

a. 算法设计思路与实现细节说明

```
7 --- --- --- --- --- --- --- --- ---
8 --- --- --- --- --- --- --- --- ---
9 --- --- r_x r_s r_j r_s --- --- ---
-0- -1- -2- -3- -4- -5- -6- -7- -8-
Winner team: r. Test minmax pass
(yolo4p) PS D:\python代码\AI Experiment\实验课作业1材料\实验课作业1材料\code>
```

```
-0- -1- -2- -3- -4- -5- -6- -7- -8-
Winner team: r. Test alphabeta pass
(yolo4p) PS D:\python代码\AI Experiment\实验课作业1材料\实验课作业1材料\code>
```

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 回合决定从子节点的中选取 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]

```

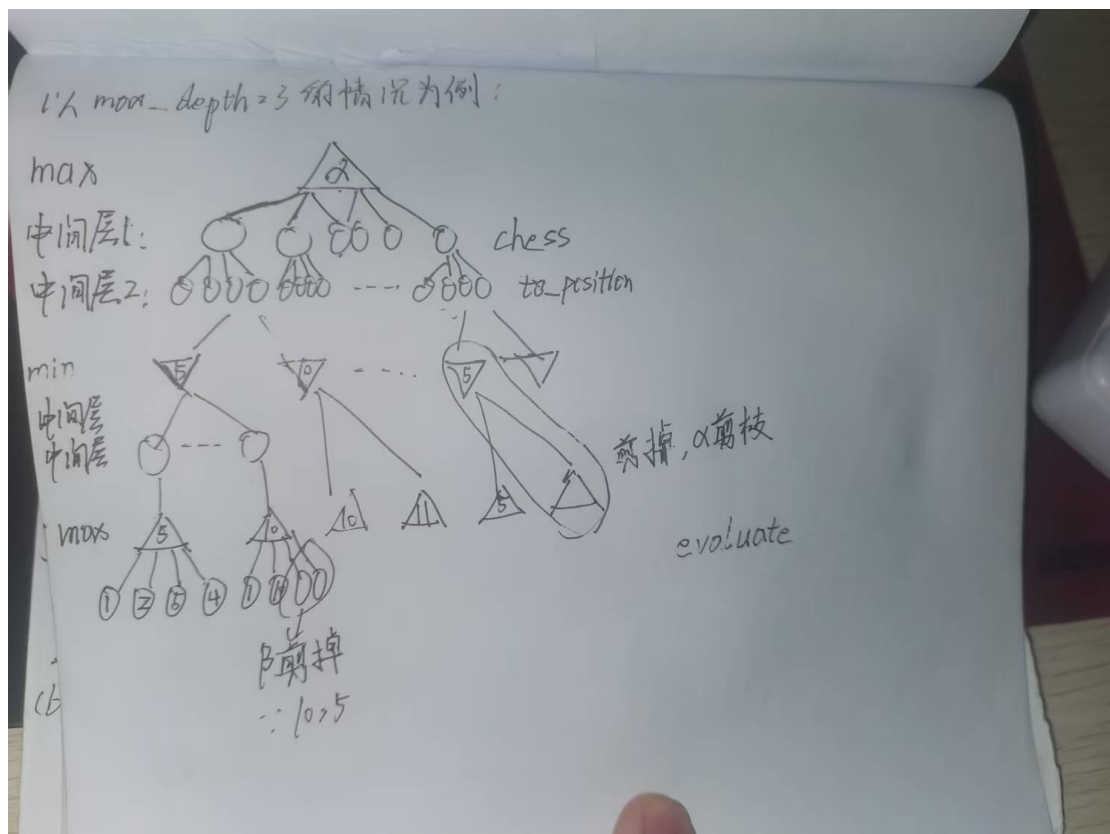
```

# 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

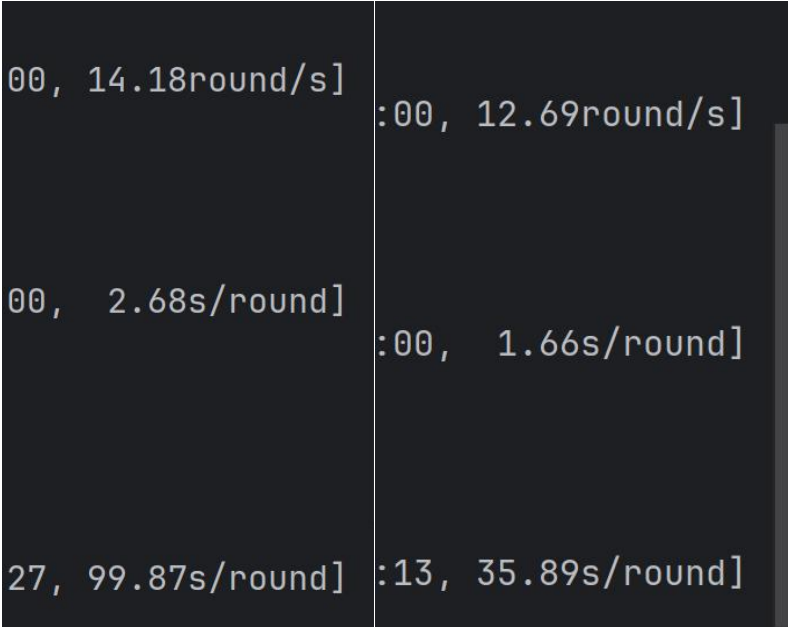
```

b. 流程图绘制 (2 分)

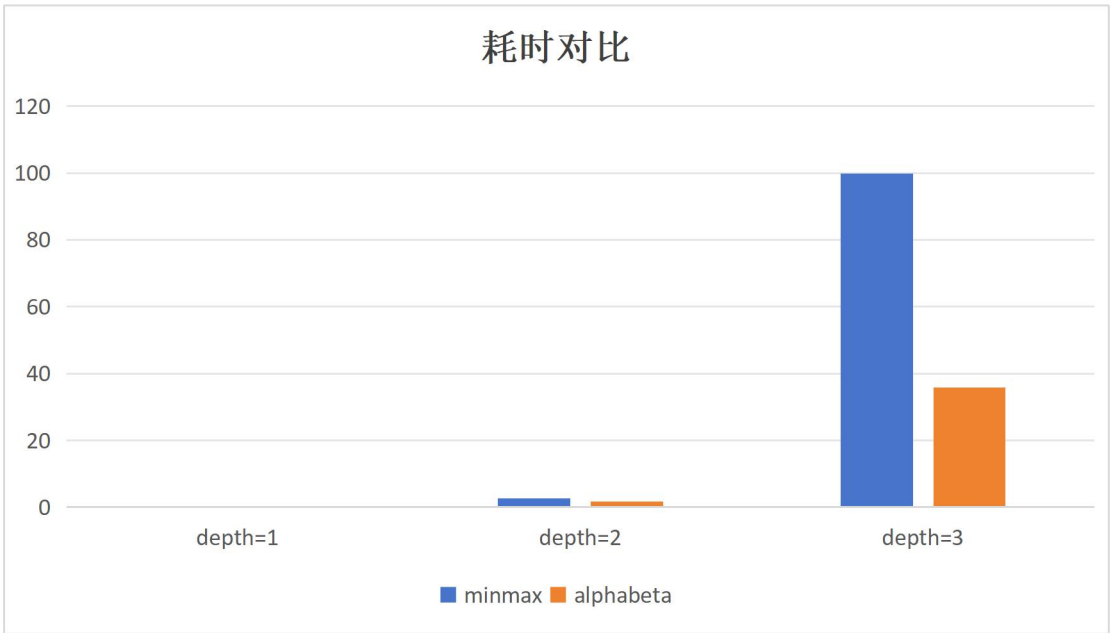


c. 算法运行效率对比实验（4 分）

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

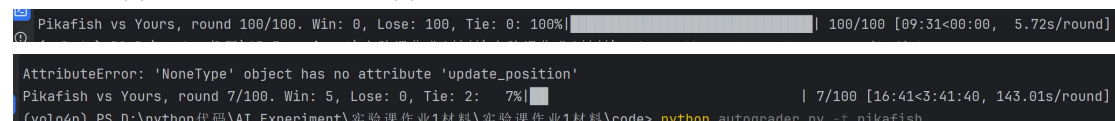
# 棋子协同奖励（车、马、炮与其他棋子配合）
for chess in chessboard.get_chess():
    if chess.team == self.team and chess.name in ["c", "m", "p"]:
        nearby_allies = 0
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            x, y = chess.row + dx, chess.col + dy
            if 0 <= x < 10 and 0 <= y < 9:
                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

```

3. 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



```

Pikafish vs Yours, round 100/100. Win: 0, Lose: 100, Tie: 0: 100% | 100/100 [09:31<00:00, 5.72s/round]
AttributeError: 'NoneType' object has no attribute 'update_position'
Pikafish vs Yours, round 7/100. Win: 5, Lose: 0, Tie: 2: 7% | 7/100 [16:41<3:41:40, 143.01s/round]
(vol04n) PS D:\python代码\AI Experiment\实验课作业1材料\实验课作业1材料\code> python autograder.py -t pikafish

```