Minimax

```python
# Minimax method
def minimax(state: TicTacToe, player: int) -> Tuple[int, int]:
    """
    Implement the Minimax algorithm with improved move selection
    """
    def minimax_score(is_maximizing):
        # Check Terminal State
        result = state.check_winner()
        if result is not None:
            if result == player:
                return 1000
            elif result == 0:
                return 0
            else:
                return -1000

        if is_maximizing:    # Check for Maximum among Successors
            best_score = float('-inf')
            available_successors = state.get_available_moves()

            for row, col in available_successors:

                state.board[row, col] = player
                score = minimax_score(False)
                state.board[row, col] = 0

                best_score = max(score, best_score)

            return best_score

        else:    # Check for Min
            best_score = float('inf')
            opponent = -1 if player == 1 else 1
            available_successors = state.get_available_moves()
            for row, col in available_successors:

                state.board[row, col] = opponent
                score = minimax_score(True)
                state.board[row, col] = 0

                best_score = min(score, best_score)

            return best_score

    best_move = None
    best_score = float('-inf')
    available_moves = state.get_available_moves()
    for row, col in available_moves:

        state.board[row, col] = player
        score = minimax_score(False)
        state.board[row, col] = 0

        if score > best_score:
            best_score = score
            best_move = (row, col)

    return best_move

```

این تابع بهترین حرکت را برای بازیکن فعلی پیدا می‌کند.

- ورودی‌ها:
  - State: اینستنسی از کلاس TicTacToe
  - Player: بازیکن فعلی (۱ یا -۱).
- خروجی: مختصات حرکت بهینه در قالب یک **تاپل**.

تابع minimax_score

این تابع امتیاز یک وضعیت خاص از بازی را محاسبه می‌کند.

- ابتدا وضعیت پایانی بررسی می‌شود:
  - اگر بازیکن فعلی برنده باشد، امتیاز **1000** بازگردانده می‌شود.
  - اگر بازی مساوی باشد، امتیاز **0** بازگردانده می‌شود.
  - اگر حریف برنده باشد، امتیاز **-1000** بازگردانده می‌شود.
- دو حالت بررسی می‌شود:

۱. **ماکسیمم‌سازی**: بازیکن فعلی بهترین امتیاز ممکن را انتخاب می‌کند.

۲. **مینیمم‌سازی**: حریف سعی می‌کند امتیاز را برای بازیکن فعلی کمینه کند.

روند جستجوی حرکت بهینه

۱. تمامی حرکات ممکن از طریق تابع get_available_moves دریافت می‌شود.

۲. هر حرکت به طور موقت روی صفحه بازی اعمال می‌شود.

۳. امتیاز حرکت از طریق فراخوانی بازگشتی minimax_score محاسبه می‌شود.

۴. پس از محاسبه امتیاز، صفحه بازی به حالت اولیه بازگردانده می‌شود.

۵. بهترین امتیاز و حرکت به روز می‌شوند.

Alpha-Beta Pruning

```python
# Alpha-beta method
def alpha_beta(state: TicTacToe, player: int) -> Tuple[int, int]:
    """
    Implement the Alpha-Beta Pruning algorithm here.
    """
    def alpha_beta_score(alpha, beta, is_maximizing):
        # Check Terminal State
        result = state.check_winner()
        if result is not None:
            if result == player:
```

```
                return 1000
            elif result == 0:
                return 0
            else:
                return -1000

        if is_maximizing:    # Check for Maximum among Successors
            max_eval = float('-inf')

            available_successors = state.get_available_moves()
            for row, col in available_successors:

                state.board[row, col] = player
                eval = alpha_beta_score(alpha, beta, False)
                state.board[row, col] = 0

                max_eval = max(max_eval, eval)
                alpha = max(alpha, eval)
                if beta <= alpha:
                    break

            return max_eval

        else:    # Check for Min
            min_eval = float('inf')
            opponent = -1 if player == 1 else 1
            available_successors = state.get_available_moves()
            for row, col in available_successors:

                state.board[row, col] = opponent
                eval = alpha_beta_score(alpha, beta,True)
                state.board[row, col] = 0

                min_eval = min(eval, min_eval)
                beta = min(beta, eval)
                if beta <= alpha:
                    break

            return min_eval

    best_move = None
    best_score = float('-inf')
    alpha = float('-inf')
    beta = float('inf')
    available_moves = state.get_available_moves()
    for row, col in available_moves:

        state.board[row, col] = player
        score = alpha_beta_score(alpha, beta, False)
        state.board[row, col] = 0

        if score > best_score:
            best_score = score
            best_move = (row, col)

    return best_move

```

الگوریتم Alpha-Beta Pruning نسخه بهینه شده‌ای از Minimax است که شاخه‌های غیرضروری درخت جستجو را حذف می‌کند تا سرعت محاسبات افزایش یابد.

تابع اصلی alpha_beta

این تابع بهترین حرکت ممکن را برای بازیکن فعلی پیدا می‌کند.

- ورودی‌ها:

- State: اینستنسی از کلاس TicTacToe
- Player: بازیکن فعلی (۱ یا -۱).

- خروجی: مختصات بهترین حرکت به صورت تاپل.

تابع alpha_beta_score

این تابع امتیاز یک وضعیت خاص را با استفاده از هرس آلفا-بتا محاسبه می‌کند.

- آلفا: بهترین امتیاز ممکن برای بازیکن ما در طول مسیر جستجو.
- بتا: بهترین امتیاز ممکن برای حریف در طول مسیر جستجو.
- هرس آلفا-بتا: اگر مشخص شود که شاخه‌ای از درخت نمی‌تواند نتیجه بهتری نسبت به امتیازات فعلی (آلفا یا بتا) بدهد، جستجو برای آن شاخه متوقف می‌شود. (به کمک شرط موجود در تابع)

روند محاسبه امتیاز

۱. وضعیت فعلی بازی بررسی می‌شود:
   - اگر بازی پایان یافته باشد:
     - برد بازیکن فعلی: امتیاز ۱۰۰۰.
     - مساوی: امتیاز ۰.
     - برد حریف: امتیاز ۱۰۰۰-.
۲. در حالت ماکسیمم‌سازی:
   - بازیکن ما بهترین امتیاز ممکن را از بین حرکات موجود انتخاب می‌کند.
   - مقدار آلفا به‌روزرسانی می‌شود.
   - اگر آلفا بزرگتر یا برابر با بتا باشد، بقیه شاخه‌ها نادیده گرفته می‌شوند.
۳. در حالت مینیمم‌سازی:
   - حریف بهترین امتیاز ممکن را از بین حرکات انتخاب می‌کند.
   - مقدار بتا به‌روزرسانی می‌شود.
   - اگر آلفا بزرگتر یا برابر با بتا باشد، بقیه شاخه‌ها نادیده گرفته می‌شوند.

نحوه پیدا کردن بهترین حرکت

۱. تمامی حرکات ممکن از طریق تابع get_available_moves دریافت می‌شوند.
۲. برای هر حرکت:

- حرکت به صورت موقت روی صفحه اعمال می‌شود.
- امتیاز حرکت با استفاده از alpha_beta_score محاسبه می‌شود.
- صفحه بازی به حالت اولیه بازگردانده می‌شود.
- اگر امتیاز حرکت از بهترین امتیاز فعلی بیشتر باشد، حرکت و امتیاز به‌روزرسانی می‌شوند.

مزایای الگوریتم Alpha-Beta

- کاهش شاخه‌های جستجو: این الگوریتم شاخه‌هایی که تأثیری بر نتیجه نهایی ندارند را نادیده می‌گیرد. (هرس می‌کند)
- سرعت بیشتر نسبت به Minimax: با حذف شاخه‌های غیرضروری، تعداد محاسبات کاهش می‌یابد.
- دقت مشابه Minimax: نتیجه نهایی با Minimax یکسان است اما از نظر سرعت و حافظه بهینه‌تر است.

Evaluation Function

```python
# evaluatiob based method
def evaluation_based(state: TicTacToe, player: int) -> Tuple[int, int]:
    """
    Implement a heuristic evaluation-based decision-making algorithm here.
    """
    def evaluate():
        lines = []
        line = []
        for i in range(3):
            line = [state.board[i, j] for j in range(3)]
            lines.append(line)
        for j in range(3):
            line = [state.board[i, j] for i in range(3)]
            lines.append(line)

        line = [state.board[i, i] for i in range(3)]
        lines.append(line)
        line = [state.board[i, 2 - i] for i in range(3)]
        lines.append(line)

        score = 0
        for line in lines:
            if line.count(player) == 3:
                return 1000000
            if line.count(player) == 2 and line.count(0) == 1:
                score += 200
            if line.count(player) == 2 and line.count(0) == 0:
                score -= 100
            if line.count(player) == 1 and line.count(-1 * player) < 2:
                score += 100
            if line.count(-1 * player) == 2 and line.count(0) == 1:
                score -= 1000000

        return score


    best_move = None
    best_score = float('-inf')
    available_moves = state.get_available_moves()
    for row, col in available_moves:
        state.board[row, col] = player
        score = evaluate()
```

```
43.         state.board[row, col] = 0
44.
45.         if score > best_score:
46.             best_score = score
47.             best_move = row, col
48.
49.     return best_move
50.
```

تابع اصلی evaluation_based

این تابع، بهترین حرکت را برای بازیکن فعلی بر اساس امتیازات ارزیابی خطوط در صفحه بازی پیدا می‌کند.

- ورودی‌ها:
  - State: اینستنسی از کلاس TicTacToe
  - Player: بازیکن فعلی (۱ یا -۱).
- خروجی: مختصات بهترین حرکت به صورت تاپل.

تابع evaluate

این تابع برای محاسبه امتیاز وضعیت فعلی بازی استفاده می‌شود.

- ابتدا خطوط افقی، عمودی و مورب صفحه بازی استخراج می‌شوند.
- برای هر خط، امتیاز زیر محاسبه می‌شود:

۱. برد بازیکن فعلی: اگر بازیکن ۳ مهره در یک خط داشته باشد، امتیاز بسیار بالایی (مثلا ۱,۰۰۰,۰۰۰) برمی‌گرداند.

۲. دو مهره بازیکن و یک خانه خالی: امتیاز مثبتی (مثلا +۲۰۰) اضافه می‌شود.

۳. دو مهره بازیکن و خط بسته (بدون خانه خالی): امتیاز منفی (مثلا -۱۰۰) در نظر گرفته می‌شود.

۴. یک مهره بازیکن و محدودیت کمتر از دو مهره حریف: امتیاز مثبتی (مثلا +۱۰۰) اضافه می‌شود.

۵. دو مهره حریف و یک خانه خالی: امتیاز بسیار منفی (مثلا -۱,۰۰۰,۰۰۰) در نظر گرفته می‌شود.

روند پیدا کردن بهترین حرکت

۱. حرکات ممکن: تمامی حرکات ممکن با استفاده از تابع get_available_moves شناسایی می‌شوند.

۲. محاسبه امتیاز هر حرکت:
   - حرکت موقتاً روی صفحه اعمال می‌شود.
   - امتیاز وضعیت جدید با استفاده از تابع evaluate محاسبه می‌شود.
   - صفحه به حالت اولیه بازگردانده می‌شود.

۳. به‌روزرسانی بهترین حرکت:

- اگر امتیاز حرکت فعلی از بهترین امتیاز فعلی بیشتر باشد، بهترین حرکت و امتیاز به‌روزرسانی می‌شوند.

ویژگی‌های الگوریتم

۱. سرعت بالا: چون به جای جستجوی درخت کامل از ارزیابی مراحل استفاده می‌کند و مرتبهٔ محاسباتی پایین‌تری دارد.

۲. سادگی پیاده‌سازی: قوانین ارزیابی مشخص هستند و محاسبات سریع انجام می‌شود.

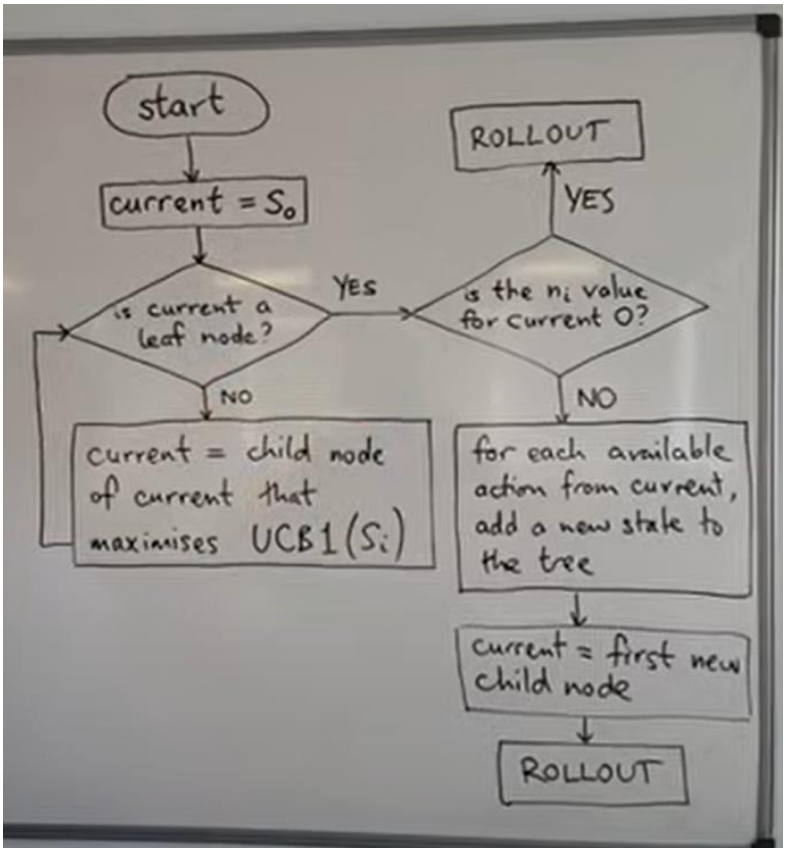
۳. نتایج قابل قبول: به خصوص در بازی‌های کوچک مانند تیک‌تاک‌تو، عملکرد خوبی دارد.

۴. محدودیت‌ها: این الگوریتم نمی‌تواند حرکات چند مرحله بعد را پیش‌بینی کند و تنها به وضعیت فعلی متکی است.

این الگوریتم گزینه‌ای سریع و ساده برای شبیه‌سازی بازی‌های استراتژیک با ابعاد کوچک است و برای بازی‌های پیچیده‌تر ممکن است نیاز به بهبود داشته باشد.

MCTS

الگوریتم پیاده سازی شده در این سوال:




```
# Monte Carlo Tree search method
def monte_carlo_tree_search(state: TicTacToe, player: int) -> Tuple[int, int]:
    """
    Implement the Monte Carlo Tree Search algorithm here.
    """
    class Node:
        def __init__(self, state: TicTacToe, parent=None, move=None):
            self.state = state
```

```
            self.t = 0
            self.n = 0
            self.move = move
            self.children = []
            self.parent = parent

    def rollout(game_node: Node, turn) -> int:
        match_res = game_node.state.check_winner()
        if match_res == player:
            return 1000
        elif match_res == (-1) * player:
            return -1000
        elif match_res == 0:
            return 0

        avail_moves = game_node.state.get_available_moves()
        if not avail_moves:
            return 0

        rnd_x, rnd_y = random.choice(avail_moves)
        game_copy = deepcopy(game_node)
        game_copy.state.board[rnd_x, rnd_y] = turn
        return rollout(game_copy, (-1) * turn)

    def backpropagate(value, node: Node):
        while node is not None:
            node.n += 1
            node.t += value
            node = node.parent

    def uct_score(node: Node, parent_n: int):
        if node.n == 0:
            return float('inf')
        return (node.t / node.n) + 2 * sqrt(log(parent_n) / node.n)

    root = Node(state)

    for _ in range(100):
        current = root
        while current.children and all(child.n > 0 for child in current.children):
            current = max(current.children,
                          key=lambda x: uct_score(x, current.n))

        if current.n > 0:
            avail_moves = current.state.get_available_moves()
            if avail_moves and not current.children:
                for row, col in avail_moves:
                    tmp = deepcopy(current.state)
                    tmp.board[row, col] = player
                    current.children.append(Node(tmp, parent=current, move=(row, col)))
                current = random.choice(current.children)

        result = rollout(current, player)

        backpropagate(result, current)

    if not root.children:
        return random.choice(state.get_available_moves())

    best_node:Node = max(root.children, key=lambda x: x.n)
    return best_node.move

```

این الگوریتم به‌جای جستجوی قطعی، از شبیه‌سازی‌های تصادفی برای ارزیابی حرکات استفاده می‌کند. هدف این است که حرکت بهینه را از طریق محاسبه میانگین نتایج شبیه‌سازی‌ها پیدا کند.

مراحل اصلی الگوریتمMCTS

۱. تعریف گره‌ها(Nodes) :

- هر گره نشان‌دهنده یک وضعیت بازی است.
- هر گره شامل موارد زیر است:
  - وضعیت فعلی بازی(state) .
  - تعداد بازی‌های شبیه‌سازی شده از این گره(n) .
  - مجموع امتیازات (t) تمام شبیه‌سازی‌ها.
  - حرکت منتهی به این گره(move) .
  - لیست فرزندان (children) که گره‌های بعدی را ذخیره می‌کند.
  - ارجاع به گره والد(parent) .

۲. شبیه‌سازی(Rollout) :

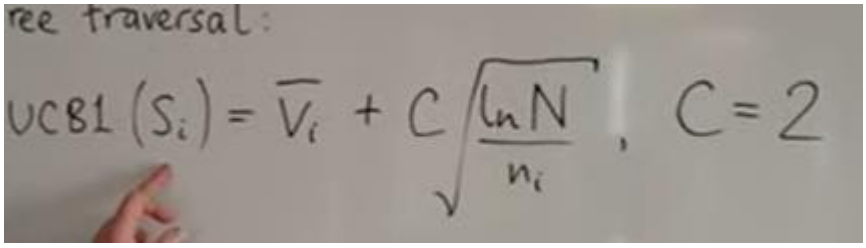
- از وضعیت فعلی بازی (گره)، بازی به‌صورت تصادفی تا انتها ادامه می‌یابد.
- نتایج شبیه‌سازی:
  - برد بازیکن(1000+) .
  - برد حریف(1000-) .
  - مساوی(0) .
- شبیه‌سازی‌های تصادفی به یافتن حرکاتی که به برد منجر می‌شوند، کمک می‌کنند.

۳. Backpropagation:

- نتیجه شبیه‌سازی به گره والد و گره‌های قبلی درخت بازگشت داده می‌شود.
- تعداد شبیه‌سازی‌ها (n) و مجموع امتیازات (t) به‌روزرسانی می‌شوند.

۴. انتخاب گره با معیار UCB1:

- معیار UCB1 برای انتخاب گره‌ها استفاده می‌شود:



- این معیار تعادل بین promising بودن و مورد آزمایش قرار گرفتن را برقرار می‌کند.

۵. ایجاد فرزند جدید:

- اگر گره جاری شبیه‌سازی شده باشد ولی تمام حرکات ممکن بررسی نشده باشد، گره‌های فرزند جدید ایجاد می‌شوند.

۶. انتخاب حرکت بهینه:

- در نهایت، گره فرزندی که بیشترین تعداد شبیه‌سازی را داشته باشد، به‌عنوان حرکت بهینه انتخاب می‌شود.

کد الگوریتم

در کد ارائه شده:

- تابع rollout: بازی را از گره فعلی به‌صورت تصادفی ادامه می‌دهد و نتیجه را بازمی‌گرداند.
- تابع backpropagate: نتایج شبیه‌سازی را به گره‌های قبلی اطلاع می‌دهد.
- حلقه اصلی: شامل ۱۰۰ آزمایش برای بهینه‌سازی حرکت است.

مزایا

۱. انعطاف‌پذیری بالا: برای بازی‌های مختلف و بدون نیاز به تغییرات زیاد قابل استفاده است.

۲. تعادل بین کاوش و بهره‌برداری: حرکات ناشناخته بررسی می‌شوند ولی حرکات امیدوارکننده نیز بیشتر مورد توجه قرار می‌گیرند.

۳. عملکرد خوب در شرایط پیچیده: در بازی‌های بزرگ و درخت‌های تصمیم‌گیری وسیع بسیار کارآمد است.

معایب

۱. زمان‌بر بودن: تعداد شبیه‌سازی‌ها ممکن است به منابع زیادی نیاز داشته باشد.

۲. تصادفی بودن: نتایج ممکن است در برخی موارد به دلیل تصادفی بودن شبیه‌سازی‌ها نوسان داشته باشد.

این الگوریتم برای بازی‌های پیچیده مانند شطرنج، تیک‌تاک‌تو و سایر بازی‌های استراتژیک قابل استفاده است و به عنوان یکی از روش‌های پیشرفته در هوش مصنوعی شناخته می‌شود.

الگوریتم ۱	الگوریتم ۲	برد الگوریتم ۱	برد الگوریتم ۲	تساوی	میانگین ۱	میانگین ۲
Minimax	Alpha-Beta	0	0	1	1.1752	0.0116
Evaluation	Minimax	0	0	1	0.000	0.1506
Minimax	MCTS	28	0	2	1.5446	0.0495
Alpha-Beta	Evaluation	0	0	1	0.0798	0.000
MCTS	Alpha-Beta	0	24	6	0.0453	0.0182
Evaluation	MCTS	86	4	10	0.000	0.0510