

**BỘ GIÁO DỤC VÀ ĐÀO TẠO TRƯỜNG ĐẠI HỌC SÀI GÒN KHOA CÔNG NGHỆ THÔNG TIN**

**Môn: Trí tuệ nhân tạo nâng cao**

**Lab03: N-Queens & Traveling Salesman Problem**

Giảng viên: Đỗ Như Tài

Sinh viên:

3122410035 - Nguyễn Khải Ca

3122410040 - Đặng Văn Chiến

3120410098 - Đỗ Trịnh Mỹ Duyên

3122410040 - Đặng Văn Chiến

3120410098 - Đỗ Trịnh Mỹ Duyên

**PHÂN CÔNG CÔNG VIỆC**

|  |  |
| --- | --- |
| 3122410035 -  Nguyễn Khải Ca | N-Queens Problem  Và Word |
| 3122410040  -  Đặng Văn Chiến | Traveling Salesman  Problem và slide báo cáo |
| 3120410098 - Đỗ Trịnh Mỹ Duyên | Traveling Salesman  Problem và Word |

Mục lục

[1. BÀI TOÁN NGƯỜI BÁN HÀNG (TRAVELING SALESMAN PROBLEM) SỬ DỤNG THUẬT TOÁN TÌM KIẾM CỤC BỘ 4](#_Toc211161263)

[I.Nội dung được trình bày trong notebook 4](#_Toc211161264)

[1.1Tổng quan về bài toán(TSP) 4](#_Toc211161265)

[1.2Các hàm hỗ trợ 4](#_Toc211161266)

[II. Các thuật toán đã triển khai 4](#_Toc211161267)

[2.1.Steepest-Ascent Hill Climbing (Leo đồi dốc nhất) 4](#_Toc211161268)

[2.2. Steepest-Ascent Hill Climbing with Random Restarts 4](#_Toc211161269)

[3. Stochastic Hill Climbing 4](#_Toc211161270)

[4. First-Choice Hill Climbing 4](#_Toc211161271)

[5. Simulated Annealing (Luyện kim Mô phỏng) 4](#_Toc211161272)

[6. Bonus: Genetic Algorithm (Thuật toán Di truyền) 4](#_Toc211161273)

[IV.So sánh kết quả 4](#_Toc211161274)

[VI. Kết luận 4](#_Toc211161275)

[2. Bài toán N-Queens (N – Quân hậu) 4](#_Toc211161276)

[I. Mô tả bài toán 4](#_Toc211161277)

[II. Các hàm hỗ trợ 4](#_Toc211161278)

[1. random\_board(n) 4](#_Toc211161279)

[**2. comb2(n)** 4](#_Toc211161280)

[**3. conflicts(board)** 4](#_Toc211161281)

[**4. show\_board(board, cols=['white', 'gray'], fontsize=48)** 4](#_Toc211161282)

[III. Các thuật toán giải quyết bài toán 4](#_Toc211161283)

[1. Steepest-ascend Hill Climbing Search 4](#_Toc211161284)

[2. Stochastic Hill Climbing 1 6](#_Toc211161285)

[3. Sochatis Hill Climbing 2 6](#_Toc211161286)

[4. Hill Climbing Search with Random Restarts 8](#_Toc211161287)

[Tham khảo 8](#_Toc211161288)

# 1. BÀI TOÁN NGƯỜI BÁN HÀNG (TRAVELING SALESMAN PROBLEM) SỬ DỤNG THUẬT TOÁN TÌM KIẾM CỤC BỘ

## I.Nội dung được trình bày trong notebook

### 1.1Tổng quan về bài toán(TSP)

Mục tiêu: Tìm một chu trình (tour) ngắn nhất đi qua tất cả các thành phố, mỗi thành phố đúng một lần, và quay trở lại điểm xuất phát.

Không gian trạng thái (State Space): Mỗi trạng thái là một hoán vị π của các thành phố, biểu thị thứ tự di chuyển.

Hàm mục tiêu (Objective Function): Tối thiểu hóa tổng độ dài của chu trình:

A black and white math equation

AI-generated content may be incorrect.

Phép toán lân cận (Local Moves): Cách để chuyển từ trạng thái này sang trạng thái khác. Trong bài, phép toán được định nghĩa là hoán đổi vị trí của hai thành phố trong chu trình (2-opt swap).

### 1.2Các hàm hỗ trợ

Để chuẩn bị cho việc triển khai thuật toán, notebook cung cấp sẵn các hàm tiện ích bằng Python, giúp đơn giản hóa quá trình làm bài:

* random\_tour(n): Tạo ra một chu trình ngẫu nhiên ban đầu gồm n thành phố.
* random\_tsp(n): Tạo một bài toán TSP ngẫu nhiên với n thành phố, bao gồm tọa độ và ma trận khoảng cách Euclid giữa chúng.
* tour\_length(tsp, tour): Tính tổng độ dài của một chu trình cho trước.
* show\_tsp(tsp, tour): Trực quan hóa vị trí các thành phố và chu trình đã tìm được bằng matplotlib.

## II. Các thuật toán đã triển khai

Phần bài tập đã hoàn thành việc triển khai 5 thuật toán tìm kiếm cục bộ khác nhau để giải quyết bài toán TSP với 10 thành phố được tạo ngẫu nhiên.

### 2.1.Steepest-Ascent Hill Climbing (Leo đồi dốc nhất)

* **Mô tả:** Đây là thuật toán leo đồi cơ bản nhất. Tại mỗi bước, nó sẽ sinh ra **tất cả các trạng thái hàng xóm** (bằng cách hoán đổi 2 thành phố) và chọn trạng thái **tốt nhất** trong số đó để di chuyển đến. Quá trình dừng lại khi không có hàng xóm nào tốt hơn trạng thái hiện tại (đạt đỉnh cục bộ).
* **Kết quả:** Thuật toán đã tìm ra một lời giải với chi phí là **24** cho ví dụ 5 thành phố.
* **Nhận xét:** Đây là một thuật toán đơn giản, dễ cài đặt và chạy nhanh. Tuy nhiên, nhược điểm lớn của nó là rất dễ bị "kẹt" ở các đỉnh tối ưu cục bộ (local optima).

### 2.2. Steepest-Ascent Hill Climbing with Random Restarts

* **Mô tả:** Để khắc phục nhược điểm của thuật toán cơ bản, phiên bản này thực hiện lại thuật toán Steepest-Ascent nhiều lần (restarts=10). Mỗi lần bắt đầu từ một trạng thái ngẫu nhiên khác nhau. Lời giải cuối cùng là lời giải tốt nhất tìm được qua tất cả các lần chạy.
* **Kết quả:** Trong ví dụ, thuật toán này cũng tìm được lời giải tối ưu cục bộ với chi phí là **24**.
* **Nhận xét:** Phương pháp này tăng đáng kể khả năng tìm ra lời giải tốt hơn bằng cách khám phá nhiều vùng khác nhau trong không gian trạng thái, giúp thoát khỏi các đỉnh cục bộ không tốt.

### 3. Stochastic Hill Climbing

* **Mô tả:** Thay vì luôn chọn hàng xóm tốt nhất, thuật toán này tìm tất cả các hàng xóm tốt hơn (uphill moves) và **chọn ngẫu nhiên một** trong số đó.
* **Kết quả:** Thuật toán đã tìm ra lời giải tốt với chi phí là **22**.
* **Nhận xét:** Việc thêm yếu tố ngẫu nhiên giúp thuật toán có những bước đi đa dạng hơn trên bề mặt tìm kiếm, đôi khi có thể dẫn đến một đỉnh cục bộ tốt hơn so với cách tiếp cận "tham lam" của Steepest-Ascent.

### 4. First-Choice Hill Climbing

* **Mô tả:** Thuật toán này không sinh ra tất cả hàng xóm. Thay vào đó, nó sinh ngẫu nhiên từng hàng xóm một và **chấp nhận ngay hàng xóm đầu tiên** tìm thấy mà tốt hơn trạng thái hiện tại.
* **Kết quả:** Tìm được lời giải tốt với chi phí là **22**.
* **Nhận xét:** Đây là một biến thể hiệu quả về mặt tính toán, vì nó không cần duyệt qua toàn bộ không gian hàng xóm ở mỗi bước, giúp tiết kiệm thời gian, đặc biệt với các bài toán có số lượng hàng xóm lớn.

### 5. Simulated Annealing (Luyện kim Mô phỏng)

* **Mô tả:** Đây là một thuật toán phức tạp hơn, lấy cảm hứng từ quá trình luyện kim. Nó không chỉ chấp nhận các bước đi tốt hơn mà còn có thể chấp nhận các bước đi **tệ hơn** với một xác suất nhất định. Xác suất này phụ thuộc vào một tham số gọi là "nhiệt độ" (Temperature), giảm dần theo thời gian. Điều này giúp thuật toán có khả năng "nhảy" ra khỏi các đỉnh tối ưu cục bộ.
* **Kết quả:** Với ví dụ 8 thành phố, thuật toán đã tìm ra lời giải rất tốt với chiều dài **26.07**.
* **Nhận xét:** Simulated Annealing là một thuật toán mạnh mẽ và thường cho kết quả tốt hơn các phiên bản Hill Climbing đơn giản, dù việc tinh chỉnh các tham số (nhiệt độ ban đầu, tốc độ giảm) có thể phức tạp hơn.

### 6. Bonus: Genetic Algorithm (Thuật toán Di truyền)

* **Mô tả:** Bạn đã cài đặt thêm thuật toán Di truyền, một phương pháp tìm kiếm dựa trên quần thể, mô phỏng quá trình tiến hóa tự nhiên. Nó hoạt động thông qua các cơ chế **chọn lọc (selection)**, **lai ghép (crossover)**, và **đột biến (mutation)**.
* **Kết quả:** Với ví dụ 8 thành phố, thuật toán này cũng đã tìm được một lời giải xuất sắc với chiều dài **26.07**.
* **Nhận xét:** Thuật toán Di truyền thường rất mạnh mẽ cho các bài toán TSP kích thước lớn vì nó khám phá song song nhiều vùng của không gian tìm kiếm, ít bị mắc kẹt ở đỉnh cục bộ hơn so với các phương pháp leo đồi.

## IV.So sánh kết quả

Bạn đã thực hiện một so sánh hiệu năng rất tốt giữa các thuật toán dựa trên **chất lượng lời giải (chi phí tốt nhất)** và **thời gian thực thi**.

A close-up of a receipt

AI-generated content may be incorrect.

* **Kết quả:** Với bài toán 5 thành phố, tất cả các thuật toán đều tìm ra lời giải tốt (chi phí 22). Thời gian thực thi rất nhanh và không có sự chênh lệch đáng kể.
* **Bình luận:** Đối với một bài toán kích thước nhỏ (5 thành phố), hầu hết các thuật toán tìm kiếm cục bộ đều có thể nhanh chóng tìm ra lời giải tối ưu. Sự khác biệt về hiệu năng và chất lượng lời giải sẽ trở nên rõ rệt hơn nhiều khi tăng số lượng thành phố. Ví dụ, với 50 thành phố, Steepest-Ascent có thể bị kẹt rất nhanh, trong khi Simulated Annealing hoặc Random Restarts sẽ có cơ hội tìm ra lời giải tốt hơn nhiều

## Kết luận

Notebook cung cấp một cái nhìn tổng quan về thuật toán tìm kiếm cục bộ cho bài toán TSP. và chúng em đã thực hiện:

**Cài đặt thành công** 5 thuật toán tìm kiếm cục bộ khác nhau, từ đơn giản đến nâng cao.

**Cài đặt thêm** thuật toán Di truyền làm phần bonus.

**Thực hiện so sánh hiệu năng** một cách có hệ thống để đánh giá các thuật toán.

# 2. Bài toán N-Queens (N – Quân hậu)

## I. Mô tả bài toán

Goal State – Trạng thái đích: Sắp xếp N quân hậu trên bàn cờ Vua NxN (số hàng x số cột) sao cho không có bất kỳ quân hậu nào nằm trên cùng hàng, cột và đường chéo.

State space : Mỗi **trạng thái** là một cách sắp xếp N quân hậu. Tổng số trạng thái có thể: (vì mỗi trong N cột có N vị trí hàng). Khi giới hạn “mỗi cột 1 quân”, ta chỉ cần thay đổi vị trí hàng của từng quân hậu khi tìm kiếm.

Objective Function: minimize: conficts(q), subject to: q chứa mỗi quan hậu một cột

Ảnh có chứa Các trò chơi và môn thể thao trong nhà, trò chơi cờ bàn, Trò chơi, cờ vua

Nội dung do AI tạo ra có thể không chính xác.

## II. Các hàm hỗ trợ

### 1. random\_board(n)

* Chức năng: Sinh ra một bàn cờ ngẫu nhiên kích thước n × n.
* Cách hoạt động:  
  Tạo mảng 1 chiều có n phần tử, mỗi phần tử là vị trí hàng (0 → n−1) của quân hậu trong từng cột.  
  → Mỗi cột luôn có đúng 1 quân hậu.
* Ví dụ:  
  random\_board(4) có thể cho ra [1, 3, 0, 2].

**2. comb2(n)**

* Chức năng: Tính số cặp có thể chọn từ n phần tử, tức là “n chọn 2” = n(n−1)/2.
* Ý nghĩa: Dùng để tính số cặp quân hậu tấn công nhau trên cùng hàng hoặc đường chéo.

**3. conflicts(board)**

* Chức năng: Tính tổng số xung đột (conflicts) giữa các quân hậu trên bàn cờ — đây là hàm mục tiêu (objective function).
* Nguyên lý:
  + Duyệt qua toàn bộ các quân hậu.
  + Đếm số lượng quân hậu trên từng hàng ngang và hai đường chéo.
  + Sử dụng comb2() để tính số cặp xung đột trên mỗi hàng hoặc chéo.
* Kết quả trả về: Tổng số cặp quân hậu đang tấn công nhau.
* Ví dụ:  
  Nếu conflicts(board) == 0 → đã tìm được lời giải hợp lệ.

**4. show\_board(board, cols=['white', 'gray'], fontsize=48)**

* Chức năng: Hiển thị trực quan bàn cờ và các quân hậu bằng Matplotlib.
* Cách hoạt động:
  + Tạo lưới bàn cờ n×n với hai màu xen kẽ.
  + Vẽ biểu tượng quân hậu (♛ – mã Unicode \u265B) tại các vị trí trong mảng board.
  + In ra tổng số xung đột hiện tại (Board with X conflicts.).
  + Mục đích: Giúp quan sát trực quan kết quả sắp xếp của thuật toán.

**Sơ qua về Hill Climbing**: Hill Climbing là một thuật toán tìm kiếm heuristic thuộc nhóm local search methods (tìm kiếm cục bộ). Ý tưởng chính: bắt đầu từ một trạng thái ban đầu, sau đó lặp lại việc di chuyển sang một trạng thái láng giềng (neighbor) nếu nó cải thiện giá trị hàm đánh giá (heuristic / objective function). Cuối cùng thuật toán dừng lại khi không còn trạng thái láng giềng nào tốt hơn trạng thái hiện tại → đạt cực trị địa phương (local optimum).

**Cách thức hoạt động của Hill Climbing**

1. Khởi tạo (Initial State)  
   Chọn một trạng thái ban đầu — thường là ngẫu nhiên hoặc heuristic đơn giản.
2. Xác định các trạng thái láng giềng (Generate Neighbors)  
   Sinh hoặc liệt kê các trạng thái có thể “gần” trạng thái hiện tại bằng cách thay đổi nhỏ (tweak, mutation).
3. Đánh giá (Evaluate Neighbors)  
   Tính giá trị của hàm đánh giá (objective / heuristic) đối với mỗi neighbor.
4. Di chuyển đến trạng thái tốt hơn (Move to Better Neighbor)  
   Nếu tồn tại neighbor có giá trị tốt hơn trạng thái hiện tại, chuyển sang neighbor đó.
   * Với Steepest-Ascent Hill Climbing, ta xem xét *tất cả* các neighbor rồi chọn neighbor tốt nhất.
   * Với Simple Hill Climbing, ta có thể chọn neighbor đầu tiên tốt hơn khi gặp nó.
   * Với Stochastic Hill Climbing, có thể chọn neighbor ngẫu nhiên theo xác suất tùy thuộc mức độ tốt hơn.
5. Kiểm tra điều kiện dừng (Termination)  
   Thuật toán kết thúc khi:
   * Không có neighbor nào tốt hơn trạng thái hiện tại → đạt cực trị địa phương.
   * Hoặc đạt được giá trị tối ưu (nếu ta biết giá trị tốt nhất).

## III. Các thuật toán giải quyết bài toán

### 1.Steepest-ascend Hill Climbing Search

Thuật toán Steepest-Ascent Hill Climbing hoạt động theo nguyên tắc:Tại mỗi bước, từ trạng thái hiện tại (board hiện tại), xét tất cả các trạng thái láng giềng (neighbors) và chọn trạng thái có giá trị hàm mục tiêu tốt nhất (ít xung đột nhất) để di chuyển đến.

def steepest(board, verb = True):

    # create copy of board

    bc = np.copy(board)

    # while sol is true loop

    sol = True

    opt = False

    while sol:

        # create dict of neighbors with conflicts as key

        nbr\_dict = {}

        # create copy of board as neighbor board

        nbr = np.copy(bc)

        # nested for loop to loop through each column and row

        for i in range(len(bc)):

            for j in range(len(bc)):

                # change col i to value j

                nbr[i] = j

                # if conflicts less than board copy add to dict

                if conflicts(nbr) <= conflicts(bc):

                    nbr\_dict[conflicts(nbr)] = nbr

                # reestablish neighbor board to have same starting point

                nbr = np.copy(bc)

        # if local min found, stop

        if conflicts(bc) == min(nbr\_dict.keys()):

            if verb:

                print('local optimum found, global optimum not found')

            sol = False

        # board copy is the neighbor with the lowest conflicts

        bc = nbr\_dict[min(nbr\_dict.keys())]

        # if conflicts is zero, global optimum found, stop

        if conflicts(bc) == 0:

            if verb:

                print('global/local optimum found')

            opt = True

            sol = False

    return [opt,bc]

Cách hoạt động của đoạn code:

1. bc = np.copy(board)  
   → Sao chép bàn cờ hiện tại để tránh thay đổi dữ liệu gốc.
2. Tạo danh sách các trạng thái láng giềng (nbr\_dict):
   * Với mỗi cột i và mỗi hàng j, chương trình đặt lại vị trí quân hậu trong cột i ở hàng j.
   * Tính số conflicts (số cặp hậu tấn công nhau).
   * Nếu số xung đột của trạng thái mới ≤ trạng thái hiện tại, lưu vào nbr\_dict.
3. Chọn trạng thái có ít xung đột nhất:
   * Nếu không có trạng thái nào tốt hơn (local optimum), dừng lại.
   * Nếu tìm được trạng thái có số xung đột bằng 0 → tìm thấy nghiệm tối ưu toàn cục (global optimum).
4. Trả kết quả:
   * opt = True nếu tìm thấy lời giải hợp lệ (không xung đột).
   * Trả về [opt, bc], trong đó bc là bàn cờ cuối cùng.

### 2. Stochastic Hill Climbing 1

Khác với Steepest-Ascent Hill Climbing (luôn chọn láng giềng tốt nhất),Stochastic Hill Climbing chọn ngẫu nhiên một láng giềng có điểm tốt hơn hoặc bằng hiện tại để di chuyển đến.Điều này giúp tránh bị “kẹt” sớm ở cực tiểu cục bộ (local optimum).

def stochastic(board, verb = True):

    bc = np.copy(board)

    sol = True

    opt = False

    while sol:

        nbr\_dict = {}

        nbr = np.copy(bc)

        for i in range(len(bc)):

            for j in range(len(bc)):

                nbr[i] = j

                if conflicts(nbr) <= conflicts(bc):

                    nbr\_dict[conflicts(nbr)] = nbr

                nbr = np.copy(bc)

        if conflicts(bc) == min(nbr\_dict.keys()):

            if verb:

                print('local optimum found, global optimum not found')

            sol = False

        # choose next board randomly from dict of neighbors

        bc = nbr\_dict[random.choice(list(nbr\_dict.keys()))]

        if conflicts(bc) == 0:

            if verb:

                print('global/local optimum found')

            opt = True

            sol = False

    return [opt,bc]

Cách hoạt động của đoạn code:

1. bc = np.copy(board)  
   → Sao chép bàn cờ ban đầu.
2. Tạo danh sách láng giềng (nbr\_dict):
   * Với mỗi quân hậu (theo cột i) và mỗi hàng j, di chuyển quân hậu đến hàng j.
   * Nếu số xung đột của trạng thái mới ≤ trạng thái hiện tại, thêm vào nbr\_dict.
3. Kiểm tra điểm cực tiểu cục bộ:
   * Nếu không có láng giềng tốt hơn → dừng lại (local optimum).
4. Chọn bước di chuyển tiếp theo một cách ngẫu nhiên:
   * bc = nbr\_dict[random.choice(list(nbr\_dict.keys()))]  
     → Chọn ngẫu nhiên một trạng thái láng giềng (thay vì luôn chọn tốt nhất).
5. Kiểm tra nghiệm tối ưu:
   * Nếu số xung đột = 0 → đã tìm thấy lời giải hợp lệ (global optimum).

### 3. Sochatis Hill Climbing 2

First-Choice Hill Climbing hoạt động giống như Stochastic Hill Climbing, nhưng thay vì xét toàn bộ các láng giềng hoặc chọn ngẫu nhiên từ danh sách các láng giềng tốt hơn, nó chỉ cần tìm được láng giềng đầu tiên tốt hơn và di chuyển ngay đến đó.  
Điều này giúp tiết kiệm thời gian khi không cần duyệt hết không gian trạng thái.

def first\_choice(board, verb = True, tries = 50):

    bc = np.copy(board)

    sol = True

    opt = False

    n = 0

    while sol:

        nbr = np.copy(bc)

        length = [i for i in range(len(bc))]

        # randomly shuffle rows and columns

        random.shuffle(length)

        for i in length:

            find = False

            random.shuffle(length)

            for j in length:

                nbr[i] = j

                # if first choice has less conflicts than current board, board copy is first choice

                if conflicts(nbr) < conflicts(bc):

                    n = 0

                    bc = np.copy(nbr)

                    find = True

                    break

                # if conflicts of neighbor are greater than or equal to current board add one to n, if n is more than tries, found local optimum

                elif conflicts(nbr) >= conflicts(bc):

                    n += 1

                    if n >= tries:

                        find = True

                        break

                nbr = np.copy(bc)

            if find:

                break

        if n >= tries:

            if verb:

                print('local optimum found, global optimum not found')

            sol = False

        if conflicts(bc) == 0:

            if verb:

                print('global/local optimum found')

            opt = True

            sol = False

    return [opt,bc]

Cách hoạt động hàm first\_choice()

1. Khởi tạo:
   * bc là bản sao của bàn cờ ban đầu.
   * n đếm số lần thử không thành công (không tìm được láng giềng tốt hơn).
2. Lặp lại cho đến khi tìm được lời giải hoặc đạt giới hạn thử (tries):
   * Trộn ngẫu nhiên thứ tự các cột và hàng (random.shuffle(length)).
   * Với từng quân hậu (theo cột i) và hàng j:
     + Nếu láng giềng có ít xung đột hơn (conflicts(nbr) < conflicts(bc)) → cập nhật ngay (bc = nbr) và reset bộ đếm n = 0.
     + Nếu không tốt hơn → tăng n += 1.
     + Nếu n vượt quá tries → dừng (đã đạt cực tiểu cục bộ).
3. Kết thúc:
   * Nếu conflicts(bc) == 0: tìm được nghiệm tối ưu toàn cục.
   * Nếu n >= tries: thuật toán dừng lại ở nghiệm cục bộ.

### Hill Climbing Search with Random Restarts

Trong Hill Climbing thông thường, nếu thuật toán rơi vào cực tiểu cục bộ, nó sẽ dừng lại mà không thể tìm được nghiệm tối ưu toàn cục.Để khắc phục điều này, Random-Restart Hill Climbing thực hiện nhiều lần khởi động ngẫu nhiên (random restart): *Mỗi lần bắt đầu với một trạng thái ban đầu ngẫu nhiên khác nhau, rồi chạy thuật toán Hill Climbing (như Steepest, Stochastic, hoặc First-Choice) cho đến khi tìm được nghiệm tối ưu hoặc hết số lần thử.*

def random\_restart(board, alg = steepest, verb = False, tries = 5, p = False):

    bc = np.copy(board)

    board\_len = len(board)

    sol = False

    if alg != first\_choice:

        # first argument returned from funnction is boolean for T/F

        opt = alg(bc,verb)[0]

        current\_bc = alg(bc,verb)[1]

        # if found optimum solution, return solution

        if opt == True:

            if p:

                print('global/local optimum found')

            return [opt,current\_bc]

        # else, create new random state and call function again

        else:

            # try 100 times to find an optimum solution

            for i in range(100):

                bc = np.copy(random\_board(board\_len))

                opt = alg(bc,verb)[0]

                current\_bc = alg(bc,verb)[1]

                if opt == False:

                    continue

                elif opt == True:

                    sol = True

                    if p:

                        print('global/local optimum found')

                    return [opt,current\_bc]

    # parameters for first\_choice is different, so have different call to function

    else:

        opt = alg(bc,verb, tries)[0]

        current\_bc = alg(bc,verb, tries)[1]

        if opt == True:

            if p:

                print('global/local optimum found')

            return [opt,current\_bc]

        else:

            for i in range(100):

                bc = np.copy(random\_board(board\_len))

                opt = alg(bc,verb,tries)[0]

                current\_bc = alg(bc,verb,tries)[1]

                if opt == False:

                    continue

                elif opt == True:

                    sol = True

                    if p:

                        print('global/local optimum found')

                    return [opt,current\_bc]

    # if optimum solution not found, sol = false and return local optimum solution

    if sol == False:

        if p:

            print('local optimum found, global optimum not found')

        return [opt,bc]

Cách hoạt động trong code

1. Khởi tạo biến
   * board: trạng thái bàn cờ ban đầu.
   * alg: thuật toán Hill Climbing được chọn (Steepest, Stochastic hoặc First-Choice).
   * tries: số lần thử tối đa cho First-Choice.
   * p: bật/tắt in kết quả.
2. Chạy thuật toán Hill Climbing ban đầu (alg)
   * Nếu tìm được nghiệm tối ưu (opt == True) → trả về ngay.
   * Nếu không → bắt đầu thực hiện các lần khởi động lại ngẫu nhiên.
3. Random restart (vòng lặp 100 lần)
   * Tạo một bàn cờ ngẫu nhiên mới bằng random\_board(board\_len).
   * Chạy lại thuật toán Hill Climbing (alg).
   * Nếu lần nào tìm được nghiệm (opt == True) → dừng và trả về kết quả.
4. Nếu hết vòng lặp vẫn không tìm được nghiệm toàn cục  
   → Thuật toán trả về nghiệm cục bộ (best local optimum found).
5. **Simulated Annealing**

Thuật toán bắt đầu với một trạng thái ngẫu nhiên và “nhiệt độ T” ban đầu cao. Trong mỗi bước, nó chọn ngẫu nhiên một trạng thái láng giềng và:

* Nếu trạng thái tốt hơn → chấp nhận.
* Nếu xấu hơn → vẫn có thể chấp nhận với xác suất phụ thuộc vào nhiệt độ.  
  Khi nhiệt độ giảm dần, thuật toán dần “đông lại”, chỉ giữ các trạng thái tốt hơn.  
  Ba kiểu làm nguội được so sánh gồm: *classic*, *fast*, và *exponential*.  
  Kết quả cho thấy việc làm nguội exponential với t0=20, a=0.99, ep=0.01 cho độ chính xác và hiệu suất cao nhất.

**Ảnh có chứa biểu đồ, hàng

Nội dung do AI tạo ra có thể không chính xác.**

**Hình ảnh mô họa cách hoạt động của thuật toán Hill climbing**

* + 1. Bắt đầu với một số giá trị x bất kỳ
    2. Thay đổi x bằng cách -1 or +1 (chọn giá trị nhỏ hơn). Trong trường hợp này x-1 và x + 1 đều là láng giềng với trạng thái hiện tại
    3. Lặp lại x – 1 và x + 1 mà lớn hơn

Vấn đề ở đây là thuật toán này sẽ thường xuyên bị kẹt tại chỗ Local minimum, thay vì Global minimum.Vì vậy TT Simulated Annealing sẽ giúp việc đó là sẽ chọn được và chấp nhận Global minimum thay vì Local minimum

**Giải thích code**

def classic(t0, t):

    return t0\*(1/math.log(1+t))

def fast(t0,t):

    return t0\*(1/(1+t))

def exponential(t0,t,a):

    if a < 0.8 or a > 1:

        print('Improper a parameter. nằm trong khoảng 0.8 và 1 ')

        return

    return t0\*(a\*\*(t))

| **Hàm** | **Mô tả** | **Tốc độ giảm** |
| --- | --- | --- |
| classic | Giảm theo logarit | Chậm |
| fast | Giảm nhanh tuyến tính | Nhanh nhất |
| exponential | Giảm theo hàm mũ với hệ số ( a ) (0.8–1) | Linh hoạt nhất |

def simulated\_annealing(board, t0 = 20 , a = 0.99, ep = 0.01, schedule = exponential, verb = False):

**Mục đích:** Tìm cách đặt N quân hậu sao cho không quân nào ăn nhau,  
bằng cách di chuyển ngẫu nhiên và giảm dần “nhiệt độ” qua thời gian.

**Bước hoạt động:**

1. Khởi tạo:
   * board: trạng thái bàn cờ ban đầu.
   * t0: nhiệt độ ban đầu (cao → chấp nhận thay đổi “xấu” nhiều).
   * a: hệ số giảm nhiệt (dùng với exponential schedule).
   * ep: ngưỡng dừng khi nhiệt độ quá thấp.
   * schedule: hàm điều khiển tốc độ giảm nhiệt.
2. Lặp cho đến khi nhiệt độ :
   * Tính T theo schedule được chọn.
   * Sinh tất cả láng giềng của bàn cờ hiện tại.
   * Chọn ngẫu nhiên 1 láng giềng (next).
   * Tính
     + Nếu : nhận ngay (tốt hơn).
     + Nếu : vẫn có **xác suất** để chấp nhận (tức là chấp nhận xấu hơn).
   * Khi nhiệt độ đủ thấp (T < ep), dừng lại.
3. Trả về:
   * [True, board] nếu tìm được lời giải hoàn hảo.
   * [False, board] nếu dừng ở cực tiểu cục bộ

# Tham khảo

[**Implementation of Backtracking and Steepest Ascent Hill Climbing Algorithms on Ferry Scheduling in Lake Toba | IEEE Conference Publication | IEEE Xplore**](https://ieeexplore.ieee.org/document/10034894/)

[**Optimization Algorithms - 5. Stochastic Hill Climbing**](https://thomasweise.github.io/aitoa-slides/05_stochastic_hill_climbing.pdf)

[**[Tutorial] Simulated Annealing in Competitive Programming - Codeforces**](https://codeforces.com/blog/entry/94437)