



دانشکده مهندسی و علوم کامپیوتر

گزارش فاز اول پروژه هوش مصنوعی و سیستم

های خبره – دکتر سلیمی بدر

پاییز ۱۴۰۳

گردآورندگان: آراس ولیزاده – امیرسامان رستم‌پیگی

داخل این کد به عامل هوشمند برای بازی “Hand of the King” طراحی کردیم که از الگوریتم مینیمکس با هرس آلفا-بتا استفاده می‌کند تا بهترین حرکت رو برای بازیکن پیدا کند.

عامل هوشمند چیه؟

عامل هوشمند به برنامه یا کد کامپیوتریه که می‌تونه وضعیت بازی رو تحلیل کند و بر اساس استراتژی‌های تعریف شده، بهترین حرکت ممکن رو انتخاب کند. اینجا عامل ما:

- حرکت‌های ممکن رو بررسی می‌کند.
- امتیاز وضعیت‌های مختلف بازی رو می‌سنجه.
- حرکتی رو انتخاب می‌کند که بیشترین احتمال برد رو داشته باشه.

الگوریتم مینیمکس چیه؟

الگوریتم مینیمکس یکی از روش‌های استاندارد برای حل مسائل بازی‌های نوبتیه، مثل شطرنج یا این بازی. توی این الگوریتم:

- بازیکن ما (عامل هوشمند) تلاش می‌کند امتیاز رو ماکزیمم کند.
- حریف تلاش می‌کند امتیاز رو مینیمم کند.
- مینیمکس این تعامل رو شبیه‌سازی می‌کند و وضعیت‌هایی که هر بازیکن می‌تونه انتخاب کند رو بررسی می‌کند.

هرس آلفا-بتا چیه؟

هرس آلفا-بتا یه بهبود برای الگوریتم مینیمکسه که باعث می‌شه شاخه‌هایی که نتیجه‌ی قطعی دارن، بررسی نشن. این کار دو تا مزیت داره:

- **سرعت بیشتر:** شاخه‌های بی‌فایده حذف می‌شن.
- **بهره‌وری بالاتر:** فقط وضعیت‌هایی که اهمیت دارن بررسی می‌شن.

مثلاً اگه توی یه بازی وضعیت A امتیاز بهتری نسبت به وضعیت B داشته باشه، نیازی نیست بقیه حرکت‌های B رو بررسی کنیم.

چرا مینیمکس اینجا خوب کار می‌کنه؟

- بازی "Hand of the King" یه بازی نوبتیه که حرکت‌های ممکنش قابل پیش‌بینیه.
- عامل هوشمند می‌تونه حرکت‌های بعدی خودش و حریف رو شبیه‌سازی کنه.
- هدف اینه که هر حرکتی که انتخاب می‌کنه، بیشترین امتیاز رو در آینده براش بیاره.

حالا میتونیم بریم سراغ اینکه بخش‌های مختلف کد رو توضیح بدیم:

```
def find_varys(cards):  
    """  
    Finds the location of Varys on the board  
    d.  
    Parameters:  
        cards (list): list of Card objects  
  
    Returns:  
        varys_location (int): location of Vary  
    s    """  
    varys = [card for card in cards if card.get_name() == 'Varys']  
    return varys[0].get_location()
```

پیدا کردن مکان واریس (تابع find_varys)

این تابع توی لیست کارت‌ها می‌گرده و کارت "Varys" رو پیدا می‌کنه. بعد مکانش رو (که یه عدد و نشون‌دهنده جای کارت توی برده) برمی‌گردونه. از این برای مشخص کردن حرکت‌های معتبر استفاده می‌کنیم.

```

def get_valid_moves(cards):
    """
    Gets the possible moves for the player.

    Parameters:
        cards (list): list of Card objects

    Returns:
        moves (list): list of possible move
    """
    vars_location = find_vars(cards)
    vars_row, vars_col = vars_location // 6, vars_location % 6
    moves = []

    for card in cards:
        if card.get_name() == 'Vars':
            continue

        row, col = card.get_location() // 6, card.get_location() % 6
        if row == vars_row or col == vars_col:
            moves.append(card.get_location())

    return moves

```

پیدا کردن حرکتهای معتبر (تابع `get_valid_moves`)

این تابع مشخص می‌کند واریس تو کدام جهت‌ها (راست، چپ، بالا یا پایین) می‌تونه حرکت کنه:

- مکان واریس رو می‌گیره.
- کارتهایی که توی همون ردیف یا ستون هستن رو پیدا می‌کنه.
- به لیست از مکان‌های ممکن رو برمی‌گردونه.

```

1 def evaluate_state(cards, player1, player2):
2     """
3     Calculates a heuristic score for the current game state by evaluating the banners
4     captured by both players and the number of cards they control.
5
6     Parameters:
7         cards (list): A list of `Card` objects representing the current game state.
8         player1 (Player): An object representing Player 1, including banners and cards.
9         player2 (Player): An object representing Player 2, including banners and cards.
10
11     Returns:
12         int: A score indicating the favorability of the game state for Player 1.
13         Positive scores favor Player 1; negative scores favor Player 2.
14     """
15     banner_weights = {
16         'Stark': 9, 'Greyjoy': 7, 'Lannister': 7, 'Targaryen': 6, 'Baratheon': 5,
17         'Tyrell': 9, 'Tully': 10
18     }
19     banner_cards = {
20         'Stark': 8, 'Greyjoy': 7, 'Lannister': 6, 'Targaryen': 5, 'Baratheon': 4,
21         'Tyrell': 3, 'Tully': 2
22     }
23
24     score = 0
25
26     # calculate score for player 1
27     for house in player1.get_banners():
28         if player1.get_banners()[house]:
29             score += banner_weights[house] * 10
30             num_cards = 0
31             for card in player1.get_cards():
32                 if card == house:
33                     num_cards += 1
34             if num_cards <= banner_cards[house] // 2 + 1:
35                 score += num_cards * 2
36
37     # Calculate score for Player 2
38     for house in player2.get_banners():
39         if player2.get_banners()[house]:
40             score -= banner_weights[house] * 10
41             num_cards = 0
42             for card in player2.get_cards():
43                 if card == house:
44                     num_cards += 1
45             if num_cards > banner_cards[house] // 2 + 1:
46                 penalty = -(banner_cards[house] // 2)
47             else:
48                 penalty = num_cards
49             score -= penalty * 2
50
51     player1_moves = len(get_player_moves(player1.get_cards(), cards))
52     player2_moves = len(get_player_moves(player2.get_cards(), cards))
53     score += (player1_moves - player2_moves) * 3 # Reward more move options
54
55     return score

```

ارزیابی وضعیت بازی (تابع evaluate_state)

این تابع واقعا قلب تصمیم‌گیری ایجنت ماست و کل عملکردش به این وابسته‌ست که بازی رو چجوری ارزیابی کنه. هدفش اینه که به هر وضعیت بازی یه امتیاز بده که این امتیاز مشخص کنه بازی چقدر به نفع بازیکن اول (عامل هوشمند ما) هست.

چطوری کار می‌کنه؟

1. امتیاز بنرها:

وقتی یه بازیکن بتونه یه خاندان رو کامل کنه (یعنی کارتهای لازم اون خاندان رو جمع کنه)، بنر اون خاندان رو می‌گیره. اینجا تابع به هر بنر یه امتیاز خاص می‌ده که بر اساس ارزش خاندانه. مثلا خاندان Stark امتیاز بالاتری داره چون تعداد کارتهای بیشتری داره، در حالی که Tully با دو کارت تکمیل می‌شه ولی امتیاز کمتری داره.

- بازیکن اول بنر بگیره؟ امتیاز مثبت می‌گیره.
- بازیکن دوم بنر بگیره؟ امتیاز منفی برای بازیکن اول ثبت می‌شه.

2. امتیاز کارت‌ها:

- اینجا تابع نگاه می‌کنه که هر بازیکن چندتا کارت از یه خاندان خاص داره.
- اگر کارت‌های بازیکن اول به تعداد کافی برسه که بنر رو بگیره، یه امتیاز اضافی می‌گیره.
 - اگر تعداد کارت‌هایش کمتر از نصف کارت‌های اون خاندان باشه، امتیاز کمتری می‌گیره، چون تو رقابت ممکنه عقب بمونه.

3. حرکتهای ممکن:

تابع تعداد حرکتهای ممکن هر بازیکن رو هم چک می‌کنه. اگه بازیکن اول حرکتهای بیشتری داشته باشه، امتیاز مثبت می‌گیره. ایده اینه که بازیکنی که گزینه‌های بیشتری داره، احتمالا کنترل بیشتری رو بازی داره.

چرا مهمه؟

این تابع باعث می‌شه ایجنت ما بدون کدوم وضعیت بازی براش بهتره. وقتی الگوریتم Minimax می‌خواد حرکت‌های مختلف رو امتحان کنه، از این تابع برای مقایسه وضعیت‌ها استفاده می‌کنه. اگه این ارزیابی درست کار کنه، ایجنت می‌تونه تصمیم‌های بهتری بگیره.

یه نکته باحال (:

اینجا ما به بنرهای کامل شده وزن بیشتری دادیم چون هدف نهایی بازی همین تصاحب بنرهاست. اما تعداد کارت‌ها و تعداد حرکت‌ها هم تاثیر داره که ایجنت تو مسیر درست حرکت کنه و شانس برای برد به حریف نده.

به زیون ساده: این تابع مثل یه متر کار می‌کنه که وضعیت بازی رو می‌سنجه و می‌گه کدوم بازیکن جلوتره و چقدر جلوئه.

```
def simulate_move(cards, player1, player2, move, player):
    """
    Simulates a player's move and returns the resulting game state.

    Parameters:
        cards (list): A list of `Card` objects representing the current game state.
        player1 (Player): The Player 1 object including banners and cards.
        player2 (Player): The Player 2 object including banners and cards.
        move (int): The index of the card to move
        player (int): The player making the move (1 for Player 1, 2 for Player 2).

    Returns:
        tuple: A tuple containing
            - list: A deep copy of the updated `cards` after the move.
            - Player: A deep copy of the updated `player1` object.
            - Player: A deep copy of the updated `player2` object.
    """
    new_cards, new_player1, new_player2 = map(
        copy.deepcopy, [cards, player1, player2])
    current_player = new_player1 if player == 1 else new_player2
    selected_house = make_move(new_cards, move, current_player)
    set_banners(new_player1, new_player2, selected_house, player)
    return new_cards, new_player1, new_player2
```

شبیه‌سازی حرکت (تابع simulate_move)

خب این تابع به جورایی مثل یک ماشین زمان می‌مونه! وقتی به حرکت رو انجام بدیم، این تابع بازی رو جلو می‌بره و وضعیت جدید رو بهمون نشون می‌ده، بدون اینکه واقعاً وضعیت اصلی بازی رو تغییر بده. از اینجا به بعد، ایجنت می‌تونه این وضعیت شبیه‌سازی‌شده رو بررسی کنه و ببینه این حرکت خوبه یا نه.

- اول از همه، وضعیت فعلی بازی شامل کارتها، بازیکن‌ها و هر چیزی که تو بازی هست رو کپی می‌کنه. این کپی باعث می‌شه که وضعیت اصلی بازی دست‌نخورده بمونه.
- بعدش، حرکت موردنظر رو انجام می‌ده. این یعنی مثلاً Varys رو از جایی به جای دیگه می‌بره و کارت مربوطه رو می‌گیره.
- توی این حرکت، ممکنه بازیکن به خاندان رو کامل کنه. وقتی این اتفاق بیفته، پرچم (بنر) اون خاندان به بازیکن داده می‌شه. برای همین، تابع set_banners هم صدا زده می‌شه تا وضعیت پرچم‌ها رو آپدیت کنه.
- در نهایت، وضعیت جدید بازی شامل کارتها و اطلاعات بازیکن‌ها (مثل کارتهای جمع‌شده و پرچم‌ها) رو برمی‌گردونه.
- نتیجه رو به صورت وضعیت شبیه‌سازی‌شده برمی‌گردونه.

```

def minimax(cards, player1, player2, depth, is_maximizing, alpha=-inf, beta=inf,
no_heuristic=False):
    """
    Uses the Minimax algorithm with Alpha-Beta pruning and optional deep search
    to determine the best move for a given game state.

    Parameters:
        cards (list): Represents the current game board as `Card` objects.
        player1 (Player): Player 1's state including banners and card
        player2 (Player): Player 2's state including banners and card
        depth (int): Depth limit for game tree exploratio
        is_maximizing (bool): Indicates if the current player is maximizing the scor
        e.
        alpha (float): Alpha value for pruning (best score for maximizing playe
        r).
        beta (float): Beta value for pruning (best score for minimizing playe
        r).
        no_heuristic (bool, optional): Flag for additional end-game evaluation when
        no moves remain.

    Returns:
        tuple: Contain
            - int: The highest or lowest score achievable from this state.
            - int or None: The index of the optimal move, or None if no valid moves
        are available.
    """
    valid_moves = get_valid_moves(cards)

    if depth == 0 or len(valid_moves) == 0:
        if no_heuristic:
            if calculate_winner(player1, player2) == 1:
                winner_score = 2000
            else:
                winner_score = -2000
            return (winner_score, None)
        else:
            return (evaluate_state(cards, player1, player2), None)

    optimal_move = None
    if is_maximizing:
        max_eval = -inf
        for move in valid_moves:
            next_cards, next_player1, next_player2 = simulate_move
(cards, player1, player2, move, player=1)
            current_eval, _ = minimax(next_cards, next_player1, next_player2, depth
- 1, False, alpha, beta, no_heuristic)
            if current_eval > max_eval:
                max_eval = current_eval
                optimal_move = move
            alpha = max(alpha, current_eval)
            if beta <= alpha:
                break # Stop exploring this branch
        return max_eval, optimal_move
    else:
        min_eval = inf
        for move in valid_moves:
            next_cards, next_player1, next_player2 = simulate_move
(cards, player1, player2, move, player=2)
            current_eval, _ = minimax(next_cards, next_player1, next_player2, depth
- 1, True, alpha, beta, no_heuristic)
            if current_eval < min_eval:
                min_eval = current_eval
                optimal_move = move
            beta = min(beta, current_eval)
            if beta <= alpha:
                break # Stop exploring this branch
        return min_eval, optimal_move

```


الگوریتم مینیمکس با هرس آلفا-بتا (تابع minimax)

این تابع از یه الگوریتم جستجو استفاده می‌کنه تا بهترین حرکت رو پیدا کنه:

1. پایه‌ی بازگشتی:

اول از همه، بررسی می‌کنه که عمق جستجو تموم شده یا حرکت دیگه‌ای باقی نمونده. اگه تموم شده باشه، وضعیت فعلی بازی رو با تابع `evaluate_state` امتیازدهی می‌کنه و برمی‌گردونه.

2. اگه نوبت بازیکن ما (Maximizing) باشه:

- تابع همه حرکت‌های ممکن رو بررسی می‌کنه.
- هر حرکت رو با تابع `simulate_move` شبیه‌سازی می‌کنه تا وضعیت جدید بازی رو به‌دست بیاره.
- بعد، به‌صورت بازگشتی Minimax رو برای حریف (Minimizing) صدا می‌زنه.
- بهترین امتیاز رو نگه می‌داره و بقیه رو دور می‌ریزه.
- از هرس آلفا-بتا استفاده می‌کنه تا حرکت‌هایی که بررسی‌شون بی‌فایده‌ست رو رد کنه و سرعتش بره بالا.

3. اگه نوبت حریف (Minimizing) باشه:

- دقیقاً مثل مرحله بالا عمل می‌کنه، ولی هدفش اینه که بدترین امتیاز ممکن رو برای بازیکن ما انتخاب کنه.
- باز هم با هرس آلفا-بتا، حرکت‌های غیرضروری رو رد می‌کنه.

4. برگشت:

- بعد از بررسی همه حرکت‌ها، تابع بهترین امتیاز و حرکت مربوط به اون امتیاز رو برمی‌گردونه.

```

def get_move(cards, player1, player2):
    """
    Determines the best move for Player 1 using the Minimax algorithm with Alpha-Beta pruning.

    Parameters:
        cards (list): A list of `Card` objects representing the current game state.
        player1 (Player): The Player 1 object including banners and cards.
        player2 (Player): The Player 2 object including banners and cards.

    Returns:
        int or None: The index of the best move for Player 1, or None if no moves are available.
    """
    # if there is no limit in time:

    if len(cards) < 25:
        depth = 9
    else:
        depth = 5

    flag = len(cards) <= 16

    ...
    if len(cards) < 30 and len(cards) > 22:
        depth =
5         flag = False
    elif len(cards) <= 22 and len(cards) > 1
6:         depth =
7         flag = False
    elif len(cards) <= 16 :
        depth =
9         flag = True
    else:
        depth =
3         flag = False
    ...
    best_move = minimax(
        cards=cards,
        player1=player1,
        player2=player2,
        depth=depth,
        is_maximizing=True,
        alpha=-inf,
        beta=inf,
        no_heuristic=flag
    )[1]

    return best_move

```

این تابع آخرین مرحله‌ای که ایجنت تصمیم می‌گیرد "الان باید چه حرکتی کنم؟". با کمک الگوریتم Minimax و تنظیمات خاصی که براساس وضعیت فعلی بازی انا انجام می‌شود، بهترین حرکت ممکن رو برای بازیکن اول (ایجنت هوشمند ما) پیدا می‌کنه.

1. بررسی تعداد کارت‌های باقی‌مونده:

- تابع اول نگاه می‌کنه چندتا کارت روی تخته باقی مونده. تعداد کارت‌ها تعیین می‌کنه که عمق جستجوی Minimax چقدر باشه:
- کارت‌های زیاد: عمق کمتر (مثلاً 5) چون بررسی همه‌ی حالت‌ها زمان‌بر می‌شه.
- کارت‌های کم: عمق بیشتر (مثلاً 9) چون بازی داره به انتها نزدیک می‌شه و بررسی دقیق‌تر لازمه.

2. فعال‌سازی حالت no heuristic:

- وقتی تعداد کارت‌ها خیلی کم می‌شه (مثلاً کمتر از 16 کارت)، به حالت خاص به اسم "no heuristic" فعال می‌شه. این حالت به جای اینکه بخوایم از تابع هیوریستیک استفاده کنیم مستقیماً از خود استیت نهایی بازی استفاده میکنیم برای اینکه مشخص کنیم کی بازیه برده یا باخت.

3. اجرای Minimax:

- بعد از تنظیم عمق و حالت جستجو، تابع minimax رو صدا می‌زنه و ازش می‌خواد که بهترین حرکت رو بر اساس وضعیت فعلی بازی پیدا کنه.
- Minimax برمی‌گردونه که کدوم حرکت بهترین امتیاز رو داره.

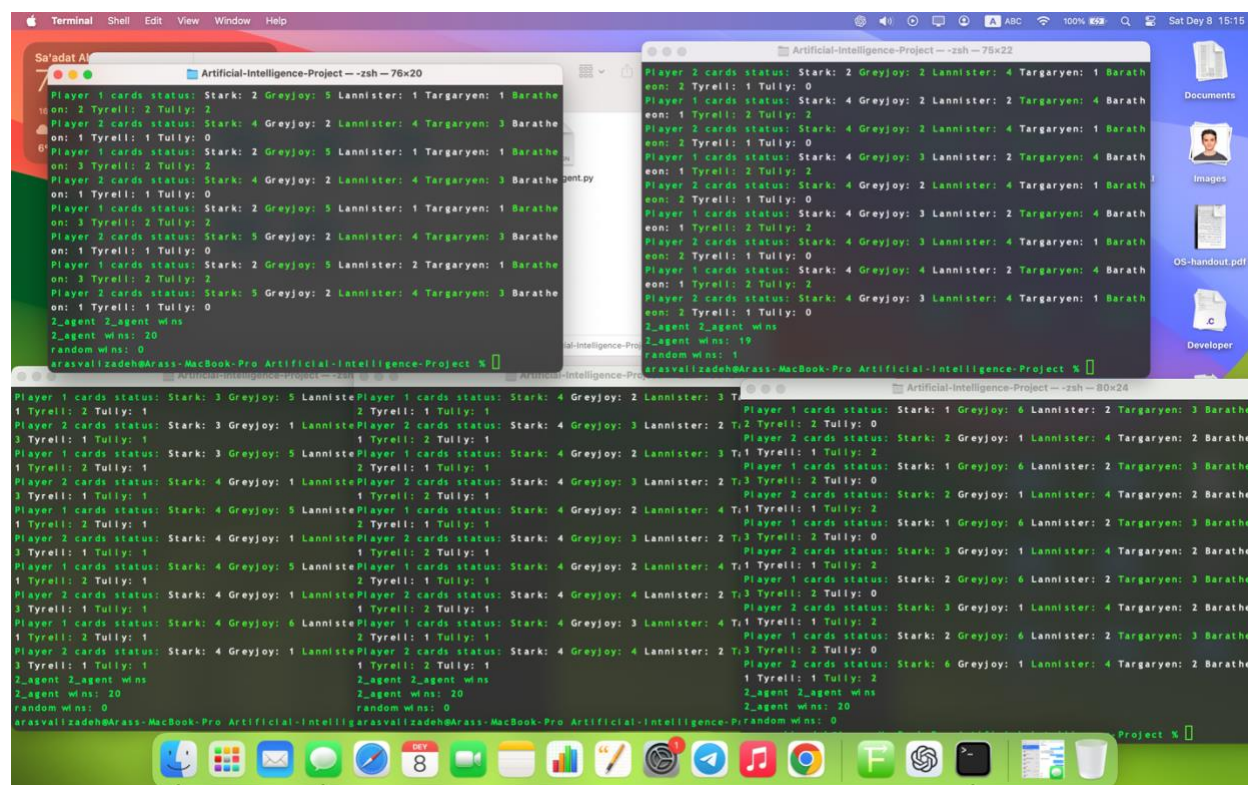
4. برگشت بهترین حرکت:

- وقتی Minimax نتیجه رو داد، این تابع شماره‌ی بهترین حرکت رو برمی‌گردونه تا ایجنت حرکتش رو انا انجام بده.

نتایج:

در نهایت ۱۰۰ بار ایجنت خودمون رو با ایجنت رندوم بازی دادیم که موفق شد ۹۹ بار ایجنت رندوم رو شکست بده و فقط ۱ بار بازه. هرچی عمق مینیماکس رو بیشتر کنیم طبعاً درصد موفقیت بیشتر میشه ولی از اونور زمان اجرای بازی هم طولانی میشه.

اسکرین شات نتایج خروجی رو پایین قرار میدم.



The screenshot shows a macOS desktop with three terminal windows open, each displaying the output of a game simulation. The windows are titled 'Artificial-Intelligence-Project --zsh --76x20', 'Artificial-Intelligence-Project --zsh --75x22', and 'Artificial-Intelligence-Project --zsh --80x24'. Each window shows a series of lines representing game states and results, including player names (Stark, Greyjoy, Lannister, Targaryen, Baratheon, Tyrell, Tully) and their respective card statuses. The results indicate that the '2_agent' (the AI agent) wins 20 out of 20 random games, while the 'random' wins 0 out of 20.

ممنون از توجه شما :