ĐẠI HỌC BÁCH KHOA HÀ NỘI

TRƯỜNG CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

BÁO CÁO NHẬP MÔN TRÍ TUỆ NHÂN TẠO

Đề tài: Bot chơi cờ tướng

Sinh viên thực hiện: 20215281 – Nguyễn Xuân Mạnh
20210096 – Nguyễn Thanh Nhật Bảo
20215278 – Đặng Trần Nam Khánh
20215279 – Lê Hoàng Long
20215284 – Trần Nhật Minh

GVHD: TS. Trần Thế Hùng

Học kỳ: 20232

Hà Nội, tháng 06/2024

MỤC LỤC

PHÂN CÔNG CÔNG VIỆC 6
Chương 1. PHÂN TÍCH ĐỀ TÀI 7
1. Giới thiệu bài toán 7
2. Mục tiêu đề tài 7
3. Ý nghĩa của đề tài 7
Chương 2. CƠ SỞ LÝ THUYẾT 8
1. Thuật toán Tham lam (Greedy Algorithm) 8
2. Thuật toán Minimax 8
3. Thuật toán Alpha-Beta Search 12
4. Thuật toán Monte Carlo 18
5. Chiến thuật [1] 21
Chương 3. THIẾT KẾ GIẢI PHÁP 24
1. Thuật toán Greedy 24
2. Thuật toán Minimax 25
3. Triển khai thuật toán Alpha-Beta Search 27
4. Triển khai thuật toán Monte Carlo Tree Search 29
Chương 4. TRIỂN KHAI THỬ NGHIỆM VÀ ĐÁNH GIÁ 32
1. Thử nghiệm về chất lượng nước đi 32
2. Thử nghiệm về thời gian cần thiết để sinh ra nước đi 32
KẾT LUẬN 34
TÀI LIỆU THAM KHẢO 35

DANH MỤC HÌNH VẼ

Hình 1 Mô phỏng thuật toán Minimax 9
Hình 2 Mô phỏng thuật toán Minimax 10
Hình 3 Mô phỏng thuật toán Minimax 11
Hình 4 Mô phỏng thuật toán Minimax 12
Hình 5 Mô phỏng thuật toán Alpha-Beta Search 13
Hình 6 Mô phỏng thuật toán Alpha-Beta Search 14
Hình 7 Mô phỏng thuật toán Alpha-Beta Search 15
Hình 8 Mô phỏng thuật toán Alpha-Beta Search 16
Hình 9 Mô phỏng thuật toán Alpha-Beta Search 17
Hình 10 Mô phỏng thuật toán Alpha-Beta Search 18
Hình 11 Mô phỏng thuật toán MCTS 19
Hình 12 Bảng trọng số quân tốt đen 21
Hình 13 Bảng trọng số quân mã đen 21
Hình 14 Bảng trọng số quân xe đen 22
Hình 15 Bảng trọng số quân pháo đen 22
Hình 16 Hàm tính giá trị của 1 quân cờ tại 1 vị trí trên bàn cờ 23
Hình 17 Hàm heuristic tính tổng trọng số tất cả quân trên bàn cờ 23
Hình 18 Thuật toán Greedy 24
Hình 19 Hàm Minimax 25
Hình 20 Hàm find_best_move_minimax 26
Hình 21 Hàm find_best_move 27
Hình 22 Hàm Alpha-Beta Search 28
Hình 23 Hàm định nghĩa Node trong MCTS 29
Hình 24 Hàm tạo các nước đi mới vào Node và mô phỏng 30
Hình 25 Hàm định nghĩa các trạng thái bàn cờ 31
Hình 26 Hàm lan truyền ngược và thuật toán MCTS 31

DANH MỤC BẢNG BIỂU

Bảng 1 So sánh chất lượng nước đi của các thuật toán 32
Bảng 2 So sánh thời gian sinh ra nước đi của các thuật toán.....32

DANH MỤC TỪ VIẾT TẮT

Từ viết tắt	Giải nghĩa
MCTS	Thuật toán Monte Carlo Tree Search

PHÂN CÔNG CÔNG VIỆC

Thành viên	**Công việc**
Nguyễn Xuân Mạnh	Nghiên cứu bài toán, lý thuyết. Quản lý và lên kết hoạch. Kết nối các module, kiểm thử và thống kê các số liệu.
Nguyễn Thanh Nhật Bảo	Nghiên cứu bài toán, lý thuyết. Cài đặt sinh các nước đi cho bot và thuật toán tham lam
Đặng Trần Nam Khánh	Nghiên cứu bài toán, lý thuyết. Cài đặt thuật toán Alpha-Beta Search
Lê Hoàng Long	Nghiên cứu bài toán, lý thuyết. Cài đặt giao diện và thuật toán Minimax
Trần Nhật Minh	Nghiên cứu bài toán, lý thuyết. Cài đặt thuật toán MCTS và nghiên cứu thêm về học tăng cường.

Chương 1. PHÂN TÍCH ĐỀ TÀI

1. Giới thiệu bài toán

Cờ tướng là một trò chơi chiến lược cổ điển được yêu thích rộng rãi ở nhiều quốc gia, đặc biệt là Trung Quốc và Việt Nam. Với lịch sử lâu đời và quy tắc phức tạp, cờ tướng không chỉ đòi hỏi người chơi có kỹ năng tư duy chiến lược mà còn cần sự kiên nhẫn và khả năng dự đoán nước đi của đối thủ. Trong bối cảnh công nghệ phát triển không ngừng, việc áp dụng trí tuệ nhân tạo (AI) vào cờ tướng đã trở thành một lĩnh vực nghiên cứu thú vị và đầy thử thách.

2. Mục tiêu đề tài

Đề tài của nhóm nhằm phát triển một hệ thống AI có khả năng chơi cờ tướng, với mục tiêu tạo ra một đối thủ chơi cờ tướng thông minh. Hệ thống này sẽ có khả năng thi đấu với người chơi và đưa ra những nước đi hợp lý dựa trên các chiến lược đã học. Đồng thời, đề tài cũng nghiên cứu và ứng dụng các thuật toán AI tiên tiến như tìm kiếm cây để cải thiện hiệu suất của bot.

3. Ý nghĩa của đề tài

Việc nghiên cứu và phát triển bot chơi cờ tướng không chỉ có ý nghĩa trong việc nâng cao khả năng giải quyết các vấn đề phức tạp của AI mà còn mang lại nhiều ứng dụng thực tiễn, như giải trí, giáo dục, và hỗ trợ cộng đồng yêu cờ tướng. Bot có thể giúp người chơi nâng cao kỹ năng, học hỏi chiến lược mới, và thách thức bản thân qua các ván cờ. Điều này không chỉ thúc đẩy tiến bộ công nghệ mà còn đóng góp vào sự phát triển của xã hội hiện đại.

Chương 2. CƠ SỞ LÝ THUYẾT

Việc xây dựng một bot chơi cờ tướng đòi hỏi sự kết hợp của nhiều thuật toán và công nghệ trí tuệ nhân tạo để đạt được hiệu suất tối ưu. Trong đề tài này, nhóm tập trung vào việc ứng dụng bốn thuật toán chính: thuật toán tham lam, thuật toán Minimax, thuật toán Alpha-Beta Search, và thuật toán Monte Carlo. Sau đó nhóm sẽ so sánh hiệu suất của các thuật toán, từ đó rút ra được các ưu, nhược điểm của 4 thuật toán này.

1. Thuật toán Tham lam (Greedy Algorithm)

Thuật toán tham lam là một phương pháp đơn giản và hiệu quả để giải quyết nhiều bài toán tối ưu. Trong cờ tướng, thuật toán tham lam có thể được sử dụng để chọn nước đi tốt nhất ở mỗi bước dựa trên một số tiêu chí đánh giá nhất định, chẳng hạn như giá trị của quân cờ ăn được hoặc vị trí chiến lược của quân cờ. Mặc dù không đảm bảo sẽ đưa ra giải pháp tối ưu cho toàn bộ ván cờ, nhưng thuật toán tham lam có thể cung cấp các nước đi hợp lý trong thời gian ngắn, giúp giảm tải tính toán cho các thuật toán phức tạp hơn.

Dựa vào trạng thái hiện tại của bàn cờ, ta sẽ lấy tất cả các nước đi hợp lệ, sau đó tính giá trị trạng thái bàn cờ cho từng nước đi. Thuật toán sẽ lựa chọn nước đi có tổng trọng số cao nhất (nếu người chơi là Max) và ngược lại, chọn nước đi có trọng số âm nhất (nếu người chơi là Min).

Ưu điểm: Nhanh, không tốn nhiều thời gian

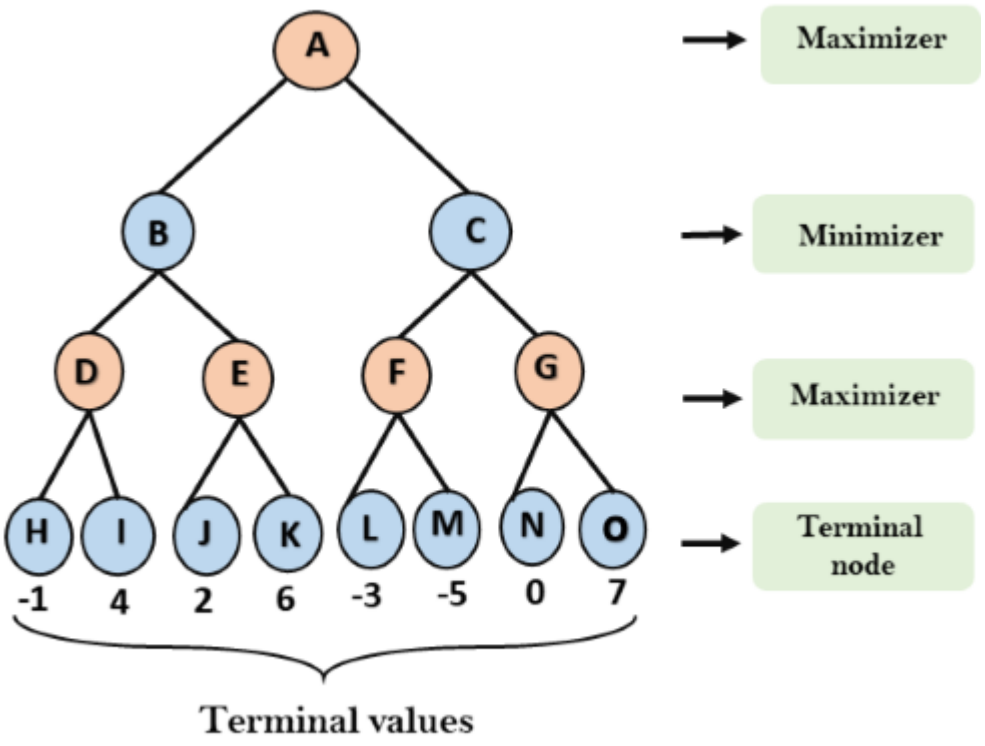
Nhược điểm: Thuật toán chỉ quan tâm đến 1 nước đi tiếp theo và tối ưu hóa trọng số nên sẽ không tối ưu trong nhiều trường hợp. Do cờ tướng còn là sự kết hợp của nhiều nước đi với nhau để tạo nên một cấu trúc chiến thuật

2. Thuật toán Minimax

Giải thuật Minimax là một thuật toán đệ quy lựa chọn bước đi kế tiếp trong một trò chơi có hai người. Xét một trò chơi đối kháng trong đó hai người thay phiên đi nước đi của mình như cờ tướng. Khi chơi bạn có thể khai triển hết không gian trạng thái nhưng khó khăn chủ yếu là bạn phải tính toán được phản ứng và nước đi của đối thủ mình như thế nào? Cách xử lý đơn giản là bạn giả sử đối thủ của bạn cũng sử dụng kiến thức về không gian trạng thái giống bạn. Giải thuật Minimax áp dụng giả thuyết này để tìm kiếm không gian trạng thái của trò chơi.

Bước 1: Trong bước đầu tiên, thuật toán tạo ra Game Tree và áp dụng hàm Minimax để nhận các giá trị và các trạng thái kết thúc.

Trong sơ đồ cây dưới đây, hãy lấy A là trạng thái bắt đầu của Tree. Giả sử maximizer thực hiện lượt đi đầu tiên có giá trị ban đầu trong trường hợp xấu nhất = – infinite và minimizer sẽ thực hiện lượt tiếp theo có giá trị ban đầu trong trường hợp xấu nhất = + infinity.



Hình 1 Mô phỏng thuật toán Minimax

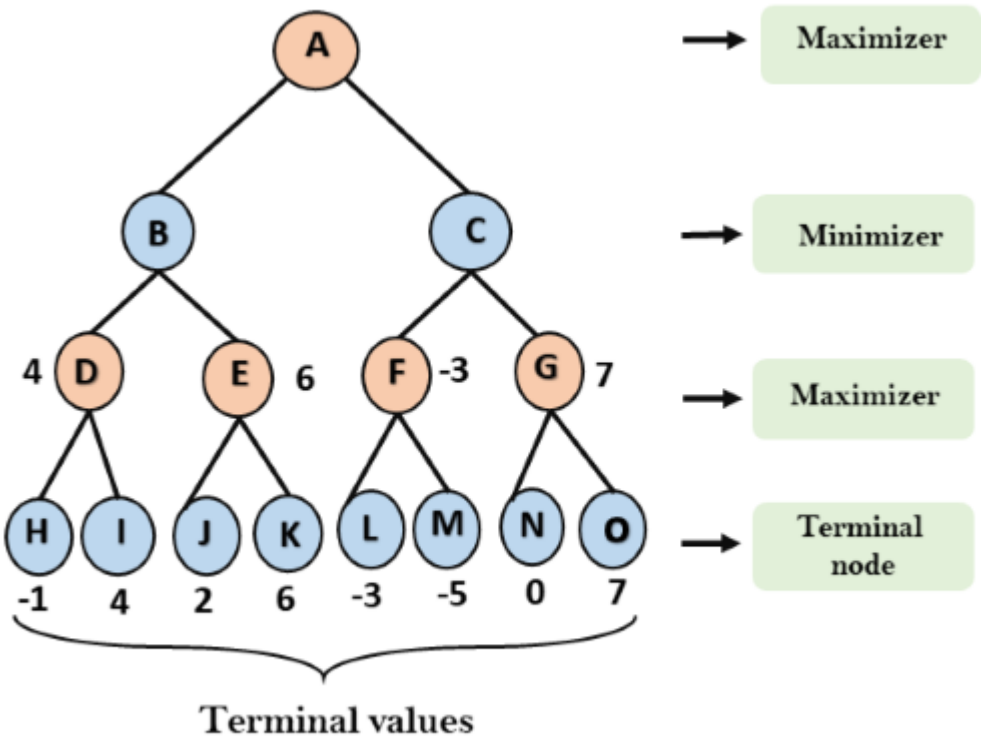
Bước 2: Bây giờ, đầu tiên chúng ta tìm giá trị cho Maximizer, giá trị ban đầu của nó là -∞, vì vậy chúng ta sẽ so sánh từng giá trị ở trạng thái đầu cuối với giá trị ban đầu của Maximizer và xác định các giá trị nút cao hơn. Nó sẽ tìm thấy mức tối đa trong số tất cả.

Đối với nút D max (-1, – -∞) => max (-1,4) = 4

Đối với nút E max (2, -∞) => max (2, 6) = 6

Đối với nút F max (-3, -∞) => max (-3, -5) = -3

Đối với nút G max (0, -∞) = max (0, 7) = 7

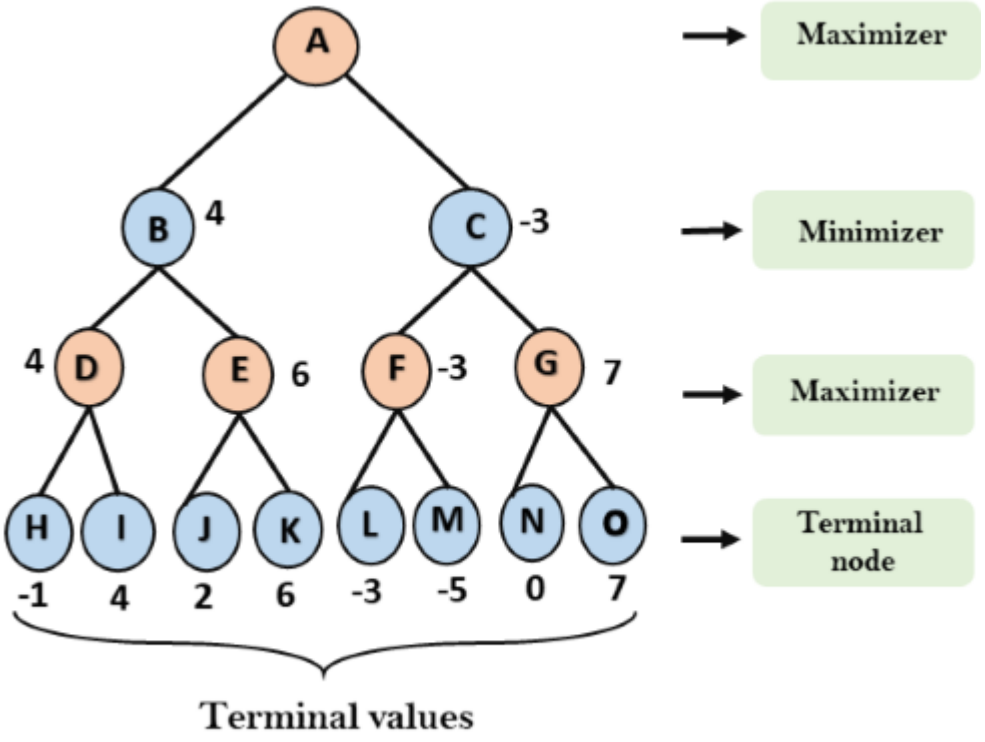


Hình 2 Mô phỏng thuật toán Minimax

Bước 3: Trong bước tiếp theo, đến lượt trình Minimizer, vì vậy nó sẽ so sánh giá trị tất cả các nút với + ∞ và sẽ tìm giá trị nút lớp thứ 3.

Đối với nút B = min (4,6) = 4

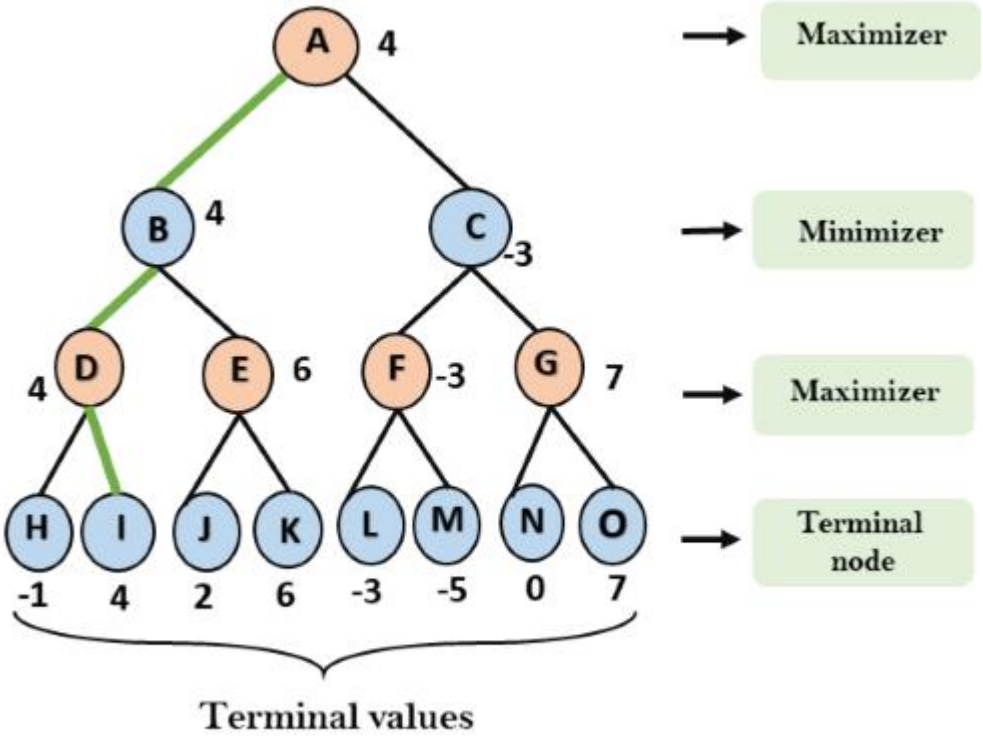
Đối với nút C = min (-3, 7) = -3



Hình 3 Mô phỏng thuật toán Minimax

Bước 4: Bây giờ đến lượt Maximizer, nó sẽ lại chọn giá trị lớn nhất của tất cả các nút và tìm giá trị lớn nhất cho nút gốc. Trong Game Tree này, chỉ có 4 lớp, do đó chúng truy cập ngay đến nút gốc, nhưng trong trò chơi thực, sẽ có nhiều hơn 4 lớp.

Đối với nút A max (4, -3) = 4



Hình 4 Mô phỏng thuật toán Minimax

Ưu điểm:

- Tìm kiếm được mọi nước đi tiếp theo sau đó lựa chọn nước đi tốt nhất, vì giải thuật có tính chất vét cạn nên không bỏ sót trạng thái

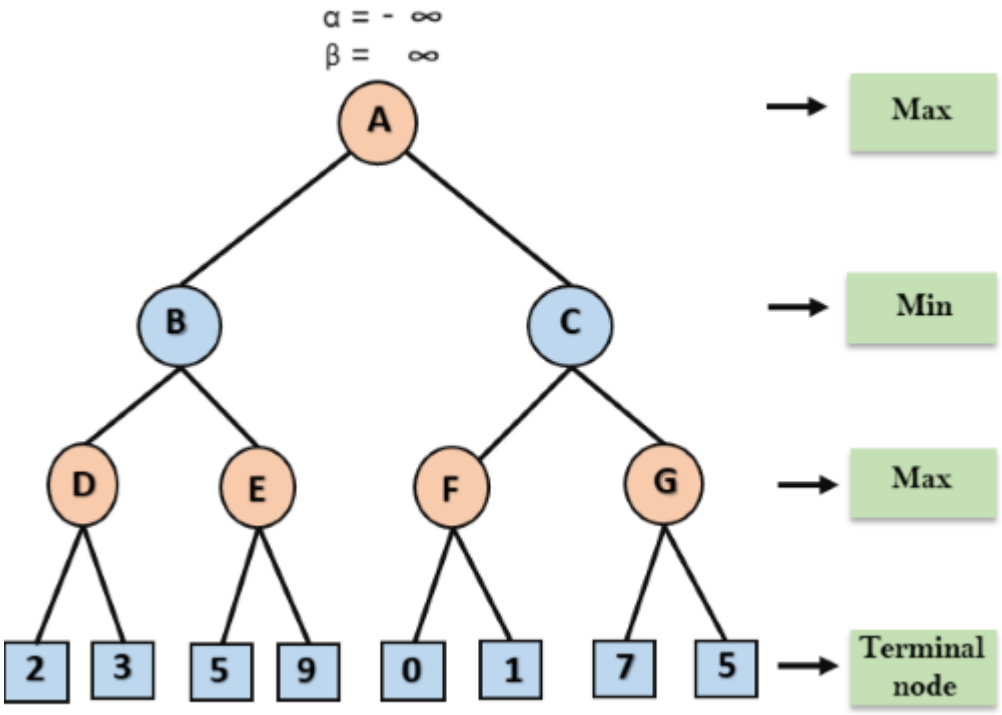
Nhược điểm:

- Đối với các trò chơi có không gian trạng thái lớn như caro, cờ tướng… việc chỉ áp dụng giải thuật Minimax có lẽ không còn hiệu quả nữa do sự bùng nổ tổ hợp quá lớn.
- Giải thuật áp dụng nguyên lý vét cạn không tận dụng được thông tin của trạng thái hiện tại để lựa chọn nước đi, vì duyệt hết các trạng thái nên tốn thời gian.

3. Thuật toán Alpha-Beta Search

Alpha-beta search là một cải tiến của thuật toán minimax, được sử dụng rộng rãi trong các trò chơi đối kháng như cờ vua và cờ tướng. Thuật toán này giúp cắt tỉa các nhánh không cần thiết trong cây tìm kiếm, qua đó giảm thiểu số lượng nước đi cần phải đánh giá. Alpha-beta search hoạt động bằng cách duy trì hai giá trị alpha và beta để theo dõi giới hạn trên và giới hạn dưới của giá trị mong đợi, từ đó loại bỏ các nước đi không khả thi. Điều này giúp tăng tốc độ tìm kiếm và cải thiện hiệu suất của bot trong việc đưa ra quyết định.

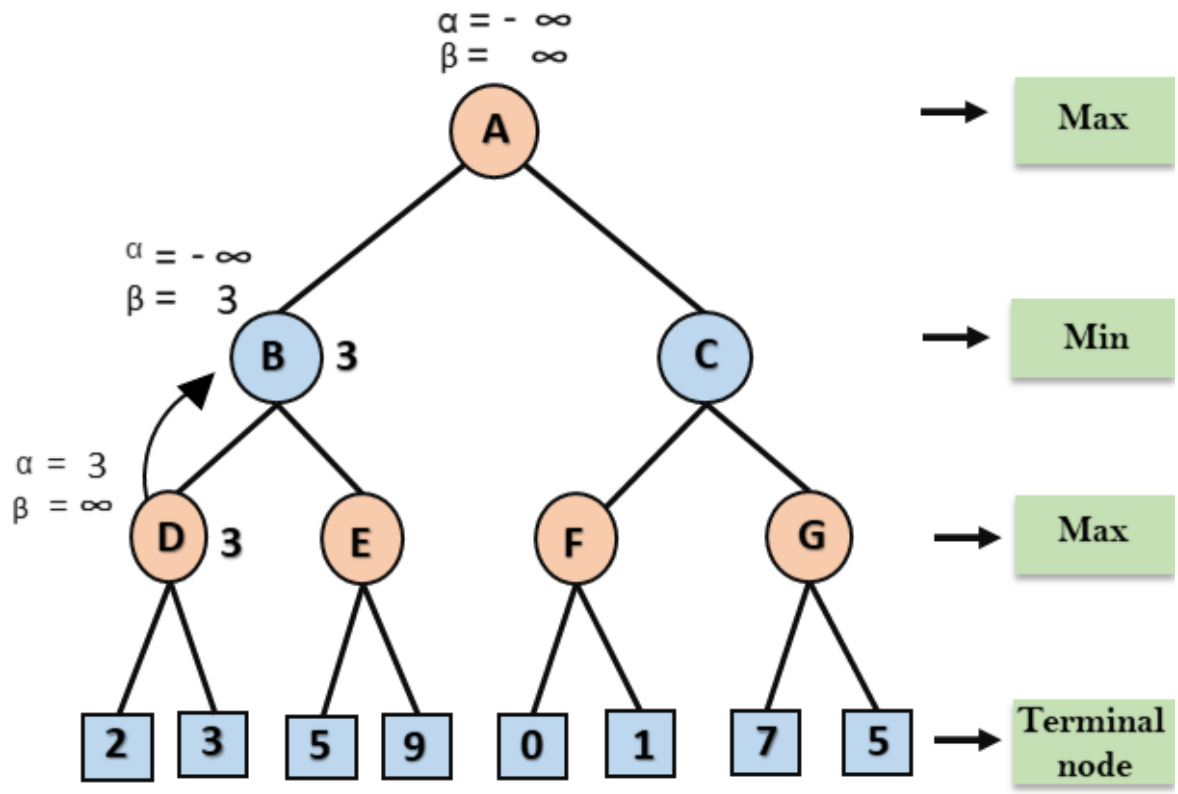
Bước 1: Ở bước đầu tiên, người chơi Max sẽ bắt đầu di chuyển đầu tiên từ nút A nơi $\alpha = -\infty$ và $\beta = +\infty$, những giá trị alpha và beta này được truyền lại cho nút B nơi lại $\alpha = -\infty$ và $\beta = +\infty$ và Nút B chuyển cùng một giá trị cho nút con D của nó.



Hình 5 Mô phỏng thuật toán Alpha-Beta Search

Bước 2: Tại nút D, giá trị của α sẽ được tính theo lượt của nó cho Max. Giá trị của α được so sánh với đầu tiên là 2 và sau đó là 3, và max (2, 3) = 3 sẽ là giá trị của α tại nút D và giá trị của nút cũng sẽ là 3.

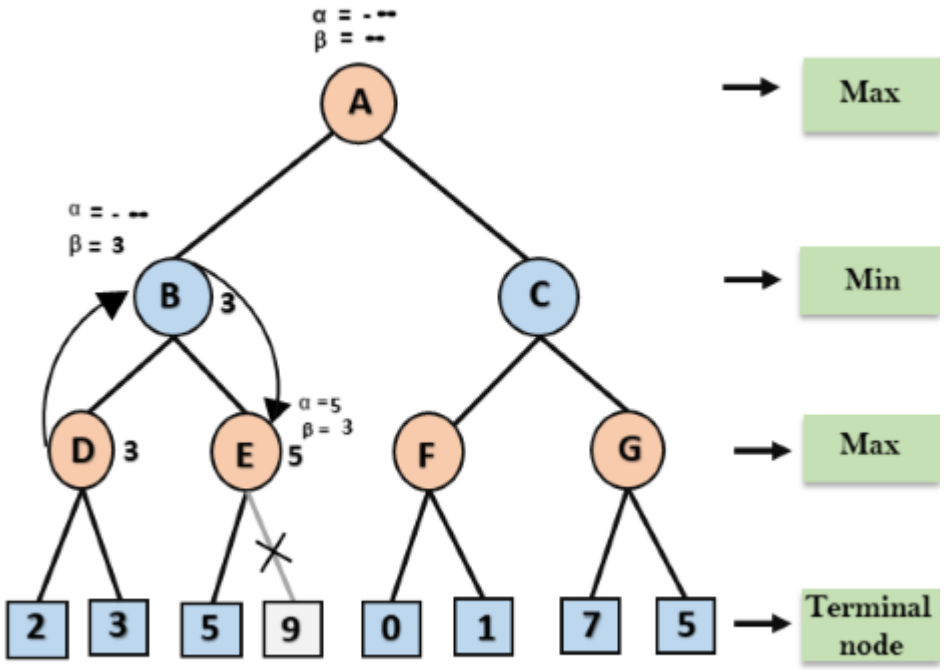
Bước 3: Bây giờ thuật toán quay ngược lại nút B, trong đó giá trị của β sẽ thay đổi vì đây là lượt của Min, Bây giờ β = + ∞, sẽ so sánh với giá trị của các nút tiếp theo có sẵn, tức là min (∞, 3) = 3, do đó tại nút B bây giờ α = -∞ và β = 3.



Hình 6 Mô phỏng thuật toán Alpha-Beta Search

Trong bước tiếp theo, thuật toán duyệt qua nút kế tiếp tiếp theo của nút B là nút E, và các giá trị của α = -∞ và β = 3 cũng sẽ được chuyển.

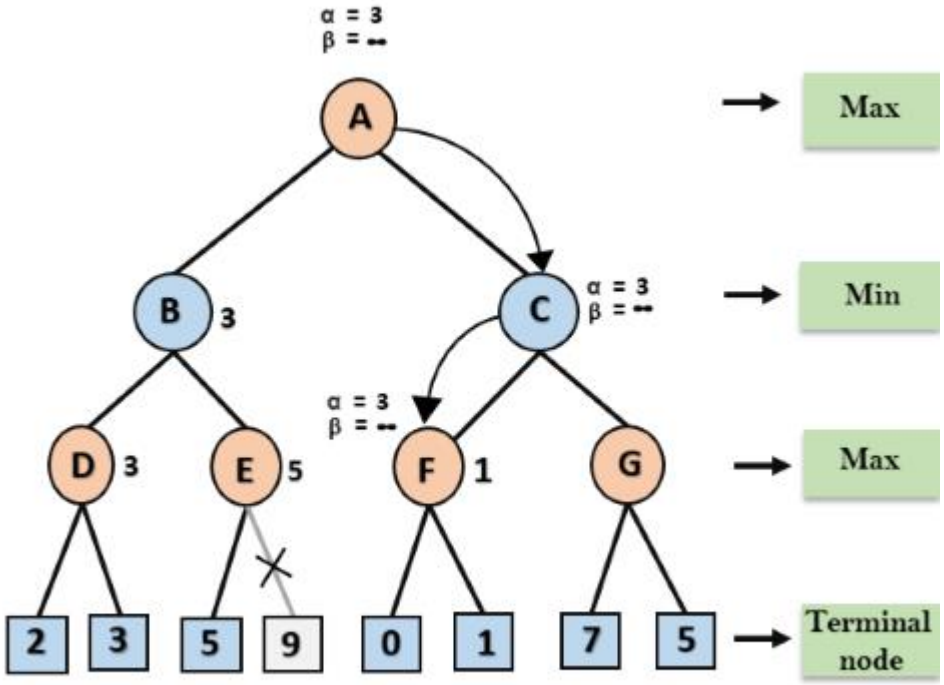
Bước 4: Tại nút E, Max sẽ đến lượt, và giá trị của alpha sẽ thay đổi. Giá trị hiện tại của alpha sẽ được so sánh với 5, vì vậy max (-∞, 5) = 5, do đó tại nút E α = 5 và β = 3, trong đó α>= β, vì vậy nhánh bên phải của E sẽ bị lược bỏ, và thuật toán sẽ không đi qua nó, và giá trị tại nút E sẽ là 5.



Hình 7 Mô phỏng thuật toán Alpha-Beta Search

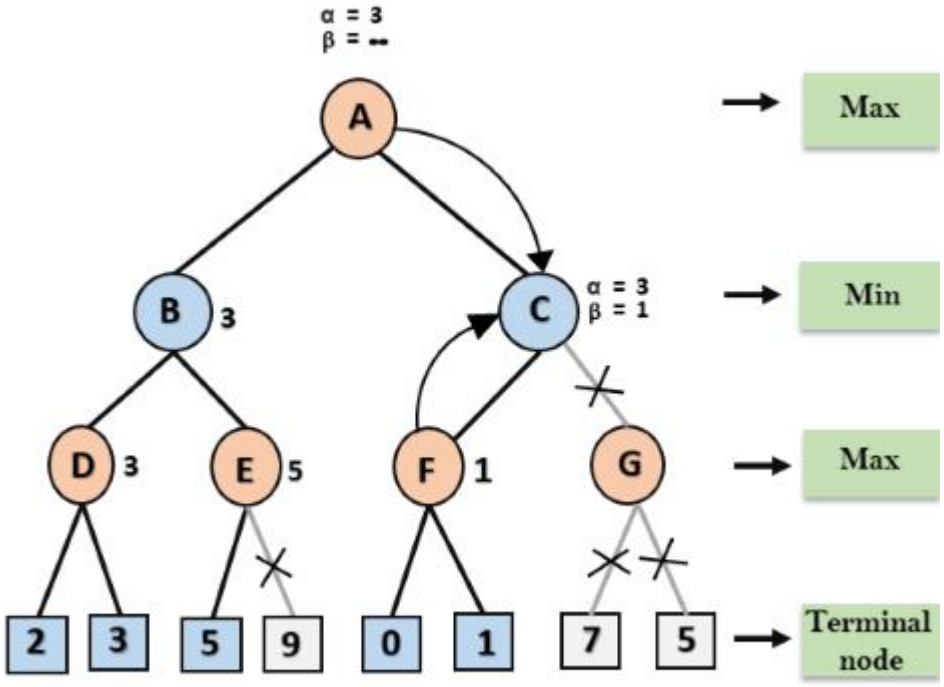
Bước 5: Ở bước tiếp theo, thuật toán lại chiếu ngược cây, từ nút B đến nút A. Tại nút A, giá trị của alpha sẽ được thay đổi giá trị lớn nhất có sẵn là 3 như max (-∞, 3) = 3 và β = + ∞, hai giá trị này bây giờ được chuyển đến người kế nhiệm bên phải của A là Nút C. Tại nút C, α = 3 và β = + ∞, và các giá trị tương tự sẽ được chuyển cho nút F.

Bước 6: Tại nút F, một lần nữa giá trị của α sẽ được so sánh với nút con bên trái là 0 và max (3,0) = 3, sau đó so sánh với nút con bên phải là 1 và max (3,1) = 3 vẫn còn α vẫn là 3, nhưng giá trị nút của F sẽ trở thành 1.



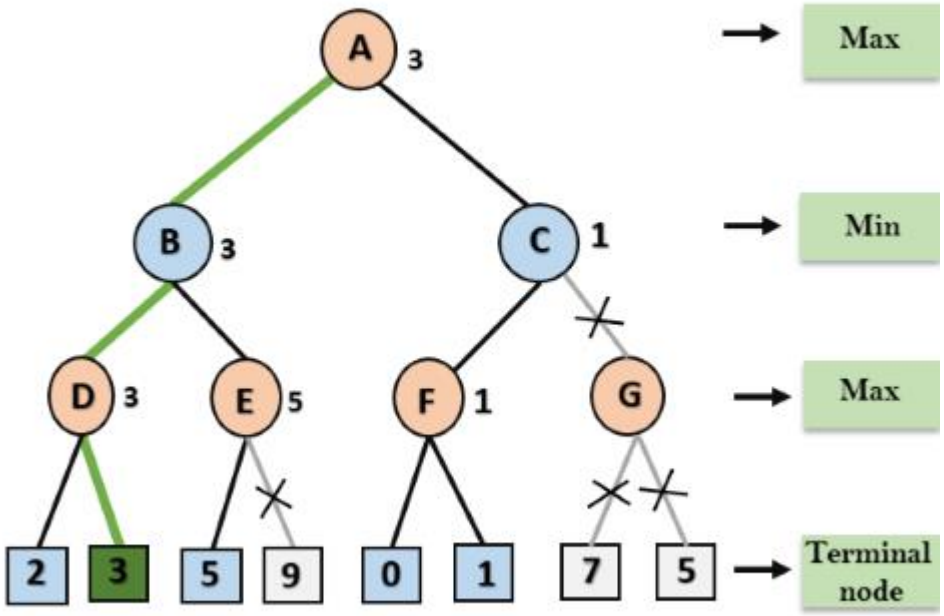
Hình 8 Mô phỏng thuật toán Alpha-Beta Search

Bước 7: Nút F trả về giá trị nút 1 cho nút C, tại C $\alpha = 3$ và $\beta = +\infty$, ở đây giá trị beta sẽ được thay đổi, nó sẽ so sánh với 1 nên min $(\infty, 1) = 1$. Bây giờ tại C, $\alpha = 3$ và $\beta = 1$, và một lần nữa nó thỏa mãn điều kiện $\alpha >= \beta$, vì vậy con tiếp theo của C là G sẽ bị lược bớt, và thuật toán sẽ không tính toàn bộ cây con G.



Hình 9 Mô phỏng thuật toán Alpha-Beta Search

Bước 8: C bây giờ trả về giá trị từ 1 đến A ở đây giá trị tốt nhất cho A là max (3, 1) = 3. Sau đây là cây trò chơi cuối cùng là hiển thị các nút được tính toán và các nút chưa bao giờ tính toán. Do đó, giá trị tối ưu cho bộ cực đại là 3 cho ví dụ này.



Hình 10 Mô phỏng thuật toán Alpha-Beta Search

Trong điều kiện lý tưởng, thuật toán Alpha-Beta Search chỉ phải xét số nút theo công thức:

$$2b^{\frac{d}{2}} - 1 \text{ với } d \text{ chẵn } (b \text{ là số nhánh}, d \text{ là chiều sâu})$$

$$b^{\frac{d+1}{2}} + b^{\frac{d}{2}} \text{ với } d \text{ lẻ } (b \text{ là số nhánh}, d \text{ là chiều sâu})$$

Ưu điểm:

- Cắt tỉa cây tìm kiếm giúp giảm đáng kể số lượng nút phải duyệt, tăng hiệu suất của thuật toán.
- Có thể tìm kiếm sâu hơn và xác định các hành động tối ưu nhanh hơn so với các thuật toán tìm kiếm thông thường.
- Độ phức tạp thời gian của thuật toán Alpha-Beta tốt hơn so với tìm kiếm theo chiều rộng và tìm kiếm theo chiều sâu vô hạn.

4. Thuật toán Monte Carlo

MCTS là một phương pháp tìm kiếm sử dụng lấy mẫu và mô phỏng ngẫu nhiên để ước tính tỷ lệ thắng thua của một trạng thái bàn cờ, từ đó xác định nước đi tốt nhất, đồng thời cân bằng giữa việc khám phá các nước đi mới và khai thác các nước đi tiềm năng.

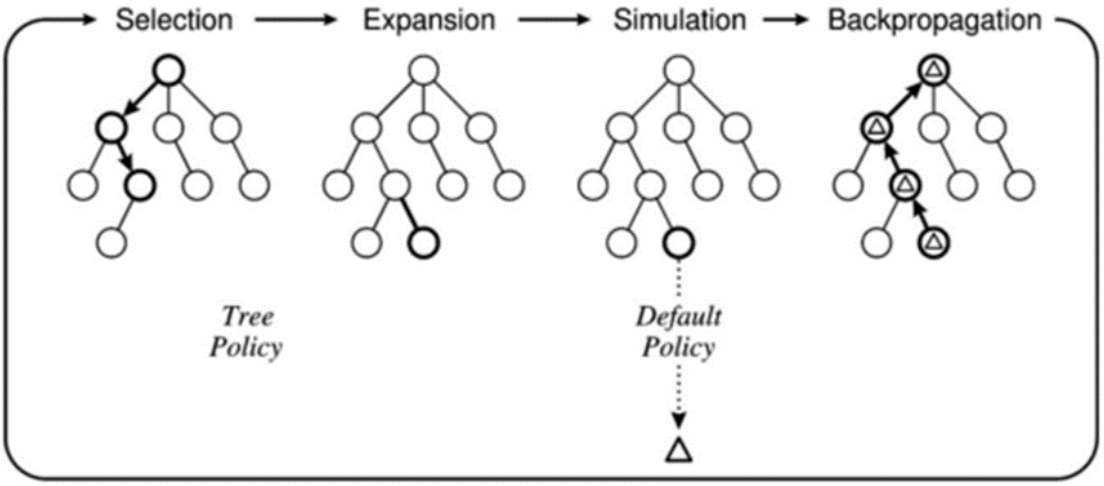
- Chọn lựa: từ một nút gốc (trạng thái bàn cờ hiện hành) cho đến nút lá, vì vậy sẽ có nhiều hướng đi được mang ra đánh giá.

- Mở rộng: thêm một nút con vào nút lá của hướng được chọn trong bước chọn lựa

- Giả lập: một ván cờ giả lập được chơi từ nút mở rộng, sau đó kết quả thắng thua của ván cờ giả lập sẽ được xác định. Để thuật toán nhanh hội tụ hơn (nhanh có kết quả thắng thua hơn) thì ta dùng thuật toán greedy để mô phỏng.

- Lan truyền ngược: kết quả thắng thua sẽ được cập nhật cho tất cảcác nút của hướng được chọn theo cách lan truyền ngược

Nước đi hứa hẹn là nút con nào có tỷ lệ thắng thua cao hơn được chọn trong giai đoạn chọn lựa. Sau khi kết thúc thời gian tìm kiếm, nút con nào được thăm nhiều nhất tại nút gốc sẽ được chọn để đi.



Hình 11 Mô phỏng thuật toán MCTS

- Bước 1: Từ trạng thái ban đầu của bàn cờ, tạo node gốc v_0 của trạng thái s_0 .Lặp lại quá trình xây dựng cây tìm kiếm trong phạm vi ngân sách tính toán (số lần lặp).Mô tả các kịch bản thử nghiệm và kết quả đạt được. Chọn một nút v từ gốc v_0 bằng cách sử dụng hàm Selection.

Công thức lựa chọn nút con $UCB_i = \frac{w_i}{n_i} + C\sqrt{\frac{lnN}{n_i}}$

Tỷ số wi/ni là tỷ lệ thắng của nút con i, wi là số lần thắng ván đấu giả lập có đi qua nút này, ni là số lần nút này được đi qua, N là số lần nút cha được đi qua, và C là tham số có thể điều chỉnh.

- Bước 2: Mở rộng nút v bằng cách chọn một hành động ngẫu nhiên và thêm nút con v_1
- Bước 3: Mô phỏng từ trạng thái của nút vl và trả về phần thưởng reward.
- Bước 4: Lan truyền ngược phần thưởng từ nút v_1 về nút gốc v_0.

Trả về nút con được thăm nhiều nhất từ gốc v_0

Nhược điểm:

- Trong một số nghiên cứu và triển khai, số lần lặp cần thiết cho thuật toán MCTS trong cờ tướng có thể nằm trong khoảng từ vài nghìn lần đến hàng triệu lần tùy thuộc

vào cấu hình cụ thể của thuật toán và mục tiêu hiệu suất. Nên nếu chỉ lặp số lượng nhỏ, nước đi sẽ không tốt

- Dù MCTS có thể làm việc không cần đến tri thức đặc trưng của trò chơi. Tuy nhiên, tri thức rất cần thiết để cải tiến các phương pháp học tăng cường, và học tăng cường là một phần của MCTS.
- Sử dụng thuật toán Greedy để thực hiện mô phỏng các nước đi, điều này có thể dẫn đến việc giảm tính đa dạng và khả năng khám phá của thuật toán MCTS. Mô phỏng bằng Greedy có thể bỏ qua các chiến lược quan trọng và không thể nắm bắt được tất cả các tình huống trong trò chơi.
- Hàm generate_next_states(state) chỉ tạo ra các trạng thái con bằng cách di chuyển các quân cờ một cách ngẫu nhiên mà không xem xét các chiến thuật và chiến lược cụ thể. Điều này có thể làm giảm tính đa dạng và khả năng khám phá của thuật toán MCTS.
- Hàm backpropagate(node, reward) đơn giản tăng lượt thăm và cộng điểm thưởng cho tất cả các nút trên đường đi từ nút lá về nút gốc mà không xem xét mức độ quan trọng của từng bước di chuyển cụ thể.

Đề xuất giải pháp:

- Sử dụng một mô hình học sâu như mạng nơ-ron tích chập (CNN) để đánh giá trạng thái của bàn cờ. Mạng nơ-ron có thể học được các đặc trưng quan trọng của bàn cờ và các vị trí quân cờ, giúp cải thiện độ chính xác của việc đánh giá và dự đoán kết quả của các nước đi. (mô hình CNN và mô hình VGG19 để đánh giá trạng thái của bàn cờ)
- Thay vì sử dụng Greedy, sử dụng một thuật toán mô phỏng cải tiến như AlphaZero để đánh giá các nước đi một cách chi tiết và hiệu quả hơn.

5. Chiến thuật [1]

```
# Quân tốt đen
b_pawn_table = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, -2, 0, 4, 0, -2, 0, 0],
    [2, 0, 8, 0, 8, 0, 8, 0, 2],
    [6, 12, 18, 18, 20, 18, 18, 12, 6],
    [10, 20, 30, 34, 40, 34, 30, 20, 10],
    [14, 26, 42, 60, 80, 60, 42, 26, 14],
    [18, 36, 56, 80, 120, 80, 56, 36, 18],
    [0, 3, 6, 9, 12, 9, 6, 3, 0]

]
```

Hình 12 Bảng trọng số quân tốt đen

```
# Quân mã đen
b_horse_table = [
    [0, -4, 0, 0, 0, 0, 0, -4, 0],
    [0, 2, 4, 4, -2, 4, 4, 2, 0],
    [4, 2, 8, 8, 4, 8, 8, 2, 4],
    [2, 6, 8, 6, 10, 6, 8, 6, 2],
    [4, 12, 16, 14, 12, 14, 16, 12, 4],
    [6, 16, 14, 18, 16, 18, 14, 16, 6],
    [8, 24, 18, 24, 20, 24, 18, 24, 8],
    [12, 14, 16, 20, 18, 20, 16, 14, 12],
    [4, 10, 28, 16, 8, 16, 28, 10, 4],
    [4, 8, 16, 12, 4, 12, 16, 8, 4]

]
```

Hình 13 Bảng trọng số quân mã đen

```
# Quân xe đen
b_chariot_table = [
    [-2, 10, 6, 14, 12, 14, 6, 10, -2],
    [8, 4, 8, 16, 8, 16, 8, 4, 8],
    [4, 8, 6, 14, 12, 14, 6, 8, 4],
    [6, 10, 8, 14, 14, 14, 8, 10, 6],
    [12, 16, 14, 20, 20, 20, 14, 16, 12],
    [12, 14, 12, 18, 18, 18, 12, 14, 12],
    [12, 18, 16, 22, 22, 22, 16, 18, 12],
    [12, 12, 12, 18, 18, 18, 12, 12, 12],
    [16, 20, 18, 24, 26, 24, 18, 20, 16],
    [14, 14, 12, 18, 16, 18, 12, 14, 14]
]
```

Hình 14 Bảng trọng số quân xe đen

```
# Quân pháo đen
b_cannon_table = [
    [0, 0, 2, 6, 6, 6, 2, 0, 0],
    [0, 2, 4, 6, 6, 6, 4, 2, 0],
    [4, 0, 7, 6, 10, 6, 7, 0, 4],
    [0, 0, 0, 2, 4, 2, 0, 0, 0],
    [-2, 0, 4, 2, 6, 2, 4, 0, -2],
    [0, 0, 0, 2, 8, 2, 0, 0, 0],
    [0, 0, -2, 4, 10, 4, -2, 0, 0],
    [2, 2, 0, -10, -8, -10, 0, 2, 2],
    [2, 2, 0, -4, -14, -4, 0, 2, 2],
    [6, 4, 0, -10, -12, -10, 0, 4, 6]
]
```

Hình 15 Bảng trọng số quân pháo đen

Bảng trọng số các quân tốt đỏ, mã đỏ, xe đỏ, pháo đỏ được lấy đảo ngược lại tương ứng

Ngoài ra, giá trị cố định của mỗi quân cờ là: tốt 30, xe 600, pháo 285, mã 270, tượng 120, sĩ 120, tướng 6000

```
def get_piece_value(piece, x, y):
        return 0

    def get_value(piece, x, y):
        if piece == 'to11' or piece == 'to21' or piece == 'to31' or piece == 'to41' or piece == 'to51':  # tốt đỏ
            return -(30 + r_pawn_table[x][y])
        elif piece == 'x11' or piece == 'x21':  # xe đỏ
            return -(600 + r_chariot_table[x][y])
        elif piece == 'p11' or piece == 'p21':  # pháo đỏ
            return -(285 + r_cannon_table[x][y])
        elif piece == 'm11' or piece == 'm21':  # mã đỏ
            return -(270 + r_horse_table[x][y])
        elif piece == 'tj11' or piece == 'tj21':  # tượng đỏ
            return -120
        elif piece == 's11' or piece == 's21':  # sỹ đỏ
            return -120
        elif piece == 'ts1':  # tướng đỏ
            return -6000
```

```
        if piece == 'to12' or piece == 'to22' or piece == 'to32' or piece == 'to42' or piece == 'to52':  # tốt đen
            return 30 + b_pawn_table[x][y]
        elif piece == 'x12' or piece == 'x22':  # xe đen
            return 600 + b_chariot_table[x][y]
        elif piece == 'p12' or piece == 'p22':  # pháo đen
            return 285 + b_cannon_table[x][y]
        elif piece == 'm12' or piece == 'm22':  # mã đen
            return 270 + b_horse_table[x][y]
        elif piece == 'tj12' or piece == 'tj22':  # tượng đen
            return 120
        elif piece == 's12' or piece == 's22':  # sỹ đen
            return 120
        elif piece == 'ts2':  # tướng đen
            return 6000


    if piece != "":
        value = get_value(piece, x, y)
        return value

    return 0
```

Hình 16 Hàm tính giá trị của 1 quân cờ tại 1 vị trí trên bàn cờ

```
2 usages
def heuristic(board):
    total_evaluation = 0
    for i in range(10):
        for j in range(9):
            total_evaluation += get_piece_value(board[i][j], i, j)
    return total_evaluation
```

Hình 17 Hàm heuristic tính tổng trọng số tất cả quân trên bàn cờ

Chương 3. THIẾT KẾ GIẢI PHÁP

1. Thuật toán Greedy

```python
def greedy_best_move(player, state):
    best_move = None
    best_weight = float('-inf') if player == 2 else float('inf')

    possible_moves = all_posible_moves(player, state.board)  # Xem xét lượt của người chơi hiện tại

    for (piece, old_position), moves in possible_moves.items():
        for new_position in moves[1]:
            # Tạo ra trạng thái bàn cờ mới
            new_state = GameState([row[:] for row in state.board])
            new_state.next_state(old_position, new_position)

            # Tính trọng số của trạng thái mới
            weight = heuristic(new_state.board)

            # Cập nhật nước đi tốt nhất
            if (player == 2 and weight > best_weight):
                best_weight = weight
                best_move = (piece, old_position, new_position)

            if (player == 1 and weight < best_weight):
                best_weight = weight
                best_move = (piece, old_position, new_position)

    state.next_state(best_move[1], best_move[2])
    return state
```

Hình 18 Thuật toán Greedy

2. Thuật toán Minimax

```
def Minimax(state: GameState, depth, isMax):
    if(depth == 0):
        return heuristic(state.board)
    moves = all_posible_moves(2 if isMax == True else 1, state.board)
    if(isMax):
        bestMove = -9999
        move_key = list(moves.keys())
        move_value = list(moves.values())
        for i in range(len(move_value)):
            for j in range(len(move_value[i][1])):
                movePieceName = move_key[i][0]
                curMove = list(move_key[i][1])
                nextMove = list(move_value[i][1][j])
                temp = state.board[nextMove[0]][nextMove[1]]
                state.board[nextMove[0]][nextMove[1]] = state.board[curMove[0]][curMove[1]]
                state.board[curMove[0]][curMove[1]] = ""
                eval = Minimax(state,depth - 1, not isMax)
                state.board[curMove[0]][curMove[1]] = state.board[nextMove[0]][nextMove[1]]
                state.board[nextMove[0]][nextMove[1]] = temp
                bestMove = max(bestMove,eval)
        return bestMove
    else:
        bestMove = 9999
        move_key = list(moves.keys())
        move_value = list(moves.values())
        for i in range(len(move_value)):
            for j in range(len(move_value[i][1])):
                movePieceName = move_key[i][0]
                curMove = list(move_key[i][1])
                nextMove = list(move_value[i][1][j])
                temp = state.board[nextMove[0]][nextMove[1]]
                state.board[nextMove[0]][nextMove[1]] = state.board[curMove[0]][curMove[1]]
                state.board[curMove[0]][curMove[1]] = ""
                eval = Minimax(state,depth - 1, not isMax)
                state.board[curMove[0]][curMove[1]] = state.board[nextMove[0]][nextMove[1]]
                state.board[nextMove[0]][nextMove[1]] = temp
                bestMove = min(bestMove,eval)
        return bestMove
```

Hình 19 Hàm Minimax

```python
def find_best_move_minimax(state : GameState,depth,isMax):
    if isMax:
        bestMove = None
        bestValue = -9999
        moves = all_posible_moves(2 if isMax == True else 1, state.board)
        move_key = list(moves.keys())
        move_value = list(moves.values())
        for i in range(len(move_value)):
            for j in range(len(move_value[i][1])):
                movePieceName = move_key[i][0]
                curMove = list(move_key[i][1])
                nextMove = list(move_value[i][1][j])
                temp = state.board[nextMove[0]][nextMove[1]]
                state.board[nextMove[0]][nextMove[1]] = state.board[curMove[0]][curMove[1]]
                state.board[curMove[0]][curMove[1]] = ""
                move_point = Minimax(state, depth - 1, not isMax)
                state.board[curMove[0]][curMove[1]] = state.board[nextMove[0]][nextMove[1]]
                state.board[nextMove[0]][nextMove[1]] = temp
                if move_point >= bestValue:
                    bestValue = move_point
                    bestMove = [curMove, nextMove]
        return bestMove
```

```python
    else:
        bestMove = None
        bestValue = 9999
        moves = all_posible_moves(2 if isMax == True else 1, state.board)
        move_key = list(moves.keys())
        move_value = list(moves.values())
        for i in range(len(move_value)):
            for j in range(len(move_value[i][1])):
                movePieceName = move_key[i][0]
                curMove = list(move_key[i][1])
                nextMove = list(move_value[i][1][j])
                temp = state.board[nextMove[0]][nextMove[1]]
                state.board[nextMove[0]][nextMove[1]] = state.board[curMove[0]][curMove[1]]
                state.board[curMove[0]][curMove[1]] = ""
                move_point = Minimax(state, depth - 1, not isMax)
                state.board[curMove[0]][curMove[1]] = state.board[nextMove[0]][nextMove[1]]
                state.board[nextMove[0]][nextMove[1]] = temp
                if move_point <= bestValue:
                    bestValue = move_point
                    bestMove = [curMove, nextMove]
        return bestMove
```

Hình 20 Hàm find_best_move_minimax

Giải thích: Khi đến lượt máy đánh, chương trình sẽ gọi hàm find_best_move_minimax với đầu vào gồm trạng thái hiện tại của trò chơi, độ sâu của cây tìm kiếm (số nguyên lớn hơn 0), máy là Max hay Min (True hoặc False). Hàm sẽ trả về nước đi hiện tại và nước đi kế tiếp của quân cờ sẽ được di chuyển của trạng thái tốt nhất mà nó tìm được nhờ áp dụng thuật toán Minimax.

3. Triển khai thuật toán Alpha-Beta Search

```
def find_best_move(state : GameState,depth,isMax):
    if isMax:
        bestMove = None
        bestValue = -9999
        moves = all_posible_moves(2 if isMax == True else 1, state.board)
        move_key = list(moves.keys())
        move_value = list(moves.values())
        for i in range(len(move_value)):
            for j in range(len(move_value[i][1])):
                movePieceName = move_key[i][0]
                curMove = list(move_key[i][1])
                nextMove = list(move_value[i][1][j])
                temp = state.board[nextMove[0]][nextMove[1]]
                state.board[nextMove[0]][nextMove[1]] = state.board[curMove[0]][curMove[1]]
                state.board[curMove[0]][curMove[1]] = ""
                move_point = AB(state, depth - 1, -99999, 99999, not isMax)
                state.board[curMove[0]][curMove[1]] = state.board[nextMove[0]][nextMove[1]]
                state.board[nextMove[0]][nextMove[1]] = temp
                if move_point >= bestValue:
                    bestValue = move_point
                    bestMove = [curMove, nextMove]
        return bestMove
```

```
    else:
        bestMove = None
        bestValue = 9999
        moves = all_posible_moves(2 if isMax == True else 1, state.board)
        move_key = list(moves.keys())
        move_value = list(moves.values())
        for i in range(len(move_value)):
            for j in range(len(move_value[i][1])):
                movePieceName = move_key[i][0]
                curMove = list(move_key[i][1])
                nextMove = list(move_value[i][1][j])
                temp = state.board[nextMove[0]][nextMove[1]]
                state.board[nextMove[0]][nextMove[1]] = state.board[curMove[0]][curMove[1]]
                state.board[curMove[0]][curMove[1]] = ""
                move_point = AB(state, depth - 1, -99999, 99999, not isMax)
                state.board[curMove[0]][curMove[1]] = state.board[nextMove[0]][nextMove[1]]
                state.board[nextMove[0]][nextMove[1]] = temp
                if move_point <= bestValue:
                    bestValue = move_point
                    bestMove = [curMove, nextMove]
        return bestMove
```

Hình 21 Hàm find_best_move

```python
def AB(state: GameState, depth, alpha, beta, isMax):
    if(depth == 0):
        return heuristic(state.board)
    moves = all_posible_moves(2 if isMax == True else 1, state.board)
    if(isMax):
        bestMove = -9999
        move_key = list(moves.keys())
        move_value = list(moves.values())
        for i in range(len(move_value)):
            for j in range(len(move_value[i][1])):
                movePieceName = move_key[i][0]
                curMove = list(move_key[i][1])
                nextMove = list(move_value[i][1][j])
                temp = state.board[nextMove[0]][nextMove[1]]
                state.board[nextMove[0]][nextMove[1]] = state.board[curMove[0]][curMove[1]]
                state.board[curMove[0]][curMove[1]] = ""
                eval = AB(state,depth - 1,alpha,beta, not isMax)
                state.board[curMove[0]][curMove[1]] = state.board[nextMove[0]][nextMove[1]]
                state.board[nextMove[0]][nextMove[1]] = temp
                bestMove = max(bestMove,eval)
                alpha = max(alpha,bestMove)
                if beta <= alpha :
                    return bestMove
        return bestMove
```

```python
    else:
        bestMove = 9999
        move_key = list(moves.keys())
        move_value = list(moves.values())
        for i in range(len(move_value)):
            for j in range(len(move_value[i][1])):
                movePieceName = move_key[i][0]
                curMove = list(move_key[i][1])
                nextMove = list(move_value[i][1][j])
                temp = state.board[nextMove[0]][nextMove[1]]
                state.board[nextMove[0]][nextMove[1]] = state.board[curMove[0]][curMove[1]]
                state.board[curMove[0]][curMove[1]] = ""
                eval = AB(state,depth - 1,alpha,beta, not isMax)
                state.board[curMove[0]][curMove[1]] = state.board[nextMove[0]][nextMove[1]]
                state.board[nextMove[0]][nextMove[1]] = temp
                bestMove = min(bestMove,eval)
                beta = min(beta,bestMove)
                if beta <= alpha :
                    return bestMove
        return bestMove
```

Hình 22 Hàm Alpha-Beta Search

Giải thích: Khi đến lượt máy đánh, chương trình sẽ gọi hàm find_best_move với đầu vào gồm trạng thái hiện tại của trò chơi, độ sâu của cây tìm kiếm (số nguyên lớn hơn 0), máy là Max hay Min (True hoặc False). Hàm sẽ trả về nước đi hiện tại và nước đi kế tiếp của quân cờ sẽ được di chuyển của trạng thái tốt nhất mà nó tìm được nhờ áp dụng thuật toán Alpha-Beta Seatch

4. Triển khai thuật toán Monte Carlo Tree Search

```python
class Node:
    def __init__(self, state, parent):
        self.state = state
        self.parent = parent
        self.children = []
        self.visits = 0
        self.score = 0

    def expand(self):
        next_states = generate_next_states(self.state)
        for state in next_states.values():
            self.children.append(Node(state, parent=self))

    2 usages
    def is_fully_expanded(self):
        return len(self.children) == len(generate_next_states(self.state))
```

```python
    1 usage
    def select_child(self, exploration_constant=1.4):
        if not self.children:
            return None
        for child in self.children:
            if child.visits == 0:
                return child
        best_child = None
        best_score = float('-inf')
        for child in self.children:
            exploitation = child.score / child.visits
            if self.parent is not None:
                exploration = math.sqrt(2 * math.log(self.parent.visits) / child.visits)
            else:
                exploration = 0  # No exploration term for the root node
            score = exploitation + exploration
            if score > best_score:
                best_child = child
                best_score = score
        return best_child
```

Hình 23 Hàm định nghĩa Node trong MCTS

```python
def generate_next_states(state):
    next_states = {}
    board = state.board
    possible_moves = all_posible_moves( team: 2, board)
    for piece, moves in possible_moves.items():
        moves = moves[1]
        for move in moves:
            new_board = copy.deepcopy(board)
            old_position = piece[1]
            next_position = move
            new_board[next_position[0]][next_position[1]] = piece[0]
            new_board[old_position[0]][old_position[1]] = ''
            next_states[(piece, old_position)] = GameState(new_board)
    return next_states
1 usage
def simulate(state):
    copy_state = copy.deepcopy(state)
    while not is_terminal(copy_state):
        player = 1 if copy_state.current_player == 'red' else 2
        copy_state = greedy_best_move(player, copy_state)
        copy_state.current_player = 'red' if player == 2 else 'black'
        if is_draw(copy_state):
            return -1000
    if is_king_captured(copy_state.board,  player: 1):
        return 1 # Người chơi 2 thắng
    elif is_king_captured(copy_state.board,  player: 2):
        return -2000 # Người chơi 1 thắng
```

Hình 24 Hàm tạo các nước đi mới vào Node và mô phỏng

```python
def evaluate_state(state):
    total_evaluation = 0
    for i in range(10):
        for j in range(9):
            total_evaluation += get_piece_value(state.board[i][j], i, j)
    return total_evaluation

4 usages
def is_king_captured(board, player):
    king_piece = f"ts{player}"
    for row in board:
        if king_piece in row:
            return False
    return True

1 usage
def is_draw(state):
    return (not any(all_posible_moves( team: 1, state.board).values()) and
            not any(all_posible_moves( team: 2, state.board).values())) or state.num_moves >= 30

1 usage
def is_terminal(state):
    board = state.board
    return is_king_captured(board,  player: 1) or is_king_captured(board,  player: 2)
```

Hình 25 Hàm định nghĩa các trạng thái bàn cờ

```python
def backpropagate(node, score):
    while node is not None:
        node.visits += 1
        node.score += score
        node = node.parent
2 usages
def mcts(state, num_iterations):
    root = Node(state,  parent: None)
    root.expand()
    for _ in range(num_iterations):
        node = root
        while node.is_fully_expanded():
            node = node.select_child()
            if not node:
                break
        if not node.is_fully_expanded():
            node.expand()
        if node.children:
            # Chọn 5 node con có state với giá trị greedy cao nhất
            top_children = heapq.nlargest( n: 3, node.children, key=lambda child: evaluate_state(child.state))
            if top_children:
                selected_child = random.choice(top_children)
                score = simulate(selected_child.state)
                backpropagate(selected_child, score)
    if root.children:
        best_child = max(root.children, key=lambda child: child.visits)
        return best_child.state
    else: return state
```

Hình 26 Hàm lan truyền ngược và thuật toán MCTS

Chương 4. TRIỂN KHAI THỬ NGHIỆM VÀ ĐÁNH GIÁ

1. Thử nghiệm về chất lượng nước đi

Sau khi thử nghiệm bằng cách tạo các ván cờ mô phỏng giữa các thuật toán với nhau, nhóm đã rút ra được thứ tự về chất lượng nước đi mà các thuật toán đã tìm ra. Một thuật toán có chất lượng nước đi tốt hơn sẽ có tỉ lệ thắng áp đảo so với thuật toán có chất lượng kém hơn. Các thuật toán có chất lượng bằng nhau thì sẽ đa số là hoà nhau.

Bảng sau đây sắp xếp theo thứ tự từ kém đến tốt của các thuật toán

Bảng 1 So sánh chất lượng nước đi của các thuật toán

STT	Strategy	Search Depth
1	Greedy	1
2	Minimax / Alpha-Beta Pruning	2
3	Minimax / Alpha-Beta Pruning	3
4	Minimax / Alpha-Beta Pruning	4

Đối với cùng 1 thuật toán, việc tăng giới hạn độ sâu của cây tìm kiếm sẽ giúp tìm ra được những nước đi chất lượng hơn.

Do các nhược điểm của MCTS nên nhóm không thể sử dụng ngay để so sánh được. Chúng ta sẽ coi MCTS là một trường hợp đặc biệt vì nó có khả năng học và cải thiện chất lượng của mình qua thời gian.

2. Thử nghiệm về thời gian cần thiết để sinh ra nước đi

Sau khi thử nghiệm bằng cách tạo ra các ván cờ mô phỏng của các thuật toán, nhóm đã đo được thời gian trung bình mà các thuật toán cần để tạo ra một nước đi mới.

Bảng 2 So sánh thời gian sinh ra nước đi của các thuật toán

Strategy	Search Depth (Iterations if MCTS)	Average Time for Each Move (ms)
Greedy	1	0.89
Minimax	2	59.90
Minimax	3	2207.20
Minimax	4	83931.38
Alpha-Beta Pruning	2	64.07
Alpha-Beta Pruning	3	787.36
Alpha-Beta Pruning	4	6618.08
MCTS	10	240.86
MCTS	50	1135.93
MCTS	100	2481.92
MCTS	500	16568.29

Với cùng các thuật toán, việc tăng giới hạn độ sâu của cây tìm kiếm sẽ tăng thêm số lượng các nút cần phải duyệt, từ đó tăng thêm thời gian cần thiết để sinh ra một nước cờ.

Mặc dù Minimax và Alpha-Beta Pruning đều sinh ra các nước cờ có cùng một chất lượng. Nhưng Alpha-Beta Pruning có hiệu suất tốt hơn đáng kể, thời gian sinh ra nước của thuật toán này nhanh hơn gấp nhiều lần thời gian của Minimax, nhất là ở những độ sâu lớn. Ví dụ, ở độ sâu bằng 4, Alpha-Beta Pruning chỉ cần 6,6 giây để tạo ra một nước đi mới, trong khi đó Minimax cần tận 83,9 giây, lâu hơn gấp 12,7 lần.

KẾT LUẬN

- Nhóm đã cài đặt thành công các thuật toán, từ đó chạy thử nghiệm và so sánh hiệu suất và chất lượng các thuật toán với nhau, giúp cho các thành viên củng cố kiến thức và hiểu hơn về các thuật toán chơi cờ nói riêng và về trí tuệ nhân tạo nói chung.

- Tuy nhóm đã cài thành công thuật toán MCTS nhưng chưa thể huấn luyện để có thể nâng cao hiệu suất của nó được.

- Trong tương lai gần, nhóm sẽ nghiên cứu và áp dụng một mô hình học sâu như mạng nơ-ron tích chập (CNN) để đánh giá trạng thái của bàn cờ. Mạng nơ-ron có thể học được các đặc trưng quan trọng của bàn cờ và các vị trí quân cờ, giúp cải thiện độ chính xác của việc đánh giá và dự đoán kết quả của các nước đi. (mô hình CNN và mô hình VGG19 để đánh giá trạng thái của bàn cờ) hoặc là sử dụng một thuật toán mô phỏng cải tiến như AlphaZero để đánh giá các nước đi một cách chi tiết và hiệu quả hơn.

TÀI LIỆU THAM KHẢO

[1] Yen, S.-J. a. Chen, J.-C. a. Yang, T.-N. a. Hsu and Shun-Chin, "Computer Chinese Chess," *ICGA journal,* pp. 3-18, 2004.