中山大学计算机学院 本科生实验报告（2023学年春季学期）

课程名称：Artificial Intelligence 人工智能

教学班级		专业（方向）	
学号 2233 6173		姓名 罗弘杰	

实验题目

实验任务

- 编写一个五子棋博弈程序，要求用Alpha-beta剪枝算法，实现人机对弈。棋局评价函数可以参考已有文献（例：基于 alpha-beta 剪枝技术的五子棋 - 知乎 (zhihu.com)），但报告要引用参考过的文献和程序。
- 结果分析
  - 微信小程序“欢乐五子棋”中的残局闯关的前20关，选择其中至少1关进行分析。AI能否胜利，分析原因；将每一步的所有分支对应的评估函数值输出，分析是否合理；分析评估函数的剪枝效果，如剪掉节点数的占比
  - 第14关较难，作为附加题
- 加分项示例
  - 算法实现优化分析，分析优化后时间和空间的变化
  - 不同评价函数对比，分析不同评价函数的剪枝效果
- 提交
  - 一周，截止时间为4月15日晚23:59，压缩包命名格式为E4_学号.zip

实验内容

1. 算法原理

   博弈树搜索：源自博弈论中的理论， 在游戏或竞争中出现双方博弈的情况， 互相出招， 见招拆招从而出现了博弈树的概念。

   极大极小值（mini-max）搜索：对于五子棋的博弈树， 可以提供一种启发式函数， 判断每一种情况的分支， 再自己出招的时候选择分值最大的情况， 在分析敌人出招的时候选择分值最小的情况，然后给定一深度depth， 在达到这个深度以后， 返回分值， 并通过选择回溯到根节点， 选择最大的分值， 就是概率上利益最大化的出招方式。

   alpha-beta剪枝：对于博弈树，要分析的结点数随深度的增长而指数级扩张， 可以根据博弈树的特点选择一种剪枝方式： 在max结点处， 若其子节点min返回的第一个value, 而且value<=max, 那么该min结点就没有扩张的必要了， 因为再怎么扩张都会比value小而不会改变其父节点的alpha， 这叫做alpha剪枝。 同理， 对于min结点， 也存在对应的剪枝方式， 叫做beta剪枝。 运用得当该剪枝方式可以大大减少扩张的结点数量从而提高博弈树的深度， 或者加快计算的速度。

2. 伪代码

```
#alpha beta初始化分别为-inf, inf
alpha-beta-puring(board, depth, alpha, beta, is_ai):
    #判断游戏结束或者博弈树深度达到了：
    if game_win(user1) or game_win(player2) or reach_depth():
```

```
        return evaluation(board)

    child_list = get_successors_ordered(board) #使用排序使得剪枝效果更好

    if is_ai:  #Max结点
        for child in child_list:
            value = alpha-beta-puring(board, depth-1, alPha, beta, not is_ai
)
            if value <= alpha:
                break  #alpha剪枝
            else:
                alpha = value
    else:      #min 结点
        for child in child_list:
            value = alpha-beta-puring(board, depth-1, alPha, beta, not is_ai
)
            if value >= beta:
                break  #beta剪枝
            else:
                beta = value
```

3. 关键代码展示（带注释）

```
# 极大极小算法加alpha-beta剪枝
def alpha_beta(is_ai, depth, alpha, beta):
    # 判断游戏是否结束，或者达到了搜索的深度
    if game_win(list1) or game_win(list2) or depth == 0:
        return evaluation(is_ai)

    blank_list = list(set(list_all).difference(set(list3)))

    order(blank_list)    # 对搜索的顺序进行排序，提高剪枝效率
    # 遍历每一个空格
    for next_step in blank_list:
        global search_count
        search_count += 1

        # 如果当前位置周围没有相邻的位置被落子，不进行搜索
        if not has_neightnor(next_step):
            continue

        if is_ai:
            list1.append(next_step)
        else:
            list2.append(next_step)
        list3.append(next_step)

        value = -alpha_beta(not is_ai, depth - 1, -beta, -alpha)    #负号表示对
手的得分

        if is_ai:
            list1.remove(next_step)
```

```
        else:
            list2.remove(next_step)
        list3.remove(next_step)

        if depth == DEPTH:  #在决策的时候
            global LOSE_FLAG
            if LOSE_FLAG:  #如果该步骤会导致敌方有胜利的机会，那么就不要走这一步
                LOSE_FLAG = False
                continue
        if value > alpha:
            if depth == DEPTH:
                next_point[0] = next_step[0]
                next_point[1] = next_step[1]
            # alpha + beta剪枝
            if value >= beta:
                print(str(depth) + " " +str(value) + "  alpha:" + str(alpha)
+ "beta:" + str(beta))
                print(list3)
                global cut_count
                cut_count += 1
                return beta
            alpha = value

    return alpha
```

启发式函数计算：

思路：建立三个列表， list1是ai下棋的位置， list2是用户下棋的位置， list3是被下棋了的位置， 对于每次计算， 如果是ai下， 就计算自己下的分数-用户下的分数， 对于用户下的，同样这么分析。 提供一个宏参数ratio, ratio用于表示策略是攻击性还是防守性。

```
# 评估函数
def evaluation(is_ai):
    total_score = 0

    if is_ai:
        my_list = list1
        enemy_list = list2
    else:
        my_list = list2
        enemy_list = list1

    # 计算我方的总分数
    score_all_arr = []  # 记录棋型的位置列表，用于判断是否重复计算
    my_score = 0
    for pt in my_list:
        m = pt[0]
        n = pt[1]
        my_score += cal_score(m, n, 0, 1, enemy_list, my_list, score_all_arr)
        my_score += cal_score(m, n, 1, 0, enemy_list, my_list, score_all_arr)
        my_score += cal_score(m, n, 1, 1, enemy_list, my_list, score_all_arr)
        my_score += cal_score(m, n, -1, 1, enemy_list, my_list,
score_all_arr)

    # 计算敌方的总分数
```

```
    score_all_arr_enemy = []
    enemy_score = 0
    for pt in enemy_list:
        m = pt[0]
        n = pt[1]
        enemy_score += cal_score(m, n, 0, 1, my_list, enemy_list,
score_all_arr_enemy)
        enemy_score += cal_score(m, n, 1, 0, my_list, enemy_list,
score_all_arr_enemy)
        enemy_score += cal_score(m, n, 1, 1, my_list, enemy_list,
score_all_arr_enemy)
        enemy_score += cal_score(m, n, -1, 1, my_list, enemy_list,
score_all_arr_enemy)

    if not is_ai and enemy_score >= 999999:  #判断现在Ai会不会赢（连五）
        global WIN_FLAG
        WIN_FLAG = True

    if not is_ai and my_score >= 50000 and not WIN_FLAG:  #如果AI还没连五，而且
用户已经活四或者连五
        global LOSE_FLAG
        LOSE_FLAG = True  # 输了
    total_score = my_score - enemy_score*10

    return total_score
```

线性计算分数：

自己下的分数要遍历每一个棋子位置的四个方向的分数，注意，由于一个方向上几个棋子的分数可能会重复计算，所以要维护一个列表，记录每一个线性分数的起始位置和方向以及分数， 从而避免重复计算这个方向上的分数。 由于五子棋中相互交叉的情况对于胜利有加成，所以还有计算一个额外的add_score， 他表示在这个点上存在相互交叉的情况， 分数将会叠加

```
# 威胁形状的得分
shape_score = [(50, (0, 1, 1, 0, 0)),
               (50, (0, 0, 1, 1, 0)),
               (200, (1, 1, 0, 1, 0)),
               (500, (0, 0, 1, 1, 1)),
               (500, (1, 1, 1, 0, 0)),
               (5000, (0, 1, 1, 1, 0)),
               (5000, (0, 1, 0, 1, 1, 0)),
               (5000, (0, 1, 1, 0, 1, 0)),
               (5000, (1, 1, 1, 0, 1)),
               (5000, (1, 1, 0, 1, 1)),
               (5000, (1, 0, 1, 1, 1)),
               (5000, (1, 1, 1, 1, 0)),
               (5000, (0, 1, 1, 1, 1)),
               (50000, (0, 1, 1, 1, 1, 0)),
               (999999, (1, 1, 1, 1, 1))]


# 计算每个位置的得分
def cal_score(m, n, x_decrict, y_derice, enemy_list, my_list, score_all_arr):
```

```
    add_score = 0  # 增加的得分
    # 在当前方向上，取得最大的棋型得分
    max_score_shape = (0, None)

    # 如果该方向上已经计算过棋型的得分，则不再计算
    for item in score_all_arr:
        for pt in item[1]:
            if m == pt[0] and n == pt[1] and x_decrict == item[2][0] and
y_derice == item[2][1]:
                return 0

    # 遍历当前位置前后5个位置
    for offset in range(-5, 1):
        pos = []
        for i in range(0, 6):
            if (m + (i + offset) * x_decrict, n + (i + offset) * y_derice) in
enemy_list:
                pos.append(2)
            elif (m + (i + offset) * x_decrict, n + (i + offset) * y_derice)
in my_list:
                pos.append(1)
            else:
                pos.append(0)
        tmp_shap5 = (pos[0], pos[1], pos[2], pos[3], pos[4])
        tmp_shap6 = (pos[0], pos[1], pos[2], pos[3], pos[4], pos[5])

        for (score, shape) in shape_score:
            if tmp_shap5 == shape or tmp_shap6 == shape:
                if tmp_shap5 == (1,1,1,1,1) or tmp_shap6 == (0,1,1,1,1,0):
                    print('win')

                if score > max_score_shape[0]:
                    max_score_shape = (score, ((m + (0+offset) * x_decrict, n
+ (0+offset) * y_derice),
                                               (m + (1+offset) * x_decrict, n
+ (1+offset) * y_derice),
                                               (m + (2+offset) * x_decrict, n
+ (2+offset) * y_derice),
                                               (m + (3+offset) * x_decrict, n
+ (3+offset) * y_derice),
                                               (m + (4+offset) * x_decrict, n
+ (4+offset) * y_derice)), (x_decrict, y_derice))

    # 将最大的棋型加入到记录列表中
    if max_score_shape[1] is not None:
        for item in score_all_arr:
            for pt1 in item[1]:
                for pt2 in max_score_shape[1]:
                    if pt1 == pt2 and max_score_shape[0] > 10 and item[0] >
10:
                        add_score += item[0]

        score_all_arr.append(max_score_shape)

    return add_score + max_score_shape[0]
```

4. 创新点&优化（如果有）

    1. 考虑了棋形叠加交叉的情况：

    在五子棋中， 两条线相交会大大增加胜利概率， 所以维护一个记录每个点上的线性棋子， 在每次计算分数的时候，与这个列表比对，如果有交叉的地方那么需要加上额外的分数

```
 # 将最大的棋型加入到记录列表中
    if max_score_shape[1] is not None:
        for item in score_all_arr:
            for pt1 in item[1]:
                for pt2 in max_score_shape[1]:
                    if pt1 == pt2 and max_score_shape[0] > 10 and item[0] >
10:
                        add_score += item[0]
```

    2. 考虑了博弈策略的选择：

    考虑到五子棋的攻防偏向策略， 设定一个ratio权重。 在计算分数的时候呈上这个分数，ratio>1就是攻击策略； ratio<1就是防守策略。

为了避免矩阵传递带来的性能消耗，使用全局变量维护矩阵：

    每次下棋只会改变一个棋子的位置， 与其传递整个矩阵，不如传递落子的位置。为此需要在全局变量区维护一个记录棋子矩阵的数据结构：

```
list1 = []  # AI方的落子记录
list2 = []  # 人类方的落子记录
list3 = []  # 所有落子记录
```

    3. 如果下棋的位置周围九宫格一个棋都没有，就不要考虑，减少计算量：

    通常不会选择一个周围都没有棋子的地方落子

```
# 判断一个位置是否有相邻的落子
def has_neightnor(pt):
    for i in range(-1, 2):
        for j in range(-1, 2):
            if i == 0 and j == 0:
                continue
            if (pt[0] + i, pt[1]+j) in list3:
                return True
    return False
```

    4. 扩展结点排序， 根据获胜概率，将上次落子和上上次落子周围的区域的结点调整到扩展队列的前面将提高找到最优解的概率，由此增加剪枝数

```
#  对搜索顺序进行优化， 将最后一次落子的位置放在列表的最前面
def order(blank_list):
    for last_pt in list3[:-2:-1]:
        for item in blank_list:
            for i in range(-1, 2):
                for j in range(-1, 2):
                    if i == 0 and j == 0:
                        continue
                    if (last_pt[0] + i, last_pt[1] + j) in blank_list:
                        blank_list.remove((last_pt[0] + i, last_pt[1] + j))
                        blank_list.insert(0, (last_pt[0] + i, last_pt[1] +
j))
```

5. 防止虚假胜利：

   在3层博弈树中，如果在第二层ai已经得到连五或者活四， 然后第三层ai还没有得到连五， 那么下一次人类下棋，就会获胜。

   防治方法： 维护一个LOSE_FLAG全局变量。 当上述情况出现的时候， 将LOSE_FLAG置为1. 然后在外部函数调用时，要判断LOSE_FLAG, 如果会导致输了， 就抛弃这一步。

```
LOSE_FLAG= False
alpha_beta():
    ...
    if depth == DEPTH:  #在决策的时候
            global LOSE_FLAG
            if LOSE_FLAG:  #如果该步骤会导致敌方有胜利的机会，那么就不要走这一步
                LOSE_FLAG = False
                continue
    ...

def evaluaion:
    ...
    if not is_ai and enemy_score >= 999999:  #判断Ai会不会赢（连五）
        global WIN_FLAG
        WIN_FLAG = True

    if not is_ai and my_score >= 50000 and not WIN_FLAG:  #如果AI还没连五，
而且用户已经活四或者连五
        global LOSE_FLAG
        LOSE_FLAG = True  # 输了
    total_score = my_score - enemy_score*10
    ...
```

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）

案例14：

```
# # # # # # # # # # # # # # #
```

```
# # # # # # # # # # # # # # #
# # # # # # # # # # # # # # #
# # # # # # 1 # # # # # # # #
# # # # # # # 0 # # # # # # #
# # # # # # # 1 0 # # # # # #
# # # # # # # 1 1 0 # # # # #
# # # # # # # 1 1 # # # # # #
# # # # # 0 # 0 0 # # # # # #    #ai应该在（8， 6） 下棋，防止人类获胜
# # # # # # # # # # # # # # #
# # # # # # # # # # # # # # #
# # # # # # # # # # # # # # #
# # # # # # # # # # # # # # #
# # # # # # # # # # # # # # #
# # # # # # # # # # # # # # #
```

可以看到棋盘中

```
def test():

    with open("lab4\\example\\14.txt", "r") as f:
        for i, line in enumerate(f.readlines()):
            for j, x in enumerate(line.strip().split(" ")):
                if x=="1":
                    list1.append((i, j))
                    list3.append((i, j))
                if x=="0":
                    list2.append((i, j))
                    list3.append((i, j))
    print(list3)
    x, y, alpha = Search(0, 1, 2, True,  (8, 5) , board = None)
    print(x, y, alpha)

if __name__  == "__main__":
    test()
    #
```

```
cut time: 270
search time: 22320
cut efficiency: 0.012096774193548387
alpha: 210450
used location: :[(3, 6), (4, 7), (5, 7), (5, 8), (6, 7), (6, 8), (6, 9), (7, 7), (7, 8), (8, 5), (8, 7), (8, 8), (8, 5), (8, 6)]
8 6 210450
```

最后一行是最佳落子位置， 以及alpha分数。 在这个棋局中，确实只有（8, 6）才能获胜。

ai怎么做到的：

使用了LOSE_FLAG的方法， 在这个棋局中， 如果不走（8， 6）， 第二步用户就会形成活四，而且这个棋局中ai无法在第三步下成连五， 所以这样的步骤所有的FLAG都会被置为1， 导被抛弃。 走（8， 6 ）则不会出现这种情况

案例一

```
# # # # # # # # # # # # # # #
# # # # # # # # # # # # # # #
# # # # # # # # # # # # # # #
# # # # # # 0 # # 0 # # # # #
# # # # # # # # 1 1 0 # # # #
# # # # # # # 0 1 1 1 # # # #
# # # # # # # 1 1 0 1 # # # #
```

```
# # # # # # # 1 0 # # # # # #
# # # # # # 1 0 0 # # # # # #
# # # # # 0 # 0 # # # # # # #
# # # # # # # # # # # # # # #
# # # # # # # # # # # # # # #
# # # # # # # # # # # # # # #
# # # # # # # # # # # # # # #
# # # # # # # # # # # # # # #
```

```
def test():

    with open("lab4\\example\\1.txt", "r") as f:
        for i, line in enumerate(f.readlines()):
            for j, x in enumerate(line.strip().split(" ")):
                if x=="1":
                    list1.append((i, j))
                    list3.append((i, j))
                if x=="0":
                    list2.append((i, j))
                    list3.append((i, j))
    print(list3)
    x, y, alpha = Search(0, 1, 2, True,  (8, 5) , board = None)
    print(x, y, alpha)

if __name__  == "__main__":
    test()
    #
```

```
cut time: 170
search time: 18784
cut efficiency: 0.009050255536626917
alpha: 1013850
used location: :[(3, 7), (3, 10), (4, 8), (4, 9), (4, 10), (5, 7), (5, 8), (5, 9), (5, 10), (6, 7), (6, 8), (6, 9), (6, 10), (7, 7), (7, 8), (8, 6), (8, 7),
 (8, 8), (9, 5), (9, 7), (8, 5), (3, 8)]
3 8 1013850
```

这个棋局是攻击有利， 1013850这个分数很大一部分来自第三层实现的连五带来的999999.

|----------如有优化，请重复1，2，分析优化后的算法结果----------|

对于排序后的优化比较（排序值得是将上次下棋的两个加入到排序队列的前端）：

排序前的案例**14**：

扩展结点从优化后的22320到41178， 剪枝效率也有所下降。

```
cut time: 247
search time: 41178
cut efficiency: 0.005998348632765068
alpha: 210450
used location: :[(3, 6), (4, 7), (5, 7), (5, 8), (6, 7), (6, 8), (6, 9), (7, 7), (7, 8), (8, 5), (8, 7), (8, 8), (8, 5), (8, 6)]
8 6 210450
```

排序前的案例一：

可以看到相比优化后， 扩展结点从18784增加到26435

```
cut time: 188
search time: 26435
cut efficiency: 0.007111783620200492
alpha: 1013850
used location: :[(3, 7), (3, 10), (4, 8), (4, 9), (4, 10), (5, 7), (5, 8), (5, 9), (5, 10), (6, 7), (6, 8), (6, 9), (6, 10), (7, 7), (7, 8), (8, 6), (8, 7),
 (8, 8), (9, 5), (9, 7), (8, 5), (3, 8)]
3 8 1013850
```

参考资料

PS：可以自己设计报告模板，但是内容必须包括上述的几个部分，不需要写实验感想

colingogogo/gobang_AI: 基于博弈树α-β剪枝搜索的五子棋AI (github.com)