به نام خدا

گزارش تمرین سری هفتم هوش مصنوعی

محمد مهدی شرف بیانی

401521372

بخش اول : توضیح الگوریتم های ارایه شده

2)الگوریتم Minimax

```python
def minimax(state: TicTacToe, player: int) -> Tuple[int, Optional[Tuple[int, int]]]:
    # Check for terminal state
    winner = state.check_winner()
    if winner is not None:
        return winner, None  # Return the utility (1, -1, or 0) and no action

    available_moves = state.get_available_moves()
    best_action = None
    best_utility = -float('inf') if player == 1 else float('inf')  # Initialize best utility

    for coordinate in available_moves:
        # Make the move
        state.board[coordinate[0], coordinate[1]] = player

        # Recursive call
        utility, _ = minimax(state, -player)

        # Undo the move
        state.board[coordinate[0], coordinate[1]] = 0

        # Update best utility and action
        if (player == 1 and utility > best_utility) or (player == -1 and utility < best_utility):
            best_utility = utility
            best_action = coordinate

    # state.board[best_action[0], best_action[1]] = player
    return best_utility, best_action
```

توضیح الگوریتم :

الگوریتم Minimax یک روش بازگشتی برای تصمیم‌گیری در بازی‌های دو نفره است که در آن هر بازیکن به صورت متناوب حرکت می‌کند. هدف این الگوریتم این است که بهترین حرکت ممکن را پیدا کند که بازیکن را به بیشترین برد یا کمترین باخت برساند، در حالی که فرض می‌کند حریف همیشه بهترین حرکت را انجام می‌دهد.

نحوه کار:

درخت جستجو: تمامی حالات ممکن بازی از وضعیت فعلی به‌صورت یک درخت جستجو نمایش داده می‌شوند.

برگ‌ها (Terminal Nodes): هر شاخه تا زمانی گسترش می‌یابد که بازی تمام شود (برد، باخت یا مساوی).

امتیازدهی: در انتهای هر شاخه، امتیاز وضعیت بازی محاسبه می‌شود:

برد: امتیاز مثبت (مثلاً +1)

باخت: امتیاز منفی (مثلاً -1)

مساوی: امتیاز صفر.

بازگشت امتیاز: امتیاز هر حالت به سطح بالاتر درخت بازگشت داده می‌شود تا بهترین حرکت انتخاب شود.

بازیکنان max و min : بازیکن max سعی دارد بیشترین امتیاز را انتخاب کند، در حالی که حریف کمترین امتیاز ممکن را انتخاب می‌کند. تا امتیاز بازیکن max را کمینه کند و از آنجایی که بازی zero sum است پس اگر امتیاز حریف کمینه شود پس امتیاز آن بازیکن بیشینه میشود.

2) الگوریتم Alpha_Beta Pruning :

```
def alpha_beta(state: TicTacToe, player: int, alpha: float = -float('inf'), beta: float = float('inf')) -> Tuple[int, Optional[Tuple[int, int]]]:
    # Check for terminal state
    winner = state.check_winner()
    if winner is not None:
        return winner, None  # Return utility (1, -1, or 0) and no action

    available_moves = state.get_available_moves()
    best_action = None

    if player == 1:  # Maximizing player
        best_utility = -float('inf')
        for coordinate in available_moves:
            # Make the move
            state.board[coordinate[0], coordinate[1]] = player
            # Recursive call with the opponent's turn
            utility, _ = alpha_beta(state, -player, alpha, beta)
            # Undo the move
            state.board[coordinate[0], coordinate[1]] = 0
            # Update the best utility and action
            if utility > best_utility:
                best_utility = utility
                best_action = coordinate

            # Update alpha and prune if possible
            alpha = max(alpha, best_utility)
            if beta <= alpha:
                break  # Beta cut-off
    else:  # Minimizing player
        best_utility = float('inf')
        for coordinate in available_moves:
            # Make the move
            state.board[coordinate[0], coordinate[1]] = player

            # Recursive call with the opponent's turn
            utility, _ = alpha_beta(state, -player, alpha, beta)

            # Undo the move
            state.board[coordinate[0], coordinate[1]] = 0
            # Update the best utility and action
            if utility < best_utility:
                best_utility = utility
                best_action = coordinate
            # Update beta and prune if possible
            beta = min(beta, best_utility)
            if beta <= alpha:
                break  # Alpha cut-off

    # state.board[best_action[0], best_action[1]] = player

    return best_utility, best_action
```

توضیح الگوریتم :

الگوریتم هرس آلفا-بتا یک بهینه‌سازی برای الگوریتم Minimax است که با کاهش تعداد حالات مورد بررسی در درخت تصمیم، کارایی جستجو را بهبود می‌بخشد. این الگوریتم از دو مقدار **آلفا** و **بتا** استفاده می‌کند تا شاخه‌های غیرضروری در درخت را حذف کند، بدون اینکه نتیجه نهایی تحت تأثیر قرار گیرد.

مفاهیم کلیدی:

1. **آلفا:(Alpha)**
   - بهترین امتیازی که بازیکن حداکثر (MAX) تا کنون پیدا کرده است.
   - آلفا نشان‌دهنده حداقل امتیازی است که بازیکن MIN اجازه خواهد داد.
   - این مقدار از بالا به پایین در درخت منتقل می‌شود.
2. **بتا:(Beta)**
   - بهترین امتیازی که بازیکن حداقل (MIN) تا کنون پیدا کرده است.
   - بتا نشان‌دهنده حداکثر امتیازی است که بازیکن MAX اجازه خواهد داد.
   - این مقدار نیز از بالا به پایین در درخت منتقل می‌شود.
3. **هرس:**
   - اگر در حین بررسی یک شاخه مشخص شود که وضعیت آن هیچ تأثیری بر نتیجه نهایی نخواهد داشت (یعنی کمتر از آلفا یا بیشتر از بتا است)، آن شاخه بررسی نمی‌شود.

نحوه کار الگوریتم:

1) درخت جستجو:

الگوریتم همچنان مانند Minimax یک درخت جستجو را بررسی می‌کند، اما:

- از مقادیر آلفا و بتا برای محدود کردن جستجو استفاده می‌کند.
- در هر گره از درخت، مقادیر آلفا و بتا به‌روزرسانی می‌شوند.

2) فرآیند بازگشتی:

الگوریتم در هر گره به صورت زیر عمل می‌کند:

- **گره:MAX**
  - مقدار آلفا را به حداکثر مقدار ممکن به‌روزرسانی می‌کند.
  - اگر مقدار فعلی یک گره از بتای موجود بیشتر شود، ادامه جستجو در این شاخه متوقف می‌شود) زیرا بازیکن MIN هرگز این شاخه را انتخاب نمی‌کند.(
- **گره:MIN**
  - مقدار بتا را به حداقل مقدار ممکن به‌روزرسانی می‌کند.
  - اگر مقدار فعلی یک گره از آلفای موجود کمتر شود، ادامه جستجو در این شاخه متوقف می‌شود) زیرا بازیکن MAX هرگز این شاخه را انتخاب نمی‌کند.(

3) برگشت مقادیر:

پس از اتمام جستجوی هر شاخه، مقدار نهایی (حداکثر یا حداقل) به سطح بالاتر برگشت داده می‌شود.

مزایا:

1. **کاهش تعداد حالات مورد بررسی:**
   - در بهترین حالت، تعداد گره‌های بررسی‌شده به O(n)O(\sqrt{n})O(n)کاهش می‌یابد.
   - این باعث بهبود سرعت اجرای الگوریتم می‌شود.
2. **حفظ نتیجه صحیح:**
   - این الگوریتم تضمین می‌کند که بهترین حرکت دقیقاً همانند Minimax خواهد بود.
3. **کاربرد در بازی‌های پیچیده:**
   - به دلیل کاهش فضای جستجو، می‌توان از آن در بازی‌هایی با فضای حالت بزرگ‌تر استفاده کرد.

معایب:

1. **وابستگی به ترتیب شاخه‌ها:**
   - کارایی الگوریتم به ترتیب گسترش شاخه‌ها بستگی دارد. اگر بهترین شاخه‌ها ابتدا بررسی شوند، عملکرد بهتر خواهد بود.
   - در غیر این صورت، ممکن است تعداد زیادی گره غیرضروری بررسی شوند.
2. **پیاده‌سازی پیچیده‌تر:**
   - نسبت به الگوریتمMinimax ، به دلیل مدیریت مقادیر آلفا و بتا و انجام هرس، پیچیدگی بیشتری دارد.

کاربردها:

- بازی‌های استراتژی مانند شطرنج، دوز(Tic-Tac-Toe) ، و تخته‌نرد.
- بهینه‌سازی در تصمیم‌گیری‌های رقابتی.
- ترکیب با روش‌های جستجوی دیگر مانند MCTS برای کاهش فضای جستجو.

این الگوریتم در بازی‌هایی که نیازمند تصمیم‌گیری سریع و دقیق هستند بسیار کاربردی است و به دلیل کاهش شاخه‌های جستجو، می‌تواند بازی‌های پیچیده‌تر را نیز مدیریت کند.

: Monte_Carlo Tree Search (3

```
from math import sqrt, log
import random
from copy import deepcopy

class Node:
    """A node in the MCTS tree."""
    def __init__(self, state: 'TicTacToe', parent: Optional['Node'] = None, move:
Optional[Tuple[int, int]] = None):
        self.state = state
        self.total_value = 0  # Total value of all simulations through this node
        self.visits = 0       # Number of visits to this node
        self.move = move      # Move that led to this state
        self.children: List[Node] = []  # Child nodes
        self.parent = parent  # Parent node

def rollout(node: Node, current_player: int,player) -> float:
        """
        Simulate a random game from the current node until completion.
        Returns the reward value of the simulation.
        """
        # Check if game is already finished
        winner = node.state.check_winner()
        if winner == player:
            return 1.0
        elif winner == -player:
            return -1.0
        elif winner == 0:  # Draw
            return 0.0

        # Get available moves and simulate random play
        moves = node.state.get_available_moves()
        if not moves:
            return 0.0

        # Make a random move and continue simulation
        row, col = random.choice(moves)
        new_state = deepcopy(node.state)
        new_state.board[row, col] = current_player
        next_node = Node(new_state, parent=node, move=(row, col))
        return rollout(next_node, -current_player,player)

def backpropagate(node: Node, value: float) -> None:
        """
        Update node statistics on the path from a leaf node to the root.
        """
        while node is not None:
            node.visits += 1
            node.total_value += value
            node = node.parent
```

```
def uct_score(node: Node, parent_visits: int, exploration_constant: float = sqrt(2)) ->
float:
        """
        Calculate the Upper Confidence Bound for Trees (UCT) score for a node.
        Balances exploitation (first term) and exploration (second term).
        """
        if node.visits == 0:
            return float('inf')

        exploitation = node.total_value / node.visits
        exploration = exploration_constant * sqrt(log(parent_visits) / node.visits)
        return exploitation + exploration


def select_best_child(node: Node) -> Node:
        """Select the child node with the highest UCT score."""
        return max(node.children, key=lambda child: uct_score(child, node.visits))

def monte_carlo_tree_search(state: 'TicTacToe', player: int, num_simulations: int = 100) -
> Tuple[int, int]:

    root = Node(state)

    # Main MCTS loop
    for _ in range(num_simulations):
        # Selection: traverse tree until we reach a node that needs expansion
        current = root
        while current.children and all(child.visits > 0 for child in current.children):
            current = select_best_child(current)

        # Expansion: if node has been visited and game isn't over, expand by adding all
possible child nodes
        if current.visits > 0:
            available_moves = current.state.get_available_moves()
            if available_moves and not current.children:
                for move in available_moves:
                    new_state = deepcopy(current.state)
                    new_state.board[move[0], move[1]] = player
                    current.children.append(Node(new_state, parent=current, move=move))
                current = random.choice(current.children)

        # Simulation: perform rollout from current node
        simulation_result = rollout(current, player,player)

        # Backpropagation: update statistics for all nodes in path
        backpropagate(current, simulation_result)

    # Return best move based on most visited child
    if not root.children:
        return random.choice(state.get_available_moves())

    best_child = max(root.children, key=lambda child: child.visits)
    return 0,best_child.move
```

توضیح الگوریتم :

الگوریتم مونت کارلو یکی از روش‌های محبوب برای تصمیم‌گیری در بازی‌ها و حل مسائل پیچیده است.

مراحل اصلی الگوریتم مونت کارلو

MCTS به‌طور کلی از چهار مرحله تشکیل شده است:

1. انتخاب (Selection):

- الگوریتم از ریشه درخت شروع می‌کند و گره‌هایی را که قبلاً بازدید شده‌اند، بر اساس معیارهایی مانند UCB1 (Upper Confidence Bound) انتخاب می‌کند.
- هدف این مرحله یافتن گره‌ای است که ارزش بیشتری برای گسترش داشته باشد.

$$\sqrt{\frac{\ln(\text{Parent Visits})}{\text{Node Visits}}} \cdot C + \frac{\text{Wins}}{\text{Visits}} = UCB1$$

- Wins تعداد بردهای گره.
- Visits: تعداد بازدیدهای گره.
- Parent Visits : تعداد بازدیدهای والد.
- C: ضریب اکتشاف که تعادل بین کاوش (Exploration) و بهره‌برداری (Exploitation) را تنظیم می‌کند.

2. گسترش (Expansion):

- اگر گره انتخاب‌شده یک حالت پایانی (Win, Lose, Draw) نباشد، یک یا چند گره فرزند (حرکت‌های ممکن) برای آن ایجاد می‌شود.
- گره‌های جدید به درخت اضافه می‌شوند.

3. شبیه‌سازی (Simulation):

- یکی از گره‌های جدید به‌طور تصادفی انتخاب می‌شود و بازی از آن حالت تا پایان (Win, Lose, Draw) به صورت تصادفی شبیه‌سازی می‌شود.
- نتیجه نهایی (برد، باخت یا مساوی) برای بازیکن جاری ثبت می‌شود.

4. بازگشت (Backpropagation):

- نتیجه شبیه‌سازی به‌طور بازگشتی از گره انتخاب‌شده به ریشه منتقل می‌شود.
- تعداد بازدیدها و مجموع امتیازات هر گره در مسیر به‌روزرسانی می‌شود.

نحوه کار الگوریتم در بازی دوز:

1. کلاس `Node` :

این کلاس نماینده یک گره در درخت MCTS است:

- `state`: وضعیت فعلی بازی به‌صورت یک شیء `TicTacToe`.
- `total_value`: مجموع امتیازهای شبیه‌سازی‌هایی که از این گره عبور کرده‌اند.
- `visits`: تعداد دفعات بازدید از این گره.
- `move`: حرکتی که منجر به این وضعیت شده است.
- `children`: لیستی از گره‌های فرزند.
- `parent`: گره والد.

2. تابع `rollout`

این تابع شبیه‌سازی یک بازی تصادفی را از گره فعلی انجام می‌دهد:

1. بررسی پایان بازی:
   - اگر بازی تمام شده باشد:
     - `1.0`: اگر بازیکن فعلی (player) برنده شود.
     - `-1.0`: اگر حریف برنده شود.
     - `0.0`: اگر بازی مساوی شود.
2. حرکت تصادفی:
   - اگر بازی تمام نشده باشد، یک حرکت تصادفی از حرکات موجود انتخاب می‌کند.
   - وضعیت جدید بازی را ایجاد کرده و شبیه‌سازی را ادامه می‌دهد.
3. بازگشت نتیجه:
   - نتیجه نهایی شبیه‌سازی به صورت بازگشتی محاسبه می‌شود.

3. تابع `backpropagate`

این تابع امتیازها و تعداد بازدیدها را در طول مسیر برگشتی از گره برگ به ریشه به‌روزرسانی می‌کند:

1. افزایش بازدیدها: تعداد بازدیدهای هر گره در مسیر افزایش می‌یابد.
2. به‌روزرسانی امتیازها: امتیاز حاصل از شبیه‌سازی به `total_value` گره‌ها اضافه می‌شود.

4. تابع `uct_score`

این تابع امتیاز UCT (Upper Confidence Bound for Trees) یک گره را محاسبه می‌کند:

2. اگر گره هنوز بازدید نشده باشد (`visits == 0`)، مقدار امتیاز infty بازمی‌گردد تا اولویت گسترش پیدا کند.

5. تابع `select_best_child`

این تابع بهترین گره فرزند را بر اساس بالاترین امتیاز UCT انتخاب می‌کند.

6. تابع `monte_carlo_tree_search`

این تابع الگوریتم MCTS را پیاده‌سازی می‌کند:

1. ایجاد گره ریشه:

- وضعیت فعلی بازی به‌عنوان ریشه درخت تعریف می‌شود.

2. حلقه شبیه‌سازی‌ها:

- تعداد مشخصی شبیه‌سازی (`num_simulations`) انجام می‌شود.

3. مراحل الگوریتم:

- Selection (انتخاب):

- گره‌ای که نیاز به گسترش دارد، با حرکت به پایین درخت و انتخاب بهترین فرزند (بر اساس امتیاز UCT) پیدا می‌شود.

- Expansion (گسترش):

- اگر گره انتخاب‌شده بازدید شده باشد و بازی تمام نشده باشد:

- تمام حرکات ممکن برای گره فعلی به‌عنوان فرزندان جدید اضافه می‌شوند.

- یکی از این گره‌ها به‌صورت تصادفی انتخاب می‌شود.

- Simulation (شبیه‌سازی):

- بازی از گره انتخاب‌شده به‌صورت تصادفی شبیه‌سازی می‌شود تا نتیجه نهایی مشخص شود.

- Backpropagation (بازگشت):

- نتیجه شبیه‌سازی به تمام گره‌های مسیر برگشتی تا ریشه منتقل می‌شود.

4. انتخاب بهترین حرکت:

- پس از اتمام شبیه‌سازی‌ها، گره فرزندی که بیشترین بازدید (`visits`) را داشته باشد، به‌عنوان بهترین حرکت انتخاب می‌شود.

- اگر گره‌ای وجود نداشته باشد (حالت خاص)، یک حرکت تصادفی انتخاب می‌شود.

7. بازگشت بهترین حرکت

- تابع در نهایت بهترین حرکت را به‌صورت یک زوج مختصات (row, col) بازمی‌گرداند.

مزایای الگوریتم MCTS:

1. اکتشاف و بهره‌برداری:

- با استفاده از UCB، الگوریتم به طور هوشمند بین بررسی گره‌های جدید و بهره‌برداری از گره‌های باارزش تعادل برقرار می‌کند.

2. عدم نیاز به تابع ارزیابی:

- برخلاف Minimax، نیازی به تابع ارزیابی پیچیده برای محاسبه ارزش وضعیت‌ها ندارد.

3. تطبیق‌پذیری:

- می‌توان از آن در بازی‌ها یا مسائل پیچیده که فضای حالت بسیار بزرگ است استفاده کرد.

4. بهبود با افزایش تعداد Rollout

- هرچه تعداد بیشتری بازی انجام شود، دقت تصمیم‌گیری آن افزایش می‌یابد.

معایب الگوریتم MCTS:

1. زمان‌بر بودن:

- برای بازی‌های بسیار پیچیده ممکن است شبیه‌سازی‌ها زمان زیادی ببرند.

2. کیفیت شبیه‌سازی تصادفی:

- اگر شبیه‌سازی‌ها به جای تصادفی بودن، استراتژیک‌تر انجام شوند، ممکن است نتیجه بهتری ارائه دهد.

3. حساسیت به ضریب

- انتخاب مقدار مناسب برای C در فرمول UCB می‌تواند بر عملکرد الگوریتم تأثیر بگذارد.

کاربردهای الگوریتم MCTS:

1. بازی‌های استراتژی:

- شطرنج، بازی گو (Go)، دوز (Tic-Tac-Toe)، تخته‌نرد و بازی‌های ویدیویی.

2.هوش مصنوعی:

- تصمیم‌گیری در شرایط پیچیده و غیرقطعی.

3.بهینه‌سازی و شبیه‌سازی:

- حل مسائل بهینه‌سازی در زمینه‌هایی مانند شبکه، لجستیک و مدیریت.

---

4) الگوریتم Evaluation Based :

```
from time import sleep


def evaluate_line(line: np.ndarray, current_player: int, win_length: int) -> int:
    """
    Evaluate a single line for potential:
    - More weight for lines closer to winning
    - Penalize opponent's potential wins
    """
    player_count = np.count_nonzero(line == current_player)
    opponent_count = np.count_nonzero(line == -current_player)
    empty_count = np.count_nonzero(line == 0)

    # Line with potential winning configuration
    if player_count > 0 and opponent_count == 0:
        potential_score = player_count * 10
        if player_count == win_length - 1:
            potential_score *= 2  # Very close to winning
        return potential_score

    # Line blocking opponent
    if opponent_count > 0 and player_count == 0:
        potential_score = -opponent_count * 10
        if opponent_count == win_length - 1:
            potential_score *= 2  # Block imminent threat
        return potential_score

    if player_count == 0 and opponent_count == 0:
        return 10
    if player_count != 0 and opponent_count != 0:
        return player_count * 10 - (opponent_count * 10)

    return 0  # Default return if no conditions met

def get_diagonals(board: np.ndarray, win_length: int = None) -> List[np.ndarray]:
    if win_length is None:
        win_length = board.shape[0]

    diagonals = []
    board_size = board.shape[0]

    # Main diagonals
    for k in range(-board_size + 1, board_size):
        diag = board.diagonal(k)
        if len(diag) >= win_length:
            diagonals.append(diag)

    # Anti-diagonals
    flipped = np.fliplr(board)
    for k in range(-board_size + 1, board_size):
```

```
        diag = flipped.diagonal(k)
        if len(diag) >= win_length:
            diagonals.append(diag)

    return diagonals

def evaluate_potential_lines(state: TicTacToe, current_player: int, win_length: int) ->
int:
    """Evaluate potential winning lines."""
    board = state.board
    board_size = board.shape[0]

    total_potential = 0

    # Rows and columns
    for i in range(board_size):
        total_potential += evaluate_line(board[i, :], current_player, win_length)
        total_potential += evaluate_line(board[:, i], current_player, win_length)

    # Diagonals
    for diag in get_diagonals(board, win_length):
        # print(diag)
        # state.display()
        # sleep(1)
        total_potential += evaluate_line(diag, current_player, win_length)

    return total_potential

def evaluate_state(state: TicTacToe, current_player: int, win_length: int = None) ->
float:
    """
    Generalized evaluation function for n×n Tic-Tac-Toe board.
    """
    board = state.board

    # If win_length not specified, set it to board size
    if win_length is None:
        win_length = board.shape[0]

    winner = state.check_winner()
    if winner:
        return 1000 * winner if winner == current_player else -1000 * winner

    score = 0
    board_size = board.shape[0]

    # Center control
    center_indices = [board_size // 2]
    if board_size % 2 == 0:
        center_indices = [board_size // 2 - 1, board_size // 2]

    for x in center_indices:
        for y in center_indices:
            if board[x, y] == current_player:
```

```
                score += 30
            elif board[x, y] == -current_player:
                score -= 30

    # Corner control
    corner_indices = [0, board_size - 1]
    for x in corner_indices:
        for y in corner_indices:
            if board[x, y] == current_player:
                score += 20
            elif board[x, y] == -current_player:
                score -= 20

    # Add potential line evaluation to score
    score += evaluate_potential_lines(state, current_player, win_length)

    return score

def minimax_with_evaluation(
        board: np.ndarray,
        depth: int,
        current_player: int,
        win_length: int = None
    ) -> float:
    # Create a game state from the board
    temp_game = TicTacToe()
    temp_game.board = board.copy()

    # Terminal state checks
    winner = temp_game.check_winner()
    if winner is not None:
        if winner == current_player:
            return 1000
        elif winner == -current_player:
            return -1000
        else:  # Draw
            return 0

    # Depth limit reached
    if depth == 0:
        return evaluate_state(temp_game, current_player, win_length)

    # Get available moves
    moves = temp_game.get_available_moves()

    if current_player == 1:
        best_score = float('-inf')
        for x, y in moves:
            new_board = board.copy()
            new_board[x, y] = current_player
            score = minimax_with_evaluation(
                new_board, depth - 1, -current_player, win_length
            )
            best_score = max(best_score, score)
```

```
        return best_score
    else:
        best_score = float('-inf')
        for x, y in moves:
            new_board = board.copy()
            new_board[x, y] = current_player
            score = minimax_with_evaluation(
                new_board, depth - 1, -current_player, win_length
            )
            best_score = max(best_score, score)
        return best_score

def evaluation_based(state: TicTacToe, player: int, max_depth: int = 1) -> Tuple[int,
int]:
    # Get available moves
    available_moves = state.get_available_moves()

    # Evaluate each possible move
    best_move = None
    best_score = float('-inf')

    for x, y in available_moves:
        # Try the move
        new_board = state.board.copy()
        new_board[x, y] = player

        # Evaluate the move using minimax
        move_score = minimax_with_evaluation(
            new_board, max_depth - 1, player
        )

        if move_score > best_score:
            best_score = move_score
            best_move = (x, y)

    # Fallback to random move if no good move found
    return 0,best_move
```

توضیحات این الگوریتم :

به طور کلی این الگوریتم تا عمق معینی از مینیمکس استفاده می کند ولی از ان به بعد ارزش استیت رابا استفاده از یک تابع ارزیابی Evaluate می کند.

توضیحات کد الگوریتمEvaluation-Based

.1توابع ارزیابی خطوط و حالت‌ها:

- **evaluate_line** :
  - این تابع یک خط از صفحه (یک سطر، ستون یا قطر) را بررسی می‌کند.
  - خطهایی که بازیکن فعلی در آن‌ها برتری دارد، امتیاز مثبت می‌گیرند.
  - خطهایی که حریف در آن‌ها برتری دارد، امتیاز منفی می‌گیرند.
  - اگر یک خط در آستانه برد باشد (مثلاً دو مهره در سه‌تایی برای برد در تیک‌تاک‌تو)، وزن بیشتری به آن اختصاص داده می‌شود.
- **evaluate_potential_lines** :
  - تمامی سطرها، ستون‌ها و قطرهای صفحه را با استفاده از تابع بالا ارزیابی می‌کند و امتیازات را جمع می‌کند.
- **evaluate_state** :
  - وضعیت فعلی صفحه بازی را ارزیابی می‌کند.
  - ابتدا بررسی می‌کند آیا بازی تمام شده است یا نه :
    - اگر بازیکن فعلی برنده است، امتیاز زیادی (مثلاً 1000) به وضعیت اختصاص داده می‌شود.
    - اگر بازی مساوی است یا بازیکن حریف برنده است، امتیاز مناسب داده می‌شود.
  - در غیر این صورت، امتیاز وضعیت با استفاده از توابع ارزیابی خطوط محاسبه می‌شود.

.2الگوریتم:Evaluation-Based

- **minimax_with_evaluation** :
  - نسخه‌ای از Minimax که از تابع ارزیابی برای تخمین امتیاز وضعیت‌ها استفاده می‌کند.
  - اگر عمق (depth) صفر باشد، به جای محاسبه دقیق تمام حالات، تابع evaluate_stateفراخوانی می‌شود.
  - در هر مرحله، این الگوریتم وضعیت بازی را برای بازیکن فعلی و حریف بررسی کرده و بهترین امتیاز را بازمی‌گرداند.
- **evaluation_based** :
  - این تابع حرکت بهینه را با استفاده از minimax_with_evaluationپیدا می‌کند.
  - ابتدا تمام حرکات ممکن را می‌یابد.
  - برای هر حرکت، حالت صفحه را پس از اعمال حرکت شبیه‌سازی می‌کند.
  - امتیاز هر حرکت را با کمک minimax_with_evaluationمحاسبه کرده و بهترین حرکت را انتخاب می‌کند.

ویژگی‌های اصلی:

- این روش از عمق (depth) محدود استفاده می‌کند تا تعداد حالات ممکن را کاهش دهد.
- به جای محاسبه دقیق تمام حالات بازی، یک تابع ارزیابی برای پیش‌بینی وضعیت بازی به کار می‌گیرد.
- تابع ارزیابی براساس پتانسیل برد یا تهدید بازیکنان، وضعیت را نمره‌دهی می‌کند.

نقاط قوت و ضعف:

- **نقاط قوت:**
  - سریع‌تر از Minimax کامل، به خصوص روی بردهای بزرگ‌تر.
  - امکان تنظیم عمق برای تعادل بین دقت و سرعت.
- **نقاط ضعف:**
  - ممکن است برخی از حالات برنده را به دلیل تخمین نادرست تابع ارزیابی از دست بدهد.
  - وابستگی به کیفیت تابع ارزیابی.

مقایسه الگوریتم های ارایه شده :

الگوریتم های ارایه شده را در قالب معیار های زیر بررسی می کنیم :

1) **حافظه مورد نیاز**

Minimax :

- حافظه مورد نیاز به تعداد گره‌های درخت جستجو بستگی دارد.
- برای بازی دوز(Tic Tac Toe) ، درخت جستجو در بدترین حالت O(bd)O(b^d)گره دارد :
  - bb:تعداد حرکات ممکن در هر مرحله (حداکثر 9 برای دوز).
  - dd:عمق درخت (9 برای دوز).
- حافظه نسبتاً زیادی مصرف می‌کند، چون تمام درخت جستجو را در حافظه نگه می‌دارد.

Alpha-Beta Pruning :

- مشابهMinimax ، اما با کاهش تعداد گره‌های بررسی‌شده.
- در حالت ایده‌آل، تعداد گره‌های بررسی‌شده به O(bd/2)O(b^{d/2})کاهش می‌یابد.
- نیاز به حافظه کمتری نسبت به Minimax دارد.

Monte Carlo Tree Search (MCTS) :

- حافظه مورد نیاز به تعداد شبیه‌سازی‌ها بستگی دارد.
- هر شبیه‌سازی نیاز به کپی‌کردن وضعیت بازی دارد.
- برای دوز، چون تعداد شبیه‌سازی‌ها محدود است (مثلاً 100)، حافظه کمتری نسبت به Minimax مصرف می‌کند.

Evaluation-Based Heuristic :

- فقط وضعیت فعلی بازی و مقادیر ارزیابی را ذخیره می‌کند.
- حافظه بسیار کمتری نسبت به سایر الگوریتم‌ها مصرف می‌کند.

---

2)زمان مورد نیاز برای انجام یک حرکت

Minimax :

- زمان اجرا(O(b^d، زیرا تمام درخت جستجو بررسی می‌شود.
- در دوز، این زمان قابل‌قبول است، اما در بازی‌های پیچیده‌تر زمان‌بر می‌شود.

Alpha-Beta Pruning :

- زمان اجرای بهینه‌تر نسبت بهMinimax ، به O(b^{d/2}) کاهش می‌یابد.
- در دوز، سرعت بیشتری نسبت به Minimax دارد.

: Monte Carlo Tree Search (MCTS)

- زمان اجرای وابسته به تعداد شبیه‌سازی‌ها است.
- اگر تعداد شبیه‌سازی‌ها محدود باشد (مثلاً 100)، زمان قابل‌قبولی دارد.
- در بازی‌های پیچیده‌تر، زمان بیشتری نسبت به Alpha-Beta نیاز دارد.

Evaluation-Based Heuristic:

- بسیار سریع است، زیرا فقط وضعیت فعلی را ارزیابی می‌کند.
- زمان اجرای O(1)برای ارزیابی هر وضعیت.

.3میزان دقت و درستی انتخاب

: Minimax

- در بازی‌های کامل (مانند دوز)، همیشه بهترین حرکت ممکن را انتخاب می‌کند.
- دقت 100% دارد.

: Alpha-Beta Pruning

- دقت مشابهMinimax ، چون همان گره‌ها را بررسی می‌کند (فقط گره‌های غیرضروری را حذف می‌کند).

: Monte Carlo Tree Search (MCTS)

- دقت وابسته به تعداد شبیه‌سازی‌ها است.
- برای تعداد شبیه‌سازی‌های کم، ممکن است بهینه‌ترین حرکت را پیدا نکند.
- در دوز، با تعداد کافی شبیه‌سازی، دقت بالایی دارد.

: Evaluation-Based Heuristic

- دقت پایین‌تر، چون از ارزیابی‌های تقریبی استفاده می‌کند.
- ممکن است حرکت بهینه را پیدا نکند.

---

4)مقیاس‌پذیری

: Minimax

- با افزایش پیچیدگی بازی، زمان و حافظه به‌سرعت افزایش می‌یابد.
- برای بازی‌های پیچیده‌تر (مانند شطرنج) غیرعملی است.

: Alpha-Beta Pruning

- مقیاس‌پذیرتر ازMinimax ، اما همچنان محدود به عمق درخت است.
- در بازی‌های پیچیده‌تر، نیاز به محدودیت عمق دارد.

: Monte Carlo Tree Search (MCTS)

- مقیاس‌پذیری بالایی دارد، چون می‌توان تعداد شبیه‌سازی‌ها را تنظیم کرد.
- برای بازی‌های پیچیده‌تر مناسب‌تر است.

: Evaluation-Based Heuristic

- بسیار مقیاس‌پذیر است، چون فقط وضعیت فعلی را ارزیابی می‌کند.
- مناسب برای بازی‌های با تعداد حالات زیاد.

---

5)پیچیدگی پیاده‌سازی

: Minimax

- ساده‌ترین الگوریتم برای پیاده‌سازی.

: Alpha-Beta Pruning

- کمی پیچیده‌تر ازMinimax ، اما همچنان ساده است.

: Monte Carlo Tree Search (MCTS)

- پیچیده‌ترین الگوریتم برای پیاده‌سازی، به‌خصوص در مدیریت درخت و شبیه‌سازی.

: Evaluation-Based Heuristic

- ساده‌ترین الگوریتم برای پیاده‌سازی.

---

6) کاربردها

: Minimax

- مناسب برای بازی‌های کوچک (مانند دوز) یا بازی‌هایی که می‌توان عمق درخت را محدود کرد.

: Alpha-Beta Pruning

- مناسب برای بازی‌های با فضای جستجوی بزرگ‌تر نسبت به.Minimax

: Monte Carlo Tree Search (MCTS)

- مناسب برای بازی‌های پیچیده‌تر که ارزیابی مستقیم گره‌ها دشوار است) مانند.(Go

: Evaluation-Based Heuristic

- مناسب برای بازی‌های سریع یا زمانی که محدودیت زمانی وجود دارد.

جدول مقایسه

معیار	Minimax	Alpha-Beta Pruning	Monte Carlo Tree Search	Evaluation-Based Heuristic
حافظه مورد نیاز	زیاد	متوسط	متوسط	کم
زمان برای یک حرکت	زیاد	متوسط	وابسته به شبیه‌سازی	بسیار کم
دقت	100%	100%	وابسته به شبیه‌سازی	پایین
مقیاس‌پذیری	کم	متوسط	بالا	بسیار بالا
پیچیدگی پیاده‌سازی	ساده	متوسط	پیچیده	بسیار ساده

نتیجه‌گیری

- **Minimax** : برای بازی‌های کوچک با عمق کم ایده‌آل است.
- **Alpha-Beta Pruning** : برای بازی‌های بزرگ‌تر که نیاز به بهینه‌سازی زمان دارند مناسب است.
- **MCTS** : بهترین انتخاب برای بازی‌های پیچیده‌تر با فضای جستجوی بزرگ است.
- **Evaluation-Based Heuristic** : مناسب برای بازی‌های سریع یا زمانی که منابع محاسباتی محدود هستند.

	A	B	C	D	E	F	G
1	Algorithm1	Algorithm2	Algorithm1_Wins	Algorithm2_Wins	Tie	Algorithm1_Avg	Algorithm2_Avg
2	minimax	alpha_beta	0	0	1	1.53	0.01
3	evaluation_based	minimax	0	1	0	0	0.25
4	minimax	monte_carlo_tree_search	30	0	0	2.25	0.02
5	alpha_beta	evaluation_based	1	0	0	0.07	0
6	monte_carlo_tree_search	alpha_beta	0	20	10	0.01	0.02
7	evaluation_based	monte_carlo_tree_search	197	3	0	0.00	0.02

- نتیجه کلی از تست دو به دوی الگوریتم های پیاده سازی شده

این جدول بطور داینامیک با استفاده از توابع گفته شده در قسمت ضمیمه گزارش بدست آمده است

کد های ضمیمه شده :

1) تابع test : دو تا الگوریتم را بعلاوه تعداد اجرای توابع به عنوان ورودی میگیرد و عملیات تست را انجام میدهد و خروجی را اعم از تعداد برد الگوریتم 1 و 2 و تعداد تساوی و میانگین زمان لازم برای انجام یک حرکت در الگوریتم 1 و 2 را در قالب یک دیکشنری باز میگرداند.

2) تابع testAllMethods : یک لیست از تست کیس ها می گیرد و هر تست کیس یک تاپل 3 تایی شامل ورودی های تابع تست می باشد و هر کدام از تست کیس ها را به عنوان ورودی به تابع تست میدهد و خروجی را در قالب یک دیکشنری که قابل تبدیل به فایل اکسل هست بازمیگرداند

3) تابع create_test_results_excel : این تابع خروجی تابع testAllMethods را گرفته و یک فایل اکسل براساس آن می سازد.