E4_22336216

人工智能实验

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	DCS315	专业（方向）	计算机科学与技术（系统结构）
学号	22336216	姓名	陶宇卓

1 实验题目

实验四：用Alpha-beta剪枝算法实现五子棋的人机对弈

2 实验内容

一、算法原理

- **博弈树搜索**

博弈树搜索是一种用于制定决策的算法，通常用于博弈和决策问题。它的原理是通过模拟游戏的可能走法和对手的反应，构建一棵树形结构，然后根据这棵树找到最优的决策方案。

具体来说，博弈树搜索通常包括以下步骤：

- **建立博弈树**：从当前局面出发，通过递归地模拟游戏的各种可能走法和对手的反应，构建一棵博弈树。树的每个节点代表一个游戏状态，每条边代表一个合法的移动。
- **评估叶子节点**：当到达博弈树的叶子节点时，评估该节点的局面价值。这通常通过一个评估函数来进行，该函数根据当前局面的优劣给出一个分数。
- **反向传播分数**：将叶子节点的评估分数传播回父节点，根据当前节点是最大化还是最小化节点，选择最大或最小的分数作为父节点的分数。
- **选择最佳决策**：在根节点处选择最优的决策，通常选择具有最高分数的子节点对应的移动作为最佳决策。
- **剪枝优化**：为了减少搜索空间，通常会使用一些剪枝策略来减少不必要的搜索。这些策略包括alpha-beta剪枝、置换表、迭代加深搜索等。

通过这些步骤，博弈树搜索能够找到在当前游戏状态下的最优决策，从而指导玩家或程序做出更好的行动。

正如上面所说，为了减少不必要的搜索，优化算法的时间和空间效率，通常会采用剪枝策略，下面介绍**α-β剪枝**：

- **α-β剪枝**

1. **极小极大算法**：在博弈树搜索中，玩家和对手轮流行动，其中玩家希望最大化评估值，而对手希望最小化评估值。在极小极大算法中，每个节点都有一个值，表示在该节点上的最好结果。对于MAX节点，它的值是它的所有子节点中最大的值；对于MIN节点，它的值是它的所有子节点中最小的值。
2. **α-β剪枝**：在搜索树的构建过程中，α-β剪枝通过维护两个值α和β来剪去不必要的分支，从而减少搜索空间。α表示MAX节点目前已知的最佳值，β表示MIN节点目前已知的最佳值。
    a. 当一个MAX节点的值大于或等于β时，表示对手已经有了更好的选择，因此MAX节点的父节点不再需要考虑这个节点的其他子节点，可以直接剪枝。
    b. 当一个MIN节点的值小于或等于α时，表示玩家已经有了更好的选择，因此MIN节点的父节点不再需要考虑这个节点的其他子节点，可以直接剪枝。
3. **剪枝优化**：通过α-β剪枝，可以有效地减少搜索空间。在搜索树的构建过程中，当发现一个节点的值可以剪枝时，就可以直接停止对该节点的搜索，从而减少计算量。

- **评估函数**

评估函数（Evaluation Function）是在博弈算法中用于评估当前棋局优劣的函数。它对当前局面进行评分，以帮助博弈算法在搜索过程中选择最优的走法。

评估函数的设计通常基于对棋局特征的分析和理解，以及对胜利条件的考虑。在象棋、围棋、五子棋等棋类游戏中，评估函数可能考虑以下几个方面：

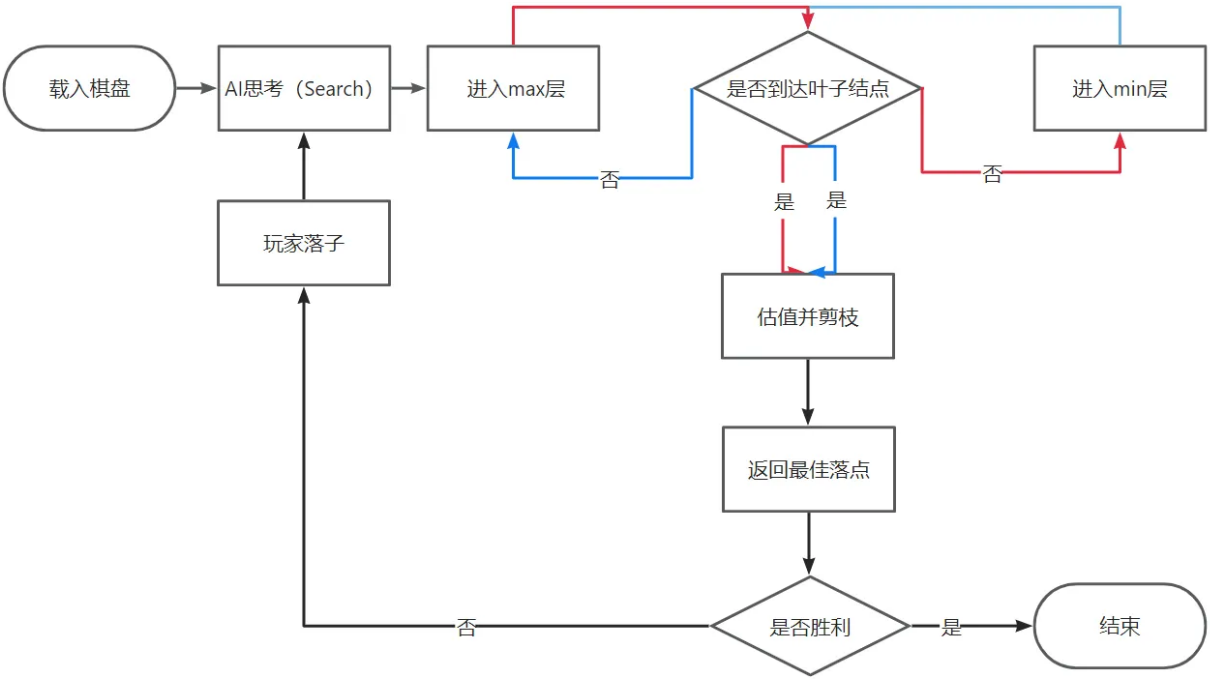
- **棋子的位置**：不同位置的棋子可能具有不同的价值。例如，在围棋中，边角的位置往往比较重要，因为它们可以控制更多的空间。在五子棋中，中心位置通常更有利于形成连珠。
- **棋子的连续性**：形成连续的棋子（如棋子的连珠）可能会增加棋局的价值。因此，评估函数可能会考虑棋局中连续的棋子数量和形成的棋型。
- **棋子的控制力**：控制了更多重要位置的玩家可能更有优势。因此，评估函数可能会考虑棋局中每个玩家所控制的位置数量和重要性。
- **进攻和防守**：评估函数应该平衡进攻和防守的需求。它应该考虑到当前局势下，玩家是处于进攻还是防守的状态，并相应地调整评分。
- **局面的稳定性**：稳定的局面可能更有利于玩家。评估函数可能会考虑局面中的不稳定因素，如可能被对手反击的位置。
- **游戏阶段**：不同的游戏阶段可能需要不同的评估方式。在开局阶段，可能更重视控制中心；而在结束阶段，可能更注重形成连珠或阻止对手的连珠。

评估函数的设计需要根据具体的游戏规则和博弈算法来进行调整和优化。一个好的评估函数可以大大提高博弈算法的性能和准确度。

- **在本次实验中的具体细节**

- **建立博弈树**：首先，根据当前棋局状态建立一颗博弈树。树的每个节点代表一个可能的棋局状态，包括所有可能的玩家走法和对手的应对。
- **评估局面**：对于博弈树的叶子节点（即终局状态），使用评估函数对局面进行评估，得出当前局面的分数。评估函数可以根据棋子的位置、连珠情况、控制力等因素来评估局面的优劣。
- **Alpha-beta剪枝**：在搜索博弈树的过程中，使用Alpha-beta剪枝进行剪枝，以减少搜索的节点数量。具体来说，Alpha-beta剪枝通过维护两个值：Alpha和Beta。Alpha代表当前节点能够保证的最低分数，Beta代表当前节点能够保证的最高分数。
    - 对于**MAX节点**（玩家的节点），如果某个子节点的评估分数大于等于Beta，则该节点不会被搜索，因为对手可以选择不走这个节点，而选择更好的节点。因此，当前节点的搜索可以结束，返回评估分数。
    - 对于**MIN节点**（对手的节点），如果某个子节点的评估分数小于等于Alpha，则该节点不会被搜索，因为玩家可以选择不走这个节点，而选择更差的节点。因此，当前节点的搜索可以结束，返回评估分数。
    - 在搜索过程中，如果发现某个节点的Alpha值大于等于Beta值，说明当前节点及其所有兄弟节点都不会被搜索，搜索可以提前结束。
- **递归搜索**：从根节点开始，通过递归搜索博弈树的每一层，直到达到指定的搜索深度或者达到终局状态。在搜索的过程中，根据当前节点的角色（MAX或MIN）选择相应的走法，更新Alpha和Beta值，并根据剪枝条件进行剪枝。
- **选择最优走法并落子**：当搜索到达根节点时，根据每个子节点的评估分数选择最优的走法作为计算机的下一步落子。

- **流程图**


载入棋盘
AI思考（Search）
进入max层
是否到达叶子结点
进入min层
否
否
是
是
玩家落子
估值并剪枝
返回最佳落点
否
是否胜利
是
结束

二、伪代码

伪代码 Plain Text

```
函数 AlphaBetaSearch(棋盘, 空格, 黑子, 白子, 是否黑子, 深度=3):
    如果深度为0或者棋盘上出现某种胜利棋型:
        返回评估分数, None
    如果当前是黑子轮次:
        调用 MaxValue 函数
    否则:
        调用 MinValue 函数

函数 MaxValue(棋盘, alpha, beta, 深度, 计数器):
    如果深度为0或者棋盘上出现某种胜利棋型:
        返回评估分数, None
    初始化 value 为负无穷, move 为 None
    对于棋盘上每个可行位置:
        获取该位置的下一步棋盘状态
        递归调用 MinValue 函数
        如果返回的评估值大于当前 value:
            更新 value 和 move
        如果 value 大于等于 beta:
            剪枝
            返回 value, move
        更新 alpha
    返回 value, move

函数 MinValue(棋盘, alpha, beta, 深度, 计数器):
    如果深度为0或者棋盘上出现某种胜利棋型:
        返回评估分数, None
    初始化 value 为正无穷, move 为 None
    对于棋盘上每个可行位置:
        获取该位置的下一步棋盘状态
        递归调用 MaxValue 函数
        如果返回的评估值小于当前 value:
            更新 value 和 move
        如果 value 小于等于 alpha:
            剪枝
            返回 value, move
        更新 beta
    返回 value, move

函数 evaluate(棋盘, 是否黑子):
    初始化分数为 0
    对于棋盘上的每个位置:
        对于四个方向:
            获取当前位置的线段
```

```
                对于每种棋型:
                    如果线段匹配棋型:
                        根据棋型给位置加分
                    否则:
                        增加一点分数
                    如果反转棋型匹配线段:
                        根据棋型给位置减分
                    否则:
                        减少一点分数
    返回总分数

函数 get_successors(棋盘, 颜色, 优先级函数, 空格=-1):
    初始化下一个棋盘为当前棋盘的副本
    获取所有空格的坐标并按照优先级排序
    对于每个优先级较高的空格坐标:
        将当前颜色的棋子放在这个空格上
        生成器返回该后继状态的坐标和对应的棋盘
        恢复空格状态

函数 get_pattern_locations(棋盘, 棋子排列):
    初始化空列表用于存放位置
    对于棋盘上每个位置:
        如果该位置是棋子排列的起始位置:
            对于所有可能的方向:
                如果棋盘上存在匹配的棋子排列:
                    将匹配的位置信息添加到列表中
    返回位置列表

函数 count_pattern(棋盘, 棋子排列):
    返回棋子排列在棋盘上的个数

函数 is_win(棋盘, 颜色, 空格=-1):
    定义四种可能的胜利棋型
    如果棋盘上存在其中一种胜利棋型:
        返回 True
    否则:
        返回 False

函数 get_line(棋盘, 起始位置x, 起始位置y, 水平方向增量dx, 垂直方向增量dy, 长度=11):
    初始化空字符串用于存放线段
    对于给定长度的线段:
        如果当前位置在棋盘范围内且不为空:
            将当前位置的棋子加入线段字符串
        否则:

```

```
                添加空位标记到线段字符串
        返回线段字符串
```

三、关键代码展示

> 完整代码见../code

- **棋型评估函数及数据结构**

> 关于为什么没用现成的count_pattern函数：单纯是因为一开始没看见😖

  - **position_value**: 这是一个二维列表，表示棋盘上每个位置的价值。这个列表用于评估每个位置的重要性，根据位置的不同给予不同的分数。
  - **SCORES**: 这是一个字典，包含不同棋型的分数。根据棋型的不同，给予不同的分数。比如连五的分数是很高的，而眠三的分数相对较低。
  - **PATTERNS**: 这是一个字典，包含了不同棋型的模式。每种棋型都有对应的匹配模式，用于在棋盘上查找这种棋型的出现情况。
  - **print_board**: 这个函数用于打印棋盘的状态，方便调试和可视化。
  - **evaluate 函数:**这个函数用于评估棋盘的状态，返回一个分数。遍历棋盘上的每个位置，检查每个位置上下左右四个方向的棋型，根据匹配的棋型给予相应的分数。除了棋型分数之外，还考虑了位置价值，即每个位置的重要性。最终将棋型分数和位置价值相加，得到最终的评分。
  - **get_line 函数**:这个函数用于获取棋盘上的一条线段，包括水平、垂直和对角线方向上的线段。给定起始位置 (i, j) 和方向 (dx, dy)，沿着这个方向获取指定长度的线段。如果超出了棋盘范围，则用数字 2 表示超出的部分，以便后续棋型匹配时处理。**为了避免把空格（–1）匹配为1，我在get_line的时候将–1转换为2。**

evaluate部分 Python

```python
position_value = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
    [0, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 0],
    [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 1, 0],
    [0, 1, 2, 3, 4, 4, 4, 4, 4, 4, 4, 3, 2, 1, 0],
    [0, 1, 2, 3, 4, 5, 5, 5, 5, 5, 4, 3, 2, 1, 0],
    [0, 1, 2, 3, 4, 5, 6, 6, 6, 5, 4, 3, 2, 1, 0],
    [0, 1, 2, 3, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1, 0],
    [0, 1, 2, 3, 4, 5, 6, 6, 6, 5, 4, 3, 2, 1, 0],
    [0, 1, 2, 3, 4, 5, 5, 5, 5, 5, 4, 3, 2, 1, 0],
    [0, 1, 2, 3, 4, 4, 4, 4, 4, 4, 4, 3, 2, 1, 0],
    [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 1, 0],
    [0, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 0],
    [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
]

SCORES = {
    "FIVE": 999999999,  # 连五
    "FOUR": 66660000,  # 活四
    "SFOUR": 6660000,  # 冲四
    "THREE": 625000,  # 活三
    "STHREE": 62500,  # 眠三
}

# 定义棋型的模式
PATTERNS = {
    "FIVE": ["11111",
    ],  # 连五
    "FOUR": [
        "211112",  # 活四
    ],
    "SFOUR": [
        "011112",  # 冲四
        "211110",  # 冲四
        "011121",  # 冲四
        "121110",  # 冲四
        "11211",  # 冲四
    ],
    "THREE": [
        "21112",  # 活三
        "211212",  # 活三
```

```
    ],
    "STHREE": [
        "01112",  # 眠三
        "21110",  # 眠三
        "011212",  # 眠三
        "212110",  # 眠三
        "012112",  # 眠三
        "211210",  # 眠三
        "11221",  # 眠三
        "12211",  # 眠三
        "12121",  # 眠三
    ],
}


def print_board(board):
    for row in board:
        for cell in row:
            if cell == -1:
                print('.', end=' ')
            elif cell == 1:
                print('B', end=' ')
            elif cell == 0:
                print('W', end=' ')
        print()


def evaluate(board, isblack):
    # 这个函数用于评估棋盘的状态，返回一个分数
    # board 是当前棋盘的状态，isblack 是一个布尔值，如果为 True 则代表当前轮到黑子，否则轮到白子
    # 定义棋型的分数
    # 初始化分数
    score = 0

    # 遍历棋盘
    for i in range(len(board)):
        for j in range(len(board[0])):
            # 检查四个方向
            for dx, dy in [(0, 1), (1, 0), (1, 1), (1, -1)]:
                if board[i][j] != -1:
                # 获取线段
                    line = get_line(board, i, j, dx, dy)
                    # 检查每种棋型
                    for pattern, values in PATTERNS.items():

```

```
                        for value in values:
                            # 将line和value都转换为字符串
                            # 使用re库的search函数来查找value在line中的位置
                            if re.search(value, line):
                                score += SCORES[pattern] * 1
                                # if str_value == '211112':
                                #     score += 90000000
                            else:
                                score += 8  # 如果没有匹配到，增加8分
                            # 反转棋型，用于检查另一种颜色的棋子
                            value = value.replace("1", "t").replace("0", "1").replace("t", "0")
                            if re.search(value, line):
                                score += SCORES[pattern] * -1
                            else:
                                score -= 8  # 如果没有匹配到，增加8分
                    # 加上位置价值

                    score += position_value[i][j] * (1 if board[i][j] == 1 else -1)
        # print_board(board)
        # print(f"当前局面评分: {score}")
        return score


    def get_line(board, i, j, dx, dy, length=11):
        # 这个函数用于获取棋盘上的一条线段
        # board 是当前棋盘的状态，i 和 j 是线段的起始位置，dx 和 dy 是线段的方向，length 是线段的长度
        line = ''
        for k in range(-5, 6):
            if 0 <= i + k * dx < len(board) and 0 <= j + k * dy < len(board[0]) and board[i + k * dx][j + k * dy] != -1:
                line += str(board[i + k * dx][j + k * dy])
            else:
                line += '2'
        return line
```

- **α-β剪枝**
  - **AlphaBetaSearch 函数定义**：参数包括棋盘 board、空格、黑子、白子的表示 EMPTY、BLACK、WHITE，当前是否轮到黑子 isblack，以及搜索深度 depth。这个函数**内部**定义了两个子函数 max_value 和 min_value，分别代表在搜索树中的 MAX 节点和 MIN 节点进行搜索。
  - **max_value 函数**：用于在搜索树的 MAX 节点进行搜索。如果达到指定深度或者存在特定棋型

（用于终止搜索），则返回当前棋盘的评估值。否则，遍历当前棋盘的所有后继状态，调用 min_value 函数，更新 alpha 值并进行 alpha 剪枝。

- **min_value 函数**：用于在搜索树的 MIN 节点进行搜索。类似于 max_value 函数，遍历后继状态，调用 max_value 函数，更新 beta 值并进行 beta 剪枝。
- **迭代搜索**：在 AlphaBetaSearch 函数中，通过调用 max_value 函数开始搜索，最终返回最佳的落子位置和对应的 alpha 值。
- **计数器**：在搜索过程中，通过一个字典 counts 记录搜索的节点数和剪枝的次数，用于性能分析和调试。

α–β Python

```python
def AlphaBetaSearch(board, EMPTY, BLACK, WHITE, isblack, depth=3):
    # 这个函数使用 alpha-beta 剪枝的方法搜索棋盘，返回最佳的落子位置
    # board 是当前棋盘的状态，EMPTY, BLACK, WHITE 分别代表空格、黑子和白子
    # isblack 是一个布尔值，如果为 True 则代表当前轮到黑子，否则轮到白子
    # depth 是搜索的深度

    # 定义 max_value 和 min_value 函数，分别用于在搜索树中的 MAX 节点和 MIN 节点进行搜索
    # 定义 counts 字典，用于记录搜索的节点数和剪枝的次数
    # 最后返回最佳的落子位置和对应的 alpha 值
    def max_value(board, alpha, beta, depth, counts):
        pattw1 = (0, 0, 0, 0, 0)
        pattw2 = (-1, 0, 0, 0, 0, -1)
        if depth == 0 or count_pattern(board, pattw1) + count_pattern(board, pattw2) > 0:
            return evaluate(board, isblack), None # 到达叶节点，返回评估值
        value = float('-inf')
        move = None
        priority_func = partial(_coordinate_priority, board=board) # 优先级函数
        for x, y, next_board in get_successors(board, BLACK if isblack else WHITE, priority=priority_func):
            counts['searchcount'] += 1
            next_value, _ = min_value(next_board, alpha, beta, depth - 1, counts)
            if next_value > value: # 更新最大值
                value, move = next_value, (x, y)
            if value >= beta: # 剪枝(alpha 剪枝)
                counts['pruncount'] += 1
                # print_board(next_board)
                # print(f"score: {value}")
                # print(f"{counts['pruncount']}pruning: alpha: {value}, beta: {beta}")
                return value, move
            # if value > alpha:
            #     print_board(next_board)
            #     print(f"Update alpha: {value}")
            alpha = max(alpha, value)
            # print(f"alpha: {alpha}, beta: {beta}")
        return value, move

    def min_value(board, alpha, beta, depth, counts):
```

```
        pattb1 = (1, 1, 1, 1, 1)
        pattb2 = (-1, 1, 1, 1, 1, -1)
        if depth == 0 or count_pattern(board, pattb1) > 0 or (count_pattern(board, pattb2) > 0 and not is_win(board, WHITE, EMPTY)):
            return evaluate(board, isblack), None # 到达叶节点，返回评估值
        value = float('inf')
        move = None
        priority_func = partial(_coordinate_priority, board=board) # 优先级函数
        for x, y, next_board in get_successors(board, WHITE if isblack else BLACK, priority=priority_func):
            counts['searchcount'] += 1
            next_value, _ = max_value(next_board, alpha, beta, depth - 1, counts)
            if next_value < value: # 更新最小值
                value, move = next_value, (x, y)
            if value <= alpha: # 剪枝(beta 剪枝)
                counts['pruncount'] += 1
                # print_board(next_board)
                # print(f"score: {value}")
                # print(f"{counts['pruncount']}pruning: alpha: {alpha}, beta: {value}")
                return value, move
            # if value < beta:
            #     print_board(next_board)
            #     print(f"Update beta: {value}")
            beta = min(beta, value)
            # print(f"alpha: {alpha}, beta: {beta}")
        return value, move

    counts = {'pruncount': 0, 'searchcount': 0}
    alpha, move = max_value(board, float('-inf'), float('inf'), depth, counts)
    print(f"搜索次数: {counts['searchcount']}, 剪枝次数: {counts['pruncount']}")
    return move[0], move[1], alpha

```

- **其它代码**

other Python

```python
def _coordinate_priority(coordinate, board, EMPTY=-1):
    x, y = coordinate[0], coordinate[1]
    ROWS = 15
    occupied_coordinates = np.array([(i, j) for i in range(ROWS) for j in range(ROWS) if board[i][j] != EMPTY])
    if len(occupied_coordinates) == 0:
        return ((x - 7) ** 2 + (y - 7) ** 2) ** 0.5
    else:
        distances = np.sqrt(np.sum((occupied_coordinates - np.array([x, y])) ** 2, axis=1))
        min_distance = np.min(distances)
        return min_distance


def get_successors(board, color, priority, EMPTY=-1):
    # 这个函数用于获取当前状态的所有后继状态
    # board 是当前棋盘的状态，color 是当前轮到的颜色，priority 是一个函数，用于确定落子位置的优先级，EMPTY 代表空格
    '''
    返回当前状态的所有后继（默认按坐标顺序从左往右，从上往下）
    ----------------参数----------------
    board       当前的局面，是 15×15 的二维 list，表示棋盘
    color       当前轮到的颜色
    EMPTY       空格在 board 中的表示，默认为 -1
    priority    判断落子坐标优先级的函数（结果为小的优先）
    ----------------返回----------------
    一个生成器，每次迭代返回一个的后继状态 (x, y, next_board)
        x           落子的 x 坐标（行数/第一维）
        y           落子的 y 坐标（列数/第二维）
        next_board  后继棋盘
    '''
    # 注意：生成器返回的所有 next_board 是同一个 list!
    from copy import deepcopy
    next_board = deepcopy(board)
    ROWS = len(board)
    idx_list = [(x, y) for x in range(15) for y in range(15) if board[x][y] == EMPTY]
    idx_list.sort(key=priority)
    # idx_list = idx_list[:49]
    idx_list = [idx for idx in idx_list if 0 < priority(idx) <= 2]
    # print(idx_list)
    for x, y in idx_list:
        next_board[x][y] = color
```

```
            yield (x, y, next_board)
            next_board[x][y] = EMPTY


def get_pattern_locations(board, pattern):
    '''
    获取给定的棋子排列所在的位置
    ---------------参数---------------
    board        当前的局面，是 15×15 的二维 list，表示棋盘
    pattern      代表需要找的排列的 tuple
    ---------------返回---------------
    一个由 tuple 组成的 list，每个 tuple 代表在棋盘中找到的一个棋子排列
        tuple 的第 0 维      棋子排列的初始 x 坐标（行数/第一维）
        tuple 的第 1 维      棋子排列的初始 y 坐标（列数/第二维）
        tuple 的第 2 维      棋子排列的方向，0 为向下，1 为向右，2 为右下，
3 为左下；
                              仅对不对称排列：4 为向上，5 为向左，6 为左上，
7 为右上；
                              仅对长度为 1 的排列：方向默认为 0
    ---------------示例---------------
    对于以下的 board（W 为白子，B为黑子）
      0 y 1   2   3   4   ...
    0 +---W---+---+---+-- ...
    x |   |   |   |   |   ...
    1 +---+---B---+---+-- ...
      |   |   |   |   |   ...
    2 +---+---+---W---+-- ...
      |   |   |   |   |   ...
    3 +---+---+---+---+-- ...
      |   |   |   |   |   ...
    ...
    和要找的 pattern (WHITE, BLACK, WHITE)：
    函数输出的 list 会包含 (0, 1, 2) 这一元组，代表在 (0, 1) 的向右下方向找
到了
    一个对应 pattern 的棋子排列。
    '''
    ROWS = len(board)
    DIRE = [(1, 0), (0, 1), (1, 1), (1, -1)]
    pattern_list = []
    palindrome = True if tuple(reversed(pattern)) == pattern else Fa
lse
    for x in range(ROWS):
        for y in range(ROWS):
            if pattern[0] == board[x][y]:
                if len(pattern) == 1:

```

```
                    pattern_list.append((x, y, 0))
                else:
                    for dire_flag, dire in enumerate(DIRE):
                        if _check_pattern(board, ROWS, x, y, pattern, dire[0], dire[1]):
                            pattern_list.append((x, y, dire_flag))
                    if not palindrome:
                        for dire_flag, dire in enumerate(DIRE):
                            if _check_pattern(board, ROWS, x, y, pattern, -dire[0], -dire[1]):
                                pattern_list.append((x, y, dire_flag + 4))
    return pattern_list


# get_pattern_locations 调用的函数
def _check_pattern(board, ROWS, x, y, pattern, dx, dy):
    for goal in pattern[1:]:
        x, y = x + dx, y + dy
        if x < 0 or y < 0 or x >= ROWS or y >= ROWS or board[x][y] != goal:
            return False
    return True


def count_pattern(board, pattern):
    # 获取给定的棋子排列的个数
    return len(get_pattern_locations(board, pattern))


def is_win(board, color, EMPTY=-1):
    # 检查在当前 board 中 color 是否胜利
    pattern1 = (color, color, color, color, color)
    pattern2 = (EMPTY, color, color, color, color)
    pattern3 = (color, color, color, EMPTY, color)
    pattern4 = (color, color, EMPTY, color, color)
    return count_pattern(board, pattern1) + count_pattern(board, pattern2) + count_pattern(board, pattern3) + count_pattern(board, pattern4)> 0
```

四、创新点&优化

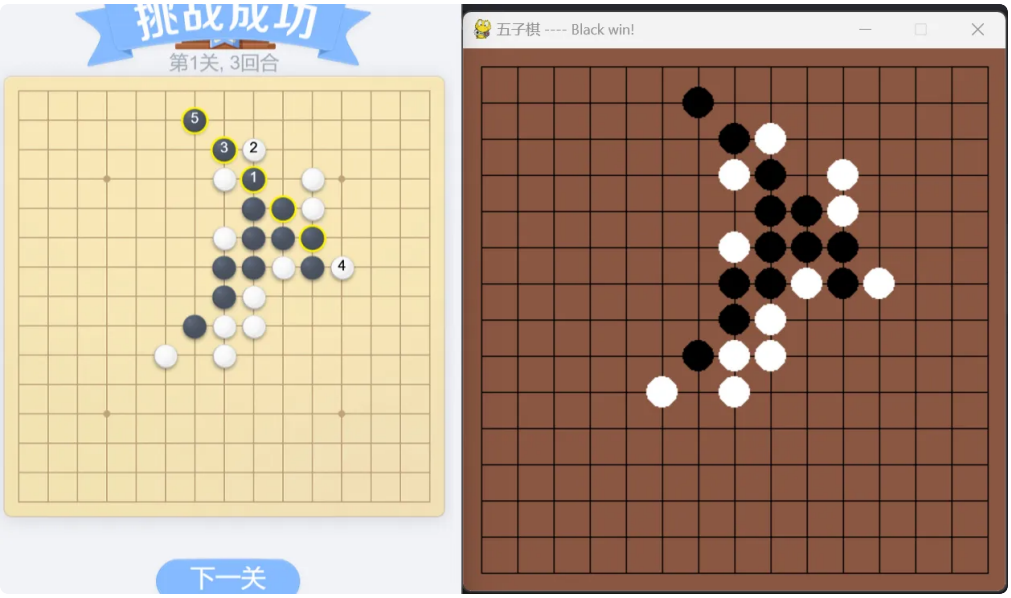
- **在获取后继结点时限制结点个数**：在优先级函数_coordinate_priority和获取后继状态的get_successors函数中，我限制了落子位置必须在已落子位置周围欧式距离两格内，这样的动态限制能应付绝大多数棋局，并且可以有效减少搜索空间。

- **使用numpy库计算欧氏距离**：在 _coordinate_priority 函数中，我们可以使用 numpy 库的向量化操作来计算所有已有棋子的最小欧氏距离。这样可以提高计算效率，因为 numpy 的向量化操作是在 C 语言级别上执行的，比 Python 的循环要快很多。

3 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）

- 实验中我测试了20个残局内AI的处理情况，结果表明**全部通关**。以第一关为例，截图如下（全部截图见../pics）：



- 更改α-β剪枝的代码，深度设置为**2**，并且在每次更新α/β值时打印当前局面及对应的值，在剪枝时同样如此。以**第一关第一步**为例，可见alpha==beta时提示剪枝。输出如下：

```python
if value >= beta: # 剪枝(alpha 剪枝)
    counts['pruncount'] += 1
    print_board(next_board)
    print(f"score: {value}")
    print(f"{counts['pruncount']}pruning: alpha: {value}, beta: {beta
}")
    return value, move
if value > alpha:
    print_board(next_board)
    print(f"Update alpha: {value}")
alpha = max(alpha, value)
print(f"alpha: {alpha}, beta: {beta}")
```

Plain Text

```
. . . . . . . . . . . . . . . . . .
. . . . . . . W . . . . . . . . . .
. . . . . . . B . . . . . . . . . .
. . . . . . . W . . W . . . . . . .
. . . . . . . . B B W . . . . . . .
. . . . . . . W B B B . . . . . . .
. . . . . . . B B W B . . . . . . .
. . . . . . . B W . . . . . . . . .
. . . . . . B W W . . . . . . . . .
. . . . . W . W . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
Update beta: 812412
. . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
. . . . . . . B . . W . . . . . . .
. . . . . . . W . . W . . . . . . .
. . . . . . . . B B W . . . . . . .
. . . . . . . W B B B . . . . . . .
. . . . . . . B B W B . . . . . . .
. . . . . . . B W . . . . . . . . .
. . . . . . B W W . . . . . . . . .
. . . . . W . W . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
Update beta: 499948
. . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
. . . . . . . B . . . . . . . . . .
. . . . . . . W W . W . . . . . . .
. . . . . . . . B B W . . . . . . .
. . . . . . . W B B B . . . . . . .
. . . . . . . B B W B . . . . . . .
. . . . . . . B W . . . . . . . . .
. . . . . . B W W . . . . . . . . .
. . . . . W . W . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
```

```
. . . . . .
. . . . . .
. . . . . .
. . . . . .
Update beta: -1375032
. . . . . .
. . . . . .
. . . . . B . . . . .
. . . . . W . . W . . .
. . . . . B B W . . .
. . . . . W B B B . . .
. . . . . B B W B . . .
. . . . . B W . . . . .
. . . . . B W W . . . . .
. . . . . W W W . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
Update beta: -34112572
. . . . . .
. . . . . .
. . . . . B . . . . .
. . . . . W . . W . . .
. . . . . B B W . . .
. . . . . W B B B . . .
. . . . . B B W B . . .
. . . . . B W . . . . .
. . . . . B W W . . . . .
. . . . . W . W . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
Update alpha: -34112572
. . . . . .
. . . . . W . . .
. . . . . B . . .
. . . . . W . . W . . .
. . . . . B B W . . .
. . . . . W B B B . . .
. . . . . B B W B . . .
. . . . . B W . . . . .
```

```
. . . . . B W W . . . . . .
. . . . . W . W . . . . . .
. . . . . . . . . . . . . .
. . . . . . . . . . . . . .
. . . . . . . . . . . . . .
. . . . . . . . . . . . . .
. . . . . . . . . . . . . .
Update beta: 812412
. . . . . . . . . . . . . .
. . . . . . . . . . . . . .
. . . . . . . W . . B . . . .
. . . . . . . W . . W . . . .
. . . . . . . . B B W . . . .
. . . . . . . W B B B . . . .
. . . . . . . B B W B . . . .
. . . . . . . B W . . . . . .
. . . . . B W W . . . . . .
. . . . . W . W . . . . . .
. . . . . . . . . . . . . .
. . . . . . . . . . . . . .
. . . . . . . . . . . . . .
. . . . . . . . . . . . . .
. . . . . . . . . . . . . .
Update beta: 562440
. . . . . . . . . . . . . .
. . . . . . . . . . . . . .
. . . . . . . . . . . B . . . .
. . . . . . . W W . W . . . .
. . . . . . . . B B W . . . .
. . . . . . . W B B B . . . .
. . . . . . . B B W B . . . .
. . . . . . . B W . . . . . .
. . . . . B W W . . . . . .
. . . . . W . W . . . . . .
. . . . . . . . . . . . . .
. . . . . . . . . . . . . .
. . . . . . . . . . . . . .
. . . . . . . . . . . . . .
. . . . . . . . . . . . . .
Update beta: -1375032
. . . . . . . . . . . . . .
. . . . . . . . . . . . . .
. . . . . . . . . . . B . . . .
. . . . . . . W . . W . . . .
. . . . . . . . B B W . . . .
```

```
. . . . . W B B B . . . .
. . . . . B B W B . . . .
. . . . . B W . . . . . .
. . . . . B W W . . . . . .
. . . . . W W W . . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
score: -34112572
1pruning: alpha: -34112572, beta: -34112572
```

- 平时输出内容为**x y alpha 搜索次数 剪枝次数 AI耗时**，所有关卡输出请见../output

```
搜索次数: 3901, 剪枝次数: 157
3 8 271738397
AI cost time:  10.226454257965088 s
搜索次数: 4614, 剪枝次数: 172
2 7 4992023920
AI cost time:  13.005861043930054 s
搜索次数: 9613, 剪枝次数: 438
6 11 6992023906
AI cost time:  29.21636176109314 s
```

2. 评测指标展示及分析

根据输出，深度为3的棋局内，AI每步落子时间为**5-180s**，具体时间取决于棋盘内棋子的布局以及数量。所有棋局也都在8回合内结束，所以这个时间还是可以接受的。

这个程序涉及到两个主要功能：Alpha-Beta剪枝搜索和评估棋局状态。我们逐一分析它们的时间和空间复杂度。

Alpha-Beta剪枝搜索

- **时间复杂度**：在每一层搜索中，最多考虑了每个空位的可能性，因此时间复杂度取决于搜索的深度。如果搜索深度为 d，每个位置平均有 m 个后继状态，那么时间复杂度为 O(m^d)。在这个程序中，搜索的深度默认为 3，因此时间复杂度大致在 O(m^3) 的数量级。
- **空间复杂度**：空间复杂度取决于递归调用的栈空间，以及每一层搜索中的状态存储。在递归调用中，栈的深度最多为搜索的深度，因此空间复杂度也与搜索深度相关。在每一层搜索中，需要存储当前状态的副本，以及一些局部变量，因此空间复杂度与棋盘的大小和搜索的深度相关。

评估棋局状态

- **时间复杂度**：评估函数的时间复杂度主要来自于对棋盘上每个位置的遍历，以及检查每个位置上下左右四个方向的棋型。对于每个位置，需要遍历四个方向，并且对于每个方向，需要检查是否匹配到了某种棋型。因此，时间复杂度与棋盘大小成正比，即为 O(n^2)，其中 n 为棋盘的边长。
- **空间复杂度**：评估函数的空间复杂度主要来自于局部变量的存储，以及棋型模式的存储。局部变量的存储空间与棋盘的大小和评估过程中使用的临时数据相关。棋型模式的存储空间取决于棋型的数量和模式的长度。

综合来看，这个程序的时间复杂度主要取决于搜索的深度和棋盘大小，而空间复杂度主要取决于棋盘的大小和评估过程中使用的临时存储空间。

4 思考题

无

5 参考资料

GitHub – colingogogo/gobang_AI: 基于博弈树α–β剪枝搜索的五子棋AI

> 值得一提的是，这个开源的代码在打第14关的时候似乎没有过（doge）

基于Python的博弈树搜索人机交互五子棋人工智能实验_python五子棋ai–CSDN博客

alpha_beta博弈树五子棋代码整理_基于alpha–beta搜索算法实现计算机博弈的代码–CSDN博客