人工智能实验

博弈树搜索

第 9 次实验

姓名：陈家豪

班级：2018 级计科 1 班

学号：18308013

目录

1 算法原理

1.1 二人零和博弈问题

不同于智能体对环境有完全控制的搜索问题，在博弈问题中，存在利益不同甚至相违背的多个智能体。这些智能体都有自身的利益取向，都会根据自身的利益改变世界。我们可以假设只有两个玩家进行博弈，游戏的状态或决策可以映射为离散的值，而游戏的状态或可以采取的行动的种类是有限的。当这种博弈是一种完全的竞争，即游戏的一方赢了、则另一方输掉了同等的数量时，这种博弈就是二人零和博弈。

一般而言，二元零和博弈没有不确定的因素、任何层面的状态都是可观察的。一个二元零和博弈由以下要素组成：

(1) 两个玩家 A（Max）和 B（Min）；

(2) 游戏状态的有限集合 S；

(3) 一个初始状态 $I \in S$ 和若干个终止状态 $T \in S$；

(4) 后继函数；

(5) 效益函数；

A 和 B 交替行动，当到达某个终止状态 T 时游戏结束。A 希望最大化终止状态的收益，而 B 希望最小化终止状态的收益。

五子棋是一种典型的二元零和博弈。在本次实验中，我们需要实现 11×11 的五子棋的人机对战。

1.2 博弈树

二人零和博弈的过程相当于对博弈树进行搜索。博弈树的内部节点和外部节点表示问题的状态，而双方玩家的行动就是轮流扩展（寻找）下一个节点。对于每个节点，我们都可以使用评价函数获取其优劣得分。博弈树搜索对于双方玩家而言，就是要找到对自己最优的节点及其对应的值。

解决博弈树搜索的方法有很多种，其中最经典的就是 MiniMax 搜索。

1.3 MiniMax 搜索

MiniMax 搜索基于一个前提假设，就是对方总是能做出最优行动。因此，己方需要做出最小化对方获得收益的行动，从而最大化己方的收益。因此，我们需要构建一个完整的博弈树，其根节点表示起始状态，每个叶子节点表示一个终止状态，标记了对应的效益值。然后，我们根据每个节点会做出的最优策略进行反向传播，最终得到根节点的预期收益值。以图1为例，A 的 Max 节点总是希望最大化收益值，B 的 Min 节点总是希望最小化收益值。经过反向传播后，处于根节点的 A 的最优收益值为 1.

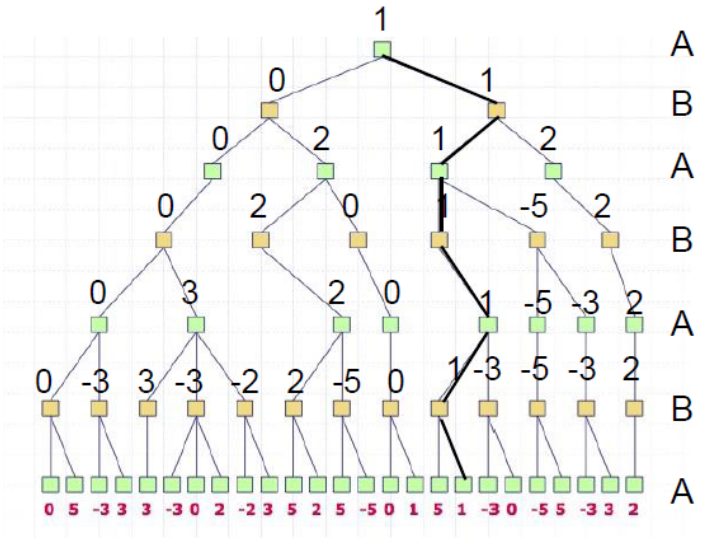

图 1: MiniMax 搜索与对应最优路径

由于整个博弈树的大小是指数增长的。因此在实现过程中，我们使用深度优先搜索算法来实现 MiniMax，从而避免存储指数级别大小的博弈树。

1.4 Alpha-beta 剪枝

虽然通过博弈树我们可以得到当前玩家的理想最优解，但随着搜索深度的增加，搜索的状态节点数目也会指数级增长。虽然可以通过限制搜索深度来减少计算过程，但过小的深度限制会导致预期收益的不准确（毕竟没有到达终止状态就停止了搜索）。我们还需要一种方式去剪掉不可能影响决策的分支、尽可能地消除部分搜索树，这就是 Alpha-beta 剪枝。

Alpha-beta 剪枝有两种类型，分别是对 Max 节点的剪枝和对 Min 节点的剪枝。对于 Max 节点 N_1 而言，设 α 是 N_1 下被遍历过的子节点中的最高值，β 是 N_1 被遍历过的兄弟节点的最低值。那么当 $\alpha > \beta$ 时，N_1 不需要遍历后续子节点。这是因为 Max 节点 N_1 的收益值至少为 α，而 N_1 的父节点是 Min 节点，其至少会选择值为 β 的节点、不再考虑 N_1。

同样，对于 Min 节点 N_2 而言，设 α 是 N_2 被遍历过的兄弟节点的最高值，β 是 N_2 下被遍历过的子节点的最低值。那么当 $\alpha > \beta$ 时，N_2 不需要遍历后续子节点。这是因为 Min 节点 N_1 的收益值至多为 β，而 N_2 的父节点是 Max 节点，其至少会选择值为 α 的节点、不再考虑 N_2。

我们可以将上述两种剪枝进行泛化。对于根节点而言，我们设置 $\alpha = -\infty, \beta = \infty$。在进行深度优先搜索时，假设父节点为 N_f，子节点为 N_s，那么 $\alpha_s = \alpha_f, \beta_s = \beta_f$。然后对 N_s 的子节点进行深度优先搜索：

(1) 当 N_s 为 Max 节点时，若在遍历其子节点时出现某个子节点的收益值大于 α_s，那么就用该收益值更新 α_s；若该收益值大于 β_s，那么停止遍历、N_s 的收益值就是该收益值（相当于剪枝）；若遍历完所有的子节点、没有出现子节点的收益值大于 β_s，那么 N_s 的收益值是其各个子节点收益值的最大值；

(2) 当 N_s 为 Min 节点时，若在遍历其子节点时出现某个子节点的收益值小于 β_s，那么就用该收益值更新 β_s；若该收益值小于 α_s，那么停止遍历、N_s 的收益值就是该收益值（相当于剪枝）；若遍历完所有的子节点、没有出现子节点的收益值小于 α_s，那么 N_s 的收益值是其各个子节点收益值的最小值；

以图2为例，我们在进行从左到右的深度优先搜索时使用了泛化的 Alpha-beta 剪枝。从本质而言，子节点“继承”父节点的 α 和 β 是将祖先节点的可选值一并纳入了考虑范围。

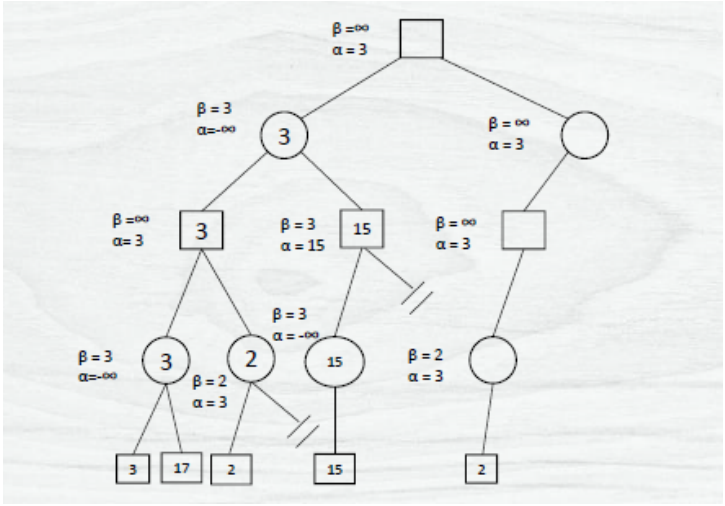

图 2: 泛化的 Alpha-beta 剪枝

1.5 评价函数

在博弈过程中，我们需要对棋局进行评价，得到当前状态的效益值。评价函数的设计利用到状态的特征值。我们根据经验或者机器学习的方式找到状态里的各个特征和相应权重，然后进行综合、输出评价值。

在本次实验中，我设计了一个如下的评价函数：对于一个 11×11 的棋盘，按行、列方向均有 $7\times11=77$ 个五元组，按对角线方向分别有 49 个五元组，因此总共有 252 个五元组。计算每个五元组的分数：

(1) 若该五元组既有黑棋又有白棋，0 分（因为对于双方来说这个五元组已经没有“用处”了）；

(2) 若该五元组既无黑棋也无白棋，0 分（因为对于双方而言机会均等）；

(3) 若该五元组只有 1 个黑棋（白棋），+10 分（−10 分）；

(4) 若该五元组只有 2 个黑棋（白棋），+30 分（−30 分）；

(5) 若该五元组只有 3 个黑棋（白棋），+120 分（−120 分）；

(6) 若该五元组只有 4 个黑棋（白棋），+450 分（−450 分）；

(7) 若该五元组有 5 个黑棋（白棋），+10000 分（−10000 分）；

将所有五元组的分数相加，即为评价函数的输出值。显然，对于黑棋方而言，其就是玩家 A，需要极大化效益值；对于白棋方而言，其就是玩家 B，需要极小化效益值；

2 伪代码/流程图

我们设 CurrentPlayer=1 为执黑棋方，CurrentPlayer=2 为执白棋方，黑棋先手。电脑和人类轮流下棋，下棋后经过一系列判定判断结果、进行换边。alpha-beta 剪枝的博弈树搜索在函数 search() 中实现，电脑通过调用该函数得到最优位置。

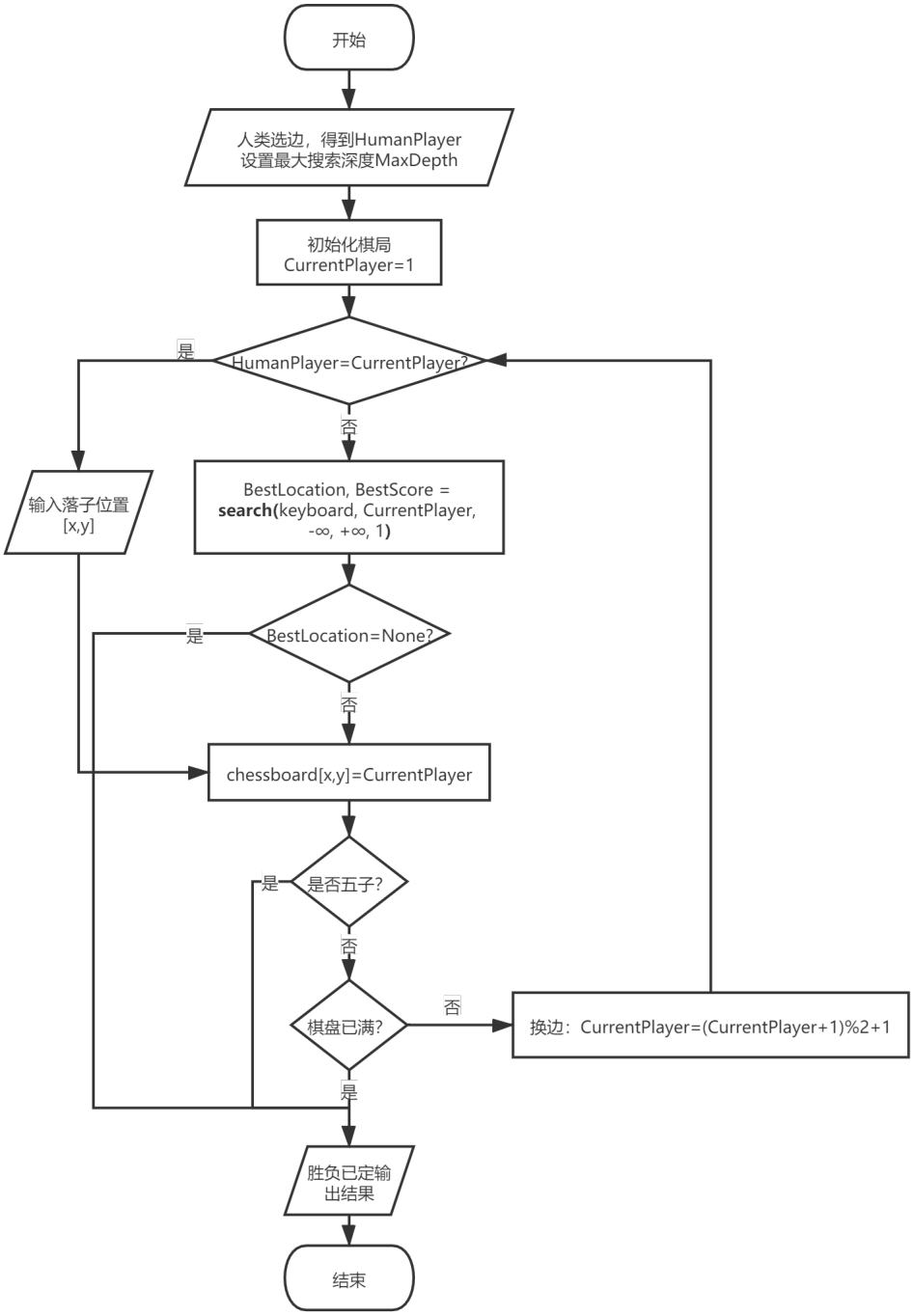

图 3: 五子棋游戏流程图

search() 函数的流程图4所示，我们需要获取当前节点的原始状态（棋盘布局）、下棋方、α、β 和当前深度，然后经过一系列判断、递归操作。若 $\alpha > \beta$，则进行剪枝处理。

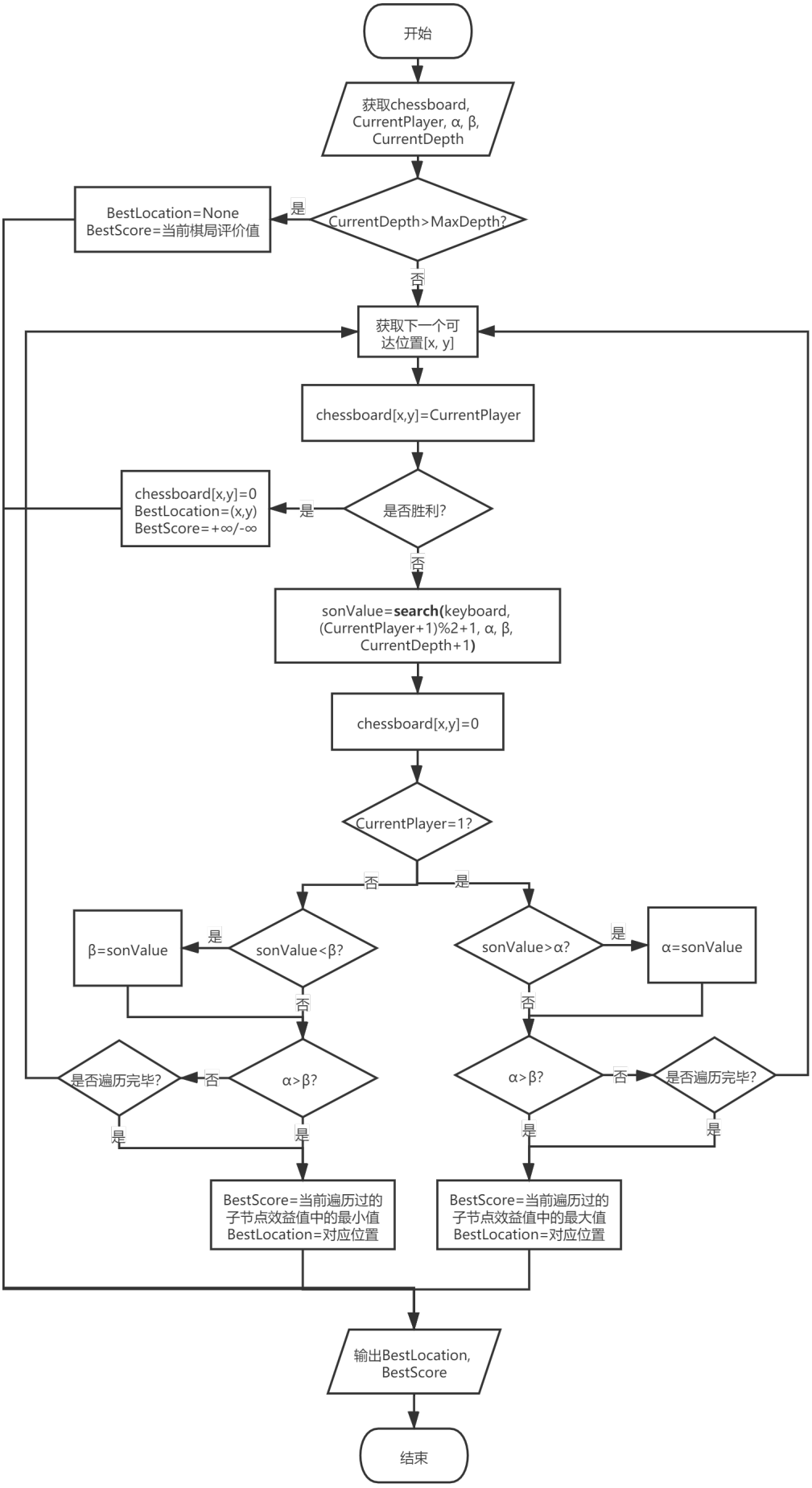

图 4: search() 函数流程图

3 关键代码

3.1 Chess 类

在本次实验中，与五子棋有关的数据结构和操作我都集成到了 Chess 类中。Chess 类的初始化函数如下所示：

```
class Chess():
    def __init__(self,N, ChosenSize, MaxDepth):
        self.size=N # 棋盘边尺寸
        self.chessboard = np.zeros((N,N), dtype=np.int)# 棋盘
        self.ChessScore = np.zeros((N,N))# 各个位置的分数表

        self.initChessboard()        # 初始化棋盘
        self.HumanSize=ChosenSize    # 人类选边
        self.MaxDepth=MaxDepth       # 最大深度


    def initChessboard(self):
        """初始化棋盘"""
        CenterIndex = int(self.size/2)
        self.chessboard[CenterIndex, CenterIndex]=1
        if CenterIndex+1<self.size:
            self.chessboard[CenterIndex, CenterIndex+1]=1# 黑棋标记1
            self.chessboard[CenterIndex+1, CenterIndex]=2# 白棋标记2
        if CenterIndex-1>0:
            self.chessboard[CenterIndex, CenterIndex-1]=2
        return

    ......
```

与计算效益值有关的函数是 getQuintuple() 和 evaluate()。前者用于得到一个五元组的得分，后者用于计算整个棋面的效益值：

```
class Chess():
    ......
    def getQuintupleScore(self, num):
        """五元组分数表"""
        tempScore=0
        if num==1:
            tempScore=10
        elif num==2:
            tempScore=30
        elif num==3:
            tempScore=120
        elif num==4:
            tempScore=450
        elif num==5:
            tempScore=10000
```

```
        return tempScore

    def evaluate(self):
        """计算返回当前棋面的效益值"""
        score=0 # 总效益值
        # 按行判断
        for i in range(self.size):
            Xnum=0
            Onum=0
            for j in range(self.size):
                if self.chessboard[i,j]==1:
                    Xnum+=1
                elif self.chessboard[i,j]==2:
                    Onum+=1
                if j<4:
                    continue
                elif j>4:
                    if self.chessboard[i,j-5]==1:
                        Xnum-=1
                    elif self.chessboard[i,j-5]==2:
                        Onum-=1
                if Xnum>0 and Onum>0:
                    score+=0
                else:
                    score+=self.getQuintupleScore(Xnum)
                    score-=self.getQuintupleScore(Onum)
        # 按列判断
        ......
        # 按 / 判断
        ......
        # 按 \ 判断
        ......
        return score

    ......
```

使用 alpha-beta 剪枝的博弈树搜索函数也在 Chess 类中实现，代码如下所示。可见该函数基于深度优先搜索的基础上完成。为了进一步剪枝、提高搜索速度，我们采用了一个暴力的方式：只搜索附近有棋子的位置。

```
class Chess():
    ......
    def search(self, CurrentPlayer, alpha, beta, CurrentDepth):
        """博弈树搜索"""
        if CurrentDepth>self.MaxDepth:# 超过最大深度，叶节点
            return self.evaluate(), None
        # 初始化BestScore, BestLocation
        if CurrentPlayer==1:
```

```
            BestScore=-float('inf')
        else:
            BestScore=float('inf')
        BestLocation=None
        Talpha=alpha
        Tbeta=beta
        # 遍历所有位置
        for i in range(self.size):
            # if CurrentDepth==1:
            #     print(str(i)+' begin')
            for j in range(self.size):
                if CurrentDepth==1:
                    self.ChessScore[i,j]=float('nan')
                if self.chessboard[i,j]==0 and self.has_neighbor(i,j):# 只考虑周围有棋的空位置
                    self.chessboard[i,j]=CurrentPlayer# 落子
                    if self.judge_win(i,j,CurrentPlayer):# 出现胜利，叶节点
                        self.chessboard[i,j]=0# 收子
                        if CurrentPlayer==1:      # 落子方为黑棋
                            if CurrentDepth==1:
                                self.ChessScore[i,j]=1000000/CurrentDepth# 除以深度是为了尽早胜利
                            BestScore=1000000/CurrentDepth
                            BestLocation=(i,j)
                        elif CurrentPlayer==2:    # 落子方为白棋
                            if CurrentDepth==1:
                                self.ChessScore[i,j]=-1000000/CurrentDepth
                            BestScore=-1000000/CurrentDepth
                            BestLocation=(i,j)
                        return BestScore, BestLocation
                    elif CurrentPlayer==1:      # 落子方为黑棋
                        # 搜索下一个节点
                        SonScore, NextLocation=self.search(2, Talpha, Tbeta, CurrentDepth+1)
                        if CurrentDepth==1:
                            self.ChessScore[i,j]=SonScore
                        # 更新最优效益值与对应位置
                        if SonScore>BestScore:
                            BestScore=SonScore
                            BestLocation=(i,j)
                        # alpha剪枝
                        if SonScore>Talpha:
                            Talpha=SonScore
                        if Talpha>Tbeta:
                            self.chessboard[i,j]=0
                            return BestScore, BestLocation
                    elif CurrentPlayer==2:    # 落子方为白棋
                        # 搜索下一个节点
```

```
                        SonScore, NextLocation=self.search(1, Talpha, Tbeta, CurrentDepth
                            +1)
                        if CurrentDepth==1:
                            self.ChessScore[i,j]=SonScore
                        # 更新最优效益值与对应位置
                        if SonScore<BestScore:
                            BestScore=SonScore
                            BestLocation=(i,j)
                        # beta剪枝
                        if SonScore<Tbeta:
                            Tbeta=SonScore
                        if Talpha>Tbeta:
                            self.chessboard[i,j]=0
                            return BestScore, BestLocation
                    self.chessboard[i,j]=0# 收子
        if BestLocation==None:
            BestScore=0
        return BestScore, BestLocation
```

此外，在 Chess 类中我还实现了用于显示当前得分的函数 displayScore()、判断当前落子是否胜利的函数 judge_win() 等。由于实现思路比较简单、也不是展示的重点，故不再列出。

3.2 主程序设计

本次实验我使用 pygame 进行可视化设计。首先我使用了 Settings 类保存游戏运行时必要的参数，代码如下所示：

```
class Settings():
    def __init__(self, humansize):
        # 屏幕数据
        self.screen_width = 900
        self.screen_height = 900
        # 五子棋相关数据
        self.CurrentPlayer=1     # 黑棋先手
        self.HumanSize = humansize# 人类所执的棋
        self.movement=False      # 用于判断人类下棋动作
        self.isFinish=0          # 游戏结束标志
```

然后我们在主程序中完成整个游戏的流程，代码如下所示：

```
def main():
    """游戏主程序"""
    # 初始化信息
    ......
    # 初始化五子棋数据
    chess=Chess(11,HumanSize,MaxDepth)
    # 初始化游戏并创建一个屏幕对象
    pygame.init()
    game_settings=Settings(HumanSize)
```

```
    screen = pygame.display.set_mode((game_settings.screen_width, game_settings.
        screen_width))
    pygame.display.set_caption("五子棋")
    board=Board(screen)
    global WChessList# 白棋显示对象列表
    global BChessList# 黑棋显示对象列表
    global ScoreList # 分数显示列表
    global step      # 当前步数
    global NumList  # 序号显示列表
    ......
    # 开始游戏的主循环
    while True:
        # 继续下棋
        if game_settings.isFinish==0:
            if game_settings.CurrentPlayer!=game_settings.HumanSize:# 轮到电脑下棋
                # 获取最优位置
                CBestScore, BestLocation=chess.search(game_settings.CurrentPlayer,-float('
                    inf'),float('inf'),1)
                chess.displayScore()
                if BestLocation==None:# 没有位置可下，平局
                    game_settings.isFinish=-1
                else:
                    chess.chessboard[BestLocation[0],BestLocation[1]]=game_settings.
                        CurrentPlayer# 落子
                    NumList.append((BestLocation[1],BestLocation[0]))
                    ScoreList[step%2]=chess.evaluate()
                    step+=1
                    # 判断胜利
                    if chess.judge_win(BestLocation[0],BestLocation[1], game_settings.
                        CurrentPlayer):
                        game_settings.isFinish=game_settings.CurrentPlayer
                    # 换边
                    if game_settings.CurrentPlayer==1:
                        BChessList.append(BlackChess(screen,BestLocation[1],BestLocation
                            [0]))
                        star.set_position(BestLocation[1],BestLocation[0])
                        game_settings.CurrentPlayer=2
                    elif game_settings.CurrentPlayer==2:
                        WChessList.append(WhiteChess(screen,BestLocation[1],BestLocation
                            [0]))
                        star.set_position(BestLocation[1],BestLocation[0])
                        game_settings.CurrentPlayer=1

        # 监视键盘和鼠标事件
        check_events(game_settings, screen, chess, star)
        #显示
        ......
```

人类玩家的动作在函数 check_events() 中完成识别与处理：

```
def check_events(settings, screen, chess, star):
    """响应按键和鼠标事件"""
    global WChessList# 白棋显示对象列表
    global BChessList# 黑棋显示对象列表
    global ScoreList # 分数显示列表
    global step      # 当前步数
    global NumList  # 序号显示列表
    for event in pygame.event.get():
        if event.type == pygame.QUIT:        # 关闭
            sys.exit()
        elif event.type == pygame.KEYDOWN:
            ......
        elif event.type == pygame.KEYUP:
            ......
        elif event.type == pygame.MOUSEBUTTONDOWN:# 检测按下鼠标
            if settings.isFinish!=0:            # 游戏结束后不再响应
                return
            if settings.movement==False:     # 只响应一次点击
                if settings.HumanSize!=settings.CurrentPlayer:# 当前下棋方不是人类时不响应
                    continue
                mouse_x, mouse_y=event.pos  # 获取鼠标点击位置
                if mouse_x>=780 or mouse_x<120 or mouse_y>=780 or mouse_y<120:
                    continue                # 点击位置不在识别范围内时不响应
                # 将鼠标点击位置映射到落子位置
                IX=int((mouse_x-120)/60)
                IY=int((mouse_y-120)/60)
                if chess.chessboard[IY,IX]!=0:
                    continue                # 落子位置已有子，不响应
                chess.chessboard[IY,IX]=settings.CurrentPlayer# 落子
                NumList.append((IX, IY))
                ScoreList[step%2]=chess.evaluate()
                step+=1
                if chess.judge_win(IY, IX, settings.CurrentPlayer):# 判断是否赢了
                    settings.isFinish=settings.CurrentPlayer
                # 切换下棋方
                if settings.CurrentPlayer==1:
                    BChessList.append(BlackChess(screen,IX,IY))# 创建黑棋显示对象
                    star.set_position(IX,IY)
                    settings.CurrentPlayer=2
                else:
                    WChessList.append(WhiteChess(screen,IX,IY))# 创建白棋显示对象
                    star.set_position(IX,IY)
                    settings.CurrentPlayer=1
                # 判断棋盘是否无子可下、到达平局
                isFinishT=True
                for i in range(chess.size):
                    for j in range(chess.size):
                        if chess.chessboard[i,j]==0:
```

```
                                isFinishT=False
                                break
                        if isFinishT==False:
                            break
                    if isFinishT and settings.isFinish==0:
                        settings.isFinish=-1

                settings.movement=True
            elif event.type == pygame.MOUSEBUTTONUP:# 检测松开鼠标
                settings.movement=False
```

以上就是程序的所有关键代码。

4 实验过程与结果分析

我们使用以上程序进行测试。人类玩家执黑棋，电脑执白棋，难度（最大搜索深度）为 4。为了进一步减少搜索量，我们设置电脑只会搜索在曼哈顿距离 1 以内有棋子的空闲点。

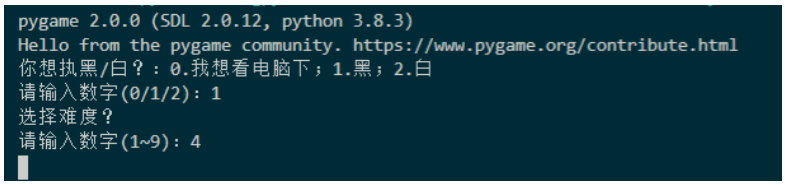

图 5: 游戏前参数设置

第一轮对弈结果如图6所示，可见我方落子后评估值为 200，轮到电脑落子后评估值变为-100。此时棋面上有 22 个五元组只有一个白棋，有 5 个五元组有两个白棋，有 24 个五元组只有一个黑棋，有 1 个五元组有两个黑棋，其余五元组要么没有棋子存在、要么既有黑棋又有白棋，因此评估值为 $-22\times10-5\times30+24\times10+1\times30=-100$。电脑在搜索落子点时的数据如图7所示，其中 nan 表示该点要么已有棋子存在、要么电脑没有搜索。可以看到，电脑落子的位置正好是搜索值最小的位置。之所以这位置的搜索值为 360、而棋面评估值为-100，这时因为搜索值是 4 步以后对于电脑的最优评估值。

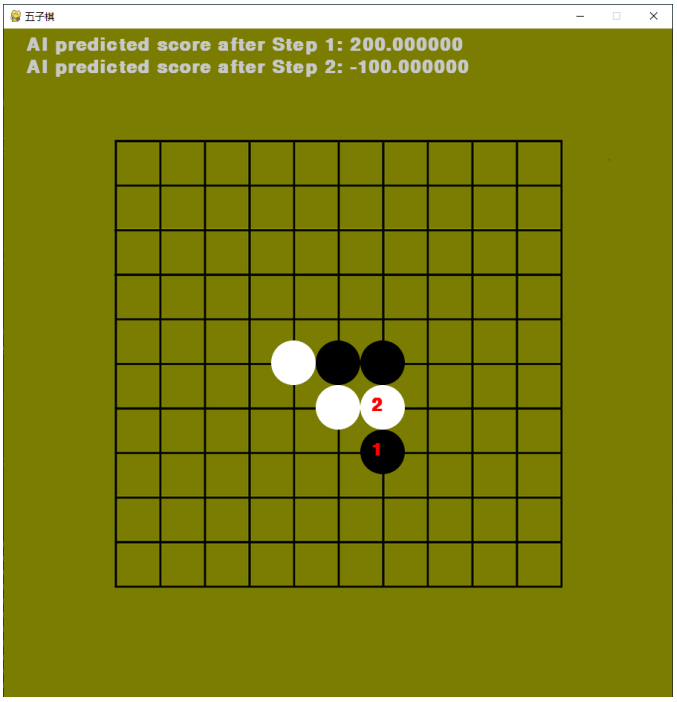

图 6: 第一轮对弈

nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
nan	nan	nan	530.0	480.0	490.0	450.0	460.0	nan	nan	nan
nan	nan	nan	470.0	nan	nan	nan	460.0	nan	nan	nan
nan	nan	nan	460.0	460.0	nan	360.0	380.0	nan	nan	nan
nan	nan	nan	nan	370.0	410.0	nan	410.0	nan	nan	nan
nan	nan	nan	nan	nan	370.0	370.0	390.0	nan	nan	nan
nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan

图 7: 电脑落子时各点的搜索值

第二轮对弈结果如图8所示，电脑搜索得到的数据如图9所示。第三轮对弈结果如图10所示，电脑搜索得到的数据如图11所示。可以看到，电脑始终在尝试降低评估值，并且落子显得有章法。可以初步判定我们的程序是正确的。

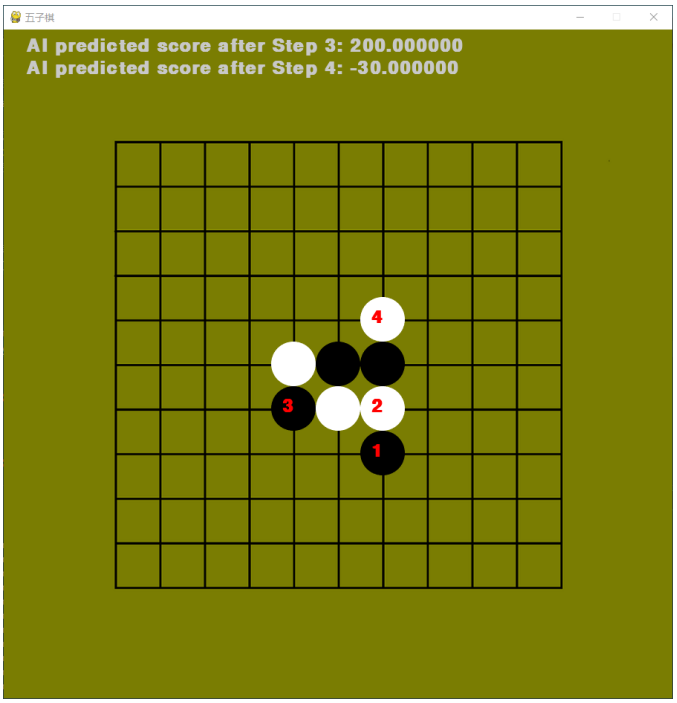

图 8: 第二轮对弈

nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
nan	nan	nan	530.0	550.0	570.0	370.0	380.0	nan	nan	nan
nan	nan	nan	390.0	nan	nan	nan	380.0	nan	nan	nan
nan	nan	nan	400.0	nan	nan	nan	380.0	nan	nan	nan
nan	nan	nan	410.0	400.0	380.0	nan	390.0	nan	nan	nan
nan	nan	nan	nan	nan	390.0	410.0	430.0	nan	nan	nan
nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan

图 9: 电脑落子时各点的搜索值

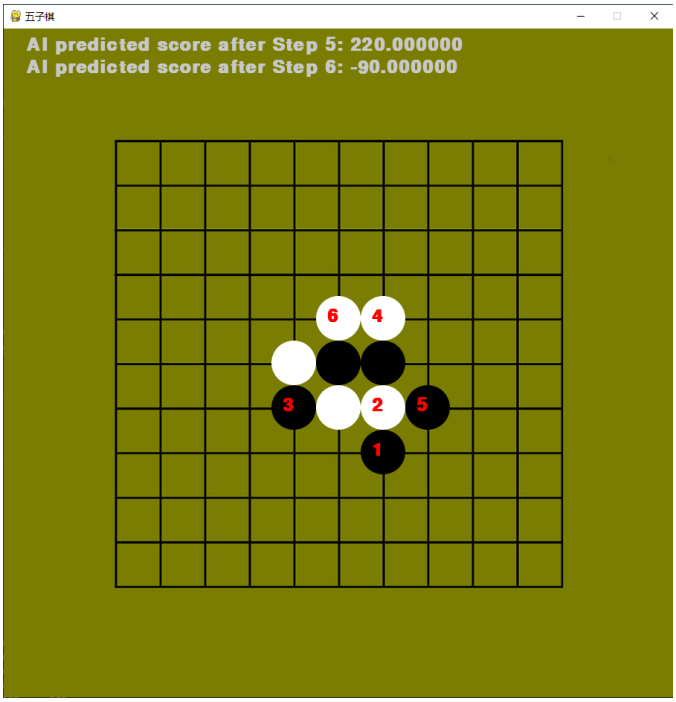

图 10: 第三轮对弈

nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
nan	nan	nan	nan	nan	760.0	720.0	730.0	nan	nan	nan
nan	nan	nan	640.0	650.0	540.0	nan	550.0	nan	nan	nan
nan	nan	nan	560.0	nan	nan	nan	540.0	550.0	nan	nan
nan	nan	nan	590.0	nan	nan	nan	nan	570.0	nan	nan
nan	nan	nan	560.0	550.0	560.0	nan	580.0	550.0	nan	nan
nan	nan	nan	nan	nan	560.0	560.0	550.0	nan	nan	nan
nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan

图 11: 电脑落子时各点的搜索值

最终对弈结果如图12所示，可见执白棋的电脑取得了胜利。

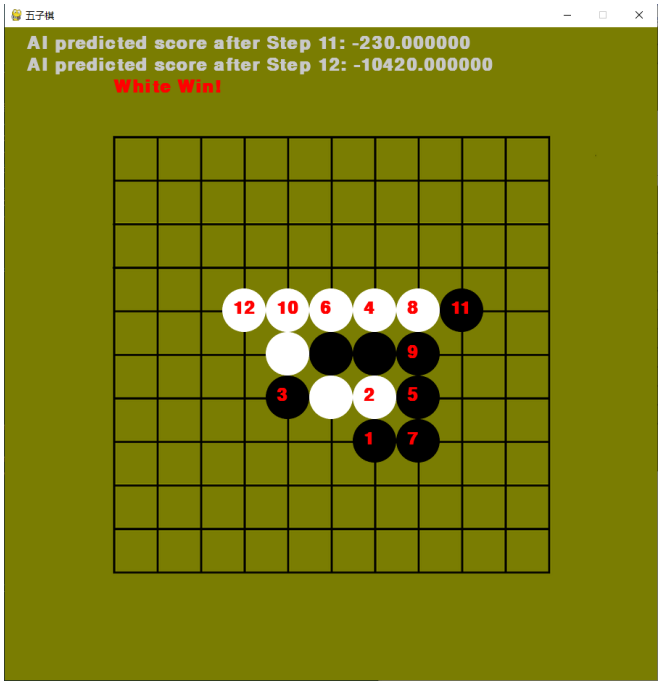

图 12: ~~水平太菜~~故意放水输的

此外，我也尝试了让电脑之间进行对弈。设置难度（最大搜索深度）为 2，电脑只会搜索在曼哈顿距离 2 以内有棋子的空闲点。最终对弈结果如图13所示，可见黑棋赢得了最终胜利。

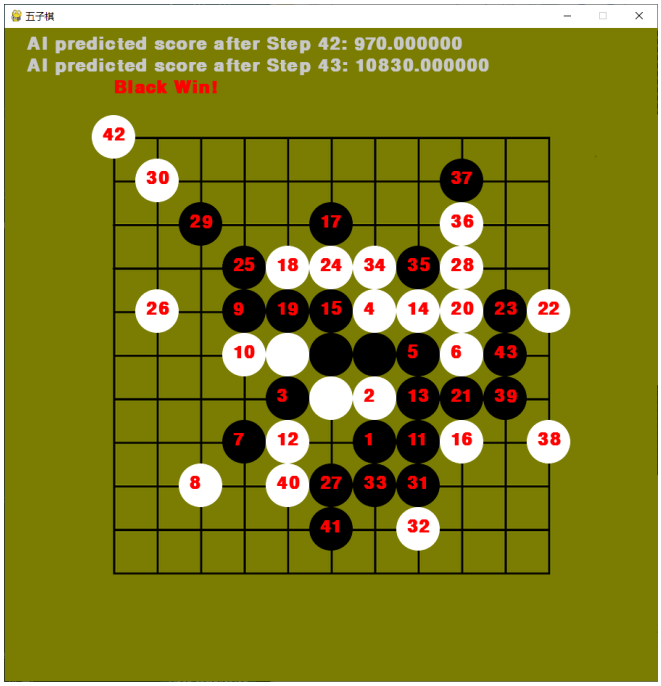

图 13: 电脑间对弈结果

5 实验总结与感想

本次实验要求我们使用博弈树搜索实现电脑下五子棋。从实验结果来看，本次实验比较有趣，也算是实现了一个“人工智能”。当设置难度为 2 时，AI 就已经与我的水平不相上下了，当难度为 4 时，我就没有赢过（笑）。

当然，本次我的程序也有不完善的地方。首先是评价函数中几个五元组的评分是我自己估摸着设置的，没有太多科学依据。以后我们可以使用机器学习的方式完善评价函数。此外，当最大搜索深度大于 3 时，搜索时间大大增加。尤其下到后期时可探索节点数量增加，等待时间到了难以忍受的地步。这说明程序还存在进一步优化的空间。

希望我能顺利完成下一次实验。