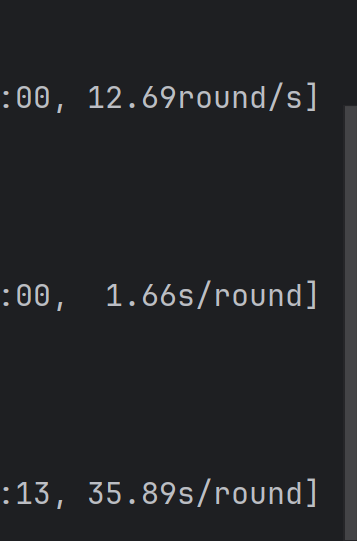
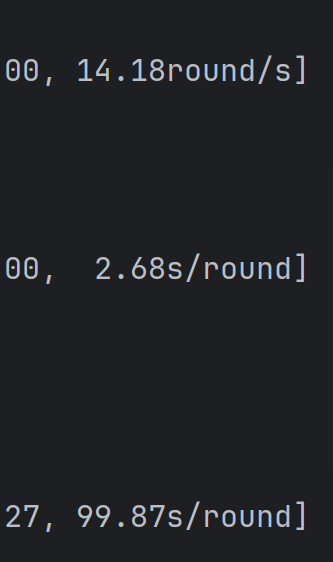
return max\_value

## 流程图绘制（2 分）



## 算法运⾏效率对⽐实验（4 分）

左图中是minmax的depth为1，2，3的运行时间，右图中是alphabeta的depth为1，2，3的和相同pikafish对弈一轮（一共5步）的时间，因为层数的增加对于minmax和alphabeta算法时间复杂度的影响为指数级的，可以看出



|  |  |  |  |
| --- | --- | --- | --- |
| Depth | 1 | 2 | 3 |
| Minmax(s) | 1/14.18 | 2.68 | 99.87 |
| Alphabeta(s) | 1/12.69 | 1.66 | 35.89 |
| 提升 | 0.895 | 1.614 | 2.783 |

## d. 学习体会与过程总结（1 分）

提高胜率最简单的办法就是加层数，但是因为层数增加时间复杂度约是指数级增加，所以看似是最简单的办法，但是对于计算机其实是最困难的办法。alphabeta剪枝，通过减去暴力搜索过程中不可能被选择的部分极大的优化了时间复杂度。

### f. 与 PikaFish AI 对弈 / 修改评估函数（+1 分）

为了提高对阵pikafish的胜率，我做出一下几点修改：

1.因为alphabeta剪枝在残局的时候没有进行特判，所以容易导致将最后的马，炮，车，卒和对方对换导致和棋或者输棋，因此我设计了残局判断函数和防止对换函数，同时可以在alphabeta剪枝的时候加入进入残局的判断，如果进入残局则加多搜索层数寻找杀棋。

def \_is\_endgame(self, chessboard):

"""检测是否进入残局（对方棋子数 <= 2）"""

opponent\_pieces = [chess for chess in chessboard.get\_chess() if chess.team != self.team]

return len(opponent\_pieces) <= 2

def \_is\_exchange\_move(self, chess, new\_pos, chessboard):

"""判断移动是否为兑换（牺牲高价值棋子）"""

target = chessboard.chessboard\_map[new\_pos[0]][new\_pos[1]]

return target and self.evaluate\_class.single\_chess\_point[target.name] >= self.evaluate\_class.single\_chess\_point[

chess.name]

2.为了增加困死的情况，我还修改了evaluate函数，通过棋子的协同，减少对方将可移动的选择的评分，使得alphabetaAgent更容易选择将对方困死的赢法  
 def evaluate(self, chessboard: ChessBoard):

point = 0

general\_pos = None # 对方将的位置

for chess in chessboard.get\_chess():

# 基础分值和位置分

point += self.get\_single\_chess\_point(chess)

point += self.get\_chess\_pos\_point(chess)

# 记录对方将的位置

if chess.name == "j" and chess.team != self.team:

general\_pos = (chess.row, chess.col)

# 残局激励：如果我方有cmzp，且对方将暴露，增加围剿分

if general\_pos:

attacker\_count = 0 # 我方可攻击对方将的棋子数

for chess in chessboard.get\_chess():

if chess.team == self.team and chess.name in ["c", "m", "z","p"]:

# 判断是否在对方将的攻击范围内

positions = chessboard.get\_put\_down\_position(chess)

if general\_pos in positions:

attacker\_count += 1

point += attacker\_count \* 200 # 每个可攻击将的棋子加200分

opponent\_general = None

for chess in chessboard.get\_chess():

if chess.name == "j" and chess.team != self.team:

opponent\_general = chess

break

if opponent\_general:

movable\_positions = chessboard.get\_put\_down\_position(opponent\_general)

freedom\_penalty = -len(movable\_positions) \* 100 # 对方将可移动位置越少，我方得分越高

point += freedom\_penalty

# 棋子协同奖励（车、马、炮与其他棋子配合）

for chess in chessboard.get\_chess():

if chess.team == self.team and chess.name in ["c", "m", "p"]:

nearby\_allies = 0

for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

x, y = chess.row + dx, chess.col + dy

if 0 <= x < 10 and 0 <= y < 9:

if chessboard.chessboard\_map[x][y] and chessboard.chessboard\_map[x][y].team == self.team:

nearby\_allies += 1

point += nearby\_allies \* 50 # 每有一个友方相邻，加50分

return point

1. 在检索时优先检索价值高的棋子，移动位置时从得分高的位置开始移动

虽然因为pikafish返回了NoneType导致错误，但是从得到的结果来看win大大提升

ChessAI1.py为更改后，ChessAI.py为更改前

