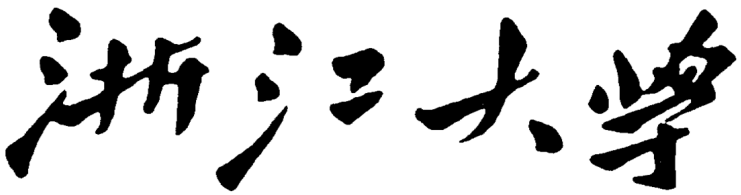

本科生实验报告

AI Reversi

课程名称:	人工智能
姓名:	陈卓
学院:	计算机科学与技术学院
专业:	计算机科学与技术
学号:	3170101214
指导老师:	郑能干

1 问题简介

翻转棋，又称黑白棋（Othello），是由黑白双方进行的一种策略游戏，游戏是在一个 8×8 的黑白棋棋盘（通常是绿色的）上进行的。双方共持 64 个棋子，每个棋子一面是黑色、一面是白色。为了方便起见，开始时每位棋手各持 32 个棋子。但是所持棋子并非个人专用，如果对手的棋子用完，他必须把多余的棋子再给对方一些。棋子如果是黑色的一面朝上，它就是黑棋；如果是白色的一面朝上，就是白棋。

行棋规则如下：

1. 黑方先行，双方交替下棋；
2. 一步合法的棋步包括：
   - 在一个空格处落下一个棋子，并且翻转对手一个或多个棋子；
   - 新落下的棋子必须落在可夹住对方棋子的位置上，对方被夹住的所有棋子都要翻转过来，可以是横着夹，竖着夹，或是斜着夹。夹住的位置上必须全部是对手的棋子，不能有空格；
   - 一步棋可以在数个（横向，纵向，对角线）方向上翻棋，任何被夹住的棋子都必须被翻转过来，棋手无权选择不去翻某个棋子。
3. 如果一方没有合法棋步，也就是说不管他下到哪里，都不能至少翻转对手的一个棋子，那他这一轮只能弃权，而由他的对手继续落子直到他有合法棋步可下；
4. 如果一方至少有一步合法棋步可下，他就必须落子，不得弃权；
5. 棋局持续下去，直到棋盘填满或者双方都无合法棋步可下；
6. 如果某一方落子时间超过 1 分钟或者连续落子 3 次不合法，则判该方失败。

本实验目标是通过搜索算法，实现一个 AI 棋手。

2 问题分析和相关工作

黑白棋多年的发展，和全世界棋手对它行棋策略的挖掘给我们带来了很多启发式的决策根据，这为 AI 黑白棋算法的发展和优化提供了基础。

黑白棋 AI 起源于 90 年代，当时棋力最强的 Thor 堪比世界级棋手，但它仅将人类总结的行棋策略，如行动力、奇偶数等，人工添加入程序中，导致棋力的局限性很大。90 年代中期，供黑白棋 AI 线上对抗的网站出现，Logistello 横空出世。Michael Buro 同样使用了已经总结的策略特征，并将这些特征进行线性组合，此外，他还将棋局按棋子数量分为若干阶段，特征之间线性组合的权重与棋局阶段相关。与 Thor 不同，Logistello 利用 LR 算法，通过上万棋谱的学习确定权重，这使得黑白棋 AI 的棋力大幅提高，迅速超过人类。

3 算法原理与实现

3.1 最小最大搜索

最小最大搜索利用递归实现，代码如下。

```
def minimax(self, board, step):
   return self._minimax(board, self.color, step)

def _minimax(self, board, color, step):
   action, val = None, None
   if step > 0:
      is_max_node = color == self.color
      legal_actions = list(board.get_legal_actions(color))
      if len(legal_actions):
         for action_t in legal_actions:
            flipped = board._move(action_t, color)
            _, val_t = self._minimax(board, reverse_color(color), step - 1)
            board.backpropagation(action_t, flipped, color)

            if val is None:
               val = val_t
               action = action_t
            else:
               if (is_max_node and (val_t > val)) or (not is_max_node and (val_t <
                   val)):
                  val = val_t
                  action = action_t
      else:
         _, val = self._minimax(board, reverse_color(color), step - 1)
   else:
      val = self.get_score(board)
   return action, val
```

3.2 Alpha-Beta 剪枝搜索

Alpha-Beta 剪枝搜索利用递归实现，代码如下。

```
def alpha_beta_prunig(self, board, step):
    return self._alpha_beta_prunig(board, self.color, step, -math.inf, math.inf)

def _alpha_beta_prunig(self, board, color, step, alpha, beta):
    action, val = None, None
    if step > 0:
        is_max_node = color == self.color
        legal_actions = list(board.get_legal_actions(color))
        if len(legal_actions):
            for action_t in legal_actions:
                flipped = board._move(action_t, color)
                _, val_t = self._alpha_beta_prunig(board, reverse_color(color), step -
                    1, alpha, beta)
                board.backpropagation(action_t, flipped, color)
                if is_max_node and val_t > alpha:
                    alpha = val_t
                    action = action_t
                if not is_max_node and val_t < beta:
                    beta = val_t
                    action = action_t
                if alpha >= beta:
                    break
            val = alpha if is_max_node else beta
        else:
            _, val = self._alpha_beta_prunig(board, reverse_color(color), step - 1,
                alpha, beta)
    else:
        val = self.get_score(board)
    return action, val
```

3.3 局面评估

由于下期时间的限制和搜索空间的庞大，算法不可能对整棵搜索树进行完全搜索，故搜索深度被限制在有限步内。在这样的情况下，对一次搜索的叶结点棋局的形势评估显得尤为重要，极端条件下，如果可以做到对局面评估完全准确，那么算法将是最优解。

本实验的评估函数从人类在黑白棋发展中总结的经验选择了 3 种特征，分不同阶段地用不同的权重将其线性组合。具体如下。

1. 聚集度 C。选择聚集度高的落子策略在开局十分有效。一方面，开局的棋子数量较少，对凝聚手[1]的分析往往费力不讨好，而通过凝聚手的定义我们可以发现，凝聚手达成的效果和棋子的集中程度

[1] 凝聚手是指产生较少边界子的棋步。

成正相关；另一方面，在开局选择集中落子可以有效避免主动向边角延伸，这一好处来源于在开局占四角的概率很低，如果选择向边角靠拢，容易为对手留下占角棋步。本实验中，聚集度 C 通过棋子位置方差体现，聚集度越高，方差越小。

$$C = var(rival) - var(self)$$

2. 行动力 A。提高自己行动力、降低对手行动力在中局尤为重要。在中局，随着棋子数增多，棋局的选择和复杂程度显著提高，在这一情况下，提升自身可落子位置数量能够提高找到凝聚手的概率，而凝聚手的效果在中局也能直观体现在行动力的提升上。

$$A = action\text{-}num(self) - action\text{-}num(rival)$$

3. 稳定子 S。稳定子指将来不会被翻色的子，例如四角。稳定子的作用在尾局的作用更为明显，因为棋局开始与中心，稳定子在前期不易获得。它的优点显而易见，因为这样的分数是不会下降的。本实验中，利用对棋局的正反双重遍历，求得了部分稳定子。

$$S = stable\text{-}num(self) - stable\text{-}num(rival)$$

对于这三个特征的权重 $\boldsymbol{w}$，本实验做如下定义

$$\boldsymbol{w} = \begin{cases} (5,4,1), & 4 \le n < 20 \\ (1,15,3), & 20 \le n < 40 \\ (1,1,10), & 40 \le n < 64 \end{cases} \tag{1}$$

其中，n 代表棋面棋子数量，用 20/40 作为开局与中局/中局与尾局的分界。根据上述策略，在开局/中局/尾局中，应该更侧重 $C/A/S$。最终棋面得分

$$W = \boldsymbol{w}(C, A, S)^T$$

伪代码如下。

```
def get_dis_score(board, color):
    get cordinates from board
    calculate variance of each side
    return var_that - var_this


def get_action_score(board, color):
    actions_this = list(board.get_legal_actions(color))
    actions_that = list(board.get_legal_actions(reverse_color(color)))
    return len(actions_this) - len(actions_that)


def get_stable_score(board, color):
    traversal board to get stable information from direction: left(0), above(1),
        left_above(2), right_above(3)
```

```
    traversal reversely board to get stable information from direction: right(4),
        below(5), right_below(6), left_below(7)

    calculate final stabel information:
       (0 or 4) and (1 or 5) and (2 or 6) and (3 or 7)

    return stable_this_cnt - stable_that_cnt


def get_score(self, board):
    cnt = board.count(self.color) + board.count(reverse_color(self.color))
    scores = [dis_score, action_score, stable_score]
    if cnt < 20:
       weights = np.array([5, 4, 1])
    elif cnt < 40:
       weights = np.array([1, 15, 3])
    else:
       weights = np.array([1, 1, 10])
    val = np.average(scores, weights=weights)
    return val
```

3.4 蒙特卡洛树搜索

本实验实现了 MCTS 算法，具体代码见 `mcts.py` 和 `ReversiNode.py`。由于最终并未选择使用 MCTS，在此不做过多说明。

4 算法测试

三种搜索算法中，Alpha-Beta 剪枝搜索是最小最大搜索的优化，因此首先比较 Alpha-Beta 剪枝和 MCTS。根据实验，在有时间限制的条件下，Alpha-Beta 剪枝搜索的表现优于 MCTS，故最终选择 Alpha-Beta 剪枝搜索算法。

在 Alpha-Beta 剪枝搜索中的权重选择源自于实战测试，首先确定一个 baseline，即以当前棋面的棋子数量差作为评估函数的 Alpha-Beta 剪枝算法为基准；之后令不同权重的算法均以 5 步深度与之对战。经过一部分测试，选择公式 1 所示数据时，先后手均至少可战胜 8 步的 baseline，并先后手均通过 Mo 平台上的高级测试。

5 分析与讨论

本实验中，在评估函数的选择上充分体现了专业知识导向的启发式函数的设计，通过对人类棋手的决策策略的建模，构建了有一定棋力的 AI 棋手。但另一方面，由于不能够量化人类棋手的策略（例如权重），具体表现与权重的个人选择有很大相关性。最为稳妥的方法应该与 Logistello 一致，即选定要使用特征，再用大量棋谱对权重进行学习、回归。