中山大学数据科学与计算机学院

计算机科学与技术专业-人工智能

本科生实验报告

（2018-2019 学年秋季学期）

学　　号：16337113

姓　　名：劳马东

教学班级：教务 2 班

专　　业：超算

一、实验题目

实现 6X6 的黑白翻转棋的人机对战，要求：

- 横排、竖排、对角线均可翻转；
- 要求使用 alpha-beta 剪枝；
- 搜索深度和评价函数不限，自己设计。在报告中说明清楚自己的评价函数及搜索策略；
- 实验结果要求展示至少连续三个回合（人和机器各落子一次指一回合）的棋局分布情况，并输出每步落子的得分。

二、实验内容

（一）算法原理

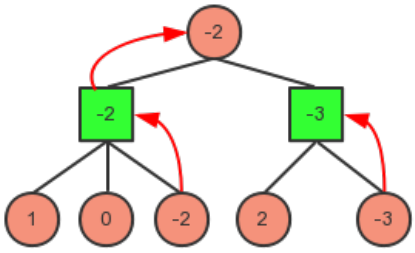
1、 博弈树

- 节点：每次落子后棋盘的状态
- 行动：在合法的位置落子
- 双方轮流扩展节点：两个玩家的的行动逐层交替出现
- 评价函数：当前棋盘状态的优劣得分
- 节点的值：对游戏双方都最优的子节点的值

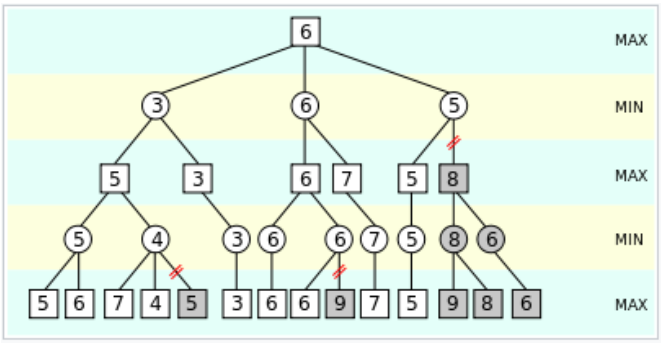
2、 *Minimax* 搜索

Minimax 搜索是一种 DFS 搜索，在节点的各层，玩家 A 和玩家 B 轮流地选择下一层中对自己最有利的子节点。不妨假设 A 选择节点值大的子节点，B 选择小的子节点（因为节点值小说明对 A 不利，换言之对 B 有利）。

如图 (a)，红色节点表示 A 的选择，绿色节点表示 B 的选择。在最底层的叶子节点，节点的值根据评估函数获得，之后，叶子节点的值往上传到父节点，绿色节点选择子节点中值最小的作为自己的值，因此绿色节点分别选择了-2 和-3，之后，-2 和-3 传到根节点，选择其中最大的一个，即-2 作为自己的值。



(a) *Minimax* 搜索例子



(b) $\alpha\beta$ 剪枝例子 1

算法 1 Minimax 搜索

输入: *node* 当前节点状态，*is_max* 是否极大节点

输出: 博弈树的值

```
1: function MINIMAX(node, is_max)
2:     if node is a leaf then
3:         return the heuristic value of node
4:     else
5:         if is_max then
6:             value ← −∞
7:             for each child of node do
8:                 value ← max(value, minimax(child, FALSE))
9:             end for
10:            return value
11:        else
12:            value ← +∞
13:            for each child of node do
14:                value ← min(value, minimax(child, TRUE))
15:            end for
16:            return value
17:        end if
18:    end if
19: end function
```

3、 $\alpha\beta$ 剪枝

在博弈树搜索中，有很多节点是没有必要扩展的。例如对于图 (b) 中的例子，DFS 搜索到值为 5 的节点时，就没有必要往下扩展它。因为它的父节点是极小节点，它会选择子节点中最小的一个，而前面搜索到的子节点的值 4 已经传给父节点，5 往下扩展必定会使它的值 >=5，这肯定不会被父节点选择。

判断一个节点是否可以剪枝，更普遍的标准是其 α 值大于或等于某个极小祖先节点的 β 值，或其 β 值小于或等于某个极大祖先节点的 α 值。因为对于极小祖先节点 A 来说，它选择的子节点的值必定比它的 β 值小，而 α 值大于 A 的 β 的后代节点 B 就对 A 的选择没有影响了，极大祖先节点同理。$\alpha\beta$ 剪枝算法的过程是：

(1) 每个节点维护两个值——α 和 β，分别表示极大节点已探索的子节点中最大的值和极小节点已探索的子节点中最小的值。

(2) 初始状态下，α 的值为 $-\infty$，β 的值为 $+\infty$。

(3) 在扩展子节点的过程中，如果发现 $\alpha \geq \beta$，就可以停止扩展。因为对于极大节点，继续探索只会使 α 值和节点值更大，而它的父节点会选择一个节点值 $\leq \beta$ 的节点，极小节点同理。

考虑图 (c)-(j) 的具体例子，从上至下各层用 r、a、g、p、c 标记：

(1) 步骤 1：初始时所有节点的 $\alpha = -\infty$、$\beta = +\infty$；

(2) 步骤 2：子节点 c_1 的 3 传给父节点 p_1，p_1 更新 β 值为 3；

(3) 步骤 3：继续探索子节点 c_2，值为 17，比 p_1 的 β 值大，不更新 p_1；

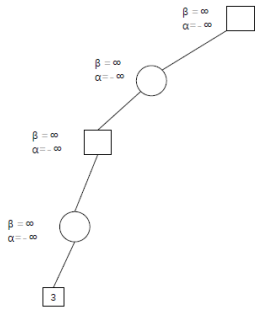
(4) 步骤 4：p_1 节点值确定为 3，向上传递给 g_1，g_1 更新 α 值为 3；

(5) 步骤 5：g_1 带着自己的 α 值向下扩展，探索到 c_3（值为 2），父节点 p_2 更新 β 值为 2，发现 $\alpha \geq \beta$，扩展结束；

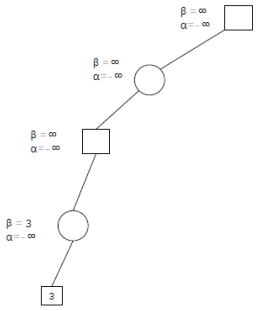
(6) 步骤 6：g_1 节点值确定为 3，向上传递给 a_1，a_1 更新 β 值为 3，然后带着这个 β 值向下探索；

(7) 步骤 7：探索节点 c_4（值为 15），更新 p_2 节点值为 15，继续往上更新 g_2 的 α 值为 15，g_1 节点值确定为 3，r 确定 α 为 3，向下扩展；

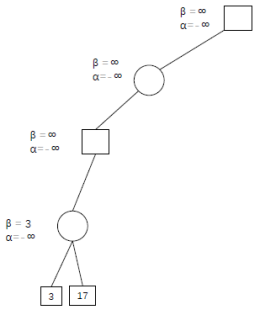
(8) 步骤 8：p_4、p_5 的过程同理。



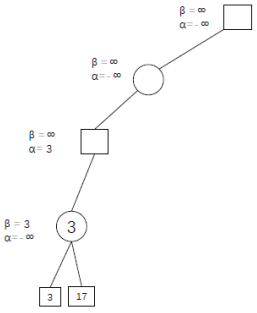
(c) step 1



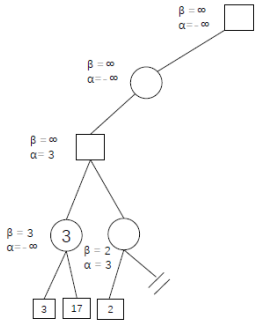
(d) step 2



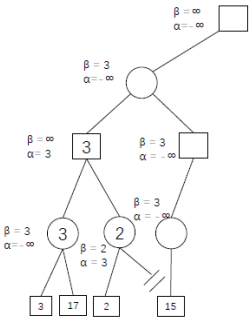
(e) step 3



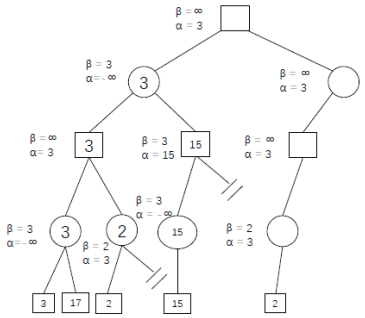
(f) step 4



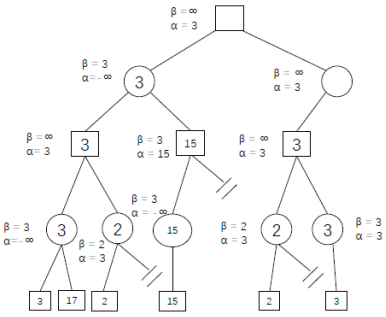
(g) step 5



(h) step 6



(i) step 7



(j) step 8

4、 *Negamax* 搜索

Minimax 算法的代码太过冗余，*Negamax* 对其进行简化。该算法基于 $\max(a,b) == -\min(-a,-b)$ 的事实来简化代码实现，即玩家 A 的节点值相当于玩家 B 节点值的负数。

算法 2 带 $\alpha\beta$ 剪枝的 Minimax 搜索

输入: *node* 当前节点状态，*alpha* 节点的 α 值，*beta* 节点的 β 值，*is_max* 是否极大节点

输出: 博弈树的值

```
1: function MINIMAX_WITH_ALPHABETA(node, alpha, beta, is_max)
2:     if node is a leaf then
3:         return the heuristic value of node
4:     else
5:         ...
6:         for each child of node do
7:             score ← minimax_with_alphabeta(child, alpha, beta, FALSE)
8:             if is_max then
9:                 value ← max(value, score)
10:                alpha ← max(alpha, value)
11:            else
12:                value ← min(value, score)
13:                beta ← min(beta, value)
14:            end if
15:            if alpha ≥ beta then
16:                break
17:            end if
18:        end for
19:        return value
20:    end if
21: end function
```

算法 3 *Negamax* 搜索

输入: *node* 当前节点状态，*factor* 极大节点取值 1，极小取值-1

输出: 博弈树的值

```
1: function NEGAMAX(node, factor)
2:     if node is a leaf then
3:         return factor× the heuristic value of node
4:     else
5:         value ← −∞
6:         for each child of node do
7:             value ← max(value, -negamax(child, −color))
8:         end for
9:         return value
10:    end if
11: end function
```

5、 带 $\alpha\beta$ 剪枝的 $Negamax$ 搜索

算法 4 带 $\alpha\beta$ 剪枝的 $Negamax$ 搜索

输入: $node$ 当前节点状态，$alpha$ 节点的 α 值，$beta$ 节点的 β 值，$factor$ 极大节点取值 1，极小取值-1

输出: 博弈树的值

```
1: function NEGAMAX_WITH_ALPHABETA(node, alpha, beta, factor)
2:     ...
3:     for each child of node do
4:         value ← max(value, -negamax_with_alphabeta(child, −beta, −alpha, −color))
5:         alpha ← max(alpha, value)
6:         if alpha ≥ beta then
7:             break
8:         end if
9:     end for
10:    return value
11: end function
```

6、 动态规划

在搜索过程中，有很多子问题重复出现，因此可以用一个表格记录已搜索节点的节点值，下一次遇到时直接查表而无需搜索。实验中发现，不使用动态规划前，6X6 棋盘每次落子搜索时间很长（等了十几分钟还没得到解），而加上动态规划之后能在十几秒以内得到解。

7、 评估函数的设计

(1) 棋子数之差

计算棋盘上先手棋子数与后手棋子数之差。

(2) sigmoid 函数

sigmoid 函数的作用是将取值为 $(-\infty, +\infty)$ 的数映射到 $(0, 1)$。评估函数首先统计棋盘上先手棋子和后手棋子个数，然后计算各自的 sigmoid 值，作为棋盘得分。

三、关键代码

1、 Minimax 搜索

代码中，初始化部分获取当前节点的棋子（对于极大节点是先手棋子，极小节点是后手棋子）。之后，获取在当前状态下能够落子的位置，而不是生成当前节点的后继状态，避免拷贝。如果无法落子，说明游戏结束，就返回对当前棋盘的评分，否则递归地扩展每一种落子位置。

递归完成之后回溯，恢复被翻转的棋子，尝试下一种落子。更新这些落子选择中最优的方案和 α、β 值，在 $\alpha \geq \beta$ 时停止扩展。除最优得分外，函数还需要记录最优落子位置，因为最顶层函数调用的目的就是要知道怎么落子。

```
def minimax(self, remain_depth, alpha, beta, is_max):
  ... # 一些初始化，获得极大极小节点的棋子
  moves_list = self._board.generate_moves(char)   # 当前可落子的位置
  if remain_depth == 0 or not moves_list:   # 到达设定深度或无法落子
      best_score = self._evaluation(self._board)
  else:
    for move in moves_list:
      all_reverse = self._board.apply_move(move, char, False)   # 落子，返回被翻的棋子
      score = self.minimax(remain_depth - 1, alpha, beta, not is_max)[0]
      for p in all_reverse:   # 回溯
          self._board[p] = opp_char
      if all_reverse:
          self._board[move] = Board.BLANK
      if is_max:    # alpha节点，选择较大的值
          if chosen_score > best_score:
              best_score = chosen_score
              best_move = move
          alpha = max(alpha, best_score)
      else:         # beta节点，选择较大的值
          if chosen_score < best_score:
              best_score = chosen_score
              best_move = move
          beta = min(beta, best_score)
      if alpha >= beta:   # alpha >= beta停止扩展
          break
    return best_score, best_move
```

2、 Negamax 搜索

Negamax 搜索的代码比 Minimax 搜索简洁，它不需要判断当前节点时极大节点还是极小节点，因为二者本身可以通过 $\max(a,b) = -\min(-a,-b)$ 这一公式转换。*color* 变量取值 ± 1，极大节点为 1，极小节点为-1，因为极小节点处的棋盘得分需要乘上-1 才能转化为极大节点得分。值得注意的是，父节点的 $-\beta$、$-\alpha$ 值传递给子节点作为 α 和 β 值。

```
def negamax(self, remain_depth, alpha, beta, color):
  ...
  best_score = -self.INF
  moves_list = self._board.generate_moves(char)
  if remain_depth == 0 or not moves_list:
      best_score = color * self._evaluation(self._board)
  else:
    for move in moves_list:
      all_reverse = self._board.apply_move(move, char, False)
      chosen_score = self.negamax(remain_depth - 1, -beta, -alpha, -color)[0]
      ... # 回溯
      if best_score < -chosen_score:
          best_score, best_move = -chosen_score, move
      alpha = max(alpha, best_score)
      if alpha >= beta:
          break
  return best_score, best_move
```

3、 动态规划

(1) 搜索前

```
s_board = str(self._board)
entry = self._table.get(s_board, None)
if entry is not None and entry.depth >= remain_depth:
    if entry.flag == Flag.VALUE:    # 保存的是节点值，返回节点值和落子
        return entry.value, entry.move
    elif entry.flag == Flag.ALPHA:   # 保存的是alpha或beta，更新alpha、beta
        alpha = max(alpha, entry.value)
    else:
        beta = min(beta, entry.value)
    if alpha >= beta:
        return entry.value, entry.move
alpha_orig = alpha
beta_orig = beta
```

(2) 搜索后

```
entry = Entry()
# 最终的best_score一定等于最终的alpha值或beta值
entry.value = best_score
entry.move = best_move
entry.depth = remain_depth
if entry.value <= alpha_orig:    # 满足上界，用于下次更新下界
    entry.flag = Flag.BETA
elif entry.value >= beta_orig:  # 满足下界，用于下次更新上界
    entry.flag = Flag.ALPHA
else:                           # 最初的alpha和beta值都没更新
    entry.flag = Flag.VALUE
self._table[s_board] = entry
```

四、实验结果及分析