دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

گزارش ۴ درس هوش مصنوعی

پیاده سازی یک بازی دونفره با قابلیت های جستجوی تخاصمی

به قلم
امیر بابامحمودی

استاد
دکتر مهدی قطعی

فروردین ۱۳۹۹

مقدمه:

در هوش مصنوعی بازی ها نوع خاصی از مسائل به شمار میروند. در این محیط های چند عاملی هر عامل باید فعالیت سایر عامل ها و تاثیر آن ها بر روند کار خود را در نظر بگیرد. اصولا بازی های هوش مصنوعی به صورت دونفری و نوبتی در نظر گرفته میشود.
نوعی از این بازی ها Zero-Sum میباشد که در آن اعداف عامل متناقض است که مجموع مقادیر سودمندی در پایان بازی صفر یا مقدار ثابتی است به این معنی که به ازای عاملی که سود کند عاملی وجود دارد که زیاد میکند .
در این گزار به تحلیل و پیاده سازی بازی connect four پرداخته شده است.

شرح بازی:

بازی connect four یک بازی دونفره در دسته بازی های Zero-Sum قرار میباشد.در این بازی که بیشتر شبیه به بازی X-O پیشرفته میباشد. یک جدول 7*6 داریم که هر بازیکن در نوبت خود میتواند مهره ی رنگ خودش را در هر ستونی که بخواهد(در صورتی که ستون مذکور جای خالی داشته باشد) بگذارد.
بازی زمانی تمام میشود که یکی از بازیکنان ۴ مهره اش را بتواند بصورت پشت سر هم بصورت عمودی , افقی یا قطری(با شیب منفی یا مثبت) بچیند .در صورتی که تمامی خانه ها پر شوند و شرایط گفته شده رخ ندهد بازی مساوی و بدون برنده به اتمام میرسد .

استراژی های برد:

۱ ـ قرار دادن مهره ها در ستون وسط:

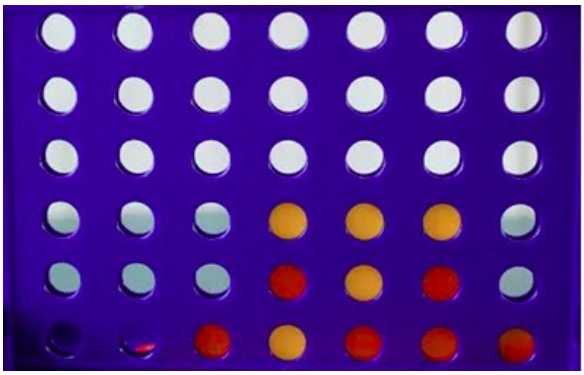
اگر شروع بازی با شما باشد یک استراژی خوب این میباشد که مهره ی خود را در ستون وسط قرار دهید. به طور کلی از آنجایی که بازی شامل ۷ ستون میباشد اینکه تعداد بیشتری مهره در وسط داشته باشیم تعداد اتصال های عمودی , افقی و قطری را بیشتر و ممکن تر میکند.

۲ ـ تله گذاشتن برای حریف:

یک استراتژی معمول مسدود کردن راه های برد حریف میباشد. جلو گیری کردن از اینکه حریف بتواند ۳ مهره ی متصل را داشته باشد با قرار داد مهره ها در ستون های کناری و بطور کلی برای مسدود کردن آن.این استراتژی همچنان از اینکه حریف بتواند برای شما تله گذاری کند را نیز جلوگیری میکند.

۳ـ ساختن "7" :

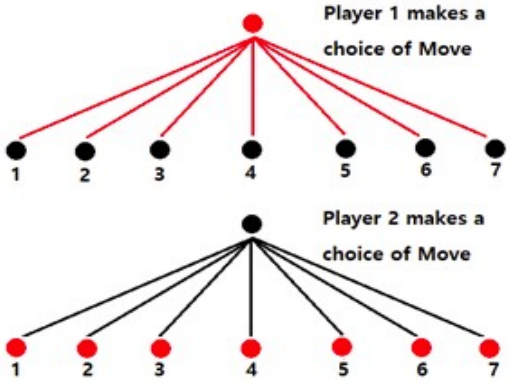
تله ی 7 نام استراتژی ای است که که یکی از عامل های بازی مهره های خود را در قالبی شبیه به 7 میچیند. منظور از آرایش به شکل 7 داشت ۳ مهره ی متصل بـه صـورت افقی و داشـتن دو مهـره ی قطـری در یکی از طرفین این اتصال افقی میباشد که یعنی 7 میتواند در هر جهتی باشد یعنی از راست به چپ , پایین به بالا یا هم از راست به چپ و هم پایین به بالا. این آرایش خوب میباشد زیـر از هـر جهت امکـان توسـعه ی آن بـرای پیروزی وجود دارد. از چپ و راست به صورت افقی و از دو طرف به صورت قطری.



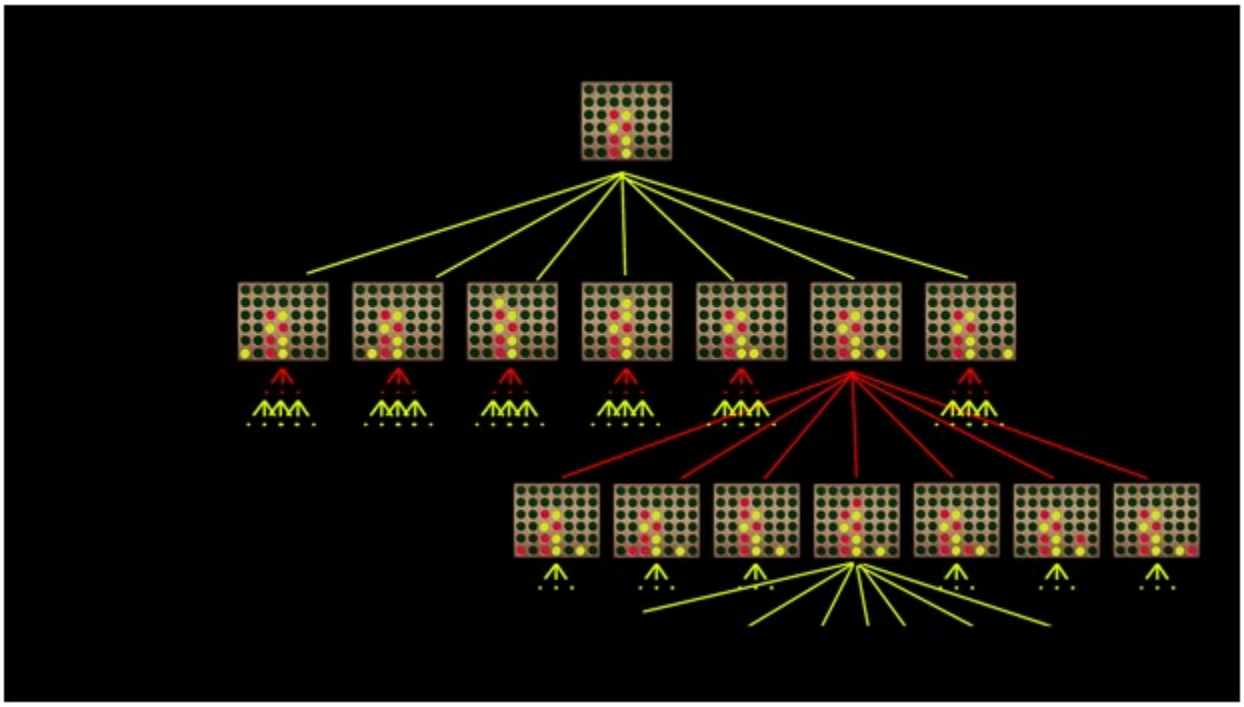
شکل۱)یه نمونه از آرایش 7 برای رنگ زرد

درخت تصمیم در بازی:

در شروع بازی عامل اول مهره ی خود را در یکی از ۷ ستون میتواند قرار دهد که یعنی نود اول دارای ۷ شاخه میباشد. حال که نوبت به بازیکن بعدی میرسد او نیز میتواند یکی از ۷ ستون را انتخاب کند. این نکته لازم به ذکر میباشد که این درخت تصمیم یک محدودیت دارد. برای مثال اگر یک ستون در مجموع ۶ بار انتخاب شود دیگر نمیتوان آن را انتخاب کرد و پر میشود که یعنی از آن مرحله به بعد یکی از شاخه های درخت کم میشود. از آنجایی که 42 = 7*6 به عنوان محدودیت دوم هرگاه هر دو بازیکن مجموعا ۴۲ بار نوبت خود را بازی کنند و هیچ ۴ مهره ی همرنگی متصل تا آن موقع رویت نشود درخت تصمیم متوقف شده و بازی بدون برنده تمام میشود.



شکل۲)نمایی از یک نود و شاخه هاش



شکل۳)نمایی از درخت تصمیم در استیتی در اواسط بازی

پیاده سازی بازی:

```python
def initial_state():
    board = np.zeros((row_number, column_number))
    return board

#dropping a peice in a section of table
def put_piece(board, row, col, piece):
    board[row][col] = piece

#checks if a column has an empty space
def validate_location(board, col):
    return board[row_number - 1][col] == 0
#gives us all columns which still have empty spaces
def valid_columns(board):
    valid_locations = []
    for col in range(column_number):
        if validate_location(board, col):
            valid_locations.append(col)
    return valid_locations
```

در ابتدا تابع initial_state رو میبینید که صفحه ی اولیه و خالی بازی را میسازد.
تابع put-piece مهره ی مورد نظر بازیکن رو در خانه ی مذکور میگذارد.

تابع validate-location بررسی میکند که صفحه ی بازی خانه ی خالی داشته باشد.
تابع valid-columns تمامی ستون هایی که جای خالی دارند را میدهد.

شکل۴)برخی از توابع مربوط به بازی

```python
def first_valid_row(board, col):
    for r in range(row_number):
        if board[r][col] == 0:
            return r
#printing the matrix of game table
def print_board(board):
    print(np.flip(board, 0))
#checks if we are in terminal or not
def check_termination(board):
    if check_winner_existence(board, human_piece
            ) or check_winner_existence(
        board, ai_piece) or len(valid_columns(board)) == 0:
        return True
    else:
        return False
```

تابع first-valide-row اولین ردیفی که در یک ستون میتوان مهره قرار داد را میدهد.

تابع check-termination رسیدن به نود ترمینال را بررسی میکند با بررسی پر شدن خانه ها یا وجود یک برنده

شکل۵)توابع پیاده سازی شده در بازی

در این تابع تمامی ۴ مهره های متوالی در سطر ها, ستون ها و قطرها بررسی شده که اگر یک رنگ بودند برنده بازی مشخص شود.
این تابع شامل ۴ بخش بوده که در یکی ۴ تا مهره متوالی سطر ها در یکی ستون ها و در دوتای بعدی قطر ها با شیب منفی و مثبت بررسی شده است.

```
def check_winner_existence(board, piece):
    # Check horizontal locations for win
    for c in range(column_number - 3):
        for r in range(row_number):
            if board[r][c] == piece and board[r][c + 1
            ] == piece and board[r][c + 2] == piece and board[r][
                c + 3] == piece:
                return True
```

شکل۶)تابع مشخص کننده وجود پیروز

تابع EVAL :

تابع eval پیاده سازی شد به صورت تجربی و طی بازی کردن با عاملی AI بدست آمده و در حقیقت به این صورت میباشد که یک لیست ۴ تایی به عنوان ۴ خانه ی متصل به هم از نظر سطری, ستونی و یا قطری را دریافت کرده و بررسی میکند با قرار دادن یک مهره ی جدید عامل در یکی از این ۴ خانه چه شرایطی پیش میاید. اگر با قرار دادن مهره ی جدید هر ۴ مهره هم رنگ و به رنگ مهره ی عامل در بیاید بیشترین امتیاز را داشته زیرا عملا یعنی عامل پیروز بازی میباشد که عدد بزرگی را بعنوان امتیاز با امتیاز ها کل جمع میزنیم.
اگر ۳ مهره هم رنگ شوند به میزان ۵ به امتیاز اضافه میکنیم و الی آخر. در نهایت نیست امتیازی که از ۴ خانه ورودی بدست میاید به عنوان خروجی میگیریم.

```
def eval(window, piece):
    score = 0

    if window.count(piece) == 4:
        score += 100
    elif window.count(piece) == 3 and window.count(empty) == 1:
        score += 5
    elif window.count(piece) == 2 and window.count(empty) == 2:
        score += 2
    if piece == human_piece and window.count(ai_piece) == 3 and window.count(empty) == 1:
        score -= 4
    if piece == ai_piece and window.count(human_piece) == 3 and window.count(empty) == 1:
        score -= 4

    return score
```

شکل۷)تابع eval

تابع total_score :

دیدیم که تابع eval تنها امتیاز ۴ خانه ی متصل را محاسبه میکند. تابع total_score امتیاز نسبت داده شده به یک استیت کلی از بازی نسبت به یک مهره را محاسبه میکند. به این صورت که مانند کاری که در تابع check_winner_existence که تمامی ۴ تایی های مختلف را در نظر گرفته و به تابع eval داده و امتیاز نهایی هر ۴ خانه را با هم جمع میکنیم. در ابتدای گزارش که درباره استراتژی بازی صحبت شد دیدیم که داشتن مهره های زیادی در خانه ی مرکزی امتیاز مثبتی به حساب میاید که در قسمت اول این تابع میبینید که به ازای تعداد مهره های یک عامل در مرکز امتیاز خاصی برای آن قائل شده ایم.

```
def total_score(board, piece):
    score = 0

    ## Score center column
    center_array = list(board[:, 3])
    center_count = center_array.count(piece)
    score += center_count * 3

    ## Score Vertical
    for c in range(column_number):
        for r in range(row_number - 3):
            window = list(board[:, c])[r:r + 4]
            score += eval(window, piece)

    ## Score posiive sloped diagonal
    for r in range(row_number - 3):
        for c in range(column_number - 3):
            window = [board[r + i][c + i] for i in range(4)]
            score += eval(window, piece)
```

شکل۸)تابع total_score

تابع alpha_beta_search :

حال به شرح الگوریتمی که عامل AI با آن بازی میکند میپردازیم:

برای این مسئله از الگوریتم apha_beta به همراه cutoff و تابع evaluation استفاده شده است که در واقع زیرمجموعه ای از الگوریتم minimax بوده که در آن تمامی شاخه ها بررسی نشده و از نظر مصرف زمانی و فضایی به صرفه تر است در این روش نه تنها تمامی شاخه های یک فرزند بررسی نمیشود بلکه ت

عمق محدودی از درخت را پیمایش کرده و جواب بهین تخمین زده میشود همانگونه که در تابع alpha_beta search میبینید که مانند کتاب پیاده سازی شده است این تابع در واقع فراخواننده تابع max_value بوده و با ورودی گرفتن وضعیت فعلی بازی این تابع را صدا میزند .

```
#implementation of alpha-beta algorighm
def alpha_beta_search(board, depth, alpha, beta):
    return max_value(board , depth , alpha , beta )
```

شکل۹)تابع alpha_beta

در تابع max_value بیشترین امتیاز گره های پایینی درخت برگردانده شده و این کار با آپدیت کردن مرحله به مرحله آلفا صورت میگیرد به گونه ای که تا عمق محدودی که به عنوان ورودی به تابع میدهیم پیمایش صورت میگیرد. لازم به ذکر است که در اول این تابع بررسی مشود که آیا به هیچ یک از حالت های ترمینال رسیده ایم یا خیر.در نهایت نیز امتیاز ماکسیمم به همراه ستونی که با قرار دادن مهره ی مورد نظر به آن رسیده ایم بازگردانده میشوند

```
def max_value(board , depth , alpha , beta ):
    is_terminal = check_termination(board)
    if depth == 0 or is_terminal:
        if is_terminal:
            if check_winner_existence(board, ai_piece):
                return (None, 10000000000)
            elif check_winner_existence(board, human_piece):
                return (None, -10000000000)
            else:  # Game is over, no more valid moves
                return (None, 0)
        else:  # Depth is zero
            return (None, total_score(board, ai_piece))
    valid_locations = valid_columns(board)
    value = -math.inf
    column = random.choice(valid_locations)
    for col in valid_locations:
        row = first_valid_row(board, col)
        b_copy = deepcopy(board)
        put_piece(b_copy, row, col, ai_piece)
        new_score = min_value(b_copy, depth - 1, alpha, beta)[1]
        if new_score > value:
            value = new_score
            column = col
        alpha = max(alpha, value)
```

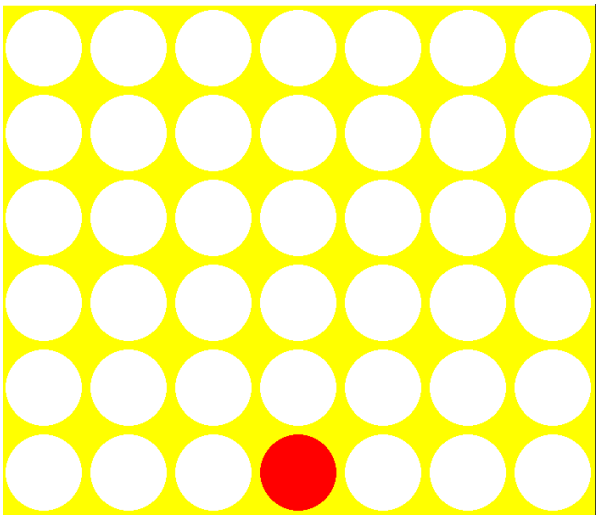
شکل۱۰)تابع max_value

در تابع min_value که ساز و کاری مانند max_value را دارا میباشد قصد در برگرداندن کمترین امتیاز از نود های پایینی درخت را داشته و این کار با محدودیت cutoff عمقی که به عنوان ورودی به تابع آن داده میشود انجام میدهیم و مانند تابع پیشین در ابتدا رسیدن به نود های ترمینال نیز بررسی شده و در نهایت امتیاز مینیمم به همراه ستونی که امتیاز مینیمم را برای ما رقم زده برگردانده میشوند.

```python
def min_value(board , depth , alpha , beta ):
    is_terminal = check_termination(board)
    if depth == 0 or is_terminal:
        if is_terminal:
            if check_winner_existence(board, ai_piece):
                return (None, 10000000000)
            elif check_winner_existence(board, human_piece):
                return (None, -10000000000)
            else:  # Game is over, no more valid moves
                return (None, 0)
        else:  # Depth is zero
            return (None, total_score(board, ai_piece))
    valid_locations = valid_columns(board)
    value = math.inf
    column = random.choice(valid_locations)
    for col in valid_locations:
        row = first_valid_row(board, col)
        b_copy = deepcopy(board)
        put_piece(b_copy, row, col, human_piece)
        new_score = max_value(b_copy, depth - 1, alpha, beta)[1]
        if new_score < value:
            value = new_score
            column = col
        beta = min(beta, value)
```

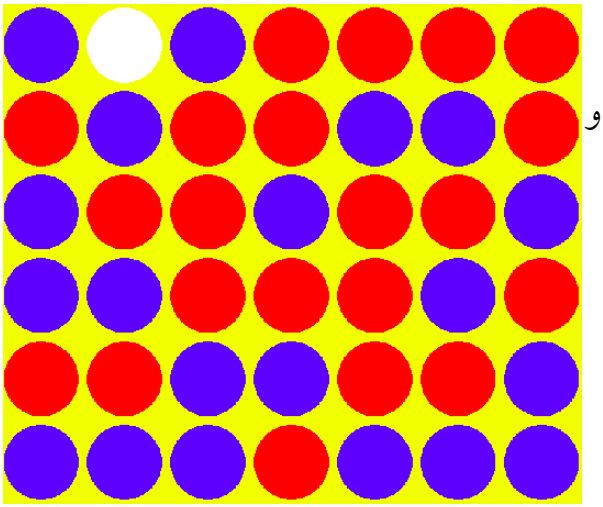
شکل۱۱)تابع min_value

یکبار اجرای بازی:



این بازی به صورت رقابت انسان با AI طراحی شده که رنگ آبی مربوط به انسان بوده و رنگ قرمز AI .
همانطورکه میبینید اگر AI شروع کننده بازی باشد مهره ی خود را در ستون وسط قرار میدهد همانگونه که استراتژی پیاده سازی شد.

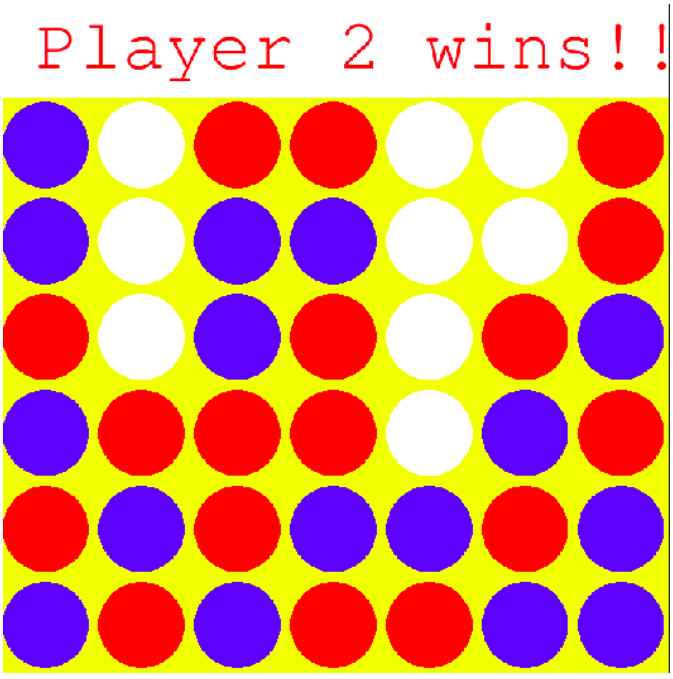
شکل ۱۲)اولین حرکت AI



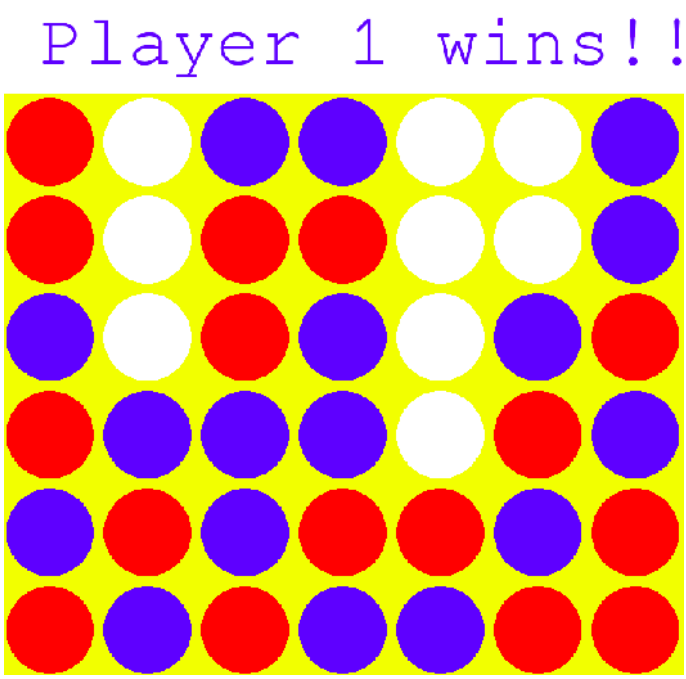
حال بازی را تا ته ادامه داده و نتیجه را نشان میدهیم:
همانگونه که میبینید بازی پر از قسمت هایی میباشد که مهره های قرمز رنگ دوتا دوتا یا سه تا سه تا چه به صورت قطری چه سطری و چه ستونی قرار دارند که نشان دهنده ی این میباشد که طبق توضیحات پیاده سازی شده در الگوریتم, الگوریتم به خوبی کار میکند و همچنین تعداد مهره های قرمز رنگ در ردیف وسط نیز بیشتر بوده که یعنی طبق استراتژی های برد پیاده سازی شده پیش رفته است.

شکل ۱۳)نتیجه ی نهایی یک بازی با AI

حال قصد داریم بازی را در دو جای مختلف ران کرده در یکی AI شروع کننده بازی بوده و در دیگری خودم و عینا حرکات AI را در بازی ای که شروع کننده میباشد در بازی ای که خودم شروع کننده هستم پیاده کنم:



شکل۱۴)بازی ای که AI شروع کننده بود



شکل ۱۵)بازی ای که انسان شروع کننده بود

تحلیل:

همانگونه که میبینید و شاید خلاف انتظار بود هر دو بازی پیروز داشت(شاید یکسری انتظار تساوی داشتند!) طی پرسش هایی که از تدریسیاران درس[1] داشتم اینطور بنظر میاید که درخت تصمیم بازی کوچک بوده(نسبت به بازی هایی مانند شطرنج) و تابعی تحت عنوان رندمایز برای حرکت AI نداریم(و RL هم پیاده سازی نشده!) هنگامی که هر دو عامل به صورت بهینه بازی کنند عامل شروع کننده پیروز میباشد. دلیل دیگر هم آن است که بازی کاملا متقارن پیش رفته و هیچ جایی ندارد که بتوان عملکرد های متفاوتی داشت(برای مثال حالت کیش شدن در شطرنج).

برای دیدن کد کامل و ران کردن آن میتوانید به لینک زیر مراجعه فرمایید:

https://github.com/amirbabamahmoudi/AI-projects/tree/main/connect_four

منابع:[1]

۱ـ Artificial Intelligence A Modern Approach , Stuart J.Russell and Peter Norvig

۲ـ https://medium.com/analytics-vidhya/artificial-intelligence-at-play-connect-four-minimax-algorithm-explained-3b5fc32e4a4f

۳ـ در پیاده سازی بخش های گرافیکی و ایده ی کلی کد از این لینک کمک گرفته شده است.
https://github.com/KeithGalli/Connect4-Python

1 اسحاق موتابی