中山大学计算机学院

人工智能

本科生实验报告

（2023学年春季学期）

课程名称：Artificial Intelligence

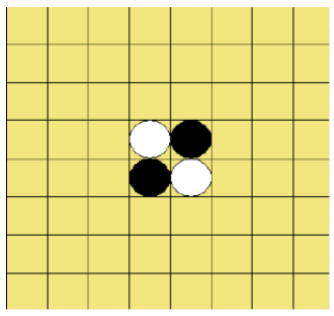
教学班级	计科1班	专业（方向）	计算机科学与技术
学号	21307035	姓名	邓栩瀛

一、实验题目

实现8×8的黑白翻转棋的人机对战：

- 要求使用alpha-beta剪枝；
- 不要求实现UI；
- 搜索深度和评价函数不限，自己设计。在报告中说明清楚自己的评价函数及搜索策略。
- 鼓励大家结合高级搜索算法优化评价函数。
- 实验结果要求展示至少连续三个回合（人和机器各落子一次指一回合）的棋局分布情况，并输出每步落子的得分。

黑白棋游戏规则：

- 黑棋先手，初始棋局如下：
- 
- 只允许在可以翻转的位置落子
- 落子后，如果在横排、竖排、对角线上有另一自己颜色的棋子，则夹在中间连续排布的另一颜色的棋子会翻转

二、实验内容

1.算法原理

$Minimax$算法

- 一种极小极大博弈树的搜索算法
- 在minimax博弈树搜索中，每个节点代表一个游戏状态，每个节点下面有若干个子节点，代表下一步所有可能的走法。
- 在max层中选择所有子节点中的最大值作为当前节点的值，在mini层所有子节点中的最小值作为当前节点的值，然后递归地搜索整个博弈树，找到最优解。
- 用DFS遍历博弈树，对于每个节点，递归调用Minimax函数，并根据当前节点是max节点还是mini节点选择对应的子节点。
- 此外，需要定义一个评价函数来评估当前游戏状态的好坏程度。对于终止节点，直接使用评价函数计算该节点的值；对于非终止节点，递归调用Minimax函数求出子节点的值，并根据当前节点是max节点还是mini节点选择最大值或最小值作为当前节点的估值。

steps

1. 如果搜索深度达到了预设的深度，或者当前节点是终止节点，就返回该节点的估值。
2. 对于当前节点的每个子节点，递归调用minimax函数，并根据当前节点是max节点还是mini节点选择最大值或最小值作为该节点的估值。
3. final:返回当前节点的估值，最大值(max节点)/最小值(mini节点)

缺点：搜索空间大，效率低

$\alpha-\beta$剪枝算法

一种Minimax博弈树的优化算法：在Minimax算法的基础上进一步减少搜索空间，提高搜索效率

$\alpha-\beta$剪枝算法通过剪枝掉一些不必要的搜索分支，从而减小搜索空间。

设置两个值：α和β。对于每个max节点，我们初始化α为负无穷大，对于每个mini节点，我们初始化β为正无穷大。在搜索过程中，如果一个max节点的最大值大于等于β，或者一个mini节点的最小值小于等于α，由于子树不会对最终结果产生影响，因此将剪掉该节点的所有子树，从而提高搜索效率。

steps

1. 如果搜索深度达到了预设的深度，或者当前节点是终止节点，就返回该节点的估值。
2. 对于一个极大节点，初始化α为$-\infty$，对于一个极小节点，初始化β为$+\infty$。
3. 对于当前节点的每个子节点，递归调用minimax函数，并根据当前节点是max节点还是mini节点选择最大值或最小值作为该节点的估值。
4. 更新：如果当前节点是max节点，就更新α的值为所有子节点中最大的估值；如果当前节点是mini节点，就更新β的值为所有子节点中最小的估值。
5. 在更新完α或β的值之后，检查α和β的大小关系，判断是否剪掉该节点的所有子树。
6. final:返回当前节点的估值，最大值(max节点)/最小值(mini节点)。

2.伪代码

$Minimax$算法

- 对于每个max节点，选择所有子节点中的最大值作为当前节点的估值；对于每个mini节点，选择所有子节点中的最小值作为当前节点的估值。
- 定义一个评价函数`evaluate`来评估当前游戏状态的好坏程度，并根据这个函数计算每个终止节点的估值

```
function minimax(node, depth, player):
    if depth = 0 or node is a terminal node:
        return evaluate(node)
    if player:
        best_score = -infinity
        for each child of node:
            max_value = minimax(child, depth - 1, false)
            best_score = max(best_score, max_value)
        return best_score
    else:
        best_score = +infinity
        for each child of node:
            min_value = minimax(child, depth - 1, true)
            best_score = min(best_score, min_value)
        return best_score
```

`node`当前节点

`depth`当前搜索深度，

`player`当前是极大节点(true)/极小节点(false)

$\alpha-\beta$剪枝算法

- 在Minimax算法的基础上，增加alpha和beta的更新操作，并在更新之后检查是否需要剪枝。
- 在每个max节点维护一个α值，表示当前极大节点能够保证的最大值；在每个mini节点中维护一个β值，表示当前极小节点能够保证的最小值。如果一个max节点的α值大于等于β，或者一个极小节点的β值小于等于α，将剪掉该节点的所有子树。

```
function alphabeta(node, depth, alpha, beta, player):
    if depth = 0 or node is a terminal node:
        return evaluate(node)
    if player:
        best_score = -infinity
        for each child of node:
            max_value = alphabeta(child, depth - 1, alpha, beta, false)
            best_score = max(best_score, max_value)
            alpha = max(alpha, best_score)
            if beta <= alpha:
                break
```

```
            return best_score
        else:
            best_score = +infinity
            for each child of node:
                min_value = alphabeta(child, depth - 1, alpha, beta, true)
                best_score = min(best_score, min_value)
                beta = min(beta, best_score)
                if beta <= alpha:
                    break
            return best_score
```

`node`当前节点

`depth`当前搜索深度，

`player`当前是极大节点(true)/极小节点(false)

`alpha` `beta`当前max节点/mini节点的最优值

3.关键代码展示（带注释）

设计一个`Board`类，用来完成棋盘的相关操作，包括以下几个函数

```
def __init__(self) #初始化棋盘
def __getitem__(self, index) #按坐标读取棋盘中的值
def print_board(self) # 打印当前棋盘的状态
def is_valid_move(self, x, y, color) # 判断是否为合法的落子点
def make_move(self, x, y, color) # 在指定位置落子
def count_chess_pieces(self) # 计算黑棋和白棋的数目
def is_game_over(self) # 判断游戏是否结束
def is_white_over(self) # 判断白棋是否还有有效的落子点
def is_black_over(self) # 判断黑棋是否还有有效的落子点
```

`Board`类中的关键函数分析：

- 判断落子是否有效函数`is_valid_move`

```
def is_valid_move(self, x, y, color):
    if self.board[x][y] != 0: # 落子点为非空，则返回false
        return False
    # 从八个方向判断落子点是否有效
    for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (1, -1), (-1,
1), (-1, -1)]:
        i, j = x + dx, y + dy
         if i < 0 or i >= 8 or j < 0 or j >= 8 or self.board[i][j] == 0 or
self.board[i][j] == color:#到达边界或当前棋格为空或当前位置为己方棋子，则更换方向继续
            continue
        while self.board[i][j] == -color:# 遇到对方棋子
            i, j = i + dx, j + dy
            if i < 0 or i >= 8 or j < 0 or j >= 8 or self.board[i][j] == 0:
```

```
                # 超出边界或者对方被围住的棋子不连续，则跳出当前循环
                break
            if i < 0 or i >= 8 or j < 0 or j >= 8 or self.board[i][j] != color:
                # 到达边界或者不是以己方棋子结束，则继续更换方向遍历
                continue
            return True
            # 如果在该方向上没有找到可以翻转的对手棋子，则继续检查下一个方向
        return False
```

- 落子函数`make_move`，大部分与`is_valid_move`函数类似，不同之处在于`is_valid_move`只需要找到一个有效方向即可返回true，而`make_move`函数需要完成八个方向的遍历并翻转相应的棋子。

```
def make_move(self, x, y, color):
    self.board[x][y] = color # (x,y)设置为当前棋子的颜色
    # 从八个方向遍历可翻转的棋子
    for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (1, -1), (-1,
1), (-1, -1)]:
        i, j = x + dx, y + dy
        #到达边界或当前棋格为空或当前位置为己方棋子，则更换方向继续
        if i < 0 or i >= 8 or j < 0 or j >= 8 or self.board[i][j] == 0 or
self.board[i][j] == color:
            continue
        record = [] # 记录对方连续的棋子的坐标
        while self.board[i][j] == -color:
            record.append((i, j)) # color为对方颜色，将坐标加入待定列表
            i, j = i + dx, j + dy
            if i < 0 or i >= 8 or j < 0 or j >= 8:# 超出边界停止遍历，否则while的
判断会报错
                break
        # 如果最后一个棋子超出边界或者不是己方棋子，则不进行翻转操作
        if i < 0 or i >= 8 or j < 0 or j >= 8 or self.board[i][j] != color:
            continue
        for (i, j) in record:
            self.board[i][j] = color
```

- 结束游戏`is_game_over`：黑棋与白棋均没有有效的落子位置，结束当前游戏

```
def is_game_over(self):
    for i in range(8):
        for j in range(8):
            if self.board[i][j] == 0 and (self.is_valid_move(i, j, BLACK) or
self.is_valid_move(i, j, WHITE)):
                return False
    return True
```

算法相关函数

评价函数`evaluate`，主要根据设定的棋子权重值以及黑白棋的数目、位置进行评价

```python
def evaluate(board, color):
    # 棋子权重(可根据需要进行修改，其中四个角落权重最大，因为无法被翻转)
    weight = [
        [150, -25, 25, 10, 10, 25, -25, 150],
        [-25, -50, -10, -10, -10, -10, -50, -25],
        [25, -10, 15, 5, 5, 15, -10, 25],
        [5, -5, 5, 0, 0, 5, -5, 5],
        [5, -5, 5, 0, 0, 5, -5, 5],
        [25, -10, 15, 5, 5, 15, -10, 25],
        [-25, -50, -10, -10, -10, -10, -50, -25],
        [150, -25, 20, 10, 10, 20, -25, 150],
    ]
    # 统计黑子和白子的数量和位置
    black_count, white_count = board.count_chess_pieces()
    black_position = []
    white_position = []
    for i in range(8):
        for j in range(8):
            if board[(i, j)] == BLACK:
                black_position.append((i, j))
            elif board[(i, j)] == WHITE:
                white_position.append((i, j))

    # 计算当前玩家的得分
    if color == BLACK:
        score = 100 * (black_count - white_count)
        for i, j in black_position:
            score += weight[i][j]
        for i, j in white_position:
            score -= weight[i][j]
    else:
        score = 100 * (white_count - black_count)
        for i, j in black_position:
            score -= weight[i][j]
        for i, j in white_position:
            score += weight[i][j]

    return score
```

$\alpha-\beta$剪枝函数`alphabeta`，返回最优落子和评价值

- 基本思路：使用递归来实现搜索，每一层都根据当前颜色进行递归，分别选择最大值或者最小值，并把搜索深度减1。当搜索深度为0或game over时，函数返回当前局面的估值。
- 使用了alpha-beta剪枝算法来减少搜索的分支数，提高搜索效率
- 对于黑棋，score越高越有利；对于白棋，score越低越有利

```python
def alphabeta(board, depth, alpha, beta, color):
    # 搜索深度为0或game over时，退出函数
```

```
    if depth == 0 or board.is_game_over():
        return None, evaluate(board, color)

    if color == BLACK:
        # 初始化最优落子位置和最大估值
        best_score = float('-inf')
        best_move = None
        # 遍历棋盘上所有的位置
        for i in range(8):
            for j in range(8):
                # 找到有效的落子位置
                if board.is_valid_move(i, j, BLACK):
                    # 深拷贝当前棋盘，并在新棋盘上完成落子
                    new_board = copy.deepcopy(board)
                    new_board.make_move(i, j, BLACK)
                    # 递归调用alphabeta函数，获取估值
                    _, score = alphabeta(new_board, depth - 1, alpha, beta,
WHITE)
                    # 如果估值比之前的最大估值更大，则更新最优落子位置和最大估值
                    if score > best_score:
                        best_score = score
                        best_move = (i, j)
                    # 更新alpha值
                    alpha = max(alpha, best_score)
                    # 如果beta值小于等于alpha值，则对当前节点进行剪枝
                    if beta <= alpha:
                        break
    else:
        # 初始化最优落子和最小估值
        best_score = float('inf')
        best_move = None
        # 遍历棋盘上所有的位置
        for i in range(8):
            for j in range(8):
                # 找到有效的落子位置
                if board.is_valid_move(i, j, WHITE):
                    # 深拷贝当前棋盘，并在新棋盘上完成落子
                    new_board = copy.deepcopy(board)
                    new_board.make_move(i, j, WHITE)
                    # 递归调用alphabeta函数，获取估值
                    _, score = alphabeta(new_board, depth - 1, alpha, beta,
BLACK)
                    # 如果估值比之前的最小估值更小，则更新最优落子位置和最小估值
                    if score < best_score:
                        best_score = score
                        best_move = (i, j)
                    # 更新beta值
                    beta = min(beta, best_score)
                    # 如果beta值小于等于alpha值，则对当前节点进行剪枝
```

```
                    if beta <= alpha:
                        break


    return best_move, best_score # 返回最优落子位置和当前局面的估值
```

主函数`main`的设计

- 玩家通过选择1或-1来完成颜色的选择（先手后手）
- 在棋盘上，用“X”表示黑棋，“O”表示白棋，“□”表示空位置
- 设置一个变量`times`用于判断当前棋子颜色，偶数为黑棋，奇数为白棋
- 游戏结束判断：`is_game_over`函数为真或者`white_score==-inf`或者`black_score==inf`
- 当黑棋/白棋没有有效落子位置时，自动轮到白棋/黑棋继续下棋
- 如果玩家输入的位置为无效落子位置，将提示重新输入，并且times不会产生变化

```
# 主函数:控制人机对战
def main():
    while True:
        print("Please choose your color: -1 for WHITE and 1 for BLACK")
        choice = int(input())
        if choice == -1 or choice == 1:
            break
    board = Board()
    print("X denotes for Black")
    print("O denotes for White")
    print("□ denotes for Empty")
    times = 0
    while not board.is_game_over():
        board.print_board()
        if times % 2 == 0:
            print("Black's turn")
            color = BLACK
        else:
            print("White's turn")
            color = WHITE
        # AI先手
        if choice == -1:
            if color == BLACK:
                if board.is_black_over():
                    print("Black does not have a suitable position, White
continues")
                    times = times + 1
                    continue
                depth = 4  # 设置搜索深度
                start = process_time()
                move, score = alphabeta(board, depth, float('-inf'),
float('inf'), color)
                end = process_time()
                print(f"Black plays ({move[0]}, {move[1]}) , SCORE is
{score}")
                print("Computer running time is ", end="")
```

```
                print(end - start, end="")
                print(" seconds")
                if board.is_valid_move(move[0], move[1], BLACK):
                    times = times + 1
                if score == inf:
                    board.make_move(move[0], move[1], BLACK)
                    board.print_board()
                    break
            else:
                if board.is_white_over():
                    print("White does not have a suitable position, Black
continues")
                    times = times + 1
                    continue
                x, y = input("Please enter your move, x y means (x,y):
").split()
                move = (int(x), int(y))
                white_score = evaluate(board, WHITE)
                if board.is_valid_move(move[0], move[1], WHITE):
                    print(f"White plays ({move[0]}, {move[1]}) , SCORE is
{white_score}")
                    times = times + 1
                if white_score == -inf:
                    board.make_move(move[0], move[1], WHITE)
                    board.print_board()
                    break
        # 玩家先手
        elif choice == 1:
            if color == WHITE:
                if board.is_white_over():
                    print("White does not have a suitable position, Black
continues")
                    times = times + 1
                    continue
                depth = 4  # 设置搜索深度
                start = process_time()
                move, score = alphabeta(board, depth, float('-inf'),
float('inf'), color)
                end = process_time()
                print(f"White plays ({move[0]}, {move[1]}), SCORE is
{score}")
                print("Computer running time is ", end="")
                print(end - start, end="")
                print(" seconds")
                if board.is_valid_move(move[0], move[1], WHITE):
                    times = times + 1
                if score == -inf:
                    board.make_move(move[0], move[1], WHITE)
                    board.print_board()
```

```
                    break
            else:
                if board.is_black_over():
                    print("Black does not have a suitable position, White
continues")
                    times = times + 1
                    continue
                x, y = input("Please enter your move, x y means (x,y):
").split()
                move = (int(x), int(y))
                black_score = evaluate(board, BLACK)
                if board.is_valid_move(move[0], move[1], BLACK):
                    print(f"Black plays ({move[0]}, {move[1]}), SCORE is
{black_score}")
                    times = times + 1
                if black_score == inf:
                    board.make_move(move[0], move[1], BLACK)
                    board.print_board()
                    break
        if not board.is_valid_move(move[0], move[1], color):
            print("INVALID MOVE, please try again.")
            continue
        board.make_move(move[0], move[1], color)

    board.print_board()
    black_count, white_count = board.count_chess_pieces()
    if black_count > white_count:
        print("Black wins")
    elif black_count < white_count:
        print("White wins")
    else:
        print("Tie")
```

三、实验结果及分析

1.实验结果展示示例

初始化及先手后手的选择,该示例选择先手（黑棋）

```
Please choose your color: -1 for WHITE and 1 for BLACK
1
X denotes for Black
O denotes for White
□ denotes for Empty
  0 1 2 3 4 5 6 7
0 □ □ □ □ □ □ □ □
1 □ □ □ □ □ □ □ □
2 □ □ □ □ □ □ □ □
3 □ □ □ X O □ □ □
4 □ □ □ O X □ □ □
5 □ □ □ □ □ □ □ □
6 □ □ □ □ □ □ □ □
7 □ □ □ □ □ □ □ □
Black's turn
```

回合1:

```
Black's turn
Please enter your move, x y means (x,y): 5 3
Black plays (5, 3), SCORE is 0
  0 1 2 3 4 5 6 7
0 □ □ □ □ □ □ □ □
1 □ □ □ □ □ □ □ □
2 □ □ □ □ □ □ □ □
3 □ □ □ X O □ □ □
4 □ □ □ X X □ □ □
5 □ □ □ X □ □ □ □
6 □ □ □ □ □ □ □ □
7 □ □ □ □ □ □ □ □
White's turn
White plays (5, 4), SCORE is -300
Computer running time is 0.021304 seconds
  0 1 2 3 4 5 6 7
0 □ □ □ □ □ □ □ □
1 □ □ □ □ □ □ □ □
2 □ □ □ □ □ □ □ □
3 □ □ □ X O □ □ □
4 □ □ □ X O □ □ □
5 □ □ □ X O □ □ □
6 □ □ □ □ □ □ □ □
7 □ □ □ □ □ □ □ □
```

回合2:

```
Black's turn
Please enter your move, x y means (x,y): 3 5
Black plays (3, 5), SCORE is 0
  0 1 2 3 4 5 6 7
0 □ □ □ □ □ □ □ □
1 □ □ □ □ □ □ □ □
2 □ □ □ □ □ □ □ □
3 □ □ □ X X X □ □
4 □ □ □ X X □ □ □
5 □ □ □ X O □ □ □
6 □ □ □ □ □ □ □ □
7 □ □ □ □ □ □ □ □
White's turn
White plays (5, 2), SCORE is -435
Computer running time is 0.029921000000000003 seconds
  0 1 2 3 4 5 6 7
0 □ □ □ □ □ □ □ □
1 □ □ □ □ □ □ □ □
2 □ □ □ □ □ □ □ □
3 □ □ □ X X X □ □
4 □ □ □ X X □ □ □
5 □ □ O O O □ □ □
6 □ □ □ □ □ □ □ □
7 □ □ □ □ □ □ □ □
```

回合3:

```
Black's turn
Please enter your move, x y means (x,y): 6 2
Black plays (6, 2), SCORE is 180
  0 1 2 3 4 5 6 7
0 □ □ □ □ □ □ □ □
1 □ □ □ □ □ □ □ □
2 □ □ □ □ □ □ □ □
3 □ □ □ X X X □ □
4 □ □ □ X X □ □ □
5 □ □ O X O □ □ □
6 □ □ X □ □ □ □ □
7 □ □ □ □ □ □ □ □
White's turn
White plays (3, 2), SCORE is -450
Computer running time is 0.057082999999999995 seconds
  0 1 2 3 4 5 6 7
0 □ □ □ □ □ □ □ □
1 □ □ □ □ □ □ □ □
2 □ □ □ □ □ □ □ □
3 □ □ O X X X □ □
4 □ □ □ O X □ □ □
5 □ □ O X O □ □ □
6 □ □ X □ □ □ □ □
7 □ □ □ □ □ □ □ □
```

在棋盘上，用“X”表示黑棋，“O”表示白棋，“□”表示空位置

对于黑棋，score越高越有利；对于白棋，score越低越有利

2.评测指标展示及分析

根据运行时间进行分析

在初始的三个回合中的性能表现不错，平均运行时间约为0.0361秒，可以在较短的时间内计算出下一步的最优算法

```
Computer running time is 0.021304 seconds
Computer running time is 0.029921000000000003 seconds
Computer running time is 0.057082999999999995 seconds
```

后续回合：

1. 棋盘状态的复杂度增加：随着游戏的进行，棋盘状态的复杂度增加，需要更长的时间来搜索和评估，从而导致运行时间增加
2. 搜索树的分支因子增加，使得整个搜索过程的时间也会相应增加

```
Computer running time is 0.138926 seconds
Computer running time is 0.17291100000000004 seconds
Computer running time is 0.236417 seconds
Computer running time is 0.21107600000000004 seconds
```

对于评价函数`evaluate`

这个评价函数是基于权重矩阵实现的，权重矩阵定义了每个位置对于棋局的重要程度，对于当前局面，会分别计算黑棋和白棋的数量和位置，然后根据权重矩阵计算出每个玩家的得分。

优点：简单易懂，容易实现和调整，可以通过调整权重矩阵的数值来改进评价函数的表现。

缺点：

没有考虑到落子顺序的影响：在同一棋局中，先手和后手的得分可能会有所不同

只考虑了当前局面的得分，而没有考虑到后续棋局可能发生的变化，可能会导致算法出现局部最优解

没有考虑一些局面特征，如连子、眼位等

在实际应用中，需要根据具体的需求和场景，优化评价函数，从而对算法进行优化。