人工智能实验报告 LAB5

（2021学年秋季学期）

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	19335174	姓名	施天予

一、 实验题目

博弈树搜索

二、 实验内容

1、算法原理

博弈树搜索

博弈树针对的是二人零和博弈的问题，二人轮流行动，行动时令自己的优势最大。可以用博弈树的每个节点表示一个确定的状态，在动作后得到的新状态作为子节点。对于每个状态都有同一个评价函数来评估双方的得分。一方通过决策使得自身的评价函数尽可能的大，另一方让队手的评价函数尽可能的小。因为二者是轮流行动的，在树的每一层让一方的评价函数取最大和最小交替进行。

由上述的特性，博弈树的搜索过程又被称为minimax搜索。博弈双方行动逐层交替，将评价函数值看做一方的分数，在那一方行动时要让分数尽可能的大，这样的节点被称为Max节点；在另一方行动时要让分数尽可能的小，这样的节点被称为Min节点。

要让一方的下一步采取最优的策略，需要进行树的搜索。在实际问题中，树往往非常大，因此只考虑一定的深度，而不是整个遍历。进行深入搜索时，轮流考虑Max节点和Min节点，每次都采取最优策略，最终得到本步的最优策略。

$\alpha-\beta$剪枝

然而如果通过纯暴力搜索博弈树以寻找最佳策略，检查的状态数目随着博弈的进行呈指数增长，效率十分低下，因此需要剪枝。在博弈树的每个节点保存两个值：α表示在该节点能达到的分数的下界，初始化为$-\infty$，β表示该节点能达到的分数的上界，初始化为∞。

Max节点的剪枝

Max节点的β值初始化时应该为父节点的β值。因为Max节点的父节点是Min节点，如果Max节点的β值大于父节点的β值，Max节点最终得到的估值必然会大于父节点的β值，从而表示的状态被不会被父节点选择。之后，Max节点依次生成子节点。每生成完一个子节点就将子节点的α值传递回来。因为子节点为Min节点，会取到分数的最小值，因此必然会取到它的下界α，也就是说，Min节点最终的的α值就是它的估值。而Max会取子节点中估值最大的，因此，要通过子节点的α值来提高自身评分的下界，也就是说，如果子节点的α值大于自身的α值，则将自身的α值更新为更大的那一个。当$\alpha > \beta$时，该节点的估值一定会大于父节点的估值上界，而父节点是Min节点，是必然不会选择当前节点的。因此所有的子节点可以停止拓展，从而实现了剪枝。

Min节点的剪枝

Min节点的α值初始化时应该为父节点的α值。因为Min节点的父节点是Max节点，如果Min节点的α值小于父节点的α值，Min节点最终得到的估值必然会小于父节点的α值，从而表示的状态不会被父节点选择。之后，Min节点依次生成子节点。每生成完一个子节点就将子节点的β值传递回来。因为子节点为Max节点，会取到分数的最大值，因此必然会取到它的上界β，也就是说，Max节点最终的β值就是它的估值。而Min节点会取子节点中估值最小的，因此要通过子节点的β值来提高自身评分的上界，也就是说，如果子节点的β值小于自身的β值，则将自身的β值更新为更小的那一个。当$\alpha > \beta$时，该节点的估值一定会小于父节点的估值下界，而父节点是Max节点，是必然不会选择当前节点的。因此所有的子节点可以停止拓展，从而实现了剪枝。

评价函数

因为黑白棋的规则有些复杂，本人在4399小游戏与电脑对战时只能通过第一关，百思不得其解黑白棋的奥妙所在。所以我只好在网上搜索了黑白棋的评价函数原理，发现在四个角落的棋子是无法翻转的，所以可以将其的评分设得比较高，而在角落周围落子很容易让对方可以在角落落子，所以设计为负分。其他部分也可进行相应设计。

```
Vmap = np.array([[100, -25, 10, 5, 5, 10, -25, 100],
                 [-25, -45, 1, 1, 1, 1, -45, -25],
                 [10, 1, 3, 2, 2, 3, 1, 10],
                 [5, 1, 2, 1, 1, 2, 1, 5],
                 [5, 1, 2, 1, 1, 2, 1, 5],
                 [10, 1, 3, 2, 2, 3, 1, 10],
                 [-25, -45, 1, 1, 1, 1, -45, -25],
                 [100, -25, 10, 5, 5, 10, -25, 100]])
```

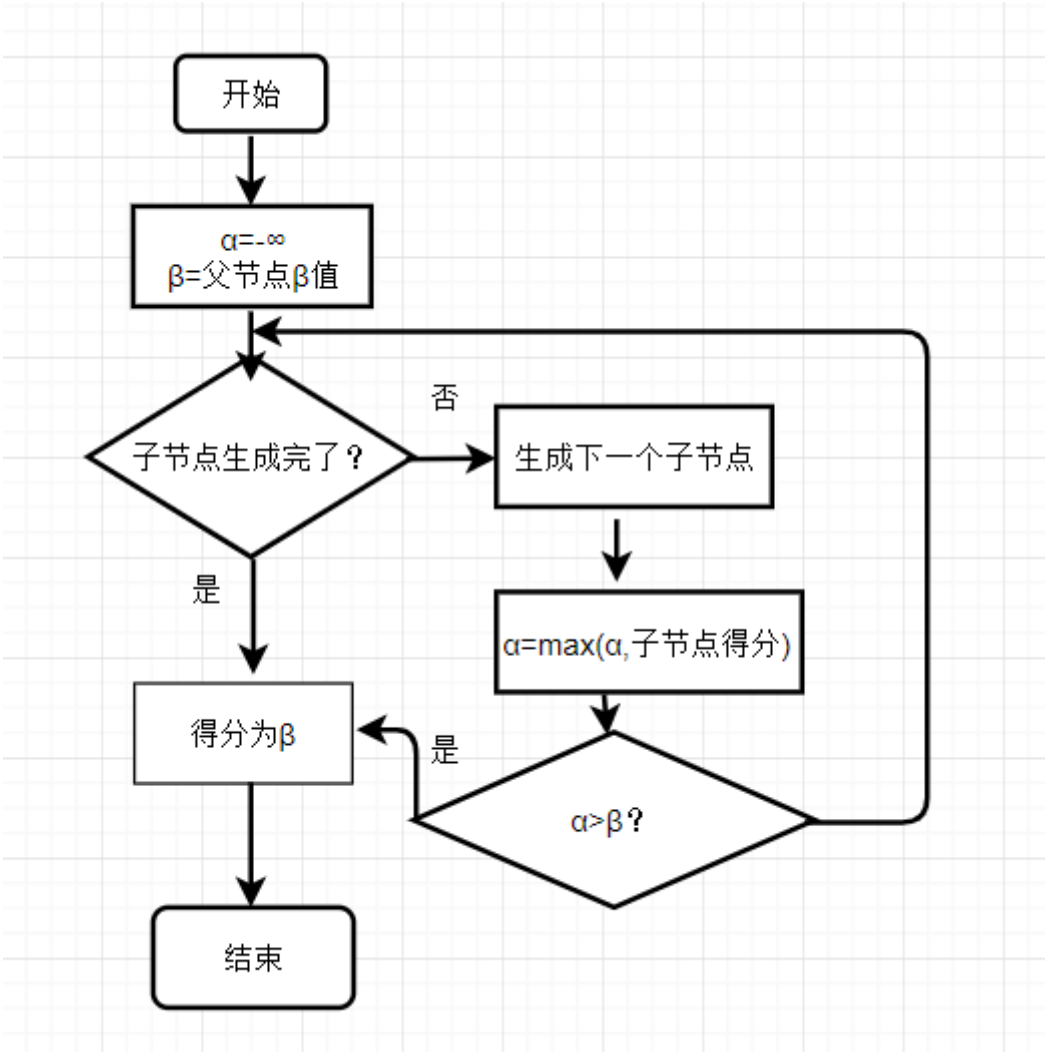
2、伪代码和流程图

MiniMax搜索（$\alpha-\beta$剪枝）

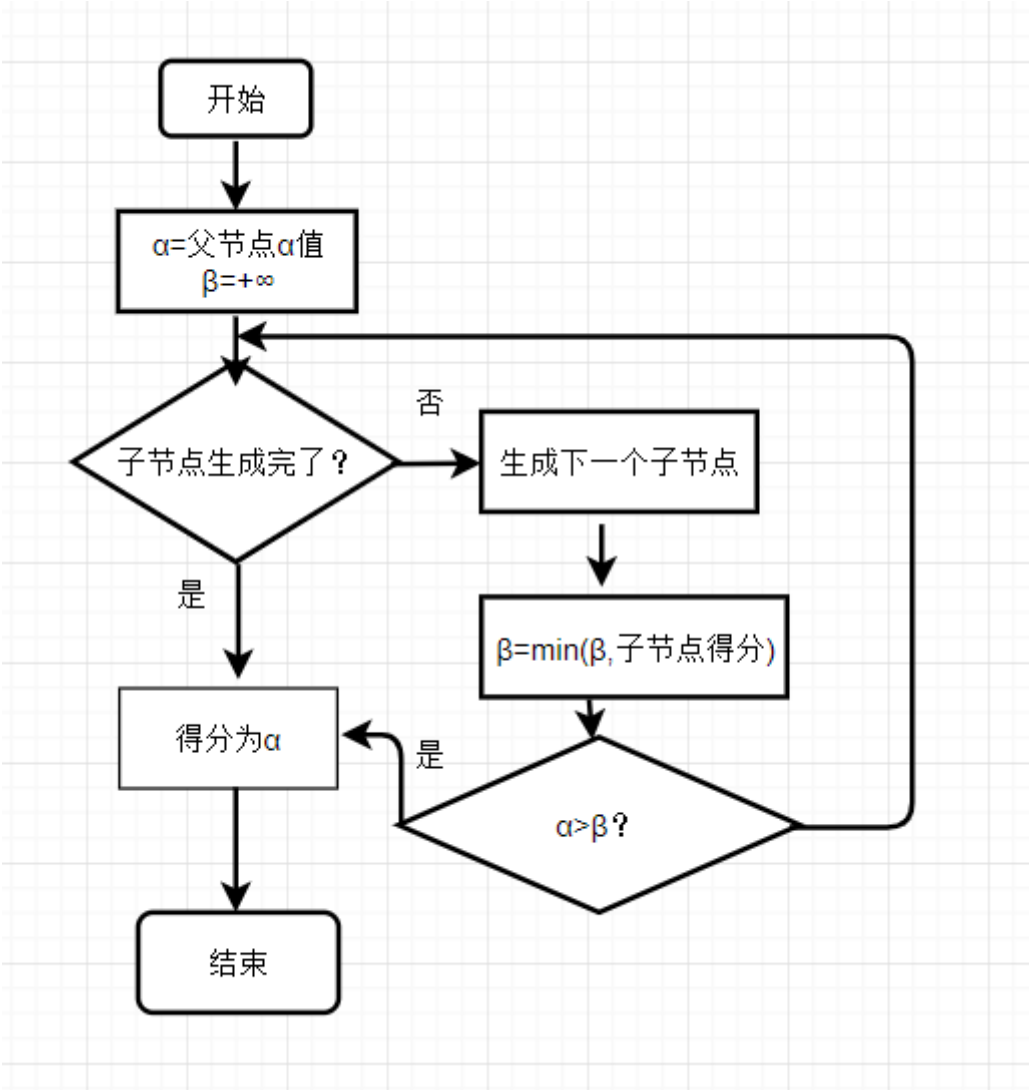
```
Function Search:
Input: state, computercolor
Output: bestMove
    for x, y in possibleMoves:  /*每个可落子的位置*/
        Move(x, y)
        score = MaxValue(state,-INF,INF,depth)
        if score > bestScore:
            bestMove = [x, y]
            bestScore = score
    return bestMove
/************************************************/
Function MaxValue:
Input: state, alpha, beta
Output: bestvalue
    if depth == 1:
        return 当前state的value
    bestValue = -INF
    for x,y in possible:    /*每个可落子的位置*/
        Move(x, y)
        bestValue = max(bestValue, MinValue(state,alpha,beta,depth-
1))
        if bestValue >= beta:
            return bestValue
        alpha = max(alpha, bestValue)
    return bestValue
/************************************************/
Function MinValue:
Input: state, alpha, beta
Output: bestvalue
    if depth == 1:
        return 当前state的value
    bestValue = INF
    for x,y in possible:    /*每个可落子的位置*/
        Move(x, y)
```

```
34          bestValue = min(bestValue, MaxValue(state,alpha,beta,depth-
    1))
35          if bestValue <= alpha:
36              return bestValue
37          beta = min(beta, bestValue)
38      return bestValue
```

生成Max节点



生成Min节点



3、关键代码展示

因为黑白棋的完整代码很长，所以就选了几个关键的函数进行讲解，其中搜索的深度depth=5

Score函数用于估计棋盘上当前局面黑棋和白棋的分数，棋盘的每个位置对应不同的分值

```
# 获取棋盘上黑白双方的估值
def Score(board):
    BoardBlack = np.zeros((8,8))
    BoardWhite = np.zeros((8,8))
    # 棋盘估值表
    Vmap = np.array([[100, -25, 10, 5, 5, 10, -25, 100],
                     [-25, -45, 1, 1, 1, 1, -45, -25],
                     [10, 1, 3, 2, 2, 3, 1, 10],
                     [5, 1, 2, 1, 1, 2, 1, 5],
                     [5, 1, 2, 1, 1, 2, 1, 5],
                     [10, 1, 3, 2, 2, 3, 1, 10],
                     [-25, -45, 1, 1, 1, 1, -45, -25],
                     [100, -25, 10, 5, 5, 10, -25, 100]])
```

```
    for x in range(8):
        for y in range(8):
            if board[x][y] == 'X':
                BoardBlack[x][y] = 1
            if board[x][y] == 'O':
                BoardWhite[x][y] = 1

    BoardBlack = BoardBlack * Vmap
    BoardWhite = BoardWhite * Vmap
    BlackValue = np.sum(BoardBlack)
    WhiteValue = np.sum(BoardWhite)
    return {'X': BlackValue, 'O': WhiteValue}
```

FlipPosition函数用于返回能被翻转的棋子位置，没有则返回False。假设要落子在当前位置，然后从当前位置向8个方向移动，如果移动过程中全是对方的棋子，且最终有一颗是自己的棋子，那么就将中间这些对方的棋子位置记录下来，最后返回

```
def FlipPosition(board, Color, x, y):
    # 如果出界了或者该位置已经有棋子，返回False
    if not OnBoard(x, y) or board[x][y] != '.':
        return False
    if Color == 'X':
        otherColor = 'O'
    else:
        otherColor = 'X'
    # 要被翻转的棋子
    Flipchess = []
    for xdirection, ydirection in [[0, 1],[1, 1],[1, 0],[1, -1],[0, -1],[-1, -1],[-1, 0],[-1, 1]]:
        i, j = x, y
        i += xdirection
        j += ydirection
        if OnBoard(i, j) and board[i][j] == otherColor:
            i += xdirection
            j += ydirection
            # 一直走到出界或不是对方棋子的位置
            while OnBoard(i, j) and board[i][j] == otherColor:
                i += xdirection
                j += ydirection
            # 出界了，则没有棋子要翻转
            if not OnBoard(i, j):
                continue
            # 是自己的棋子
            if board[i][j] == Color:
                while True:
                    i -= xdirection
                    j -= ydirection
                    # 回到了起点则结束
```

```
                    if i == x and j == y:
                        break
                    # 需要翻转的棋子
                    Flipchess.append([i, j])
    # 没有要被翻转的棋子，则走法非法。翻转棋的规则。
    if len(Flipchess) == 0:
        return False
    return Flipchess
```

ValidMoves函数用于返回可落子的位置，如果该位置能够有翻转的棋子，那么该位置就是可以落子的

```
def ValidMoves(board, Color):
    validMoves = []
    for x in range(8):
        for y in range(8):
            if FlipPosition(board, Color, x, y) != False:
                validMoves.append([x, y])
    return validMoves
```

makeMove函数用于将棋子落在(x, y)位置，并在棋盘中翻转所有应当翻转的棋子

```
def makeMove(board, Color, x, y):
    Flipchess = FlipPosition(board, Color, x, y)
    if Flipchess == False:
        return False
    board[x][y] = Color
    for i, j in Flipchess:
        board[i][j] = Color
    return True
```

ComputerMove函数用于电脑选择最佳走法，从所有可落子的位置中一一计算，进行MiniMax搜索，选出最优的走法，返回走法和效益值

```
def ComputerMove(board, computerColor):
    bestMove = []
    possibleMoves = ValidMoves(board, computerColor)
    bestScore = 0
    for x, y in possibleMoves:
        dupeBoard = CopyBoard(board)
        makeMove(dupeBoard, computerColor, x, y)
        score = MaxValue(dupeBoard,computerColor,-INF,INF,depth)
        if score is not INF and score > bestScore:
            bestMove = [x, y]
            bestScore = score
    if len(bestMove) == 0:
        for x, y in possibleMoves:
```

```
            bestMove = [x,y]
            break
    return bestMove, bestScore
```

Minimax搜索的Max节点层，选择效益值最大的节点值。如果效益值 >= 任何祖先Max节点alpha值，则进行beta剪枝

```
def MaxValue(board,Color,alpha,beta,depth):
    if depth == 1:
        return Score(board)[Color]
    bestValue = -INF
    if Color == 'X':
        otherColor = 'O'
    else:
        otherColor = 'X'
    possible = ValidMoves(board,Color)
    for x,y in possible:
        copyBoard = CopyBoard(board)
        makeMove(copyBoard,Color,x,y)
        bestValue = max(bestValue,
MinValue(copyBoard,otherColor,alpha,beta,depth-1))
        if bestValue >= beta:    # Max节点alpha剪枝：效益值 >= 任何祖先Min节点beta值
            return bestValue
        alpha = max(alpha, bestValue)
    return bestValue
```

Minimax搜索的Min节点层，选择效益值最小的节点值。如果效益值 <= 任何祖先Min节点beta值，则进行alpha剪枝

```
def MinValue(board,Color,alpha,beta,depth):
    if depth == 1:
        return Score(board)[Color]
    bestValue = INF
    if Color == 'X':
        otherColor = 'O'
    else:
        otherColor = 'X'
    possible = ValidMoves(board,Color)
    for x,y in possible:
        copyBoard = CopyBoard(board)
        makeMove(copyBoard,Color,x,y)
        bestValue = min(bestValue,
MaxValue(copyBoard,otherColor,alpha,beta,depth-1))
        if bestValue <= alpha:  # Min节点beta剪枝：效益值 <= 任何祖先Max节点alpha值
            return bestValue
        beta = min(beta, bestValue)
```

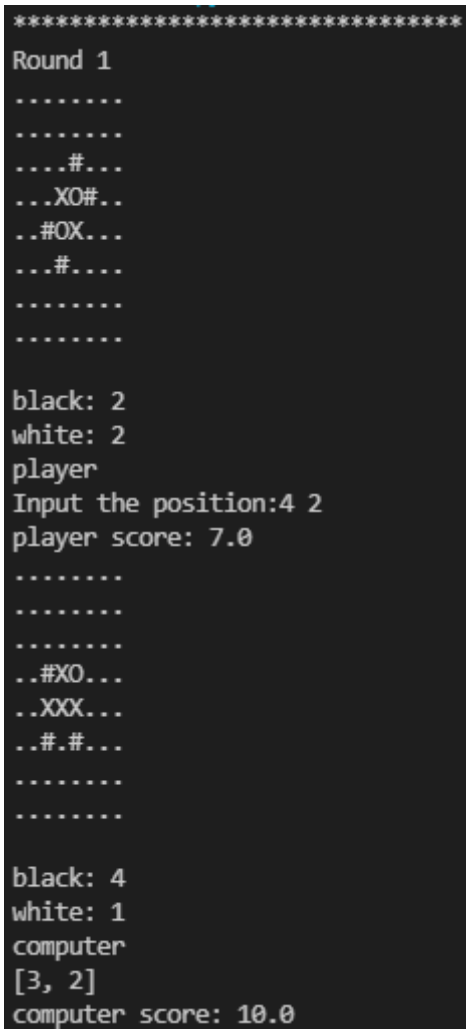
```
17      return bestValue
```

三、 实验结果及分析

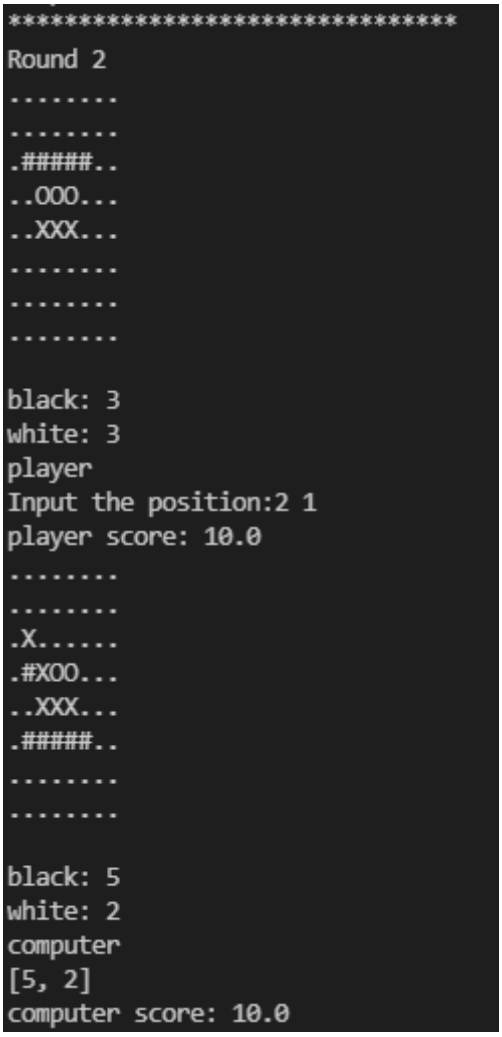
1、实验结果展示示例

搜索深度为5，玩家先手。X代表黑棋（先手），O代表白棋（后手）。每一步输出落子的位置，黑棋白棋的个数，以玩家和电脑的分数。

第一回合：玩家落子位置（4，2），得分7分；电脑落子位置（3，2），得分10分。



第二回合：玩家落子位置（2，1），得分10分；电脑落子位置（5，2），得分10分。



第三回合：玩家落子位置（2，2），得分12分；电脑落子位置（1，2），得分24分。

```
********************************
Round 3
........
........
.X#.#...
..XOO#..
..XOX...
..O.#...
..#.....
........

black: 4
white: 4
player
Input the position:2 2
player score: 12.0
........
#.#.....
.XX#....
.#XXO...
.#XOX#..
..O.#...
........
........

black: 6
white: 3
computer
[1, 2]
computer score: 24.0
```

三个回合后的棋局，电脑7：3领先

```
********************************
Round 4
...#....
.#O.....
.XO##...
.#OXO#..
.#OOX...
.#O##...
........
........

black: 3
white: 7
```

2、评测指标展示及分析

我也尝试了多种搜索深度，比较玩家和电脑的得分（玩家：电脑）

	深度为1	深度为3	深度为5
第一回合	4：5	6：7	7：10
第二回合	7：5	8：9	10：10
第三回合	8：8	11：21	12：24

可以发现，随着搜索深度的增大，电脑“预见未来”的能力越来越强，玩家与电脑的得分差距也会越来越大；同时随着回合的深入，玩家与电脑的得分差距也会越来越大，局面越来越被动。经过多次尝试，“愚蠢”的我没有一次能战胜电脑。