# 中山大学计算机学院

# 人工智能

# 本科生实验报告

课程名称：Artificial Intelligence

| 学号 | 22336259 | 姓名 | 谢宇桐 |
|---|---|---|---|

## 一、 实验题目

### 博弈树搜索--中国象棋

## 二、 实验内容

### 1. 算法原理

博弈树启发式搜索策略即为结点排序技术。适用于此处的两个玩家对抗博弈的一种算法。在每次走出下一步时，玩家需要经过搜索算出对自己利益最大且对对方最不利的一步。

**极小极大算法（Minimax）：**

即深度优先搜索+逆向归纳法，是博弈树搜索的基本方法。首先假定，有一可以对所有的博弈进行评估的值。当值大于 0 时，表示博弈对我方有利，对对方不利。小于 0 时，表示博弈对我方不利，对对方有利：

1、当轮到我方走棋时，首先按照一定的搜索深度生成出给定深度 d 以内的所有状态，计算所有叶节点的评价函数值。然后从 d-1 层节点开始逆向计算：

2、对于我方要走的节点（用 MAX 标记，称为极大节点）取其子节点中的最大值为该节点的值（因为我方总是选择对我方有利的棋）。

3、对于对方要走的节点（用 MIN 标记，称为极小节点）取其子节点中的最小值为该节点的值（对方总是选择对我方不利的棋）。

4、一直到计算出根节点的值为止。获得根节点取值的那一分枝，即为所选择的最佳走步。

**Alpha-Beta 剪枝：**

由于我们的搜索空间仍然非常庞大，在最开始的几层，可做的决策是相当多的。所以我们用到了 α — β 剪枝技术，减少没有必要的搜索，及时终止，从而加快了搜索速度。

算法先到达底层，然后根据往上往下传承，以及更新规则更新。剪枝就发生在这个过程：Max 节点:其取子节点的最大值， α 只更新更大，为下限。Max 传承其父节点 Min 的 β 值，为上限。α 根据 Max 节点的子枝信息由空更新到最大。在更新过程中如果出现 α >= β，则 Max 节点的余枝被剪枝。如果遍历 Max 节点的子节点均未出现 α >= β，将 β 更新成这个遍历以后的 α 。

### 2. 关键代码展示（可选）

MyAI.py:

```python
class MyAI(object):
    def __init__(self, team, deepest=3):
        self.team = team
        self.deepest = deepest   #最深度
        self.evaluate_class = Evaluate(self.team) #评估分数
        self.best_move = None  #最佳步
        self.last_move = []   #记前面几步，以防陷入循环

#获取下一步最佳移动。它调用了alpha_beta方法进行Alpha-Beta搜索，找到最佳的移动。
    def get_next_step(self, chessboard: ChessBoard):
        self.alpha_beta(chessboard, 0, -float('inf'), float('inf'), True)

        if self.best_move:#如果存在最佳移动，将最佳移动解包为当前行、列和下一个行、列。
            cur_row, cur_col, nxt_row, nxt_col = self.best_move

            # 去重
            self.last_move.append((nxt_row, nxt_col, cur_row, cur_col))
            if len(self.last_move) > 3:
                self.last_move.pop(0)

            # 加深
            if len(chessboard.get_chess()) <= 18:
                self.deepest = 4
            if len(chessboard.get_chess()) <= 15:
                self.deepest = 5

            return cur_row, cur_col, nxt_row, nxt_col
        return None

    @staticmethod
    def make_move(chessboard, chess, new_row, new_col):
        # 记录旧位置和棋子
        old_row, old_col = chess.row, chess.col
```

```python
        taken_chess = chessboard.chessboard_map[new_row][new_col] #被吃掉的棋子
        # 执行移动
        chessboard.chessboard_map[old_row][old_col] = None #移除旧位置
        chessboard.chessboard_map[new_row][new_col] = chess #在新位置放置棋子
        chess.update_position(new_row, new_col)  #更新棋子的位置信息
        return old_row, old_col, taken_chess

    @staticmethod
    def undo_move(chessboard, chess, old_row, old_col, taken_chess):
        # 撤销移动
        chessboard.chessboard_map[chess.row][chess.col] = taken_chess
        chessboard.chessboard_map[old_row][old_col] = chess
        chess.update_position(old_row, old_col)

其中包含Minimax算法和剪枝，在每一层递归中，根据当前轮到的玩家是最大化玩家还是最小化玩家，选择最大化或最小化评估值。
    def alpha_beta(self, chessboard, depth, alpha, beta, max_player):
        if depth == self.deepest:  #限制搜索深度
            return self.evaluate_class.evaluate(chessboard)

        if max_player: #最大化玩家
            max_eval = -float('inf') #初始化最大评估值为负无穷
            for chess in chessboard.get_chess():
                if chess.team == self.team: #检查当前棋子是否属于当前玩家的队伍。
                    for nxt_row, nxt_col in chessboard.get_put_down_position(chess): #遍历当前棋子可以移动到的下一个可能位置。

                        if (chess.row, chess.col, nxt_row, nxt_col) in self.last_move:
                            continue #检查这个移动是否已经在上一次移动中使用过，如果是则跳过。

                        old_row, old_col, taken_chess = self.make_move(chessboard, chess, nxt_row, nxt_col)
                        eval = self.alpha_beta(chessboard, depth + 1, alpha, beta, False) #递归调用alpha_beta函数，以当前棋盘状态的变化为基础，继续搜索下一层
                        self.undo_move(chessboard, chess, old_row, old_col, taken_chess) #恢复之前的移动，以便进一步尝试其他移动。
```

```
                    if eval > max_eval:
                        max_eval = eval
                        if depth == 0:
                            self.best_move = (old_row, old_col, nxt_row, nxt_col)
                    alpha = max(alpha, eval)
                    if beta <= alpha: #如果beta值小于或等于alpha值，表示当前节点不再具有最佳解，可以剪枝并退出循环。
                        break
            return max_eval
        else:
            min_eval = float('inf')
            for chess in chessboard.get_chess():
                if chess.team != self.team:
                    for nxt_row, nxt_col in chessboard.get_put_down_position(chess):
                        if (chess.row, chess.col, nxt_row, nxt_col) in self.last_move:
                            continue
                        old_row, old_col, taken_chess = self.make_move(chessboard, chess, nxt_row, nxt_col)
                        eval = self.alpha_beta(chessboard, depth + 1, alpha, beta, True)
                        self.undo_move(chessboard, chess, old_row, old_col, taken_chess)
                        min_eval = min(min_eval, eval)
                        beta = min(beta, eval)
                        if beta <= alpha:
                            break
            return min_eval
```

main.py:

因为之前的主函数不能实现我们的 AI 与程序之间的对战，所以我们重新写主函数，同时写一个 ai_move 函数

```
def ai_move(ai, game, chessboard, screen):
    cur_row, cur_col, nxt_row, nxt_col = ai.get_next_step(chessboard)
    ClickBox(screen, cur_row, cur_col)
    chessboard.move_chess(nxt_row, nxt_col)
    ClickBox.clean()
    # 产生将军局面
    if chessboard.judge_attack_general(game.get_player()):
        print('--- 黑方被将军 ---\n') if game.get_player() == 'r' else print('--- 红方被将军 ---\n')
        if chessboard.judge_win(game.get_player()):
            print('--- 红方获胜 ---\n') if game.get_player() == 'r' else print('--- 黑方获胜 ---\n')
            game.set_win(game.get_player())
            return
        else:
            game.set_attack(True)
    # 产生必胜局面
    else:
        if chessboard.judge_win(game.get_player()):
            print('--- 红方获胜 ---\n') if game.get_player() == 'r' else print('--- 黑方获胜 ---\n')
            game.set_win(game.get_player())
            return
        game.set_attack(False)
    # 产生和棋局面
    if chessboard.judge_draw():
        print('--- 和棋 ---\n')
        game.set_draw()

    game.back_button.add_history(chessboard.get_chessboard_str_map())
    game.exchange()
    return
```

```python
def main():
    # 初始化pygame
    pygame.init()
    # 创建用来显示画面的对象（理解为相框）
    screen = pygame.display.set_mode((750, 667))
    # 游戏背景图片
    background_img = pygame.image.load("images/bg.jpg")
    # 游戏棋盘
    # chessboard_img = pygame.image.load("images/bg.png")
    # 创建棋盘对象
    chessboard = ChessBoard(screen)
    # 创建计时器
    clock = pygame.time.Clock()
    # 创建游戏对象（像当前走棋方、游戏是否结束等都封装到这个对象中）
    game = Game(screen, chessboard)
    game.back_button.add_history(chessboard.get_chessboard_str_map())

    # 是否调换先后手（False MyAI红方先手    True chessAI红方先手）
    reverse = False

    if reverse:
        player1 = ChessAI(game.red)
        player2 = MyAI(game.black)
        print(f'红色方: ChessAI\n'
              f'黑色方: MyAI\n')
    else:
        player1 = MyAI(game.red)
        player2 = ChessAI(game.black)
        print(f'红色方: MyAI\n'
              f'黑色方: ChessAI\n')

    counter = 0

    while not game.show_win and not game.show_draw:
```

```python
    while not game.show_win and not game.show_draw:
        currentAI = player1 if counter % 2 == 0 else player2
        counter += 1

        ai_move(currentAI, game, chessboard, screen)

        #update_display(game.screen, background_img, game.chessboard, game)


        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
        # 显示游戏背景
        screen.blit(background_img, (0, 0))
        screen.blit(background_img, (0, 270))
        screen.blit(background_img, (0, 540))
```

```
107         chessboard.show_chessboard_and_chess()
108
109         # 标记点击的棋子
110         ClickBox.show()
111
112         # 显示可以落子的位置图片
113         Dot.show_all()
114
115         # 显示游戏相关信息
116         game.show()
117
118         # 显示screen这个相框的内容（此时在这个相框中的内容像照片、文字等会显示出来）
119         pygame.display.update()
120
121         # FPS（每秒钟显示画面的次数）
122         clock.tick(120)  # 通过一定的延时，实现1秒钟能够循环60次
123
124 if __name__ == '__main__':
125     main()
126
```

Game.py:

这里的文件做一些小改动。

```
    self.show_draw_count = 0
    self.show_draw_time = 300
    self.chessboard = chessboard
    self.red = 'r'
    self.black = 'b'
    self.AI_mode = True
    self.back_button = BackChess(screen)
```

# 三、 实验结果及分析

## 1. 实验结果展示示例（可图可表可文字，尽量可视化）

**AI 红方先手：**

```
旧位置:  6 1 新位置:  8 2 5步双方没有互吃棋子
旧位置:  9 5 新位置:  9 3 0步双方没有互吃棋子
--- 红方被将军 ---

--- 黑方获胜 ---
```

**我方先手：**

```
旧位置:  8 3 新位置:  0 3 3步双方没有互吃棋子
旧位置:  0 1 新位置:  0 3 0步双方没有互吃棋子
--- 黑方被将军 ---

--- 红方获胜 ---
```

**2. 评测指标展示及分析（机器学习实验必须有此项，其它可分析运行时间等）**

根据实验要求，我们需要分别使用 MyAI 先手（执红棋）和后手（执黑棋）的对局结果。可以看到，我们的 AI 运行时间要比对方 AI 运行时间要短很多，这得益于α-β剪枝算法。但同时也有个问题，在对面 AI 运行时退出此程序非常卡，且刚开始时我的程序会黑屏，再次亮起时已经走了第一步。咨询同学后得知是因为我方 AI 太快，对方 AI 应加密过，pyarmor 解密较慢导致。具体代码解析已将注释标注在代码中，下面是一些补充分析：

搜索深度控制：通过 deepest 参数控制搜索的深度，可以根据棋盘状态的复杂性来调整搜索深度。根据当前棋盘上棋子的数量，动态调整搜索深度，使得在关键时刻可以更深入地搜索，提高 AI 的水平。

移动选择策略：在最大化和最小化玩家之间切换，通过递归搜索所有可能的移动，以找到最优的走法。当评估值更高时，更新当前最佳移动步骤，以便在搜索结束时返回最佳的移动。

移动历史记录：通过 last_move 列表记录前几步的移动，防止陷入循环，避免重复选择相同的移动序列。

# 四、 参考资料

最大最小法及α-β 剪枝算法图解——CSDN
博弈树与α-β剪枝——CSDN
博弈树 α-β剪枝——CSDN