VESTACKA INTELIGENCIJA
3. FAZA PROJEKTA

SLAGANJE(BYTE)

Naziv tima : DavidDusanVeljko
Dušan Stojanovic 17450
Veljko Marković 17745
David Stanisavljević 17943

alpha-beta

```python
def min_max_alpha_beta(self, depth, maximizing_player, alpha, beta):
    key = self.cache_key()
    if key in self.cache:
        return self.cache[key]

    if depth == 0 or self.is_over():
        return self.evaluate_state(), None

    if maximizing_player:
        max_eval = float('-inf')
        best_move = None
        moves = self.generate_all_moves()
        self.curr_moves=moves

        for move in moves:
            new_game = self.copy_game()
            new_game.execute_move(move)
            eval, _ = new_game.min_max_alpha_beta(depth - 1, False, alpha, beta)
            if eval > max_eval:
                max_eval = eval
                best_move = move
            alpha = max(alpha, max_eval)
            if beta <= alpha:
                break
        result = max_eval, best_move
        self.cache[key] = result
        return result
    else:
        min_eval = float('inf')
        best_move = None
        moves = self.generate_all_moves()

        for move in moves:
            new_game = self.copy_game()
            new_game.execute_move(move)
            eval, _ = new_game.min_max_alpha_beta(depth - 1, True, alpha, beta)
            if eval < min_eval:
                min_eval = eval
                best_move = move
            beta = min(beta, min_eval)
            if beta <= alpha:
                break
        result = min_eval, best_move
        self.cache[key] = result
        return result
```

```python
def cache_key(self):
    move_history=tuple(self.move_history)
    return (
        self.current_player.checker_color,
        self.player1.score,
        self.player2.score,
        move_history,
    )
```

evaluacija

```python
def evaluate_state(self):
    score=0
    curr=self.current_player
    other=self.get_other_player()
    #broj mogucih poteza
    c_count=len(self.curr_moves)
    p_count=len(self.generate_all_moves(other.checker_color))
    if c_count>p_count:
        score+=1
    elif p_count<p_count:
        score-=1
    #ko poseduje vise stackova(prvi element stack-a ai ili player)
    total,c_checkers,p_checkers,c_stacks,p_stacks=self.get_counts_stack()
    if c_stacks>p_stacks:
        score+=1
    elif c_stacks<p_stacks:
        score-=1
    #ko poseduje veci stepen kontrole u stackovima tj. ko u tim stackovima ima ukupno veci broj figura
    c_degree, p_degree = self.calculate_degree_of_stack_control((total,c_checkers,p_checkers))
    if c_degree>p_degree:
        score+=1
    elif c_degree<p_degree:
        score-=1
    #ko ima vise poena kada se napravi potez terminator ili covek
    c_score=curr.score
    p_score=other.score
    if c_score>p_score:
        score+=1
    elif c_score<p_score:
        score-=1

    return score
```

```python
def get_counts_stack(self):
    color=self.current_player.checker_color
    stack_count = 0
    match = 0
    not_match=0
    curr_stacks=0
    other_stacks=0

    for row in range(self.board.num_of_fields):
        for col in range(self.board.num_of_fields):
            field = self.board.fields[row][col]

            if field.field_type == field_black and not field.is_empty() and len(field.stack) > 1:
                stack_count += 1
                for checker in field.stack:
                    if checker==color:
                        match+=1
                    else:
                        not_match+=1
                    if field.stack[0] == color:
                        curr_stacks+=1
                    else:
                        other_stacks+=1


    return stack_count,match,not_match, curr_stacks, other_stacks

def calculate_degree_of_stack_control(self,arg):
    total_stacks, color_count, not_color_count = arg

    if total_stacks == 0:
        return 0.0, 0.0

    stack_control = color_count / total_stacks
    not_stack_control = not_color_count / total_stacks

    return stack_control,not_stack_control
```

ai move

```python
def make_ai_move(self):
    depth = DEPTH
    best_score = float('-inf')
    best_move = None

    moves = self.generate_all_moves()

    for move in moves:
        new_game = self.copy_game()
        new_game.execute_move(move)
        score, _ = new_game.min_max_alpha_beta(depth , False, float('-inf'), float('inf'))

        if score > best_score:
            best_score = score
            best_move = move

    print(best_move)
    self.execute_move(best_move)
```