中山大学计算机学院

人工智能

本科生实验报告

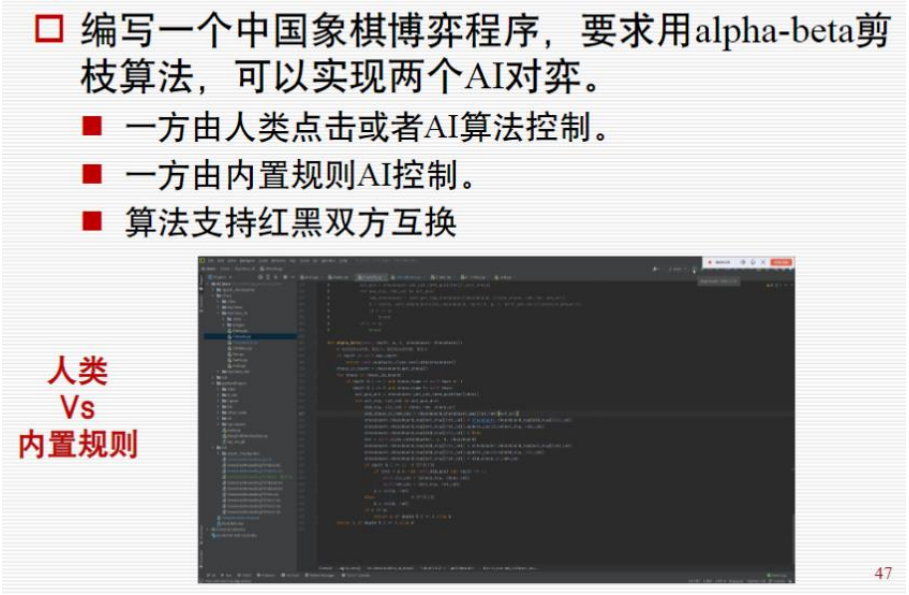
（2022 学年春季学期）

课程名称：Artificial Intelligence

教学班级	202320346	专业（方向）	计算机科学与技术
学号	21312450	姓名	林隽哲

一、 实验题目

利用博弈树搜索实现象棋 AI



二、 实验内容

1. 算法原理

假设：

- 玩家 A 和玩家 B 的**行动逐层交替**；
- A 和 B 的**利益关系对立**，即假设 A 要使分数更大，B 就要使分数更小；
- A 和 B 均采取**最优策略**。

Minimax 搜索：

- 找到博弈树中内部节点的值，其中 **Max 节点（A）**的每一步扩展要使**收益最大，Min 节点（B）**的扩展要使**收益最小**。
- 但随着博弈的进行，必须检查的游戏状态的数目呈**指数增长**。

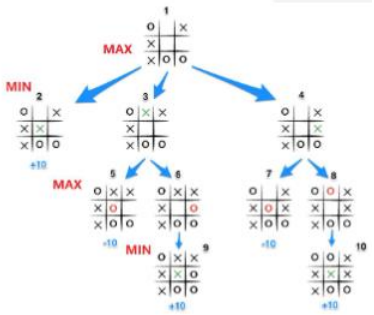
Alpha-Beta 剪枝：

- 剪掉**不可能影响决策的分支**，尽可能地消除部分搜索树，是缓解 Minimax 搜索空间复杂度过大的一种方案。
- 算法原理如下：
  - **Max 节点记录 alpha 值，Min 节点记录 beta 值**
  - Max 节点的 **alpha 剪枝：效益值 ≥ 任何祖先 Min 节点的 beta 值**
  - Min 节点的 **beta 剪枝：效益值 ≤ 任何祖先 Max 节点的 alpha 值**

2. 伪代码

3. Minimax搜索

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```



4. Alpha-beta剪枝

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

3. 关键代码展示（带注释）

Alpha-Beta pruning 算法的实现：

```python
def alpha_beta(self, chessboard: ChessBoard, depth, alpha, beta, is_max):

    # This Alpha-Beta pruning algorithm is based on recursion.
    # The termination condition of the recursion is to reach the specified depth or the leaf node.
    # Due to the diversity of chess moves, we cannot discuss all cases of the subsequent moves of the current situation at once,
    # so we often need to artificially specify the depth to prevent the algorithm space from being too large.

    # if reach the leaf node, then return the evaluate value
    if depth == 0:
        return self.evaluate_class.evaluate(chessboard)

    if is_max:
        max_eval = -float('inf')
        count_child = 0
        for chess in chessboard.get_chess():

            if chess.team != self.team:
                continue

            put_down_position = chessboard.get_put_down_position(chess)
            count_child += len(put_down_position)

            for pos in put_down_position:
                # save the old position and old chess on the new position
                old_row, old_col = chess.row, chess.col
                chess_bak = chessboard.chessboard_map[pos[0]][pos[1]]
                # move the current chess to the new position
                chess.row, chess.col = pos[0], pos[1]
                chessboard.chessboard_map[old_row][old_col] = None
                chessboard.chessboard_map[pos[0]][pos[1]] = chess

                eval = self.alpha_beta(chessboard, depth - 1, alpha, beta, False)

                # restore the old position and old chess on the new position
                chess.row, chess.col = old_row, old_col
                chessboard.chessboard_map[old_row][old_col] = chess
                chessboard.chessboard_map[pos[0]][pos[1]] = chess_bak

                # udpate the alpha value of the max node
                max_eval = max(max_eval, eval)
                alpha = max(alpha, eval)
                # if alpha is greater than beta, then cut the branch
                if beta <= alpha:
                    break
        return max_eval if count_child > 0 else self.evaluate_class.evaluate(chessboard=chessboard)
    else:
        min_eval = float('inf')
        count_child = 0
        for chess in chessboard.get_chess():

            if chess.team == self.team:
                continue

            put_down_position = chessboard.get_put_down_position(chess)
            count_child += len(put_down_position)

            for pos in put_down_position:
                # save the old position and old chess on the new position
                old_row, old_col = chess.row, chess.col
                chess_bak = chessboard.chessboard_map[pos[0]][pos[1]]
                # move the current chess to the new position
                chess.row, chess.col = pos[0], pos[1]
                chessboard.chessboard_map[old_row][old_col] = None
                chessboard.chessboard_map[pos[0]][pos[1]] = chess

                eval = self.alpha_beta(chessboard, depth - 1, alpha, beta, True)

                # restore the old position and old chess on the new position
                chess.row, chess.col = old_row, old_col
                chessboard.chessboard_map[old_row][old_col] = chess
                chessboard.chessboard_map[pos[0]][pos[1]] = chess_bak
                # update the beta value of the min node
                min_eval = min(min_eval, eval)
                beta = min(beta, eval)
                # if beta <= alpha, then cut the branch
                if beta <= alpha:
                    break
        return min_eval if count_child > 0 else self.evaluate_class.evaluate(chessboard=chessboard)
```

这里同时解释我对该算法的简单优化：

我们知道，在不讨论走法的优劣的条件下，象棋的一个局势往往都能派生出数量相当庞大的派生局势，并且让一个局势走向终盘往往需要经过许多步（尤其是在两者实力相当

的情况下）。这将意味着其对应的博弈树将会拥有相当大的宽度与深度，从而程序的计算量也会相当大，相应的程序的运行时间也会相当长。对此我选择通过使用多进程将博弈树的第一层进行分配、然后并行处理，从而达到一定的加速效果。具体实现如下：

```python
max_eval = -float('inf')
next_step = None

chesses_board_list = []
old_pos_list = []
new_pos_list = []

chesses = chessboard.get_chess()
for chess in chesses:

    if chess.team != self.team:
        continue

    for pos in chessboard.get_put_down_position(chess):

        old_pos_list.append((chess.row, chess.col))
        new_pos_list.append((pos[0], pos[1]))

        chessboard_copy = ChessBoard(None)
        chessboard_copy.set_chessboard_str_map(chessboard.get_chessboard_str_map())

        chessboard_copy.chessboard_map[chess.row][chess.col] = None
        chessboard_copy.chessboard_map[pos[0]][pos[1]] = Chess(None, chess.team + '_' + chess.name, pos[0], pos[1])

        chessboard_copy.image = None
        for row in chessboard_copy.chessboard_map:
            for c in row:
                if c:
                    c.image = None

        chesses_board_list.append(chessboard_copy)


with futures.ProcessPoolExecutor(max_workers=32) as executor:

    if len(chesses) <= 8:
        layer = 5
    elif len(chesses) <= 24:
        layer = 4
    else:
        layer = 3

    future_to_chessboard = {executor.submit(
        self.alpha_beta, chessboard_copy, layer, -float('inf'), float('inf'), False)
        : chessboard_copy for chessboard_copy in chesses_board_list}

    for future in futures.as_completed(future_to_chessboard):
        chessboard_copy = future_to_chessboard[future]
        try:
            eval = future.result()
            eval = eval * random.uniform(0.9, 1.1)
            if eval > max_eval:
                max_eval = eval
                index = chesses_board_list.index(chessboard_copy)
                next_step = (*old_pos_list[index], *new_pos_list[index])
        except Exception as exc:
            print('generated an exception: %s' % exc)
```

这里给出修改后代码运行时的 cpu 利用率：

可见 cpu 利用率十分轻易就拉满了，实际的运行时间也减少了约 10 倍左右。

修改 main 函数：

最后修改 main 函数如下：

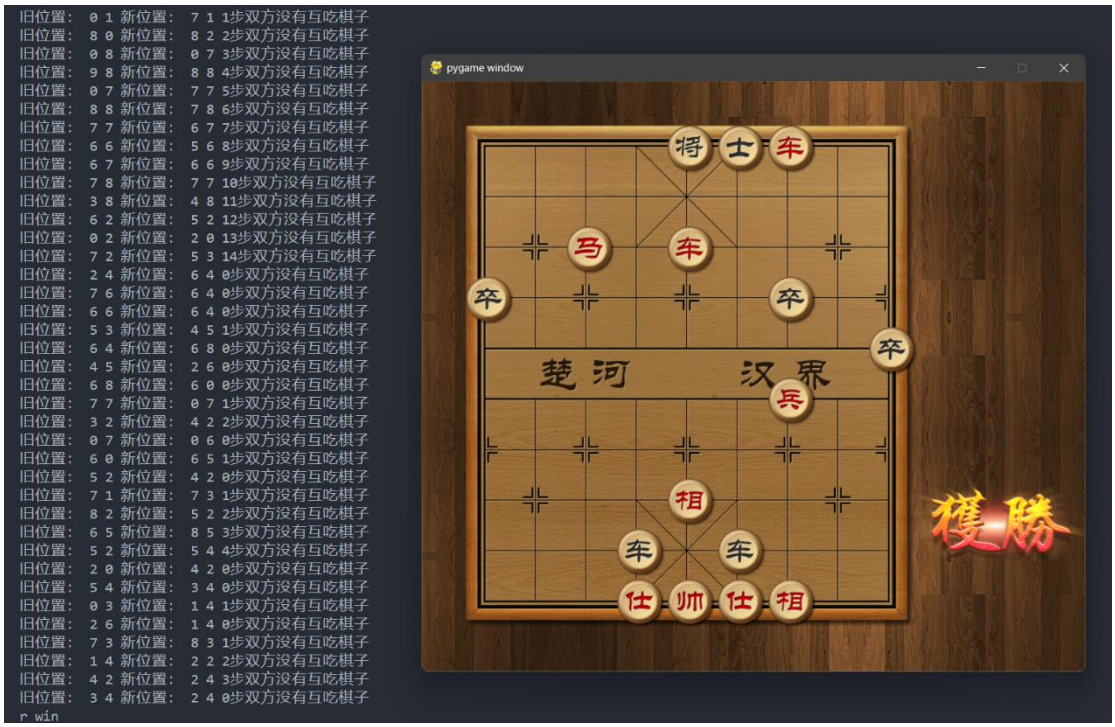
```python
def ai_action(game, ai):

    if not game.get_player() == ai.team:
        return
    if game.show_win or game.show_draw:
        return

    screen = game.screen
    chessboard = game.chessboard
    chessboard_copy = ChessBoard(None)
    chessboard_copy.set_chessboard_str_map(chessboard.get_chessboard_str_map())

    cur_row, cur_col, nxt_row, nxt_col = ai.get_next_step(chessboard_copy)
    ClickBox(screen, cur_row, cur_col)
    chessboard.move_chess(nxt_row, nxt_col)
    ClickBox.clean()

    if chessboard.judge_attack_general(game.get_player()):
        if chessboard.judge_win(game.get_player()):
            print(f"{game.get_player()} win")
            game.set_win(game.get_player())
        else:
            game.set_attack(True)
    else:
        if chessboard.judge_win(game.get_player()):
            print(f"{game.get_player()} win")
            game.set_win(game.get_player())
        game.set_attack(False)

    if chessboard.judge_draw():
        print("draw")
        game.set_draw()

    game.exchange()


def main():
    background_img = pygame.image.load("images/bg.jpg")

    pygame.init()
    screen = pygame.display.set_mode((750, 667))
    chessboard = ChessBoard(screen=screen)
    game = Game(screen=screen, chessboard=chessboard)

    te_ai = ChessAI(game.user_team)
    my_ai = MyAI(game.computer_team)
    # te_ai = ChessAI(game.computer_team)
    # my_ai = MyAI(game.user_team)

    clock = pygame.time.Clock()

    ai_action_thread = None

    while True:

        if not game.show_win and not game.show_draw and game.AI_mode:
            if ai_action_thread is None or not ai_action_thread.is_alive():
                if game.get_player() == my_ai.team:
                    ai_action_thread = threading.Thread(target=ai_action, args=(game, my_ai))
                elif game.get_player() == te_ai.team:
                    ai_action_thread = threading.Thread(target=ai_action, args=(game, te_ai))
                ai_action_thread.start()

        screen.blit(background_img, (0, 0))
        screen.blit(background_img, (0, 270))
        screen.blit(background_img, (0, 540))
        chessboard.show_chessboard_and_chess()
        game.show()
        pygame.display.update()

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                if ai_action_thread is not None:
                    ai_action_thread.join()
                sys.exit()

        clock.tick(60)
```

原程序运行时会出现在 AI 计算下一步时程序窗口卡死的状况，我对此进行了简单的修改：将窗口的更新与 ai 的计算分为两个线程，从而 ai 的计算不会直接影响到窗口的更新。

三、 实验结果及分析

连续两次游戏结果如下：

我方先手（我的 AI 为红方），结果为我方胜利：



在将博弈树深度限制提升到 4 层后，我方 AI 在先手的条件下能大概率战胜对手 AI。

我方后手（我的 AI 为黑方），结果为和棋：

由于奖励值计算函数中只给出了棋子的奖励值和整个棋面的奖励值，这也将意味着在执行到最后的几步中有可能会形成双方不断重复相同的步骤而导致和棋的问题。我尝试在奖励值的计算中加入一定的随机变量，但最终大致都带来了更坏的结果。或许要带来相应的突破，需要在奖励函数中添加更多的规则，这将在今后进行探究。

四、 参考资料

1. 课堂 PPT