ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

BÁO CÁO BÀI TẬP LỚN MÔN TRÍ TUỆ NHÂN TẠO

(Bot giải quyết bài toán Connect 4)

Danh sách thành viên Nhóm 5

STT	Mã số sinh viên	Họ và tên	Ghi chú
1	23021620	Thái Khắc Mạnh	
2	23021664	Nguyễn Văn Phúc	
3	23021720	Trần Duy Thành	
4	23021580	Thiều Quang Huy	

Hà Nội, tháng 5/2025

Mục lục

I. Giới thiệu

II. Thiết kế chương trình

1. Thiết kế tổng quan
2. Tổng quan thuật toán và kỹ thuật chính

III. Thuật toán sử dụng

1. Negamax và Alpha-Beta Pruning
    1.1. Negamax
    1.2. Alpha-Beta Pruning
    1.3. Triển khai thuật toán trong mã nguồn
2. Heuristic
    2.1. Mục đích và vai trò
    2.2. Phân tích mã nguồn

IV. Tối ưu hóa

1. Move ordering
2. Transposition Table
3. Principal Variation Search
    3.1. Iterative Deepening
    3.2. Null-window search
    3.3. Principal Variation Search

I. Giới thiệu

Báo cáo này trình bày tổng quan về bài tập lớn của nhóm em và những thuật toán chúng em đã áp dụng vào trong bài tập lớn trong việc xây dựng một chương trình để giải quyết trò chơi Connect 4. Chương trình sẽ nhận vào một trạng thái bàn cờ bất kỳ, sau đó có thể tìm kiếm và xác định nước đi tối ưu hoặc nước đi mạnh nhất có thể cho người chơi hiện tại.

Để đối phó với không gian trạng thái lớn và phức tạp của Connect 4, chương trình không dựa vào việc duyệt toàn bộ cây trò chơi bởi như vậy sẽ vô cùng tốn thời gian. Vậy nên ý tưởng cơ bản trong việc xử lý bài toán này là triển khai thuật toán tìm kiếm Minimax, kết hợp với kỹ thuật Alpha-Beta Pruning. Tuy nhiên, hạn chế của cách làm này là độ sâu tìm kiếm càng lớn thì thời gian lại càng lâu. Để đáp ứng với giới hạn một nước chỉ đi trong 10 giây thì độ sâu tìm được chỉ ở mức 5 hoặc 6.

Sau đó, nhóm đã tìm hiểu và chia thành 2 nhóm nhỏ để thực hiện hai hướng khác nhau: một bên thực hiện huấn luyện mô hình học tăng cường (reinforcement learning), một bên thực hiện phát triển theo cách giải solver của Pascal Pons. Tuy nhiên việc huấn luyện mô hình còn xảy ra nhiều bug và việc huấn luyện đòi hỏi tài nguyên và thời gian vô cùng lớn. Điều này cũng xảy ra tương tự đối với bên solver trong việc sinh bảng trạng thái - điểm (table lookup) cho Solver bằng code Python (việc giải theo cách vét cạn và đưa ra kết quả chính xác tuyệt đối có thể lên tới 2,5 phút cho code C++ trong khi chúng ta có khoảng 4.5 nghìn tỷ state). Phần code triển khai các phương án này được nằm trong các branch khác của github (ngoại trừ branch master).

Vậy nên nhóm quyết định sử dụng Negamax (một biến thể của minimax) nhưng nâng cao hiệu suất tính toán, độ sâu tìm kiếm bằng cách sử dụng một loạt các kỹ thuật tối ưu hóa. Các kỹ thuật nổi bật bao gồm: Bitboard, Iterative Deepening, Principal Variation, Transposition Tables và Move Ordering. Trong giới hạn 10 giây, hiện tại chương trình đã có thể tìm kiếm độ sâu từ 9 đến 11,

Ở những phần dưới, chúng ta sẽ đi sâu phân tích từng thành phần, thuật toán và kỹ thuật tối ưu hóa được đề cập ở trên, làm rõ cách chúng hoạt động và đóng góp vào hiệu năng chung của AI Connect 4 này.

Link github

Link video demo

II. Thiết kế chương trình, các thuật toán đã sử dụng

1. Thiết kế tổng quan

Ba thành phần lớp chính cấu thành nên hệ thống:

- **Lớp Board:** Lớp này chịu trách nhiệm lưu trữ vị trí các quân cờ, kiểm tra điều kiện và thực hiện các nước đi, kiểm tra điều kiện thắng, tạo ra một key cho mỗi trạng thái bàn cờ để lưu vào Transposition Table và một số hàm tiện tích khác như tính toán vị trí có thể thắng, thua, ... Điểm đặc biệt quan trọng là việc sử dụng biểu diễn bitboard, một kỹ thuật tối ưu hóa sử dụng các số nguyên lớn và phép toán bit để mô tả trạng thái bàn cờ, cho phép thực hiện các thao tác kiểm tra và cập nhật với tốc độ cao với tài nguyên bộ nhớ nhỏ.
- **Lớp MoveSorter:** Đóng vai trò như một công cụ phụ trợ cho việc sắp xếp nước đi. Nó nhận danh sách các nước đi có thể thực hiện từ một thế cờ cùng với điểm số đánh giá sơ bộ (do class Solver cung cấp) và sắp xếp chúng theo thứ tự ưu tiên giảm dần. Điều này giúp thuật toán tìm kiếm tập trung vào các nước đi triển vọng nhất trước tiên. Lớp này được hoạt động dựa trên tư tưởng priority queue.
- **Lớp Solver:** Là thành phần trung tâm, điều phối toàn bộ quá trình tìm kiếm nước đi. Nó khởi tạo và quản lý các thuật toán tìm kiếm chính, tương tác với lớp Board để mô phỏng các nước đi, sử dụng MoveSorter để tối ưu hóa thứ tự duyệt, quản lý Transposition Tables, áp dụng heuristic, và xử lý các ràng buộc về thời gian hoặc độ sâu tìm kiếm. Cuối cùng, Solver trả về nước đi được đánh giá là tốt nhất.

2. Các thuật toán và kỹ thuật chính

Để giải quyết bài toán tìm kiếm nước đi tối ưu trong Connect 4, lớp Solver tích hợp và phối hợp các thuật toán, kỹ thuật sau:

- **Negamax với Alpha-Beta pruning:** Thuật toán tìm kiếm cốt lõi. Negamax duyệt cây trò chơi theo chiều sâu để xác định giá trị của từng thế cờ từ góc nhìn của người chơi hiện tại. Nó sử dụng cắt tỉa Alpha-Beta là một kỹ thuật tối ưu hóa mạnh mẽ, cho phép bỏ qua việc đánh giá các nhánh của cây tìm kiếm mà chắc chắn không chứa nước đi tốt nhất, giảm đáng kể khối lượng tính toán.
- **Heuristic:** Khi việc tìm kiếm không thể đi đến các trạng thái kết thúc cuối cùng do giới hạn về độ sâu, heuristic được gọi để đưa ra một đánh giá ước lượng về giá trị của một thế cờ, dựa trên các đặc điểm chiến thuật và vị trí quan trọng trên bàn cờ
- **Principal Variation Search (PVS):** Một biến thể tối ưu hóa của Alpha-Beta, giúp giảm thêm số nút cần duyệt so với Alpha-Beta tiêu chuẩn bằng cách search trên một cửa sổ windows có kích thước nhỏ (size = 1) để quyết định xem có search tiếp trên khoảng đó không.

- **Iterative Deepening (ID):** Chương trình không tìm kiếm ngay lập tức đến một độ sâu cố định lớn. Thay vào đó, nó thực hiện tìm kiếm bắt đầu từ độ sâu 1 và tăng dần. Phương pháp này cho phép quản lý thời gian linh hoạt (ngắt khi gặp timeout) và các lần duyệt sau có thể sử dụng thông tin từ các lần tìm kiếm nông hơn để cải thiện hiệu quả của các lần tìm kiếm sâu hơn. Bên cạnh đó, ID có thể tìm kiếm được các node lá sớm, giúp dễ dàng đưa ra quyết định.
- **Transposition Table (TT):** Sử dụng Hash Table để lưu trữ thông tin về các thế cờ đã được đánh giá trước đó (bao gồm điểm số, độ sâu phân tích). Khi gặp lại một thế cờ đã có trong TT, chương trình có thể tái sử dụng thông tin này thay vì tính toán lại, giúp tăng tốc độ tìm kiếm một cách đáng kể, đặc biệt là ở các thế cờ lặp lại.
- **Move Ordering:** Hiệu quả của cắt tỉa Alpha-Beta phụ thuộc rất lớn vào thứ tự các nước đi được xem xét tại mỗi nút. Chương trình áp dụng các heuristic để ưu tiên các nước đi có khả năng, giúp tối đa hóa khả năng cắt tỉa sớm.

III. Thuật toán sử dụng

1. Negamax và Alpha-Beta pruning

1.1. Negamax

Thuật toán tìm kiếm được triển khai trong hàm `_negamax`. Ưu điểm chính của Negamax so với Minimax là nó đơn giản hóa việc triển khai mã nguồn. Thay vì sử dụng sử dụng:

```
if maximizingPlayer then
	value := −∞
	for each child of node do
		value := max(value, minimax(child, depth − 1, FALSE))
	return value
else (* minimizing player *)
	value := +∞
	for each child of node do
		value := min(value, minimax(child, depth − 1, TRUE))
```

Thì ta có thể rút ngắn code chỉ còn 1 dòng như sau:

```
value := max(value, −negamax(child, depth − 1, −color))
```

bằng cách thay thế boolean thành color với giá trị có thể là 1 hoặc -1 để đổi giữa 2 người chơi mỗi lần minimize hay maximize.

1.2. Alpha-Beta pruning

Để làm cho quá trình tìm kiếm trở nên khả thi bằng cách cách giảm đáng kể số lượng nút cần duyệt trên cây trò chơi, Negamax được tăng cường bằng cách kết hợp với Alpha-Beta pruning. Trong quá trình duyệt cây, 2 giá trị biên được duy trì:

- Alpha: đại diện cho giá trị tốt nhất, tức cao nhất cho người chơi hiện tại (người cố gắng tối đa hóa số điểm tại nút đang xét), dựa trên các nhánh đã được khám phá dọc theo đường đi từ nút gốc tới nút hiện tại.
- Beta: Đại diện cho giá trị tốt nhất, tức thấp nhất mà đối thủ (người cố gắng tối thiểu hóa điểm số người chơi hiện tại) có thể đạt được, dựa trên các nhánh đã được khám phá.

Điều kiện cắt tỉa xảy ra khi Alpha >= Beta. Ý nghĩa của điều kiện này là:

Tại một nút đang xét, người chơi hiện tại đã tìm thấy một nhánh mang lại kết quả (Alpha) là tốt hơn hoặc bằng với kết quả (Beta) mà đối thủ đã nhận được tại nhánh khác. Do đó đối thủ sẽ không bao giờ chọn đi vào nhánh này. Vì vậy việc tiếp tục khám phá các nước còn lại tại nút này và các nhánh con của nó là không cần thiết và có thể cắt bỏ.

1.3. Triển khai thuật toán trong mã nguồn

- Hàm nhận trạng thái của bàn cờ `board`, các biên `alpha`, `beta`, và độ sâu tìm kiếm còn lại `depth`
- Base cases:
  - Đối thủ đã thắng ở nước trước đó `board.has_won(...)`. Trả về điểm thua `loss_score.`
  - Bàn cờ đầy (hòa) `board.nb_moves() >= self.W * self.H`. Trả về 0.
  - Đạt giới hạn độ sâu `depth <= 0`. Trả về giá trị ước lượng từ hàm `heuristic` hoặc điểm thắng tuyệt đối nếu có nước thắng ngay.
- Kiểm tra Transposition Table:

  Trước khi tìm kiếm sâu hơn, hàm kiểm tra trong `trans_table` bằng hàm `_handle_tt_lookup`. Nếu nước đi hợp lệ sẽ trả về kết quả ngay lập tức, tránh việc tính toán lại.
- Sinh và sắp xếp nước đi:

  Các nước đi hợp lệ được tạo ra bằng `board.possible()` và sau đó được sắp xếp với `_generate_and_sort_moves` dựa trên các tiêu chí khác nhau. Việc sắp xếp vô cùng quan trọng vì nó tăng khả năng cắt tỉa Alpha-Beta sớm, giúp giảm số nút cần duyệt.
- Vòng lặp duyệt nước đi: Hàm sẽ duyệt từng nước đi đã được sắp xếp
  - Thực hiện nước đi tiếp theo trên bản sao bàn cờ `board_copy.play(next_move)`
  - Gọi đệ quy Negamax `score = -self._negamax(board_copy, -beta, -alpha, depth - 1)`. Đoạn code này còn tích hợp thêm logic của PVS để tối ưu hóa thêm, sẽ được đề cập cụ thể sau.
  - So sánh `score` vừa nhận được với `best_score` (điểm tốt nhất được tìm tới nay trong số các nước đi của nút hiện tại). Nếu `score` tốt hơn, cập nhật `best_score`. Sau đó nếu `best_score` tốt hơn hơn `alpha`, ta sẽ cập nhật `alpha`
  - Sau khi cập nhật `alpha`, kiểm tra điều kiện cắt tỉa.
- Lưu `best_score` vào TT phục vụ những lần tìm kiếm sau, đồng thời trả về `best_score` cho hàm `_negamax`. (Trong mã nguồn có đề cập tới các cờ `TT_EXACT, TT_UPPERBOUND,TT_LOWERBOUND` sẽ được giải thích sau).

2. Heuristic

2.1. Mục đích và vai trò

Khi thuật toán tìm kiếm Negamax duyệt cây trò chơi, nó không phải lúc nào cũng có thể đi đến các terminal nodes (tức là các trạng thái thắng, thua, hoặc hòa) do giới hạn về độ sâu tìm kiếm (để kiểm soát thời gian tính toán). Trong những trường hợp này, thuật toán cần một cách

để "đoán" xem một thế cờ không phải là kết thúc cuối cùng thì tốt hay xấu như thế nào. Đây chính là nhiệm vụ của hàm heuristic.
Hàm heuristic được gọi (`depth <= 0`) để cung cấp một điểm số ước lượng cho thế cờ hiện tại. Điểm số này không đảm bảo là giá trị thực của thế cờ (vì chưa nhìn thấy kết quả cuối cùng), nhưng nó phản ánh một đánh giá dựa trên các đặc điểm chiến thuật và vị trí quan trọng trên bàn cờ. Thuật toán tìm kiếm sẽ sử dụng điểm số heuristic này để so sánh các nhánh khác nhau và hướng sự tìm kiếm về phía những thế cờ có vẻ có lợi hơn

2.2. Phân tích mã nguồn hàm `heuristic`
(Phần phân tích này sẽ tập trung chú trọng vào cách thức để đưa ra hàm heuristic này, những đoạn code liên quan đến bitboard sẽ không được đề cập tới)

- Đầu vào là trạng thái của bàn cờ hiện tại. Ta sẽ khởi tạo điểm số `score = 0`
- Kiểu soát cột giữa: Với bàn cờ kích thước 6 x 7, mọi nước đi chiến thắng bằng cách có 4 quân cờ chéo nhau hay nằm ngang hàng nhau đều có sự xuất hiện của một quân cờ ở cột giữa. Bởi vậy ta sẽ cộng 3 điểm cho mỗi quân ở cột giữa, tăng lợi thế giành chiến thắng (`score += center_count * 3`)
- Những mối đe dọa thắng ngay (3 quân cờ + 1 ô trống có thể đi): Đây là yếu tố không thể thiếu trong hàm heuristic vì nó là những thế cờ có thể dẫn tới nước chiến thắng. Ta sẽ thưởng điểm cao cho những nước đi tạo ra chúng, nhưng đồng thời sẽ trừ điểm nặng hơn nếu đối thủ có thể thế cờ này. Mục đích là để tránh những nước đi tạo thế cờ đẹp cho đối thủ. Cụ thể ta sẽ cộng 5 điểm cho mỗi nơi có 3 quân và 1 ô trống (`score += player_threats_3p1e * 5`) và trừ 6 điểm cho mỗi nơi mà đối thủ có (`score -= opponent_threats_3o1e * 6`)
- Những nước đi tiềm năng (2 quân cờ + 2 ô trống): Tuy không phải mối đe dọa trực tiếp nhưng nó là nền tảng cho những nước tiếp theo. Đương nhiên những nước này có giá trị chiến lược thấp hơn với những nước thắng ngay. Bởi vậy ta sẽ cộng 2 điểm cho mỗi nước đi tiềm năng (`score += score_22 * 2`) và trừ 3 điểm với mỗi nới đối thủ có nó (`score -= score_22_opponent * 3`)

Những trọng số được lựa chọn trọng việc thưởng/phạt như trên đều dựa trên kinh nghiệm cá nhân sau khi chơi khoảng 100 ván game, vậy nên có thể vẫn còn sai sót và có thể tinh chỉnh để trở nên tốt hơn.

IV. Tối ưu hoá

1. Move Ordering

Trong thuật toán Negamax kết hợp với cắt tỉa alpha-beta, thứ tự duyệt các nước đi (action) ảnh hưởng rất lớn đến hiệu quả của việc cắt tỉa. Để tối ưu hóa, ta thường sắp xếp các nước đi sao cho những nước có khả năng tốt nhất được khám phá trước. Khi đó, khoảng [alpha, beta] sẽ co lại nhanh hơn. Việc này giúp tăng xác suất cắt tỉa (pruning), từ đó giảm đáng kể số lượng trạng thái (state) cần duyệt.
Chiến lược sắp xếp thứ tự các nước đi mặc định là từ ở giữa, ra 2 bên.
Từ transposition table và từ Iterative deepening, ta có thể sắp xếp các nước đi theo thứ tự ưu tiên như sau từ cao xuống thấp:

- Nước đi thắng trực tiếp

- Nước đi chặn đối phương thắng trực tiếp
- Nước đi từ Transposition Table
- Nước đi từ Iterative deepening
- Còn lại theo thứ tự mặc định

2. Transposition Table

Trong quá trình duyệt cây tìm kiếm, một trạng thái (state) có thể được truy cập từ nhiều đường đi khác nhau, thông qua các chuỗi hành động (action) khác nhau. Nếu không kiểm soát, việc tính toán lại cùng một trạng thái nhiều lần sẽ gây lãng phí tài nguyên tính toán.

Để khắc phục điều này, ta sử dụng một **transposition table** (hash table) lưu trữ kết quả của các trạng thái đã được đánh giá trước đó. Khi gặp lại một trạng thái, thuật toán có thể tra cứu bảng này để tái sử dụng kết quả đã tính, thay vì phải duyệt lại toàn bộ cây con từ trạng thái đó.

Giá trị lưu trữ bao gồm: Depth, score, flag
Depth: lưu trữ thông tin về độ sâu mà state đó đã tính đến. Nếu depth lưu trong transposition table lớn hơn hoặc bằng depth hiện tại, ta có thể sử dụng kết quả trong transposition table để trả về kết quả ngay lập tức, gây cắt tỉa, hoặc cập nhật cận alpha/beta
Flag:

+ EXACT: xác định được chính xác giá trị của state đó được tính toán trong khoảng [alpha, beta] của lần tìm kiếm trước đó. Xảy ra khi trong lần tìm kiếm trước đó không bị cắt tia bởi alpha hay beta, tức (alpha < score < beta). Khi đó, nếu gặp lại state này, độ sâu lưu trữ (depth) đủ lớn so với độ sâu yêu cầu hiện tại, ta có thể sử dụng trực tiếp score để làm kết quả mà không cần tìm kiếm thêm.
+ LOWERBOUND (Fail-high): Giá trị score được lưu trữ là một cận dưới cho khoảng [alpha,beta], tức từ lần tìm kiếm trước đó, đã tìm được số điểm tối thiểu của state này. Khi gặp lại state này, ta có thể sử dụng nó để nâng giá trị alpha của lần tìm kiếm hiện tại, tăng xác suất cắt tỉa
+ UPPERBOUND (Fail-low): Giá trị này cho biết giá trị chính xác của state này không vượt quá giá trị đã lưu. Nên khi gặp lại state này, nếu độ sâu > độ sâu hiện tại và giá trị đã lưu < alpha hiện tại, ta có thể nói đánh giá state này không cải thiện được giá trị alpha. Nếu không cắt tỉa alpha, ta có thể sử dụng score này để hạ thấp beta, thu hẹp cửa sổ tìm kiếm và tăng hiệu quả cắt tỉa

3. Principal Variation Search

3.1. Iterative Deepening

Iterative Deepening là việc duyệt các trạng thái theo depth từ thấp đến cao. Nó có khả năng tìm được những winning state ở các depth sớm. Nếu ta tìm ở depth quá cao ngay từ đầu, ta có thể mất rất nhiều thời gian để có thể tìm ra winning state ở depth nông hơn ở các nước đi sau đó.

Hơn nữa, khi duyệt ở depth d, việc sử dụng nước đi tối ưu cho depth d để ưu tiên duyệt đầu tiên cho depth d + 1 sẽ tăng cao xác suất cắt tỉa, từ đó tối ưu được thuật toán hơn nữa.

3.2. Null-window search

Kỹ thuật này sử dụng cửa sổ [alpha; alpha +1] để kiểm tra xem điểm của state tiếp theo có chắc chắn không tốt hơn alpha hay không. Nếu NWS trả về giá trị v <= alpha, khi đó, ta có thể bỏ qua state này. Nếu NWS trả về giá trị v >= alpha + 1, điều này cho thấy state này có khả năng tốt hơn nước đi tốt nhất hiện tại.

3.3. Principal Variation Search (PVS)

Khi ta có thứ tự sắp xếp các nước đi hiệu quả, sử dụng thuật toán PVS thay cho negamax. PVS hoạt động dựa trên giả thuyết: nếu việc sắp xếp thứ tự các nước đi đủ tốt, thì nước đi đầu tiên được xét tại mỗi nút có khả năng cao là nước đi tối ưu (thuộc về Principal Variation). Principal Variation là chuỗi các nước đi tốt nhất bắt đầu từ vị trí hiện tại.
Do đó, thuật toán PVS xử lý nước đi đầu tiên khác so với các nước đi còn lại

- Khi duyệt nước đi đầu tiên, ta duyệt với cửa sổ đầy đủ [alpha, beta] vì đây là nước đi có khả năng là nước đi tốt nhất.
- Tiếp theo, ta duyệt các nước đi còn lại với null-window ([alpha, alpha+1]) để loại bỏ tất cả các nước không tốt hơn nước hiện tại. Nếu tìm được kết quả trả về lớn hơn alpha (tức là nước đi tốt hơn nước đi hiện tại), ta sẽ duyệt lại một lần nữa với cửa sổ đầy đủ, để tìm được kết quả chính xác