به نام خدا

دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده برق و کامپیوتر

هوش مصنوعی پاییز ۹۸

پروژه دو

بازی

نام و نام خانوادگی

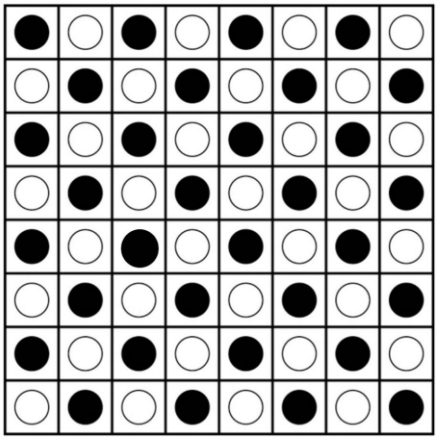
علیرضا زارع نژاد اشکذری

شماره دانشجویی

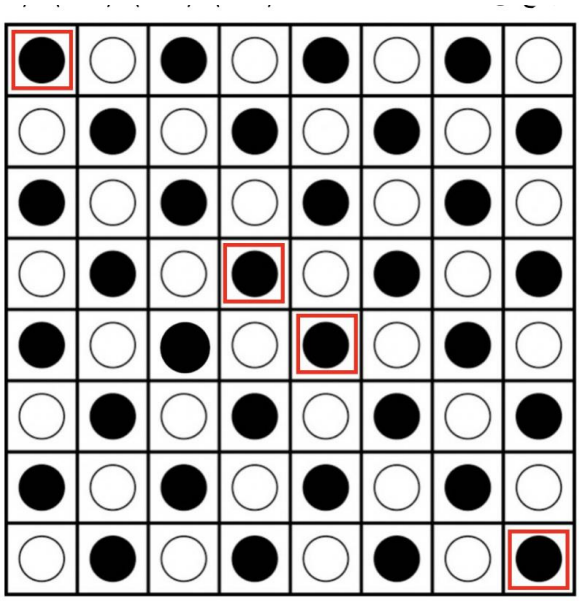
۸۱۰۱۹۶۴۷۴

شرح مختصر پروژه

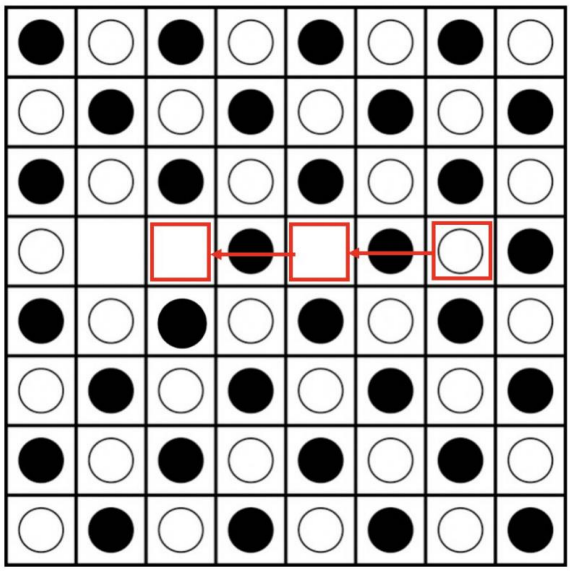
در این بازی یک صفحه ۸*۸ شامل مهره‌های سفید و سیاه به شکل زیر داده شده است.



در ابتدا بازیکن اول که پیش‌فرض بازیکن سیاه مد نظر است یکی از مهره‌های خانه‌های مشخص شده زیر را حذف می کند.



در حرکت بعدی بازیکن مقابل می بایست یکی از خانه‌های مجاور خانه ی حذف شده در مرحله قبل را حذف کند. سپس بازی آغاز می‌شود. در هر مرحله بازیکن می‌تواند مهره خود را از روی مهره حریف عبور دهد و مهره ی حریف را حذف نماید به شرط آنکه فضای خالی موجود باشد. همچنین به شرط آنکه مسیر حرکت عوض نشود بازیکن می‌تواند چندین بار حرکت کند.به عنوان مثال شکل زیر را در نظر می گریم.



بازی زمانی تمام می‌شود که بازیکن حرکتی برای انجام نداشته باشد.

بررسی کد ها و توابع آماده در پروژه

- متد **reset**: شرایط برد را به شرایط شروع باز می‌گرداند.
- متد **contains**: اگر سطر و ستون داده شده نشانگر یک مکان معتبر در صفحه باشد true را برمی‌گرداند.
- متد **opponent**: رنگ حریف را برمی‌گرداند.
- متد **distance**: فاصله بین دو نقطه را در یک خط عمودی یا افقی روی صفحه برمی گرداند. پرش های مورب مجاز نیست.
- متد **makeMove**: صفحه فعلی را با صفحه بعدی ایجاد شده توسط این حرکت به روز کنید.
- متد **nextBoard**: با توجه به حرکت یک بازیکن از (r1 ,c1) به (r2, c2) ، این حرکت را روی نسخه‌ای از صفحه فعلی اجرا می کند. در صورت عدم اعتبار حرکت ، یک GameError رخ می‌دهد. این متد یک کپی از صفحه را برمی گرداند ، و صفحه مورد نظر را تغییر نمی دهد.
- متد **openingMove**: در صورتی که حداکثر یکی از بازیکنان حرکت کرده باشد true برمی‌گرداند.
- متد **generateFirstMoves**: حرکات اختصاصی را برای حرکت اول بازی را برمی‌گرداند.
- متد **generateSecondMoves**: حرکات خاص را برای حرکت دوم بازی بر اساس جایی که اولین حرکت در آن اتفاق افتاده است برمی‌گرداند.
- متد **check**: بررسی می کند که آیا پرش با شروع از (r,c) و رفتن به جهتی که توسط دلتای ردیف (rd) و دلتای ستون (cd) انجام می شود ، امکان پذیر است.از factor برای بررسی بازگشتی جهش های متعدد در همان جهت استفاده می شود و همه‌ی جهش‌های ممکن را در جهت مشخص برمی گرداند.
- متد **generateMoves**: با استفاده از پیکربندی صفحه فعلی، تمام حرکت‌های قانونی را برای بازیکن داده شده ایجاد و برمی گرداند.

گام پروژه

یک کلاس به نام MinimaxPlayer را پیاده سازی کنید که از کلاس Player و کلاس Game ارث‌بری کند. کلاس شما باید متد initialize و getMove را از کلاس Player پیاده‌سازی کند.

ابتدا agentخود را به شکلی بنویسید که با استفاده از درخت Minimax حرکت بعدی خود را مشخص کند و از یک evauation function خوب استفاده کنید. در ادامه برای بهبود agent خود از alpha beta prunning برای هرس کردن درخت‌تان استفاده کنید تا تصمیم‌های بهتری بگیرد.

لذا تنها کافی است دو کلاس گفته شده را طراحی کنیم و روند آن در زیر توضیح داده شده است.

کلاس MinmaxPlayer

```
class MinMaxPlayer(Player, Game):
    def __init__(self, n, max_depth=3):
        super().__init__(n)
        self.max_depth = max_depth

    def initialize(self, side):
        self.side = side
        self.name = "MinMaxPlayer"

    def evaluate_function(self, board):
        moves = self.generateMoves(board, self.side)
        return len(moves)

    def getMove(self, board):
        return self.utill(board, self.side, 1)

    def utill(self, board, cur_side, depth=0):
        my_side = (cur_side == self.side)

        if depth == self.max_depth:
            return self.evaluate_function(board)
        else:
            moves = self.generateMoves(board, cur_side)
            chosen_move = []
            if my_side:
                best_value = -math.inf
                for move in moves:
                    new_board = self.nextBoard(board, cur_side, move)
                    evaluated_value = self.utill( new_board , self.opponent(cur_side), depth + 1)
                    if evaluated_value > best_value:
                        best_value = evaluated_value
                        chosen_move = move
                if depth == 1:
                    return chosen_move
                else:
                    return best_value
            else:
                best_value = math.inf
                for move in moves:
                    new_board = self.nextBoard(board, cur_side, move)
                    evaluated_value = self.utill( new_board, self.opponent(cur_side), depth + 1 )
                    if evaluated_value < best_value:
                        best_value = evaluated_value
                        chosen_move = move
                if depth == 1:
                    return chosen_move
                else:
                    return best value
```

با توجه به قطعه کد بال برای agent مان constructorای بمی نویسیم که حداکثر depth را در زمان ساختن آن به constructor بدهیم. زمان ساختن درخت، آن را تا این ارتفاع ساخته و زمانی که به این عمق رسیدیم دیگر درخت را نمی‌سازیم و از evaluation function مناسب استفاده می‌کنیم.این عمق در متغیر self.depth ذخیره شده است.در مورد evalution function نیز تعداد حرکات ممکن مرحله مورد نظر نعریف می کنیم.

در تابع utill نیز که مشخص می‌کند با متغیر cur_side که الان در لول max هستیم یا در لول min و حال اگر به عمق مورد نظر تعریف شده رسیده باشیم یعنی به برگ‌ها کافی است که مقدار eval function برگردانده شود در غیر این صورت باید اگر در لول max هستیم ماکسیمم مقدار node های بچه را برگردانده و آن حرکت را انجام دهیم و اگر در لول min هستیم مینیمم مقدار node های بچه یا به عبارتی حرکت‌های ممکن را برگردانیم.که برای ای کار هر بار new_booard جدید را می‌سازیم و نوع بازیکن را عوض می‌کنیم و depth را یکی زیاد می کنیم. پس از ماکسیمم و مینیمم گیری مناسب حرکت مناسب برگردانده می شود.

کلاس AlphaBetaPlayer

```
class AlphaBetaPlayer(Player, Game):
    def __init__(self, n, max_depth=4):
        super().__init__(n)
        self.max_depth = max_depth

    def initialize(self, side):
        self.side = side
        self.name = "AlphaBetaPlayer"

    def evaluate_function(self, board):
        moves = self.generateMoves(board, self.side)
        return len(moves)

    def getMove(self, board):
        return self.utill(board, -math.inf, math.inf, self.side, 1)
```

```python
def utill(self, board, alpha, beta, cur_side, depth=1):
    our_turn = (cur_side == self.side)

    if self.max_depth < depth:
        return self.evaluate_function(board)

    else:
        moves = self.generateMoves(board, cur_side)
        chosen_move = []
        if our_turn:
            best_value = -math.inf
            for move in moves:
                new_board = self.nextBoard(board, cur_side, move)
                evaluated_value = self.utill( new_board, alpha, beta, self.opponent(cur_side), depth + 1 )
                if evaluated_value > best_value:
                    best_value = evaluated_value
                    chosen_move = move
                alpha = max(alpha, best_value)
                if beta < alpha:
                    break
            if depth == 1:
                return chosen_move
            else:
                return best_value
        else:
            best_value = math.inf
            for move in moves:
                new_board = self.nextBoard(board, cur_side, move)
                evaluated_value = self.utill(new_board, alpha, beta, self.opponent(cur_side), depth + 1)
                if evaluated_value < best_value:
                    best_value = evaluated_value
                    chosen_move = move
                beta = min(beta, best_value)
                if beta < alpha:
                    break
            if depth == 1:
                return chosen_move
            else:
                return best_value
```

برای این بخش دو پارامتر alpha و bata را به تابع مورد نظر که مشابه minmax هست می‌دهیم. آلفا نشان دهنده ی بیشترین مقدار utility پیدا شده برای بازیکن سیاه و بتا کمترین مقدار utility پیدا شده برای بازیکن سفید است.حال اگر در پیمایش یک نود (برای مثال برای آلفا) مقدار utility یکی از برگ‌های آن نود از الفا کمتر باشد می‌توانیم از پیماش سایر برگ‌های آن نود صرف نر کنیم چرا که در این حالت نود مینیمم مقدار برگ‌های خود را به عنوان یوتیلیتی انتخاب می‌کند و این مقدار عددی کمتر از بهترین یوتیلیتی پیدا شده برای آن نود (الفا) می‌باشد پس بنابراین پیمایش سایر برگ‌های آن نود تأثیری مثبتی نخواهد داشت.

آیا حرکات انجام این دو الگوریتم با یکدیگر متفاوت است؟

خیر – اگر عمق آن‌ها با هم یکی باشد حرکات آن‌ها با هم فرقی ندارد. زیرا تنها کاری که در الگوریتم alpah-beta انجام می‌دهیم این است که حرکاتی که به اندازه حرکات اصلی خوب نیستند را حذف می‌کنیم نه اینکه حرکات درست تا آن عمق را حذف کنیم.

زمان اجرایی دو الگوریتم را با یکدیگر مقایسه کنید؟

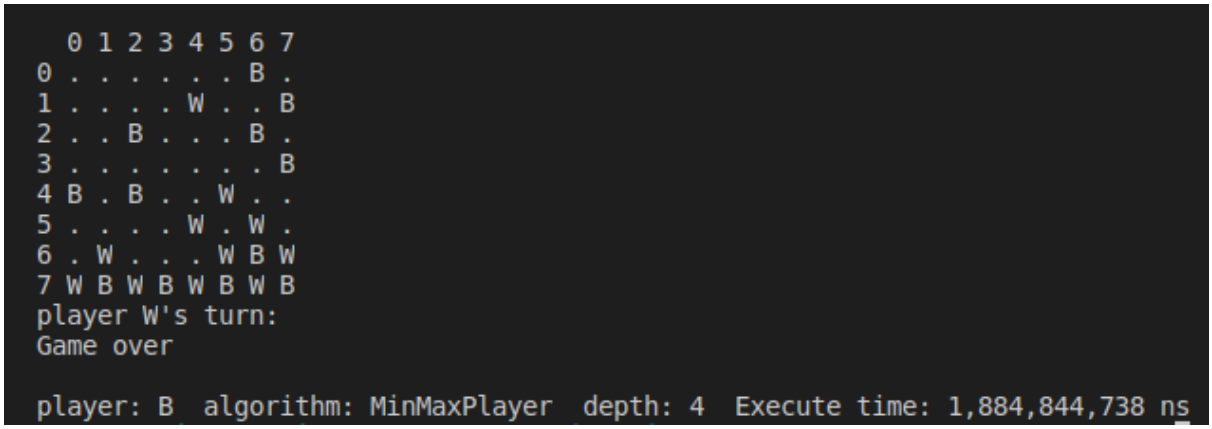
زمان اجرایی الگوریتم minmax به eval function وابسته است همچنین پس از هرس شدن شاخه‌های اضافی در alpha beta این مقدار زمانی در این الگوریتم کاسته می شود.

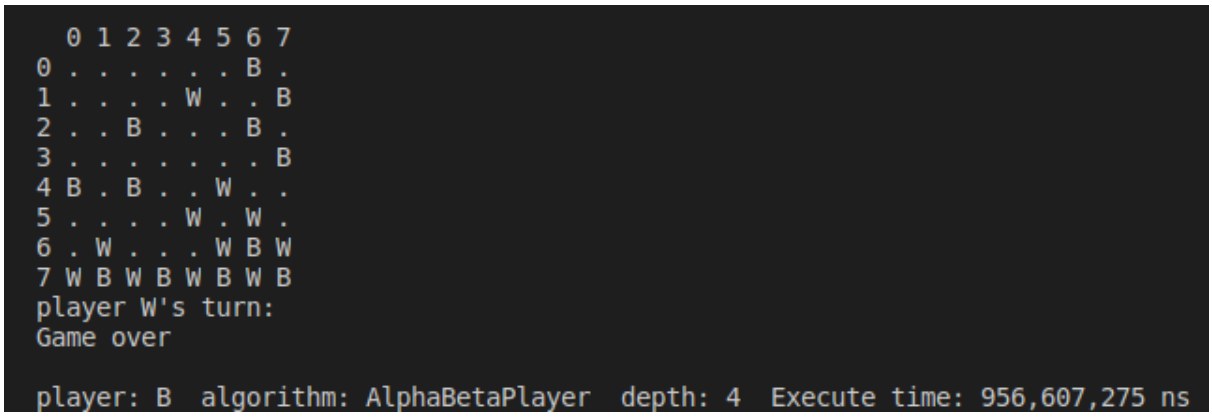
در اینجا حالت‌های مختلف بازی را تست کرده و زمان آن را بررسی می کنیم.

Eval را نهایتاً تعداد حرکت‌های خود منهای تعداد حرکت‌های حریف کردم.

در ابتدا خروجی به ازای عمقهای مختلف در دو حالت minmax و alpha beta با بازیکن simple player و زمان آن آورده می شود.

Depth = 4 , player1 : minmax , player2 : simple:





depth = 5 , player1 : minmax , player2 : simple :

```
  0 1 2 3 4 5 6 7
0 B . . . . . . W
1 . . . . . . . .
2 B . . W . . . W
3 . . . . W . . .
4 . . . . . . . .
5 . . . B . . . B
6 . . . . . . . .
7 W . . B . . . B
player W's turn:
Game over

player: B  algorithm: MinMaxPlayer  depth: 5  Execute time: 12,206,453,623 ns
```

```
  0 1 2 3 4 5 6 7
0 B . . . . . . W
1 . . . . . . . .
2 B . . W . . . W
3 . . . . W . . .
4 . . . . . . . .
5 . . . B . . . B
6 . . . . . . . .
7 W . . B . . . B
player W's turn:
Game over

player: B  algorithm: AlphaBetaPlayer  depth: 5  Execute time: 2,668,104,297 ns
```

depth = 6 , player1 = minmax , player2 = simple

```
  0 1 2 3 4 5 6 7
0 B . . . B . . .
1 . B . B . . . B
2 B . B . . W . .
3 . B . . . . . .
4 . . . W . W . .
5 W . W . W B W .
6 B . . W B W B W
7 W B W B W B W B
player W's turn:
Game over

player: B  algorithm: MinMaxPlayer  depth: 6  Execute time: 94,284,408,697 ns
```

```
  0 1 2 3 4 5 6 7
0 B . . . B . . .
1 . B . B . . . B
2 B . B . . W . .
3 . B . . . . . .
4 . . . W . W . .
5 W . W . W B W .
6 B . . W B W B W
7 W B W B W B W B
player W's turn:
Game over

player: B  algorithm: AlphaBetaPlayer  depth: 6  Execute time: 10,467,785,418 ns
```

حال حالتی را در نظر بگیریم که بازیکن اول mimmax player باشد و بازیکن دوم alpha beta و به ازای عمق های مختلف خروجی به صورت زیر است

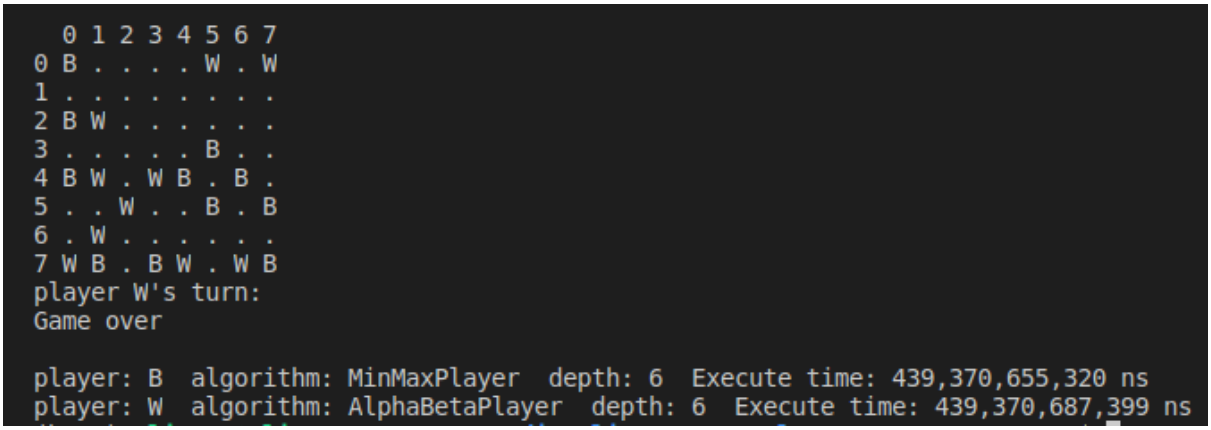
: player1 : minmax , player2 : alphabeta , depth = 4 ,

```
  0 1 2 3 4 5 6 7
0 . W . . . . . W
1 . . W . . B . B
2 . . . W . . . .
3 . . . . . . . B
4 B . . . . . . .
5 . . W . . . . B
6 B . . . . . B W
7 W B W . . B W B
player B's turn:
Game over

player: B  algorithm: MinMaxPlayer  depth: 4  Execute time: 2,666,278,493 ns
player: W  algorithm: AlphaBetaPlayer  depth: 4  Execute time: 2,666,317,547 ns
```

: depth = 5 , player1 :minmax , player2 : alphabeta

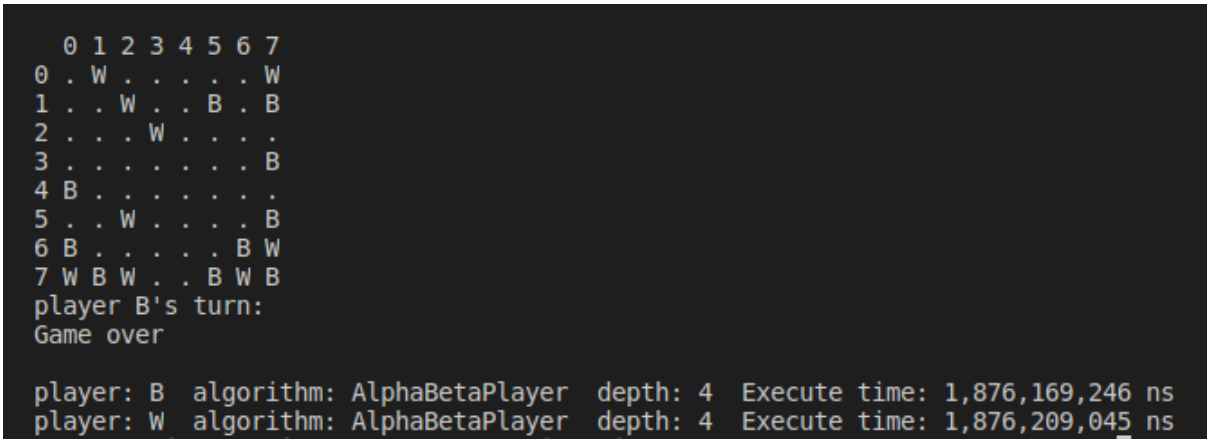
```
  0 1 2 3 4 5 6 7
0 B . B . B . . W
1 W . . B . . . B
2 B . . . . W B W
3 W B W . W . . B
4 B . B . . W . W
5 W . W . . B W B
6 B . . W . W . W
7 W . . B . B . B
player W's turn:
Game over

player: B  algorithm: MinMaxPlayer  depth: 5  Execute time: 18,350,483,459 ns
player: W  algorithm: AlphaBetaPlayer  depth: 5  Execute time: 18,350,514,643 ns
```

: depth = 6 , player1 : minmax , player2 : alphabeta



حال حالتی در نظر بگیریم که جفت بازیکن ها از alpha beta استفاده کنند.

: Depth = 4



: depth = 5

```
  0 1 2 3 4 5 6 7
0 B . B . B . . W
1 W . . B . . . B
2 B . . . . W B W
3 W B W . W . . B
4 B . B . . W . W
5 W . W . . B W B
6 B . . W . W . W
7 W . . B . B . B
player W's turn:
Game over

player: B  algorithm: AlphaBetaPlayer  depth: 5  Execute time: 5,409,209,619 ns
player: W  algorithm: AlphaBetaPlayer  depth: 5  Execute time: 5,409,218,585 ns
```

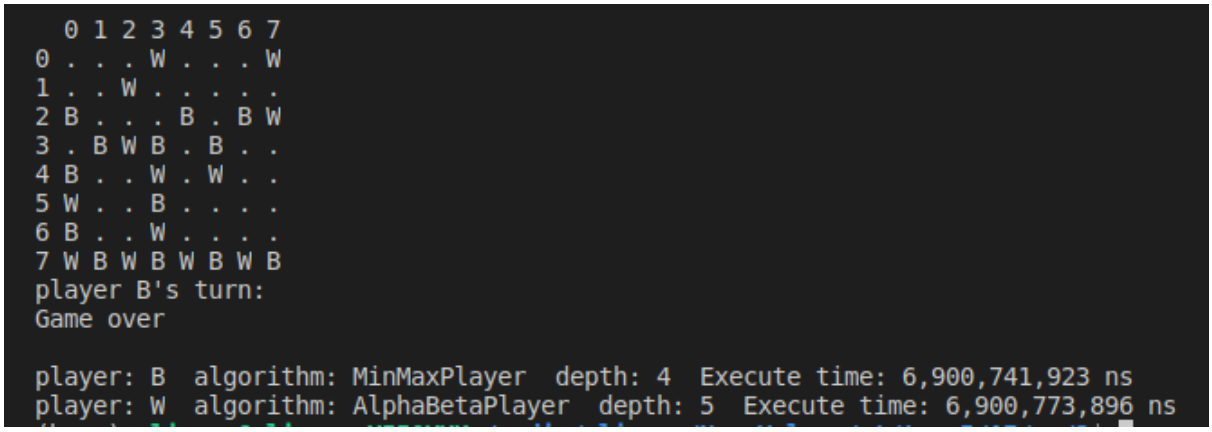
و در صورتی که جفت بازیکن ها از minmax استفاده کنند:

depth = 4

```
  0 1 2 3 4 5 6 7
0 . W . . . . . W
1 . . W . . B . B
2 . . . W . . . .
3 . . . . . . . B
4 B . . . . . . .
5 . . W . . . . B
6 B . . . . . B W
7 W B W . . B W B
player B's turn:
Game over

player: B  algorithm: MinMaxPlayer  depth: 4  Execute time: 3,741,204,298 ns
player: W  algorithm: MinMaxPlayer  depth: 4  Execute time: 3,741,237,742 ns
```

و نهایتاً حالتی که عمق جست‌وجو فرق کند:

:player1 : minmax , depth1 = 4 , player2 : alphabeta , depth2 = 5



واضح است هر کدام که عمق بیشتری داشته باشند برنده می باشد.

پایان