به نام خدا

هوش مصنوعی پروژه دوم

الگوریتم min-max

نرگس غلامی

۸۱۰۱۹۸۴۴۷

هدف پروژه : هدف پروژه، آشنایی با الگوریتم **min max** از طریق بازی چکرز بود.

توضیح کلی در مورد پروژه: در این بازی دو بازیکن داریم که یکی در صدد ماکسیمم کردن امتیازات خود و دیگری در صدد مینیموم کردن آن است. قصد دو بازیکن این است که بیشترین مهره‌های حریف زده شود. حال باید این بازی با استفاده از الگوریتم **minimax** پیاده سازی شود.

توضیح در مورد توابعی که باید توسط ما پر می‌شد:

تابع **getValidMoves**

```python
def getValidMoves(self, piece):

    step = -1 if piece.color == RED else 1  #red --> step = -1
    #print("get", piece.row, piece.col, piece.color)
    if piece.king:
        moves_left = (self._traverseLeft(piece.row+1, piece.row+3, 1, piece.color, piece.col-1, skipped=[]))
        moves_right = (self._traverseRight(piece.row+1, piece.row+3, 1, piece.color, piece.col+1, skipped=[]))
        moves_left2 = (self._traverseLeft(piece.row-1,  piece.row-3, -1, piece.color, piece.col-1, skipped=[]))
        moves_right2 = (self._traverseRight(piece.row-1,  piece.row-3, -1, piece.color, piece.col+1, skipped=[]))
        if moves_left2:
            for i in range(len(moves_left2)):
                moves_left.append(moves_left2[i])
        if moves_right2:
            for i in range(len(moves_right2)):
                moves_right.append(moves_right2[i])


    elif piece.color == RED:
        moves_left = self._traverseLeft(piece.row-1 ,  piece.row-3, step, RED, piece.col-1, skipped=[])
        moves_right = self._traverseRight(piece.row-1 , piece.row-3, step, RED, piece.col+1, skipped=[])

    elif piece.color == WHITE:
        moves_left = (self._traverseLeft(piece.row+1 , piece.row+3, step, WHITE, piece.col-1, skipped=[]))
        moves_right = (self._traverseRight(piece.row+1 ,piece.row+3, step, WHITE, piece.col+1, skipped=[]))

    return moves_right, moves_left
```

اولین تابع، تابع **getValidMoves** است که حرکات مجاز یک مهره را بسته به این که شاه است و یا این که رنگش چیست برمی‌گرداند. من نوع داده‌ی **moves** را به لیست تغییر دادم. در نتیجه یک لیست مخصوص حرکت به سمت راست و یک لیست مخصوص حرکت به سمت چپ دارم که با استفاده از توابع **traverseLeft** و **traverseRight** به دست آمده‌اند.

توضیح ورودی‌های تابع:

ورودی اول، ردیف خانه‌ای است که باید آن را بررسی نماید پس برای رنگ قرمز یک واحد کمتر از جایی است که ایستاده است و برای سفید یک واحد بیشتر است.

ورودی دوم به معنی این است که تا کجا باید پیش رود. بررسی باید تا حداقل سه خانه آن طرف‌تر اتفاق بیفتد.

ورودی سوم که همان **step** است به معنی نوع حرکت مهره قرمز و سفید است که به ترتیب برابر ۱- و ۱ می‌باشد.

بقیه‌ی ورودی‌ها هم که مشخصا به ترتیب رنگ، چپ رفتن و راست رفتن و آرایه **skipped**(مخصوص تابع بازگشتی) را نشان می‌دهد.

تابع minimax :

```
def minimax(board , depth, maxPlayer):
    if(depth < 0):
        return board.evaluate(), board

    moves = []
    color = WHITE if maxPlayer else RED
    v = INFINITY_MIN if maxPlayer else INFINITY_MAX

    finalBoard = copy.deepcopy(board)
    pieces = board.getAllPieces(color)
    for piece in pieces:
        moves_right, moves_left = board.getValidMoves(piece)
        moves = [moves_right, moves_left]
        for move in moves:
            if not len(move):
                continue
            sampleBoard = copy.deepcopy(board)
            samplePiece = sampleBoard.getPiece(piece.row, piece.col)
            sampleBoard = simulateMove(samplePiece, move, sampleBoard, color, len(move))
            newV, newBoard = minimax (sampleBoard , depth-1, not maxPlayer)
            if (maxPlayer and v < newV) or (not maxPlayer and v > newV):
                finalBoard = copy.deepcopy(sampleBoard)
                v = newV

    return v, finalBoard
```

تابع بعدی تابع **minimax** است. این تابع روی تمام مهره‌های یک رنگ حلقه می‌زند و برای هر کدام نیز تمام حرکت‌های موجود را پیدا می‌کند و سپس با استفاده از تابع **simulateMove** ، بورد آن را شبیه‌سازی می‌کند. این بورد را دوباره به تابع **minimax** می‌دهد، سپس همینطور تابع بازگشتی را ادامه می‌دهد تا به عمق کمتر از صفر برسد. هنگامی که به این مرحله رسید از **evaluate function** ارزش آن بورد را می‌گیرد و بازمی‌گرداند. اگر نوبت مهره سفید باشد، از تمام بوردهای بازگردانده شده، آن بوردی را انتخاب می‌کند که ارزش بیشتری داشته باشد ولی اگر نوبت قرمز باشد سعی بر مینیموم کردن ارزش بورد دارد، پس آنی را انتخاب می‌کند که ارزش کمتری داشته باشد.

تابع simulateMove :

```
def simulateMove(piece, move, board, color, skip):
    if(skip > 1):
        board.move(piece, move[-1][0], move[-1][1])
        i = 0
        while i < len(move):
            board.remove2(move[i][0], move[i][1], color)
            i += 2
    else:
        board.move(piece, move[0][0], move[0][1])

    return board
```

همان‌طور که گفته شد داده ساختار **move** لیست می‌باشد و من از همین روش برای متوجه شدن **skip** استفاده کردم. اگر طول لیست بزرگتر از یک باشد به این معناست که حرکت مورد نظر طول بیشتر از یک دارد. حال اگر قرار بر **skip** بود مهره را به خانه‌ی آخر انتقال می‌دهیم و بقیه‌ی خانه‌ها را حذف می‌نماییم ولی اگر **skip** لازم نبود تنها مهره را به خانه‌ی بعدی انتقال می‌دهیم و حذفی در کار نیست.

تحلیل وقتی که هر دو عمق برابر یک می‌باشد:

در این حالت دو مهره دوراندیشی آنچنانی ندارند و فقط به یک قدم آینده خود و حرکت حریف فکر می‌کنند. بنابراین بازی خیلی هوشمندانه پیش نمی‌رود ولی از طرف دیگر سرعت محاسبات بسیار بالاست و فضای اشغالی کمتر است.

تحلیل وقتی که عمق یکی برابر با دو و عمیق دیگری برابر پنج می‌باشد:

در این جا یکی از مهره‌ها تا دو حرکت بعد را حدس می‌زند ولی دیگری تا پنج حرکت بعد را می‌تواند حدس بزند. در نتیجه، داستان به این صورت پیش می‌رود که مهره‌ای که عمق بیشتری را می‌تواند ببیند خیلی قوی‌تر از مهره‌ی دیگر رفتار می‌کند و برنده‌ی بازی می‌شود ولی از طرف دیگر انگار زمان فکر کردنش هم بیشتر است و زمان بیشتری طول می‌کشد تا حرکت بعدی خود را اعلام کند.

تحلیل وقتی که هر دو عمق برابر پنج می‌باشد:

در این حالت هر دو مهره تا پنج حرکت بعد خود را می‌توانند ببینند. در نتیجه تصمیمات بهتری خواهند گرفت ولی از آن طرف زمانی که صرف محاسبات می‌شود خیلی زیاد می‌شود و حافظه‌ی بسیاری هم برای ذخیره این اطلاعات نیاز است.

زمانی که پروژه من برای محاسبه با عمق پنج صرف می‌کرد خیلی زیاد بود.

نتیجه گیری کلی: این الگوریتم یکی از الگوریتم‌های خوب برای انتخاب استراتژی مناسب در جهت برنده شدن است. اگر عمق را زیاد در نظر بگیریم بازی هوشمندانه‌تر جلو می‌رود ولی به زمان بیشتری نیاز داریم و از آن طرف در صورت داشتن عمق کمتر، در مدت زمان کمتری می‌توانیم جواب را بدست آوریم ولی حرکتی که انجام می‌دهیم هوشمندی کمتری دارد.

مواردی برای بهبود پروژه: پروژه جالبی بود، ولی من علاقه داشتم خودم بقیه‌ی قسمت‌هایش را بنویسم چون باعث می‌شد که به قسمت‌های مختلفش مسلط‌تر باشم.

با تشکر از زحمات شما.

منبع:

از اسلایدهای استاد استفاده شد.