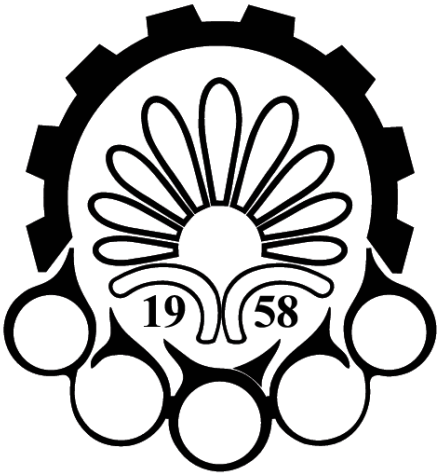



**Amirkabir University of Technology**
**(Tehran Polytechnic)**

گزارش ٤ درس هوش مصنوعی

استاد: دکتر مهدی قطعی

نام دانشجو: مریم علیپور

شماره دانشجویی: ۹۶۱۲۰۳۷

**چکیده:** الگوریتم Minimax الگوریتمی نسبتاً ساده است که برای تصمیم‌گیری بهینه در تئوری بازی و هوش مصنوعی استفاده می‌شود. باز هم، از آنجا که این الگوریتم‌ها بیشتر به کارآیی خود متکی هستند، می‌توان با استفاده از هرس آلفا ـ بتا عملکرد الگوریتم را بسیار بهبود بخشید ـ در این گزارش قرار است بازی دوز را با هرس آلفا ـ بتا پیاده سازی کنیم.

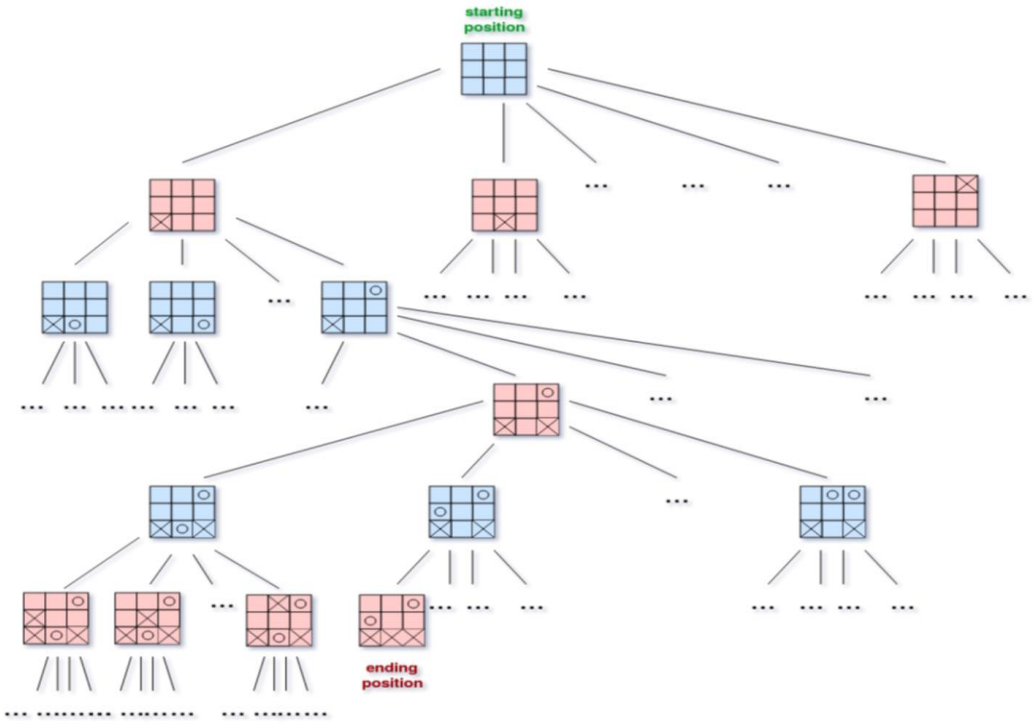
**کلمات کلیدی:** هرس آلفا ـ بتا، الگوریتم ابتکاری، الگوریتم شبیه سازی تبرید، روش تاگوچی.

## ۱ـ مقدمه:

Minimax یک الگوریتم بازگشتی است که برای انتخاب یک حرکت بهینه برای یک بازیکن استفاده می‌شود با این فرض که بازیکن دیگر نیز به طور بهینه بازی می‌کند. در بازی‌هایی مانند tic-tac-toe ، isola checkers ، chess ، go و بسیاری دیگر از بازی‌های دو نفره استفاده می‌شود. چنین بازی‌هایی را بازی با اطلاعات کامل می‌نامند زیرا دیدن تمام حرکات احتمالی یک بازی خاص امکان‌پذیر است.
ممکن است بازی‌های دو نفره وجود داشته باشد که اطلاعات کاملی ندارند زیرا حرکت حریف را نمی توان پیش بینی کرد.

## ۲ـ الگوریتم Minimax چگونه کار می‌کند؟

دو بازیکن درگیر یک بازی هستند، به نام MIN و MAX. بازیکن MAX سعی می‌کند بالاترین امتیاز ممکن را کسب کند و MIN سعی می‌کند که MAX کمترین امتیاز ممکن را بدست آورد، یعنی MIN و MAX سعی می‌کنند مخالف یکدیگر عمل کنند.
روند کلی الگوریتم Minimax به شرح زیر است:
گام اول: ابتدا تمام درخت بازی را با موقعیت فعلی بازی تا حالت ترمینال تولید کنید. اینگونه درخت درخت برای بازی tic-tac-toe به نظر می رسد.




گام دوم: برای بدست آوردن مقادیر سودمندی برای تمام حالت‌های ترمینال، تابع سودمندی را اعمال کنید.

گام سوم: سودمندی گره‌های بالاتر را با کمک سودمندی گره‌های ترمینال تعیین کنید.

گام چهارم: مقادیر سودمندی را با در نظر گرفتن یک لایه تا زمان رسیدن به ریشه درخت با کمک برگ‌ها محاسبه کنید.

گام پنجم: در نهایت، تمام مقادیر محاسبه شده به ریشه درخت، یعنی بالاترین نقطه می رسد. در آن مرحله، MAX باید بالاترین مقدار را انتخاب کند.

## ۳ـ هرس آلفا ـ بتا:

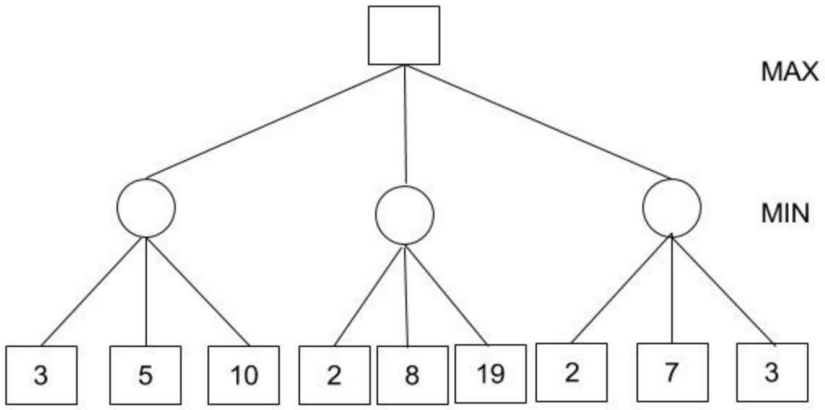
روشی که می‌خواهیم در این مقاله بررسی کنیم، هرس آلفا بتا نام دارد. اگر هرس آلفا ـ بتا را به یک الگوریتم استاندارد minimax اعمال کنیم، همان حرکت استاندارد را برمی‌گرداند، اما همه گره‌هایی را که احتمالاً تاثیری در تصمیم نهایی ندارند حذف می‌کند.

آلفا: این بهترین انتخاب تاکنون برای بازیکن MAX است. ما می‌خواهیم بیشترین مقدار ممکن را در اینجا بدست آوریم.

بتا: این بهترین انتخاب تاکنون برای MIN است و باید کمترین مقدار ممکن باشد.

## ۴ـ الگوریتم هرس آلفاـ بتا چگونه کار می‌کند؟

۱-مقدار آلفا = ـ بی نهایت و بتا = بی نهایت را به عنوان بدترین موارد ممکن آغاز کنید. شرط هرس یک گره زمانی است که آلفا از بتا بزرگتر یا مساوی شود.




۲ـ با اختصاص مقادیر اولیه آلفا و بتا به ریشه شروع کنید و از آنجا که آلفا کمتر از بتا است ، آن را هرس نمی کنیم.

۳ـ این مقادیر آلفا و بتا را به گره فرزند در سمت چپ منتقل کنید. و اکنون از مقدار سودمندی حالت ترمینال، مقادیر alpha و beta را به روز می‌کنیم، بنابراین لازم نیست مقدار بتا را به روز کنیم. باز هم، ما هرس نمی‌کنیم زیرا شرایط ثابت است. به همین ترتیب، برای گره فرزند سوم نیز.

## ۵ـ نتیجه گیری

هرس آلفا بتا تفاوت عمده‌ای در ارزیابی درختان بزرگ و پیچیده بازی ایجاد می‌کند. حتی اگر tic-tac-toe یک بازی ساده باشد، اما هنوز هم می‌توان فهمید که چگونه بدون ابتکار عمل آلفاـ بتا، الگوریتم زمان بیشتری را برای توصیه حرکت در اولین نوبت طول می‌کشد.

بازی‌ها بسیار جذاب هستند و نوشتن برنامه‌های بازی نیز هیجان انگیز هستند. همانطور که انتظار نداریم ماشین مسابقه ای کاملاً در جاده‌ای پر از دست انداز کار کند، نباید انتظار داشته باشیم که الگوریتم‌های بازی برای هر شرایطی مناسب باشند. الگوریتم minimax نیز همینطور است. شاید بهترین راه حل برای انواع بازی‌های رایانه‌ای که نیاز به هوش مصنوعی دارند نباشد. اما با اجرای خوب، می‌تواند رقیبی سرسخت ایجاد کند.

## ۶ـ پیاده سازی بازی دوز (Tic Tac Toe)

۱ـ با کانستراکتور برد اصلی بازی را میسازیم و تعداد سطر و ستون بازی و هدف(طول دوز) ست میکنیم.

```python
import sys
import random

class TicTacToeGame:

    def __init__(self, rows:int, columns:int, goal:int, max_depth:int=4):
        # Create the game state
        self.state = []
        self.tiles = {}
        self.inverted_tiles = {}
        tile = 0
        for y in range(rows):
            row = []
            for x in range(columns):
                row += '.'
                tile += 1
                self.tiles[tile] = (y, x)
                self.inverted_tiles[(y, x)] = tile
            self.state.append(row)
        # Set the number of noughts and crosses in a row that is needed to win the game
        self.goal = goal
        # Create vectors
        self.vectors = [(1,0), (0,1), (1,1), (-1,1)]
        # Set lengths
        self.rows = rows
        self.columns = columns
        self.max_row_index = rows - 1
        self.max_columns_index = columns - 1
        self.max_depth = max_depth
        # Heuristics for cutoff
        self.winning_positions = []
        self.get_winning_positions()
        # Set the starting player at random
        #self.player = 'O'
        self.player = random.choice(['X', 'O'])

```

## ۲ـ پیدا کردن موقعیت برنده بازی

```python
    def get_winning_positions(self):
        # Loop the board
        for y in range(self.rows):
            for x in range(self.columns):

                # Loop vectors
                for vector in self.vectors:

                    # Get the start position
                    sy, sx = (y, x)
                    # Get vector deltas
                    dy, dx = vector
                    # Create a counter
                    counter = 0
                    # Loop until we are outside the board
                    positions = []
                    while True:
                        # Add the position
                        positions.append(self.inverted_tiles.get((sy, sx)))
                        # Check if we have a winning position
                        if (len(positions) == self.goal):
                            # Add winning positions
                            self.winning_positions.append(positions)
                            # Break out from the loop
                            break
                        # Update the position
                        sy += dy
                        sx += dx

                        # Check if the loop should terminate
                        if(sy < 0 or abs(sy) > self.max_row_index or sx < 0 or abs(sx) > self.max_columns_index):
                            break

```

## ۳ـ نمایش(پرینت) برد بازی در ترمینال

```python
    # Print the current game state
    def print_state(self):
        for y in range(self.rows):
            print('| ', end='')
            for x in range(self.columns):
                if (self.state[y][x] != '.'):
                    print(' {0}  | '.format(self.state[y][x]), end='')
                else:
                    digit = str(self.inverted_tiles.get((y,x))) if len(str(self.inverted_tiles.get((y,x)))) > 1 else ' ' + str(self.inverted_tiles.get((y,x)))
                    print('{0}  | '.format(digit), end='')
            print()
        print()
```

## ٤ـ لوپ اصلی بازی

```
73      def play(self):
74          # Variables
75          result = None
76          # Create an infinite loop
77          print('Starting board')
78          while True:
79              # Draw the state
80              self.print_state()
81              # Get a move from a player
82              if (self.player == 'X'): # AI
83                  # Print AI move
84                  print('Player X moving (AI) ...')
85                  # Get the best move
86                  max, py, px, depth = self.max(-sys.maxsize, sys.maxsize)
87                  # Get a heuristic move at cutoff
88                  print('Depth: {0}'.format(depth))
89                  if(depth > self.max_depth):
90                      py, px = self.get_best_move()
91                  # Make a move
92                  self.state[py][px] = 'X'
93                  # Check if the game has ended, break out from the loop in that case
94                  result = self.game_ended()
95                  if(result != None):
96                      break
97                  # Change turn
98                  self.player = 'O'
99              elif (self.player == 'O'): # Human player
100
101                 # Print turn
102                 print('Player O moving (Human) ...')
103                 # Get a recommended move
104                 min, py, px, depth = self.min(-sys.maxsize, sys.maxsize)
105                 # Get a heuristic move at cutoff
106                 print('Depth: {0}'.format(depth))
107                 if(depth > self.max_depth):
108                     py, px = self.get_best_move()
109                 # Print a recommendation
110                 print('Recommendation: {0}'.format(self.inverted_tiles.get((py, px))))
111                 # Get input
112                 number = int(input('Make a move (tile number): '))
113                 tile = self.tiles.get(number)
114                 # Check if the move is legal
115                 if(tile != None):
116
117                     # Make a move
118                     py, px = tile
119                     self.state[py][px] = 'O'
120                     # Check if the game has ended, break out from the loop in that case
121                     result = self.game_ended()
122                     if(result != None):
123                         break
124                     # Change turn
125                     self.player = 'X'
126                 else:
127                     print('Move is not legal, try again.')
128         # Print result
129         self.print_state()
130         print('Winner is player: {0}'.format(result))
```

۵ـ تابع ارزیابی برای بدت آوردن بهترین حرکت ( بر اساس هیورستیک)

```python
def get_best_move(self):
    # Create an heuristic dictionary
    heuristics = {}
    # Get all empty cells
    empty_cells = []
    for y in range(self.rows):
        for x in range(self.columns):
            if (self.state[y][x] == '.'):
                empty_cells.append((y, x))
    # Loop empty positions
    for empty in empty_cells:
        # Get numbered position
        number = self.inverted_tiles.get(empty)
        # Loop winning positions
        for win in self.winning_positions:
            # Check if number is in a winning position
            if(number in win):
                # Calculate the number of X:s and O:s in the winning position
                player_x = 0
                player_o = 0
                start_score = 1
                for box in win:
                    # Get the position
                    y, x = self.tiles[box]
                    # Count X:s and O:s
                    if(self.state[y][x] == 'X'):
                        player_x += start_score if self.player == 'X' else start_score * 2
                        start_score *= 10
                    elif (self.state[y][x] == 'O'):
                        player_o += start_score if self.player == 'O' else start_score * 2
                        start_score *= 10
                # Save heuristic
                if(player_x == 0 or player_o == 0):
                    # Calculate a score
                    score = max(player_x, player_o) + start_score
                    # Update the score
                    if(heuristics.get(number) != None):
                        heuristics[number] += score
                    else:
                        heuristics[number] = score
    # Get the best move from the heuristic dictionary
    best_move = random.choice(empty_cells)
    best_count = -sys.maxsize
    for key, value in heuristics.items():
        if(value > best_count):
            best_move = self.tiles.get(key)
            best_count = value
    # Return the best move
    return best_move
```

۶ـ بررسی وضعیت بازی که آیا به پایان رسیده یا نه

```python
    def game_ended(self) -> str:
        # Check if a player has won
        result = self.player_has_won()
        if(result != None):
            return result
        # Check if the board is full
        for y in range(self.rows):
            for x in range(self.columns):
                if (self.state[y][x] == '.'):
                    return None
        # Return a tie
        return 'It is a tie!'
```

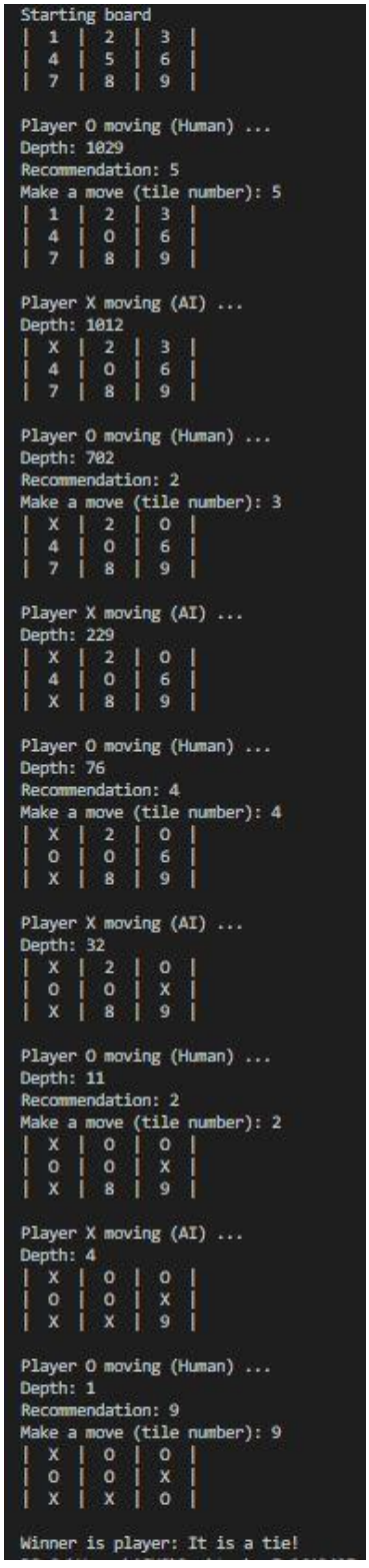
۷ـ پیدا کردن برنده بازی و returnکردن آن – در غیر این صورت بازی مساوی شده و tie برمیگردانیم

```python
    def player_has_won(self) -> str:
        # Loop the board
        for y in range(self.rows):
            for x in range(self.columns):

                # Loop vectors
                for vector in self.vectors:

                    # Get the start position
                    sy, sx = (y, x)
                    # Get vector deltas
                    dy, dx = vector
                    # Create counters
                    steps = 0
                    player_x = 0
                    player_o = 0
                    # Loop until we are outside the board or have moved the number of steps in the goal
                    while steps < self.goal:
                        # Add steps
                        steps += 1
                        # Check if a player has a piece in the tile
                        if(self.state[sy][sx] == 'X'):
                            player_x += 1
                        elif(self.state[sy][sx] == 'O'):
                            player_o += 1
                        # Update the position
                        sy += dy
                        sx += dx

                        # Check if the loop should terminate
                        if(sy < 0 or abs(sy) > self.max_row_index or sx < 0 or abs(sx) > self.max_columns_index):
                            break
                    # Check if we have a winner
                    if(player_x >= self.goal):
                        return 'X'
                    elif(player_o >= self.goal):
                        return 'O'
        # Return None if no winner is found
        return None
```

۸ـ تابع min و maxکه الگوریتم a-b pruning را روی درخت جستجو، اجرا میکنند

```python
    def min(self, alpha:int=-sys.maxsize, beta:int=sys.maxsize, depth:int=0):

        # Variables
        min_value = sys.maxsize
        by = None
        bx = None

        # Check if the game has ended
        result = self.game_ended()
        if(result != None):
            if result == 'X':
                return 1, 0, 0, depth
            elif result == 'O':
                return -1, 0, 0, depth
            elif result == 'It is a tie!':
                return 0, 0, 0, depth
        elif(depth > self.max_depth):
            return 0, 0, 0, depth
        # Loop the board
        for y in range(self.rows):
            for x in range(self.columns):
                # Check if the tile is empty
                if (self.state[y][x] == '.'):
                    # Make a move
                    self.state[y][x] = 'O'
                    # Get max value
                    max, max_y, max_x, depth = self.max(alpha, beta, depth + 1)

                    # Set min value to max value if it is lower than curren min value
                    if (max < min_value):
                        min_value = max
                        by = y
                        bx = x

                    # Reset the tile
                    self.state[y][x] = '.'
                    # Do an alpha test
                    if (min_value <= alpha):
                        return min_value, bx, by, depth
                    # Do a beta test
                    if (min_value < beta):
                        beta = min_value
        # Return min value
        return min_value, by, bx, depth


    def max(self, alpha:int=-sys.maxsize, beta:int=sys.maxsize, depth:int=0):
        # Variables
        max_value = -sys.maxsize
        by = None
        bx = None
        # Check if the game has ended
        result = self.game_ended()
        if(result != None):
            if result == 'X':
                return 1, 0, 0, depth
            elif result == 'O':
                return -1, 0, 0, depth
            elif result == 'It is a tie!':
                return 0, 0, 0, depth
        elif(depth > self.max_depth):
            return 0, 0, 0, depth
        # Loop the board
        for y in range(self.rows):
            for x in range(self.columns):
                # Check if the current tile is empty
                if (self.state[y][x] == '.'):

                    # Add a piece to the board
                    self.state[y][x] = 'X'
                    # Set max value to min value if min value is greater than current max value
                    min, min_y, min_x, depth = self.min(alpha, beta, depth + 1)
                    # Adjust the max value
                    if (min > max_value):
                        max_value = min
                        by = y
                        bx = x
                    # Reset the tile
                    self.state[y][x] = '.'
                    # Do a beta test
                    if (max_value >= beta):
                        return max_value, bx, by, depth
                    # Do an alpha test
                    if (max_value > alpha):
                        alpha = max_value
        # Return max value
        return max_value, by, bx, depth

```

۱۰ـ اجرای بازی: (مساوی شد)



## منابع:


https://medium.com/@hackerearth/minimax-algorithm-with-alpha-beta-pruning-e٧٧٧a٥beab٣٦

https://towardsdatascience.com/lets-beat-games-using-a-bunch-of-code-part-١-tic-tac-toe-١٥٤٣e٩٨١fec١

hhttps://github.com/deerishi/Tic-Tac-Toe-Using-Alpha-Beta-Minimax-Search

https://livebook.manning.com/book/classic-computer-science-problems-in-python/chapter-٨/٢٨

https://www.annytab.com/minimax-algorithm-in-python/