کاوه سنایی

۶۱۰۳۹۸۱۳۶

در ابتدا از کاربر تعداد سطر و ستون را میگیریم که محیط بازی چند در چند باشد.
سپس از کاربر حالت بازی را میگیریم که بازی در چه حالتی ران شود که حالت ها شامل موارد زیر است:

- بازیکن مقابل بازیکن
- بازیکن مقابل هوش مصنوعی
- هوش مصنوعی مقابل هوش مصنوعی

تابع هیوریستیک:
یک تابع نوشتیم که یک آرایه ۴ تایی از جدول بازی را گرفته و با توجه به تعداد مهره ها به شکل زیر امتیاز میدهد:

```
def calc_score_for_part_of_borad(part_of_board, piece):
    score = 0
    curr_piece = player_piece1
    if piece == player_piece1:
        curr_piece = player_piece2

    if part_of_board.count(piece) == 4:
        score += 500
    elif part_of_board.count(piece) == 3 and part_of_board.count(empty) == 1:
        score += 50
    elif part_of_board.count(piece) == 2 and part_of_board.count(empty) == 2:
        score += 20

    if part_of_board.count(curr_piece) == 3 and part_of_board.count(empty) == 1:
        score -= 40
```

تابع فوق مقدار هیوریستیک را برای یک آرایه یک بعدی و ۴ تایی برمیگرداند ولی ما در بازی با جدول مهره ها سر و کار داریم بنابراین تمامی ۴ تایی های افقی و عمودی و قطری را بدست آورده و مقدار هیوزیستیک را برای هر کدام محاسبه و جمع میزنیم.

```
def Score_Horizontal(board, piece):
    score = 0
    for r in range(rows_num):
        row_arr = [int(i) for i in list(board[r,:])]
        for c in range(columns_num - 3):
            board_part = row_arr[c:c+WINDOW_LENGTH]
            score += calc_score_for_part_of_borad(board_part, piece)
    return score
```

```
def Score_Vertical(board, piece):
    score = 0
    for c in range(columns_num):
        column_arr = [int(i) for i in list(board[:,c])]
        for r in range(rows_num-3):
            board_part = column_arr[r:r+WINDOW_LENGTH]
            score += calc_score_for_part_of_borad(board_part, piece)
    return score

def Score_diagonal(board, piece):
    score = 0
    for r in range(rows_num-3):
        for c in range(columns_num-3):
            board_part = [board[r+i][c+i] for i in range(WINDOW_LENGTH)]
            score += calc_score_for_part_of_borad(board_part, piece)
    for r in range(rows_num-3):
        for c in range(columns_num-3):
            board_part = [board[r+3-i][c+i] for i in range(WINDOW_LENGTH)]
            score += calc_score_for_part_of_borad(board_part, piece)
    return score
```

از میان ستون های جدول ستون های وسطی به دلیل اینکه امکان بیشتری برای براورد کردن شرط برد بازی فراهم میکنند از اهمیت بیشتری برخوردار هستند بنابراین باید وزن آن ها در تعیین مقدار تابع هیوریستیک بیشتر باشد در همین راستا تابع زیر را تعریف میکنیم.

```
def Score_center_columns(board, piece):
    score = 0
    center = [int(i) for i in list(board[:, columns_num//2])]
    center_numbers = center.count(piece)
    score += center_numbers * 3
    return score
```

سپس مقادیر فوق را با استفاده از تابع زیر جمع میکنیم:

```
def calc_score(board, piece):
    score = 0
    score += Score_center_columns(board, piece)
    score += Score_Horizontal(board, piece)
    score += Score_Vertical(board, piece)
    score += Score_diagonal(board, piece)
    return score
```

توضیح الگوریتم minmax :
این الگوریتم را به صورت بازگشتی مینویسیم در ورودی صفحه بازی، آلفا،بتا ، عمق و یک مقدار بولی دریافت میکند که این مقدار تعیین میکند در سطح فعلی درخت باید مقادیر را کمینه کنیم یا بیشینه.

```
def minimax(board, depth, alpha, beta, maximizingPlayer):
    valid_locations = find_valid_locations(board)
    is_leaf = check_win(board)
    if depth == 0 or is_leaf:
        if is_leaf:
            if win_conditions(board, player_piece2):
                return (None, 100000000000000)
            elif win_conditions(board, player_piece1):
                return (None, -10000000000000)
            else:
                return (None, 0)
        else:
            return (None, calc_score(board, player_piece2))
    if maximizingPlayer:
        value = -math.inf
        column = random.choice(valid_locations)
        for col in valid_locations:
            row = find_next_empty_row(board, col)
            b_copy = board.copy()
            make_a_move(b_copy, row, col, player_piece2)
            new_score = minimax(b_copy, depth-1, alpha, beta, False)[1]
            if new_score > value:
                value = new_score
                column = col
            alpha = max(alpha, value)
            if alpha >= beta:
                break
        return column, value

    else:
        value = math.inf
        column = random.choice(valid_locations)
        for col in valid_locations:
            row = find_next_empty_row(board, col)
            b_copy = board.copy()
            make_a_move(b_copy, row, col, player_piece1)
            new_score = minimax(b_copy, depth-1, alpha, beta, True)[1]
            if new_score < value:
                value = new_score
                column = col
            beta = min(beta, value)
```

```
            if alpha >= beta:
                break
        return column, value
```

اول الگوریتم شرط خروج را بررسی میکنیم،  الگوریتم زمانی فراخوانی خودش را تمام میکند که صفحه بدست آمده شرط اتمام بازی را داشته باشد. اگر عمق داده شده صفر باشد بدیهی است که به جستوجو در درخت ادامه نمیدهد اگر شرط خروج الگوریتم براورده نشود ستون هایی که فضای خالی دارند را پیدا میکنیم سپس مهره اضافه کرده و الگوریتم را بطور بازگشتی فراخوانی میکنیم با توجه به مقدار بولی آلفا و یا بتا را به روز رسانی کرده و متناظر با به روز رسانی خود پرونینگ مناسب را انجام میدهیم.

بقیه کد هم پیاده سازی محیط گرافیکی بازی و قوانین آن می باشد.