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	22336326	姓名	朱禹溪

一、 实验题目

3.博弈树搜索--中国象棋

文件介绍

- `main.py`是main函数的程序，直接运行这个文件可以实现人机博弈对抗。
- 其他`.py`文件都是程序运行所需要的类，包括`ChessBoard`、`Game`等。
- `images`文件夹是可视化界面所需的图片。
- 对手AI在`ChessAI.py`中实现，对手AI类已被`pyarmor`加密，需要安装`pyarmor`库才能运行此py文件。另外，我们提供了`linux`、`windows`、`mac`三个版本的加密文件，根据自己电脑的系统选择对应版本的程序代码。
- `MyAI.py`提供了`ChessAI.py`中部分代码逻辑，其中包括了`Evaluate`、`ChessMap`、`ChessAI`三个类。`Evaluate`类提供了当前象棋局面的奖励值，即每个棋子在棋盘上发任意位置都会有一个奖励值，所有棋子的奖励值之和为整个棋面的奖励值。提供的奖励值仅仅作为参考，如果想要以更大的概率打败对手AI，建议修改奖励值。`ChessAI`是实现算法的核心类，须在此类中实现搜索算法。
- 最终评估方法：与对手AI共博弈2次，其中先手、后手各评估一次（`在main.py中未实现算法的红黑机指定代码，需自行实现`）。积分规则：胜一局记3分，平一局记1分，负一句记0分。

代码运行

建议使用`python3.7或python3.6`运行代码

需要安装`pygame`、`numpy`、`pyarmor`库：

```
pip install pygame numpy pyarmor
```

开始程序的命令：

二、 实验内容

1. 算法原理

本次实验中主要使用了 Alpha-Beta 剪枝算法来于优化博弈树搜索。

递归搜索：从当前棋局状态开始，算法会递归地探索可能的走法，考虑双方玩家的行动。在每一层，算法交替考虑最大化和最小化玩家的行动，并评估当前局面的价值。

Alpha 值和 Beta 值：Alpha 和 Beta 值是算法用于剪枝的关键。Alpha 值代表在 Max 层面上带来的最大值，Beta 值代表在 Min 层面上带来的最小值。通过比较当前的节点值和 Alpha/Beta 值，算法可以准确评估哪些分支不值得深入搜索。

剪枝：在搜索树的过程中，如果算法发现某些分支不够优秀，会剪掉这些分支，这样就不必继续搜索这些分支，从而提高搜索效率。

搜索顺序：在每一层，算法会根据当前是最大化还是最小化玩家，有选择地深入搜索。最大化玩家希望获得更高的分数，因此会尝试获得更大的 Alpha 值；最小化玩家希望获得更低的分数，因此会尝试获得更小的 Beta 值。

深度限制：为了避免无限的搜索，通常会设置一个搜索深度限制。当达到深度限制后，算法将评估当前局面的得分，然后返回。

2. 关键代码展示（可选）

```python
def get_next_step(self, chessboard: ChessBoard):
    old_pos = self.old_pos
    new_pos = self.new_pos
    self.alpha_beta(1,-999999,999999,chessboard)
    if old_pos==self.old_pos and new_pos==self.new_pos:
        self.max_depth=2
        self.alpha_beta(1,-999999,999999,chessboard)
    return self.old_pos+self.new_pos
```

```python
def alpha_beta(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()
        for cs in chess_list:
            #max层
            if depth%2==1 and cs.team ==self.team:
                next=chessboard.get_put_down_position(cs)   #获取当前棋子可以走的列表
                for new_x,new_y in next:
                    last_x=cs.row
                    last_y=cs.col
                    #保存下一步位置的棋
                    origin_chess=chessboard.chessboard_map[new_x][new_y]
                    #移动棋子
                    chessboard.chessboard_map[new_x][new_y]=chessboard.chessboard_map[last_x][last_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[last_x][last_y]=None
                    temp=self.alpha_beta(depth+1,a,b,chessboard)
                    #复原棋局
                    chessboard.chessboard_map[last_x][last_y]=chessboard.chessboard_map[new_x][new_y]
                    chessboard.chessboard_map[last_x][last_y].update_position(last_x,last_y)
                    chessboard.chessboard_map[new_x][new_y]=origin_chess
                    #self.step.append(new_x,new_y)
                    # if len(self.step)>1:
                    #     if new_x==self.step[-2][0] and new_y==self.step[-2][1]:
                    #         temp-=300
                    #         repeat=1
                    #     else:
                    #         repeat=0
                    #     if repeat==1 and new_x==self.step[-2][0]and new_y==self.step[-2][1]:
```

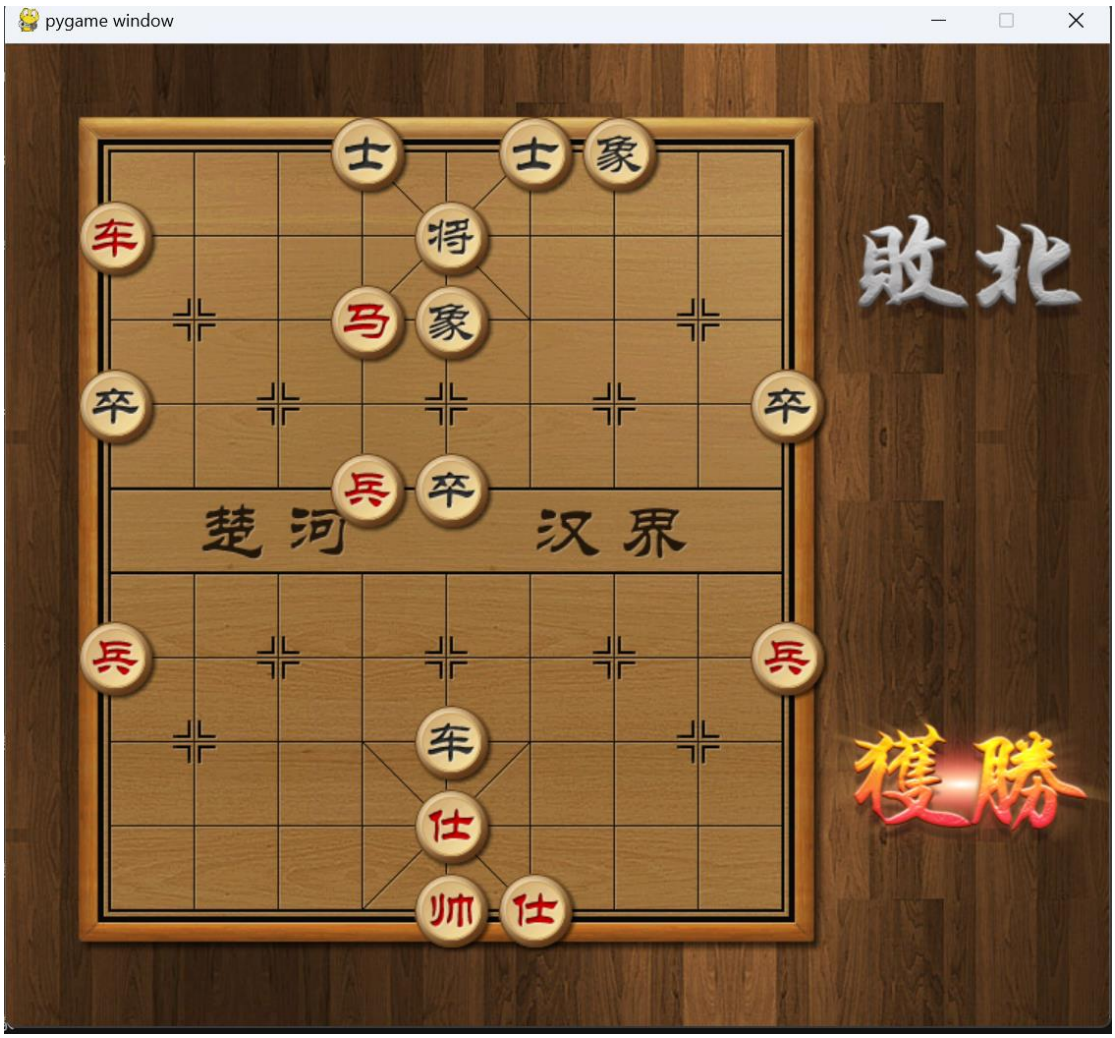
```
                    #     if repeat==1 and new_x==self.step[-2][0]and new_y==self.step[-2][1]:
                    #         temp-=60000
                    #     else :
                    #         repeat=0
                        #1、得分大于当前值或者还没赋值，如果是第一层,则可以设置要移动的坐标
                    if(temp>a or not self.old_pos) and depth==1:
                        self.old_pos=[cs.row,cs.col]
                        self.new_pos=[new_x,new_y]

                    a=max(a,temp)
                    if b<=a:#剪枝
                        return a
            #min
            elif depth%2==0 and cs.team!=self.team:
                next=chessboard.get_put_down_position(cs)
                for new_x,new_y in next:
                    last_x=cs.row
                    last_y=cs.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[last_x][last_y]
                    #更新图片
                    chessboard.chessboard_map[new_x][new_y].update_position(new_x,new_y)
                    chessboard.chessboard_map[last_x][last_y]=None
                    #深度优先搜索
                    temp=self.alpha_beta(depth+1,a,b,chessboard)
                    #复原棋局
                    chessboard.chessboard_map[last_x][last_y]=chessboard.chessboard_map[new_x][new_y]
                    chessboard.chessboard_map[last_x][last_y].update_position(last_x,last_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
```

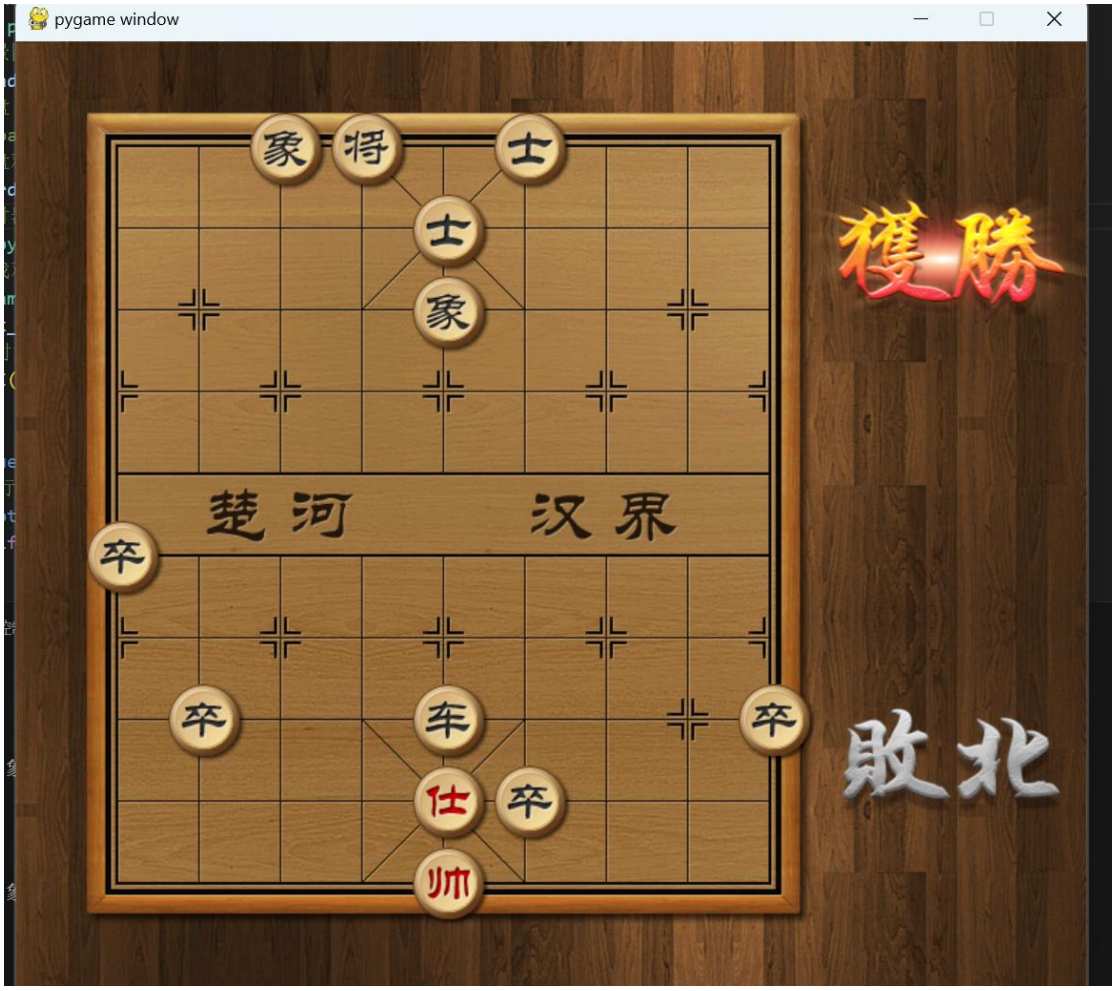
三、 实验结果及分析

1. 实验结果展示示例（可图可表可文字，尽量可视化）

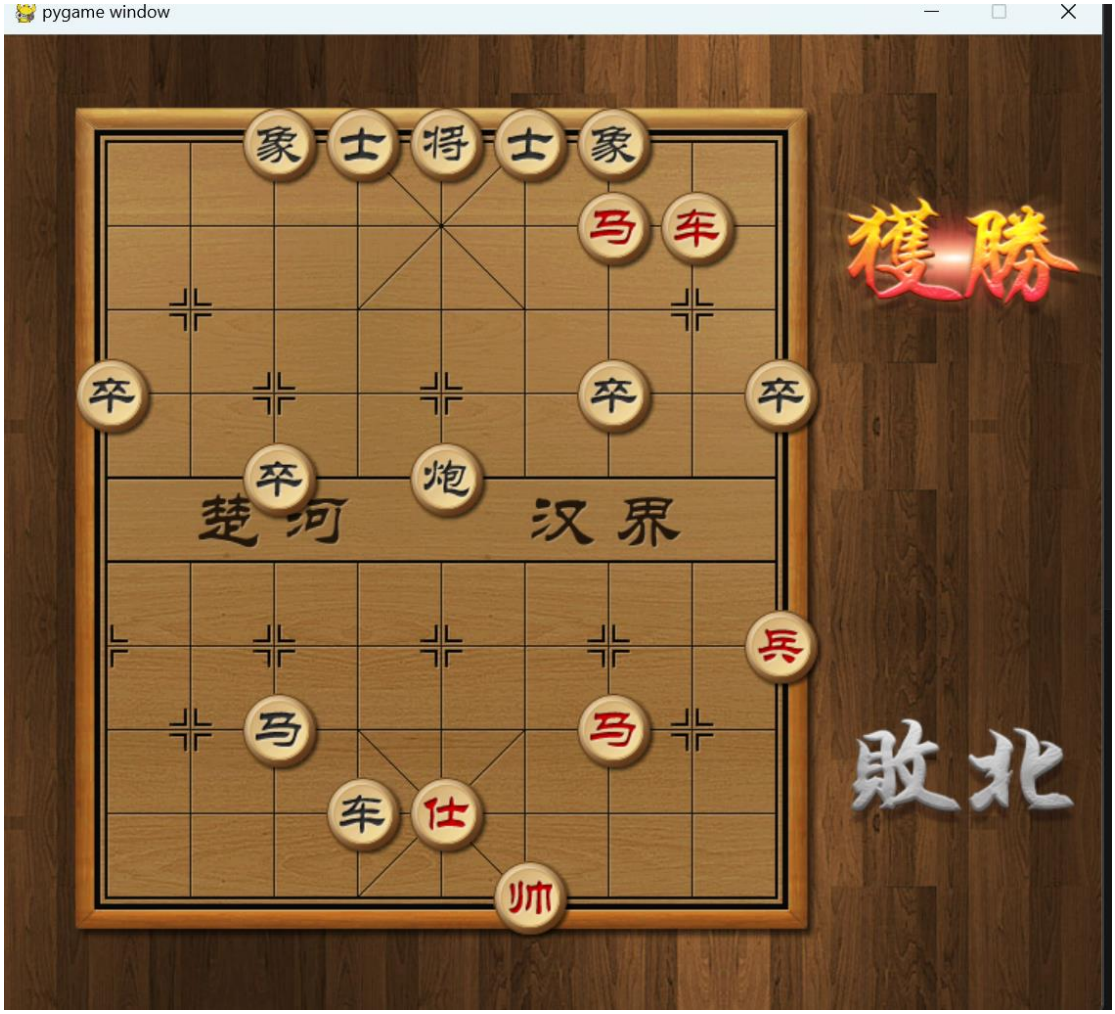
人机对战（人执红）：



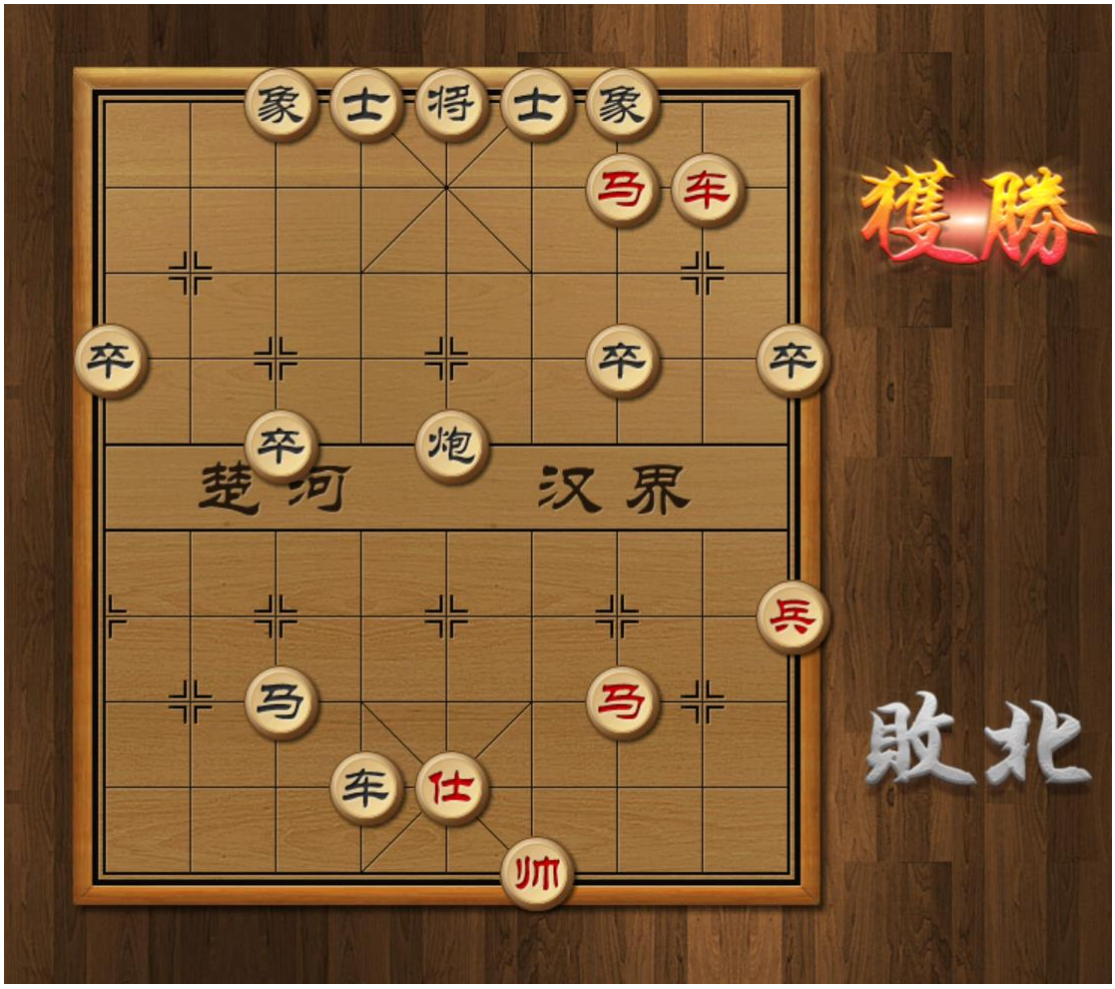
人机对战（人执黑）



Ai 互搏（人执红）：



Ai 互搏（人执黑）：



3. 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）

对于 Alpha-Beta 剪枝算法，其时间复杂度取决于搜索树的大小和有效的剪枝操作。

搜索树分支因子：在这段代码中，每一步棋的走法都会产生一个分支，因此搜索树的分支因子取决于每一步棋的可选移动数量。

最大搜索深度：在算法中，通过 self.max_depth 来控制搜索的最大深度。当达到最大深度时，会返回当前局面的评估值。

剪枝：Alpha-Beta 剪枝算法的关键在于剪掉那些不会对最终结果产生影响的分支，因此有效的剪枝操作可以显著减少搜索的节点数量。

接下来我们大致计算一下这段代码的复杂度：

对于每一个棋局状态，它的搜索树的分支因子取决于每个棋子可以走的位置数，即平均分支因子为 (b)。最大搜索深度为 self.max_depth。

在最坏情况下，需要遍历完整的搜索树，因此复杂度为 (O(b^d))，其中 (d) 为最大搜索深度。

整体而言前期每走一步需要时间大概在 2.5s 左右，后期棋子数量减少搜索 难度降低速度会大幅提高，整体一盘棋局会在 3 分钟以内。程序还存在一个 bug，就是当遇到重复局面的时候无法及时脱离（算法中注释掉的就是原本想用于脱离的），导致 ai 对战最后会长将或者重复走闲棋，并且可能是由于没有将可走的棋子的列表进行随机的排列，互博时走的较为呆板，每次都是一样的棋局，后续还需要进一步的修改。

四、 参考资料

怎样做一道阿尔法贝塔剪枝的题(图解)-CSDN 博客

4-搜索算法.pptx（课上 ppt）