中山大学计算机学院

人工智能

本科生实验报告

（2022 学年春季学期）

课程名称：Artificial Intelligence

教学班级	计科 2 班	专业（方向）	计算机科学与技术
学号	22336126	姓名	李漾

一、 实验题目

博弈树搜索-中国象棋

二、 实验内容

1. 算法原理

Alpha-Beta 剪枝算法在搜索树的深度优先遍历中，通过维护两个值：alpha 和 beta，来剪除不必要的搜索路径。

Alpha 值表示 Max 层节点所能保证的最低分数，初始化为负无穷。

Beta 值表示 Min 层节点所能保证的最高分数，初始化为正无穷。

当搜索到达叶子节点时，返回节点的评估值。

在 Max 层节点，遍历其子节点，如果发现某个子节点的值大于等于 Beta 值，则剪枝，不再考虑该子节点的其他分支。

在 Min 层节点，遍历其子节点，如果发现某个子节点的值小于等于 Alpha 值，则剪枝，不再考虑该子节点的其他分支。

在 Max 层节点，更新 Alpha 值为所有子节点中的最大值。

在 Min 层节点，更新 Beta 值为所有子节点中的最小值。

当搜索完所有子节点或者发生剪枝时，返回当前节点的 Alpha 或 Beta 值。

2. 伪代码

```
function alpha_beta_search(node, depth, alpha, beta, maximizingPlayer):
    if depth == 0 or node is a terminal node:
        return the heuristic value of node

    if maximizingPlayer:
        value = -infinity
        for each child of node:
            value = max(value, alpha_beta_search(child, depth - 1, alpha, beta, FALSE))
```

```
                alpha = max(alpha, value)
                if beta <= alpha:
                    break #  β -cutoff
            return value
        else:
            value = +infinity
            for each child of node:
                value = min(value, alpha_beta_search(child, depth - 1, alpha, beta, TRUE))
                beta = min(beta, value)
                if beta <= alpha:
                    break #  α -cutoff
            return value

    # 初始调用
    alpha_beta_search(root, depth, -infinity, +infinity, TRUE)
```

3. 关键代码展示（带注释）

该项目的关键代码应该是 MyAI 里的 α - β 剪枝

```
def a_b_cut(self, depth, a, b, chessboard: ChessBoard):
        if depth>=self.max_depth:
            return self.evaluate_class.evaluate(chessboard) #到达深度限制则返回评估值
        else:
            chess_list=chessboard.get_chess()   #获取所有棋子对象
            for chess in chess_list:
                #该层为 max
                if depth%2==1 and chess.team ==self.team:
                    next=chessboard.get_put_down_position(chess)   #获取当前棋子可以走的列表
                    for new_x,new_y in next:
                        old_x=chess.row
                        old_y=chess.col
                        #保存下一步位置上的棋
                        origin_chess=chessboard.chessboard_map[new_x][new_y]
                        #走到下一步
                        chessboard.chessboard_map[new_x][new_y]=chessboard.chessboard_map[old_x]
[old_y]
                        #更新图片
                        chessboard.chessboard_map[new_x][new_y].update_position(new_x,new_y)
                        #原来的位置置为空
                        chessboard.chessboard_map[old_x][old_y]=None
                        #深度优先搜索
                        temp=self.a_b_cut(depth+1,a,b,chessboard)
                        #复原当前棋局
```

```
                    chessboard.chessboard_map[old_x][old_y]=chessboard.chessboard_map[new_x]
[new_y]
                    chessboard.chessboard_map[old_x][old_y].update_position(old_x,old_y)
                    chessboard.chessboard_map[new_x][new_y]=origin_chess

                        #该点的权值大于当前 a 值，则意味着有更好的走法
                        #或者之前没有记录过最佳移动步骤，防止在深度搜索的过程中多次记录相同的最佳步
骤
                        #如果是第一层,就记录下最好的走法
                    if(temp>a or not self.past) and depth==1:
                        self.past=[chess.row,chess.col]
                        self.now=[new_x,new_y]

                    a=max(a,temp)
                    #max 层只更新节点的 a 值
                    if b<=a:#剪枝
                        return a
                #该层为 min
                elif depth%2==0 and chess.team!=self.team:
                    next=chessboard.get_put_down_position(chess)   #获取当前棋子可以走的列表
                    for new_x,new_y in next:
                        old_x=chess.row
                        old_y=chess.col
                        #保存下一步位置的棋
                        origin_chess=chessboard.chessboard_map[new_x][new_y]
                        #走到下一步
                        chessboard.chessboard_map[new_x][new_y]=chessboard.chessboard_map[old_x]
[old_y]
                        #更新图片
                        chessboard.chessboard_map[new_x][new_y].update_position(new_x,new_y)
                        chessboard.chessboard_map[old_x][old_y]=None
                        #深度优先搜索
                        temp=self.a_b_cut(depth+1,a,b,chessboard)
                        #复原当前棋局
                        chessboard.chessboard_map[old_x][old_y]=chessboard.chessboard_map[new_x]
[new_y]
                        chessboard.chessboard_map[old_x][old_y].update_position(old_x,old_y)
                        chessboard.chessboard_map[new_x][new_y]=origin_chess

                        b=min(b,temp)

                        if b<=a:#剪枝
                            return b
```

```
            if depth%2==1:
                return a
            else:
                return b
            #alpha-beta 剪枝算法其实就是深度优先搜索过程中维护了两个值，不断更新并且进行判断，最终找到最优路径。因此在编写程序时可以进行递归的操作
            #而在本项目中需要将棋子移动到新的位置，更新棋子的位置信息，再将原来位置上的棋子暂时保存起来。
            #递归调用 self.a_b_cut() 方法，继续搜索下一层。恢复原来的棋局状态，以便进行下一次搜索。
            #在理论实践和实验的实践中可以发现实际就是访问到叶子节点后不断向上更新 alpha&beta 值，最终使得更快速找到一条 minmax 路径
```

4. 创新点&优化（如果有）

有一些疑问点，尝试修改后好像仍没有解决：

（1）可能是电脑配置的问题，写成两个 AI 后 pygame 的页面无法加载出第一步。通常第一步都会是在黑屏的情况。以及如果中间进行截图的时候会出现窗口卡死的情况。

（2）Game 里的规则是说判断三局重复走的话就会判平手，但是我的算法写出后可能总会进入一些僵局比如双方重复走的情况，但是最后都是判断黑棋胜出，好像是判断和棋的情况最后写成了黑棋获胜。

（3）自己编写的 AI 缺少一些随机元素，每次都是贯彻进行一模一样的招数，而没办法做到随机应变，最终就是落到和黑棋陷入僵局的情况。

（4）在尝试进行参数修改的时候，我认为应该将红车的位置权值改大一些，但是进行尝试的时候发现修改参数了之后有时候当判断对方获胜的时候会出现如下报错然后退出程序。尝试在判断成功的时候加入 time.sleep()函数，但是好像仍然会出现以下报错。最后就没有进行参数的修改

```
Traceback (most recent call last):
  File "d:/人工智能/象棋code/AIchess_win/main.py", line 149, in <module>
    main()
  File "d:/人工智能/象棋code/AIchess_win/main.py", line 83, in main
    chessboard.move_chess(nxt_row, nxt_col)
  File "d:\人工智能\象棋code\AIchess_win\ChessBoard.py", line 363, in move_chess
    self.chessboard_map[new_row][new_col].update_position(new_row, new_col)
AttributeError: 'NoneType' object has no attribute 'update_position'
```

（然后因为本来也不会下象棋，还不是太理解象棋的章法，最终就没有进行参数的修改，因此我的红棋每次都是惨败的情况）

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）

红棋先手的情况

黑棋先手的情况：

以及上面提到的僵局，我进行了视频的录制：
打包放在压缩包中

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）

利用 alpha-beta 剪枝很大程度上降低了对最优路径的运算时间，但是我设计的 5 层深度的树运行下来仍然时间较长，打一局下来就花费大约 15 分钟。但是如果改为 4 层就会很快输给对手，6 层的话就会卡非常久了。所以很遗憾最终没办法调试出胜利的局面。