人工智能实验

Lab10 博弈树搜索

姓名：郭俊楠　学号：17341045　时间：2019 年 11 月 26 日

1　算法原理

本次实验中，我们需要使用 **alpha-beta 剪枝搜索算法**。

相比与普通的 **minimax 搜索算法**，**alpha-beta 剪枝搜索算法**增加了剪枝这一步骤，在某些情况下可以剪去大量的搜索分支，避免一些不必要的搜索步骤，从而大大减少空间资源的占用与搜索时间。而算法进行剪枝的关键思路在于：**（以一个极大层非叶节点 s 为例）当节点 s 已经递归搜索了一些它的子节点，并更新了目前已知的极大值 α，如果在搜索它的下一个子节点 c 的过程中，发现节点 c 的极小值 $\beta < \alpha$，那么便可以立即停止搜索节点 c；因为如果当游戏状态从节点 s 转移到节点 c 时，理智的极小值方必然做出最优选择并使效益值小于 α，导致极大值方的利益比预期的低，而极大值方会避免出现这种状况，即避免从节点 s 转移到节点 c，所以继续在节点 c 上进行搜索是无意义的。**

从上述思路中我们可以发现，在某些条件下进行剪枝可以在不会影响搜索结果的情况下提高搜索效率，而显然当剪枝点越接近根节点、在兄弟节点中越靠前时，算法的搜索效率将提升得越高，这就要求我们对子节点进行一定的**排序**。另外，如果我们提前**过滤**一些极有可能会被剪枝的子节点，那么这虽然有较小的可能性会在一段博弈过程内产生小影响，但也对子节点的排序效率以及剪枝过程有较大的帮助。

2　实现思路

2.1　搜索策略

显然，这次实验中我们会使用 **alpha-beta 剪枝搜索算法**。而根据前面算法原理的分析过程，我们可以确定一些搜索策略的实现细节：

- **对子节点进行排序**。

  正如我们前面所说，对子节点进行排序很有可能会提升搜索效率，但问题是我们要根据什么对子节点进行排序。这里，我们可以考虑对每个子节点所代表的局势进行评价，并根据评价分数和当前执棋方进行递增或递减排序。这种排序在理论上并不是严格最优的，因为当前局势占优并不代表之后的局势一定占优；相反，我们是为了平衡排序效果与搜索效率而做出了理想化假设：**理智的双方进行零和博弈时，一方在当前局势占优很有可能意味着它在最终的局势占优**。

- **对子节点进行过滤**。

  我们面对的博弈过程是五子棋游戏，棋盘大小为 11x11，即有 121 个格子。如果将所有空格作为搜索的子节点，那么一个节点可能就会有 100 个左右的子节点；即使我们对子节点进行排序，使得大部分节点会被剪枝，排序过程也十分影响搜索效率。所以，我们可以对子节点进行两步过滤：

- **过滤掉四邻域内没有棋子的空格**。

  显然，许多没有邻居的棋子对局势的影响比较小，而有邻居的棋子既可能与本方棋子一起扩大优势，又可能对对方棋型造成阻碍，这就导致没有邻居的棋子更有可能被剪枝。所以，将其过滤掉往往利大于弊。

- **限定子节点数量**。

  在我们的期望中，经过排序后，剪枝点会比较靠近子节点中的前面，而后面的子节点不论会不会被剪枝，对搜索过程都不会造成太大的影响。于是，我们可以在对子节点进行排序后只考虑靠前的几个子节点，这样可以减少空间资源的使用。

2.2 评价函数

对于五子棋游戏，如果想要判断局势的好坏，一个简单直了的方式便是观察敌我两方的棋子分布。于是，我们设计的评价函数主要基于对局势的判断，即敌我两方各种棋型的数量。但一个显而易见的问题是，五子棋的棋型数量是一个很庞大的数，且不论我们能不能找出所有棋型，即使我们知道所有棋型，判断当前棋局中存在这些棋型中的哪些便是一件很困难且耗时的事情。万幸的是，只要找出一些很常见、对局势的影响力较大的棋型并进行统计，就可以使我们的评价函数有比较好的表现。

这里我们不展开讨论五子棋规则和棋型的种类与定义，而是针对一些棋型的评分进行分析。相关棋型及其对应的分数如表 1 所示，其中：1 表示本方棋子，-1 表示对方棋子，0 表示空格。

棋型	示例	分数
连五	1 1 1 1 1	1e8
活四	0 1 1 1 1 0	1e5
冲四	0 1 1 1 1 -1	5e3
活三	0 1 1 1 0	1e3
眠三	0 1 1 1 -1	5e2
活二	0 1 1 0	1e2

表 1: 棋型评分表

可以看到，我们主要考虑六种棋型，且两两之间有一定的差距。我们的评分过程主要遵循以下原则：

- **每种棋型评分的高低主要看其能够在对手干扰下达成连五的可能性的大小**。

  这条原则是很显然的，因为连五代表着己方的胜利，那么越有可能达成连五，即越有可能达成本方的胜利，相应的评分自然越高。另外，有两点值得注意：

  - **对方的干扰是存在的**。

    例如，如果对方不进行干扰，那么活四和冲四的评分应该是一样的，因为两者都只差一步便构成连五；但这种情况是不现实的，考虑对方有可能进行干扰，活四有两个格子的机会可以构成连五，而冲四只有一个，很显然活四更有可能构成连五，其分数自然更高。

  - **棋型评分之间的差距跟两者的转化难度有关**。

上述棋型要么可以直接转化为连五，要么会在转化为连五的过程中先转化为其他棋型。于是，一个棋型转化为连五的可能性可以用它转化为某个中间棋型的可能性与该中间棋型转化为连五的可能性来衡量。自然地，棋型评分之间的差距也将代表着一方转化为另一方的难度。比如，活四与活三的评分相差 10000 以上，而活三与活四的评分相差 300, 因为活三转化为活四的过程中受到的干扰更大，成功的可能性更小。

- **棋型之间的分数为整数，且应该成倍数关系**。

  这项原则的存在主要是为了方便我们进行验算和测试合理性等过程，且我们倾向于认为多个某种棋型的组合是可以与单个更高评分的棋型相当的，这也是分数之间成倍数的原因。

- **构成棋型所需的本方棋子数差距越大，棋型之间的分数差距越大**。

  这项原则是第一条原则的一种表现，本方棋子数差距越大，棋型之间的转化难度越高，分数差距自然也越大。

我们依照评分原则确定了相应棋型的分数，但这些分数并不是绝对的或最优的，而只是可行的某种方案。或许在经过一些细微的改动后，棋型的分数会更符合实际，而这个过程或许会需要借助强化学习等工具，这脱离了我们的主题，所以我们不在此进行深入讨论。

确立了评分之后，我们的评价函数自然也就确立了下来：**在棋局中统计极大值方各个棋型的数目，并以相应评分为权重，加权求和后得到极大值方的分数；之后以同样方法计算极小值方的分数，并对下一步棋的执棋方奖励额外的分数，再从极大值方的分数中减去极小值方的分数，最后便得到了当前棋局的最终分数**。

这里要注意的是，我们对下一步棋的执棋方奖励额外的分数，是因为一般来说在拥有同等棋型的情况下下一步棋的执棋方往往更加有优势。比如，当两方都拥有活四时，下一步棋的执棋方可以通过构成连五直接取得胜利。

此外，在 **alpha-beta 剪枝搜索**的过程中，我们需要频繁地对棋局进行评价，所以要注意评价函数的效率。这里我们将棋局中各个方向的向量组成列表，然后进行卷积处理，再进行相应判断；而比起逐点遍历，这种方式的效率更高。具体实现可见关键代码部分。

3 伪代码

下面我们给出关键部分的伪代码。

首先，在 **alpha-beta 剪枝搜索**的过程中，我们需要获取子节点列表以进行下一步搜索。但正如上文所说的那样，我们还需要额外进行过滤、排序等工作。相关伪代码如算法 1 所示，我们忽略那些下一步落子周围没有棋子的子节点，根据子节点对应局势的评分对子节点进行排序，并对最终子节点的数目做出限制。

在明确了如何获取子节点列表后，我们可以直接给出 **alpha-beta 剪枝搜索**的过程，伪代码如算法 2 所示，根据具体实现的情况，这里我们对搜索深度做出了限制，虽然损失了一部分性能，但确保了一定的搜索效率。

在评价一个棋局时，我们可以先获取各个方向的向量，即行向量、列向量、主对角线方向的向量以及副对角线的向量，然后使用每个棋型模板与各个向量做卷积并计数，就可以对棋型进行统计。从这里可以看出，要对棋型和评分做出变动，我们只需要维护一个字典即可，这样可以确保评价函数的核心没有太大的变动。相关伪代码如算法 3 所示。

Algorithm 1 To get a children list

```
function GETCHILDRENLIST(node, Player, LimitedAmount)
    ChildList ← node.Successors(Player)
    ChildrenScoreList ← List()
    for child in ChildrenList do
        if GetNeighborNum(child) > 0 then
            score ← Evaluate(child)
            if Player == MAX then
                ChildrenScoreList.append((child, score))
            else
                ChildrenScoreList.append((child, -score))
    ChildrenScoreList.DescendSort(key="the second element")
    FinalChildrenList ← List()
    for i = 1:min(Len(ChildrenScoreList), LimitedAmount) do
        FinalChildrenList.append(ChildrenScoreList[i][0])
    return FinalChildrenList
```

Algorithm 2 Alpha-beta pruning algorithm

```
function ALPHABETA(node, alpha, beta, Player, MaxDepth)
    if GameOver(node) or MaxDepth ≤ 0 then
        return Evaluate(node)
    ChildrenList ← GetChildrenList(node, Player, LimitedAmount)
    if Player == MAX then
        for child in ChildrenList do
            NextBeta ← AlphaBeta(child, alpha, beta, MIN, MaxDepth-1)
            alpha ← max(alpha, NextBeta)
            if alpha ≥ beta then
                break
        return alpha
    else
        for child in ChildrenList do
            NextAlpha ← AlphaBeta(child, alpha, beta, MAX, MaxDepth-1)
            beta ← min(beta, NextAlpha)
            if alpha ≥ beta then
                break
        return beta
```

Algorithm 3 Evaluation function

function EVALUATE(node)
    $Board \leftarrow node.GetBoard()$
    $VectorList \leftarrow GetVectorList(Board)$
    $ModelList \leftarrow GetModelList()$
    $MaxScore \leftarrow 0$
    $MinScore \leftarrow 0$
    for *model* in *ModelList* **do**
        for *vector* in *VectorList* **do**
            $result \leftarrow convolve(vector, model)$
            $MaxScore \leftarrow MaxScore + count(result, 1)$
            $MinScore \leftarrow MinScore + count(result, -1)$
    $score \leftarrow MaxScore - MinScore$
    return *score*

4 关键代码

下面我们展示五子棋 AI 部分的关键代码并加以必要的说明，五子棋游戏本体部分的代码不多赘述。

4.1 alpha-beta 剪枝搜索

alpha-beta 剪枝搜索部分的代码如下所示。这里的实现思路与前面展示的伪代码逻辑是一致的，但为了提高搜索效率，我们使用了一些辅助的变量。其中，*valid_board* 表示每个空格的四邻域内有多少棋子，且置非空格的值为 0，于是我们就可以比较高效地发现哪些空格是有邻居的，因为维护这个变量的代价主要为在更新每步棋的时候更新 *valid_board* 的一部分，而这要比我们每次都去遍历每个格子并搜索它的邻居有效率得多。

```
def alpha_beta(board, valid_board, turn, alpha, beta, depth, board_vec):
    '''
        alpha-beta剪枝搜索算法
        参数:
            board: numpy.array棋盘，0为空格，-1为AI棋子，+1为玩家棋子
            valid_board: 可行步，>0表示可行
            turn: 极大(1) 或 极小(-1)
            alpha, beta: 极大极小值
            depth: 限制深度
            board_vec: 各个方向的向量
        返回值:
            (best_row, best_col): 已知最好的下一步旗
            _: 效益值
    '''
    # 检查是否到达终止条件
    if depth==0 or if_game_over(board, turn, board_vec):
        return None, None, evaluate(board, turn, board_vec)

    # 获取可行步列表
    childList =getNextSteps(board, turn, valid_board, board_vec)
```

```
21
22      best_row, best_col =0, 0
23
24      if turn ==1:
25          # max
26          for row, col in childList:
27              # 递归调用
28              board[row, col] =turn
29              valid_board[row, col], temp =0, valid_board[row, col]
30              for i in range(max(0, row-1), min(board.shape[0], row+2)):
31                  for j in range(max(0, col-1), min(board.shape[1], col+2)):
32                      if board[i, j] ==0:
33                          valid_board[i, j] +=1
34              _, _, next_beta =alpha_beta(board, valid_board, -turn, alpha, beta, depth-1,
                                                                    board_vec)
35              for i in range(max(0, row-1), min(board.shape[0], row+2)):
36                  for j in range(max(0, col-1), min(board.shape[1], col+2)):
37                      if board[i, j] ==0:
38                          valid_board[i, j] -=1
39              board[row, col] =0
40              valid_board[row, col] =temp
41              # 更新alpha
42              if alpha <next_beta:
43                  alpha =next_beta
44                  best_row, best_col =row, col
45              # 剪枝判断
46              if alpha >=beta:
47                  break
48          return best_row, best_col, alpha
49      else:
50          # min
51          for row, col in childList:
52              # 递归调用
53              board[row, col] =turn
54              valid_board[row, col], temp =0, valid_board[row, col]
55              for i in range(max(0, row-1), min(board.shape[0], row+2)):
56                  for j in range(max(0, col-1), min(board.shape[1], col+2)):
57                      if board[i, j] ==0:
58                          valid_board[i, j] +=1
59              _, _, next_alpha =alpha_beta(board, valid_board, -turn, alpha, beta, depth-1,
                                                                    board_vec)
60              for i in range(max(0, row-1), min(board.shape[0], row+2)):
61                  for j in range(max(0, col-1), min(board.shape[1], col+2)):
62                      if board[i, j] ==0:
63                          valid_board[i, j] -=1
64              board[row, col] =0
65              valid_board[row, col] =temp
66              # 更新beta
67              if beta >next_alpha:
68                  beta =next_alpha
69                  best_row, best_col =row, col
70              # 剪枝判断
71              if beta <=alpha:
```

```
72                 break
73         return best_row, best_col, beta
```

4.2 获取子节点

下面的代码实现了我们获取子节点的过程，包括对子节点的两次过滤以及排序的操作。可以看到，由于 *valid_board* 的存在，我们可以略去判断每个空格是否有四邻居的过程，这对我们的搜索过程是很有帮助的。

```
1   def getNextSteps(board, turn, valid_board, board_vec):
2     '''
3         获取可行的下一步列表
4         参数:
5             board: numpy.array棋盘，0为空格，-1为AI棋子，+1为玩家棋子
6             turn: 极大(1) 或 极小(-1)
7             valid_board: 邻居统计
8             board_vec: 各个方向的向量
9         返回值:
10            next_steps: 下一步列表
11    '''
12    # 经过排序后，末尾的空位价值一般不高，可以忽略
13    LIMITED_AMOUNT =8
14    # 只选择四邻域内有棋子的空位
15    next_steps =[]
16    for i in range(board.shape[0]):
17        for j in range(board.shape[1]):
18            if valid_board[i, j] >0:
19                next_steps.append((i, j))
20
21    # 根据评分进行排序
22    next_steps.sort(key=lambda x: getPointScore(board, x[0], x[1], turn, board_vec),
                                                    reverse=True)
23
24    return next_steps[:LIMITED_AMOUNT]
```

4.3 评价函数

下面的代码实现了评价函数。注意到我们的卷积核（即模板）经过特殊的设计，使得它在与向量做卷积时，如果向量相应部分与模板匹配，则会使结果呈现 1（极大值方棋型）或 -1（极小值方棋型），方便我们进行判断。但要注意有些模板由于结构问题未能检测到边缘部分，所以需要进行额外的边缘检测；当然，我们也可以选择对向量进行 *padding* 操作，但这种做法的效率显然比较低。

```
1   def evaluate(board, turn, board_vec):
2     '''
3         评价函数，得出当前分数
4         参数:
5             board: numpy.array棋盘，0为空格，-1为AI棋子，+1为玩家棋子
6             turn: 执棋方
7             board_vec: 各个方向的向量
```

```
        返回值:
            score: 当前分数
    '''
    # 计算两方评分
    max_score =0
    min_score =0

    # 各个方向的向量，过滤棋子少的向量
    board_vec =[vec for vec in get_board_vec(board) if np.sum(vec!=0) >1]

    # 遍历各种棋型
    for model, model_score in score_dict:
        # 统计频数并更新分数
        pos, neg =conv2d(board_vec, model)
        max_score +=pos *model_score
        min_score +=neg *model_score

    # 边缘棋型检测
    for vec in board_vec:
        for model, model_score in front_edge_score_dict:
            if len(vec) >=len(model):
                result =np.sum(vec[:len(model)] *model)
                if result ==1:
                    max_score +=model_score
                elif result ==-1:
                    min_score +=model_score

        for model, model_score in back_edge_score_dict:
            if len(vec) >=len(model):
                result =np.sum(vec[-len(model):] *model)
                if result ==1:
                    max_score +=model_score
                elif result ==-1:
                    min_score +=model_score


    # 执棋方有额外加分
    if turn ==1:
        max_score +=max_score //10
    else:
        min_score +=min_score //10

    # 最终得分
    score =max_score -min_score

    return score
```

5 实验结果分析

游戏开局如图 1 所示，先手执黑棋。下面我们展示不同深度限制上三个连续回合的棋局分布情况，其中玩家先手，AI 后手；且为了方便比较，玩家下棋保持一致，观察 AI 是否有不同

的表现。

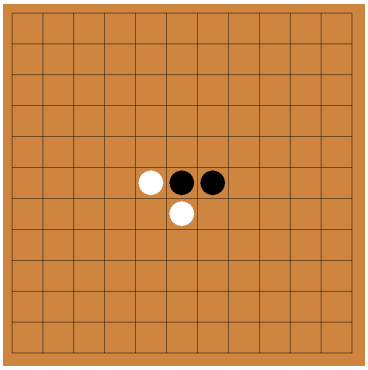
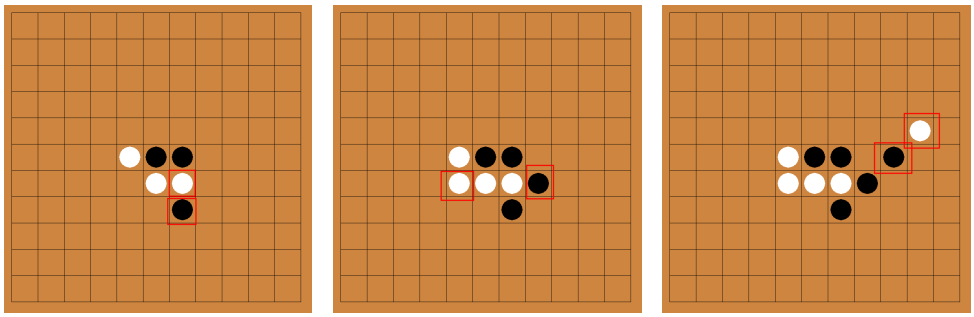


图 1: 棋局初始分布，先手执黑棋

当限制深度为 2 时，棋局分布如图 2 所示。可以看到，回合 2 的时候，白棋（AI）有进攻的趋势；而回合 3 的时候，白棋（AI）进行了防守。这说明了 AI 有进攻和防守的能力。日志输出如图 3 所示，可以看到，随着双方的落子，棋局的分数在上下波动；AI 花费 1 秒左右进行搜索并落子，速度较快。



(a) 回合 1　　(b) 回合 2　　(c) 回合 3

图 2: 限制深度为 2 时三个连续回合的棋局分布，红框表示回合内双方所下的棋

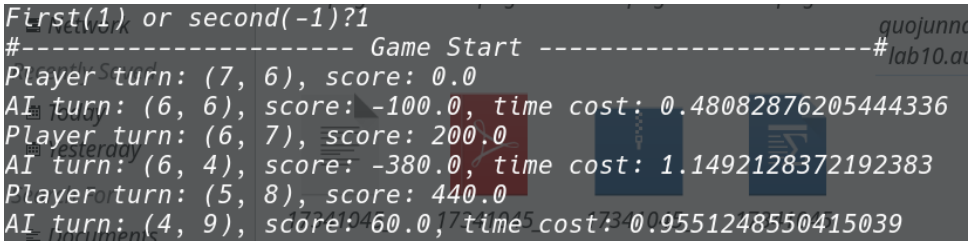
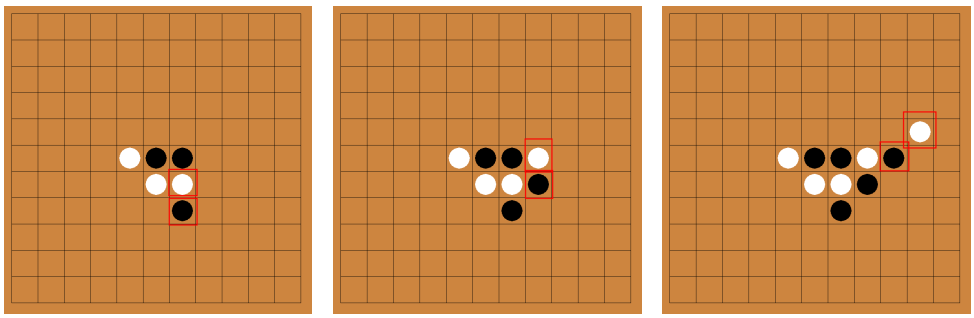


图 3: 限制深度为 2 时三个连续回合的日志输出

当限制深度为 4 时，棋局分布如图 4 所示。与前面进行比较，发现回合 1 和回合 3 的落子是一样的；但关于回合 2 的落子，当限制深度为 4 时，白棋（AI）的落子既可能构成进攻棋型，又可以阻碍对方棋型，综合了进攻和防守，很明显比限制深度为 2 时的落子考虑更加全面。但观察日志输出（图 5），可以发现 AI 搜索所花费的时间在 15 秒左右，要比限制深度为 2 时的耗时高得多。总的来说，这是花费更多时间进行更深层次的搜索，以换取更优的结果。



(a) 回合 1　　(b) 回合 2　　(c) 回合 3

图 4: 限制深度为 4 时三个连续回合的棋局分布，红框表示回合内双方所下的棋

```
First(1) or second(-1)?1
#-------------------- Game Start --------------------#
Player turn: (7, 6), score: 0.0
AI turn: (6, 6), score: -100.0, time cost: 11.940649986267090
Player turn: (6, 7), score: 200.0
AI turn: (5, 7), score: 120.0, time cost: 16.677584886550903
Player turn: (5, 8), score: 990.0
AI turn: (4, 9), score: 560.0, time cost: 15.655818462371826
```

图 5: 限制深度为 4 时三个连续回合的日志输出

6　改进思路与方向

从实验结果来看，我们的 AI 还有提升的空间，一方面是速度上的提升，一方面则是性能上的提升。可以考虑的改进方向如下：

- **完善评价函数。**
  评价函数的完善可分为两部分：一是匹配策略，显然匹配与检测过程占评价过程的很大一部分，如果能对匹配策略进行改进，将给搜索效率带来很大的提升；二是棋型种类及评分，棋型的种类越多，评分越贴近实际，评价函数就越能准确地评价当前棋局，而要想在这方面进行改进，既可以进行多次的尝试和试验，也可以借助强化学习等手段。
- **子节点的排序**
  目前我们根据棋局的评分对子节点进行排序，但这可能还是显得有点粗糙。如果能找到一个可以更准确反映剪枝可能性的排序依据，搜索过程的剪枝数目将会提升，搜索效率也会相应提高。
- **代码重构**
  实现过程我们着重考虑了代码可读性，所以不可避免地牺牲了一点代码性能。如果有必要，我们可以对代码进行重构，使得代码更加高效。此外，目前我们使用 python 语言来实现项目，但这个过程并没有用到太高级的库，所以可以考虑使用 C/C++ 进行重构，相应的工作量不会特别大，但那将会使代码的效率有很大的提升。