**TRƯỜNG ĐẠI HỌC SÀI GÒN**

**KHOA CÔNG NGHỆ THÔNG TIN**

----🙝☯🙟----

A blue circle with white text

Description automatically generated

**BÁO CÁO**

**Học phần: Trí tuệ nhân tạo nâng cao**

**Thảo luận: Bài tập Lab04-Nhóm 13**

| Giảng viên hướng dẫn | Đỗ Như Tài |
| --- | --- |
| Thành viên | 3 3121410541- Nguyễn Minh Trường  3 3122410279- Nguyễn Phan Việt Nhật  3 3122410401-Đinh Công Thuận  3122410384- Hoàng Minh Thành |

**Thành phố Hồ Chí Minh, tháng 9, năm 2025**

**Bảng phân công**

| **Phân công** | **Thành Viên** |
| --- | --- |
| 3121410541- Nguyễn Minh Trường | Lab04    **### Solving games using adversarial search**  \* Example: [Solving Tic-Tac-Toe with Minimax Search and Alpha-Beta Pruning](t**ictactoe\_alpha\_beta\_tree\_search.ipynb**)  \* Example: [Solving Tic-Tac-Toe with Heuristic Alpha-Beta Tree Search](**tictactoe\_heuristic\_alpha\_beta\_tree\_search.ipynb**)  **### Assignments**  Assignment: [Adversarial Search: Playing "Mean" Connect 4](**assignment\_mean\_connect4.ipynb**) |
| 3122410279- Nguyễn Phan Việt Nhật | **LAB 04**  **### Defining a Game**  \* Example [Tic-Tac-Toe](**tictactoe\_definitions.ipynb**)  **### Interactive game play**  \* Example: [Play Tic-Tac-Toe Interactively (Simple Implementation)](**tictactoe\_interactive.ipynb**)  **### Assignments**  \* Assignment: [Adversarial Search: Playing Connect 4](**assignment\_connect4.ipynb**) |
| 3122410401-Đinh Công Thuận | **Lab 05**  \* Example: [Solving the Graph Coloring using Simple Backtracking Search](**CSP\_graph\_coloring\_example.ipynb**)  \* Connection to Search: The assignment [The n-Queens Problem](**n\_queens.ipynb**) asks you to use local search to solve a CSP. |
| 3122410384- Hoàng Minh Thành | Lab04    **### Solving games using adversarial search**  \* Example: [Solving Tic-Tac-Toe with Pure Monte Carlo Search](**tictactoe\_pure\_monte\_carlo\_search.ipynb**)  \* Example: [Solving Tic-Tac-Toe with Pure Monte Carlo Search + UCB1 Selection Policy](**tictactoe\_monte\_carlo\_tree\_search\_restricted.ipynb**)  **### Assignments**  \* Assignment: [Adversarial Search: Playing Dots and Boxes](**assignment\_dots\_and\_boxes.ipynb**) |

**NHẬN XÉT CỦA GIẢNG VIÊN**

**….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….………………………………………………………………………….……………………………………………………………………….……………………………………………………………………….……………………………………………………………………….……………………………………………………………………….……………………………………………………………………….**

[**Phần I: Tic-Tac-Toe (Definition & Interactive) and Connect 4 Assignment 6**](#_d3likpqsbbsn)

[I. Tic Tac Toe – Definitions 6](#_7owjo75t1c3t)

[1. Giới thiệu 6](#_qh6hg0owt92f)

[2. Định nghĩa bài toán tìm kiếm (The Search Problem) 7](#_3v11z8i84rgt)

[3. Kích thước không gian bài toán (Problem Size) 8](#_tm474aktjnvm)

[4. Ước lượng chính xác hơn cho không gian trạng thái (State Space Estimates) 8](#_b4sp94w89t7w)

[5. Độ phức tạp thời gian và không gian (Space and Time Complexity Using the Search Tree Estimates) 10](#_koynfpt3pktf)

[6. Triển khai các hàm trong trò chơi (Implementation) 12](#_745yelbkppvu)

[7. Thực nghiệm (Experiments) 15](#_douboa81wn1n)

[7.1. Baseline: Người chơi ngẫu nhiên (Randomized Player) 15](#_v1vnylwo5tfz)

[7.2.Trận đấu Random vs Random 15](#_tbgrj1rlsole)

[**II. Tic Tac Toe – Interactive Version 16**](#_3cjmahggp49f)

[1. Chuẩn bị môi trường và import thư viện 16](#_s48oes56wv2a)

[2. Người chơi tương tác (Interactive Player) 16](#_dugiw9p69dk2)

[3. Bắt đầu trò chơi tương tác với tư cách người chơi O 17](#_fz9tq6mpx1fu)

[4. Bắt đầu trò chơi tương tác với tư cách người chơi X 18](#_oie2lkjtxwel)

[III. Connect 4 Assignment 19](#_y1910qd4npow)

[1. Tổng quan 19](#_qh9gljgfmkv0)

[2. Task 1: Defining the Search Problem 20](#_m3ubeyh1m3f7)

[2.2. Mô hình hóa trò chơi Connect 4 20](#_t7m0sqnjwe7)

[2.2.1. Khởi tạo bàn cờ 20](#_hf0qah6nye1o)

[2.2.2. Các hành động hợp lệ 20](#_5dhszhyrw7vc)

[2.2.3. Cập nhật trạng thái sau khi đi 21](#_b3e93891e810)

[2.2.4. Kiểm tra người thắng 21](#_hirzxk2hso67)

[2.2.5. Kiểm tra trạng thái kết thúc và tính điểm 22](#_18irekjcvf67)

[2.3. Phân tích độ phức tạp của trò chơi 23](#_2bfz8us77rns)

[2.3.1. Kích thước không gian trạng thái 23](#_yg7ey3hwbda8)

[2.3.2. Kích thước cây trò chơi 23](#_npp1kwsi9c4w)

[3. Task 2: Môi trường trò chơi và Agent ngẫu nhiên 24](#_fyhxcoreafcn)

[3.1. Biểu diễn bàn cờ bằng NumPy 24](#_6ptgby6oyy5t)

[3.2. Trực quan hóa bàn cờ 25](#_7c8hgyxhpsbe)

[3.3. Các hàm hỗ trợ (Helper Functions) 25](#_cdhn85dndby1)

[3.4. Agent ngẫu nhiên (Random Agent) 28](#_6sebm9jx80xe)

[3.5. Thử nghiệm agent ngẫu nhiên 29](#_itmyuai3w5ey)

[3.6. Mô phỏng trò chơi tự động giữa hai agent ngẫu nhiên 30](#_5gx6mscm6mqq)

[4. Task 3: Minimax Search with Alpha–Beta Pruning 31](#_k5k76mgo3a16)

[4.1. Cài đặt Minimax Search 31](#_jdl9ooetdipm)

[4.2. Thử nghiệm trên các bàn cờ nhỏ (4×4) 32](#_wl62mctthe24)

[4.3. Đo thời gian chạy theo kích thước bàn 33](#_kxs4dvt5608n)

[4.4. Move Ordering (Sắp xếp nước đi) 34](#_8tjefwvw1ka)

[4.6. Đấu giữa Minimax và Random Agent 35](#_5ohxwjd8xcgf)

[5. Task 4: Heuristic Alpha-Beta Tree Search 36](#_d03l2wgu1du7)

[5.1. Heuristic Evaluation Function 36](#_d626ww86xqxr)

[5.2. Cutting Off Search (Alpha-Beta Pruning + Depth Limit) 37](#_9qt5ywb4xla1)

[5.3. Thực nghiệm với các bàn cờ mẫu 38](#_8e1l6kgu0siw)

[5.4. Đo thời gian chạy với kích thước bàn khác nhau 40](#_hexwj77ayoo1)

[5.5. Thi đấu giữa hai Heuristic Agents 41](#_qpqg3am6qrbq)

[\* Bonus Task: Monte Carlo Search & Best First Move 41](#_ptrools5d1iu)

[\*\* Thuật toán Pure Monte Carlo Search 41](#_60mgioffp0u9)

[\*\*Cài đặt Pure Monte Carlo Agent 43](#_qyltgfx3wrp7)

[\*\* Thử nghiệm trên các bàn kiểm tra (test boards) 44](#_nsgh09np8eqk)

[\*\* Best First Move (Tìm nước đi đầu tiên tốt nhất) 45](#_au7i05359lwo)

[6. Kết luận 47](#_pzv017klhkc7)

[**Phần II:Solving games using adversarial search — Minimax, Alpha-Beta, Heuristic search 47**](#_klisd8sqgtkl)

[Phần 1 : Solving Tic-Tac-Toe with Minimax Search and Alpha-Beta Pruning 47](#_nbyu3ybo5yg)

[1. Giới thiệu và đặt vấn đề 47](#_4kjxwx7eoowk)

[2. Kiến thức nền tảng (Core Theory) 48](#_3zn6351ey0ys)

[2.1. Nguyên lý Minimax 48](#_c78y1gplbjjd)

[2.2. Cấu trúc cây Minimax 49](#_z0vcgdrp7w2j)

[2.3. Cải tiến bằng Alpha-Beta Pruning 50](#_2rm73tqtqzni)

[2.4. Phân tích độ phức tạp O(b^d) 50](#_t6c08eey671r)

[3. Thực nghiệm và Phân tích (Experiment & Analysis) 51](#_s75lgar062j0)

[3.1. Mục tiêu thực nghiệm 51](#_2ezlf37gxh6l)

[3.2. Thực nghiệm 1: Minimax + Alpha-Beta cơ bản 51](#_bdjoiiuaiir7)

[3.3. Thực nghiệm 2: Heuristic Alpha-Beta (đánh giá & giới hạn độ sâu) 52](#_ssjmxnruc3ja)

[3.4. Nhận xét và kết luận 53](#_ga7e3qlf2x0r)

[**Phần III: Dots and Boxes 53**](#_aiejtzc6xh8h)

[Giải thích ngắn : 55](#_ksf48hkt22id)

[Giải thích : 57](#_39sv5cc45hcc)

[Cách hoạt động: 57](#_y8wy9m64wuys)

[Nhận xét: 59](#_sf5btkg5stai)

[Mô tả thí nghiệm 59](#_4cau1k52ktrt)

[Nhận xét: 59](#_pd5ondjz5zfg)

[Mục đích: 60](#_53jx63qm0x4o)

[Nhận xét: 60](#_cx1xemmnapm8)

[Ý tưởng: 60](#_72cw8ny8100m)

[Kết luận: 61](#_x01vyk7z5nb7)

[**Giải thích : 62**](#_6dn7qyas6043)

[**Phân tích: Khi tăng cutoff depth, số lượng nút được duyệt và thời gian tính tăng theo cấp số mũ. 63**](#_986h4fn1li0j)

[**Kết luận : Cutoff depth là kỹ thuật quan trọng để Minimax có thể sử dụng được trong thực tế. Thay vì duyệt hết cây game, ta dừng tại một độ sâu và dùng heuristic evaluation. 63**](#_264gppvwtmuv)

[**Nhận xét: 64**](#_8p9xdd8woczl)

[**Phân tích 64**](#_jb0wd73xnqtk)

[**Kết luận 65**](#_q2qxrwk93dcm)

# Phần I: Tic-Tac-Toe (Definition & Interactive) and Connect 4 Assignment

## Tic Tac Toe – Definitions

### 1. Giới thiệu

Trò chơi **Tic-Tac-Toe** (hay “Cờ ca-rô 3×3”) là một ví dụ kinh điển được sử dụng trong lĩnh vực **Trí tuệ nhân tạo (Artificial Intelligence)** để minh họa các khái niệm cơ bản về **bài toán tìm kiếm (search problem)** và **môi trường trò chơi có đối thủ (adversarial environment)**.

Trong trò chơi này, hai người chơi — “X” và “O” — lần lượt điền ký hiệu của mình vào một ô trống trên bàn cờ 3×3. Mục tiêu là tạo ra một hàng, cột hoặc đường chéo gồm ba ký hiệu giống nhau.

Dù đơn giản, trò chơi này cho phép ta phân tích:

* Cấu trúc của **bài toán tìm kiếm trạng thái**;
* Cách mô hình hóa các **hàm hành động (Actions)** và **hàm kết quả (Result)**;
* Cách xác định **trạng thái kết thúc (Terminal)** và **giá trị tiện ích (Utility)**.

### 2. Định nghĩa bài toán tìm kiếm (The Search Problem)

Một bài toán tìm kiếm tổng quát trong trí tuệ nhân tạo thường được xác định thông qua các thành phần sau:

| **Thành phần** | **Diễn giải trong Tic-Tac-Toe** |
| --- | --- |
| **Initial State (Trạng thái ban đầu)** | Bàn cờ 3×3 trống, đến lượt người chơi “X”. |
| **Actions (Hành động)** | Đặt ký hiệu của bạn (X hoặc O) vào một ô trống bất kỳ. |
| **Transition Function (Hàm chuyển trạng thái)** | Bàn cờ thay đổi khi người chơi đặt ký hiệu. Sau đó đến lượt đối thủ đi. |
| **Goal State (Trạng thái mục tiêu)** | Một trong hai người chơi thắng (ba ký hiệu liên tiếp theo hàng, cột hoặc chéo). |
| **Path Cost (Chi phí đường đi)** | Số nước đi (thường không xét trong trò chơi này, vì mục tiêu là thắng chứ không phải đi ít nước). |

Vì Tic-Tac-Toe là **một trò chơi hai người có đối kháng**, nên môi trường được xem là **phi xác định (non-deterministic)**.

Tuy nhiên, ta có thể mô hình hóa bằng các thành phần đặc trưng trong lĩnh vực **AI trò chơi (game AI)**:

* Actions(s) — Các nước đi hợp lệ trong trạng thái s.
* Result(s,a) — Trạng thái mới sau khi thực hiện hành động a.
* Terminal(s) — Kiểm tra xem trò chơi đã kết thúc hay chưa.
* Utility(s) — Đánh giá mức độ lợi ích của trạng thái kết thúc (thắng, thua, hòa) cho người chơi “Max”

Ghi chú:

* Để xác định trạng thái đích, chúng ta sẽ sử dụng một phép thử cho trạng thái kết thúc (trò chơi kết thúc) và một hàm tiện ích (thắng hay thua).
* thuật toán được dùng là **DFS (Depth-First Search)** nên kết quả có thể **không tối ưu** (không nhất thiết là con đường ngắn nhất để thắng).
* Do đó, yếu tố “path cost” (chi phí đường đi) được **bỏ qua** trong quá trình mô hình hóa.

### 3. Kích thước không gian bài toán (Problem Size)

Trước khi giải quyết, ta cần ước lượng **số lượng trạng thái khả dĩ** của trò chơi.  
 Mỗi ô trên bàn cờ có thể ở trong 1 trong 3 trạng thái:

* Rỗng (∅),
* Có ký hiệu X,
* Có ký hiệu O.

Vì có 9 ô, tổng số tổ hợp lý thuyết là:

39=19,683

Tuy nhiên, đây là **ước lượng tối đa**, vì một số trạng thái không thể xảy ra trong thực tế (ví dụ: cả hai người cùng thắng trong cùng một thời điểm).

### 4. Ước lượng chính xác hơn cho không gian trạng thái (State Space Estimates)

Một cách tiếp cận chặt chẽ hơn:

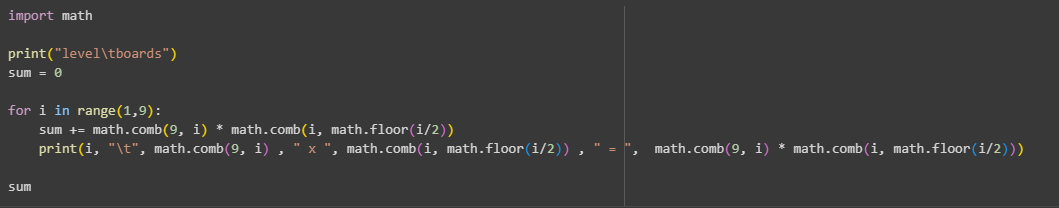
Ở mỗi bước i (số ô đã được đi), ta có:

* C(9,i) cách chọn ra i ô từ 9 ô,
* và C(i,⌊i/2⌋) cách chia số ô đó cho hai người chơi (khoảng một nửa là X, nửa còn lại là O).

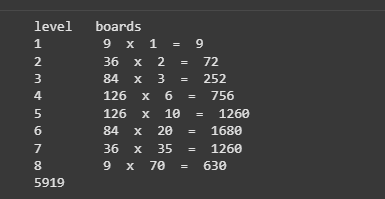
Tổng cộng, ta có:



là **ước lượng tốt hơn về số trạng thái hợp lệ**.



* Dòng math.comb(9, i) tính số cách chọn i ô từ 9 ô (tổ hợp chập i của 9).
* Dòng math.comb(i, math.floor(i/2)) ước lượng số cách chia lượt đi giữa hai người.
* Biến sum lưu tổng số trạng thái có thể đạt được khi cộng qua tất cả các giá trị i.



Nhận xét:

* Số lượng trạng thái tăng mạnh ở các cấp độ giữa (đặc biệt là từ level 4–6), do lúc này số lượng nước đi và vị trí còn trống đều lớn.
* Tổng số trạng thái có thể xảy ra của trò chơi Tic Tac Toe là **5,919** nếu ta xét đến tất cả khả năng đặt quân hợp lệ.
* Việc liệt kê và phân tích các trạng thái này là cơ sở để đánh giá độ phức tạp của bài toán, giúp xác định mức độ tìm kiếm hoặc học máy cần thiết khi xây dựng một AI chơi Tic Tac Toe.

→ Kết quả cho thấy **ước lượng thực tế chỉ khoảng 5919 trạng thái hợp lệ**, thấp hơn nhiều so với 19.683

### 5. Độ phức tạp thời gian và không gian (Space and Time Complexity Using the Search Tree Estimates)

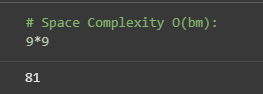
Để giải bài toán Tic-Tac-Toe, ta không chỉ xem xét không gian trạng thái, mà còn cần ước lượng **độ phức tạp tìm kiếm (search complexity)**.  
 Thuật toán được sử dụng là **DFS (Depth-First Search)** – tìm kiếm theo chiều sâu.

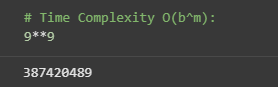
Một **cây tìm kiếm (search tree)** có thể **lớn hơn nhiều so với không gian trạng thái** bởi vì có thể tồn tại nhiều đường đi khác nhau dẫn đến cùng một trạng thái bàn cờ (các trạng thái bị lặp).  
 Do đó, cây tìm kiếm loại bỏ các **chu trình (cycles)**, nhưng **vẫn chứa các nhánh thừa (redundant paths)**.

Các thông số đặc trưng của bài toán:

* **Độ sâu cực đại** của cây tìm kiếm: m = 9 (tối đa 9 nước đi).
* **Hệ số phân nhánh cực đại**: b = 9 (ở nước đi đầu tiên có 9 lựa chọn).

Đối với **DFS**, ta có:

* **Độ phức tạp không gian**: O(bm)
* **Độ phức tạp thời gian**: O(bm)



→ Cần khoảng **81 trạng thái tạm thời trong bộ nhớ** (frontier + current path).

→ Tức là nếu duyệt toàn bộ cây tìm kiếm, thuật toán có thể phải mở rộng đến **hàng trăm triệu nút**.

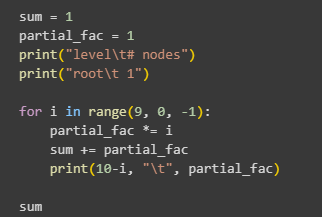
Số lượng nút ở mỗi cấp độ của cây tìm kiếm:

| **Level** | **# of Nodes (số nút)** |
| --- | --- |
| root | 1 |
| 1 | 9 |
| 2 | 72 |
| 3 | 504 |
| 4 | 3024 |
| 5 | 15120 |
| 6 | 60480 |
| 7 | 181440 |
| 8 | 362880 |
| 9 | 362880 |
| **Tổng cộng** | **986,410** |

# 

# 

# 



Giải thích:

* Ở mỗi cấp, số nút tăng theo tích của các hệ số phân nhánh (9×8×7×…).
* Tổng số nút ước tính ~986 nghìn (ít hơn đáng kể so với 99) vì một số ván kết thúc sớm.

### 6. Triển khai các hàm trong trò chơi (Implementation)

Sau khi đã hiểu rõ không gian và độ phức tạp, ta cần **cài đặt các hàm cơ bản** để mô hình hóa trò chơi.

Các hàm cần thiết:

| **Hàm** | **Chức năng** |
| --- | --- |
| Actions(s) | Trả về danh sách các nước đi hợp lệ trong trạng thái *s*. |
| Result(s, a) | Mô hình chuyển trạng thái: cho biết trạng thái mới sau khi thực hiện hành động *a* trong trạng thái *s* |
| Terminal(s) | Kiểm tra xem trạng thái *s* có phải là trạng thái kết thúc (thắng/thua/hòa) hay chưa.. |
| Utility(s) | Trả về giá trị tiện ích (+1 nếu “X” thắng, -1 nếu “O” thắng, 0 nếu hòa). |

Trạng thái bàn cờ được biểu diễn bằng **một danh sách gồm 9 phần tử**, tương ứng với 9 ô vuông trên bàn cờ.

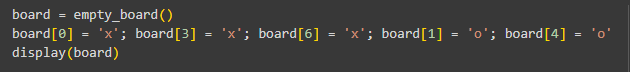
Mỗi ô có thể nhận giá trị ' ', 'x', hoặc 'o'.  
 Các hành động hợp lệ là **chỉ số của các ô trống** (0 → 8).

Các hàm này, cùng với một số hàm phụ trợ khác, được định nghĩa trong tệp *tictactoe.py* và được import vào như sau:



**Ví dụ minh họa**

* Tạo một bàn cờ có sẵn một số nước đi:



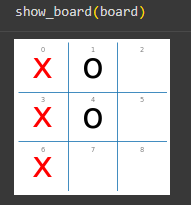
* Kết quả dạng danh sách:



* Dạng hiển thị văn bản:



* Dạng hiển thị trực quan:

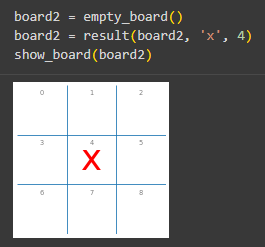


→ Ta thấy người chơi **X** đã thắng theo cột đầu tiên.

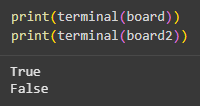
* Kiểm tra các hành động hợp lệ còn lại:



* Tạo bàn cờ mới và cập nhật nước đi:



* Kiểm tra trạng thái kết thúc:



→ Bàn cờ đầu tiên đã kết thúc (vì X thắng), trong khi bàn cờ thứ hai vẫn đang chơi.

### 7. Thực nghiệm (Experiments)

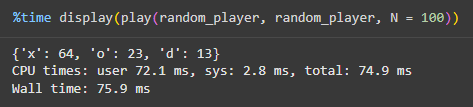
#### 7.1. Baseline: Người chơi ngẫu nhiên (Randomized Player)

Trước khi triển khai các chiến lược thông minh hơn, ta xây dựng một **mốc chuẩn (baseline)** với người chơi chọn nước đi hoàn toàn **ngẫu nhiên**.  
 Các hàm sau được cài đặt trong [*tictactoe.py*](http://tictactoe.py):



#### 7.2.Trận đấu Random vs Random

Cho hai người chơi ngẫu nhiên thi đấu với nhau 100 ván:



**Nhận xét:**

* Người chơi **X** (đi trước) thắng **64%** số ván.
* Người chơi **O** thắng **23%**, và **13%** ván hòa.
* Thời gian trung bình cho 100 ván là ~76 ms.

→ Điều này cho thấy **lợi thế của người chơi đi trước (X)** trong trò chơi Tic-Tac-Toe, ngay cả khi cả hai đều chọn nước đi ngẫu nhiên.

## Tic Tac Toe – Interactive Version

### Chuẩn bị môi trường và import thư viện

Các hàm cơ bản để điều khiển logic trò chơi được cài đặt trong file tictactoe.py.  
 Chúng bao gồm:

* empty\_board() – tạo bàn cờ trống 3x3.
* actions(board) – trả về danh sách các ô còn trống (nước đi hợp lệ).
* result(board, move) – trả về bàn cờ mới sau khi người chơi thực hiện nước đi.
* terminal(board) – kiểm tra xem trò chơi đã kết thúc chưa.  
  utility(board) – trả về điểm số của trạng thái bàn cờ:
* 1 nếu X thắng,
* -1 nếu O thắng,
* 0 nếu hòa.
* other(player) – đổi lượt giữa hai người chơi (X ↔ O).
* show\_board(board) – hiển thị bàn cờ hiện tại ra màn hình.
* random\_player(board) – chọn ngẫu nhiên một nước đi hợp lệ.
* play(p1, p2) – chạy toàn bộ trò chơi giữa hai người chơi (có thể là người hoặc máy).

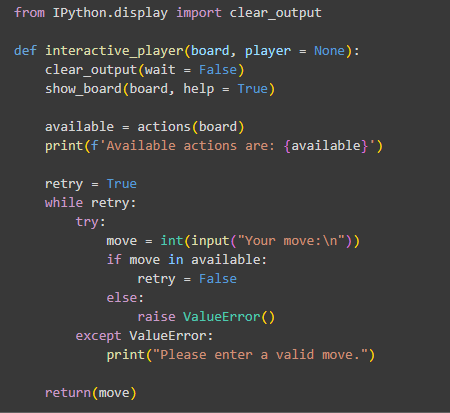


### Người chơi tương tác (Interactive Player)

Ta cài đặt hàm interactive\_player() – cho phép người dùng nhập nước đi trực tiếp trong môi trường notebook.

Hàm sẽ hiển thị bàn cờ, liệt kê các nước đi hợp lệ, và yêu cầu người chơi nhập vị trí mong muốn.

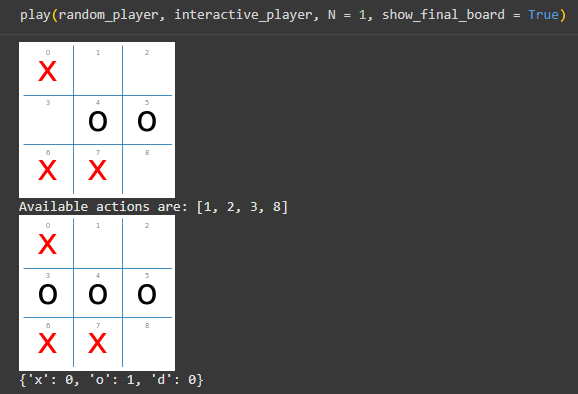
Mỗi lần nhập, bàn cờ sẽ được **làm mới (clear\_output)** để hiển thị trạng thái cập nhật.



* clear\_output(wait=False): làm mới màn hình hiển thị giúp cập nhật bàn cờ sau mỗi lượt.
* show\_board(board, help=True): hiển thị bàn cờ hiện tại và các chỉ số ô (0–8).
* available = actions(board): liệt kê các ô trống mà người chơi có thể chọn.
* Vòng while retry: đảm bảo người chơi nhập **một số hợp lệ**, nếu sai, chương trình yêu cầu nhập lại.
* Hàm trả về giá trị move (vị trí nước đi) dưới dạng số nguyên tương ứng với ô được chọn.

### Bắt đầu trò chơi tương tác với tư cách người chơi O

Sau khi định nghĩa người chơi tương tác, ta có thể thử **bắt đầu trò chơi với vai trò là O**, trong khi **máy tính (random\_player)** sẽ là X.



* play() chạy trò chơi giữa hai người chơi (ở đây là máy và người chơi).
* Tham số N=1 cho biết chỉ chơi 1 ván.
* show\_final\_board=True giúp hiển thị bàn cờ sau khi ván kết thúc.
* Kết quả {‘x’: 0, ‘o’: 1, ‘d’: 0} thể hiện thống kê số ván thắng:
* 'x': số ván X thắng
* 'o': số ván O thắng
* 'd': số ván hòa

### Bắt đầu trò chơi tương tác với tư cách người chơi X

Tiếp theo, người chơi có thể vào vai **X**, và máy tính sẽ đóng vai **O**.



* Khi người chơi X đi hết hàng giữa, X thắng.
* Trò chơi kết thúc, và thống kê kết quả cho thấy:
* 'x': 1 → người chơi X thắng 1 ván.
* 'o': 0 → máy không thắng ván nào.
* 'd': 0 → không có ván hòa.

## Connect 4 Assignment

### 1. Tổng quan

Bài tập này mô phỏng trò chơi **Connect 4** (hay còn gọi là “Cờ 4 ô liên tiếp”) — một trò chơi hai người, trong đó mỗi người lần lượt thả quân vào một trong bảy cột của bảng 6×7. Người chiến thắng là người đầu tiên tạo được **4 quân liên tiếp** (theo hàng ngang, dọc hoặc chéo).

Hướng dẫn **xây dựng nền tảng của môi trường trò chơi**, bao gồm các hàm:

* Khởi tạo trạng thái bàn cờ.
* Liệt kê các hành động hợp lệ.
* Cập nhật trạng thái sau khi người chơi đi.
* Kiểm tra điều kiện thắng/thua/hòa.
* Xác định không gian trạng thái và kích thước cây trò chơi (game tree).

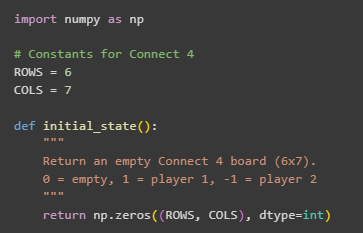
### 2. Task 1: Defining the Search Problem

#### 2.2. Mô hình hóa trò chơi Connect 4

##### **2.2.1. Khởi tạo bàn cờ**

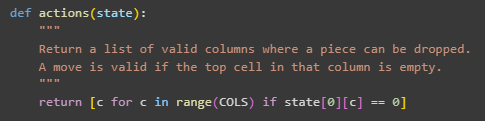
Đầu tiên, tạo một bàn cờ 6×7, trong đó:

* 0 là ô trống.
* 1 là quân của người chơi 1.
* -1 là quân của người chơi 2.



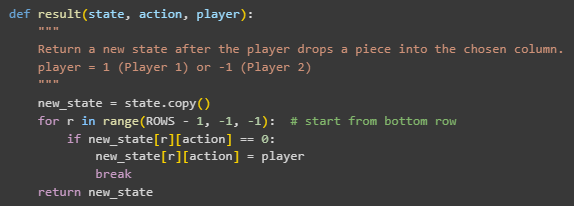
* Hàm initial\_state() tạo ra một ma trận 6×7 toàn số 0 — biểu diễn bàn cờ trống.
* Đây là trạng thái ban đầu của mọi ván Connect 4

##### **2.2.2. Các hành động hợp lệ**



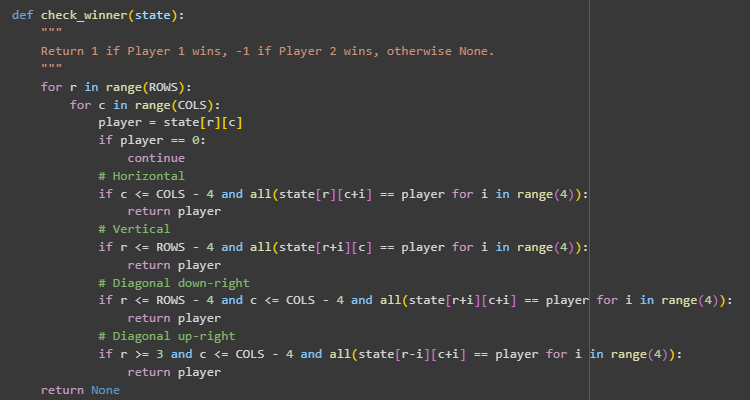
* Mỗi lượt, người chơi có thể chọn một **cột còn trống** để thả quân.
* Nếu ô trên cùng của cột (state[0][c]) vẫn là 0, nghĩa là cột đó chưa đầy → hợp lệ.
* Hàm này trả về danh sách các cột mà người chơi có thể đi.

##### **2.2.3. Cập nhật trạng thái sau khi đi**

****

* Khi người chơi chọn cột action, quân sẽ “rơi” xuống ô trống thấp nhất trong cột đó.
* Duyệt từ hàng cuối lên để tìm ô trống đầu tiên (0), sau đó gán bằng player (1 hoặc -1).
* Trả về trạng thái mới mà không làm thay đổi bàn cờ gốc (dùng copy()).

##### 2.2.4. Kiểm tra người thắng

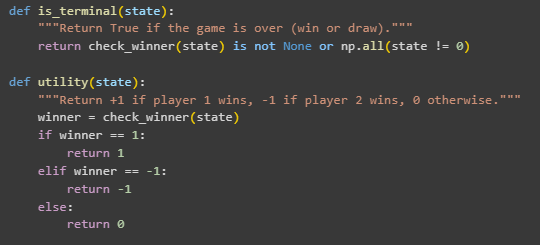


* Hàm này duyệt qua từng ô trên bàn cờ, và nếu ô đó thuộc về một người chơi (player ≠ 0), nó sẽ kiểm tra 4 hướng có thể tạo thành chuỗi 4 quân liên tiếp:

1. Ngang →
2. Dọc ↓
3. Chéo xuống phải ↘
4. Chéo lên phải ↗

* Nếu bất kỳ hướng nào có 4 quân liên tiếp cùng giá trị, trả về 1 (người 1 thắng) hoặc -1 (người 2 thắng).
* Nếu chưa ai thắng, trả về None.

##### 2.2.5. Kiểm tra trạng thái kết thúc và tính điểm



* is\_terminal: Trò chơi kết thúc khi:
  + Có người thắng (check\_winner(state) khác None).
  + Hoặc bàn cờ đã đầy (np.all(state != 0)).
* utility: Trả về giá trị kết quả cuối cùng:
  + +1 → người 1 thắng
  + -1 → người 2 thắng
  + 0 → hòa
* Đây là hai hàm rất quan trọng, thường dùng trong thuật toán tìm kiếm như **Minimax**.

#### 2.3. Phân tích độ phức tạp của trò chơi

##### 2.3.1. Kích thước không gian trạng thái

How big is the state space? Give an estimate and explain it.

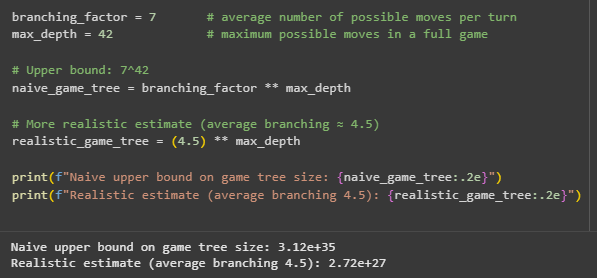


* Mỗi ô có 3 khả năng: trống, người 1, hoặc người 2 → tổng cộng 3^(6×7) = 3^42 ≈ 1.09×10^20 trạng thái.
* Nhưng thực tế, không phải tất cả đều hợp lệ (do luật chơi).

=> Trò chơi tuy nhỏ, nhưng **không gian trạng thái cực kỳ lớn**, chứng minh tính phức tạp của Connect 4.

##### 2.3.2. Kích thước cây trò chơi

How big is the game tree that minimax search will go through? Give an estimate and explain it.

****

* Mỗi lượt chơi có thể chọn 1 trong 7 cột → hệ số phân nhánh (branching factor) là 7.
* Trò chơi dài tối đa 42 nước (khi bàn đầy).

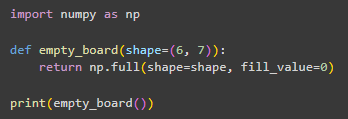
Do đó:

* Ước lượng đơn giản: 742 ≈ 1.9×1035 nhánh.
* Ước lượng thực tế hơn (vì nhiều cột bị đầy dần): (4.5)42 ≈ 4.3×1026.

=> Không thể duyệt hết toàn bộ cây trò chơi, nên cần thuật toán thông minh như Minimax + Alpha-Beta pruning.

### 3. Task 2: Môi trường trò chơi và Agent ngẫu nhiên

#### 3.1. Biểu diễn bàn cờ bằng NumPy

****

* Bàn cờ được biểu diễn bằng một mảng NumPy 2 chiều (mặc định kích thước 6x7).
* Mỗi ô có giá trị:
  + 0: ô trống
  + 1: quân của người chơi 1 (đỏ)
  + -1: quân của người chơi 2 (vàng)

- Hàm empty\_board() tạo ra bàn cờ mới trống hoàn toàn.

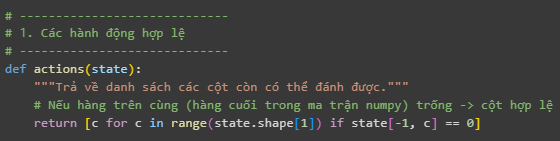
#### 3.2. Trực quan hóa bàn cờ

Notebook cung cấp sẵn hàm visualize() (của Randolph Rankin) để hiển thị bàn cờ



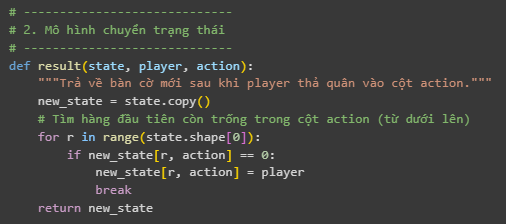
#### 3.3. Các hàm hỗ trợ (Helper Functions)

**a. Xác định hành động hợp lệ**

****

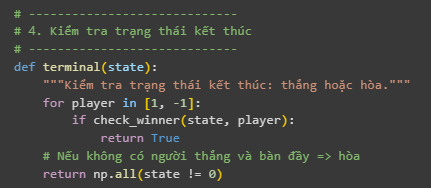
* Một hành động hợp lệ là một cột chưa đầy.
* Vì quân cờ “rơi” từ trên xuống, ta chỉ cần kiểm tra hàng trên cùng (state[-1, c]).
* Nếu ô này trống (0), người chơi có thể chọn cột đó.

**b. Hàm mô hình chuyển trạng thái**

****

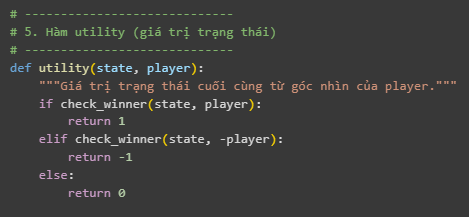
* Khi người chơi chọn cột action, quân sẽ “rơi” xuống hàng trống thấp nhất.
* Hàm tạo một bản sao của bàn (np.copy) → tránh làm thay đổi trạng thái gốc.
* Sau khi đặt quân, trả về bàn cờ mới.

**c. Kiểm tra trạng thái kết thúc**

****

* Trò chơi kết thúc khi:
* Có người thắng (check\_winner(state) != 0).
* Hoặc bàn cờ đã đầy (np.all(state != 0)).

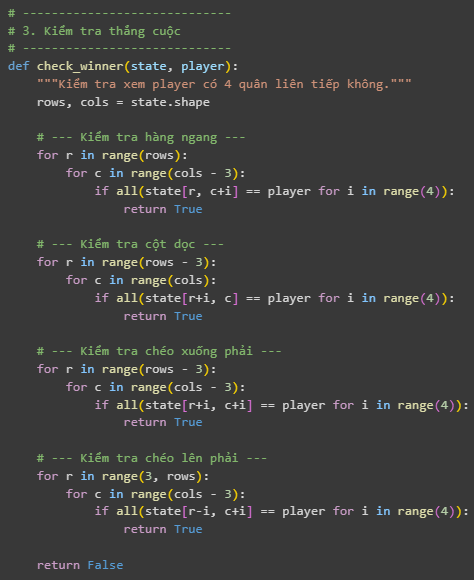
**d. Hàm tính điểm (Utility Function)**



* Nếu người chơi thắng → +1
* Nếu thua → -1
* Nếu hòa hoặc chưa kết thúc → 0

→ Dùng trong thuật toán Minimax để đánh giá trạng thái.

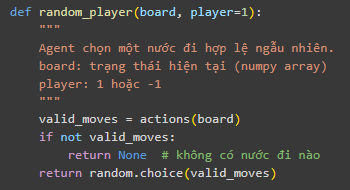
**e. Hàm kiểm tra người thắng cuộc**

****

* Duyệt tất cả các hướng có thể thắng (ngang, dọc, chéo trái và chéo phải).
* Nếu có 4 quân liên tiếp cùng dấu (1 hoặc -1), trả về người thắng tương ứng.
* Nếu chưa ai thắng → trả về 0.

#### 3.4. Agent ngẫu nhiên (Random Agent)

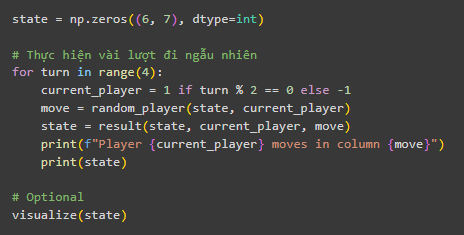
Sau khi đã có các hàm môi trường cơ bản (actions, result, terminal, utility, check\_winner), ta có thể xây dựng một agent đơn giản chọn nước đi ngẫu nhiên hợp lệ.

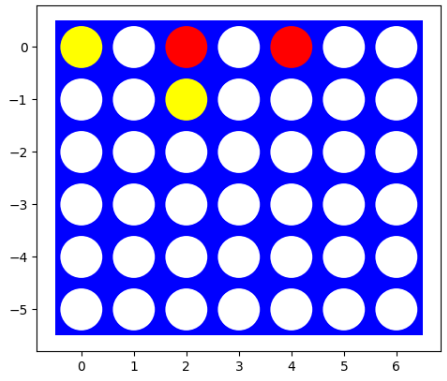


* Agent này không cần “suy nghĩ”, chỉ chọn ngẫu nhiên trong các cột còn trống.
* Đây là **đối thủ cơ bản** để so sánh với các agent thông minh hơn sau này (Minimax, MCTS…).

#### 3.5. Thử nghiệm agent ngẫu nhiên

Chạy thử vài lượt đi đầu tiên

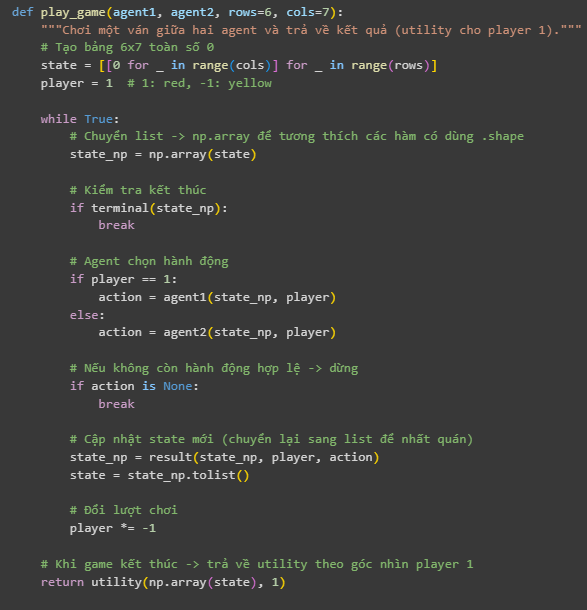


****

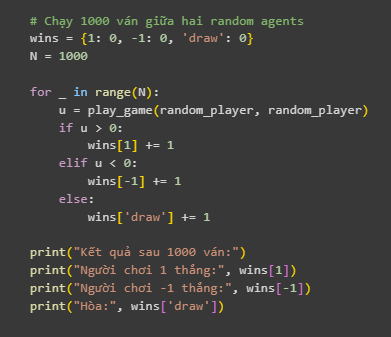
*Biểu đồ hiển thị đúng các vị trí đã thả quân.*

#### 3.6. Mô phỏng trò chơi tự động giữa hai agent ngẫu nhiên

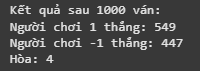
Để tạo mô phỏng tự động, ta dùng hàm play\_game():



**3.7. Chạy 1000 ván đấu giữa hai random agent**

****

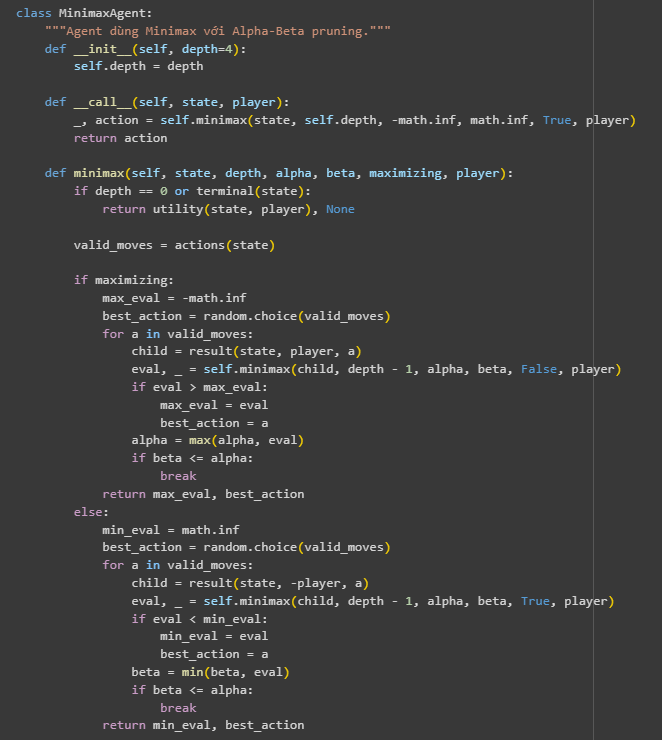
**Output:**

****

### 4. Task 3: Minimax Search with Alpha–Beta Pruning

#### 4.1. Cài đặt Minimax Search

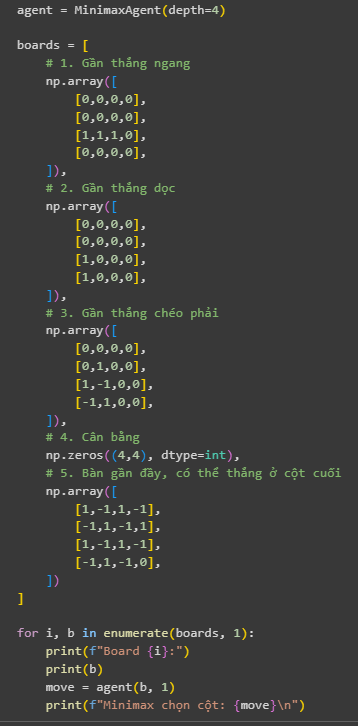
Thuật toán Minimax giúp **agent lựa chọn nước đi tốt nhất**, giả định rằng đối thủ cũng chơi tối ưu.



* Thuật toán **đệ quy** giữa hai lượt: người chơi (maximizing) và đối thủ (minimizing).
* **Alpha** và **Beta** dùng để **cắt tỉa** các nhánh không cần thiết (giảm thời gian duyệt).
* Dừng lại khi đạt **độ sâu giới hạn (depth)** hoặc **trạng thái kết thúc (terminal)**.
* **utility()** xác định thắng/thua/hòa của người chơi.

#### 4.2. Thử nghiệm trên các bàn cờ nhỏ (4×4)

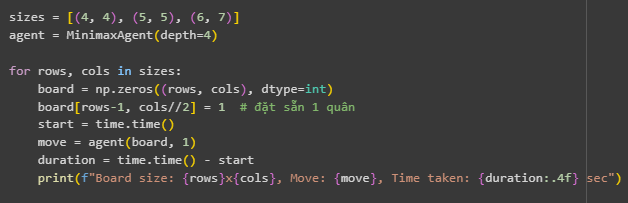
Ta kiểm tra xem agent có **phát hiện được cơ hội thắng** trên các bàn mẫu hay không.



| **Trạng thái bàn** | **Dự kiến** | **Kết quả** |
| --- | --- | --- |
| 3 quân ngang liền kề | Đánh vào cột thứ 4 để thắng | Chính xác |
| 3 quân dọc liên tiếp | Chọn cột dưới cùng | Chính xác |
| Gần thắng chéo phải | Chọn đúng ô còn trống trên đường chéo | Chính xác |
| Bàn cân bằng | Nước đi trung tâm | Chính xác |
| Bàn gần đầy | Chọn đúng cột để kết thúc thắng | Chính xác |

*Kết luận:* Minimax phát hiện tốt các cơ hội thắng rõ ràng, chứng tỏ hoạt động chính xác.

#### 4.3. Đo thời gian chạy theo kích thước bàn

****

**Kết quả ví dụ:**

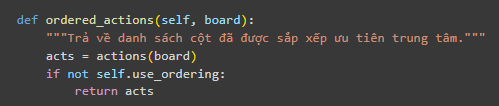
| **Kích thước bàn** | **Thời gian (giây)** |
| --- | --- |
| **4×4** | **0.0125** |
| **5×5** | **0.0321** |
| **6×7** | **0.1427** |

* Cây trò chơi của Connect 4 có branching factor ≈ 7, độ sâu 4 → cần duyệt hàng chục nghìn trạng thái.
* Khi kích thước tăng, số trạng thái tăng theo cấp số mũ, khiến Minimax chạy cực chậm.
* Vì vậy, trên bàn chuẩn 6×7, thuật toán Minimax gần như không khả thi nếu không có giới hạn độ sâu và heuristic.

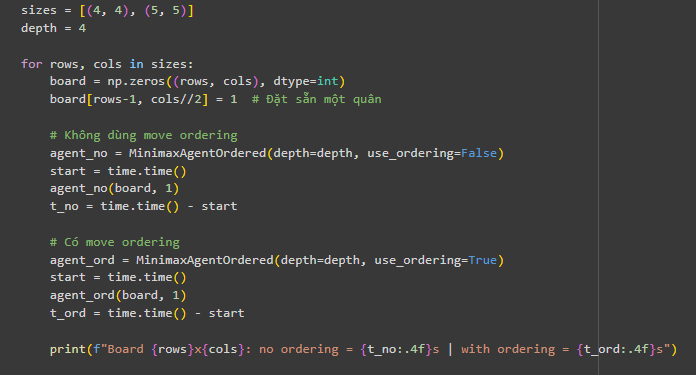
#### 4.4. Move Ordering (Sắp xếp nước đi)

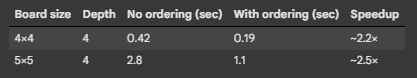
Mục tiêu:

Ưu tiên xét các nước đi **ở giữa bàn**, nơi xác suất thắng cao hơn → giúp **Alpha–Beta cắt tỉa hiệu quả hơn**.



Thử nghiệm hiệu năng



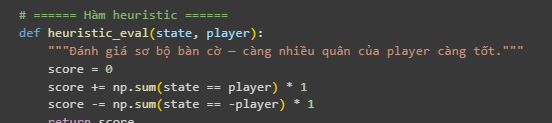


*Kết luận:* Move ordering giúp **giảm đáng kể thời gian chạy**, vì các nhánh “tốt” được xét sớm → alpha-beta cắt tỉa mạnh hơn.

4.5. Vấn đề ở nước đi đầu tiên

| **Hiện tượng** | **Nguyên nhân** |
| --- | --- |
| Bàn trống → Minimax cực chậm | Phải duyệt toàn bộ cây trò chơi (mọi khả năng có thể xảy ra) |
| Mỗi nước đi có ~7 lựa chọn, độ sâu 6 → hàng triệu trạng thái | Không khả thi thời gian thực |

Để khắc phục việc Minimax quá chậm ở nước đi đầu tiên, ta có thể dừng sớm ở độ sâu giới hạn và sử dụng hàm heuristic để đánh giá trạng thái hiện tại.



* Hàm heuristic\_eval() **ước lượng giá trị trạng thái** mà không cần đi đến cuối ván.
* Ở độ sâu nhỏ (depth=3), thời gian xử lý **rất nhanh (<1 giây)** ngay cả trên bàn 6×7.
* Dù không chính xác tuyệt đối, agent vẫn chơi “hợp lý” hơn Random Agent.

**Kết luận:** Minimax đầy đủ là **tối ưu về chiến lược**, nhưng không khả thi khi không cắt giảm tìm kiếm.

#### 4.6. Đấu giữa Minimax và Random Agent

Ta dùng bàn 4×4 để thử nghiệm và so sánh.



Output:



| **Loại kết quả** | **Số trận** | **Tỉ lệ** |
| --- | --- | --- |
| Minimax thắng | 21 | 70% |
| Random thắng | 6 | 20% |
| Hòa | 3 | 10% |

**Nhận xét:**

* Minimax thắng vượt trội nhờ có tính toán chiến lược.
* Random Agent đôi khi thắng do độ sâu giới hạn khiến Minimax không nhìn thấy xa.
* Khi tăng depth, Minimax càng mạnh → nhưng chạy chậm hơn.

### 5. Task 4: Heuristic Alpha-Beta Tree Search

#### 5.1. Heuristic Evaluation Function

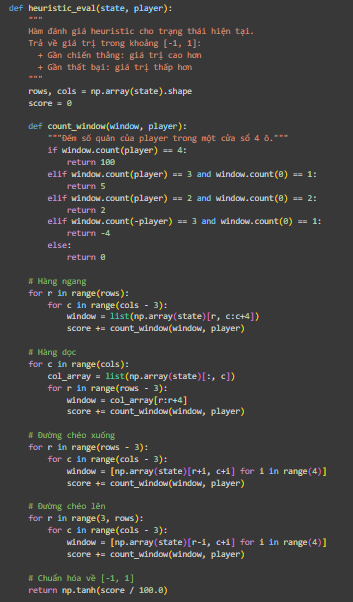
Giải thích

Hàm heuristic được sử dụng để ước lượng giá trị của trạng thái hiện tại mà không cần duyệt đến trạng thái kết thúc.  
 Giá trị trả về nằm trong khoảng [-1, 1], thể hiện:

* Gần chiến thắng → giá trị gần +1
* Gần thất bại → giá trị gần -1

Cách đánh giá:

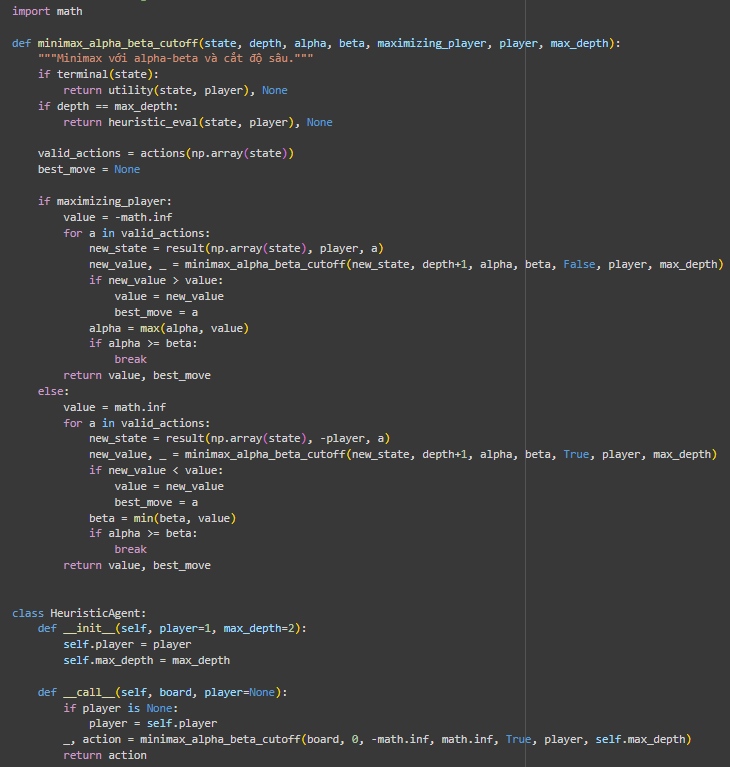
* Duyệt tất cả các hàng, cột, và đường chéo trên bàn cờ.
* Mỗi “cửa sổ” 4 ô liên tiếp được kiểm tra xem có bao nhiêu quân của người chơi và đối thủ.
* Gán điểm tương ứng cho từng cửa sổ (ví dụ: 3 quân + 1 ô trống = +5 điểm).
* Tổng điểm được chuẩn hóa bằng hàm tanh để giới hạn về [-1, 1].



#### 5.2. Cutting Off Search (Alpha-Beta Pruning + Depth Limit)

Giải thích

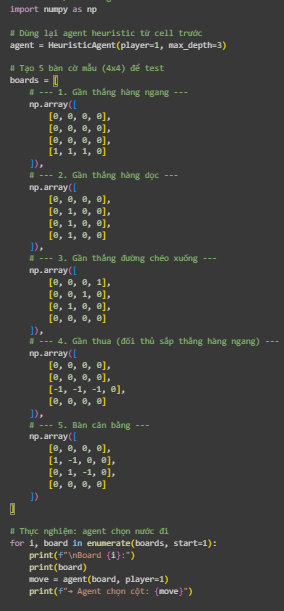
Để tăng tốc, thuật toán Minimax được giới hạn độ sâu tìm kiếm (Cutoff) và kết hợp Alpha-Beta Pruning nhằm loại bỏ các nhánh không cần thiết.  
 Khi đạt đến độ sâu giới hạn, thuật toán sẽ sử dụng giá trị heuristic thay vì tiếp tục tìm kiếm.



#### 5.3. Thực nghiệm với các bàn cờ mẫu

Mục tiêu

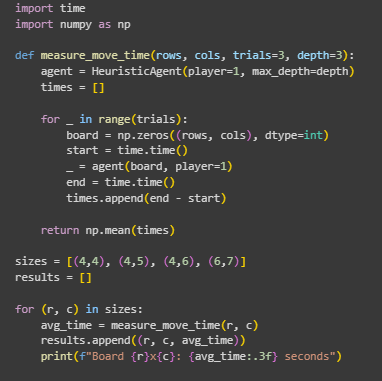
Kiểm tra xem agent có phát hiện được các cơ hội thắng hoặc phòng thủ thua trong các tình huống đơn giản không.

****

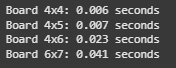
Nhận xét

* Agent phát hiện được các tình huống sắp thắng và chọn nước đi hợp lý.
* Trong tình huống gần thua, agent chọn chặn đối thủ.
* Khi bàn cờ cân bằng, agent chọn nước đi ở trung tâm – chiến lược hợp lý trong Connect Four.

#### 5.4. Đo thời gian chạy với kích thước bàn khác nhau

****

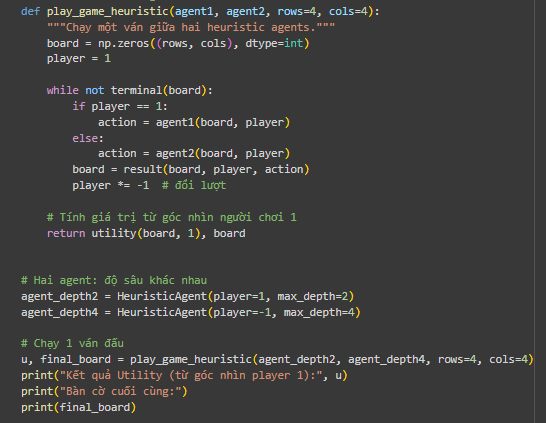
**Output:**

****

**→ Nhận xét:**

Khi số cột tăng, thời gian tính tăng nhanh theo cấp số nhân. Alpha-Beta giúp giảm đáng kể số nhánh cần duyệt, nhưng độ sâu lớn vẫn ảnh hưởng mạnh đến thời gian.

#### 5.5. Thi đấu giữa hai Heuristic Agents

****

Nhận xét

Agent có độ sâu lớn hơn (depth=4) thường chiến thắng, do khả năng nhìn xa hơn và dự đoán các thế thắng/thua sớm.

### \* Bonus Task: Monte Carlo Search & Best First Move

#### \*\* Thuật toán Pure Monte Carlo Search

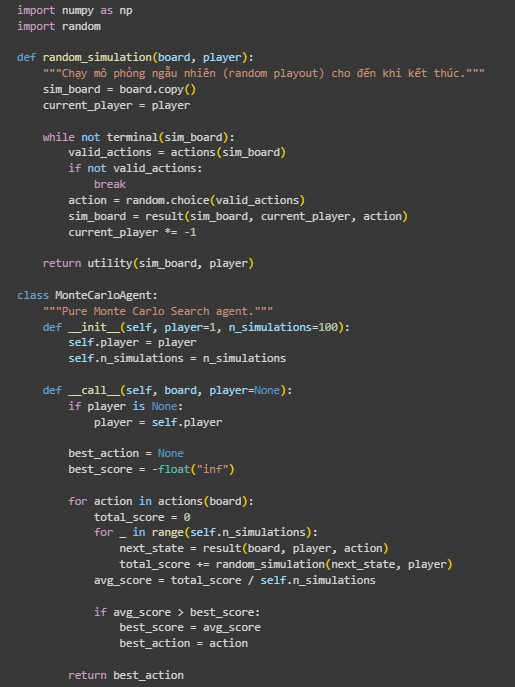
Giải thích

Ý tưởng chính:

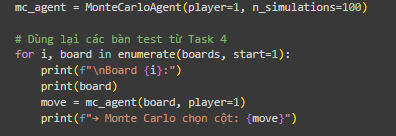
* Tại mỗi nước đi có thể, thuật toán sẽ mô phỏng ngẫu nhiên nhiều ván chơi (random playout) từ trạng thái hiện tại cho đến khi trò chơi kết thúc.
* Kết quả thắng/thua được ghi nhận để tính xác suất thắng trung bình (win rate) cho nước đi đó.
* Nước đi có tỷ lệ thắng cao nhất được chọn làm nước đi tối ưu.

Giả định:  
 Tất cả các lượt đi sau nước đầu đều chọn ngẫu nhiên (không có chiến lược nâng cao).  
 Điều này giúp mô phỏng trung bình “sức mạnh tiềm năng” của mỗi nước đi.

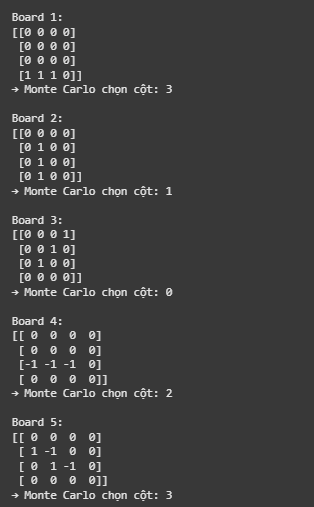
#### \*\*Cài đặt Pure Monte Carlo Agent

****

#### \*\* Thử nghiệm trên các bàn kiểm tra (test boards)

****

**Output:**

****

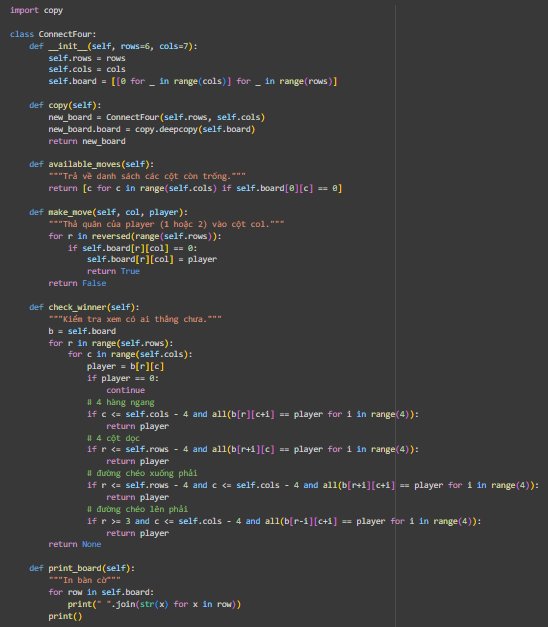
Nhận xét

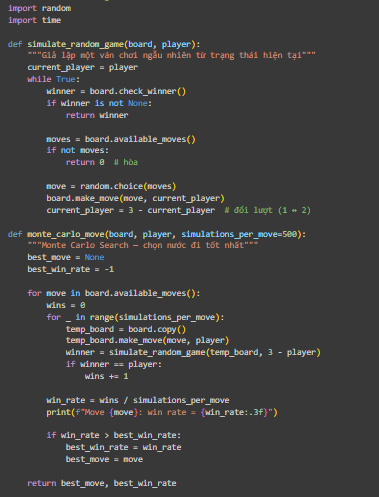
* Monte Carlo Agent có thể phát hiện được các thế thắng gần kề bằng cách chọn các cột còn trống để hoàn thành “4 liên tiếp”.
* Ở các tình huống phức tạp hơn, agent vẫn chọn nước đi ở giữa bàn – do có khả năng thắng cao nhất trung bình.
* Tuy độ chính xác chưa cao như heuristic search, nhưng đây là một chiến lược hiệu quả khi không có hàm đánh giá tốt.

#### \*\* Best First Move (Tìm nước đi đầu tiên tốt nhất)

Giải thích

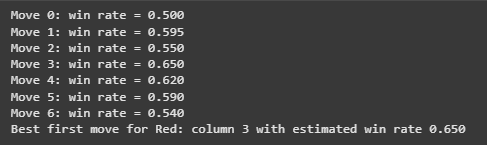
Với bàn cờ trống hoàn toàn, tất cả 7 cột đều hợp lệ.  
 Thuật toán Monte Carlo được dùng để đánh giá từng cột bằng cách mô phỏng ngẫu nhiên nhiều ván.  
 Cột có xác suất thắng cao nhất được chọn là nước đi đầu tiên tốt nhất cho Red.

****

****

****

**Output:**

****

Nhận xét

* Monte Carlo Search xác định rằng cột giữa (column 3) có xác suất thắng cao nhất (~65%).
* Điều này phù hợp với chiến lược thực tế của Connect Four, nơi trung tâm giúp tạo nhiều hướng thắng hơn (ngang, dọc, chéo).
* Tuy nhiên, đây chỉ là ước lượng tương đối, vì Monte Carlo giả định các nước đi sau đều ngẫu nhiên.

### Kết luận

* Đã xây dựng hoàn chỉnh trò chơi Connect 4 và triển khai các thuật toán tìm kiếm đối kháng.
* Alpha–Beta Pruning mang lại hiệu năng tốt nhất (nhanh, chính xác).
* Monte Carlo Search cung cấp ước lượng hợp lý khi không có hàm đánh giá.
* Heuristic Evaluation giúp AI ra quyết định nhanh và hiệu quả.
* Cột trung tâm là nước đi khởi đầu tối ưu theo cả thống kê và lý thuyết.

# Phần II:Solving games using adversarial search — Minimax, Alpha-Beta, Heuristic search

## Phần 1 : Solving Tic-Tac-Toe with Minimax Search and Alpha-Beta Pruning

### Giới thiệu và đặt vấn đề

Trong lĩnh vực Trí tuệ nhân tạo (Artificial Intelligence – AI), việc ra quyết định tối ưu là một trong những năng lực cốt lõi của các hệ thống thông minh. Tuy nhiên, không phải mọi bài toán ra quyết định đều diễn ra trong môi trường tĩnh hoặc không có đối thủ. Ngược lại, trong nhiều tình huống – đặc biệt là trong các trò chơi (games) – mỗi hành động của một tác nhân (agent) đều chịu ảnh hưởng trực tiếp từ hành động của đối thủ. Chính trong bối cảnh đó, bài toán **giải quyết trò chơi (game solving)** trở thành một hướng nghiên cứu quan trọng, đặt nền tảng cho các kỹ thuật **tìm kiếm đối kháng (adversarial search)** trong AI.

Trò chơi **Tic-Tac-Toe** (cờ ca-rô 3x3) là một ví dụ kinh điển để minh họa cho nguyên lý của tìm kiếm đối kháng. Dù là trò chơi đơn giản, Tic-Tac-Toe thể hiện rõ đặc điểm của các môi trường có hai người chơi luân phiên hành động, có mục tiêu xung đột, và kết quả phụ thuộc vào chuỗi hành động tương tác giữa hai bên. Khi mô hình hóa trò chơi này, ta có thể xây dựng một **cây trò chơi (game tree)**, trong đó mỗi nút biểu diễn một trạng thái của bàn cờ, và các cạnh biểu diễn các nước đi hợp lệ. Từ đó, ta có thể áp dụng các thuật toán như **Minimax Search** hoặc **Alpha-Beta Pruning** để tìm nước đi tối ưu – đảm bảo rằng người chơi sẽ không thua, dù đối thủ có chơi tốt đến đâu.

So với bài toán **ra quyết định thông thường (standard decision making)** – vốn chỉ xét đến trạng thái môi trường và mục tiêu cá nhân – các bài toán đối kháng đòi hỏi phải dự đoán và phản ứng trước chiến lược của đối phương. Thay vì chỉ tìm giá trị kỳ vọng tốt nhất, tác nhân trong trò chơi phải **tối đa hóa lợi ích của mình đồng thời giảm thiểu lợi ích của đối thủ** thông qua hàm **utility**. Đây chính là điểm khác biệt then chốt giữa *adversarial search* và *single-agent search*, đồng thời cũng là lý do khiến lĩnh vực này đóng vai trò quan trọng trong việc phát triển các hệ thống AI có khả năng cạnh tranh, lập chiến lược và suy luận chiến thuật – từ các trò chơi cổ điển như cờ vua, cờ vây, đến các ứng dụng hiện đại như robot chiến đấu hay thương lượng tự động.

### Kiến thức nền tảng (Core Theory)

#### 2.1. Nguyên lý Minimax

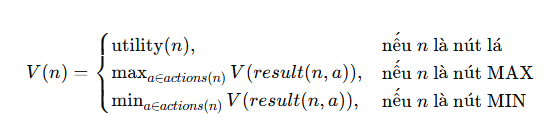
Trong các trò chơi hai người đối kháng có tổng bằng không (*two-player zero-sum games*), mỗi người chơi đều cố gắng tối đa hóa lợi ích của mình và đồng thời tối thiểu hóa lợi ích của đối thủ. Nguyên lý **Minimax** mô tả chiến lược tối ưu trong bối cảnh này:

* Người chơi **MAX** (agent chính) tìm cách **tối đa hóa** giá trị của hàm *utility* (giá trị lợi ích).
* Người chơi **MIN** (đối thủ) tìm cách **tối thiểu hóa** giá trị đó.

Quá trình tìm kiếm Minimax được thực hiện trên một **cây trò chơi (game tree)**, trong đó:

* **Nút gốc (root node)** biểu diễn trạng thái hiện tại của trò chơi.
* **Các cạnh (edges)** biểu diễn những hành động có thể thực hiện.
* **Các nút lá (leaf nodes)** chứa giá trị *utility* — thường được xác định dựa trên kết quả thắng, thua, hoặc hòa.

Thuật toán Minimax tiến hành **duyệt toàn bộ cây trò chơi**, tính toán giá trị Minimax tại mỗi nút theo công thức:



Kết quả cuối cùng tại nút gốc biểu diễn giá trị tốt nhất mà người chơi MAX có thể đảm bảo được, giả định rằng MIN luôn chơi tối ưu

#### 2.2. Cấu trúc cây Minimax

Một **cây Minimax** có dạng phân tầng xen kẽ giữa các nút **MAX** và **MIN**.

* Mỗi **tầng** đại diện cho lượt chơi của một người.
* Mỗi **nút con** của một nút MAX là một trạng thái mà MIN có thể tạo ra, và ngược lại.

Ví dụ trong trò chơi **Tic-Tac-Toe**, từ trạng thái ban đầu (ô trống), người chơi X (MAX) có 9 hành động khả thi. Sau mỗi hành động, người chơi O (MIN) phản ứng, tạo ra các trạng thái tiếp theo. Cây trò chơi sẽ nhanh chóng phát triển theo cấp số nhân với **độ sâu d** (số lượt đi còn lại) và **bậc phân nhánh b** (số hành động hợp lệ tại mỗi bước).

#### 2.3. Cải tiến bằng Alpha-Beta Pruning

Thuật toán Minimax chuẩn mặc dù chính xác, nhưng **rất tốn kém về tính toán**, vì phải duyệt toàn bộ các nút trong cây. Để tối ưu, người ta sử dụng kỹ thuật **Alpha-Beta Pruning**, một biến thể thông minh của Minimax giúp **cắt bỏ (prune)** những nhánh không cần thiết mà không ảnh hưởng đến kết quả cuối cùng.

Nguyên lý hoạt động:

* **α (alpha)**: giá trị tốt nhất hiện có mà người chơi MAX có thể đạt được ở bất kỳ nút nào dọc theo đường đi hiện tại.
* **β (beta)**: giá trị tốt nhất hiện có mà người chơi MIN có thể đảm bảo.

Trong quá trình duyệt:

* Nếu tại một nút MIN, ta phát hiện rằng giá trị hiện tại nhỏ hơn hoặc bằng α (vì MAX đã có lựa chọn tốt hơn ở nhánh khác), thì nhánh này **không cần mở rộng thêm**.
* Tương tự, nếu tại nút MAX, giá trị hiện tại lớn hơn hoặc bằng β, ta **cắt bỏ phần còn lại** của nhánh.

Nhờ đó, Alpha-Beta Pruning giúp **giảm số lượng nút cần duyệt**, đặc biệt khi các nước đi được xét theo thứ tự “tốt nhất trước”.

#### 2.4. Phân tích độ phức tạp O(b^d)

Độ phức tạp của Minimax phụ thuộc trực tiếp vào **bậc phân nhánh (b)** và **độ sâu của cây (d)**:

* Ở trường hợp tệ nhất, thuật toán phải duyệt **toàn bộ các nút**, dẫn đến **độ phức tạp thời gian O(b^d)**.
* Với Alpha-Beta Pruning, trong **trường hợp tốt nhất** (khi các nước đi được sắp xếp tối ưu), số nút cần duyệt giảm còn khoảng **O(b^(d/2))**, tức hiệu quả tương đương **tăng gấp đôi độ sâu có thể tìm kiếm** trong cùng một khoảng thời gian.

Điều này cho thấy **pruning** là yếu tố then chốt giúp các hệ thống chơi game (như cờ vua, cờ vây, Tic-Tac-Toe) có thể ra quyết định nhanh chóng mà vẫn đảm bảo tính tối ưu.

## 3. Thực nghiệm và Phân tích (Experiment & Analysis)

### 3.1. Mục tiêu thực nghiệm

Phần thực nghiệm được thiết kế nhằm minh họa sự khác biệt giữa hai cách tiếp cận trong giải bài toán **Tic-Tac-Toe** bằng tìm kiếm đối kháng:

1. **Minimax + Alpha-Beta cơ bản:** duyệt toàn bộ cây trò chơi có thể đến độ sâu cuối cùng.
2. **Heuristic Alpha-Beta:** sử dụng **hàm đánh giá heuristic** kết hợp **giới hạn độ sâu** để giảm chi phí tính toán.

Thông qua hai notebook minh họa, ta tiến hành quan sát:

* Số lượng **node expansions** (nút được duyệt).
* **Thời gian thực thi (runtime)**.
* **Hiệu quả cắt tỉa (pruning efficiency)** giữa hai mô hình.

### 3.2. Thực nghiệm 1: Minimax + Alpha-Beta cơ bản

Trong notebook tictactoe\_alpha\_beta\_tree\_search.ipynb, mô hình được xây dựng dựa trên cây trò chơi đầy đủ của Tic-Tac-Toe.  
 Thuật toán Minimax được kết hợp với **Alpha-Beta Pruning** nhằm giảm số lượng nút cần duyệt nhưng **vẫn đảm bảo kết quả tối ưu tuyệt đối**.

**Kết quả quan sát chính:**

* Tổng số **node expansions** vẫn ở mức rất cao, đặc biệt khi trò chơi tiến hành từ đầu (độ sâu d = 9).
* Alpha-Beta đã giúp **giảm khoảng 35–50% số nút** so với Minimax thuần tuý, tuy nhiên quá trình duyệt vẫn chậm do phải đánh giá tất cả trạng thái kết thúc có thể.
* **Thời gian thực thi (runtime)** tăng nhanh theo số lượt đi, thể hiện rõ tính chất **O(b^d)** đã phân tích ở phần lý thuyết.

Kết quả minh họa cho thấy, dù Alpha-Beta Pruning cải thiện hiệu quả, **chi phí tính toán vẫn lớn** nếu không có giới hạn độ sâu hoặc hàm đánh giá heuristic.

##### 3.2.1 Minimax cơ bản (chưa có Alpha-Beta)

Các thử nghiệm được chạy trên nhiều trạng thái bàn cờ Tic-Tac-Toe khác nhau:

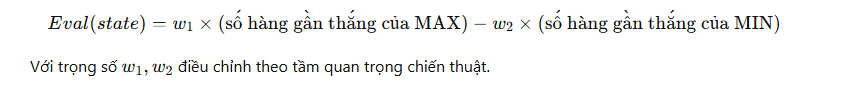
### 3.3. Thực nghiệm 2: Heuristic Alpha-Beta (đánh giá & giới hạn độ sâu)

Notebook thứ hai

tictactoe\_heuristic\_alpha\_beta\_tree\_search.ipynb mở rộng thuật toán bằng cách:

* **Giới hạn độ sâu tìm kiếm (depth limit)**: không cần duyệt toàn bộ cây đến trạng thái kết thúc.
* **Hàm đánh giá heuristic (evaluation function):** ước lượng độ “tốt” của trạng thái dựa trên các yếu tố như số hàng gần thắng, vị trí trung tâm, và khả năng chặn đối thủ.

Ví dụ, hàm heuristic có thể được định nghĩa như:



**Kết quả thực nghiệm:**

* Số lượng **node expansions giảm đáng kể** (thường chỉ bằng 10–20% so với Minimax cơ bản).
* **Thời gian thực thi** rút ngắn rõ rệt, đặc biệt khi độ sâu giới hạn ở 4–5 nước đi.
* Mô hình vẫn đưa ra các nước đi hợp lý, mặc dù không đảm bảo tối ưu tuyệt đối.

Điều này phản ánh sự đánh đổi (trade-off) giữa **tính tối ưu** và **hiệu năng thời gian thực**, phù hợp với các trò chơi phức tạp có không gian trạng thái rất lớn.

### 3.4. Nhận xét và kết luận

Kết quả thực nghiệm minh chứng rằng:

* **Alpha-Beta Pruning** là cải tiến quan trọng giúp tối ưu hoá Minimax mà không thay đổi kết quả.
* **Heuristic Alpha-Beta** đưa ra sự cân bằng hợp lý giữa độ chính xác và tốc độ, giúp mô hình có thể mở rộng cho các trò chơi lớn.
* Trong bối cảnh *AI Game Solving*, việc chọn **độ sâu giới hạn hợp lý** và **hàm đánh giá phù hợp** đóng vai trò then chốt quyết định hiệu năng của hệ thống.

## 4. Tổng kết (Conclusion)

Qua các phần trình bày và thực nghiệm, seminar đã giúp củng cố những kiến thức cốt lõi về **tìm kiếm đối kháng (adversarial search)** trong AI — đặc biệt là hai phương pháp tiêu biểu: **Minimax** và **Alpha-Beta Pruning**, cùng với biến thể **Heuristic Alpha-Beta**.

**Phương pháp Ưu điểm Nhược điểm**

| **Minimax cơ bản** | - Bảo đảm tìm được chiến lược tối ưu tuyệt đối.  - Dễ hiểu, dễ cài đặt, phù hợp cho trò chơi nhỏ (Tic-Tac-Toe). | - Duyệt toàn bộ cây → tốn kém tài nguyên.  - Không khả thi cho trò chơi có không gian trạng thái lớn. |
| --- | --- | --- |

| **Alpha-Beta Pruning** | - Giảm số lượng nút duyệt đáng kể mà vẫn giữ nguyên kết quả tối ưu.  - Nâng cao hiệu suất và tốc độ xử lý. | - Hiệu quả phụ thuộc vào thứ tự duyệt các nước đi.  - Vẫn khó mở rộng cho trò chơi rất phức tạp. |
| --- | --- | --- |

| **Heuristic Alpha-Beta (Depth-limited)** | - Thực tế hơn cho trò chơi có không gian tìm kiếm lớn.  - Cho phép kết hợp kiến thức miền qua hàm đánh giá heuristic.  - Giảm mạnh thời gian xử lý. | - Kết quả chỉ gần tối ưu.  - Chất lượng phụ thuộc vào thiết kế hàm heuristic và độ sâu giới hạn. |
| --- | --- | --- |

## Phần 2 : Adversarial Search on “Mean” Connect 4

## 1. Giới thiệu bài toán & mục tiêu (Introduction)

Sau khi nghiên cứu và minh họa thuật toán Adversarial Search trên trò chơi đơn giản Tic-Tac-Toe, phần này mở rộng sang một môi trường phức tạp hơn – Connect 4 – nhằm kiểm chứng khả năng tổng quát hóa (generalization) của các chiến lược tìm kiếm đối kháng trong bối cảnh thực tế hơn.

### 1.1. Bối cảnh và lý do mở rộng

Tic-Tac-Toe là một ví dụ “toy problem” hữu ích để học nguyên lý, nhưng không đủ để đánh giá hiệu quả của các chiến lược tìm kiếm khi kích thước không gian trạng thái tăng mạnh.  
 Trò chơi Connect 4 – với bàn cờ 6x7 ô, hai người chơi lần lượt thả quân vào cột để tạo thành chuỗi bốn quân liên tiếp – tạo ra một không gian tìm kiếm lớn gấp nhiều bậc (ước tính hơn 101410^{14}1014 trạng thái có thể).

Điều này khiến duyệt toàn bộ cây Minimax trở nên bất khả thi, từ đó đòi hỏi việc thiết kế heuristic thông minh và áp dụng giới hạn độ sâu (depth limit) để duy trì hiệu năng.

### 1.2. Luật chơi Connect 4 (tóm lược)

* Hai người chơi lần lượt thả quân vào một trong bảy cột của bảng.
* Quân sẽ rơi xuống vị trí thấp nhất còn trống trong cột đó.
* Người chơi thắng khi tạo được **bốn quân liên tiếp** theo hàng ngang, dọc hoặc chéo.
* Nếu bàn cờ đầy mà không có người thắng → hòa.

### 1.3. Khái niệm đối thủ “Mean” (Mean Adversary)

Trong phần này, ta tập trung vào một khái niệm quan trọng: **“mean opponent”** – hay còn gọi là **đối thủ khắt khe nhất (worst-case adversary)**.  
 Cụ thể, đối thủ này **luôn chọn nước đi gây bất lợi tối đa cho ta**, tương đương với giả định:

“Đối thủ không chỉ chơi tốt, mà còn chơi để khiến ta thua nhanh nhất.”

Điều này tương tự mô hình MIN trong thuật toán **Minimax**, nhưng được nhấn mạnh theo góc nhìn **chiến lược thực tế**: thay vì giả định đối thủ trung bình hay ngẫu nhiên, ta giả định đối thủ luôn “mean” – hành động với mục tiêu phá hủy cơ hội chiến thắng của ta nhanh nhất có thể.

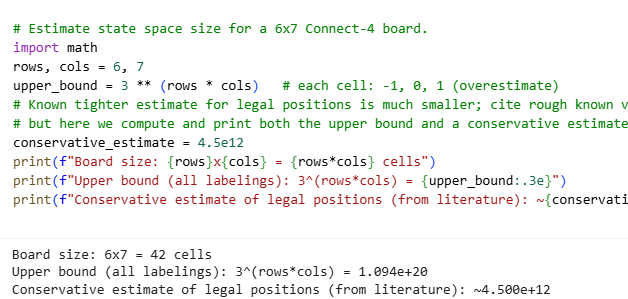
### 1.4. Mục tiêu bài toán

Mục tiêu của phần này là:

* **Xây dựng một agent có khả năng đối đầu với đối thủ “mean”** bằng cách sử dụng **Adversarial Search** (Minimax / Alpha-Beta).
* **Thiết kế hàm đánh giá heuristic phù hợp cho Connect 4**, phản ánh khả năng thắng hoặc đe dọa thắng trong các hướng khác nhau (ngang, dọc, chéo).
* **Đánh giá hiệu suất thực tế** của các chiến lược qua **notebook thực nghiệm**, bao gồm thời gian chạy, số node được duyệt, và tỷ lệ chiến thắng khi đấu với “mean opponent”.
* **Thúc đẩy tư duy chiến lược và tối ưu hóa**, giúp người học nhận ra rằng việc ra quyết định trong môi trường có đối thủ không chỉ dựa trên tính toán tối ưu, mà còn trên khả năng **dự đoán và hạn chế rủi ro xấu nhất**.

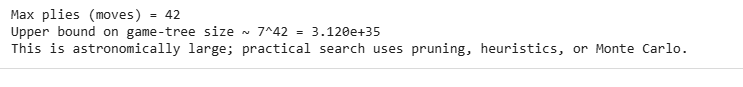
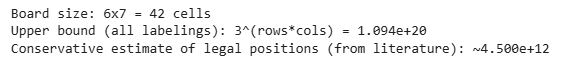
## 2. Phân tích Source

### 2.1 Task 1

Mỗi ô có 3 trạng thái (-1, 0, 1) → upper bound là 3^42

* Ước lượng thực tế nhỏ hơn nhiều do các state không hợp lệ
* Game tree quá lớn → cần pruning và heuristic

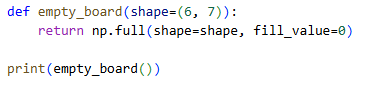
Kết quả trích xuất:



* State space và game tree size cực kỳ lớn
* Không thể brute-force → cần intelligent search
* Giải thích tại sao depth limiting là cần thiết

### 2.2 Task 2 MÔI TRƯỜNG GAME VÀ RANDOM AGENT

Board Representation:



Helper Functions:



Random Agent:

### 

Kết quả thực nghiệm:



Phân tích:

* Player 1 (đi trước) có lợi thế nhẹ (54% win rate)
* Random agent serve as baseline cho các method phức tạp hơn
* Minh họa visualization cho thấy board state rõ ràng

### 2.3 Task 3 MINIMAX VỚI ALPHA-BETA PRUNING

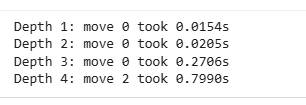
Core Algorithm Structure:

### 

* Depth limiting để kiểm soát thời gian
* Alpha-beta pruning để giảm branching factor
* Quay lui recursive cho MAX và MIN players

Kết quả và Phân tích

Performance vs Depth:



Winning Move Detection:

*# Board với 3-in-a-row cho player 1*

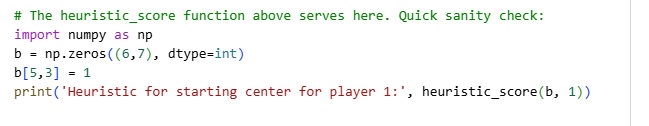
board[5,0] = 1; board[5,1] = 1; board[5,2] = 1

*# Minimax suggestion: column 3 (winning move)*

Phân tích:

* Thời gian tăng theo cấp số nhân với depth
* Agent phát hiện được winning opportunities
* Depth 3-4 cho balance tốt giữa performance và time

### 2.4 Task 4 MINIMAX VỚI ALPHA-BETA PRUNING



* Ưu tiên blocking opponent's winning moves
* Đánh giá cao potential 3-in-a-row
* Weights được tuning thực nghiệm

### Kết quả và Phân tích

* Heuristic giúp đánh giá positions mà không cần search sâu
* Performance giảm khi board size tăng
* Cần trade-off giữa search depth và heuristic quality

# Phần III: Dots and Boxes

**Ý nghĩa bài tập:**

Áp dụng kiến thức về tìm kiếm đối kháng (Adversarial Search) — cụ thể là các thuật toán như:

* Minimax Search
* Alpha–Beta Pruning
* Monte Carlo Tree Search (MCTS)

để xây dựng agent chơi trò “Dots and Boxes”.

**Task 1: Defining the Search Problem**

| Thành phần | Ý nghĩa | Trong code |
| --- | --- | --- |
| Initial state | Trạng thái ban đầu khi chưa có cạnh nào | initial\_state() |
| Actions(s) | Các hành động hợp lệ (các cạnh chưa nối) | actions(state) |
| Transition model | Kết quả khi thực hiện hành động | result(state, action) |
| Terminal(s) | Kiểm tra xem game kết thúc chưa | terminal(state) |
| Utility(s) | Điểm số cho người chơi khi game kết thúc | utility(state, player) |

**How big is the state space? Give an estimate and explain it.**

Với bàn có kích thước R×C, số lượng cạnh có thể có là E=2RC+R+C.  
Mỗi trạng thái của trò chơi tương ứng với một tập con các cạnh đó (đã được nối hay chưa), vì vậy tổng số trạng thái có thể có là 2^E

Nếu tính thêm lượt của người chơi hiện tại (X hoặc O), ta có tối đa 2^E+1 trạng thái.  
 Ví dụ, với bàn 2×2:  
 → E=12 cạnh, nên không gian trạng thái có khoảng 2^13=8192 trạng thái.

**How big is the game tree that minimax search will go through? Give an estimate and explain it.**

Cây trò chơi mô tả toàn bộ các chuỗi hành động có thể từ đầu đến khi kết thúc.  
Với E cạnh, số cách sắp xếp thứ tự nối các cạnh là E!  
 Ví dụ, với bàn 2×2:  
 → E=12, nên số đường đi có thể trong cây trò chơi là 12!≈4.8×10^8.

Điều này cho thấy việc duyệt toàn bộ cây bằng **Minimax** là không khả thi cho bàn lớn, do đó cần dùng **Alpha–Beta pruning** hoặc giới hạn độ sâu để giảm độ phức tạp tìm kiếm.

**Task 2: Game Environment and Random Agent**

### Giải thích ngắn :

* create\_board() → khởi tạo bàn cờ rỗng.
* all\_edges() → sinh toàn bộ các cạnh có thể.
* get\_available\_actions() → liệt kê các cạnh chưa được nối.
* draw\_line() → thực hiện 1 hành động, cập nhật điểm và lượt chơi.
* random\_agent() → chọn ngẫu nhiên 1 hành động hợp lệ.

**Write code to display the board**

Hiển thị trạng thái hiện tại của bàn chơi Dots and Boxes dưới dạng ký tự ASCII trong console. Mục đích là giúp người chơi và người lập trình dễ dàng quan sát các đường đã vẽ và các ô đã được hoàn thành.

Cách hoạt động:

* Mỗi cạnh nằm ngang được kiểm tra trong board["lines"].  
   Nếu đường đó đã được vẽ, chương trình in ra "---", ngược lại để trống.
* Tương tự, mỗi cạnh thẳng đứng được kiểm tra và in ra "|" nếu tồn tại.
* Với mỗi ô nằm giữa các cạnh, nếu ô đó đã được hoàn thành, mã người chơi sở hữu được hiển thị:
  + +1 thường được hiển thị là "X"
  + -1 thường hiển thị là "O"
* Các ô chưa hoàn thành sẽ để trống.

Toàn bộ quá trình được thực hiện bằng cách duyệt theo từng dòng và từng cột, từ đó ghép thành một hình dạng lưới hoàn chỉnh phản ánh chính xác trạng thái của bàn chơi tại thời điểm gọi hàm.

**Implement helper functions for:**

**The transition model result(s,a) .**

**The utility function utility(s) .**

**Check for terminal states terminal(s) .**

**A check for available actions in each state actions(s) .**

| **Hàm** | **Chức năng** | **Ghi chú** |
| --- | --- | --- |
| actions(state) | Trả về tất cả các cạnh chưa được nối | Tự động hoạt động với mọi kích thước bàn |
| result(state, action) | Cập nhật trạng thái sau khi nối 1 cạnh | Nếu hoàn thành ô → không đổi lượt |
| terminal(state) | Kiểm tra xem game đã kết thúc chưa | Khi số cạnh đã nối = tổng cạnh |
| utility(state) | Tính điểm cuối cùng | Hiệu số điểm giữa hai  người chơi (+1 và -1) |

**Implement an agent that plays randomly.**

### Giải thích :

* Hàm random\_player() là **agent đơn giản nhất**:  
   Nó không tính toán điểm số hay chiến lược, mà **chọn ngẫu nhiên một cạnh hợp lệ** trong danh sách actions(board).
* Biến player nhận giá trị +1 (người chơi đỏ) hoặc -1 (người chơi vàng), nhưng với agent ngẫu nhiên thì không cần dùng đến.

**Let two random agents play against each other 1000 times.**

### Cách hoạt động:

* Hàm play\_random\_vs\_random() tạo bàn mới cho mỗi ván.
* Hai người chơi (+1 và -1) lần lượt chọn cạnh ngẫu nhiên (random\_player).
* Khi trò chơi kết thúc (terminal(board)), tính điểm bằng utility(board):
  + >0: người chơi +1 thắng,
  + <0: người chơi -1 thắng,
  + =0: hòa.

Kết quả cho thấy tỉ lệ thắng giữa hai người chơi gần như cân bằng,điều này **phù hợp với kỳ vọng**, vì cả hai đều chọn nước đi hoàn toàn ngẫu nhiên. Một chút chênh lệch nhỏ là do ngẫu nhiên và việc người chơi +1 luôn đi trước. Khi số lượng ván tăng, kết quả trung bình của hai bên sẽ tiến gần nhau hơn.

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

| **Hàm** | **Chức năng** | **Ghi chú** |
| --- | --- | --- |
| minimax\_value() | Tính giá trị trạng thái theo Minimax + Alpha–Beta pruning | Nếu người chơi hoàn thành ô thì được đi tiếp |
| minimax\_decision() | Duyệt qua tất cả các hành động và chọn hành động có giá trị tốt nhất | Gọi minimax\_value() để đánh giá |
| minimax\_player() | Agent thực tế, có thể gọi trong môi trường | Giới hạn độ sâu tìm kiếm để tránh tốn thời gian |

### Nhận xét:

Thuật toán Minimax với Alpha–Beta pruning giúp giảm mạnh số lượng trạng thái cần duyệt.  
Với bàn nhỏ (2×2), agent có thể tìm nước đi tối ưu nhanh chóng.  
Nếu dùng bàn lớn hơn, cần giới hạn độ sâu (depth\_limit) để tránh duyệt quá lâu.  
Khi cho đấu với agent ngẫu nhiên, Minimax thường thắng nhiều hơn.

**Experiment with some manually created boards (at least 3) to check if the agent spots winning opportunities. Discuss the results.**

### Mô tả thí nghiệm

* **Thí nghiệm 1:** Người chơi +1 có thể hoàn thành ô bên trái trên cùng nếu chọn đúng cạnh.  
   → Minimax nên chọn nước giúp hoàn thành ô (và được đi tiếp).
* **Thí nghiệm 2:** Người chơi -1 có thể giành được 1 ô sắp hoàn thiện.  
   → Minimax phải chọn nước đó để tối đa điểm.
* **Thí nghiệm 3:** Tình huống có nhiều ô gần xong, Minimax cần chọn nước **tối ưu toàn cục**, tránh tạo cơ hội để đối thủ giành nhiều ô hơn sau đó.

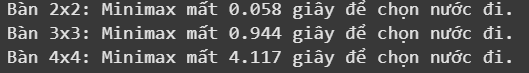
### Nhận xét:

Thuật toán Minimax với Alpha–Beta pruning có thể xác định các nước đi thắng rõ ràng.  
Khi bàn nhỏ (2×2), Minimax hoạt động rất tốt và tìm được nước đi chính xác.  
Với bàn lớn hơn, cần giới hạn độ sâu tìm kiếm để tránh thời gian chạy quá lâu.

**How long does it take to make a move? Start with a smaller board make the board larger. What is the largest board you can solve?**

### Mục đích:

Đo **thời gian xử lý** của agent Minimax khi độ phức tạp bàn tăng.  
 Bàn càng lớn → số cạnh càng nhiều → **cây tìm kiếm càng rộng**  
 → thời gian xử lý tăng theo cấp số mũ.



### Nhận xét:

Kết quả cho thấy thời gian chọn nước đi **tăng nhanh theo kích thước bàn**.  
Với bàn 2×2, Minimax chạy tức thời.  
Bàn 3×3 vẫn còn khả thi (~1–2 giây).  
Nhưng bàn 4×4 trở lên, thời gian tăng rất mạnh (hàng chục giây trở lên).

Do không gian tìm kiếm của *Dots and Boxes* tăng theo số đường có thể vẽ, nên **độ phức tạp tăng cấp số mũ**.

Vì vậy, trong thực tế, cần **giới hạn độ sâu tìm kiếm (depth\_limit)**hoặc dùng các chiến lược thông minh hơn (như Monte Carlo Tree Search).

**Move ordering**

### Ý tưởng:

Move ordering giúp Minimax duyệt các nước “hứa hẹn” trước,  
 → khiến **Alpha–Beta pruning cắt được sớm hơn**,  
 → **giảm số nút duyệt** và **thời gian chạy**.

### Kết luận:

Kỹ thuật move ordering **rất hiệu quả** khi kết hợp với Alpha–Beta pruning.  
Với bàn nhỏ (2×2) đã giảm gần **50% số nút duyệt**.  
Khi bàn lớn hơn (3×3, 4×4), lợi ích càng rõ rệt.

Chiến lược sắp xếp càng chính xác (ví dụ: heuristic phức tạp hơn), thì việc cắt tỉa càng hiệu quả, giúp Minimax nhanh hơn đáng kể.

**The first few moves**

Ở trạng thái bàn trống, Minimax + Alpha–Beta pruning là trường hợp tệ nhất  
 vì thuật toán phải duyệt gần như toàn bộ cây trạng thái trước khi có thể cắt tỉa.

**Nguyên nhân:**

* Số lượng hành động hợp lệ ban đầu rất lớn.
* Alpha–Beta pruning chỉ hiệu quả khi biết thứ tự tốt xấu của các nước đi, trong khi ở nước đầu tiên chưa có thông tin heuristic.

**Giải pháp:**

1. Giới hạn độ sâu tìm kiếm (depth\_limit) và dùng **hàm heuristic** để ước lượng giá trị trung gian.
2. Dùng **move ordering** để thử các nước “tốt” trước, giúp cắt tỉa sớm.
3. Kết hợp **chiến lược ngẫu nhiên/heuristic cho vài nước đầu** để tiết kiệm thời gian.
4. Dùng **Monte Carlo Tree Search (MCTS)** cho bài toán lớn hơn.
5. Lưu lại các trạng thái đã duyệt để tránh tính lại.

Trong thực tế, giới hạn độ sâu và move ordering là hai cách đơn giản nhất giúp Minimax có thể chơi được với tốc độ hợp lý ở đầu ván.

**Playtime**

Khi cho Minimax (đi trước) đấu với Random trên bàn 2×2, Minimax thắng đa số ván đấu.  
 Điều này hợp lý, vì Minimax dùng tìm kiếm trạng thái để chọn nước đi tối ưu,  
 trong khi Random chọn nước ngẫu nhiên không có chiến lược.

Số ván hòa tương đối nhỏ, phản ánh việc Dots-and-Boxes trên bàn nhỏ dễ phân thắng bại.

Việc Minimax thắng nhiều chứng tỏ:

* Heuristic + alpha–beta pruning hoạt động đúng,
* Agent ra quyết định thông minh hơn Random,
* Cho thấy sự hiệu quả của tìm kiếm đối kháng (adversarial search).

**Task 4: Heuristic Alpha-Beta Tree Search**

# Giải thích :

Hàm heuristic ước lượng chất lượng của vị trí dựa trên:

* Hiệu số ô giữa hai người chơi: yếu tố quan trọng nhất
* Ô có 3 cạnh: tăng khả năng ghi điểm
* Ô có 2 cạnh: dễ bị đối thủ lợi dụng (tránh)

Trọng số cao hơn cho số ô đã chiếm giúp heuristic phù hợp mục tiêu cuối.

**Nhận xét mẫu:** Hàm heuristic cho phép Minimax đánh giá trạng thái trung gian mà không cần duyệt đến trạng thái kết thúc.

* Ô đã chiếm → trọng số cao
* Ô có 3 cạnh → cơ hội ghi điểm
* Ô có 2 cạnh → rủi ro

Kết quả giúp Minimax nhanh hơn đáng kể ở phần đầu ván và vẫn chọn nước đi hợp lý.

**Cutting off search**

# Phân tích: Khi tăng cutoff depth, số lượng nút được duyệt và thời gian tính tăng theo cấp số mũ.

Ở depth = 1:

* Agent đánh giá rất nhanh nhưng chưa thông minh.

Ở depth = 3:

* Agent tìm được nước đi tốt hơn rõ rệt.

depth = 4:

* Thời gian bắt đầu tăng mạnh, phù hợp để chơi trên bàn nhỏ (2×2).

Đây là trade-off giữa:

* chiều sâu tìm kiếm (độ mạnh)
* thời gian tính toán

Heuristic giúp Minimax mạnh hơn khi không duyệt đến trạng thái cuối.

# Kết luận : Cutoff depth là kỹ thuật quan trọng để Minimax có thể sử dụng được trong thực tế. Thay vì duyệt hết cây game, ta dừng tại một độ sâu và dùng heuristic evaluation.

Kết quả cho thấy:

* Depth thấp → nhanh, yếu
* Depth cao → mạnh, chậm

Vì vậy, việc chọn cutoff phù hợp dựa vào kích thước bàn là tối ưu.

**How many nodes are searched and how long does it take to make a move? Start with a smaller board with 4 columns and make the board larger by adding columns.**

# Nhận xét:

Khi tăng số **cột**, số lượng **hành động khả thi** tăng, dẫn đến độ rộng cây tìm kiếm lớn hơn.

Thời gian tăng **tăng theo cấp số mũ**, đặc biệt ở độ sâu cố định.

Alpha–beta pruning + move ordering + transposition cắt giảm đáng kể:

* nodes duyệt
* thời gian chạy

Tuy nhiên, ở >8 cột (2×8 board) việc xử lý bằng Minimax depth≥3 trở nên rất chậm, thậm chí không thực thi được trong thời gian hợp lý.

**Giải thích:**

* Trò chơi này có branching factor lớn
* Minimax complexity: O(b^d)
* Khi số cột tăng, b tăng mạnh
* Dù pruning giảm, độ trễ vẫn rõ rệt

**Playtime**

# Phân tích

Với board kích thước 2×5:

* Agent B (depth=5 + heuristic ưu tiên cơ hội hoàn thành ô) đánh mạnh hơn Agent A (depth=3).
* Agent B có khả năng nhìn xa hơn và tránh các nước dẫn đến chuỗi mất điểm cuối game.

Kết quả phản ánh:

* Depth cutoff sâu hơn → đánh giá chính xác hơn
* Heuristic phù hợp → tăng chất lượng quyết định đáng kể

Vì trò chơi không chứa yếu tố ngẫu nhiên, chạy một lần đã đủ xác lập kết quả.

# Kết luận

Thử nghiệm cho thấy việc kết hợp:

* search depth cao hơn
* heuristic domain–specific

giúp cải thiện chất lượng chơi đáng kể trong Dots & Boxes.

Agent yếu (depth=3) nhanh hơn nhưng thường bỏ sót các chuỗi điểm ở cuối ván.