به نام خدا

تمرین کامپیوتری سری دوم

قسمت دوم: بازی

گزارش :

در ابتدای کار یک تابع main برای اجرا کردن برنامه میسازیم :

def main():

    othello = Othello(8, 7, True)

    winner = othello.play()

    print(f"Winner: {winner}")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

در ادامه یک تابع هیوریستیک بدون هرس با minimax عادی مینویسیم.

پیاده سازی بدین صورت هست که سه تابع

def get\_best\_move(self, player)

def minimax(self, player, depth)

 def heuristic(self, player)

را داریم که هرکدام کار خاصی را انجام میدهد.

تابع get\_best\_move یک کپی از بورد میسازد و در ان کپی حالات مختلف را برای بهترین حرکت بررسی میکند.

def get\_best\_move(self, player):

        moves = self.get\_valid\_moves(player)

        if len(moves) == 0:

            return None

        best\_move, best\_value = None, -math.inf

        for move in moves:

            new\_board = copy.deepcopy(self.board)

            self.make\_move(player, move)

            value = self.minimax(-player, self.minimax\_depth - 1)

            if value > best\_value:

                best\_move, best\_value = move, value

            self.board = new\_board

        return best\_move

تابع بعدی تابع minimax هست که بدون هرس(prune) میباشد و میایید برای تابع best move ما بهترین مقدار را برمیگرداند.

def minimax(self, player, depth):

        if depth == 0:

            return self.heuristic(player)

        moves = self.get\_valid\_moves(player)

        if len(moves) == 0:

            return self.heuristic(player)

        best\_value = -math.inf if player == 1 else math.inf

        for move in moves:

            new\_board = copy.deepcopy(self.board)

            self.make\_move(player, move)

            value = self.minimax(-player, depth - 1)

            if player == 1:

                best\_value = max(best\_value, value)

            else:

                best\_value = min(best\_value, value)

            self.board = new\_board

            if self.prune and best\_value == 1:

                break

        return best\_value

در ادامه تابع heuristic ساده ای داریم که صرفا میایید در هر مرحله برای ما تفاوت مقدار مهره های سفید و سیاه را بر اساس player انتخابی برمیگرداند.

def heuristic(self, player):

        white\_count = sum([row.count(1) for row in self.board])

        black\_count = sum([row.count(-1) for row in self.board])

        if player == 1:

            return white\_count - black\_count

        else:

            return black\_count - white\_count

برای تعداد نود در هر دور یک تابع اضافه میکنیم که در هر دور اجرای تابع play به ما نود های دیده شده را میدهد :

def get\_number\_of\_moves\_for\_each\_round(self):

        moves = self.get\_valid\_moves(self.current\_turn)

        return len(moves)

یک تابع محاسبه تایم نیز اضافه میکنیم :

def main():

    othello = Othello(8, 1, False)

    start\_time = time.time()

    winner = othello.play()

    end\_time = time.time()

    print(f"Winner: {winner}")

    print(f"Execution time: {end\_time - start\_time} seconds")

برای عمق 1 :



برای عمق 3 :



برای عمق 5 :



که خب برای زمان اجرا معقول است که افزایش داشته باشیم.

بعد از اجرای کد به دفعات متوجه میشویم که minimax ساده به علت اینکه همیشه بهترین گزینه را انتخاب میکند نمیتواند در برابر تابع random نتیجه مطلوبی بدهد و بعد از تعداد زیادی اجرا به برد زیر 50% میرسیم.

حال برای بهبود این موضوع از ما خواسته شده که از روش هرس استفاده کنیم که نتیجه و زمان را بهبود میبخشد و تعداد نود های اضافی را هرس میکند. برای این کار توابع جدیدی پیاده سازی میکنیم :

 def calculate\_the\_best\_move(self, player):

        moves = self.get\_valid\_moves(player)

        if len(moves) == 0:

            return None

        best\_move = None

        best\_score = -math.inf

        for move in moves:

            board\_copy = copy.deepcopy(self.board)

            self.make\_move(player, move)

            score = self.minimax(-player, self.minimax\_depth, -math.inf, math.inf)

            if score > best\_score:

                best\_score = score

                best\_move = move

            self.board = board\_copy

        return best\_move

    def minimax(self, player, depth, alpha, beta):

        if depth == 0:

            return self.evaluate(player)

        moves = self.get\_valid\_moves(player)

        if len(moves) == 0:

            return self.evaluate(player)

        if player == 1:

            best\_score = -math.inf

            for move in moves:

                board\_copy = copy.deepcopy(self.board)

                self.make\_move(player, move)

                score = self.minimax(-player, depth - 1, alpha, beta)

                best\_score = max(best\_score, score)

                alpha = max(alpha, best\_score)

                self.board = board\_copy

                if self.prune and alpha >= beta:

                    break

            return best\_score

        else:

            best\_score = math.inf

            for move in moves:

                board\_copy = copy.deepcopy(self.board)

                self.make\_move(player, move)

                score = self.minimax(-player, depth - 1, alpha, beta)

                best\_score = min(best\_score, score)

                beta = min(beta, best\_score)

                self.board = board\_copy

                if self.prune and alpha >= beta:

                    break

            return best\_score

    def evaluate(self, player):

        score = 0

        for i in range(self.size):

            for j in range(self.size):

                if self.board[i][j] == player:

                    score += 1

                elif self.board[i][j] == -player:

                    score -= 1

        return score

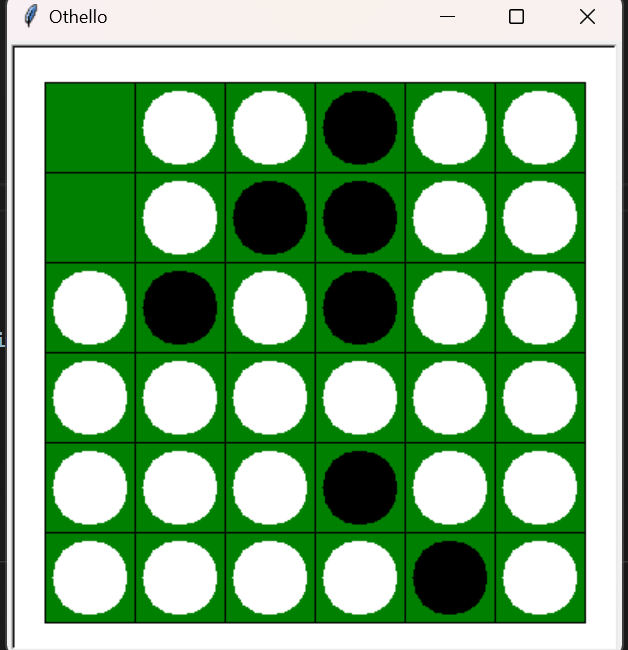
    def get\_best\_move(self):

        return self.calculate\_the\_best\_move(1)

در این توابع از هرس و alpha beta pruning برای یافتن جواب استفاده کردیم و روش حل مشابه روش توضیح داده شده در مراجع است.

حال در عمق های 5 و 7 اجرا میگیریم :

در عمق 5 میبینیم که به شدت قوی ظاهر شد و شکل نهایی به صورت زیر شد :

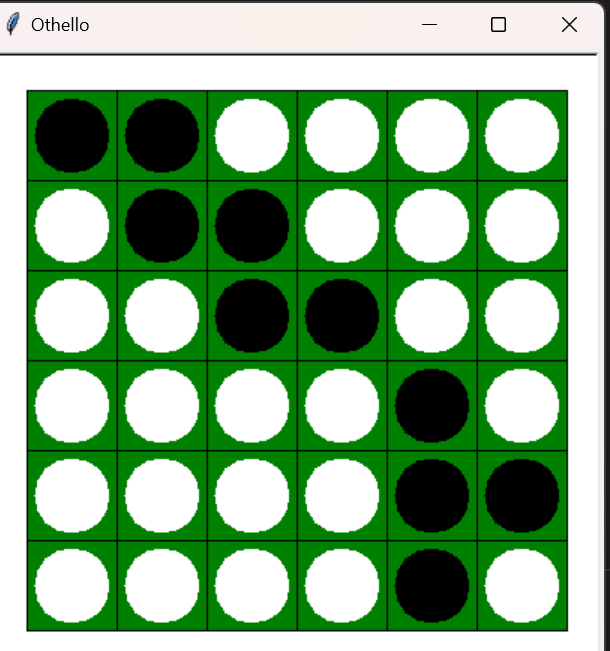


همچنین تایم اجرا :



حال در عمق 7 تست میکنیم :

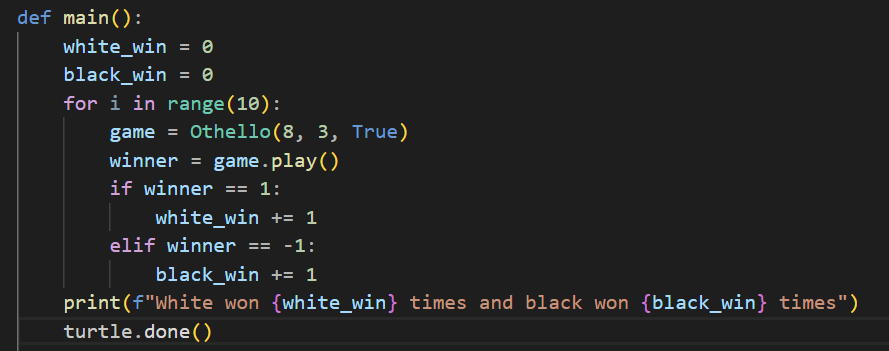
میبینیم که تمامی simulation ها به برد ما میرسد که یک نمونه ی آن شکل زیر است :



و همچنین زمان اجرا به علت محاسبات بیشتر افزایش چشمگیری داشته :



برای محاسبه ی درصد برد در بازی برای حالت alpha beta pruning یک main جدید با لوپ 10 تایی تعریف میکنیم و بعد از اتمام کار تعداد برد سفید(AI ما) و سیاه را چاپ میکنیم :



در اینجا به عنوان مثال برای حالت هرس عمق 3 داده شده است.

برای عمق 3 داریم :



برای عمق 5 داریم :

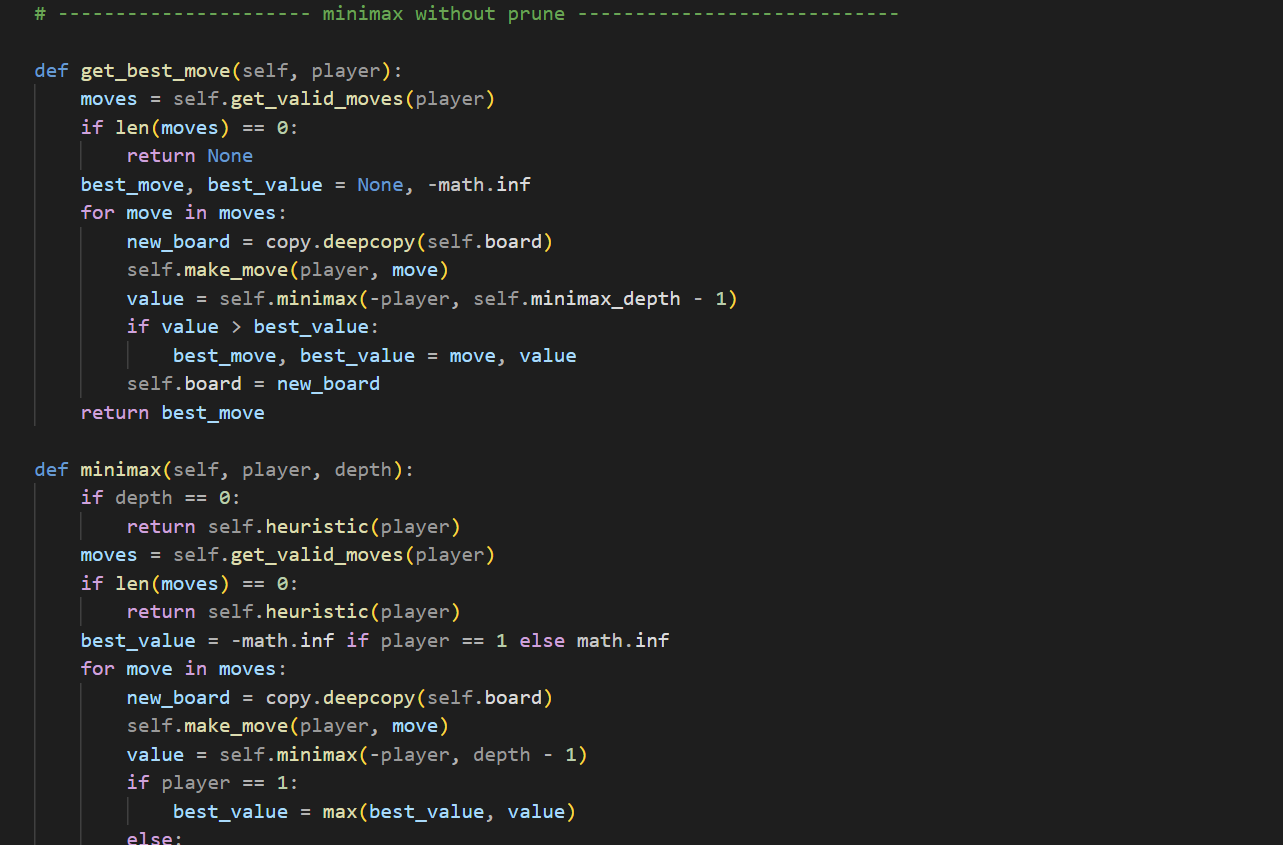


برای عمق 7 بعد از 2 بار اجرا داریم :



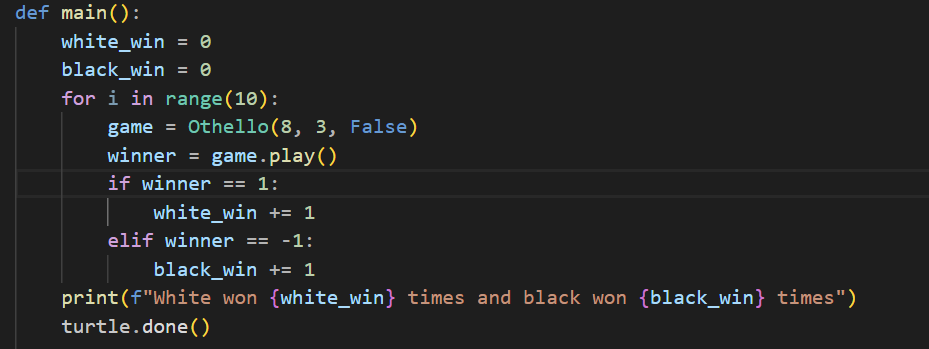


حال تابع اول که minimax عادی ما بود را از کامنت دراورده و این بار با آن اجرا میکنیم.



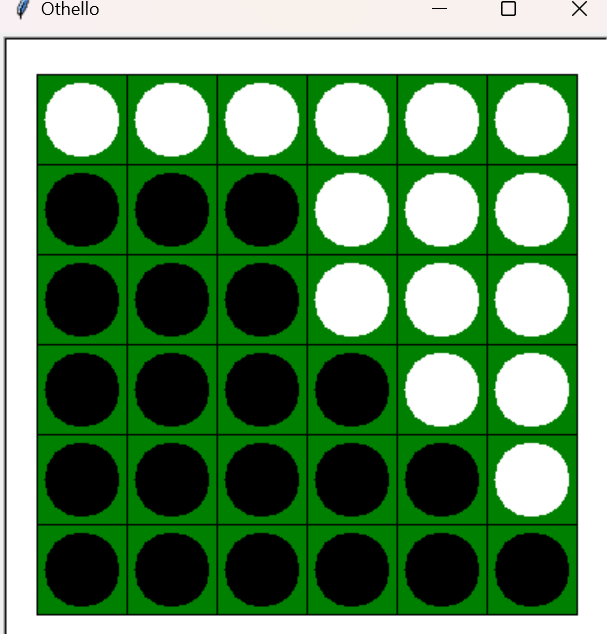
کد uncomment شده.

در main نیز True برای هرس کردن به False تبدیل شده :



حال برای عمق های 3 و 5 و 7 ران میگیریم :

عمق 3 :



یک نمونه از باخت ما به تابع random .

خروجی به صورت زیر شد :



در عمق 5 :



در عمق 7 : ( بعد از 3 بار اجرا)







جواب سوالات :

1) موارد محاسبه به این صورت است که با یک تابع میاییم و تمامی حالت های ممکن را جایگذاری میکنیم و سپس حالتی که در آن بالاترین امتیاز را داشتیم به عنوان حرکت بعدی انتخاب میکنیم.

2) بله رابطه ای وجود دارد. هرچه عمق الگوریتم بیشتر باشد ما میتوانیم حالت های بیشتری را بررسی کنیم و در نتیجه پیش بینی ما دقیق تر خواهد بود و هیوریستیک بهتری خواهیم داشت در نتیجه الگوریتم ما شانس برد بیشتری دارد. اما زمان و تعداد گره های دیده شده افزایش میابند.

3) بله، می‌توانیم ترتیب دیدن فرزندان هر نود در هرس الفا بتا را به گونه‌ای انتخاب کنیم که در هرس الفا بتا بیشترین هرس را داشته باشیم.

یک روش ساده برای این کار این است که فرزندان هر نود را بر اساس مقدار بتای آن‌ها مرتب کنیم و سپس آن‌ها را به ترتیب از بزرگ به کوچک در هرس الفا بتا بچینیم. برای مثال، اگر فرض کنیم داریم یک درخت دو سطحی با مقادیر زیر داریم:

5

/ \

2 7

اولین گام این است که فرزندان هر نود را بر اساس مقدار بتای آن‌ها مرتب کنیم:

5

/ \

2 7

سپس، برای هر یال در هرس الفا بتا، فرزند با بیشترین مقدار بتایی که دارد را به سمت بالایی از نود بزرگتر منتقل می‌کنیم. در این حالت، این مرتبه برداری به همراه هرس الفا بتا، به درخت زیر منجر خواهد شد:

7

/ \

2 5

بنابراین، با توجه به مثال بالا، می‌توانیم ترتیب دیدن فرزندان هر نود را به گونه‌ای انتخاب کنیم که در هرس الفا بتا بیشترین هرس را داشته باشیم.

4) ضریب انشعاب گراف را گویند که در این بازی در هر عمق مقدار آن تعداد حرکت های مجاز ما در هر مرحله بازی میباشد. با پیشرفت بازی مقدار آن ابتدا افزایش میابد زیرا انتخاب های بیشتری خواهیم داشت اما در انتهای بازی مقدار آن کاهش داشته و حتی میتواند در صورت باخت زودهنگام به صفر برسد.

5) در هرس الفا بتا، الگوریتم به دنبال برش حداقلی از درخت بازی است که باعث می شود از تعداد حالت های قابل محاسبه کاسته شود. با هر برشی که در درخت انجام می شود، بخش هایی از فضای جستجو حذف می شوند که به نظر می رسد بازیکن حریصانه ترین انتخاب را خواهد داشت. به عبارت دیگر، الگوریتم با هر برش، با حذف بخش های بیشتری از فضای جستجو، می تواند به سرعت تر برای پیدا کردن بهترین حرکت پیش برود.

علاوه بر این، در هر برش، تعداد حالت هایی که باید مورد بررسی قرار گیرند کاهش می یابد، به عبارتی الگوریتم برای بررسی کردن کمترین تعداد حالت ها نیاز به زمان کمتری دارد. به همین دلیل، با هر برش، الگوریتم به دنبال حل بهترین حرکت با دقت یکسان خود می شود اما با سرعت بیشتر. بنابراین، با هر برش در هرس الفا بتا، الگوریتم به دنبال بهترین حرکت با حفظ دقت خود به صورت سریعتر پیش می رود.

6) هنگام بازی با یک بازیکن تصادفی، الگوریتم مینیمکس ممکن است موثرترین استراتژی نباشد زیرا فرض می‌کند که حریف همیشه بهترین حرکت ممکن را انجام خواهد داد و زمانی که حریف به صورت تصادفی بازی می‌کند، این فرض درست نیست و الگوریتم ممکن است در نهایت حرکات غیربهینه انجام دهد و نتیجه ی مطلوبی ندهد.