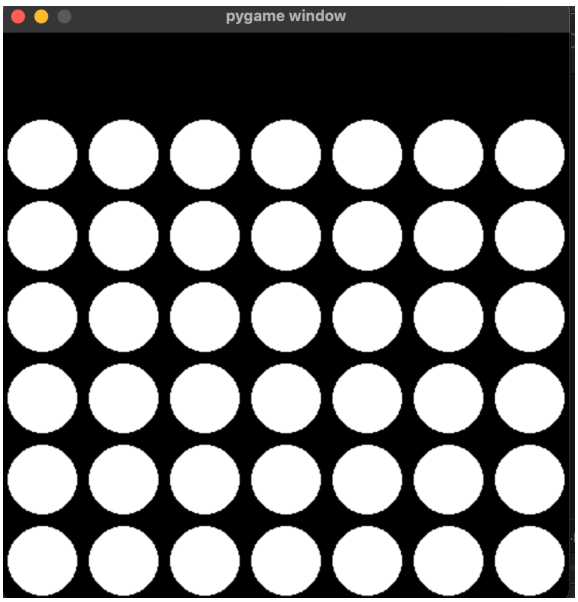
امیرمحمد رمضان نادری
۶۱۰۳۹۸۱۲۶

پروژه هوش مصنوعی

این مسئله four connect می باشد و ما قصد داریم محیطی طراحی کنیم که دو انسان, انسان و هوش مصنوعی و دو هوش مصنوعی بتوانند با یکدیگر بازی کنند. قانون بازی به این صورت است که هر بازیکن اگر بتواند تعداد مهره های پشت سر هم خودش را به صورت عمودی, افقی و یا مورب به عدد چهار برساند برنده بازی است.

ابتدا سطر و ستون محیط بازی را از کاربر می گیریم. معمولا این محیط به صورت یک ماتریس ۶*۷ می باشد. محیط بازی را می توانید در زیر مشاهده کنید:



در مرحله بعد می توانید حالت بازی را انتخاب کنید به این صورت که می خواهید یک انسان با هوش مصنوعی بازی کند یا دو هوش مصنوعی با هم بازی کنند و یا دو انسان با یکدیگر بازی کنند. برای هوش مصنوعی از الگوریتم minimax استفاده می کنیم. زمانی که برنده بازی مشخص شود, بازی به اتمام می رسد, نام برنده چاپ می شود و محیط بازی بسته می شود.

تابع امتیاز دهی که در این بازی طراحی شده است و یا همان تابع heuristic به این صورت است که یک آرایه چهارتایی از مهره ها را می گیرد که می تواند افقی, عمودی و

یا مورب باشد و در آخر با توجه به تعداد مهره های خودش یک عددی نسبت می دهد. این تابع در کد return_score نام دارد.

```
Score = 0
if count of PLAYER_PIECE in array equals 4 then
        score+= 100
else if count of PLAYER_PIECE in array is 3 then
        score+= 70
else if count of PLAYER_PIECE in array is 2 then
        score+= 20
if count of OPPONENT_PIECE in array is 3 then
        score -=40
```

ستون های وسط به دلیل اینکه امکان بیشتری برای براورد کردن شرط برد بازی فراهم میکنند از اهمیت بیشتری برخوردار هستند بنابراین باید وزن آن ها در تعیین مقدار تابع هیوریستیک بیشتر باشد:

```
center_array = [int(i) for i in list(board[:, self.col_number//2])]
center_count = center_array.count(piece)
score += center_count * 3
```

بقیه مقادیر هم که به صورت افقی و ... در ستون های غیر وسط می باشند به صورت زیر است:

```
for r in range(self.row_number-3):
    for c in range(self.col_number-3):
        candidate = [board[r+i][c+i] for i in range(4)]
        score += self.return_score(candidate, piece)

        for r in range(self.row_number-3):
            for c in range(self.col_number-3):
                candidate = [board[r+3-i][c+i] for i in range(4)]
                score += self.return_score(candidate, piece)

        for r in range(self.row_number):
            row_array = [int(i) for i in list(board[r,:])]
            for c in range(self.col_number-3):
                candidate = row_array[c:c+4]
                score += self.return_score(candidate, piece)

        for c in range(self.col_number):
            col_array = [int(i) for i in list(board[:,c])]
            for r in range(self.row_number-3):
                candidate = col_array[r:r+4]
                score += self.return_score(candidate, piece)
```

الگوریتم minimax نیز به صورت بازگشتی پیاده شده است. به این صورت که مقدار آلفا و بتا را به همراه مقدار بولینی به عنوان ورودی می گیرد که مقدار بولین بیانگر این است که در عمق فعلی باید بیشترین امتیاز و یا کمترین را در نظر بگیریم.
در ماکسیمم گیری, حالتی اتفاق میفتد که حریف به مینیمم خود برسد و در آنجا مقدار آلفا تنظیم می شود. اگر در ادامه اکشنی صورت بگیرد که آلفا آن بزرگ تر از بتا باشد, شاخه را ادامه می دهیم و عمل beta_pruning صورت می گیرد.
حال اگر مقدار بولین برابر False باشد دنبال حالتی هستیم که حریف ماکزیمم شود. در این حالت بتا تنظیم می شود. حال اگر عملی صورت بگیرد که آلفا بزرگ تر مساوی از بتا باشد, شاخه را ادامه نمی دهیم و alpha_pruning انجام می شود. می توانید در کد تابع minimax_algorithm را ببینید. شرط خروج نیز در حالتی است که عمق برابر صفر باشد و یا به برگ درخت برسیم(برد , باخت).