راهنمای کامل هوش مصنوعی بازی اُتللو

Complete Guide to Othello AI with Minimax Algorithm

محمدمهدی شریف بیگی
MohammadMahdi Sharifbeigy

Advanced AI Implementation with Enhanced Minimax

لینک ارائه

۱۲ شهریور ۱۴۰۴

فهرست مطالب

۱ مقدمه

این مستند راهنمای جامعی برای درک و پیاده‌سازی هوش مصنوعی در بازی اُتللو ارائه می‌دهد. این سیستم با استفاده از الگوریتم مینی‌مکس بهبود یافته با تکنیک‌های Pruning Alpha-Beta و ارزیابی چندبُعدی قادر به رقابت در سطوح مختلف هوش مصنوعی است.

۱.۱ ویژگی‌های کلیدی

- پیاده‌سازی الگوریتم مینی‌مکس با Pruning Alpha-Beta
- تابع ارزیابی چندمؤلفه برای تحلیل استراتژیک
- سیستم وزن‌دهی موقعیتی پیشرفته
- تشخیص فاز بازی و تطبیق استراتژی
- پنج سطح هوش مصنوعی از مبتدی تا استادکل
- واسط کاربری پیشرفته با انیمیشن‌ها و تحلیل‌های بصری

۲ الگوریتم مینی‌مکس و تصمیم‌گیری هوش مصنوعی

۱.۲ اصول بنیادی الگوریتم مینی‌مکس

الگوریتم مینی‌مکس یکی از پایه‌ای‌ترین و قدرتمندترین الگوریتم‌های تصمیم‌گیری در بازی‌های دو نفره با اطلاعات کامل محسوب می‌شود. این الگوریتم بر اساس فرض اینکه هر دو بازیکن به صورت بهینه بازی می‌کنند، بهترین حرکت ممکن را پیدا می‌کند.

۱.۱.۲ اصل عملکرد

الگوریتم مینی‌مکس بر پایه درخت جستجوی بازی عمل می‌کند که در آن:

- گره‌های Maximizer: نشان‌دهنده نوبت بازیکن هوش مصنوعی که سعی در بیشینه‌سازی امتیاز دارد
- گره‌های Minimizer: نشان‌دهنده نوبت حریف که سعی در کمینه‌سازی امتیاز هوش مصنوعی دارد
- گره‌های برگ: حالت‌های پایانی بازی که با تابع ارزیابی امتیازدهی می‌شوند

۲.۱.۲ فرمولاسیون ریاضی

اگر $V(n)$ ارزش گره n در درخت بازی باشد، آنگاه:

$$V(n) = \begin{cases} \max_{s \in \text{successors}(n)} V(s) & \text{اگر } n \text{ باشد گره Maximizer} \\ \min_{s \in \text{successors}(n)} V(s) & \text{اگر } n \text{ باشد گره Minimizer} \\ \text{Evaluate}(n) & \text{اگر } n \text{ باشد گره برگ} \end{cases}$$

۲.۲ بهینه‌سازی Pruning Alpha-Beta

یکی از مهم‌ترین بهینه‌سازی‌های الگوریتم مینی‌مکس، تکنیک Pruning Alpha-Beta است که به طور قابل توجهی تعداد گره‌های بررسی شده را کاهش می‌دهد.

۱.۲.۲ اصول Alpha-Beta

- Alpha (α): بهترین امتیاز تضمین شده برای بازیکن Maximizer
- Beta (β): بهترین امتیاز تضمین شده برای بازیکن Minimizer
- شرط Pruning: اگر $\alpha \geq \beta$ باشد، شاخه‌های باقی‌مانده نادیده گرفته می‌شوند

۳ سیستم وزن‌دهی مکانی و استراتژی

۱.۳ ماتریس ارزش‌های مکانی

یکی از کلیدی‌ترین عوامل در قدرت تصمیم‌گیری هوش مصنوعی، ماتریس ارزش‌های مکانی است که در کد به صورت زیر تعریف شده:

```
    POSITION_VALUES = [
    [100, -20,  10,   5,   5,  10, -20, 100],
    [-20, -50,  -2,  -2,  -2,  -2, -50, -20],
    [ 10,  -2,  -1,  -1,  -1,  -1,  -2,  10],
    [  5,  -2,  -1,  -1,  -1,  -1,  -2,   5],
    [  5,  -2,  -1,  -1,  -1,  -1,  -2,   5],
    [ 10,  -2,  -1,  -1,  -1,  -1,  -2,  10],
    [-20, -50,  -2,  -2,  -2,  -2, -50, -20],
    [100, -20,  10,   5,   5,  10, -20, 100]
    ]

```

Listing 1: Position Values Matrix Definition

۲.۳ تحلیل عمیق ماتریس مکانی

این ماتریس بر اساس اصول استراتژیک بازی اُتللو طراحی شده و هر عدد دلیل مشخصی دارد:

۱.۲.۳ گوشه‌ها (ارزش +۱۰۰)

گوشه‌ها با ارزش +۱۰۰ مهم‌ترین موقعیت‌های تخته محسوب می‌شوند زیرا:

- پایداری مطلق: مهره‌های قرار گرفته در گوشه هیچ‌گاه قابل تغییر نیستند
- کنترل استراتژیک: هر گوشه کنترل شده به بازیکن امکان تسلط بر کل ضلع مجاور را می‌دهد

- مزیت بلندمدت: در مراحل پایانی بازی، کنترل گوشه‌ها معمولاً تعیین‌کننده پیروزی است
- محدودسازی حریف: گرفتن گوشه، گزینه‌های استراتژیک حریف را به شدت محدود می‌کند

۲.۲.۳ مربع‌های مجاور گوشه (ارزش -۲۰)

این موقعیت‌ها با ارزش منفی -۲۰ به دلایل زیر خطرناک محسوب می‌شوند:

- فرصت‌طلبی حریف: قرار دادن مهره در این موقعیت‌ها اغلب به حریف امکان گرفتن گوشه را می‌دهد
- ضعف تاکتیکی: این مربع‌ها معمولاً در ابتدای بازی بی‌ثبات هستند
- ریسک بالا: مزیت کوتاه‌مدت کسب شده اغلب در مقایسه با ضرر بلندمدت ناچیز است

۳.۲.۳ مربع‌های X (ارزش -۵۰)

این موقعیت‌ها با ارزش -۵۰ خطرناک‌ترین موقعیت‌های تخته محسوب می‌شوند:

- آسیب‌پذیری شدید: قرار دادن مهره در این موقعیت‌ها تقریباً همیشه منجر به از دست دادن گوشه می‌شود
- اشتباه استراتژیک: تجربه نشان می‌دهد که بازیکنان تازه‌کار اغلب از این موقعیت‌ها استفاده می‌کنند
- مزیت فوری حریف: حریف می‌تواند بلافاصله از این اشتباه استفاده کرده و کنترل گوشه را به دست آورد

۴.۲.۳ لبه‌ها (ارزش +۱۰/+۵)

موقعیت‌های لبه‌ای ارزش مثبت +۱۰ یا +۵ دارند زیرا:

- پایداری نسبی: مهره‌های لبه‌ای معمولاً در برابر تغییر مقاوم‌تر هستند
- کنترل منطقه‌ای: کنترل لبه‌ها امکان تسلط بر مناطق بزرگ‌تری از تخته را فراهم می‌کند
- آماده‌سازی برای گوشه: موقعیت‌های لبه‌ای اغلب مقدمه‌ای برای کنترل گوشه‌ها هستند

۵.۲.۳ مناطق داخلی (ارزش -۱/-۲)

موقعیت‌های مرکزی ارزش منفی کوچک دارند زیرا:

- بی‌ثباتی: این موقعیت‌ها به راحتی قابل تغییر هستند
- انعطاف‌پذیری بالا: در مراحل ابتدایی بازی، نگه‌داشتن این موقعیت‌ها خالی گزینه‌های بیشتری فراهم می‌کند
- استراتژی تأخیری: تأخیر در اشغال مناطق مرکزی معمولاً مزیت استراتژیک محسوب می‌شود

۳.۳ تأثیر ماتریس بر تصمیم‌گیری هوش مصنوعی

۱.۳.۳ مرحله ارزیابی اولیه

هوش مصنوعی در ابتدا تمام حرکات ممکن را شناسایی می‌کند و سپس با استفاده از ماتریس مکانی، امتیاز اولیه هر حرکت را محاسبه می‌کند:

```
    # Move ordering for better alpha-beta pruning
    move_scores = []
    for move in valid_moves:
    quick_score = 0
    r, c = move

    # Corner priority - absolute strategic advantage
    if (r, c) in [(0, 0), (0, 7), (7, 0), (7, 7)]:
    quick_score += 1000  # Override position value with extreme priority

    # High impact moves - number of pieces flipped
    quick_score += len(valid_moves[move]) * 10

    # Strategic positional value from matrix
    quick_score += POSITION_VALUES[r][c]

    move_scores.append((quick_score, move))

    # Sort moves by strategic value (best first for maximizing player)
    move_scores.sort(key=lambda x: x[0], reverse=maximizing_player)
    ordered_moves = [move for _, move in move_scores]

```

Listing 2: Initial Move Scoring with Position Values

۲.۳.۳ تأثیر بر مرتب‌سازی حرکات

ماتریس مکانی مستقیماً بر ترتیب بررسی حرکات تأثیر می‌گذارد که این امر برای کارایی Pruning Alpha-Beta حیاتی است:

۱. اولویت گوشه‌ها: حرکات گوشه‌ای همیشه در اولویت بالا قرار می‌گیرند

۲. اجتناب از موقعیت‌های خطرناک: حرکات با ارزش منفی بالا در اولویت پایین قرار می‌گیرند

۳. بهینه‌سازی جستجو: ترتیب بهینه حرکات منجر به حذف شاخه‌های بیشتری در Alpha-Beta می‌شود

۴ تابع ارزیابی

تابع ارزیابی این سیستم از هفت مؤلفه اصلی تشکیل شده که هر کدام نقش مهمی در تصمیم‌گیری دارند:

۱.۴ شمارش مهره‌ها با در نظرگیری Parity

```
    # Basic piece count difference
    my_pieces = sum(row.count(player) for row in board)
    opp_pieces = sum(row.count(opponent) for row in board)
    piece_diff = my_pieces - opp_pieces

    # Parity bonus in endgame - who gets the last moves
    if total_pieces > 55:
    remaining_moves = 64 - total_pieces
    if remaining_moves % 2 == 1:  # Odd number of moves left
    piece_diff += 0.5  # Slight advantage to current player

    score += phase_weights['piece'] * piece_diff

```

Listing 3: Piece Count with Parity Consideration

این مؤلفه اهمیت ویژه‌ای در مراحل پایانی بازی دارد زیرا تعداد مهره‌ها معیار نهایی پیروزی است.

۲.۴ (Mobility Analysis)

```
    # Calculate available moves for both players
    temp_game = Othello(sounds={})
    temp_game.board = board
    my_moves = len(temp_game.get_valid_moves(player))
    opp_moves = len(temp_game.get_valid_moves(opponent))

    if my_moves + opp_moves > 0:
    mobility_ratio = (my_moves - opp_moves) / (my_moves + opp_moves + 1)
    score += phase_weights['mobility'] * mobility_ratio * 100

    # Critical mobility situations
    if my_moves == 0 and opp_moves > 0:
    score -= 500  # Very dangerous position - no moves available
    elif opp_moves == 0 and my_moves > 0:
    score += 500  # Excellent position - opponent cannot move

```

Listing 4: Enhanced Mobility Calculation

تعداد حرکت های باقیمانده یکی از مهم‌ترین فاکتورهای استراتژیک در اُتللو محسوب می‌شود زیرا داشتن گزینه‌های بیشتر معادل کنترل بیشتر بر بازی است.

۳.۴ کنترل گوشه‌ها با جریمه مجاورت

```
    corners = [(0, 0), (0, 7), (7, 0), (7, 7)]
    corner_adjacencies = [
    [(0,1), (1,0), (1,1)],  # Adjacent to (0,0)
    [(0,6), (1,7), (1,6)],  # Adjacent to (0,7)
    [(6,0), (7,1), (6,1)],  # Adjacent to (7,0)
    [(6,7), (7,6), (6,6)]   # Adjacent to (7,7)
    ]

    my_corners = opp_corners = 0
    for i, (r, c) in enumerate(corners):
    if board[r][c] == player:
    my_corners += 1
    elif board[r][c] == opponent:
    opp_corners += 1
    elif board[r][c] == EMPTY:
    # Penalty for occupying squares adjacent to empty corners
    for adj_r, adj_c in corner_adjacencies[i]:
    if board[adj_r][adj_c] == player:
    score -= 25  # Dangerous position near empty corner
    elif board[adj_r][adj_c] == opponent:
    score += 25  # Opponent is in dangerous position

    score += phase_weights['corner'] * (my_corners - opp_corners)

```

Listing 5: Corner Control with Adjacency Penalties

۴.۴ تحلیل پایداری پیشرفته

```
    def count_advanced_stable_pieces(board, player):
    stable_count = 0

    for r in range(8):
    for c in range(8):
    if board[r][c] == player:
    stability_score = 0

    # Corner pieces are always completely stable
    if (r, c) in [(0, 0), (0, 7), (7, 0), (7, 7)]:
    stable_count += 1
    continue

    # Edge stability analysis
    if r == 0 or r == 7 or c == 0 or c == 7:
    edge_stable = True
    # Check if entire edge is controlled
    if r == 0 or r == 7:  # Top/bottom edge
    for dc in [-1, 1]:
    nc = c + dc
```

```
        while 0 <= nc < 8:
        if board[r][nc] != player:
        edge_stable = False
        break
        nc += dc
        if edge_stable:
        stability_score += 1

        # Internal stability (completely surrounded)
        surrounded = True
        for dr in [-1, 0, 1]:
        for dc in [-1, 0, 1]:
        if dr == 0 and dc == 0:
        continue
        nr, nc = r + dr, c + dc
        if 0 <= nr < 8 and 0 <= nc < 8:
        if board[nr][nc] != player:
        surrounded = False
        break
        if not surrounded:
        break

        if surrounded:
        stability_score += 0.5

        stable_count += stability_score

        return stable_count

```

Listing 6: Advanced Stability Analysis

۵ تطبیق فازی و استراتژی پویا

۱.۵ تشخیص فاز بازی

سیستم هوش مصنوعی بازی را به سه فاز تقسیم می‌کند و برای هر فاز وزن‌های متفاوتی اعمال می‌کند:

```
        def advanced_evaluate_board(board, player, total_pieces, depth_remaining=0):
        opponent = -player

        # Dynamic phase detection based on total pieces on board
        if total_pieces < 20:  # Opening phase
        phase_weights = {
          'piece': 1,         # Piece count less important early
          'mobility': 20,     # Mobility is crucial in opening
          'corner': 150,      # Corner control extremely important
          'edge': 10,         # Edge control moderately important
          'stability': 100,   # Piece stability matters
          'position': 15,     # Positional values important
```

```
        'parity': 5         # Parity less relevant early
    }
    elif total_pieces < 52:  # Mid-game phase
    phase_weights = {
        'piece': 8,         # Piece count starts mattering more
        'mobility': 15,     # Mobility remains important
        'corner': 120,      # Corners still crucial
        'edge': 15,         # Edge control more important
        'stability': 120,   # Stability becomes critical
        'position': 12,     # Position values matter
        'parity': 10        # Parity consideration increases
    }
    else:  # End-game phase
    phase_weights = {
        'piece': 25,        # Piece count is final victory condition
        'mobility': 8,      # Mobility less important
        'corner': 140,      # Corners decide endgame
        'edge': 20,         # Edge control very important
        'stability': 140,   # Stability determines final outcome
        'position': 5,      # Position values less relevant
        'parity': 30        # Parity becomes crucial
    }

    score = 0

    # Apply all evaluation components with phase-specific weights
    score += phase_weights['piece'] * piece_diff
    score += phase_weights['mobility'] * mobility_ratio * 100
    score += phase_weights['corner'] * (my_corners - opp_corners)
    # ... other components

    return score

```

Listing 7: Dynamic Phase Detection and Weighting

۲.۵ توضیح فازهای مختلف بازی

۱.۲.۵ فاز شروع بازی (**Opening Phase** - تا ۲۰ مهره)

در این فاز، هوش مصنوعی تمرکز اصلی خود را بر روی موارد زیر می‌گذارد:

- حداکثرسازی تحرک: با وزن ۲۰، بیشترین تأکید بر حفظ گزینه‌های بیشتر است
- اجتناب از مخاطرات گوشه: با وزن ۱۵۰ برای گوشه‌ها، سیستم به شدت از دادن فرصت گوشه به حریف اجتناب می‌کند
- کم‌اهمیتی شمارش مهره: با وزن تنها ۱، تعداد مهره‌ها در این مرحله اولویت پایینی دارد

۲.۲.۵ فاز میانه بازی (**Mid-game Phase** - ۲۰ تا ۵۲ مهره)

در این فاز حساس، توازن بین عوامل مختلف برقرار می‌شود:

- تعادل استراتژی: همه عوامل وزن معقولی دارند
- اهمیت پایداری: با وزن ۱۲۰، ثبات موقعیت‌ها کلیدی می‌شود
- کنترل لبه‌ها: با افزایش وزن به ۱۵، کنترل لبه‌ها اهمیت می‌یابد

۳.۲.۵ فاز پایان بازی (**End-game Phase**- بیش از ۵۲ مهره)

در مراحل نهایی، اولویت‌ها به طور کامل تغییر می‌کند:

- اهمیت حیاتی شمارش مهره: با وزن ۲۵، تعداد نهایی مهره‌ها تعیین‌کننده است
- کنترل مطلق گوشه‌ها: با وزن ۱۴۰، گوشه‌ها کلید پیروزی هستند
- اهمیت Parity: با وزن ۳۰، کنترل آخرین حرکات بسیار مهم است

۶ مدیریت عمق جستجو و سطوح هوش مصنوعی

۱.۶ سیستم سطح‌بندی هوش مصنوعی

```
    class Difficulty(Enum):
    EASY = 2        # Depth 2 - basic planning
    MEDIUM = 4      # Depth 4 - intermediate strategy
    HARD = 6        # Depth 6 - advanced planning
    EXPERT = 8      # Depth 8 - expert-level analysis
    GRANDMASTER = 10 # Depth 10 - master-level deep analysis

    # Adaptive time limits for each difficulty
    time_limits = {
      Difficulty.EASY: 1.0,      # 1 second thinking time
      Difficulty.MEDIUM: 3.0,    # 3 seconds thinking time
      Difficulty.HARD: 8.0,      # 8 seconds thinking time
      Difficulty.EXPERT: 15.0,   # 15 seconds thinking time
      Difficulty.GRANDMASTER: 30.0  # 30 seconds thinking time
    }

```

Listing 8: AI Difficulty System Definition

۲.۶ تأثیر عمق جستجو بر قدرت تصمیم‌گیری

عمق جستجو تأثیر نمایی بر قدرت هوش مصنوعی دارد:

۱.۲.۶ محاسبه تعداد حالات بررسی شده

اگر به طور متوسط در هر حالت b حرکت معتبر وجود داشته باشد، تعداد کل حالات بررسی شده در عمق d برابر است با:
حالات بررسی شده $\approx b^d$
برای اُتللو معمولاً $b \approx ۸$ است، بنابراین:

- عمق ۲: $۸^۲ = ۶۴$ حالت
- عمق ۴: $۸^۴ = ۴,۰۹۶$ حالت
- عمق ۶: $۸^۶ = ۲۶۲,۱۴۴$ حالت
- عمق ۸: $۸^۸ = ۱۶,۷۷۷,۲۱۶$ حالت
- عمق ۱۰: $۸^{۱۰} = ۱,۰۷۳,۷۴۱,۸۲۴$ حالت

۳.۶ بهینه‌سازی‌های کارایی

۱.۳.۶ مرتب‌سازی حرکات برای Alpha-Beta

```
        # Strategic move ordering based on position values and game knowledge
        move_scores = []
        for move in valid_moves:
        quick_score = 0
        r, c = move

        # Highest priority: Corner moves (game-winning potential)
        if (r, c) in [(0, 0), (0, 7), (7, 0), (7, 7)]:
        quick_score += 1000

        # High priority: Moves that flip many pieces
        quick_score += len(valid_moves[move]) * 10

        # Strategic positional value from our matrix
        quick_score += POSITION_VALUES[r][c]

        move_scores.append((quick_score, move))

        # Sort moves: best first for maximizer, worst first for minimizer
        move_scores.sort(key=lambda x: x[0], reverse=maximizing_player)
        ordered_moves = [move for _, move in move_scores]

```

Listing 9: Move Ordering for Efficient Pruning

این مرتب‌سازی کارایی Pruning Alpha-Beta را به طور قابل توجهی افزایش می‌دهد زیرا:

- حذف زودهنگام شاخه‌ها: حرکات بهتر زودتر بررسی شده و منجر به حذف شاخه‌های ضعیف‌تر می‌شوند
- کاهش زمان جستجو: در بهترین حالت، Alpha-Beta از $O(b^d)$ به $O(b^{d/۲})$ کاهش می‌یابد
- عمق بیشتر در زمان محدود: با همان زمان محاسباتی، عمق بیشتری قابل دستیابی است

۷ تحلیل الگوهای استراتژیک

سیستم قادر به تشخیص و ارزیابی الگوهای مختلف استراتژیک است:

۱.۷ الگوی X-square

مربع‌های X (خانه‌هایی که به صورت قطری مجاور گوشه‌ها هستند، مانند (۱،۱) در کنار (۰،۰)) یکی از خطرناک‌ترین تله‌ها در بازی اتللو به شمار می‌روند. اشغال این خانه‌ها در حالی که گوشه مربوطه خالی است، یک اشتباه استراتژیک بزرگ محسوب می‌شود، زیرا به طور مستقیم به حریف اجازه می‌دهد تا در حرکت بعدی خود گوشه را تصاحب کند. هوش مصنوعی با اعمال جریمه سنگین برای قرار گرفتن در این موقعیت‌ها، از این خطای رایج اجتناب می‌کند. این جریمه تضمین می‌کند که حتی اگر اشغال مربع X منجر به برگرداندن تعداد زیادی مهره شود، ارزش منفی بلندمدتِ از دست دادن گوشه، بر مزیت کوتاه‌مدت غلبه خواهد کرد. در واقع، سیستم یاد گرفته است که دادن کنترل گوشه به حریف، تقریباً همیشه به شکست منجر می‌شود.

```
    def evaluate_patterns(board, player):
    score = 0
    opponent = -player

    # X-square pattern detection (dangerous squares next to corners)
    x_squares = [(1, 1), (1, 6), (6, 1), (6, 6)]
    corner_pairs = [
    ((0, 0), (1, 1)), ((0, 7), (1, 6)),
    ((7, 0), (6, 1)), ((7, 7), (6, 6))
    ]

    for (cr, cc), (xr, xc) in corner_pairs:
    if board[cr][cc] == EMPTY and board[xr][xc] == player:
    score -= 20  # Heavy penalty for X-square occupation

    return score

```

Listing 10: X-Square Pattern Detection

۲.۷ الگوی کنترل دیوار

«دیوار» به یک ردیف یا ستون در لبه‌های تخته گفته می‌شود که به طور کامل توسط یک بازیکن کنترل شده است. ساختن یک دیوار یک مزیت استراتژیک بسیار قدرتمند است، زیرا تمام مهره‌های موجود در آن دیوار پایدار می‌شوند و دیگر هرگز توسط حریف برگردانده نخواهند شد. این مهره‌های پایدار به عنوان یک لنگرگاه امن عمل کرده و به بازیکن اجازه می‌دهند تا با اطمینان به مناطق داخلی تخته نفوذ کند. تابع ارزیابی هوش مصنوعی، تشکیل چنین دیوارهایی را تشخیص داده و برای آن امتیاز مثبت بالایی در نظر می‌گیرد. این امتیاز به هوش مصنوعی انگیزه می‌دهد تا حرکاتی را انتخاب کند که به ایجاد یا تکمیل دیوارهای خودی کمک کرده و از تشکیل دیوارهای حریف جلوگیری کند. کنترل لبه‌ها به طور مستقیم به محدود کردن تحرک حریف و افزایش پایداری مهره‌های خودی منجر می‌شود.

```
    # Wall patterns (edges controlled by one player)
    for edge in [0, 7]:  # Top and bottom edges
    edge_control = sum(
    1 if board[edge][c] == player
    else -1 if board[edge][c] == opponent
    else 0 for c in range(8)
    )
    if abs(edge_control) > 4:  # Strong edge control
    score += edge_control * 5

    for edge in [0, 7]:  # Left and right edges
    edge_control = sum(
    1 if board[r][edge] == player
    else -1 if board[r][edge] == opponent
    else 0 for r in range(8)
    )
    if abs(edge_control) > 4:  # Strong edge control
    score += edge_control * 5

```

Listing 11: Wall Control Pattern Analysis

۸ تحلیل عملکرد و بهینه‌سازی

۱.۸ مدیریت زمان هوشمند

```
    def enhanced_ai_move_thread(game):
    try:
    start_time = time.time()
    total_pieces = sum(
    row.count(PLAYER_BLACK) + row.count(PLAYER_WHITE)
    for row in game.board
    )

    # Adaptive time allocation based on difficulty
    time_limits = {
      Difficulty.EASY: 1.0,
      Difficulty.MEDIUM: 3.0,
      Difficulty.HARD: 8.0,
      Difficulty.EXPERT: 15.0,
      Difficulty.GRANDMASTER: 30.0
    }

    time_limit = time_limits[AI_DIFFICULTY]

    # Run minimax with time constraint
    _, best_move, evaluated_moves = enhanced_minimax_alphabeta(
    game, AI_DIFFICULTY.value, -math.inf, math.inf, True,
```

```
        game.current_player, total_pieces, start_time, time_limit
        )

        # Store decision analysis for debugging
        game.ai_decision_log = evaluated_moves
        game.ai_think_time = time.time() - start_time

        # Minimum thinking time for realistic appearance
        min_think_time = 0.5
        if game.ai_think_time < min_think_time:
        time.sleep(min_think_time - game.ai_think_time)

        if best_move:
        pygame.event.post(
        pygame.event.Event(pygame.USEREVENT, {'move': best_move})
        )

        except Exception as e:
        print(f"AI Error: {e}")

```

Listing 12: Intelligent Time Management

۹ آنالیز نتایج و قدرت تصمیم‌گیری

۱.۹ مثال عملی: انتخاب بین دو حرکت

فرض کنید هوش مصنوعی در موقعیتی قرار دارد که دو حرکت اصلی در دسترس دارد:

- حرکت A: موقعیت (۱،۱) - مربع X نزدیک گوشه
- حرکت B: موقعیت (۳،۲) - موقعیت داخلی معمولی

۱.۱.۹ محاسبه امتیاز اولیه

```
        # Move A: Position (1,1) - X-square near corner
        move_a_score = 0
        move_a_score += len(flipped_pieces_a) * 10  # e.g., 3 pieces * 10 = 30
        move_a_score += POSITION_VALUES[1][1]       # -50 (very dangerous)
        move_a_score += 0  # No corner bonus
        # Total for Move A = 30 + (-50) + 0 = -20

        # Move B: Position (2,3) - Internal position
        move_b_score = 0
        move_b_score += len(flipped_pieces_b) * 10  # e.g., 2 pieces * 10 = 20
        move_b_score += POSITION_VALUES[2][3]       # -1 (slightly negative)
        move_b_score += 0  # No corner bonus
        # Total for Move B = 20 + (-1) + 0 = 19
```

```
14
```

Listing 13: Example Move Evaluation Calculation

۲.۱.۹ تصمیم نهایی

با توجه به محاسبات بالا:

- حرکت A امتیاز ۲۰- دارد (منفی به دلیل خطر X-square)
- حرکت B امتیاز ۱۹+ دارد (مثبت و امن)

بنابراین هوش مصنوعی حرکت B را انتخاب می‌کند زیرا:

۱. اجتناب از ریسک: موقعیت X-square خطر از دست دادن گوشه را به همراه دارد

۲. مزیت کوتاه‌مدت در برابر زیان بلندمدت: اگرچه حرکت A مهره‌های بیشتری می‌چرخاند، اما خطر استراتژیک آن بسیار بالاست

۳. حفظ موقعیت امن: حرکت B موقعیت کلی را بهبود می‌بخشد بدون ایجاد آسیب‌پذیری

۱۰ مقایسه با روش‌های جایگزین

۱.۱۰ مقایسه با جستجوی کامل

روش	پیچیدگی زمانی	کیفیت تصمیم	عملی بودن
جستجوی کامل	$O(b^n)$	کامل	غیرعملی
مینی‌مکس ساده	$O(b^d)$	خوب	محدود
مینی‌مکس + Alpha-Beta	$O(b^{d/۲})$	خوب	قابل قبول

۱۱ ضمائم

۱.۱۱ ضمیمه الف: کد کامل الگوریتم Minimax پیشرفته

```
1       def enhanced_minimax_alphabeta(game_state, depth, alpha, beta,
2       maximizing_player, ai_player,
3       total_pieces, start_time, time_limit=10.0):
4       """
5       Enhanced Minimax algorithm with Alpha-Beta Pruning, time management,
6       move ordering, and comprehensive evaluation for Othello AI
7
8       Parameters:
9       - game_state: Current state of the game
10      - depth: Remaining search depth
11      - alpha: Best score guaranteed for maximizing player
```

```
        - beta: Best score guaranteed for minimizing player
        - maximizing_player: True if current player wants to maximize score
        - ai_player: The AI player identifier (1 or -1)
        - total_pieces: Current number of pieces on board
        - start_time: Search start time for time management
        - time_limit: Maximum allowed thinking time
        """

        # Time management - prevent infinite thinking
        if time.time() - start_time > time_limit:
        return advanced_evaluate_board(game_state.board, ai_player,
        total_pieces, depth), None, []

        # Terminal condition check - reached maximum depth or game ended
        if depth == 0 or game_state.game_over:
        eval_score = advanced_evaluate_board(game_state.board, ai_player,
        total_pieces, depth)
        return eval_score, None, []

        valid_moves = game_state.valid_moves
        if not valid_moves:
        # Pass turn to opponent when no moves available (Othello rules)
        next_state = game_state.copy()
        next_state._switch_player()
        return enhanced_minimax_alphabeta(next_state, depth - 1, alpha, beta,
        not maximizing_player, ai_player,
        total_pieces, start_time, time_limit)

        # Move ordering for maximum Alpha-Beta pruning efficiency
        move_scores = []
        for move in valid_moves:
        quick_score = 0
        r, c = move

        # Priority 1: Corner moves have absolute strategic importance
        if (r, c) in [(0, 0), (0, 7), (7, 0), (7, 7)]:
        quick_score += 1000

        # Priority 2: Moves that flip more pieces
        quick_score += len(valid_moves[move]) * 10

        # Priority 3: Strategic positional value from matrix
        quick_score += POSITION_VALUES[r][c]

        move_scores.append((quick_score, move))

        # Sort moves for optimal pruning (best first for maximizer)
        move_scores.sort(key=lambda x: x[0], reverse=maximizing_player)
        ordered_moves = [move for _, move in move_scores]

        evaluated_moves = []
        best_move = ordered_moves[0]

        if maximizing_player:
```

```
        max_eval = -math.inf
        for move in ordered_moves:
        # Time constraint check during search
        if time.time() - start_time > time_limit:
        break

        # Generate next game state
        next_state = game_state.copy()
        next_state.make_move(move[0], move[1])

        # Recursive minimax call with player switch
        evaluation, _, _ = enhanced_minimax_alphabeta(
        next_state, depth - 1, alpha, beta, False,
        ai_player, total_pieces + 1, start_time, time_limit)

        evaluated_moves.append((evaluation, move))

        # Update best move if current evaluation is better
        if evaluation > max_eval:
        max_eval = evaluation
        best_move = move

        # Alpha-Beta pruning logic
        alpha = max(alpha, evaluation)
        if beta <= alpha:  # Cutoff condition
        break  # Remaining branches can be pruned

        evaluated_moves.sort(key=lambda x: x[0], reverse=True)
        return max_eval, best_move, evaluated_moves

        else:  # minimizing_player
        min_eval = math.inf
        for move in ordered_moves:
        # Time management during search
        if time.time() - start_time > time_limit:
        break

        # Create next game state
        next_state = game_state.copy()
        next_state.make_move(move[0], move[1])

        # Recursive call with maximizing player
        evaluation, _, _ = enhanced_minimax_alphabeta(
        next_state, depth - 1, alpha, beta, True,
        ai_player, total_pieces + 1, start_time, time_limit)

        evaluated_moves.append((evaluation, move))

        # Track minimum evaluation for minimizing player
        if evaluation < min_eval:
        min_eval = evaluation
        best_move = move

        # Beta update and pruning check
```

```
        beta = min(beta, evaluation)
        if beta <= alpha:  # Alpha-Beta cutoff
        break  # Prune remaining branches

        evaluated_moves.sort(key=lambda x: x[0])
        return min_eval, best_move, evaluated_moves

```

Listing 14: Complete Enhanced Minimax with Alpha-Beta Pruning

۲.۱۱ ضمیمه ب: کد کامل تابع ارزیابی

```
        def advanced_evaluate_board(board, player, total_pieces, depth_remaining=0):
        """
        Complete advanced evaluation function with all strategic components
        """
        opponent = -player

        # Phase-specific weight determination
        if total_pieces < 20:  # Opening
        phase_weights = {
          'piece': 1, 'mobility': 20, 'corner': 150, 'edge': 10,
          'stability': 100, 'position': 15, 'parity': 5
        }
        elif total_pieces < 52:  # Mid-game
        phase_weights = {
          'piece': 8, 'mobility': 15, 'corner': 120, 'edge': 15,
          'stability': 120, 'position': 12, 'parity': 10
        }
        else:  # End-game
        phase_weights = {
          'piece': 25, 'mobility': 8, 'corner': 140, 'edge': 20,
          'stability': 140, 'position': 5, 'parity': 30
        }

        score = 0

        # 1. Piece count with parity consideration
        my_pieces = sum(row.count(player) for row in board)
        opp_pieces = sum(row.count(opponent) for row in board)
        piece_diff = my_pieces - opp_pieces

        if total_pieces > 55:
        remaining_moves = 64 - total_pieces
        if remaining_moves % 2 == 1:
        piece_diff += 0.5

        score += phase_weights['piece'] * piece_diff

        # 2. Mobility calculation
```

```
        temp_game = Othello(sounds={})
        temp_game.board = board
        my_moves = len(temp_game.get_valid_moves(player))
        opp_moves = len(temp_game.get_valid_moves(opponent))

        if my_moves + opp_moves > 0:
        mobility_ratio = (my_moves - opp_moves) / (my_moves + opp_moves + 1)
        score += phase_weights['mobility'] * mobility_ratio * 100

        # Critical mobility situations
        if my_moves == 0 and opp_moves > 0:
        score -= 500
        elif opp_moves == 0 and my_moves > 0:
        score += 500

        # 3. Corner and adjacency analysis
        corners = [(0, 0), (0, 7), (7, 0), (7, 7)]
        corner_adjacencies = [
        [(0,1), (1,0), (1,1)], [(0,6), (1,7), (1,6)],
        [(6,0), (7,1), (6,1)], [(6,7), (7,6), (6,6)]
        ]

        my_corners = opp_corners = 0
        for i, (r, c) in enumerate(corners):
        if board[r][c] == player:
        my_corners += 1
        elif board[r][c] == opponent:
        opp_corners += 1
        elif board[r][c] == EMPTY:
        for adj_r, adj_c in corner_adjacencies[i]:
        if board[adj_r][adj_c] == player:
        score -= 25
        elif board[adj_r][adj_c] == opponent:
        score += 25

        score += phase_weights['corner'] * (my_corners - opp_corners)

        # 4. Edge control
        edges = [(i, 0) for i in range(8)] + [(i, 7) for i in range(8)] + \
        [(0, i) for i in range(1, 7)] + [(7, i) for i in range(1, 7)]

        my_edges = sum(1 for r, c in edges if board[r][c] == player)
        opp_edges = sum(1 for r, c in edges if board[r][c] == opponent)
        score += phase_weights['edge'] * (my_edges - opp_edges)

        # 5. Advanced stability
        my_stable = count_advanced_stable_pieces(board, player)
        opp_stable = count_advanced_stable_pieces(board, opponent)
        score += phase_weights['stability'] * (my_stable - opp_stable)

        # 6. Positional values from matrix
        position_score = 0
        for r in range(8):
        for c in range(8):
```

```
        if board[r][c] == player:
        position_score += POSITION_VALUES[r][c]
        elif board[r][c] == opponent:
        position_score -= POSITION_VALUES[r][c]
        score += phase_weights['position'] * position_score

        # 7. Pattern evaluation
        score += evaluate_patterns(board, player) * 10

        # 8. Depth bonus for deeper analysis
        if depth_remaining > 0:
        score += depth_remaining * 2

        return score

```

Listing 15: Complete Advanced Evaluation Function