# **Definitions**

**Defining the Game: Tic-Tac-Toe**

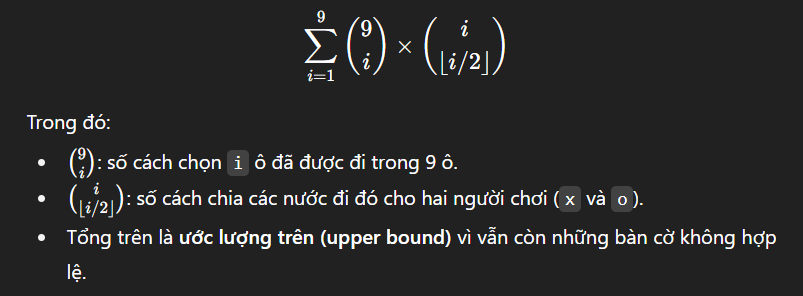
**The Search Problem**

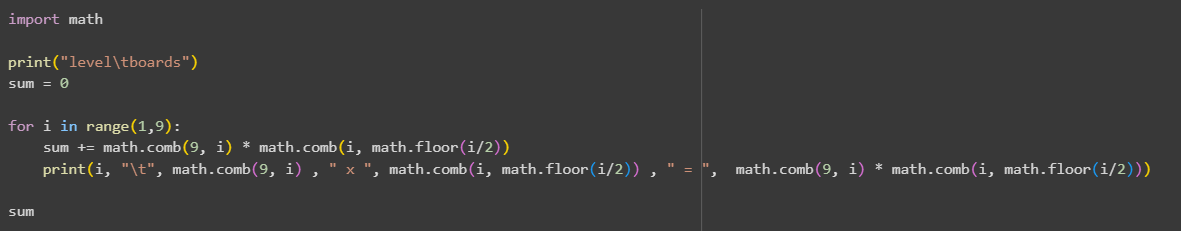
* Trạng thái ban đầu (Initial State): Bàn cờ trống, người chơi X đi trước.
* Hành động (Actions): Đặt ký hiệu của mình (X hoặc O) vào ô trống bất kỳ.
* Hàm chuyển trạng thái (Transition function / Result): Cập nhật bàn cờ sau khi người chơi thực hiện nước đi, sau đó đến lượt đối thủ (môi trường phi xác định vì đối thủ có thể đi bất kỳ cách nào).
* Trạng thái đích (Goal / Terminal): Trò chơi kết thúc — có người thắng (3 ký hiệu liên tiếp hàng, cột hoặc chéo) hoặc hòa.
* Hàm lợi ích (Utility): Đánh giá kết quả cuối cùng (thắng/thua/hòa) cho người chơi Max (thường là X).
* Chi phí đường đi (Path cost): Số lượt đi, nhưng trong trò này ta bỏ qua vì dùng DFS (Depth-First Search) — không tìm lời giải tối ưu về số nước đi.

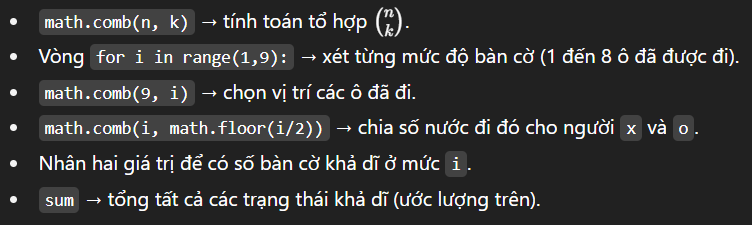
## Problem Size

Ước lượng kích thước không gian trạng thái (state space) của trò chơi Tic-Tac-Toe.

* Mỗi trạng thái là một bàn cờ có 9 ô, mỗi ô có thể là trống, x, hoặc o → tối đa
* 3\*\*9=19683 trạng thái.
* Tuy nhiên, nhiều trạng thái là không hợp lệ (ví dụ: cả hai người cùng thắng).
* Để ước lượng chặt chẽ hơn, ta tính:







## Space and Time Complexity Using the Search Tree Estimates

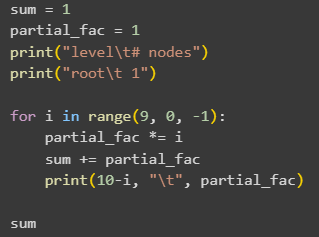
Phân tích độ phức tạp không gian (space) và thời gian (time) của trò chơi Tic-Tac-Toe khi biểu diễn dưới dạng cây tìm kiếm (search tree) thay vì không gian trạng thái (state space).

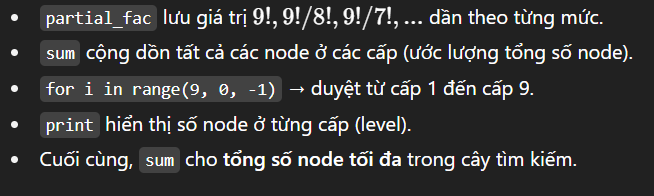
* **Độ sâu tối đa (m):** 9 (vì tối đa 9 nước đi).
* **Hệ số phân nhánh tối đa (b):** 9 (nước đầu tiên có 9 lựa chọn).
* **DFS (Depth-First Search):**
  + **Space complexity:** O(bm)=9×9=81 (do chỉ lưu nhánh hiện tại + frontier).
  + **Time complexity:** O()==387,420,489 (số node có thể mở rộng).

Tuy nhiên, hệ số phân nhánh **giảm dần** sau mỗi lượt đi:

* Level 1: 9 cách → 9!/8!=9
* Level 2: 9×8=9!/7!
* Level 3: 9×8×7=9!/6!
* Level 9: 9!=362,880

Tổng số node (ước lượng trên) trong cây tìm kiếm là:





**Implementation**

1. Actions(s) – xác định **các nước đi hợp lệ** trong trạng thái s  
   → trả về danh sách các ô trống có thể đánh (0–8).
2. Result(s, a) – **mô hình chuyển trạng thái**  
   → trả về **bàn cờ mới** sau khi người chơi thực hiện hành động a.
3. Terminal(s) – kiểm tra **trạng thái kết thúc**  
   → True nếu trò chơi kết thúc (có người thắng hoặc hòa).
4. Utility(s) – **giá trị lợi ích (điểm)** cho người chơi Max (x)  
   → +1 nếu x thắng, -1 nếu o thắng, 0 nếu hòa.

* Mỗi bàn cờ là một **list gồm 9 phần tử**:

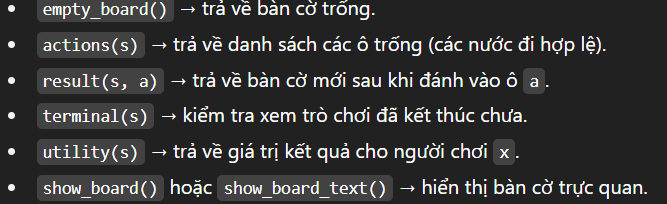
' ' : ô trống

'x' : người chơi X

'o' : người chơi O

* Các hành động (actions) là **chỉ số ô** (0–8).

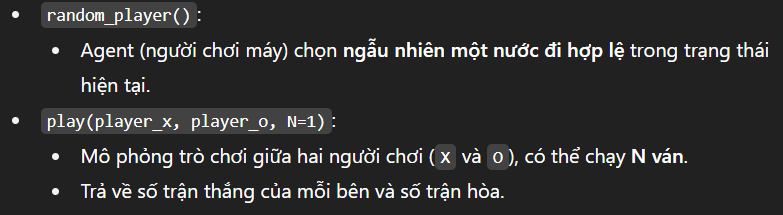




## Experiments

Mục tiêu:  
→ Dùng người chơi chọn nước đi ngẫu nhiên để làm **mốc so sánh (baseline)** cho các thuật toán thông minh hơn sau này.





**%time display(play(random\_player, random\_player, N=100))**

* Chạy 100 ván giữa **hai người chơi ngẫu nhiên**.
* Kết quả ví dụ:

**{'x': 58, 'o': 33, 'd': 9}**

→ x thắng 58 ván, o thắng 33 ván, hòa 9 ván.

# Heuristic alpha beta tree search

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

### Introduction

1. Trò chơi đối kháng (Zero-sum game):

* Max (người chơi X) cố gắng **tối đa hóa điểm số**.
* Min (người chơi O) cố gắng **tối thiểu hóa điểm số**.
* Kết quả:
  + X thắng → giá trị +1
  + O thắng → giá trị -1
  + Hòa → giá trị 0

1. Các loại tìm kiếm trong game:

**Minimax search:** duyệt toàn bộ cây trò chơi để chọn nước đi tối ưu.

* **Alpha-Beta pruning:** loại bỏ (cắt tỉa) các nhánh không cần thiết để tăng tốc Minimax.
* **Heuristic Alpha-Beta Tree Search:** cắt bớt độ sâu tìm kiếm và **dùng hàm heuristic** để ước lượng giá trị trạng thái khi chưa tới nút cuối.
* **Monte Carlo Tree Search:** mô phỏng nhiều ván chơi ngẫu nhiên để ước lượng giá trị nước đi.

**Cài đặt Heuristic Alpha-Beta Tree Search cho Tic-Tac-Toe** — thuật toán sẽ tìm trong cây trạng thái các nước đi khả dĩ, dùng heuristic để đánh giá khi không thể duyệt hết, và trả về **nước đi tối ưu tiếp theo** (không phải toàn bộ kế hoạch chơi).

### Heuristic Evaluation Function

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

* state: bàn cờ hiện tại (dạng danh sách 9 phần tử, gồm 'x', 'o', hoặc ' ').
* player: người đang cần đánh giá (thường là 'x').

**u = utility(state, player)**

**if u is not None:**

**return u, True**

**-** Gọi hàm utility() để kiểm tra:

**+** 'x' thắng → u = +1

**+** 'o' thắng → u = -1

**+** hòa → u = 0

**-** Nếu ván đã kết thúc → **trả ngay giá trị thật**, không cần heuristic nữa.  
→ return (giá trị, True)

**score = 0**

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

Chuyển danh sách 9 ô thành ma trận 3x3 để dễ xử lý hàng, cột, chéo.

**diagonals = np.array([**

**[board[i][i] for i in range(len(board))],**

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

**])**

* Lấy 2 đường chéo chính và phụ.
* Sau đó xét trên:
  + các **hàng** (board),
  + các **cột** (np.transpose(board)),
  + các **đường chéo** (diagonals).

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

**for row in a\_board:**

**if sum(row == player) == 2 and any(row == ' '):**

**score += 0.4**

**if sum(row == other(player)) == 2 and any(row == ' '):**

**score -= 0.4**

* Nếu người chơi player có **2 quân trên cùng hàng/cột/chéo và 1 ô trống**, → **gần thắng** → tăng điểm +0.4.
* Nếu đối thủ cũng có “2 trong hàng” → nguy cơ thua → trừ điểm -0.4.
* Tối đa chỉ có **2 vị trí kiểu này** trên bàn, nên giá trị score luôn nằm trong [-0.8, +0.8] → đảm bảo trong phạm vi [-1, 1].

**return score, False**

- Trả về (giá trị heuristic, False)

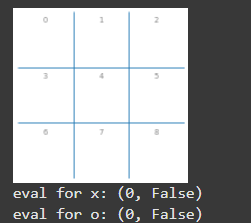
+ False vì đây **không phải trạng thái kết thúc**.

**board = empty\_board()**

**show\_board(board)**

**print(f"eval for x: {eval\_fun(board)}")**

**print(f"eval for o: {eval\_fun(board, 'o')}")**



* Không ai thắng, không có hàng nào “2 cùng loại + 1 trống”.
* → utility = None → dùng heuristic.
* score = 0.

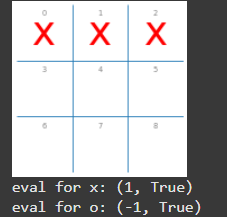
⟹ Cả hai người đều có “điểm tiềm năng” bằng 0 (trạng thái trung lập).

**board = empty\_board()**

**board[0] = 'x'**

**board[1] = 'x'**

**board[2] = 'x'**



* utility(state, 'x') = +1
* utility(state, 'o') = -1

⟹ Ván đã kết thúc → trả về **giá trị thật (utility)**, không dùng heuristic.

**board = empty\_board()**

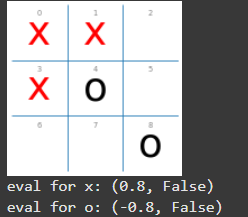
**board[0] = 'x'**

**board[1] = 'x'**

**board[3] = 'x'**

**board[4] = 'o'**

**board[8] = 'o'**



* Không ai thắng (utility = None)
* Heuristic tính:
  + Hàng 0: X có 2 + 1 trống → +0.4
  + Không hàng/chéo nào khác phù hợp.
  + O không có 2 + 1 trống.
* Tổng: score = +0.4 cho X  
  → score = -0.4 cho O (vì nguy cơ bị thua cao hơn)

⟹ X có lợi thế nhỏ (gần thắng 1 hàng).

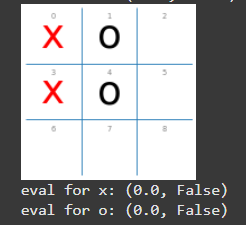
**board = empty\_board()**

**board[0] = 'x'**

**board[1] = 'o'**

**board[3] = 'x'**

**board[4] = 'o'**

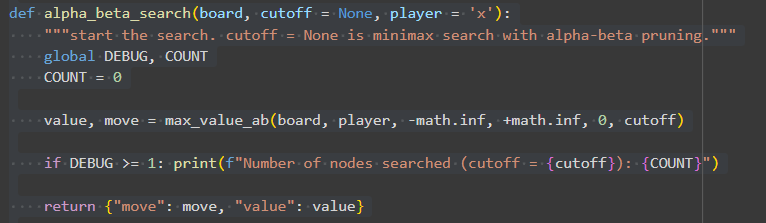


X: chưa có hàng/chéo nào 2 + 1 trống → 0 điểm.

O: cột giữa có 2 + 1 trống (ô (2,1) trống) → -0.4 cho X, +0.4 cho O.

⟹ O có lợi thế (có cơ hội tạo 3 hàng dọc giữa).

### Search with Cutoff



Nếu cutoff=None → thuật toán sẽ chạy **Minimax đầy đủ (duyệt toàn bộ cây)**.

Nếu cutoff có giá trị (ví dụ cutoff=3) → dừng tìm kiếm ở độ sâu đó, dùng eval\_fun() để **ước lượng giá trị nút lá**.

Gọi đệ quy max\_value\_ab() cho người chơi chính (Max – ví dụ 'x').

In số lượng **nút đã duyệt** (COUNT) để thấy hiệu quả của việc cắt tỉa.



**state**: bàn cờ hiện tại

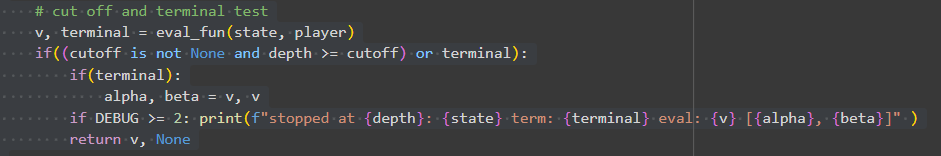
**player**: người chơi chính (ví dụ 'x')

**alpha**: giá trị tốt nhất hiện tại của Max (ban đầu -inf)

**beta**: giá trị tốt nhất hiện tại của Min (ban đầu +inf)

**depth**: độ sâu trong cây

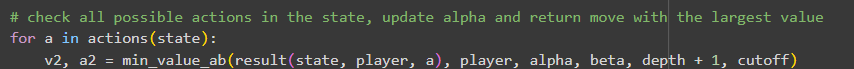
**cutoff**: giới hạn độ sâu



Nếu đạt đến cutoff hoặc **trạng thái kết thúc** (thắng/thua/hòa):  
→ **không mở rộng thêm**, trả về giá trị heuristic v.



Bắt đầu từ giá trị rất thấp vì Max đang muốn **tối đa hóa**.



Với mỗi hành động a:

result(state, player, a) → bàn cờ mới sau khi Max đi.

Gọi min\_value\_ab() để xem đối thủ (Min) phản ứng thế nào.

## 

Nếu nước đi này tốt hơn → cập nhật v và move.

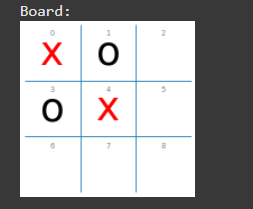
Nếu v >= beta → **cắt tỉa (prune)** vì Min sẽ không cho Max đạt giá trị cao hơn → dừng duyệt nhánh còn lại.



Tương tự nhưng đảo chiều logic

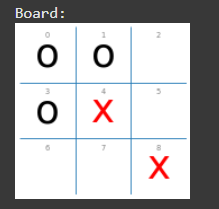
### Some Tests

**x is about to win (play 8)**



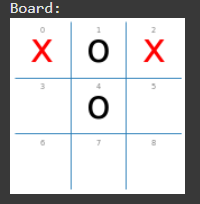
Nước tốt nhất cho X là **ô 8** (vì X sẽ có đường chéo [0,4,8] để thắng).

**o is about to win**



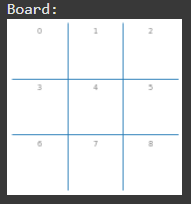
O đã chiếm 2 hàng đầu và 2 cột đầu — X **không thể thắng**, chỉ cố ngăn chặn.  
Nếu còn ô trống giúp O tạo hàng 3 → X cần chặn ngay.

**x can draw if it chooses 7**



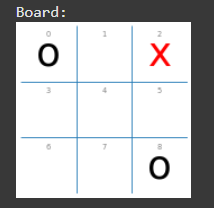
Nếu X chọn ô **7**, kết quả cuối cùng là **hòa**.  
Nếu chọn sai, O có thể thắng.

**Empty board: Only a draw an be guaranteed**



Trò Tic-Tac-Toe hoàn hảo luôn dẫn đến **hòa** nếu cả hai chơi tối ưu.

**A bad situation**



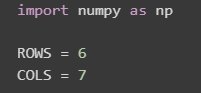
O kiểm soát hai góc → X ở thế khó, khó thắng được.

# Adversarial Search: Playing Connect 4

## Task 1: Defining the Search Problem

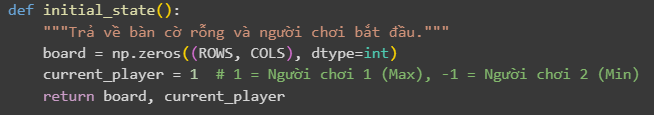
**Bài toán tìm kiếm đối kháng (adversarial search)** cho trò **Connect 4**, bao gồm các thành phần:

* **Trạng thái ban đầu (initial state)** – bàn cờ trống, người chơi 1 bắt đầu.
* **Tập hành động (actions)** – các cột còn trống để thả quân.
* **Hàm chuyển trạng thái (transition model)** – mô tả việc thả quân và đổi lượt chơi.
* **Trạng thái kết thúc (goal state)** – khi có người thắng hoặc bàn cờ đầy.
* **Hàm tiện ích (utility)** – xác định giá trị thắng (+1), thua (–1) hoặc hòa (0).



Xác định kích thước bàn cờ: **6 hàng × 7 cột** (chuẩn Connect 4).

Dùng thư viện numpy để dễ xử lý ma trận bàn cờ.



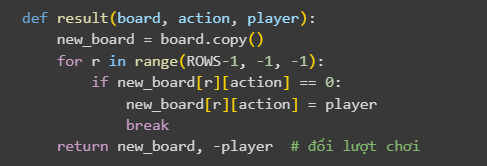
Tạo **bàn cờ rỗng** (mọi ô = 0).

Người chơi 1 (được ký hiệu là 1) bắt đầu.

Người chơi 2 được ký hiệu là -1.



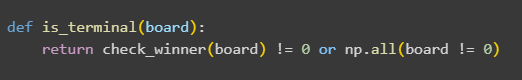
Trả về **danh sách các cột hợp lệ** để thả quân



**Tạo bàn cờ mới** sau khi người chơi thả quân vào cột action.

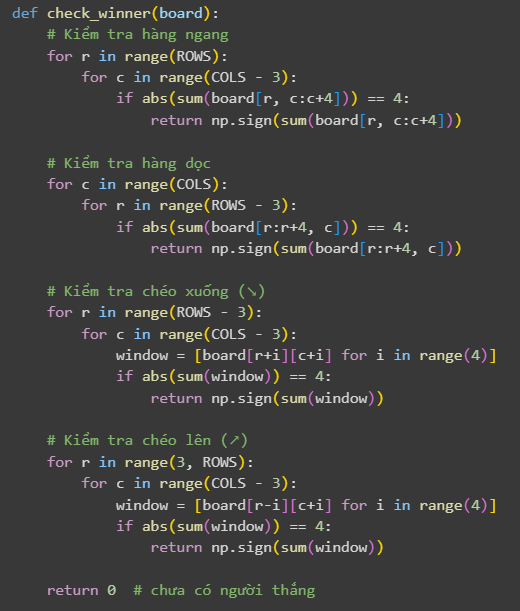
Quân rơi xuống vị trí **trống thấp nhất** trong cột.

Sau khi đi xong, **đổi lượt chơi** (-player).



Trả về True nếu:

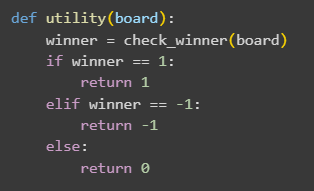
* Có người thắng, **hoặc**
* Bàn cờ **đã đầy** (không còn ô trống).



Hàm này duyệt **từng hàng, từng cột, và 2 đường chéo** để xem có 4 quân giống nhau liên tiếp không.

Nếu có:

* 1 → người chơi 1 thắng
* -1 → người chơi 2 thắng
* 0 → chưa ai thắng



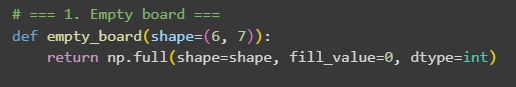
Xác định **giá trị tiện ích** của trạng thái kết thúc:

* +1 → người chơi 1 thắng
* -1 → người chơi 2 thắng
* 0 → hòa hoặc chưa có kết quả

## Task 2: Game Environment and Random Agent

Các bước chính:

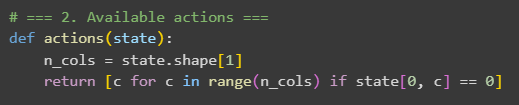
1. **Tạo bàn cờ** (mảng numpy).
2. **Xác định hành động hợp lệ** (các cột còn trống).
3. **Mô hình chuyển trạng thái** (thả quân vào cột).
4. **Kiểm tra người thắng** (4 quân liên tiếp).
5. **Kiểm tra trạng thái kết thúc** (thắng hoặc hòa).
6. **Hàm tiện ích (utility)** – xác định thắng, thua hoặc hòa.
7. **Random agent** – chọn ngẫu nhiên một cột hợp lệ để đi.



Tạo bàn cờ **rỗng** có kích thước tùy chọn (mặc định 6×7).

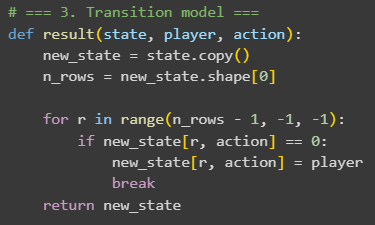
Các ô ban đầu có giá trị 0 (chưa có quân).

Dùng np.full() để tạo ma trận toàn số 0.



Trả về **danh sách các cột còn trống** (các ô đầu tiên của cột là 0).

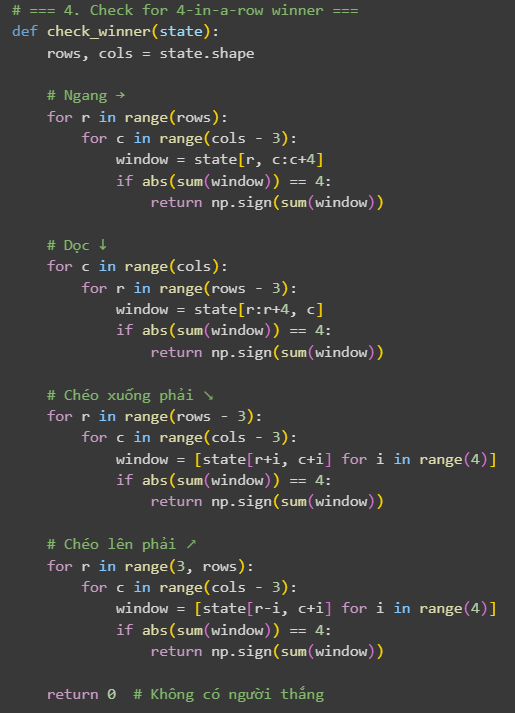
Nếu cột đã đầy, không thể chọn cột đó.



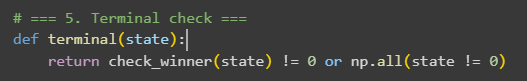
Tạo **bản sao bàn cờ mới**.

Quân được **thả xuống vị trí trống thấp nhất** trong cột được chọn (action).

player là người chơi hiện tại (1 hoặc -1)



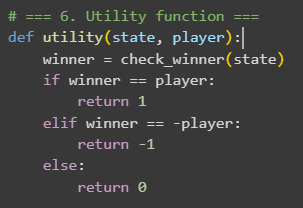
* Kiểm tra có ai **nối được 4 quân liên tiếp** theo:
  + Hàng ngang (→)
  + Cột dọc (↓)
  + Chéo xuống phải (↘)
  + Chéo lên phải (↗)
* Nếu tìm thấy 4 quân giống nhau:
  + Trả về 1 → người chơi 1 thắng
  + Trả về -1 → người chơi 2 thắng
  + Không ai thắng → 0.



Trả về True nếu:

Có người thắng, **hoặc**

Tất cả ô đều được đi (hòa).

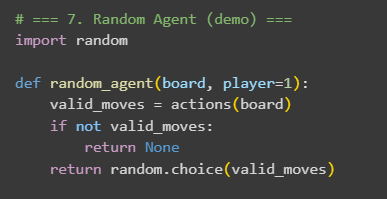


Đánh giá giá trị kết thúc:

+1 nếu người chơi hiện tại thắng

–1 nếu thua

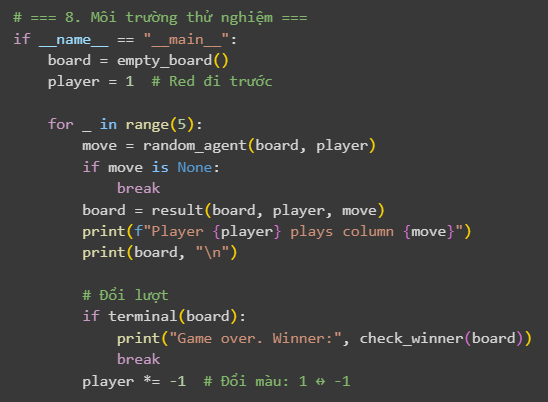
0 nếu hòa hoặc chưa có kết quả.



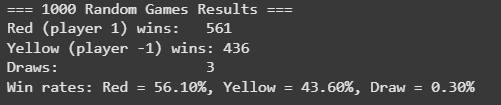
Chọn **ngẫu nhiên một cột hợp lệ** để đi.

Nếu không còn nước đi → trả về None.

Dùng để **kiểm thử** hoặc làm **đối thủ mặc định** cho các agent thông minh hơn.



* Tạo bàn cờ rỗng.
* Hai người chơi **luân phiên** thực hiện nước đi trong 5 lượt (có thể kết thúc sớm nếu có người thắng).
* Sau mỗi nước đi:
  + In ra người chơi và cột đã chọn.
  + Cập nhật lại bàn cờ (result()).
  + Kiểm tra trạng thái kết thúc (terminal()).
  + Đổi lượt (player \*= -1).



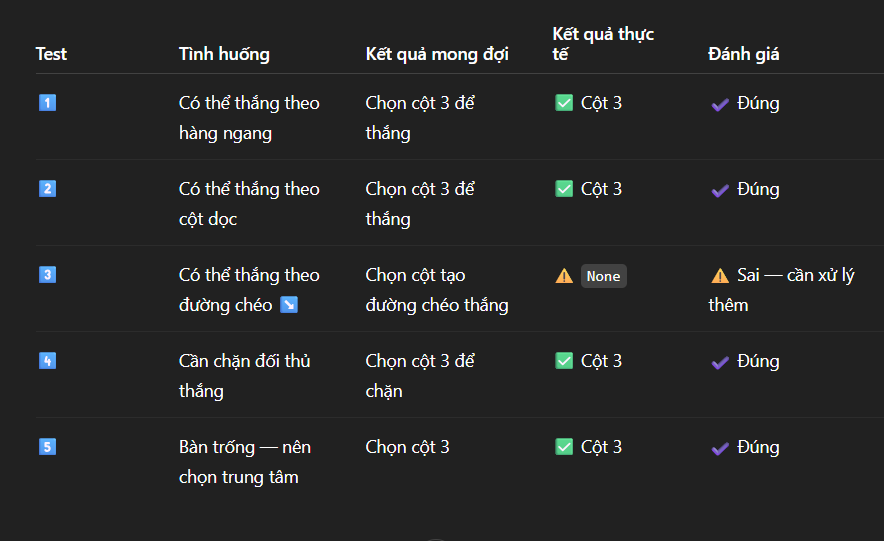
Điều này **hoàn toàn bình thường** và phù hợp với lý thuyết:

* Trong **Connect Four**, **người chơi đi trước có lợi thế tự nhiên** — họ được quyền đi nước đầu tiên và có thể chiếm vị trí trung tâm sớm hơn.
* Ngay cả khi **cả hai bên đều chơi ngẫu nhiên**, lợi thế này vẫn tồn tại, vì người đi trước có xác suất cao hơn trong việc “vô tình” tạo được 4 quân liên tiếp trước đối thủ.

Vì vậy, tỷ lệ thắng khoảng **55–57% cho người chơi 1** là **hoàn toàn hợp lý** khi cả hai đều chơi ngẫu nhiên.

## Task 3: Minimax Search with Alpha-Beta Pruning

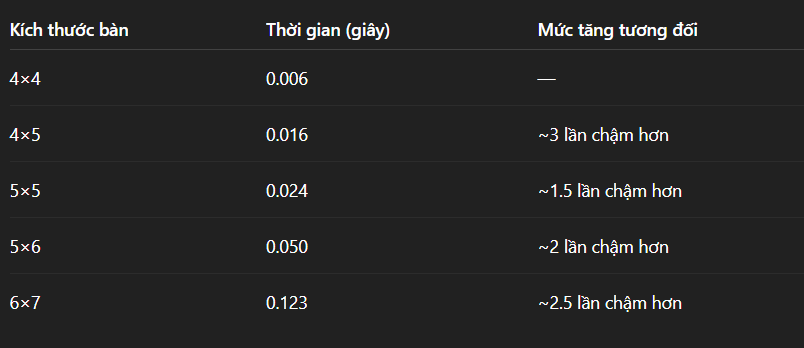
### **Implement the Search**



Test 3 thất bại (trả về None) vì:

Ở vị trí hiện tại, **không còn nước đi hợp lệ tạo ra thắng ngay trong cùng lượt**, nên trong nhánh minimax đệ quy, **hàm** actions() **trả về danh sách rỗng**.

Khi đó, best\_move không được cập nhật, dẫn đến trả về None.



Cây tìm kiếm **minimax** có độ phức tạp **hàm mũ** theo số nước đi hợp lệ và độ sâu tìm kiếm:

với

b = hệ số nhánh (số lượng cột hợp lệ ≈ số nước đi khả thi),

d = độ sâu tìm kiếm.

Với trò **Connect Four**:

b ≈ 7b

d=4 → khoảng = 240174=2401 trạng thái

d= 6 → khoảng =117,649 trạng thái

d=8 → khoảng 5.7× trạng thái

Ngay cả khi có **cắt tỉa alpha–beta**, thuật toán vẫn phải duyệt qua **hàng nghìn đến hàng triệu trạng thái** nếu độ sâu ≥ 6.

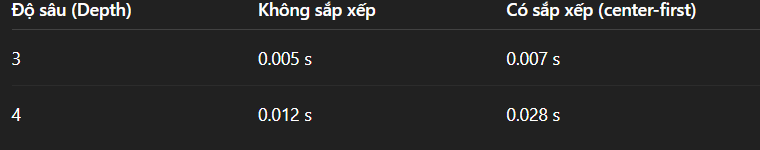
### Move ordering

Trong Connect Four, **nước đi ở cột giữa** thường tốt hơn, vì:

Có nhiều khả năng tạo chuỗi 4 liên tiếp.

Linh hoạt để mở rộng sang hai phía.

Do đó, ta **ưu tiên các cột gần trung tâm trước**.



Khi **bảng trống**, mọi nước đi đều tương đương →  
**sắp xếp nước đi chỉ thêm chi phí xử lý nhỏ**, mà **chưa giúp cắt tỉa** vì tất cả node đều mở rộng giống nhau.

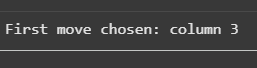
Tuy nhiên, **khi bàn có nhiều quân hơn**, và **một số nước đi tốt rõ rệt**,  
→ Việc duyệt “nước đi tốt trước” giúp **alpha–beta cắt bỏ nhiều nhánh tồi sớm hơn**.  
→ Khi đó, **move ordering** sẽ **giảm mạnh thời gian**.

### The first few moves

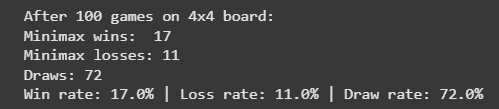
Để tránh việc tìm kiếm quá sâu ở trạng thái đầu, ta có thể:

1. **Chọn trung tâm ngay từ đầu** — nước đi giữa bàn thường là tối ưu nhất trong Connect 4.
2. **Khi bàn còn ít quân (ví dụ <4)** — ưu tiên các cột gần trung tâm (vì có nhiều hướng thắng hơn).
3. **Sau đó mới dùng Minimax với độ sâu giới hạn** để đánh giá các trạng thái phức tạp hơn.

→ Điều này giúp **giảm đáng kể thời gian tính toán**, vì Minimax không cần duyệt toàn bộ cây trò chơi ngay từ bước đầu tiên.



(Trên bàn 6×7, cột giữa là cột thứ 3 — đánh số từ 0.)



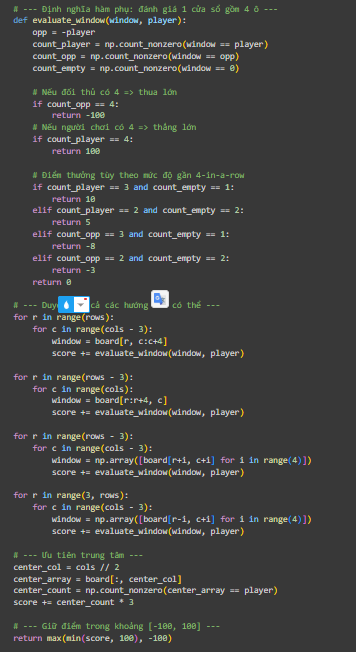
Tỷ lệ **hòa cao (72%)** là hợp lý vì bàn 4×4 quá nhỏ — khó tạo được 4 liên tiếp.

**Minimax thắng nhiều hơn Random**, cho thấy nó có khả năng ra quyết định tốt hơn, dù dùng **heuristic đơn giản**.

Khi tăng kích thước bàn hoặc độ sâu tìm kiếm, **tỷ lệ thắng của Minimax sẽ tăng rõ rệt**.

## Task 4: Heuristic Alpha-Beta Tree Search

## Heuristic evaluation function



**Duyệt qua tất cả các nhóm 4 ô liên tiếp (window)**

theo hàng ngang

theo cột dọc

theo chéo xuống (↘️)

theo chéo lên (↗️)

**Đánh giá từng nhóm 4 ô**:

Nếu người chơi có 4 quân liên tiếp → +100 (thắng mạnh)

Nếu đối thủ có 4 quân liên tiếp → −100 (thua nặng)

Nếu người chơi có 3 quân + 1 ô trống → +10 (gần thắng)

Nếu người chơi có 2 quân + 2 ô trống → +5

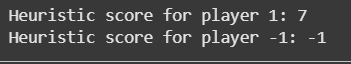
Nếu đối thủ có 3 quân + 1 ô trống → −8 (nguy hiểm)

Nếu đối thủ có 2 quân + 2 ô trống → −3

**Ưu tiên trung tâm bàn cờ**

Người chơi nào có nhiều quân ở cột giữa được thưởng thêm điểm, vì trung tâm có nhiều hướng chiến thắng hơn.

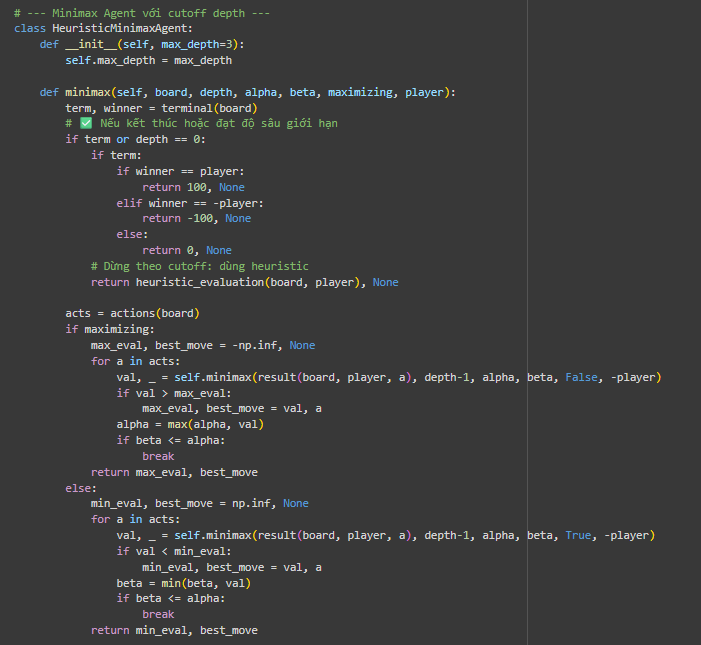
**Chuẩn hóa điểm cuối cùng trong khoảng** [-100, 100]  
→ tránh việc điểm số phóng đại do cộng dồn quá nhiều cửa sổ.



Người chơi **1 (đỏ)** có thế mạnh hơn một chút → điểm dương (+7).

Người chơi **-1 (vàng)** yếu thế hơn → điểm âm (−1).

## Cutting Off Search



Sử dụng class HeuristicMinimaxAgent (độ sâu = 3).

Tạo các bàn cờ trống với kích thước tăng dần:

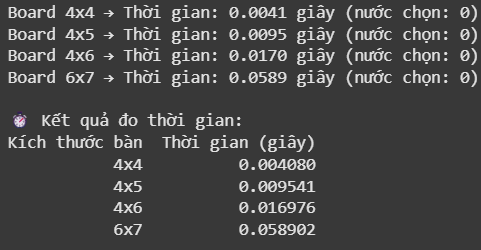
4×4

4×5

4×6

6×7 (chuẩn Connect Four)

Đo thời gian cần để agent chọn một nước đi đầu tiên (nặng nhất vì phải xét toàn bộ cây tìm kiếm).



Khi bàn cờ **tăng số cột và hàng**, **số nhánh hành động (branching factor)** tăng lên rất nhanh.

Mỗi lần mở rộng một cột mới, agent phải duyệt thêm hàng nghìn trạng thái tiềm năng.

Với độ sâu = 3, thuật toán vẫn chạy được,  
nhưng nếu tăng lên độ sâu = 5 hoặc 6 → thời gian có thể **tăng gấp hàng chục lần** (vì tăng theo cấp số nhân).

## Playtime

Cho **hai agent heuristic Minimax** có **độ sâu cắt khác nhau (cutoff depth)** thi đấu với nhau để:

So sánh khả năng “nhìn xa” của từng agent.

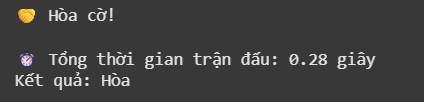
Quan sát kết quả trận đấu (thắng/thua/hòa).

Kiểm chứng hiệu quả của độ sâu tìm kiếm.

· **Agent 1 (Player 1):** HeuristicMinimaxAgent(max\_depth=3)  
→ nhìn trước 3 nước, phản ứng nhanh nhưng kém chiến lược.

· **Agent 2 (Player -1):** HeuristicMinimaxAgent(max\_depth=5)  
→ nhìn xa hơn, dự đoán tốt hơn nhưng tốn thời gian hơn.

· **Kích thước bàn:** 4×5  
(vừa đủ để thấy sự khác biệt mà không khiến thuật toán chạy quá lâu).



Hai agent thay phiên thả quân, tất cả nước đi đều hợp lệ.

Trò chơi kéo dài **20 lượt**, sau đó bàn cờ đầy mà **không có ai 4 quân liên tiếp**.