

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 | 计科 2 班 | 专业(方向) | 计算机科学与技术 |
| 学号 | 21311274 | 姓名 | 林宇浩 |

一、 实验题目

博弈树搜索 Alpha-beta 剪枝实现五子棋残局求解

二、 实验内容

1. 算法原理

博弈树搜索是一种用于解决游戏策略问题的算法，其中博弈树表示了游戏中可能的决策和对应的结果。Alpha-beta剪枝算法是一种用于减少搜索空间的优化技术。Alpha-beta剪枝算法基于以下观察：在搜索树的某些分支中，存在着不必要继续探索的情况。例如，在一个最小化节点中，如果一个已经被搜索的子节点的值比当前节点的最优值更高，那么该最小化节点的其他子节点将不会影响到该节点的价值，因为最终的最小值已经超过了当前节点的最优值。因此，可以跳过该节点的所有未搜索子节点，从而减少搜索的时间。具体而言，Alpha-beta剪枝算法通过维护两个值alpha和beta，分别表示最大值和最小值节点的当前最优值。在搜索树的遍历过程中，当遇到一个最小化节点时，算法会检查该节点是否会影响其父节点的最优值。如果该节点的值小于或等于当前的最小值beta，则可以剪枝，即停止搜索该节点的剩余子节点，并返回到该节点的父节点。同样地，当遇到一个最大化节点时，算法会检查该节点是否会影响其父节点的最优值。如果该节点的值大于或等于当前的最大值alpha，则可以剪枝，即停止搜索该节点的剩余子节点，并返回到该节点的父节点。通过这种方式，Alpha-beta剪枝算法可以剪掉大部分不必要的搜索，从而大大减少搜索的时间，提高搜索效率。

2. 伪代码

```
def AlphaBeta(n, Player, alpha, beta):
    # 如果 n 是终止状态，则返回 n 的效用值
    if n is TERMINAL:
        return V(n)
    # 获取所有可能的后继状态
    ChildList = n.Successors(Player)
    if Player == MAX:
        bestValue = -infinity
        for child in ChildList:
            value = AlphaBeta(child, MIN, alpha, beta)
            if value > bestValue:
                bestValue = value
            if beta is not None and bestValue >= beta:
                break
        return bestValue
    else:
        bestValue = infinity
        for child in ChildList:
            value = AlphaBeta(child, MAX, alpha, beta)
            if value < bestValue:
                bestValue = value
            if alpha is not None and bestValue <= alpha:
                break
        return bestValue
```

```

# 如果当前玩家是 MAX 玩家
for c in ChildList:
    # 递归计算后继状态的效用值
    alpha = max(alpha, AlphaBeta(c, MIN, alpha, beta))
    if beta <= alpha:
        # beta 剪枝
        break
    return alpha

else:
    # 如果当前玩家是 MIN 玩家
    for c in ChildList:
        # 递归计算后继状态的效用值
        beta = min(beta, AlphaBeta(c, MAX, alpha, beta))
        if beta <= alpha:
            # alpha 剪枝
            break
    return beta

```

3. 关键代码展示（带注释）

```

import copy

def exist_lian_5(lines): # 存在连五
    color = lines[0][3]
    for i in range(4):
        if lines[i][3:] == [color, color, color, color, color]:
            return 1
    return 0

def exist_huo_4(lines): # 存在活四
    color = lines[0][3]
    for i in range(4):
        if lines[i][2:] == [-1, color, color, color, color, -1]:
            return 1
    return 0

def num_of_chong_4(lines): # 冲四的数量
    num = 0
    color = lines[0][3]
    anti_color = (255, 255, 255) if color == (0, 0, 0) else (0, 0, 0)
    for i in range(4):
        if lines[i][3:] == [color, color, color, -1, color]:
            num += 1
        elif lines[i][3:] == [color, -1, color, color, color]:
            num += 1
        elif lines[i][2:] == [anti_color, color, color, color, color, -1]:
            num += 1
        elif lines[i][2:] == [-1, color, color, color, color, anti_color]:
            num += 1
        elif lines[i][3:] == [color, color, -1, color, color]:
            num += 1
    return num

```

```

def num_of_huo_3(lines): # 活三的数量
    color = lines[0][3]
    num = 0
    for i in range(4):
        if lines[i][2:7] == [-1, color, color, color, -1]:
            num += 1
        elif lines[i][2:] == [-1, color, color, -1, color, -1]:
            num += 1
        elif lines[i][2:] == [-1, color, -1, color, color, -1]:
            num += 1
    return num

def num_of_mian_3(lines): # 眠三的数量
    color = lines[0][3]
    anti_color = (255, 255, 255) if color == (0, 0, 0) else (0, 0, 0)
    num = 0
    for i in range(4):
        if lines[i][2:] == [anti_color, color, color, color, -1, -1]:
            num += 1
        elif lines[i][1:7] == [-1, -1, color, color, color, anti_color]:
            num += 1
        elif lines[i][2:] == [anti_color, color, color, -1, color, -1]:
            num += 1
        elif lines[i][2:] == [-1, color, -1, color, color, anti_color]:
            num += 1
        elif lines[i][2:] == [-1, color, color, -1, color, anti_color]:
            num += 1
        elif lines[i][2:] == [anti_color, color, -1, color, color, -1]:
            num += 1
        elif lines[i][3:] == [color, -1, color, -1, color]:
            num += 1
        elif lines[i][3:] == [color, color, -1, -1, color]:
            num += 1
        elif lines[i][3:] == [color, -1, -1, color, color]:
            num += 1
        elif lines[i][1:] == [anti_color, -1, color, color, color, -1,
                             anti_color]:
            num += 1
    return num

def num_of_huo_2(lines): # 活二的数量
    color = lines[0][3]
    num = 0
    for i in range(4):
        if lines[i][1:6] == [-1, -1, color, color, -1]:
            num += 1
        elif lines[i][2:7] == [-1, color, color, -1, -1]:
            num += 1
        elif lines[i][2:7] == [-1, color, -1, color, -1]:
            num += 1
        elif lines[i][2:] == [-1, color, -1, -1, color, -1]:
            num += 1
    return num

def num_of_mian_2(lines): # 眠二的数量
    color = lines[0][3]
    anti_color = (255, 255, 255) if color == (0, 0, 0) else (0, 0, 0)
    num = 0
    for i in range(4):
        if lines[i][0:6] == [-1, -1, -1, color, color, anti_color]:
            num += 1
        elif lines[i][2:] == [anti_color, color, color, -1, -1, -1]:
            num += 1
        elif lines[i][1:7] == [-1, -1, color, -1, color, anti_color]:
            num += 1

```



```

    num += 1
elif lines[i][2:] == [anti_color, color, -1, color, -1, -1]:
    num += 1
elif lines[i][2:] == [-1, color, -1, -1, color, anti_color]:
    num += 1
elif lines[i][2:] == [anti_color, color, -1, -1, color, -1]:
    num += 1
elif lines[i][3:] == [color, -1, -1, -1, color]:
    num += 1
return num

def get_four_lines(board, x, y): # 得到(x,y)点为中心, 竖直、水平、左斜、右斜四个方向的棋子连线
    vertical = [] # 竖直
    left = [] # 左斜
    right = [] # 右斜
    horizon = [] # 水平
    j_left = y - 3
    j_right = y + 3
    for i in range(x - 3, x + 4 + 1):
        if i < 0 or i > 14:
            vertical.append(None)
            left.append(None)
            right.append(None)
        else:
            vertical.append(board[i][y])
            if j_left < 0 or j_left > 14:
                left.append(None)
            else:
                left.append(board[i][j_left])
            if j_right < 0 or j_right > 14:
                right.append(None)
            else:
                right.append(board[i][j_right])
            if j_left < 0 or j_left > 14:
                horizon.append(None)
            else:
                horizon.append(board[x][j_left])
        j_left += 1
        j_right -= 1
    return [horizon, vertical, left, right]

def valuation_function(board): # 评估函数
    num = {"black_l5": 0, "black_h4": 0, "black_c4": 0, "black_h3": 0,
    "black_m3": 0, "black_h2": 0, "black_m2": 0,
    "white_l5": 0, "white_h4": 0, "white_c4": 0, "white_h3": 0,
    "white_m3": 0, "white_h2": 0, "white_m2": 0}
    for x in range(15):
        for y in range(15):
            if board[x][y] != -1:
                lines = get_four_lines(board, x, y) # 得到(x,y)点为中心, 竖直、水平、左斜、右斜四个方向的棋子连线
                if board[x][y] == (0, 0, 0): # 黑子情况
                    num["black_l5"] += exist_lian_5(lines)
                    num["black_h4"] += exist_huo_4(lines)
                    num["black_c4"] += num_of_chong_4(lines)
                    num["black_h3"] += num_of_huo_3(lines)
                else: # 白子情况
                    num["white_l5"] += exist_lian_5(lines)
                    num["white_h4"] += exist_huo_4(lines)

```



```
num["white_c4"] += num_of_chong_4(lines)
num["white_h3"] += num_of_huo_3(lines)
if num["white_l5"] != 0: # 对方存在连五
    return -100000

if num["black_l5"] != 0: # 己方存在连五
    return 100000

# 再一步对方就赢:
if num["white_h4"] != 0: # 对方存在活四
    return -99990
if num["white_c4"] != 0: # 对方存在冲四
    return -99980

# 再一步己方就赢
if num["black_c4"] > 1: # 己方活四数量大于 1 (四四杀)
    return 99990
if num["black_h4"] != 0: # 己方存在活四
    return 99980
if num["black_h3"] != 0 and num["black_c4"] != 0: # 己方存在活三和冲四 (四三
杀)
    return 99970

# 再两步对方就赢
if num["white_h3"] != 0: # 对方存在活三
    return -99970

# 再两步己方就赢
if num["black_h3"] > 1: # 己方活三数量大于 1 (三三杀)
    return 99960

for x in range(15):
    for y in range(15):
        if board[x][y] != -1:
            lines = get_four_lines(board, x, y)
            if board[x][y] == (0, 0, 0):
                num["black_m3"] += num_of_mian_3(lines)
                num["black_h2"] += num_of_huo_2(lines)
                num["black_m2"] += num_of_mian_2(lines)
            else:
                if exist_lian_5(lines) != 0:
                    return -100000
                num["white_m3"] += num_of_mian_3(lines)
                num["white_h2"] += num_of_huo_2(lines)
                num["white_m2"] += num_of_mian_2(lines)

black_total = 0
black_total += num["black_c4"] * 2500 # 我方存在冲四
black_total += num["black_h3"] * 2500 # 我方存在活三
black_total += num["black_m3"] * 625 # 我方存在眠三
black_total += num["black_h2"] * 150 # 我方存在活二
black_total += num["black_m2"] * 35 # 我方存在冲二
white_total = 0
white_total += num["white_m3"] * 625 # 对方存在眠三
white_total += num["white_h2"] * 150 # 对方存在活二
white_total += num["white_m2"] * 35 # 对方存在冲二
return black_total - white_total

# 另外一种设计:
# black_total += num["black_c4"] * 10000
# black_total += num["black_h3"] * 1000
# black_total += num["black_m3"] * 100
# black_total += num["black_h2"] * 10
# black_total += num["black_m2"] * 1
# white_total += num["white_m3"]
# white_total += num["white_h2"]
# white_total += num["white_m2"]
```



```
def get_alpha_or_beta(n, board, ancestors_alpha, ancestors_beta, deep): #  
alpha-beta 剪枝  
    if n == deep: # 博弈树叶子结点  
        return valuation_function(board)  
    elif n % 2 == 0: # MAX 结点  
        node_alpha = -float('inf')  
        for x in range(15): # 遍历棋盘  
            for y in range(15):  
                if board[x][y] == -1:  
                    if have_chess_in_range_2(board, x, y) == False: # 如果结点太  
偏僻则不用扩展  
                    continue  
  
                child_board = copy.deepcopy(board) # 扩展子结点  
                child_board[x][y] = (0, 0, 0)  
  
                node_alpha = max(node_alpha, get_alpha_or_beta(n + 1,  
child_board, max(node_alpha, ancestors_alpha), ancestors_beta, deep)) # 递归  
实现  
  
    if node_alpha > ancestors_beta: # 剪枝操作  
        return node_alpha  
    else: # MIN 结点  
        node_beta = float('inf')  
        for x in range(15): # 遍历棋盘  
            for y in range(15):  
                if board[x][y] == -1:  
                    if have_chess_in_range_2(board, x, y) == False: # 如果结点太  
偏僻则不用扩展  
                    continue  
  
                child_board = copy.deepcopy(board) # 扩展子结点  
                child_board[x][y] = (255, 255, 255)  
  
                node_beta = min(node_beta, get_alpha_or_beta(n + 1,  
child_board, ancestors_alpha, min(node_beta, ancestors_beta), deep)) # 递归实  
现  
  
    if ancestors_alpha > node_beta: # 剪枝操作  
        return node_beta  
    return node_beta  
  
def have_chess_in_range(board, x, y): # 判断正方形范围内是否存在棋子  
    for i in range(x - 2, x + 3):  
        if i < 0 or i > 14:  
            continue  
        for j in range(y - 2, y + 3):  
            if j < 0 or j > 14:  
                continue  
            if board[i][j] != -1:  
                return True  
    return False  
  
def have_chess_in_range_2(board, x, y): # 判断正方形范围内是否存在棋子  
    for i in range(x - 1, x + 2):  
        if i < 0 or i > 14:  
            continue  
        for j in range(y - 1, y + 2):  
            if j < 0 or j > 14:  
                continue  
            if board[i][j] != -1:  
                return True  
    return False
```

```
deep = 3 # 深度
```

```
def AlphaBetaSearch(board, EMPTY, BLACK, WHITE, black, file):
    file.write("白方落子后对棋盘进行评估: (○代表白子, ●代表黑子) \n")

    max_beta = -float('inf') # 用于挑选 beta 值最大的子结点
    max_x = 0
    max_y = 0
    global deep

    for x in range(15): # 遍历棋盘
        for y in range(15):
            if board[x][y] == -1:
                if have_chess_in_range(board, x, y) == False: # 如果位置太偏僻则
                    不用扩展
                print('_\t', end='')
                file.write("_\t")
                continue

            child_board = copy.deepcopy(board) # 扩展子结点
            child_board[x][y] = (0, 0, 0)

            child_beta = get_alpha_or_beta(1, child_board, max_beta,
float('inf'), deep) # alpha-beta 剪枝获得子结点的 beta 值

            print(child_beta, '\t', sep='', end='') # 打印评估结果
            file.write("{}\t".format(child_beta))

            if child_beta == 100000: # 已经找到获胜结点
                deep = 1
            if child_beta == max_beta: # 新 beta 值和当前最大值相等, 则选取更靠近
                棋盘中心的
                if abs(x - 7) + abs(y - 7) < abs(max_x - 7) + abs(max_y -
7):
                    max_x = x
                    max_y = y
                elif child_beta > max_beta: # 更新最大 beta 值
                    max_beta = child_beta
                    max_x = x
                    max_y = y
                else: # 打印棋盘
                    print("●" if board[x][y] == (0, 0, 0) else "○", '\t', sep='',
end='')
                    file.write("{}\t".format("●" if board[x][y] == (0, 0, 0) else
"○"))
                print('')
                file.write("\n")
            file.write("根据棋盘的评估, 黑方选择落子的位置为{}\n\n".format((max_x, max_y)))
    return max_x, max_y, max_beta
```

4. 创新点&优化

1、自行设计了评估函数：（1）这个评估函数考虑了几乎全部绝杀的情况，并且不同的绝杀情况进行了符合逻辑的优先级排序，最后的效果是 20 关残局全部都在很少的回合数内就取得了获胜。（2）评估函数将判断对方存在连五的情况设为了最先判断的条件，如果存在对方存在连五则直接返回，以节约时间。（3）评估函数分成了两次棋盘遍历，第一次是进行绝杀情况的判断，第二次是进行眠三、活二、眠二的数量统计，这样做可能会导致少数几个回合时间增加，但是整个游戏的时间得到了减少。

2、alpha-beta 剪枝中，扩展子结点的方式改为一个一个扩展，即不使用函数返回包含全部子结点的列表，而是使用了 for 循环一个一个遍历子结点，这样可以为剪枝节省一定的时间。



间和空间。

3、定义了一个函数判断某个点的一定范围附近是否具有棋子，如果这个点附近没有棋子，则说明此点过于偏僻，可以不用评估以节省时间。

4、如果找到了可以获胜的结点，则将后面回合的搜索深度减为 1，这样可以节省最后几个回合的时间，同时可以防止其选择其他也可以获胜但回合数更多的结果。

三、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）

20个关卡的全部结果详见文件，这里选择其中较难的3个关卡进行分析：

(1) 第14关:

根据棋盘记住 黑方选择落子的位置为(8, 6)此时alpha值为-6900

首先算法选择了活三

白棋落子的位置为(5, 9)

接下来白棋堵截活三，我们的算法选择了冲四，99960 表示算法发现了存在两个活三的绝杀情况，即上图的 8230 处。

白棋落子的佈置法(12-1)

白棋被迫堵截冲四后，果然我们的算法选择了上一步发现的可能存在两个活三的位置作为下一步，只不过算法当前已经判断到即使后面存在两个活三也不能绝杀，但是该位置的评估值为 6320 还是较高的



白棋落子的位置为(7, 10)

根据棋盘评估，黑方选择落子的位置为(8, 11),此时alpha值为:10760

接下来白棋选择了冲四，算法被迫堵截。可以看到前面位置的评估值都为-100000，但是当扫描到必须堵截的位置之后，剩下位置的评估值都小于要被迫堵截的位置的评估值，说明算法有在进行剪枝。

白棋落子的位置为(4, 6)

根据棋盘评估，黑方选择落子的位置为(10, 11),此时alpha值为:99970

之后白棋对我们进行了防守，算法在一个比较意想不到的位置发现了己方存在活三和冲四的绝杀情况。

白棋落子的位置为(8, 9)

根据棋盘评估，黑方选择落子的位置为(9, 11),此时alpha值为:99980

白棋在(8, 9)位置进行防守。可以看到，不管白棋是防守(8, 9)还在(8, 10)我们都可以制造出绝杀情况，说明上一步算法的评估是合理的。我们选择了99980的位置为后面创造活四。

白棋落子的位置为(8, 10)

这里在描述的上边域我们使用的是 α 和 β 表示的。

这里白棋被迫去堵截我们的冲四放开了我们的活三，我们也就成功的創造出了活四即将取得胜利。



白棋落子的位置为(6, 11)

根据鬼畜体，黑子选择萨拉的位置是(11, 5)。此时， \bar{x}_1 的值为 -1000000。

根据棋盘评估，黑方选择落子的位置为(11, 11),此时alpha值为:100000

成功在 8 步内获胜。

(2) 第 17 关

根据棋盘评估，黑方选择落子的位置为(8, 10)，此时aiMove值为：02375

标注差生选择了上图(8-10)位置创造出了一个冲四

左側葉子的位置 (a - e)

根据棋盘评估，黑方选择落子的位置为(7, 9),此时alpha值为:2975

之后白棋被迫堵截冲四。算法又继续创造了一个活三。

白棋落子的位置为(9, 4)

根据棋盘评估 黑方选择落子的位置为(10, 3)。此时alpha值为:99960

白棋没有堵截我们的活三，白棋选择了冲四。算法选择去堵截冲四，同时探测到己方活三数量大于1的绝杀情况，即99960的含义。



白棋落子的位置为(5, 7)

根据棋盘评估，黑方选择落子的位置为(9, 9), 此时alpha值为: 99980

白棋冲四之后去堵截我们之前创造的活三，此时我方局势已经比较乐观，算法探测到多个可以绝杀的位置。

白棋落子的位置为(9, 6)

根据棋盘评估，黑方选择落子的位置为(6, 6)。此时alpha值为-100000。

根据棋盘评估，黑方选择落子的位置为(6, 6), 此时alpha值为: 1000000

白棋也创造出了活三数量大于1的绝杀情况，但是我方比对方快了一步，算法已经找到了可以到达的获胜结点。

白棋落子的位置为(5, 5)

根据棋盘评估，黑方选择落子的位置为(10, 10)，此时alpha值为:100000

根据收益计算，东方进阶基金的收益率为(111-100)/100=11%，此时的单位净值为1.11。

最终6步取胜。

(3) 第9天:

根据棋盘评估，黑方选择落子的位置为(8, 7)，此时alpha值为:150

在這種情況下，我們應該考慮的並非是「誰對誰錯」，而是如何才能讓

第 9 天的主要难点在于存在一些关键步骤不能出错，比如第一步如果算法选择去堵截白方的活四，那么就会直接输掉，这对评估函数有一定的要求。这里我们看到，虽然算法没有发现堵截白棋活四的位置会导致被对方绝杀，因为绝杀的出现在比较后面的回合，但是算法还是能识别到这一位置白方优势更大，从而选择了我方进行冲四。



白棋落子的位置为(5, 7)

根据棋盘评估，黑方选择落子的位置为(7, 6),此时alpha值为:-210

白棋被迫堵截我方的冲四。算法发现了多个会被白棋绝杀的位置，算法选择了优势最大的位置落子进行防守。

白棋落子的位置为(7, 5)

根据棋盘规则，黑王选择落子的位置是(2, 4)此时，上一子值为-255

根据棋盘评估，黑方选择落子的位置为(8, 4),此时alpha值为:265

白棋创造了一个活三，算法选择堵截这个活三继续进行防守。

白棋落子的位置为(9, 8)

根据棋盘评估，黑方选择落子的位置为(9, 4),此时alpha值为:99960

白棋创造了两个活二扩大优势。算法此时发现己方活三数量大于 1 的绝杀情况，即 99960 位置。

白棋落子的位置为(4, 8)

根据棋盘评估，黑方选择落子的位置为(3, 9)，此时alpha值为: 99980

根据棋盘评估，黑方选择落子的位置为(3, 9),此时alpha值为:99980



白棋落子的位置为(5, 8)

根据棋盘评估，黑方选择落子的位置为(7, 4)，此时alpha值为：100000

白棋创造出多个活二、活三。算法继续之前确定好的绝杀路线，选择了(7, 4)位置创造出活四。

白棋落子的位置为(10, 4)

根据权益价值，点力起升机子的权益价值为 11,770 万元人民币。

成功获胜。

2. 评测指标展示及分析（机器学习实验必须有此项，其它可分析运行时间等）

| | 关卡 1 | 关卡 2 | 关卡 3 | 关卡 4 | 关卡 5 |
|---------|---------|---------|---------|----------|---------|
| 总剪枝发生次数 | 974 | 1791 | 4803 | 4795 | 1910 |
| 平均每步剪枝数 | 324 | 358 | 800.5 | 799 | 477 |
| 总用时 | 14.6711 | 57.3289 | 85.1469 | 114.7796 | 35.7009 |
| 平均每步用时 | 4.8903 | 11.4657 | 14.1911 | 19.1257 | 89.2523 |

| | 回合 1 | 回合 2 | 回合 3 | 回合 4 | 回合 5 |
|--------|---------|----------|---------|----------|----------|
| 剪枝发生次数 | 219 | 1523 | 168 | 1477 | 1207 |
| 用时 | 8. 1777 | 25. 1530 | 2. 3098 | 18. 4259 | 10. 1969 |

与预期稍有不同，整体来看，剪枝次数发生越多反而用时越长。经过分析，这应该是由于子结点数量和回合数造成的。在剪枝次数相近的情况下，比如第一关和第二关、第三关和第四关。相比第二关，第一关子结点数更少，所用的回合数也更少，导致两关剪枝次数相近的情况下，第一关平均用时更少。第三关和第四关回合数一样，但是第三关子结点数更少，导致第三关用时更少。

四、 参考资料

教程和课件