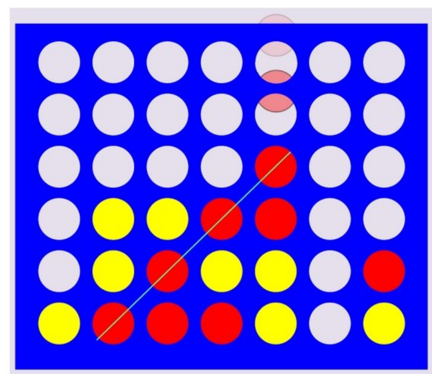
پروژه پایانی درس (قسمت الف)

کیمیا اسماعیلی - 610398193

مقدمه:

بازی کانکت 4 (connect-4)، اتصال چهارتایی، بازی‌ای شبیه به دوز) از بوردگیم‌های محبوب و قدیمی‌ای است که راهبرد ساده‌ای هم دارد. دو بازیکن بر یک صفحه عمودی بازی می‌کنند و هر کدام یک رنگ برای دیسک‌های خود انتخاب میکند و معمولا 21 دیسک از آن رنگ دارد. هر کدام از بازیکنان به نوبت دیسک رنگ انتخاب‌شده خود را رو داخل صفحه بازی 7*6 قرار میدهند. اولین کسی که بتواند 4 دیسک با رنگ یکسان در یک ردیف یا ستون یا قطر قرار دهد، برنده بازی است.



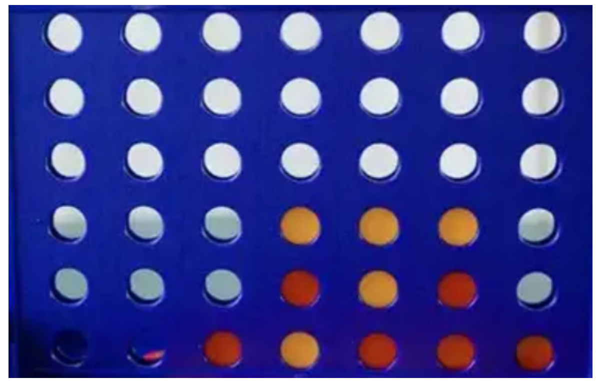
بازی کانکت 4:

این بازی به عنوان یک بازی zero-sum دسته‌بندی میشود. به این دلیل، الگوریتم مینیماکس که یک قانون تصمیم‌گیری در مطالعات هوش مصنوعی است، میتواند راه حل خوبی باشد و در این بین درخت تصمیم توسط الگوریتم مینیماکس در این بازی استفاده میشود.

استراتژی های برد:

1. **استفاده از خانه ستون وسط:**
   اگر بازیکن حرکت اول را دارد، بهتر است که دیسک خود را در ستون وسطی قرار دهد. از آنجایی که صفحه 7 ستون دارد، قرار دادن دیسکها در وسط به ما این امکان را میدهد تا اتصال ما به صورت عمودی، افقی و قطری باشد که 5 حالت برای ما به وجود می آورد.
2. **درست کردن تله برای رقیب:**
   یک راه معمول برای نباختن آن است که راه رقیب را برای بردن ببندیم. برای مثال، از اینکه رقیبمان بتواند تعداد دیسکهای خود در یک خط را افزایش دهد، به وسیله بستن راه او با کمک دیسکهای خودمان، جلوگیری کنیم. این استراتژی همچنین از تله گذاری رقیب برای ما جلوگیری میکند.

3. درست کردن "7":
"7-تله" نام یک حرکت استراتژیک است که در آن یک بازیکن دیسکهای خود را در حالتی قرار میدهد که شبیه به یک 7 شوند. این حالت به صورتی است که سه دیسک به صورت افقی‌متصل از راست‌ترین دیسک، به دو دیسک قطری‌متصل، وصل شده اند. این 7 میتواند به هر صورتی درست شود مثلا برعکس یا قرینه یا... . این حالت دیسکها استراتژی خوبی است زیرا به بازیکن این شانس را میدهد که در جهات متنوعی بتواند 4 دیسک را ردیف کند.

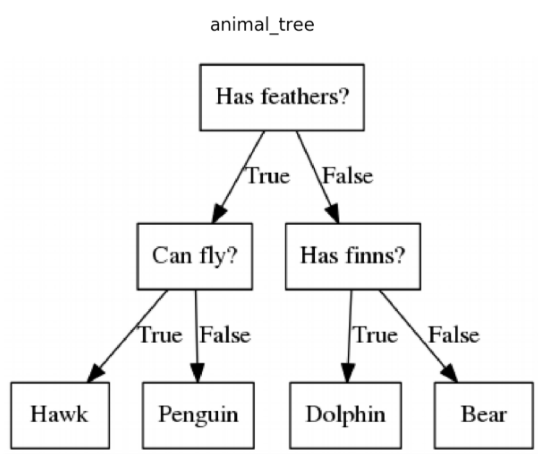


در اینجا دیسکهای زرد یک 7 درست کرده اند.

منطق و ریاضیات پشت‌صحنه این بازی:

- **درخت تصمیم:**

یک درخت تصمیم یک ساختار درختی است که در آن هر گره درونی نمایانگر یک تست بر یک اتربیوت است. هر شاخه نمایانگر یک خروجی از آن تست است و هر گره برگی (گره ترمینال) یک لیبل کلاس دارد. در مثال زیر یک فلوی ممکن این است: اگر حیوان پر داشته باشد و نتواند پرواز کند، پنگوئن است. فلوی دیگر: حالا اگر حیوان پر نداشته باشد و دارای باله باشد، دلفین است.

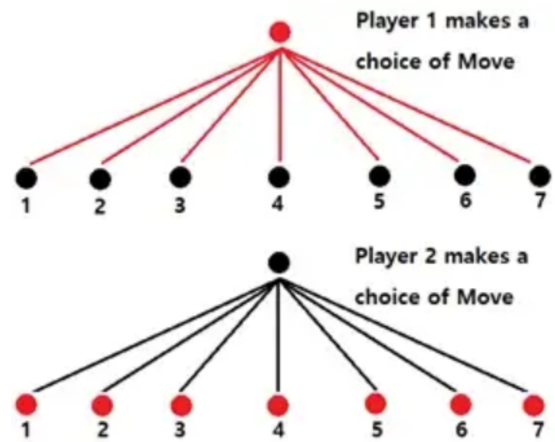


درخت تصمیم در مطالعات متنوعی میتواند به کار برده شود، مانند برنامه های استراتژیک تجاری، ریاضیات و... . علاوه بر آن، از آنجایی که درخت تصمیم تمام انتخاب های ممکن را

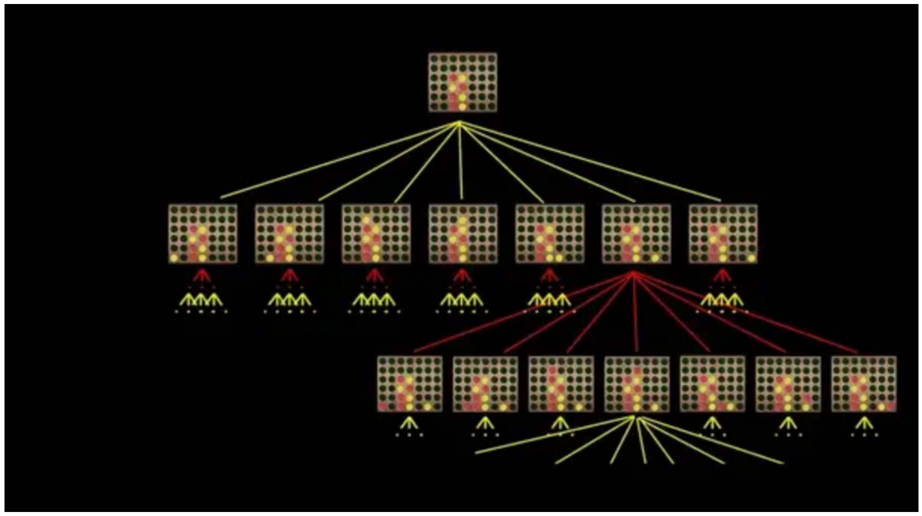
نمایش میدهد، میتواند در بازی های منطقی مانند کانکت 4 بهره گیری شود تا به عنوان یک جدول look-up از آن استفاده کنیم.

- **درخت تصمیم در کانکت فور:**

زمانی که بازی شروع میشود، اولین بازیکن این امکان را دارد که یک ستون از هفت ستون ممکن را انتخاب کند تا دیسک خود را در آن قرار دهد. هفت ستون، به ما هفت شاخه در هر زمان در درخت تصمیم به ما میدهند. بعد از آن که بازیکن اول یک حرکت انجام داد، بازیکن دوم میتواند یک ستون از هفتتا در پیروی از انتخاب بازیکن اول در درخت تصمیم، انتخاب کند. توجه داریم که درخت تصمیم با مواردی خاص عملیات خود را ادامه میدهد. ابتدا اگر هر دو بازیکن یک ستون مشابه را 6 بار انتخاب کنند، این ستون دیگر برای هیچ بازیکنی در دسترس نیست، یعنی از شاخه های قابل انتخاب آنها یکی کم میشود. ثانیا اگر دو بازیکن همه انتخاب ها را کرده باشند (42 خانه پر شده باشد) و همچنان هیچ چهار دیسک با رنگ مشابهی در یک خط نیستند، بازی با تساوی تمام میشود و درخت تصمیم متوقف میشود. نهایتا اگر هر بازیکن 4 دسک خود را در یک خط ردیف کند، درخت متوقف میشود و بازی با برد آن بازیکن تمام میشود.



نمایش حرکات ممکن برای هر ایتریشن بازی کانکت 4 در درخت تصمیم



درخت تصمیم حرکات ممکن کانکت 4

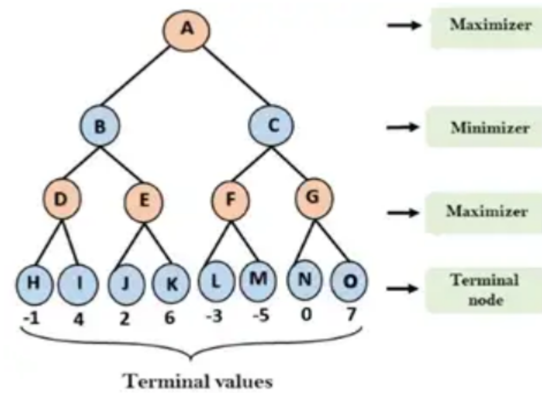
الگوریتمMiniMax:

الگوریتم مینیماکس (درخت کمینه بیشینه یا درخت بازی) یک الگوریتم بازگشتی است که در تصمیم‌گیری و نظریه بازیها و هوش مصنوعی به وفور استفاده میشود. این الگوریتم حرکات بهینه برای بازیکن بدست می آورد (با فرض اینکه رقیب ما هم به صورت بهینه بازی میکند). برای مثال دو رقیب min و max را در نظر میگیریم: ماکس سعی میکند مقادیر را بشینه کند در صورتی که مین هر مقداری که کمینه است را انتخاب میکند. الگوریتم یک depth-first search (DFS)k اجرا میکند که بدین معناست کل درخت بازی را تا بیشترین عمق ممکن تا گره های برگی جستجو میکند. سودوکد الگوریتم را در زیر میبینید:

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

به طور اولیه الگوریتم کل درخت بازی را ساخته و مقادیر utility را برای استیت های ترمینال، به وسیله به کارگیری تابع utility، بدست می آورد. برای مثال در نمودار درختی زیر A را استیت اولیه درخت در نظر میگیریم. فرض میکنیم که ماکسیمایزر اولین حرکت را میکند و این یک مقدار اولیه worst-case را

به همراه دارد که مساوی با بینهایت منفی است. سپس مینیمایزر حرکت خود را انجام میدهد که مقدار اولیه worst-case را به همراه دارد که مساوی با بینهایت مثبت است.



الگوریتم مینیماکس به صورت درخت - مرحله اول

ابتدا مینیمایزر را با مقدار اولیه ∞- فرض میکنیم. هر گره ترمینال را با مقدار ماکسیمایزر مقایسه میکنیم و نهایتا مقدار ماکسیمم را در هر گره ماکسیمایزر ذخیره میکنیم. سومین ردیف (ماکسیمایزر) از بالا را برای مثال در نظر میگیریم.

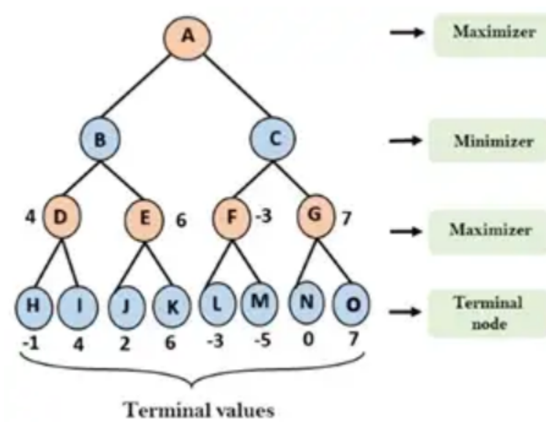
برای گره D $\max(-1, -\infty) \rightarrow \max(-1,4) = 4$

برای گره E $\max(2, -\infty) \rightarrow \max(2, 6) = 6$

برای گره F $\max(-3, -\infty) \rightarrow \max(-3,-5) = -3$

برای گره G $\max(0, -\infty) \rightarrow \max(0, 7) = 7$

مرحله دوم:

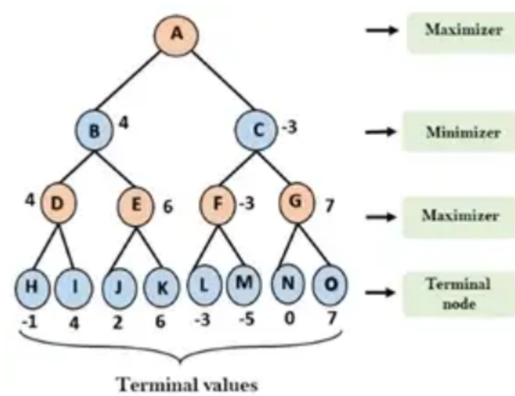


سپس مقادیر را از هر گره با مقدار مینیمایزر مقایسه میکنیم که ∞+ است.

برای گره $B= \min(4, 6) = 4$

برای گره $C= \min(-3, 7) = -3$

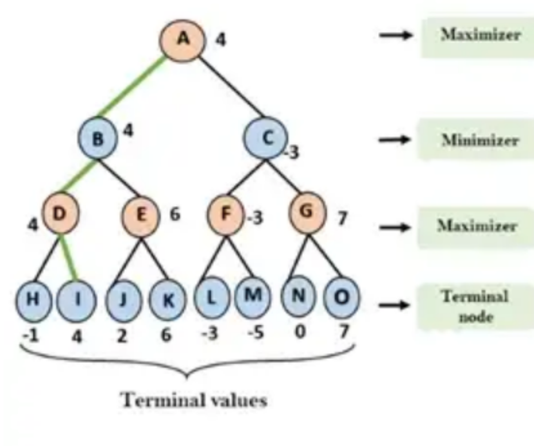
مرحله سوم:



نهایتا ماکسیمایزر دوباره مقدار ماکسیمم را بین گره B و C انتخاب میکند که در این وضعیت 4 است. به این ترتیب، ما مسیر بهینه بازی را بدست می آوریم: $A \rightarrow B \rightarrow D \rightarrow I$

برای گره A max(4, -3)= 4

مرحله آخر:



الگوریتم هرس آلفا-بتا:

الگوریتمی است که کارایی الگوریتم مینیماکس را بهبود می‌بخشد. با استفاده از هرس آلفا-بتا، بخش‌هایی از درخت کمینه بیشینه که پیمایششان بی تأثیر است پیمایش نمی‌شوند و به این ترتیب پیمایش درخت کمینه بیشینه تا یک عمق مشخص در زمانی کمتر صورت می‌گیرد. برای بررسی ایده هرس آلفا-بتا این دو مسئلهٔ مشابه را در نظر میگیریم:

- تعدادی زیر مجموعهٔ ناتهی و متناهی از مجموعهٔ اعداد حقیقی در اختیار داریم. ارزش (یا امتیاز) هر یک از این مجموعه‌ها را برابر با کوچک‌ترین عضو آن تعریف می‌کنیم. هدفمان یافتن مجموعه‌ای با بیش‌ترین ارزش است. فرض کنید که ارزش یکی از مجموعه‌ها برابر با m است. در این صورت مجموعه‌ای که دست کم یک عضو کوچک‌تر از m داشته باشد، پاسخ مسئله نخواهد بود. پس نیازی به بررسی اعضای این مجموعه (و یافتن ارزش آن) نیست. (چرا که ارزش آن کوچک‌تر از m است)
- همان پرسش بالا را این گونه تغییر می‌دهیم: ارزش هر مجموعه برابر با بزرگ‌ترین عضو آن تعریف می‌شود و هدف یافتن کم ارزش‌ترین مجموعه است. در این حالت نیز اگر مجموعه‌ای با ارزش m وجود داشته باشد مجموعه‌هایی که حداقل یک عضو بزرگ‌تر از m دارند، نمی‌توانند پاسخ مسئله باشند.

از همین ایده می‌توان برای بهینه کردن پیمایش روی درخت کمینه بیشینه بهره جست. شکل زیر را به عنوان بخشی از یک درخت کمینه بیشینه در نظر بگیرید:

فرض کنید ارزش (امتیاز) راس سبز را برابر با بیش‌ترین ارزش نسبت داده شده به فرزندان آن راس تعریف کنیم و (مطابق با تعریف درخت کمینه بیشینه) ارزش راس قرمز برابر با کم‌ترین ارزش نسبت داده شده به فرزندان آن راس تعریف شود. حال اگر هدف یافتن ارزش راس سبز باشد، نیازی به پیمایش زیر درخت راس قرمز و یافتن ارزش این راس نیست. چرا که ارزش راس قرمز حداکثر برابر ۳ می‌باشد در حالی که راس سبز فرزندی با ارزش ۵ دارد.

هوش مصنوعی در کانکت 4: پیاده‌سازی مینیماکس:

در زیر یک سودوکد از الگوریتم مینیماکس و پیاده‌سازی آن برای کانکت 4 داریم. در کد ما الگوریتم اصلی مینیماکس را با اضافه کردن استراتژی هرس آلفا-بتا پیشرفته تر کرده‌ایم تا بتوانیم سرعت محاسبات را بیشتر و حافظه کمتری استفاده کنیم. سودوکد:

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cut-off *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if β ≤ α then
                break (* α cut-off *)
        return value
```

این قسمت از کد:

```python
def minimax(board, depth, alpha, beta, maximizingPlayer):
    valid_locations = get_valid_locations(board)
    is_terminal = is_terminal_node(board)
    if depth == 0 or is_terminal:
        if is_terminal:
            if winning_move(board, AI_PIECE):
                return (None, 100000000000000)
            elif winning_move(board, PLAYER_PIECE):
                return (None, -10000000000000)
            else: # Game is over, no more valid moves
                return (None, 0)
        else: # Depth is zero
            return (None, score_position(board, AI_PIECE))
    if maximizingPlayer:
        value = -math.inf
        column = random.choice(valid_locations)
        for col in valid_locations:
            row = get_next_open_row(board, col)
            b_copy = board.copy()
            drop_piece(b_copy, row, col, AI_PIECE)
            new_score = minimax(b_copy, depth-1, alpha, beta, False)[1]
            if new_score > value:
                value = new_score
                column = col
            alpha = max(alpha, value)
```

```python
            if alpha >= beta:
                break
        return column, value

    else: # Minimizing player
        value = math.inf
        column = random.choice(valid_locations)
        for col in valid_locations:
            row = get_next_open_row(board, col)
            b_copy = board.copy()
            drop_piece(b_copy, row, col, PLAYER_PIECE)
            new_score = minimax(b_copy, depth-1, alpha, beta, True)[1]
            if new_score < value:
                value = new_score
                column = col
            beta = min(beta, value)
            if alpha >= beta:
                break
        return column, value
```

ما maximizingPlayer را از کد به عنوان مثال در نظر میگیریم. ابتدا، برنامه به لوکیشن های معتبر از هر ستون نگاه میکند، سپس به طور بازگشتی امتیاز جدید را در جدول look-up محاسبه میکند و نهایتا مقدار بهینه را از گره های فرزند آپدیت میکند. در نظر داریم که آلفا اینجا new_score است و زمانی که از مقدار فعلی بزرگتر است، عملیات بازگشتی را متوقف میکند و مقدار جدید را آپدیت میکند تا زمان و حافظه کمتری استفاده کند.

همانطور که گفته شد، جدول look-up طبق تابع evaluate_window زیر محاسبه میشود. اندازه window را چهار گذاشتیم چون به دنبال اتصال چهار دیسک هستیم. با در نظر گرفتن یک روش امتیازدهی برای بازی، اگر چهار دیسک متصل باشند، یک امتیاز مثبت میدهیم (در اینجا 100 امتیاز میدهیم). زمانی که 3 دیسک متصل باشند، امتیازی کمتر از حالت چهارتا اتصال دارد. زمانی که دو دیسک متصل داریم، باز امتیازی کمتر از حالت سه اتصال داریم.نهایتا زمانی که رقیب 3 دیسک متصل دارد، بازیکن با گرفتن امتیاز منفی تنبیه میشود که نمایانگر آن است که این حرکت برای بازیکن فعلی بهینه نبوده است.

```python
def evaluate_window(window, piece):
        score = 0
        opp_piece = PLAYER_PIECE
        if piece == PLAYER_PIECE:
                opp_piece = AI_PIECE

        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 window.count(opp_piece) == 3 and window.count(EMPTY) == 1:
                score -= 4

        return score
```

با مجموعه امتیازاتی که به وجود می‌آوریم، برنامه نیاز دارد همه امتیازات برای هر حرکت ممکن برای هر بازیکن در زمان بازی را محاسبه کند. تابع score_position این قسمت از کد پایین را اجرا میکند. بازیکن هوش مصنوعی باید از این تابع بهره جوید تا حرکت بهینه را پیش‌بینی کند.

```python
def score_position(board, piece):
    score = 0

    ## Score center column
    center_array = [int(i) for i in list(board[:, COLUMN_COUNT//2])]
    center_count = center_array.count(piece)
    score += center_count * 3

    ## Score Horizontal
    for r in range(ROW_COUNT):
            row_array = [int(i) for i in list(board[r,:])]
            for c in range(COLUMN_COUNT-3):
                    window = row_array[c:c+WINDOW_LENGTH]
                    score += evaluate_window(window, piece)

    ## Score Vertical
    for c in range(COLUMN_COUNT):
            col_array = [int(i) for i in list(board[:,c])]
            for r in range(ROW_COUNT-3):
                    window = col_array[r:r+WINDOW_LENGTH]
                    score += evaluate_window(window, piece)
```

```
## Score posiive sloped diagonal
for r in range(ROW_COUNT-3):
        for c in range(COLUMN_COUNT-3):
                window = [board[r+i][c+i] for i in range(WINDOW_LENGTH)]
                score += evaluate_window(window, piece)

for r in range(ROW_COUNT-3):
        for c in range(COLUMN_COUNT-3):
                window = [board[r+3-i][c+i] for i in range(WINDOW_LENGTH)]
                score += evaluate_window(window, piece)

return score
```

نتایج بدست آمده:

با توجه با تعداد دفعاتی که هوش مصنوعی انسان را در این بازی شکست داده است، در میابیم که این بردها با منطق و حجم زیادی از اطلاعات پردازش‌شده است. در اینجا بازیکن هوش مصنوعی از یک الگوریتم مینیماکس استفاده میکند تا حرکات بهینه را از قبل شناسایی کند و بازیکن انسان را با دانستن همه حرکات ممکن به طور منطقی، شکست دهد. زمانی که عمق‌ها را در تابع مینیماکس از زیاد (مثلا 6) تا کم (مثلا 2) تنظیم کنیم، بازیکن هوش مصنوعی ممکن است اجرای بدتری داشته باشد. با این حال استراتژی و الگوریتم به کار رفته در این پروژه به طور عالی نتایج بهینه میدهد.