

راهنمای کامل هوش مصنوعی بازی اُتلو

Complete Guide to Othello AI with Minimax Algorithm

محمد مهدی شریف بیگی

MohammadMahdi Sharifbeigy

Advanced AI Implementation with Enhanced Minimax

لینک ارائه

۱۲ شهریور ۱۴۰۴

فهرست مطالب

۳	۱	مقدمه
۳	۱.۱	ویژگی‌های کلیدی
۳	۲	الگوریتم مینی مکس و تصمیم‌گیری هوش مصنوعی
۳	۱.۲	اصول بنیادی الگوریتم مینی مکس
۳	۱.۱.۲	اصل عملکرد
۳	۲.۱.۲	فرمولاسیون ریاضی
۴	۲.۲	بهینه‌سازی Pruning Alpha-Beta
۴	۱.۲.۲	اصول Alpha-Beta
۴	۳	سیستم وزن‌دهی مکانی و استراتژی
۴	۱.۳	ماتریس ارزش‌های مکانی
۴	۲.۳	تحلیل عمیق ماتریس مکانی
۴	۱.۲.۳	گوشه‌ها (ارزش +۱۰۰)
۵	۲.۲.۳	مربع‌های مجاور گوشه (ارزش -۲۰)
۵	۳.۲.۳	مربع‌های X (ارزش -۵۰)
۵	۴.۲.۳	لبه‌ها (ارزش +۱۰/+۵)
۵	۵.۲.۳	مناطق داخلی (ارزش -۱/-۲)
۶	۳.۳	تأثیر ماتریس بر تصمیم‌گیری هوش مصنوعی
۶	۱.۳.۳	مرحله ارزیابی اولیه
۶	۲.۳.۳	تأثیر بر مرتب‌سازی حرکات

۴	تابع ارزیابی	۶
۱.۴	شمارش مهره‌ها با در نظرگیری Parity	۷
۲.۴	(Mobility Analysis)	۷
۳.۴	کنترل گوشه‌ها با جریمه مجاورت	۷
۴.۴	تحلیل پایداری پیشرفته	۸
۵	تطبیق فازی و استراتژی پویا	۹
۱.۵	تشخیص فاز بازی	۹
۲.۵	توضیح فازهای مختلف بازی	۱۰
۱.۲.۵	فاز شروع بازی (Opening Phase - تا ۲۰ مهره)	۱۰
۲.۲.۵	فاز میانه بازی (Mid-game Phase - ۲۰ تا ۵۲ مهره)	۱۱
۳.۲.۵	فاز پایان بازی (End-game Phase - بیش از ۵۲ مهره)	۱۱
۶	مدیریت عمق جستجو و سطوح هوش مصنوعی	۱۱
۱.۶	سیستم سطح‌بندی هوش مصنوعی	۱۱
۲.۶	تأثیر عمق جستجو بر قدرت تصمیم‌گیری	۱۱
۱.۲.۶	محاسبه تعداد حالات بررسی شده	۱۲
۳.۶	بهینه‌سازی‌های کارایی	۱۲
۱.۳.۶	مرتب‌سازی حرکات برای Alpha-Beta	۱۲
۷	تحلیل الگوهای استراتژیک	۱۳
۱.۷	الگوی X-square	۱۳
۲.۷	الگوی کنترل دیوار	۱۳
۸	تحلیل عملکرد و بهینه‌سازی	۱۴
۱.۸	مدیریت زمان هوشمند	۱۴
۹	آنالیز نتایج و قدرت تصمیم‌گیری	۱۵
۱.۹	مثال عملی: انتخاب بین دو حرکت	۱۵
۱.۱.۹	محاسبه امتیاز اولیه	۱۵
۲.۱.۹	تصمیم نهایی	۱۵
۱۰	مقایسه با روش‌های جایگزین	۱۶
۱.۱۰	مقایسه با جستجوی کامل	۱۶
۱۱	منابع و مراجع	۱۶
۱۲	ضمائم	۱۶
۱.۱۲	ضمیمه الف: کد کامل الگوریتم Minimax پیشرفته	۱۶
۲.۱۲	ضمیمه ب: کد کامل تابع ارزیابی	۱۹

۱ مقدمه

این مستند راهنمای جامعی برای درک و پیاده‌سازی هوش مصنوعی در بازی اتللو ارائه می‌دهد. این سیستم با استفاده از الگوریتم مینی مکس 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 مینی مکس 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 مینی مکس و تصمیم‌گیری هوش مصنوعی

۱.۲ اصول بنیادی الگوریتم مینی مکس

الگوریتم مینی مکس 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 مینی مکس 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:

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

۲.۲ بهینه‌سازی Pruning Alpha-Beta

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

۱.۲.۲ اصول Alpha-Beta

- α Alpha: بهترین امتیاز تضمین شده برای بازیکن Maximizer
- β Beta: 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:

```

1 POSITION_VALUES = [
2   [100, -20, 10, 5, 5, 10, -20, 100],
3   [-20, -50, -2, -2, -2, -2, -50, -20],
4   [ 10, -2, -1, -1, -1, -1, -2, 10],
5   [ 5, -2, -1, -1, -1, -1, -2, 5],
6   [ 5, -2, -1, -1, -1, -1, -2, 5],
7   [ 10, -2, -1, -1, -1, -1, -2, 10],
8   [-20, -50, -2, -2, -2, -2, -50, -20],
9   [100, -20, 10, 5, 5, 10, -20, 100]
10 ]
11
```

Listing 1: Position Values Matrix Definition

۲.۳ تحلیل عمیق ماتریس مکانی

این ماتریس بر اساس اصول استراتژیک بازی اتللو 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، نگرداشتن 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:

```

1  # Move ordering for better alpha-beta pruning
2  move_scores = []
3  for move in valid_moves:
4      quick_score = 0
5      r, c = move
6
7      # Corner priority - absolute strategic advantage
8      if (r, c) in [(0, 0), (0, 7), (7, 0), (7, 7)]:
9          quick_score += 1000 # Override position value with extreme priority
10
11     # High impact moves - number of pieces flipped
12     quick_score += len(valid_moves[move]) * 10
13
14     # Strategic positional value from matrix
15     quick_score += POSITION_VALUES[r][c]
16
17     move_scores.append((quick_score, move))
18
19     # Sort moves by strategic value (best first for maximizing player)
20     move_scores.sort(key=lambda x: x[0], reverse=True)
21     ordered_moves = [move for _, move in move_scores]
22

```

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

```

1  # Basic piece count difference
2  my_pieces = sum(row.count(player) for row in board)
3  opp_pieces = sum(row.count(opponent) for row in board)
4  piece_diff = my_pieces - opp_pieces
5
6  # Parity bonus in endgame - who gets the last moves
7  if total_pieces > 55:
8      remaining_moves = 64 - total_pieces
9      if remaining_moves % 2 == 1: # Odd number of moves left
10         piece_diff += 0.5 # Slight advantage to current player
11
12  score += phase_weights['piece'] * piece_diff
13

```

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)

```

1  # Calculate available moves for both players
2  temp_game = Othello(sounds={})
3  temp_game.board = board
4  my_moves = len(temp_game.get_valid_moves(player))
5  opp_moves = len(temp_game.get_valid_moves(opponent))
6
7  if my_moves + opp_moves > 0:
8      mobility_ratio = (my_moves - opp_moves) / (my_moves + opp_moves + 1)
9      score += phase_weights['mobility'] * mobility_ratio * 100
10
11  # Critical mobility situations
12  if my_moves == 0 and opp_moves > 0:
13      score -= 500 # Very dangerous position - no moves available
14  elif opp_moves == 0 and my_moves > 0:
15      score += 500 # Excellent position - opponent cannot move
16

```

Listing 4: Enhanced Mobility Calculation

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

۳.۴ کنترل گوشه‌ها با جریمه مجاورت

```

1  corners = [(0, 0), (0, 7), (7, 0), (7, 7)]
2  corner_adjacencies = [
3  [(0,1), (1,0), (1,1)], # Adjacent to (0,0)
4  [(0,6), (1,7), (1,6)], # Adjacent to (0,7)
5  [(6,0), (7,1), (6,1)], # Adjacent to (7,0)
6  [(6,7), (7,6), (6,6)] # Adjacent to (7,7)
7  ]
8
9  my_corners = opp_corners = 0
10 for i, (r, c) in enumerate(corners):
11     if board[r][c] == player:
12         my_corners += 1
13     elif board[r][c] == opponent:
14         opp_corners += 1
15     elif board[r][c] == EMPTY:
16         # Penalty for occupying squares adjacent to empty corners
17         for adj_r, adj_c in corner_adjacencies[i]:
18             if board[adj_r][adj_c] == player:
19                 score -= 25 # Dangerous position near empty corner
20             elif board[adj_r][adj_c] == opponent:
21                 score += 25 # Opponent is in dangerous position
22
23     score += phase_weights['corner'] * (my_corners - opp_corners)
24

```

Listing 5: Corner Control with Adjacency Penalties

۴.۴ تحلیل پایداری پیشرفته

```

1  def count_advanced_stable_pieces(board, player):
2      stable_count = 0
3
4      for r in range(8):
5          for c in range(8):
6              if board[r][c] == player:
7                  stability_score = 0
8
9              # Corner pieces are always completely stable
10             if (r, c) in [(0, 0), (0, 7), (7, 0), (7, 7)]:
11                 stable_count += 1
12                 continue
13
14             # Edge stability analysis
15             if r == 0 or r == 7 or c == 0 or c == 7:
16                 edge_stable = True
17                 # Check if entire edge is controlled
18                 if r == 0 or r == 7: # Top/bottom edge
19                     for dc in [-1, 1]:
20                         nc = c + dc

```



```

21 while 0 <= nc < 8:
22     if board[r][nc] != player:
23         edge_stable = False
24         break
25     nc += dc
26     if edge_stable:
27         stability_score += 1
28
29     # Internal stability (completely surrounded)
30     surrounded = True
31     for dr in [-1, 0, 1]:
32         for dc in [-1, 0, 1]:
33             if dr == 0 and dc == 0:
34                 continue
35             nr, nc = r + dr, c + dc
36             if 0 <= nr < 8 and 0 <= nc < 8:
37                 if board[nr][nc] != player:
38                     surrounded = False
39                     break
40             if not surrounded:
41                 break
42
43     if surrounded:
44         stability_score += 0.5
45
46     stable_count += stability_score
47
48     return stable_count
49

```

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:

```

1 def advanced_evaluate_board(board, player, total_pieces, depth_remaining=0):
2     opponent = -player
3
4     # Dynamic phase detection based on total pieces on board
5     if total_pieces < 20: # Opening phase
6         phase_weights = {
7             'piece': 1,          # Piece count less important early
8             'mobility': 20,      # Mobility is crucial in opening
9             'corner': 150,       # Corner control extremely important
10            'edge': 10,          # Edge control moderately important
11            'stability': 100,     # Piece stability matters
12            'position': 15,       # Positional values important

```

```

13     'parity': 5          # Parity less relevant early
14 }
15 elif total_pieces < 52: # Mid-game phase
16     phase_weights = {
17         'piece': 8,      # Piece count starts mattering more
18         'mobility': 15,  # Mobility remains important
19         'corner': 120,   # Corners still crucial
20         'edge': 15,      # Edge control more important
21         'stability': 120, # Stability becomes critical
22         'position': 12,  # Position values matter
23         'parity': 10     # Parity consideration increases
24     }
25 else: # End-game phase
26     phase_weights = {
27         'piece': 25,      # Piece count is final victory condition
28         'mobility': 8,    # Mobility less important
29         'corner': 140,    # Corners decide endgame
30         'edge': 20,       # Edge control very important
31         'stability': 140, # Stability determines final outcome
32         'position': 5,    # Position values less relevant
33         'parity': 30      # Parity becomes crucial
34     }
35
36     score = 0
37
38     # Apply all evaluation components with phase-specific weights
39     score += phase_weights['piece'] * piece_diff
40     score += phase_weights['mobility'] * mobility_ratio * 100
41     score += phase_weights['corner'] * (my_corners - opp_corners)
42     # ... other components
43
44     return score
45

```

Listing 7: Dynamic Phase Detection and Weighting

۲.۵ توضیح فازهای مختلف بازی

۱.۲.۵ فاز شروع بازی (Opening Phase - ۲۰ تا 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

```

1  class Difficulty(Enum):
2      EASY = 2          # Depth 2 - basic planning
3      MEDIUM = 4       # Depth 4 - intermediate strategy
4      HARD = 6          # Depth 6 - advanced planning
5      EXPERT = 8        # Depth 8 - expert-level analysis
6      GRANDMASTER = 10 # Depth 10 - master-level deep analysis
7
8      # Adaptive time limits for each difficulty
9      time_limits = {
10         Difficulty.EASY: 1.0,      # 1 second thinking time
11         Difficulty.MEDIUM: 3.0,    # 3 seconds thinking time
12         Difficulty.HARD: 8.0,      # 8 seconds thinking time
13         Difficulty.EXPERT: 15.0,   # 15 seconds thinking time
14         Difficulty.GRANDMASTER: 30.0 # 30 seconds thinking time
15     }
16

```

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: $b^d \approx$ حالات بررسی شده

برای اتللو معمولاً $b \approx 8$ است، بنابراین:

- عمق ۲: $8^2 = 64$ حالت
- عمق ۴: $8^4 = 4,096$ حالت
- عمق ۶: $8^6 = 262,144$ حالت
- عمق ۸: $8^8 = 16,777,216$ حالت
- عمق ۱۰: $8^{10} = 1,073,741,824$ حالت

۳.۶ بهینه‌سازی‌های کارایی

۱.۳.۶ مرتب‌سازی حرکات برای Alpha-Beta

```

1  # Strategic move ordering based on position values and game knowledge
2  move_scores = []
3  for move in valid_moves:
4      quick_score = 0
5      r, c = move
6
7      # Highest priority: Corner moves (game-winning potential)
8      if (r, c) in [(0, 0), (0, 7), (7, 0), (7, 7)]:
9          quick_score += 1000
10
11     # High priority: Moves that flip many pieces
12     quick_score += len(valid_moves[move]) * 10
13
14     # Strategic positional value from our matrix
15     quick_score += POSITION_VALUES[r][c]
16
17     move_scores.append((quick_score, move))
18
19     # Sort moves: best first for maximizer, worst first for minimizer
20     move_scores.sort(key=lambda x: x[0], reverse=maximizing_player)
21     ordered_moves = [move for _, move in move_scores]
22

```

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 $O(b^{d/2})$ 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 بلندمدت 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.

```

1  def evaluate_patterns(board, player):
2      score = 0
3      opponent = -player
4
5      # X-square pattern detection (dangerous squares next to corners)
6      x_squares = [(1, 1), (1, 6), (6, 1), (6, 6)]
7      corner_pairs = [
8          ((0, 0), (1, 1)), ((0, 7), (1, 6)),
9          ((7, 0), (6, 1)), ((7, 7), (6, 6))
10     ]
11
12     for (cr, cc), (xr, xc) in corner_pairs:
13         if board[cr][cc] == EMPTY and board[xr][xc] == player:
14             score -= 20 # Heavy penalty for X-square occupation
15
16     return score
17

```

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.

```

1  # Wall patterns (edges controlled by one player)
2  for edge in [0, 7]: # Top and bottom edges
3      edge_control = sum(
4          1 if board[edge][c] == player
5          else -1 if board[edge][c] == opponent
6          else 0 for c in range(8)
7      )
8      if abs(edge_control) > 4: # Strong edge control
9          score += edge_control * 5
10
11     for edge in [0, 7]: # Left and right edges
12         edge_control = sum(
13             1 if board[r][edge] == player
14             else -1 if board[r][edge] == opponent
15             else 0 for r in range(8)
16         )
17         if abs(edge_control) > 4: # Strong edge control
18             score += edge_control * 5
19

```

Listing 11: Wall Control Pattern Analysis

۸ تحلیل عملکرد و بهینه‌سازی

۱.۸ مدیریت زمان هوشمند

```

1  def enhanced_ai_move_thread(game):
2      try:
3          start_time = time.time()
4          total_pieces = sum(
5              row.count(PYER_BLACK) + row.count(PYER_WHITE)
6              for row in game.board
7          )
8
9          # Adaptive time allocation based on difficulty
10         time_limits = {
11             Difficulty.EASY: 1.0,
12             Difficulty.MEDIUM: 3.0,
13             Difficulty.HARD: 8.0,
14             Difficulty.EXPERT: 15.0,
15             Difficulty.GRANDMASTER: 30.0
16         }
17
18         time_limit = time_limits[AI_DIFFICULTY]
19
20         # Run minimax with time constraint
21         _, best_move, evaluated_moves = enhanced_minimax_alphabeta(
22             game, AI_DIFFICULTY.value, -math.inf, math.inf, True,

```

```

23     game.current_player, total_pieces, start_time, time_limit
24 )
25
26 # Store decision analysis for debugging
27 game.ai_decision_log = evaluated_moves
28 game.ai_think_time = time.time() - start_time
29
30 # Minimum thinking time for realistic appearance
31 min_think_time = 0.5
32 if game.ai_think_time < min_think_time:
33     time.sleep(min_think_time - game.ai_think_time)
34
35 if best_move:
36     pygame.event.post(
37         pygame.event.Event(pygame.USEREVENT, {'move': best_move})
38     )
39
40 except Exception as e:
41     print(f"AI Error: {e}")
42

```

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

```

1 # Move A: Position (1,1) - X-square near corner
2 move_a_score = 0
3 move_a_score += len(flipped_pieces_a) * 10 # e.g., 3 pieces * 10 = 30
4 move_a_score += POSITION_VALUES[1][1]      # -50 (very dangerous)
5 move_a_score += 0 # No corner bonus
6 # Total for Move A = 30 + (-50) + 0 = -20
7
8 # Move B: Position (2,3) - Internal position
9 move_b_score = 0
10 move_b_score += len(flipped_pieces_b) * 10 # e.g., 2 pieces * 10 = 20
11 move_b_score += POSITION_VALUES[2][3]      # -1 (slightly negative)
12 move_b_score += 0 # No corner bonus
13 # Total for Move B = 20 + (-1) + 0 = 19

```

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
مینی مکس ساده	$O(b^d)$	خوب	محدود
مینی مکس + Alpha-Beta	$O(b^{d/2})$	خوب	قابل قبول

۱۱ منابع و مراجع

۱. P. Rosenbloom. (۱۹۸۲). Program Othello World-Championship-Level A.

۲. M. Buro. (۱۹۹۷). Logistello vs. Murakami Takeshi Year: the of Match Othello The.

۳. S. Russell, P. Norvig. (۲۰۲۰). Approach Modern A Intelligence: Artificial.

۴. A. T. Marsland. (۱۹۸۶). Pruning Game-Tree of Review A.

۵. S. I. Herschberg, J., H. Herik, den van. (۱۹۸۵). Program Othello an of Construction The.

۱۲ ضمایم

۱.۱۲ 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
12 - beta: Best score guaranteed for minimizing player
13 - maximizing_player: True if current player wants to maximize score
14 - ai_player: The AI player identifier (1 or -1)
15 - total_pieces: Current number of pieces on board
16 - start_time: Search start time for time management
17 - time_limit: Maximum allowed thinking time
18 """
19
20 # Time management - prevent infinite thinking
21 if time.time() - start_time > time_limit:
22     return advanced_evaluate_board(game_state.board, ai_player,
23     total_pieces, depth), None, []
24
25 # Terminal condition check - reached maximum depth or game ended
26 if depth == 0 or game_state.game_over:
27     eval_score = advanced_evaluate_board(game_state.board, ai_player,
28     total_pieces, depth)
29     return eval_score, None, []
30
31 valid_moves = game_state.valid_moves
32 if not valid_moves:
33     # Pass turn to opponent when no moves available (Othello rules)
34     next_state = game_state.copy()
35     next_state._switch_player()
36     return enhanced_minimax_alphabeta(next_state, depth - 1, alpha, beta,
37     not maximizing_player, ai_player,
38     total_pieces, start_time, time_limit)
39
40 # Move ordering for maximum Alpha-Beta pruning efficiency
41 move_scores = []
42 for move in valid_moves:
43     quick_score = 0
44     r, c = move
45
46 # Priority 1: Corner moves have absolute strategic importance
47 if (r, c) in [(0, 0), (0, 7), (7, 0), (7, 7)]:
48     quick_score += 1000

```

```

49
50 # Priority 2: Moves that flip more pieces
51 quick_score += len(valid_moves[move]) * 10
52
53 # Priority 3: Strategic positional value from matrix
54 quick_score += POSITION_VALUES[r][c]
55
56 move_scores.append((quick_score, move))
57
58 # Sort moves for optimal pruning (best first for maximizer)
59 move_scores.sort(key=lambda x: x[0], reverse=maximizing_player)
60 ordered_moves = [move for _, move in move_scores]
61
62 evaluated_moves = []
63 best_move = ordered_moves[0]
64
65 if maximizing_player:
66     max_eval = -math.inf
67     for move in ordered_moves:
68         # Time constraint check during search
69         if time.time() - start_time > time_limit:
70             break
71
72     # Generate next game state
73     next_state = game_state.copy()
74     next_state.make_move(move[0], move[1])
75
76     # Recursive minimax call with player switch
77     evaluation, _, _ = enhanced_minimax_alphabeta(
78         next_state, depth - 1, alpha, beta, False,
79         ai_player, total_pieces + 1, start_time, time_limit)
80
81     evaluated_moves.append((evaluation, move))
82
83     # Update best move if current evaluation is better
84     if evaluation > max_eval:
85         max_eval = evaluation
86         best_move = move
87
88     # Alpha-Beta pruning logic
89     alpha = max(alpha, evaluation)
90     if beta <= alpha: # Cutoff condition
91         break # Remaining branches can be pruned
92
93     evaluated_moves.sort(key=lambda x: x[0], reverse=True)
94     return max_eval, best_move, evaluated_moves
95
96 else: # minimizing_player
97     min_eval = math.inf
98     for move in ordered_moves:
99         # Time management during search
100         if time.time() - start_time > time_limit:
101             break

```

```

103 # Create next game state
104 next_state = game_state.copy()
105 next_state.make_move(move[0], move[1])
106
107 # Recursive call with maximizing player
108 evaluation, _, _ = enhanced_minimax_alphabeta(
109     next_state, depth - 1, alpha, beta, True,
110     ai_player, total_pieces + 1, start_time, time_limit)
111
112 evaluated_moves.append((evaluation, move))
113
114 # Track minimum evaluation for minimizing player
115 if evaluation < min_eval:
116     min_eval = evaluation
117     best_move = move
118
119 # Beta update and pruning check
120 beta = min(beta, evaluation)
121 if beta <= alpha: # Alpha-Beta cutoff
122     break # Prune remaining branches
123
124 evaluated_moves.sort(key=lambda x: x[0])
125 return min_eval, best_move, evaluated_moves
126

```

Listing 14: Complete Enhanced Minimax with Alpha-Beta Pruning

۲.۱۲ ضمیمه ب: کد کامل تابع ارزیابی

```

1 def advanced_evaluate_board(board, player, total_pieces, depth_remaining=0):
2     """
3     Complete advanced evaluation function with all strategic components
4     """
5     opponent = -player
6
7     # Phase-specific weight determination
8     if total_pieces < 20: # Opening
9         phase_weights = {
10             'piece': 1, 'mobility': 20, 'corner': 150, 'edge': 10,
11             'stability': 100, 'position': 15, 'parity': 5
12         }
13     elif total_pieces < 52: # Mid-game
14         phase_weights = {
15             'piece': 8, 'mobility': 15, 'corner': 120, 'edge': 15,
16             'stability': 120, 'position': 12, 'parity': 10
17         }
18     else: # End-game
19         phase_weights = {
20             'piece': 25, 'mobility': 8, 'corner': 140, 'edge': 20,
21             'stability': 140, 'position': 5, 'parity': 30

```

```

22     }
23
24     score = 0
25
26     # 1. Piece count with parity consideration
27     my_pieces = sum(row.count(player) for row in board)
28     opp_pieces = sum(row.count(opponent) for row in board)
29     piece_diff = my_pieces - opp_pieces
30
31     if total_pieces > 55:
32         remaining_moves = 64 - total_pieces
33         if remaining_moves % 2 == 1:
34             piece_diff += 0.5
35
36     score += phase_weights['piece'] * piece_diff
37
38     # 2. Mobility calculation
39     temp_game = Othello(sounds={})
40     temp_game.board = board
41     my_moves = len(temp_game.get_valid_moves(player))
42     opp_moves = len(temp_game.get_valid_moves(opponent))
43
44     if my_moves + opp_moves > 0:
45         mobility_ratio = (my_moves - opp_moves) / (my_moves + opp_moves + 1)
46         score += phase_weights['mobility'] * mobility_ratio * 100
47
48     # Critical mobility situations
49     if my_moves == 0 and opp_moves > 0:
50         score -= 500
51     elif opp_moves == 0 and my_moves > 0:
52         score += 500
53
54     # 3. Corner and adjacency analysis
55     corners = [(0, 0), (0, 7), (7, 0), (7, 7)]
56     corner_adjacencies = [
57         [(0,1), (1,0), (1,1)], [(0,6), (1,7), (1,6)],
58         [(6,0), (7,1), (6,1)], [(6,7), (7,6), (6,6)]
59     ]
60
61     my_corners = opp_corners = 0
62     for i, (r, c) in enumerate(corners):
63         if board[r][c] == player:
64             my_corners += 1
65         elif board[r][c] == opponent:
66             opp_corners += 1
67         elif board[r][c] == EMPTY:
68             for adj_r, adj_c in corner_adjacencies[i]:
69                 if board[adj_r][adj_c] == player:
70                     score -= 25
71                 elif board[adj_r][adj_c] == opponent:
72                     score += 25
73
74     score += phase_weights['corner'] * (my_corners - opp_corners)
75

```

```

76 # 4. Edge control
77 edges = [(i, 0) for i in range(8)] + [(i, 7) for i in range(8)] + \
78 [(0, i) for i in range(1, 7)] + [(7, i) for i in range(1, 7)]
79
80 my_edges = sum(1 for r, c in edges if board[r][c] == player)
81 opp_edges = sum(1 for r, c in edges if board[r][c] == opponent)
82 score += phase_weights['edge'] * (my_edges - opp_edges)
83
84 # 5. Advanced stability
85 my_stable = count_advanced_stable_pieces(board, player)
86 opp_stable = count_advanced_stable_pieces(board, opponent)
87 score += phase_weights['stability'] * (my_stable - opp_stable)
88
89 # 6. Positional values from matrix
90 position_score = 0
91 for r in range(8):
92     for c in range(8):
93         if board[r][c] == player:
94             position_score += POSITION_VALUES[r][c]
95         elif board[r][c] == opponent:
96             position_score -= POSITION_VALUES[r][c]
97     score += phase_weights['position'] * position_score
98
99 # 7. Pattern evaluation
100 score += evaluate_patterns(board, player) * 10
101
102 # 8. Depth bonus for deeper analysis
103 if depth_remaining > 0:
104     score += depth_remaining * 2
105
106 return score
107

```

Listing 15: Complete Advanced Evaluation Function