MỤC LỤC

Contents

[Example 2](#_Toc212318089)

[Tic-Tac-Toe 2](#_Toc212318090)

[Solving Tic-Tac-Toe with AND-OR-Tree Search 3](#_Toc212318091)

[Play Tic-Tac-Toe Interactively (Simple Implementation) 6](#_Toc212318092)

[Solving Tic-Tac-Toe with Minimax Search and Alpha-Beta Pruning 7](#_Toc212318093)

[Phương pháp 1: Recursive DFS algorithm for minimax search 8](#_Toc212318094)

[Phương pháp 2: Cắt tỉa Alpha-Beta (Alpha-Beta Pruning) 11](#_Toc212318095)

[Phương pháp 3: Sắp xếp nước đi (Move Ordering) 13](#_Toc212318096)

[Solving Tic-Tac-Toe with Heuristic Alpha-Beta Tree Search 14](#_Toc212318097)

[Solving Tic-Tac-Toe with Monte Carlo Tree Search 19](#_Toc212318098)

[Phương thức UCB1() 22](#_Toc212318099)

[Các cột trong bảng 24](#_Toc212318100)

[Solving Tic-Tac-Toe with Monte Carlo Tree Search Restricted 27](#_Toc212318101)

[Result: 29](#_Toc212318102)

[Pure Monte Carlo Search vs. Random: 30](#_Toc212318103)

# Example

# Tic-Tac-Toe

Không gian trạng thái

Không gian trạng thái tổng ước tính là 39 trạng thái (vì mỗi ô có 3 giá trị mà tổng cộng 9 ô).

Ràng buộc chặt chẽ hơn về không gian tìm kiếm, ta có thể chọn  ô từ bảng được cho bởi  (chập i của 9) và trong đấy ta có thể chọn được 1 nửa là của người chơi o bởi . Kết hợp lại, ta có công thức . Sau khi chạy đoạn code, ta tính

được không gian tìm kiếm giảm mạnh thành 5919 trạng thái.

Độ phức tạp của không gian và thời gian khi sử dụng “search tree heuristic”.

Ta có độ sâu tối đa của cây tìm kiếm hoàn chỉnh là  và hệ số phân nhánh tối đa cho một nhánh là . Và DFS nên có độ phức tạp không gian  (đường đi hiện tại cộng với các node con chưa duyệt) và độ phức tạp thời gian  (số lượng nút mở rộng). Tuy nhiên trong thực tế, tổng số lượng nút trong cây trò chơi lúc hơn . Vì ở tầng thứ 1, hệ số phân nhánh sẽ là 9 nhưng khi xuống tới tầng thứ 2 hệ số phân nhánh sẽ là 8 và tương tự có các tầng tiếp theo. Tổng số lượng nút của bài toán tic-tac-toe là 986410.

Implementation

Ta cần cài đặt một số hàm sau:

· Action(s): Các nước đi hợp lệ của trạng thái s.

· Result(s, a): Kết quả của trạng thái s sau khi đi nước a.

· Terminal(s): Kiểm tra trạng thái cuối cùng.

· Utility(s): Đánh giá kết quả cuối cùng cho người chơi x.

Thí nghiệm

Sau khi để 2 random agent chơi 100 game tic-tac-toe, người chơi x đã thắng 68 lần, thua 24 lần, 8 lần hòa.

# Solving Tic-Tac-Toe with AND-OR-Tree Search

AND-OR-tree search là 1 thuật toán thuộc loại hành động không xác định (Nondeterministic actions)

Solving Tic-Tac-Toe using AND-OR-Tree Search

Non-deterministic Results

Hàm Results không xác định (non-deterministic) sẽ trả về cho trạng thái x1 tập các trạng thái mới mà phản ánh tất cả các phản hồi khả thi từ đối thủ.

Recursive AND-OR Search (DFS)

Gồm 3 hàm chính and\_or\_search(), or\_search(), and\_search(). Cụ thể, hàm and\_or\_search() sẽ bắt đầu cây AND-OR-tree search và nhận 3 tham số là state(trạng thái hiện tại của trò chơi), player (người chơi hiện tại), draw\_is\_win(được sử dụng để xem kết quả hòa có được tính là chiến thắng hay không) và hàm sẽ trả về kế hoạch hành động có điều kiện(chiến lược tối ưu có thể người chơi tới chiến thắng).

Hàm or\_search() biểu diễn bước OR trong cây tìm kiếm và hàm này cũng nhận 3 tham số như hàm and\_or\_search(), có chức năng chính là khi người chơi thực hiện 1 nước đi, hàm sẽ thử mọi khả năng có thể và trả về 1 kế hành động có điều kiện cho hành động đầu tiên, trong đó chỉ có trạng thái mục tiêu là nút lá. Nếu không tìm thấy thì trả về None.

Hàm and\_search() biểu diễn bước AND trong cây tìm kiếm và tham số nhận vào tương tự như hàm and\_or\_search(). Đại diện cho tất cả các nước đi có thể của đối thủ. Hàm sẽ theo dõi tất cả các trạng thái khả thi(gọi or\_search()) và chỉ trả về kế hoạch hành động có điều kiện nếu tất cả các đường dẫn tới trạng thái mục tiêu(goal state).

Some Tests

Với ví dụ coi kết quả hòa như là thắng,

Với output là:

Win or draw:

Number of nodes searched: 22

[2,

[['if', ['x', 'o', 'x', 'o', 'x', 'o', ' ', ' ', ' '], 'then', [6, 'win']],

['if',

['x', 'o', 'x', 'o', 'x', ' ', 'o', ' ', ' '],

'then',

[5,

[['if', ['x', 'o', 'x', 'o', 'x', 'x', 'o', 'o', ' '], 'then', [8, 'win']],

['if',

['x', 'o', 'x', 'o', 'x', 'x', 'o', ' ', 'o'],

'then',

[7, 'draw']]]]],

['if',

['x', 'o', 'x', 'o', 'x', ' ', ' ', 'o', ' '],

'then',

[5,

[['if', ['x', 'o', 'x', 'o', 'x', 'x', 'o', 'o', ' '], 'then', [8, 'win']],

['if',

['x', 'o', 'x', 'o', 'x', 'x', ' ', 'o', 'o'],

'then',

[6, 'win']]]]],

['if',

['x', 'o', 'x', 'o', 'x', ' ', ' ', ' ', 'o'],

'then',

[5,

[['if',

['x', 'o', 'x', 'o', 'x', 'x', 'o', ' ', 'o'],

'then',

[7, 'draw']],

['if',

['x', 'o', 'x', 'o', 'x', 'x', ' ', 'o', 'o'],

'then',

[6, 'win']]]]]]]

CPU times: user 3.14 ms, sys: 966 μs, total: 4.11 ms

Wall time: 4.03 ms

Hàm and\_or\_search() đã tìm kiếm tổng cộng 22 nút mà trong đó có 7 kế hoạch hành động dẫn tới chiến thắng cho người chơi x và 2 lần dẫn đến hòa. Nhưng có 1 vấn đề của là người chơi x có thể thắng với việc đánh vào ô 8 nhưng thuật toán AND-OR-tree search lại không đánh ở ô 2 bởi vì khi đánh ở ô 2 thì thuật toán vẫn tìm được kế hoạch hành động có điều kiện để dẫn tới trạng thái đích và hàm or\_search()sẽ trả kết quả về cho and\_or\_search() và kết thúc, dẫn tới các ô khả khác như 5,6,7,8 đều không được đánh thử.

Và đây là output nếu không tính kết quả hòa cũng là thắng.

Win only:

Number of nodes searched: 30

[2,

[['if', ['x', 'o', 'x', 'o', 'x', 'o', ' ', ' ', ' '], 'then', [6, 'win']],

['if', ['x', 'o', 'x', 'o', 'x', ' ', 'o', ' ', ' '], 'then', [8, 'win']],

['if',

['x', 'o', 'x', 'o', 'x', ' ', ' ', 'o', ' '],

'then',

[5,

[['if', ['x', 'o', 'x', 'o', 'x', 'x', 'o', 'o', ' '], 'then', [8, 'win']],

['if',

['x', 'o', 'x', 'o', 'x', 'x', ' ', 'o', 'o'],

'then',

[6, 'win']]]]],

['if', ['x', 'o', 'x', 'o', 'x', ' ', ' ', ' ', 'o'], 'then', [6, 'win']]]]

CPU times: user 5.48 ms, sys: 0 ns, total: 5.48 ms

Wall time: 4.98 ms

Thuật toán AND-OR-tree search vẫn lựa chọn nước đi là 8 tiếp nhưng lần này chỉ còn đúng 5 lần chiến thắng và không còn kế hoạch hành động có điều kiện dẫn tới hòa nữa.

Experiments

And-Or Tree Search vs. Random

Sau khi chơi 100 ván tic-tac-toe, với lợi thế được đi trước, thuật toán AND-OR-tree search đã dành được kết quả 92 thắng và 8 hòa.

Random vs. And-Or Tree Search

Tuy bị mất lợi thế đi trước vào thuật toán Random, vì đi nước đi ngẫu nhiên nên thuật toán AND-OR-tree search vẫn dành được 69 chiến thắng và 31 trận hòa.

And-Or Tree Search vs. And-Or Tree Search

Khi không còn sử dụng thuật toán ngẫu nhiên và chỉ đánh đúng 1 ván, thì kết quả của ván này hòa vì cả 2 thuật toán đều cố gắng tìm ra nước đi tốt nhất của mình.

# Play Tic-Tac-Toe Interactively (Simple Implementation)

Interactive Player

from IPython.display import clear\_output

def interactive\_player(board, player = None):

clear\_output(wait = False)

show\_board(board, help = True)

available = actions(board)

print(f'Available actions are: {available}')

retry = True

while retry:

try:

move = int(input("Your move:\n"))

if move in available:

retry = False

else:

raise ValueError()

except ValueError:

print("Please enter a valid move.")

return(move)

Hàm interactive\_player() được dùng để lấy nước đi của người chơi thật(human player) thay vì để máy(AI agent) tự chọn nước đi. Ban đầu, hàm sẽ xóa đi các dòng in trước đó và hiển thị bàn cờ hiện tại. Sau đó, liệt kê tất cả các ô còn trống mà người chơi có thể đánh. Tiếp đó, người chơi sẽ chọn nước đi. Nếu hợp lệ trả nước đi cho chương trình chính để cập nhật trạng thái bàn cờ, ngược lại, người chơi sẽ phải nhập lại nước đi.

Start an interactive game as x

play(interactive\_player, random\_player, N = 1, show\_final\_board=True)

Available actions are: [5]

{'x': 1, 'o': 0, 'd': 0}

Ta thấy với thí nghiệm này, vì chỉ còn 1 nước đi hợp lệ cho người chơi x và sau khi nước này người chơi x đã giành chiến thắng.

Start an interactive game as o

play(random\_player, interactive\_player, N = 1, show\_final\_board = True)

Available actions are: [1, 2, 3, 8]

{'x': 0, 'o': 1, 'd': 0}

Tương tự như ví dụ trước, nhưng ở đây lượt đi trước lại là o và người chơi(human player) đã chọn ô 3 trong số 4 ô hợp lệ và giành chiến thắng.

# Solving Tic-Tac-Toe with Minimax Search and Alpha-Beta Pruning

Giới thiệu bài toán

Bài toán giải quyết trò chơi Tic Tac Toe, một trò chơi hai người chơi, có tổng bằng không (zero-sum game).

- Bản chất: Đây là một trò chơi đối kháng (adversarial search), nơi một người chơi (max) cố gắng thắng, và người chơi kia (min) cũng cố gắng thắng (đồng nghĩa với việc làm cho max thua).

- Người chơi:

o Max (x): Cố gắng tối đa hóa kết quả.

o Min (o): Cố gắng tối thiểu hóa kết quả.

- Giá trị (Utillity): kết quả của trò chơi được định lượng:

o X thắng +1

o O thắng -1

o Hòa 0

- Mục tiêu: Thuật toán phải xác định được nước đi “Tối ưu” từ trạng thái bàn cờ hiện tại, giả định đối thủ cũng chơi tối ưu

Phương pháp làm:

Ở đây có ba phương pháp chính và tăng dần về độ phức tạp và hiệu quả:

## Phương pháp 1: Recursive DFS algorithm for minimax search

Đây là thuật toán tìm kiếm đệ quy theo chiều sâu (DFS). Nó duyệt qua toàn bộ cây trò chơi có thể có từ trạng thái hiện tại.

Code:

# global variables

DEBUG = 1 # 1 ... count nodes, 2 ... debug each node

COUNT = 0

def minimax\_search(board, player = 'x'):

"""start the search."""

global DEBUG, COUNT

COUNT = 0

value, move = max\_value(board, player)

if DEBUG >= 1: print(f"Number of nodes searched: {COUNT}")

return { "move": move, "value": value}

def max\_value(state, player):

"""player's best move."""

global DEBUG, COUNT

COUNT += 1

# return utility of state if it is a terminal state

v = utility(state, player)

if DEBUG >= 2: print("max in: " + str(state) + str([v]) )

if v is not None: return v, None

v, move = -math.inf, None

# check all possible actions in the state, return move with the largest value

for a in actions(state):

v2, a2 = min\_value(result(state, player, a), player)

if v2 > v:

v, move = v2, a

if DEBUG >= 2: print("max out: " + str(state) + str([v, move]) )

return v, move

def min\_value(state, player):

"""opponent's best response."""

global DEBUG, COUNT

COUNT += 1

# return utility of state if it is a terminal state

v = utility(state, player)

if DEBUG >= 2: print("min in: " + str(state) + str([v]) )

if v is not None: return v, None

v, move = +math.inf, None

# check all possible actions in the state, return move with the smallest value

for a in actions(state):

v2, a2 = max\_value(result(state, other(player), a), player)

if v2 < v:

v, move = v2, a

if DEBUG >= 2: print("min out: " + str(state) + str([v, move]) )

return v, move

Hàm hỗ trợ:

def utility(state, player = 'x'):

"""utility of state. None defined for non-terminal states."""

goal = check\_board(state)

if goal == player: return +1 # win

if goal == 'd': return 0 # draw

if goal == other(player): return -1 # loss

return None # utility is not defined

minimax\_search(board, player = 'x'): Hàm khởi động chính, gọi max\_value để bắt đầu.

max\_value(state, player): (Hàm cho người chơi max)

- Kiểm tra utility(state, player = 'x') Nếu là trạng thái kết thúc (thắng/thua/hòa), trả về giá trị đó

- Khởi tạo giá trị tốt nhất v = -math.inf

- Lặp qua tất cả các actions (nước đi) có thể

- Với mỗi bước đi, nó gọi đệ quy min\_value (giả lập đối thủ chơi nước tốt nhất của họ)

- Nó chọn nước đi nào trả về giá trị v2 lớn nhất (cập nhật v = max(v,v2)).

min\_value(state, player): hàm cho người chơi min

- Tương tự, kiểm tả utility

- Khởi tạo giá trị tốt nhất v=+math.inf

- Lặp qua các actions

- Với mỗi nước đi, nó gọi đệ quy max\_value (giả lập MAX chơi nước tốt nhất của mình)

- Nó chọn nước đi nào trả về giá trị v2 nhỏ nhất (cập nhật v = min(v, v2)).

Kết quả: Luôn tìm ra nước đi tối ưu. Rất chậm. Khi chạy với bàn cờ trống, nó phải duyệt 549,946 nút (trạng thái)

## Phương pháp 2: Cắt tỉa Alpha-Beta (Alpha-Beta Pruning)

**Ý tưởng:** Đây là một sự **tối ưu hóa** của Minimax. Nó giúp giảm đáng kể số lượng nút cần duyệt bằng cách "cắt tỉa" những nhánh tìm kiếm mà nó biết chắc sẽ không bao giờ được chọn.

Code:

DEBUG = 1 # 1 ... count nodes, 2 ... debug each node

COUNT = 0

def alpha\_beta\_search(board, player = 'x'):

"""start the search."""

global DEBUG, COUNT

COUNT = 0

value, move = max\_value\_ab(board, player, -math.inf, +math.inf)

if DEBUG >= 1: print(f"Number of nodes searched: {COUNT}")

return { "move": move, "value": value }

def max\_value\_ab(state, player, alpha, beta):

"""player's best move."""

global DEBUG, COUNT

COUNT += 1

# return utility of state is a terminal state

v = utility(state, player)

if DEBUG >= 2: print(f"max: {state} [alpha,beta]=[{alpha},{beta}] v={v}")

if v is not None:

if DEBUG >= 2: print(f" found terminal state. backtracking.")

return v, None

v, move = -math.inf, None

# check all possible actions in the state, update alpha and return move with the largest value

for a in actions(state):

v2, a2 = min\_value\_ab(result(state, player, a), player, alpha, beta)

if DEBUG >= 2: print(f"max: {state} (backtracked) [alpha,beta]=[{alpha},{beta}] v={v2}")

if v2 > v:

v, move = v2, a

alpha = max(alpha, v)

if v >= beta:

if DEBUG >= 2: print(f" v>=beta ({v}>={beta}): pruning remaining subtree (actions). backtracking.")

return v, move

return v, move

def min\_value\_ab(state, player, alpha, beta):

"""opponent's best response."""

global DEBUG, COUNT

COUNT += 1

# return utility of state is a terminal state

v = utility(state, player)

if DEBUG >= 2: print(f"min: {state} [alpha,beta]=[{alpha},{beta}] v={v}")

if v is not None:

if DEBUG >= 2: print(f" found terminal state. backtacking.")

return v, None

v, move = +math.inf, None

# check all possible actions in the state, update beta and return move with the smallest value

for a in actions(state):

v2, a2 = max\_value\_ab(result(state, other(player), a), player, alpha, beta)

if DEBUG >= 2: print(f"min: {state} (backtracked) [alpha,beta]=[{alpha},{beta}] v={v2}")

if v2 < v:

v, move = v2, a

beta = min(beta, v)

if v <= alpha:

if DEBUG >= 2: print(f" v<=alpha ({v}<={alpha}): pruning remaining subtree (actions). backtracking.")

return v, move

return v, move

Giải thích code:

Các hàm max\_value\_ab và min\_value\_ab giờ đây có thêm 2 tham số:

· alpha: Giá trị tốt nhất (cao nhất) mà Max có thể đảm bảo tại thời điểm đó.

· beta: Giá trị tốt nhất (thấp nhất) mà Min có thể đảm bảo tại thời điểm đó.

Logic cắt tỉa trong max\_value\_ab (Max):

· Sau khi nhận được giá trị v từ min\_value\_ab, nó cập nhật alpha = max(alpha, v).

· Kiểm tra điều kiện: if v >= beta: return v, move.

· *Giải thích:* Nếu giá trị v của nhánh này còn tệ hơn (lớn hơn) cả giá trị beta mà Min đã chắc chắn có thể đạt được ở nhánh khác, Min sẽ không bao giờ đi nước này. Do đó, Max không cần tìm kiếm các nhánh con còn lại nữa.

Logic cắt tỉa trong min\_value\_ab (Min):

· Sau khi nhận được giá trị v từ max\_value\_ab, nó cập nhật beta = min(beta, v).

· Kiểm tra điều kiện: if v <= alpha: return v, move.

· *Giải thích:* Nếu giá trị v của nhánh này còn tệ hơn (nhỏ hơn) cả giá trị alpha mà Max đã chắc chắn có thể đạt được ở nhánh khác, Max sẽ không bao giờ đi nước này. Do đó, Min không cần tìm kiếm các nhánh con còn lại.

**Kết quả:** Nhanh hơn Minimax rất nhiều. Khi chạy với bàn cờ trống, nó chỉ phải duyệt 18,297 nút.

## Phương pháp 3: Sắp xếp nước đi (Move Ordering)

Ý tưởng: Đây là một kỹ thuật để tăng hiệu quả cho Alpha-Beta. Hiệu quả của Alpha-Beta phụ thuộc rất nhiều vào thứ tự duyệt các nước đi. Nếu duyệt các nước "tốt nhất" trước, khả năng cắt tỉa sẽ xảy ra sớm hơn.

Code:

def actions(board):

"""return possible actions as a vector ot indices"""

actions = np.where(np.array(board) == ' ')[0].tolist()

priority = [1,0,1,

0,2,0,

1,0,1]

priority = [priority[i] for i in actions]

actions =[a for \_,a in sorted(zip(priority,actions), reverse=True)]

return actions

Giải thích code:

· Hàm actions(board) được định nghĩa lại (ở ô [19]) để thực hiện việc sắp xếp.

· Nó tạo một danh sách priority (độ ưu tiên):

o Ô trung tâm (vị trí 4): ưu tiên 2

o Các góc (0, 2, 6, 8): ưu tiên 1

o Các cạnh: ưu tiên 0

· Hàm này trả về danh sách các nước đi (ô còn trống) đã được sắp xếp theo độ ưu tiên từ cao đến thấp.

**Kết quả:** Khi kết hợp với Alpha-Beta, số nút duyệt giảm từ 18,297 xuống chỉ còn 7,275 nút.

# Solving Tic-Tac-Toe with Heuristic Alpha-Beta Tree Search

Bài toán vẫn là giải quyết trò chơi Tic-Tac-Toe, một trò chơi đối kháng có tổng bằng không (win: +1, lose: -1, draw: 0).

Tuy nhiên, tệp này tập trung vào một phương pháp khác, được liệt kê là lựa chọn số 3: **Heuristic Alpha-Beta Tree Search** (Tìm kiếm cây Alpha-Beta sử dụng Heuristic).

Ý tưởng cốt lõi là:

· Không phải lúc nào cũng tìm kiếm toàn bộ cây trò chơi, vì việc này có thể quá tốn thời gian.

· Thay vào đó, chúng ta sẽ đặt một **giới hạn độ sâu (cutoff)**.

· Khi đạt đến độ sâu này (mà trò chơi vẫn chưa kết thúc), chúng ta cần một cách để "ước tính" xem bàn cờ đó tốt cho người chơi nào. Việc ước tính này được thực hiện bằng một **hàm đánh giá heuristic**

**Phương pháp làm:**

Phương pháp chính là sửa đổi thuật toán Alpha-Beta Pruning từ trước để tích hợp hai khái niệm mới:

1. Hàm đánh giá heuristic

Đây là một hàm do con người định nghĩa để gán một điểm số cho một trạng thái bàn cờ chưa kết thúc.

Code:

import numpy as np

def eval\_fun(state, player = 'x'):

"""heuristic for utility of state. Returns score for a node:

1. For terminal states it returns the utility.

2. For non-terminal states, it calculates a weighted linear function using features of the state.

The features we look at are 2 in a row/col/diagonal where the 3rd square is empty. We assume that

the more of these positions we have, the higher the chance of winning.

We need to be careful that the utility of the heuristic stays between [-1,1].

Note that the largest possible number of these positions is 2. I weigh the count by 0.4,

guaranteeing that is in the needed range.

Function Returns: heuristic value, terminal?"""

# terminal state?

u = utility(state, player)

if u is not None: return u, True

score = 0

board = np.array(state).reshape((3,3))

diagonals = np.array([[board[i][i] for i in range(len(board))],

[board[i][len(board)-i-1] for i in range(len(board))]])

for a\_board in [board, np.transpose(board), diagonals]:

for row in a\_board:

if sum(row == player) == 2 and any(row ==' '): score += .4

if sum(row == other(player)) == 2 and any(row ==' '): score -= .4

return score, False

Giải thích code eval\_fun(state, player)

· Đầu tiên, nó gọi utility(state, player). Nếu đây là trạng thái kết thúc (thắng/thua/hòa), nó trả về giá trị thực (ví dụ 1) và True (nghĩa là terminal).

· Nếu chưa kết thúc, nó sẽ tính score:

· Nó duyệt qua tất cả các hàng, cột, và 2 đường chéo.

· Với mỗi hàng/cột/đường chéo đó, nó kiểm tra:

o Nếu player (người chơi) có 2 quân cờ và 1 ô trống: score += 0.4. Đây là một "cơ hội" thắng.

o Nếu other(player) (đối thủ) có 2 quân cờ và 1 ô trống: score -= 0.4. Đây là một "mối đe dọa".

· Cuối cùng, nó trả về score (ví dụ 0.8, 0.0, -0.4) và False (nghĩa là chưa terminal).

Lưu ý: Tác giả chọn 0.4 để đảm bảo điểm heuristic luôn nhỏ hơn điểm thắng/thua thực sự (là 1 và -1).

2. Tìm kiếm có giới hạn độ sâu (Cutoff Search)

**Phương pháp:** Thuật toán Alpha-Beta sẽ được truyền thêm một tham số cutoff (ví dụ: tìm kiếm sâu 2 nước, 4 nước, v.v.). Khi đạt đến độ sâu này, thuật toán sẽ dừng tìm kiếm và gọi hàm eval\_fun để lấy giá trị ước tính.

Code:

import math

# global variables

DEBUG = 1 # 1 ... count nodes, 2 ... debug each node

COUNT = 0

def alpha\_beta\_search(board, cutoff = None, player = 'x'):

"""start the search. cutoff = None is minimax search with alpha-beta pruning."""

global DEBUG, COUNT

COUNT = 0

value, move = max\_value\_ab(board, player, -math.inf, +math.inf, 0, cutoff)

if DEBUG >= 1: print(f"Number of nodes searched (cutoff = {cutoff}): {COUNT}")

return {"move": move, "value": value}

def max\_value\_ab(state, player, alpha, beta, depth, cutoff):

"""player's best move."""

global DEBUG, COUNT

COUNT += 1

# cut off and terminal test

v, terminal = eval\_fun(state, player)

if((cutoff is not None and depth >= cutoff) or terminal):

if(terminal):

alpha, beta = v, v

if DEBUG >= 2: print(f"stopped at {depth}: {state} term: {terminal} eval: {v} [{alpha}, {beta}]" )

return v, None

v, move = -math.inf, None

# check all possible actions in the state, update alpha and return move with the largest value

for a in actions(state):

v2, a2 = min\_value\_ab(result(state, player, a), player, alpha, beta, depth + 1, cutoff)

if v2 > v:

v, move = v2, a

alpha = max(alpha, v)

if v >= beta: return v, move

return v, move

def min\_value\_ab(state, player, alpha, beta, depth, cutoff):

"""opponent's best response."""

global DEBUG, COUNT

COUNT += 1

# cut off and terminal test

v, terminal = eval\_fun(state, player)

if((cutoff is not None and depth >= cutoff) or terminal):

if(terminal):

alpha, beta = v, v

if DEBUG >= 2: print(f"stopped at {depth}: {state} term: {terminal} eval: {v} [{alpha}, {beta}]" )

return v, None

v, move = +math.inf, None

# check all possible actions in the state, update beta and return move with the smallest value

for a in actions(state):

v2, a2 = max\_value\_ab(result(state, other(player), a), player, alpha, beta, depth + 1, cutoff)

if v2 < v:

v, move = v2, a

beta = min(beta, v)

if v <= alpha: return v, move

return v, move

· Hàm alpha\_beta\_search giờ đây nhận thêm tham số cutoff.

· Các hàm đệ quy max\_value\_ab và min\_value\_ab có thêm 2 tham số: depth (độ sâu hiện tại) và cutoff (giới hạn).

· Thay đổi quan trọng nhất nằm ở ngay đầu các hàm max\_value\_ab và min\_value\_ab:

# cut off and terminal test

v, terminal = eval\_fun(state, player)

if((cutoff is not None and depth >= cutoff) or terminal):

if(terminal):

alpha, beta = v, v

if DEBUG >= 2: print(f"stopped at {depth}: {state} term: {terminal} eval: {v} [{alpha}, {beta}]" )

return v, None

Giải thích:

· Trước khi làm bất cứ điều gì, nó gọi eval\_fun để lấy giá trị v và trạng thái terminal.

· Nó kiểm tra if:

o terminal: Trò chơi đã kết thúc ở đây?

o (cutoff is not None and depth >= cutoff): Chúng ta đã đạt đến độ sâu tìm kiếm giới hạn chưa?

· Nếu một trong hai điều kiện là đúng, nó sẽ ngừng tìm kiếm sâu hơn và trả về v (giá trị thực hoặc giá trị heuristic).

· Nếu không, phần còn lại của hàm (vòng lặp for a in actions(state) và logic cắt tỉa alpha-beta) vẫn chạy như bình thường.

3. Thử nghiệm

Notebook chạy thử nghiệm với các độ sâu cutoff = 2, cutoff = 4 và cutoff = None (tìm kiếm hoàn hảo).

Kết quả:

Khi cutoff = 2 (tìm kiếm rất nông), AI đôi khi chọn nước đi không tối ưu vì heuristic bị "đánh lừa".

Khi cutoff = 4, AI chơi tốt hơn nhiều. Khi đấu heuristic4\_player (cutoff=4) với alpha\_beta\_player (hoàn hảo), kết quả luôn là hòa, cho thấy heuristic ở độ sâu 4 là đủ tốt để chơi tối ưu trong Tic-Tac-Toe.

Lợi ích lớn nhất là tốc độ:

· Tìm kiếm hoàn hảo (cutoff=None) từ bàn cờ trống: **18,297 nút**.

· Tìm kiếm với cutoff=4: **541 nút**.

· Tìm kiếm với cutoff=2: **26 nút**.

**Kết luận:** Tìm kiếm heuristic nhanh hơn *rất nhiều* (duyệt ít nút hơn) trong khi vẫn có thể đưa ra quyết định tối ưu (nếu heuristic và độ sâu đủ tốt)

# Solving Tic-Tac-Toe with Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) là một thuật toán tìm kiếm dùng nhiều trong AI chơi game (như Tic-Tac-Toe, Connect4...)  
Mục tiêu của MCTS là chọn nước đi tốt nhất bằng cách mô phỏng ngẫu nhiên nhiều ván chơi từ trạng thái hiện tại, rồi thống kê kết quả để ra quyết định.

Thay vì duyệt toàn bộ cây trạng thái (như Minimax), MCTS chỉ duyệt sâu hơn ở những nhánh có tiềm năng, nên hiệu quả hơn với không gian trạng thái lớn.

**Các hàm phụ được sử dụng trong bài toán trên**

def empty\_board():

return [' '] \* 9

board = empty\_board()

display(board)

Dùng để tạo board rỗng

def show\_board(board):

"""display the board"""

b = np.array(board).reshape((3,3))

print(b)

board = empty\_board()

show\_board(board)

print()

print("Add some x's")

board[0] = 'x'; board[3] = 'x'; board[6] = 'x';

show\_board(board)

Kết quả sau khi thực hiện

[[' ' ' ' ' ']

[' ' ' ' ' ']

[' ' ' ' ' ']]

Add some x's

[['x' ' ' ' ']

['x' ' ' ' ']

['x' ' ' ' ']]

def check\_win(board):

"""check the board and return one of x, o, d (draw), or n (for next move)"""

board = np.array(board).reshape((3,3))

diagonals = np.array([[board[i][i] for i in range(len(board))],

[board[i][len(board)-i-1] for i in range(len(board))]])

for a\_board in [board, np.transpose(board), diagonals]:

for row in a\_board:

if len(set(row)) == 1 and row[0] != ' ':

return row[0]

# check for draw

if(np.sum(board == ' ') < 1):

return 'd'

return 'n'

Hàm check\_win(board): một hàm kiểm tra trạng thái thắng/thua/hòa trong trò chơi Tic-Tac-Toe (cờ ca-rô 3x3).

Hàm này kiểm tra bàn cờ hiện tại (board) và trả về:

* 'x' nếu người chơi X thắng
* 'o' nếu người chơi O thắng
* 'd' nếu hòa (draw)
* 'n' nếu chưa kết thúc (vẫn còn nước đi tiếp theo)

def get\_actions(board):

"""return possible actions as a vector ot indices"""

return np.where(np.array(board) == ' ')[0].tolist()

# randomize the action order

#actions = np.where(np.array(board) == ' ')[0]

#np.random.shuffle(actions)

#return actions.tolist()

show\_board(board)

get\_actions(board)

**Hàm get\_actions(board)** này là một hàm lấy danh sách các nước đi hợp lệ trong trò chơi Tic-Tac-Toe, tức là các ô còn trống mà người chơi có thể đánh được.

Các hàm chính được sử dụng trong thuật toán monte carlo

def playout(state, action, player = 'x'):

"""Perfrom a random playout starting with the given action on the fiven board

and return the utility of the finished game."""

state = result(state, player, action)

current\_player = other(player)

while(True):

# reached terminal state?

u = utility(state, player)

if u is not None:

return u

# we use a random playout policy

a = np.random.choice(get\_actions(state))

state = result(state, current\_player, a)

#print(state)

# switch between players

current\_player = other(current\_player)

board = empty\_board()

display([ playout(board, 0) for i in range(20) ])

Hàm playout có chức năng mô phỏng một ván chơi ngẫu nhiên trong trò chơi hai người (như cờ caro, cờ tic-tac-toe, v.v.) bắt đầu từ một hành động cụ thể, và trả về giá trị tiện ích (utility) của kết quả cuối cùng

**Tham số đầu vào:**

* state: trạng thái hiện tại của bàn cờ.
* action: nước đi đầu tiên mà người chơi player sẽ thực hiện.
* player: người chơi bắt đầu (mặc định là 'x').

**Kết thúc khi trò chơi đến trạng thái cuối:**

* Trả về giá trị tiện ích (utility) của người chơi ban đầu (player) khi trò chơi kết thúc.

class UCT\_Node:

def \_\_init\_\_(self, state, parent):

self.state = state.copy() # là **bàn cờ hiện tại**, ví dụ ['x',' ','o',' ','x',' ',' ',' ','o']

self.u = 0 # tổng điểm nhận được qua các lần mô phỏng.

(ví dụ: thắng = +1, thua = -1, hòa = 0)

self.n = 0 # số lần node này được **ghé thăm** trong quá trình MCTS.

self.parent = parent # node cha (để biết node này được sinh ra từ đâu).

self.children = {} #

def \_\_str\_\_(self):

return f"UCT Node for state {self.state} with {len(self.children.keys())} children: ({self.u}/{self.n})"

def UCB1(self):

if(self.n < 1): return +math.inf

if(self.parent is None): return +math.inf # CHECK!!!

self.u/self.n + C \* math.sqrt(math.log(self.parent.n)/self.n)

n = UCT\_Node(empty\_board(), None)

n.n = 10

n.u = 5

print(n)

print(n.UCB1())

## Phương thức UCB1()

Đây là công thức **Upper Confidence Bound** dùng để **chọn nút con tốt nhất** trong MCTS — nó cân bằng giữa:

* **Exploration (khám phá):** thử những hành động ít được chọn để có thể tìm ra lựa chọn tốt hơn.
* **Exploitation (khai thác):** ưu tiên hành động có kết quả trung bình cao nhất.

Công thức:



Trong đó:

* u = tổng điểm (utility)
* n = số lần nút này được thăm
* N\_parent = số lần nút cha được thăm
* C = hệ số điều chỉnh (thường là √2)

Output khi chạy đoạn code trên :

UCT Node for state [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '] with 0 children: (5/10)

Inf

Thử nghiệm 1:

Code :

# x is about to win (play 8)

board = empty\_board()

board[0] = 'x'

board[1] = 'o'

board[3] = 'o'

board[4] = 'x'

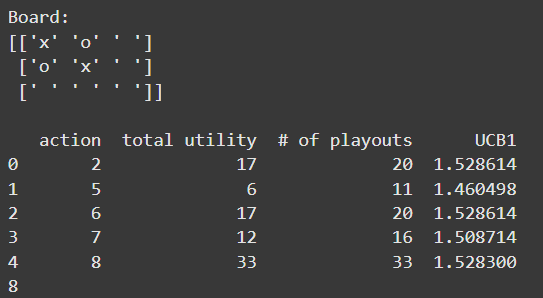
print("Board:")

show\_board(board)

print()

display(UCT\_depth1(board ))

Kết quả:



## Các cột trong bảng

|  |  |  |
| --- | --- | --- |
| **Cột** | **Ý nghĩa** | **Diễn giải cụ thể** |
| **action** | Hành động (chỉ số cột, nước đi, hoặc lựa chọn kế tiếp) | Ví dụ: “đặt quân ở cột 0”, “đi ô 2”, v.v. |
| **total utility** | Tổng điểm thu được từ tất cả các mô phỏng (playouts) đi qua hành động này | Càng cao nghĩa là hành động này thường dẫn đến kết quả tốt hơn |
| **# of playouts** | Số lần hành động này được chọn (nút con được thăm) trong quá trình MCTS | Thể hiện mức độ “khám phá” hành động đó |
| **UCB1** | Giá trị UCB1 được tính theo công thức | Dùng để chọn hành động tiếp theo — hành động có **UCB1 lớn nhất** sẽ được chọn |

Giải thích

Các hành động 0, 2, và 4 có UCB1 gần như bằng nhau (~1.528).  
 → MCTS coi chúng đều tiềm năng, vì chúng có tỷ lệ thắng cao và được thăm đủ nhiều.

Hành động 1 có UCB1 thấp nhất (1.46), vì hiệu suất (utility trung bình) thấp và chưa đủ “thuyết phục”.

Thử nghiệm 2:

#### x can draw if it chooses 7.

board = empty\_board()

board[0] = 'x'

board[1] = 'o'

board[2] = 'x'

board[4] = 'o'

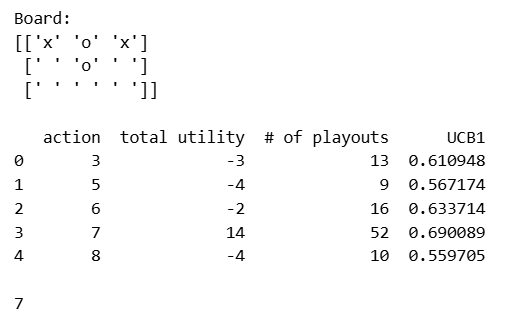
print("Board:")

show\_board(board)

print()

display(UCT\_depth1(board))

Kết quả khi chạy đoạn code trên:



Giá trị ở ô thứ 7 đạt mức cao nhất vậy nên agent sẽ chọn action thứ 7.

Ô này có nhiệm vụ ngăn ô ‘O’ chiến thắng.

Và nếu đánh ô này thì ‘X’ có thể hòa

Thí nghiệm 3:

Agent sử dụng Thuật toán monte carlo đấu với agent random

def uct10\_player(board, player = 'x'):

action = UCT\_depth1(board, N = 10, player = player)

return action

def uct100\_player(board, player = 'x'):

action = UCT\_depth1(board, N = 100, player = player)

return action

DEBUG = 1

print("UCT vs. random:")

display(play(uct10\_player, random\_player, N = 1))

Giải thích

Agent thứ nhất: uct10\_player

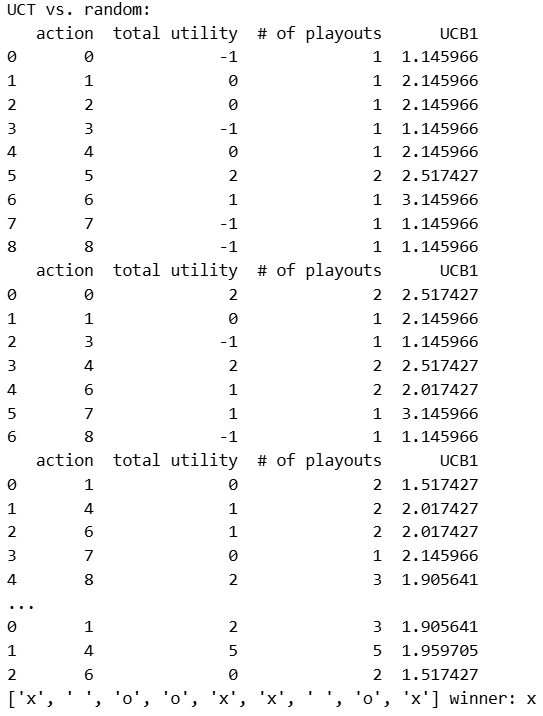
* Đây là AI agent sử dụng thuật toán Monte Carlo Tree Search (MCTS), cụ thể là biến thể UCT (Upper Confidence bound applied to Trees).
* Ở mỗi lượt chơi, nó:
* Lấy trạng thái bàn cờ hiện tại (board).
* Mô phỏng (tức là thử chơi ngẫu nhiên) 10 lần (N=10).
* Tính toán xác suất thắng trung bình cho từng nước đi.
* Chọn nước đi có giá trị UCB (Upper Confidence Bound) cao nhất — tức là tốt nhất giữa việc khám phá và khai thác.

Vì vậy, uct10\_player là **AI thông minh**, biết học từ kết quả mô phỏng ngẫu nhiên

Agent thứ hai: random\_player

* Đây là một **AI rất đơn giản**, chỉ **chọn nước đi ngẫu nhiên** mỗi lượt.
* Nó không học, không đánh giá, chỉ chơi ngẫu nhiên

Kết quả khi chạy đoạn code



Người chơi ‘O’ là random player như vậy Random player không thể chiến thắng player sử dụng Monte Carlo Search ‘X’

# Solving Tic-Tac-Toe with Monte Carlo Tree Search Restricted

Giới thiệu bài toán:

Bài toán dựa trên ý tưởng của trò chơi tic-tac-toe trong đó độ sâu của cây bị giới hạn lại với độ sâu có kích thước là 1

Cell 1:

import numpy as np

def playout(state, action, player = 'x'):

"""Perform a random playout starting with the given action on the given board

and return the utility of the finished game."""

state = result(state, player, action)

current\_player = other(player)

while(True):

# reached terminal state?

u = utility(state, player)

if u is not None:

return u

# we use a random playout policy

a = np.random.choice(actions(state))

state = result(state, current\_player, a)

#print(state)

# switch between players

current\_player = other(current\_player)

board = empty\_board()

print(playout(board, 0))

print(playout(board, 0))

print(playout(board, 0))

Result:

-1

0

1

Giải thích:

Đây là **một “trò chơi mô phỏng ngẫu nhiên” (random playout)**.

Trong đó các thành phần bao gồm:

* state: bàn cờ hiện tại,
* action: nước đi ban đầu của người chơi player,
* player: người bắt đầu.

Sau đó:

1. Thực hiện hành động đó.
2. Cho hai bên chơi ngẫu nhiên cho đến khi kết thúc.
3. Trả về kết quả thắng/thua/hòa dưới dạng **utility**.

import pandas as pd

import math

DEBUG = 1

def UCT\_depth1(board, N = 100, player = 'x'):

"""Upper Confidence bound applied to Trees for limited tree depth of 1.

Simulation budget is N playouts."""

global DEBUG

C = math.sqrt(2) # tradeoff constant

# the tree is 1 action deep

acts = actions(board)

u = [0] \* len(acts) # total utility through actions

n = [0] \* len(acts) # number of playouts through actions

n\_parent = 0 # total playouts so far (i.e., number of playouts through parent)

# make sure we try each action once

UCB1 = [+math.inf] \* len(acts)

for i in range(N):

# Select

action\_id = UCB1.index(max(UCB1))

# Expand

# UTC would expand the tree. We keep the tree at depth 1, essentially performing

# Pure Monte Carlo search with an added UCB1 selection policy.

# Simulate

p = playout(board, acts[action\_id], player = player)

# Back-Propagate (i.e., update counts and UCB1)

u[action\_id] += p

n[action\_id] += 1

n\_parent += 1

for action\_id in range(len(acts)):

if n[action\_id] > 0:

UCB1[action\_id] = u[action\_id] / n[action\_id] + C \* math.sqrt(math.log(n\_parent) / n[action\_id])

# return action with largest number of playouts

action = acts[n.index(max(n))]

if DEBUG >= 1:

print(pd.DataFrame({'action':acts,

'total utility':u,

'# of playouts':n,

'UCB1':UCB1}))

print()

print(f"Best action: {action}")

return action

board = empty\_board()

display(board)

%timeit -n 1 -r 1 UCT\_depth1(board, N = 1000)

### Result:

[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']

action total utility # of playouts UCB1

0 0 38 110 0.699849

1 1 28 91 0.697332

2 2 26 87 0.697346

3 3 7 46 0.700204

4 4 238 448 0.706858

5 5 -5 14 0.636246

6 6 25 86 0.691504

7 7 1 33 0.677336

8 8 24 85 0.685510

Best action: 4

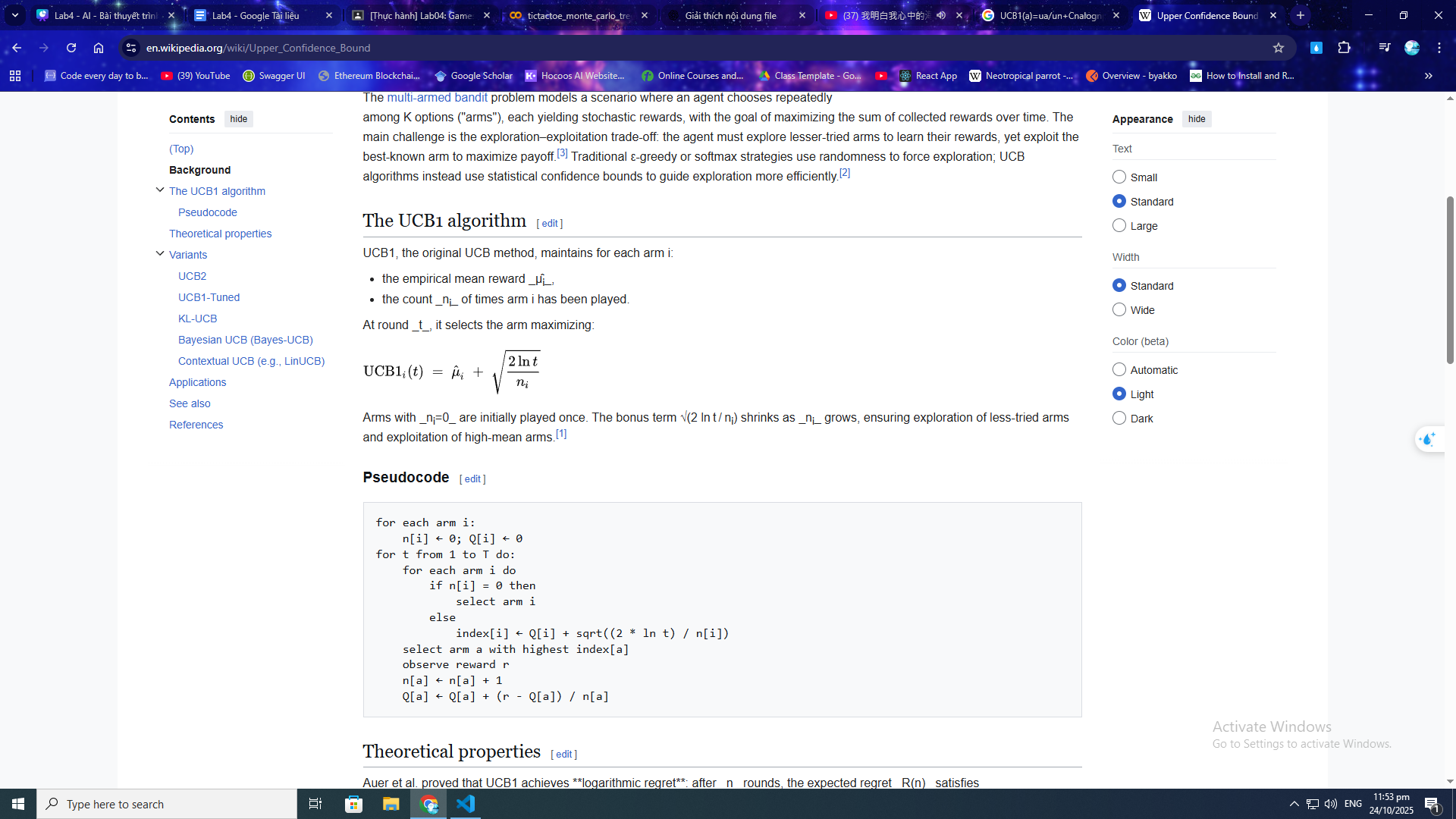
250 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

Giải thích chi tiết:

Đây là Monte Carlo Tree Search (MCTS) nhưng chỉ với độ sâu = 1, nghĩa là chỉ xét các hành động từ trạng thái hiện tại, không mở rộng sâu thêm.

Các bước:

1. **Khởi tạo:**
   * acts: danh sách các nước đi có thể.
   * u, n: lưu tổng điểm và số lần thử cho mỗi nước đi.
   * UCB1: công thức chọn nước đi cân bằng giữa *khám phá (exploration)* và *khai thác (exploitation)*.
2. **Lặp N lần (budget N):**
   * **Chọn (Select):** chọn nước đi có UCB1 cao nhất.
   * **Mô phỏng (Simulate):** chạy playout từ nước đi đó.
   * **Cập nhật (Backpropagate):** cập nhật u, n, và tính lại UCB1.
3. **Chọn nước có nhiều mô phỏng nhất (best action).**



MCTS cần **cân bằng giữa exploration và exploitation**:

* *Exploration:* Thử các nhánh chưa được thăm nhiều để tìm chiến lược mới.
* *Exploitation:* Ưu tiên nhánh đã có giá trị trung bình thắng cao.

# Pure Monte Carlo Search vs. Random:

def ucb1\_10\_player(board, player = 'x'):

action = UCT\_depth1(board, N = 10, player = player)

return action

def ucb1\_100\_player(board, player = 'x'):

action = UCT\_depth1(board, N = 100, player = player)

return action

Hai “bot” chơi:

* Một bot chạy 10 mô phỏng/lượt (N=10)
* Một bot chạy 100 mô phỏng/lượt (N=100)