**中山大学计算机学院**

**人工智能**

**本科生实验报告**

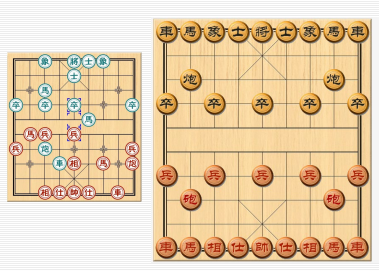
**（2022学年春季学期）**

课程名称：Artificial Intelligence

|  |  |  |  |
| --- | --- | --- | --- |
| 教学班级 | **系统结构班** | 专业（方向） | **计算机科学与技术** |
| 学号 | **21307358** | 姓名 | **曾慧蕾** |

# 实验题目

编写一个中国象棋博弈程序，要求用alpha-beta 剪枝算法，可以实现人机对弈。棋局评估方法可以参考已有文献，要求具有下棋界面，界面编程也可以参考网上程序，但正式实验报告要引用参考过的文献和程序。



界面示例

# 实验内容

1. 算法原理

象棋搜索下一步走法时，主要运用了Minimax搜索与Alpha-beta剪枝的原理。下面分别进行原理解释。

**Minimax算法：**

Minimax搜索是指在零和博弈中，玩家均会在可选的选项中选择将其N步后优势最大化或者令对手优势最小化的选择。该算法在博弈过程中会将玩家双方分为MAX和MIN，双方都要使得自身的收益最大化。收益值如何取决于评估函数的设计。一般的评估函数是以其中一方的收益来设计函数。因此对这一方来说要使得自身评估函数值最大，对于另一方来说要使得评估函数值最小。

在博弈的过程中会得到一颗博弈树，若决策树某一层均为己方决策依据状态（即接下来是己方进行动作），则己方必定会选择使得己方收益最大化的路径，将该层称为MAX层。若决策树某一层均为对手决策依据状态（即接下来是对手进行动作），则对手必定会选择使得己方收益最小化的路径，将该层成为MIN层。从最底层开始往上推，就能得到最终的决策。

**Alpha-beta剪枝的深度优先Minimax算法：**

Alpha-beta剪枝是对Minimax算法的一种优化。在Minimax搜索进行的过程中，需要检查的游戏状态数目会呈指数级增长，Alpha-beta剪枝即是在Minimax产生的决策树的基础上剪掉不可能影响决策的分支，尽可能地消除掉部分的搜索树，从而提高算法效率。

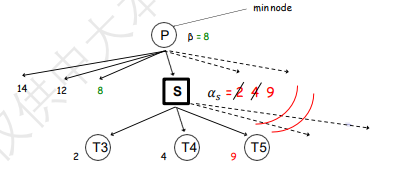
Alpha-beta剪枝的规则如下：

**对MAX层结点s：**

设a是s被遍历过的子节点的最高值，b是当前遍历过的s的兄弟节点中的最低值。

如果a≥b，可以停止遍历s的子节点了

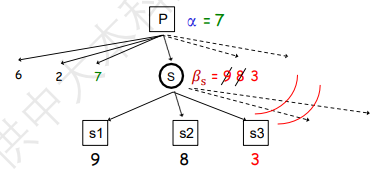
前面的Min节点不会来到通过Max节点S的父母P来到Max节点S这个状态的，因为Min一定会选择Max节点S的值更小的兄弟节点。



**对MIN层结点s：**

设a是s被遍历过的兄弟节点的最高值，b是当前遍历过的s的子节点中的最低值。

如果b≤a，那么可以停止扩展s的子节点

Max节点一定不会选择Min节点s，因为它会优先选择Min节点S值更高的兄弟节点，放大一点来说, 在Min节点 S, 如果β 变得 ≤ 某个Max祖先节点的α值, 那么S节点的扩展就可以停止了

1. 伪代码

输入：决策树深度depth，alpha值a，beta值b，当前地图map

输出：下一个决策动作

alpha\_beta(self,depth,a,b,state,tree):

    if depth<self.max\_depth:#若未到最大深度

        next\_step=get\_next\_step(state)

        if state.mm=='MAX':

            for ns in next\_step:

                child=alpha\_beta(depth+1,a,b,state) #得到子节点的a或b值

                a=max(a,child)

                if depth==1 and child>a:

                    a=child

                    state.add\_hisitory(tree)#对于未初始化的 保存当前行动

                if a>=b:

                    return a

        else:

            for ns in next\_step:

                child=alpha\_beta(depth+1,a,b,state) #得到子节点的a或b值

                b=min(b,child)

                # if depth==1 and child>a:

                #     a=child

                #     state.add\_hisitory(tree)#对于未初始化的 保存当前行动

                if a>=b:

                    return b

if state.mm=='MAX': return a

else: return b

    else:

        return evaluate(state)

本算法值得注意的一点是如何在一次递归里面考虑到第三层的剪枝。对于这个问题，根据每一个输入的状态（第一层）得到他们所有下一步的行动（第二层），再对下一步的行动进行alpha-beta剪枝算法得到子节点的a/b值即可。

1. 关键代码展示（带注释）

本次实验主要完成的是ChessAI的补充，补充如下：

get-next-step函数：主要作用是调用alpha-beta算法并得到最优决策。

    def get\_next\_step(self, chessboard: ChessBoard):

        '''

        :该函数应当返回四个值

            1 要操作棋子的横坐标

            2 要操作棋子的纵坐标

            3 落子的横坐标

            4 落子的纵坐标

        '''

        self.alpha\_beta(1,-99999999,99999999,chessboard) #a往大取 b往小取 故初始值a为负无穷b为正无穷

        # if chessboard.chessboard\_map[self.new\_pos[0]][self.new\_pos[1]]==None:

        #     chess\_list=chessboard.get\_chess()

        #     chess=chess\_list[0]

        #     return chess.row,chess.col,chess.row,chess.col

        return self.old\_pos[0],self.old\_pos[1],self.new\_pos[0],self.new\_pos[1] #old\_pos:要操作的 new\_pos：落子

alpha\_beta函数：做出决策，并返回

def alpha\_beta(self, depth, a, b, chessboard: ChessBoard):

    if depth<self.max\_depth:

        chess\_list=chessboard.get\_chess() #所有能走的棋子 chesslist的元素去看chess类

        for chess in chess\_list:

            old\_x=chess.row

            old\_y=chess.col

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

            #我方是红方 是max层 AI是黑方 是min层

            #max层 我方

            if depth%2==1 and self.team==chess.team:

                for new\_x,new\_y in forward:

                    child=self.update(chessboard,old\_x,old\_y,new\_x,new\_y,depth,a,b)

                    if (child>a or not self.old\_pos)and depth==1: #如果是第一次落子 还未设置old pos：

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

                        self.new\_pos=[new\_x,new\_y]

                    a=max(a,child)

                    if a>=b:

                        return a #alpha剪枝时a改变 b固定

                #min层 AI方

            elif depth%2==0 and self.team!=chess.team:

                for new\_x,new\_y in forward:

                    child=self.update(chessboard,old\_x,old\_y,new\_x,new\_y,depth,a,b)

                    b=min(b,child)

                    if a>=b:

                        return b #beta剪枝时a固定 b改变

            #若循环完的时候都没有剪枝

        if depth%2==1:

            return a

        else:

            return b

    else:

        return self.evaluate\_class.evaluate(chessboard) #超过深度 在评估函数里找效益最好的返回

update函数：主要作用是更新在每次移动棋子时的图像变化

    def update(self,chessboard: ChessBoard,old\_x,old\_y,new\_x,new\_y,depth,a,b):

        temp=chessboard.chessboard\_map[new\_x][new\_y]

        # temp=chessboard

        chessboard.chessboard\_map[new\_x][new\_y]=chessboard.chessboard\_map[old\_x][old\_y]

        chessboard.chessboard\_map[old\_x][old\_y]=None

        chessboard.chessboard\_map[new\_x][new\_y].update\_position(new\_x,new\_y)

        child=self.alpha\_beta(depth+1,a,b,chessboard)

        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]=temp

        # chessboard=temp

        # chessboard.chessboard\_map[old\_x][old\_y].update\_position(old\_x,old\_y)

        return child

1. 创新点&优化（如果有）

本次实验的优化主要在于一些小功能的丰富和补充中。

我一共添加了如下功能：退出游戏、红方认输以及一个整活功能。三个功能的补充主要在main、Game、chessboard三个文件中进行。

整活功能是因为我打不过AI怒而将所有棋子换成车（虽然换了我也不会玩唉。。）

其他优化还有诸如获胜/失败等图片的补充。

Game文件：

class Quit(pygame.sprite.Sprite):

    def \_\_init\_\_(self, screen):

        self.screen = screen

        self.image = pygame.image.load("D:\\code\\python\\file\\lab5\\images\\btn\_exit.png")

        self.rect = self.image.get\_rect()

        self.rect.topleft = (200, 620)

    def show(self):

        self.screen.blit(self.image, self.rect)

    def clicked\_back(self, chessboard: ChessBoard, event):

        if event.type == pygame.MOUSEBUTTONUP and self.rect.collidepoint(event.pos):

            print("EXIT!")

            return True

class Defeat(pygame.sprite.Sprite):

    def \_\_init\_\_(self, screen):

        self.screen = screen

        self.image = pygame.image.load("D:\\code\\python\\file\\lab5\\images\\btn\_lose.png")

        self.rect = self.image.get\_rect()

        self.rect.topleft = (615, 180)

    def show(self):

        self.screen.blit(self.image, self.rect)

    def clicked\_back(self, chessboard: ChessBoard, event):

        if event.type == pygame.MOUSEBUTTONUP and self.rect.collidepoint(event.pos):

            print("RED ONE GIVE UP!")

            return True

class Zhenghuo(pygame.sprite.Sprite):

    def \_\_init\_\_(self, screen):

        self.screen = screen

        self.image = pygame.image.load("D:\\code\\python\\file\\lab5\\images\\zhenghuo.png")

        self.rect = self.image.get\_rect()

        self.rect.topleft = (600, 570)

        self.history\_map = list()

    def show(self):

        self.screen.blit(self.image, self.rect)

    def clicked\_back(self, chessboard: ChessBoard, event):

        if event.type == pygame.MOUSEBUTTONUP and self.rect.collidepoint(event.pos):

            print("here we go!")

            chessboard.create\_chess\_6()

            return True

Game类中：

class Game(object):

def \_\_init\_\_(self, screen, chessboard):

#-------------------

self.quit\_button=Quit(screen)

        self.defeat\_button=Defeat(screen)

        self.zhenghuo\_button=Zhenghuo(screen)

  def show(self):

        self.quit\_button.show()

        self.defeat\_button.show()

        self.zhenghuo\_button.show()

main文件：

if game.quit\_button.clicked\_back(chessboard,event):

    pygame.quit()

    sys.exit()  # 退出程序

if game.defeat\_button.clicked\_back(chessboard,event):

    # game.set\_win(game.get\_player())

    game.set\_win('b')

if game.zhenghuo\_button.clicked\_back(chessboard,event):

    screen.blit(boom\_img, (110, 110))

    break

Chessboard文件： 主要是一个棋盘再设置的函数的添加

def create\_chess\_6(self):

    self.chessboard\_map = [

        ["b\_c", "b\_m", "b\_x", "b\_s", "b\_j", "b\_s", "b\_x", "b\_m", "b\_c"],

        ["", "", "", "", "", "", "", "", ""],

        ["", "b\_p", "", "", "", "", "", "b\_p", ""],

        ["b\_z", "", "b\_z", "", "b\_z", "", "b\_z", "", "b\_z"],

        ["", "", "", "", "", "", "", "", ""],

        ["", "", "", "", "", "", "", "", ""],

        ["r\_c", "", "r\_c", "", "r\_c", "", "r\_c", "", "r\_c"],

        ["", "r\_c", "", "", "", "", "", "r\_c", ""],

        ["", "", "", "", "", "", "", "", ""],

        ["r\_c", "r\_c", "r\_c", "r\_c", "r\_j", "r\_c", "r\_c", "r\_c", "r\_c"],

    ]

    for row, line in enumerate(self.chessboard\_map):

        for col, chess\_name in enumerate(line):

            if chess\_name:

                # 将创建的棋子添加到属性map中

                self.chessboard\_map[row][col] = Chess(self.screen, chess\_name, row, col)

            else:

                self.chessboard\_map[row][col] = None

# 实验结果及分析

实验结果展示示例（可图可表可文字，尽量可视化）



具体效果请见附件中的视频。

# 参考资料

**超算习堂-课时14-第八讲实验课**

**超算系统-课时11-第六讲**