پروژه هوش مصنوعی و سیستم های خبره

شایان کبریتی - 400243065

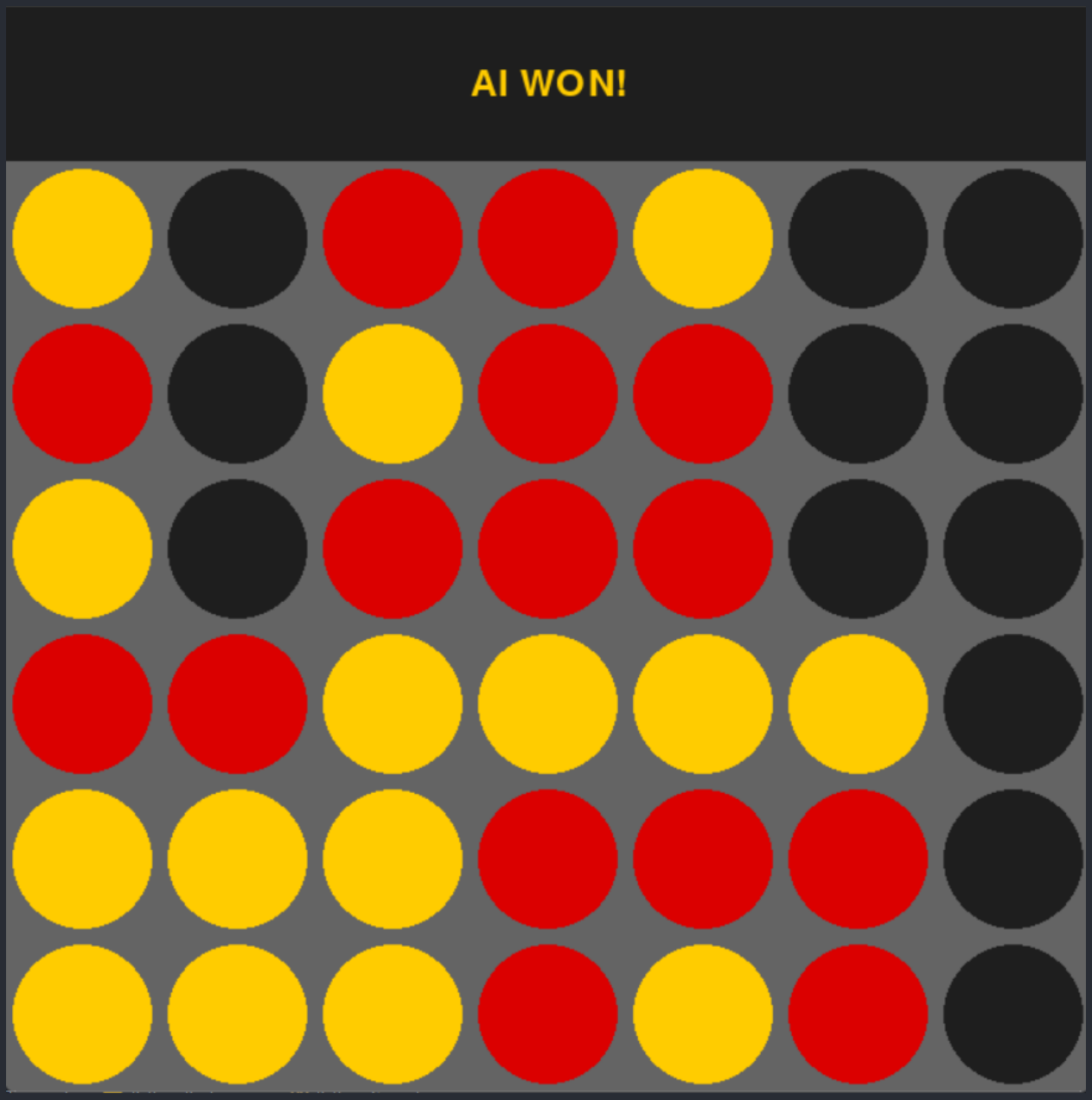
فهرست مطالب

نمایی از بازی

- منو:

در منوی بازی تنظیمات بازی شامل لول بندی و نوع ایجنت مورد نظر قابل تنظیم است. بعد از آن سه گزینه که به ترتیب بازی با ایجنت (شروع با شما و شروع با ایجنت) و اطلاعات سازنده می باشد، مشاهده می شود.

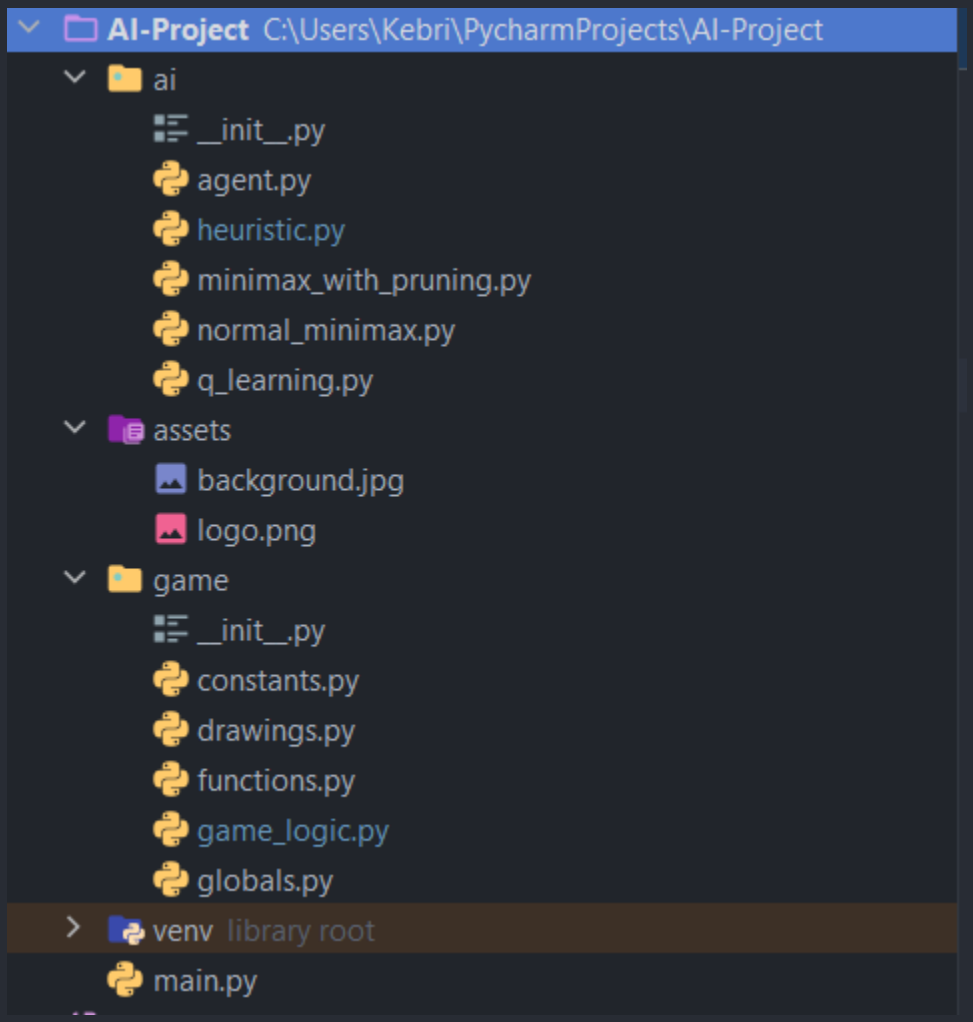
- **صفحه بازی:**



نمونه ای از بازی انجام شده با ایجنت AlphaBeta با لول Medium

جزئیات پیاده سازی بازی

- **ساختار پروژه:**



- **بخش پیاده سازی بازی (game):**

این بخش شامل موارد زیر می باشد:

- Constants: متغیر های ثابت مثل اندازه صفحه، سطر و ستون بازی و ...
- Drawings: توابع مربوط به رسم گرافیک ها در بازی
- Functions: توابع منطقی پیاده شده مربوط به بازی و صفحه آن. شرح برخی از توابع اصلی آن:

```
def put_piece(board, col, piece): It puts piece in column col of the given board
def is_valid_move(board, col): checks if we can put piece in column col
def next_available_row(board, col): returns the first row that in the given column has empty piece
def is_a_win(board, piece): checks if win has been happened
def get_available_moves(board): returns columns we can put our piece into
def game_state(board): returns games state (RED_WON, YELLOW_WON, DRAW, PENDING)
```

- Game_logic: منطق اصلی بازی و جا به جایی بین وضعیت ها و صفحات در این صفحه هندل می شود.
- Globals: متغیر هایی که در سرتاسر پکیج استفاده و مورد تغییر واقع می شوند

- **بخش فایل های جانبی (assets):**

در این بخش فایل های جانبی مثل عکس ها برای زیبایی بصری نگهداری می شوند.

جزئیات پیاده سازی ایجنت (AI)

- **بخش هندل و کانفیگ کردن تنظیمات ایجنت (agent.py):**

```
def run():
    if globals.ALGORITHM == NORMAL_MINIMAX:
        if globals.LEVEL == EASY:
            globals.MINIMAX_DEPTH = 3
        elif globals.LEVEL == MEDIUM:
            globals.MINIMAX_DEPTH = 6
        elif globals.LEVEL == HARD:
            globals.MINIMAX_DEPTH = 7
        return normal_minimax_with_transposition_table(copy.deepcopy(game.globals.board))
        # return normal_minimax(copy.deepcopy(game.globals.board))

    elif globals.ALGORITHM == ALPHABETA_MINIMAX:
        if globals.LEVEL == EASY:
            globals.MINIMAX_DEPTH = 3
        elif globals.LEVEL == MEDIUM:
            globals.MINIMAX_DEPTH = 6
        elif globals.LEVEL == HARD:
            globals.MINIMAX_DEPTH = 7
        return alpha_beta_minimax(copy.deepcopy(game.globals.board))

    elif globals.ALGORITHM == Q_LEARNING:
        iteration_count = 0
        if globals.LEVEL == EASY:
            iteration_count = 1e3
        elif globals.LEVEL == MEDIUM:
            iteration_count = 1e4
        elif globals.LEVEL == HARD:
            iteration_count = 1e6
        if not q_agent.is_trained:
            q_agent.train(iteration_count=int(iteration_count))
        return q_agent.get_action(copy.deepcopy(game.globals.board)), 0
```

شاید یکی از مورد بحث ترین قسمت های پروژه همین بخش باشد. چرا که در این بخش عمق درخت Minimax و تعداد iteration های Q learning انتخاب و تنظیم می شوند.
بدیهی است که هر چه عمق درخت بیشتر باشد، ایجنت دقیق تر عمل خواهد کرد.
همچنین بدیهی است که افزایش عمق درخت باعث ایجاد سربار زیاد و کند شدن ایجنت

می شود. به همین دلیل، عمقی ایده آل می دانیم که هم از لحاظ سرعت و هم از لحاظ دقت مطلوب ما باشد (تعادل بین این دو حفظ شود).
طبق کد عمق ایده آل ما همان عمق داده شده به لول MEDIUM است (عمق HARD دقیق اما کند و عمق EASY سریع اما نا دقیق است). همانطور که مشخص است لول بندی ها بر اساس عمق درخت تقسیم بندی شده اند. این موارد در مورد iteration ایجنت Q Learning نیز صادق است.

- **پیاده سازی minimax عادی:**

این فایل دارای دو متود است:

- Normal_minimax: تابع اول minimax عادی را طبق روال تدریس شده پیاده می کند(عمق ایده آل این متود 3 و 4 است). حال از آنجایی که این متود به تنهایی کند است، تصمیم به استفاده از Dynamic Programming برای متود بعدی کرده ایم.
- Normal_minimax_with_transposition_table: تفاوت این متود با متود قبلی داشتن پارامتر transposition_table است. این پارامتر حالت هایی که خاتمه میابند را ذخیره می کند تا در دفعات بعد که تابع بازگشتی صدا زده شد، حالات تکراری که قبلا اجرا شده اند، دوباره اجرا نشوند و بجای آن از transposition_table مقدار نهایی آن ها برداشته شود. (عمق ایده آل این متود 6 و 7 است).

- **پیاده سازی minimax با هرس آلفا بتا:**

تنها متود این فایل alpha_beta_minimax هست. که تفاوت آن با توابع قبلی داشتن پارامتر آلفا و بتا و داشتن شرط هرس کردن در دل متود است. (عمق ایده آل این متود 6 و 7 است).

- پیاده سازی توابع هیوریستیک:

منطق اصلی پیاده سازی هیوریستیک بازی، اولویت دادن به ستون های وسط و البته چهار تایی هایی از خانه هاست که با کمترین حرکت ما میتوان یک چهارتایی درست کرد.

- Score_line: به چهارتایی که داده میشود امتیاز میدهد

```
def score_line(line, piece):
    opponent_piece = opponent(piece)
    score = 0

    if line.count(piece) == 3 and line.count(EMPTY_PIECE) == 1:
        score += 10
    elif line.count(piece) == 2 and line.count(EMPTY_PIECE) == 2:
        score += 5
    elif line.count(opponent_piece) == 3 and line.count(EMPTY_PIECE) == 1:
        score -= 10

    return score
```

- توابع زیر مربوط به امتیاز دهی به چهار تایی های افقی و عمودی هستند:

```
def get_col(board, col):
    return [row[col] for row in board]

def vertical_line(board, r, c):
    return [board[r][c] for r in range(r, r + 4)]

def score_horizontal_lines(board, piece):
    score = 0
    for row in range(ROW_COUNT):
        for col in range(COLUMN_COUNT - 3):
            line = board[row][col:col + 4]
            score += score_line(line, piece)
    return score

def score_vertical_lines(board, piece):
    score = 0
    for row in range(ROW_COUNT - 3):
        for col in range(COLUMN_COUNT):
            score += score_line(vertical_line(board, row, col), piece)
    return score
```

- این توابع صرفا برای دادن یک امتیاز مازاد بر امتیاز توابع بالا، برای اولویت دادن به ستون وسط هستند:

```python
def score_column(board, piece, col, score):
    column = get_col(board, col)
    center_count = column.count(piece)
    return center_count * score

def score_center_columns(board, piece):
    center_column = COLUMN_COUNT // 2
    return score_column(board, piece, center_column, 5) + score_column(board, piece, center_column-1, 2) + score_column(board, pi
```

- تابع زیر، تابع اصلی هیوریستیک است که با توابع minimax ما با آن کار خواهد کرد. این تابع از ترکیبی از توابع بالا استفاده می کند:

```python
def score_state(board, piece):
    return score_center_columns(board, piece) + score_vertical_lines(board, piece) + score_horizontal_lines(board, piece)
```

- پیاده سازی ایجنت با Q-Learning:

پیاده سازی این بخش از این شبه کد الهام گرفته شده است:

Data: A: Action state set
Data: S: Agent state set
Data: T a number of iterations or the end of the Game
Data: Hyper parameters : α, γ, ϵ
Data: α learning rate
Data: γ discount factor
Data: ϵ probability of exploration
Result: Q

Initialize $Q(s, a) = 0$, for all $s \in S$, $a \in A$
Initialize s_t
while $t < T$ **do**
    Choose a_t thanks to the ϵ-greedy policy $(a_t = \pi(s_t, \epsilon, rnd))$
    Take action a_t and observe the reward r_t and new sate s_{t+1}
    Update $Q(s_t, a_t)$ depending on α, γ and s_{t+1}
    $s_t \leftarrow s_{t+1}$
    $t \leftarrow t + 1$
end

Algorithm 4: Q-learning for estimating π

- طبق شبه کد بالا فیلد های زیر را خواهیم داشت:

```python
class QLearningAgent:
    def __init__(self):
        self.q_table = {}
        self.is_trained = False
        self.epsilon = 0.5  # Exploration rate
        self.alpha = 0.5  # Learning rate
        self.gamma = 0.5  # Discount factor
```

- تابع update_q:

```python
def update_q(self, state, action, next_state, reward):
    state_tuple = tuple(map(tuple, state))
    next_state_tuple = tuple(map(tuple, next_state))

    current_q = self.q_table.get((state_tuple, action), 0)

    max_next_q = max(self.q_table.get((next_state_tuple, next_action), 0) for next_action in range(COLUMN_COUNT))

    new_q = current_q + self.alpha * (reward + self.gamma * max_next_q - current_q)

    self.q_table[(state_tuple, action)] = new_q
```

این تابع طبق تعریف زیر نوشته شده است:

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

where r_t is the reward received when moving from the state s_t to the state s_{t+1}, and α is the learning rate $0 < \alpha \leq 1$).

- حال با استفاده از تابع بالا، بدنه اصلی بخش train و یادگیری ایجنت نوشته می شود:

```
def train(self, iteration_count):
    self.is_trained = False
    board = create_board()

    for i in range(iteration_count):
        while game_state(board) == globals.PENDING:
            action = self.get_action(board)

            if not is_valid_move(board, action):
                continue

            before_board = copy.deepcopy(board)
            put_piece(board, action, globals.AI_PIECE)

            if is_a_win(board, globals.AI_PIECE):
                reward = 1000
            elif is_a_win(board, globals.PLAYER_PIECE):
                reward = -1000
            else:
                reward = 0

            self.update_q(before_board, action, board, reward)

        board = create_board()
        globals.turn = toggled_turn(globals.turn)
    self.is_trained = True
```

برای برد ایجنت، ریوارد 1000 تایی و برای باخت آن جریمه 1000 تایی در نظر گرفته شده است. همچنین اگر ایجنت به حالت تساوی برسد، به آن ریواردی نخواهیم داد (خنثی). طبق همین ریوارد ها و تابع update_q، جدول حرکت های ایجنت (Q-Table) پر و یا train می شود. در نهایت با استفاده از تابع get_action، طبق Q-Table پر شده، بهترین حرکت از نظر ایجنت انتخاب می شود. همچنین برای افزایش دقت ایجنت به تعداد iteration_count، ایجنت train می شود. همچنین ایجنت قبل از اجرای اولین حرکت train می شود، به همین علت اولین حرکت ایجنت، کند خواهد بود.

منابع

منبع 1
منبع 2