

TRƯỜNG ĐẠI HỌC SÀI GÒN
KHOA CÔNG NGHỆ THÔNG TIN



Môn: Trí tuệ nhân tạo nâng cao

Bài tập tuần Lab03

Giảng viên hướng dẫn: Đỗ Như Tài

Sinh viên thực hiện

Nguyễn Thành Long

MSSV: 3121410297

Huỳnh Hoàng Huy

MSSV: 3121410226

Nguyễn Trần Uyên Nhi

MSSV: 3122410281

Phan Thanh Thịnh

MSSV: 3123410360

Tháng 10 - 2025

Mục lục

Phần A: Giải quyết vấn đề N-Queens bằng cách sử dụng Tìm kiếm cục bộ	5
I Giới thiệu bài toán	6
1 Mục tiêu	6
2 Không gian trạng thái	6
3 Hàm mục tiêu	6
4 Phép di chuyển cục bộ	6
5 Điều kiện dừng	6
II Thuật toán Steepest-Ascent Hill Climbing	9
1 Mục tiêu	9
2 Ý tưởng	9
3 Nhận xét	9
III Thuật toán Stochastic Hill Climbing	12
1 Ý tưởng	12
2 Nguyên lý hoạt động	12
3 Nhận xét	12
IV Thuật toán First-Choice Hill Climbing	15
1 Nguyên lý	15
2 Bước thực hiện	15
3 Nhận xét	15
V Thuật toán Hill Climbing với Random Restarts	18
1 Ý tưởng	18
2 Các bước	18
VI Thuật toán Simulated Annealing	20
1 Nguyên lý	20
2 Cách hoạt động	20
3 Nhận xét	20
VII Phân tích và So sánh hiệu năng	23
1 Mục tiêu	23
2 Phương pháp thực nghiệm	23
3 Nhận xét	23
Phần B: Giải quyết bài toán người đi du lịch bằng cách sử dụng Tìm kiếm cục bộ	25
I Giới thiệu bài toán	26
1 Mục tiêu	26
2 Không gian trạng thái	26
3 Hàm mục tiêu	26
4 Phép biến đổi cục bộ	26

5	Các hàm hỗ trợ	26
5.1	Hàm random_tour(n)	26
5.2	Hàm random_tsp(n)	27
5.3	Hàm tour_length(tsp, tour)	27
5.4	Hàm show_tsp(tsp, tour=None)	27
II	Sử dụng R để tìm lời giải cho bài toán TSP	28
1	Giải bài toán TSP trong R	28
2	Đo thời gian thực thi thuật toán	29
III	Thuật toán Steepest-Ascent Hill Climbing	30
1	Mục tiêu	30
2	Ý tưởng	30
3	Cài đặt thuật toán	30
4	Kết quả chạy thuật toán	32
5	Nhận xét	32
IV	Thuật toán Stochastic Hill Climbing	33
1	Cài đặt thuật toán	33
2	Kết quả chạy thuật toán	33
3	Nhận xét	34
V	Thuật toán First-choice Hill Climbing	35
1	Nguyên lý hoạt động	35
2	Ưu điểm	35
3	Nhược điểm	35
4	Cài đặt thuật toán	35
5	Kết quả chạy thuật toán	36
6	Nhận xét	36
VI	Thuật toán Simulated Annealing	37
1	Nguyên lí hoạt động	37
2	Ưu điểm	37
3	Nhược điểm	37
4	Cài đặt thuật toán	38
5	Kết quả chạy thuật toán	38
6	Nhận xét	38
VII	So sánh hiệu năng của các thuật toán	40
1	Mục tiêu	40
2	Dựa vào 3 tiêu chí	40
3	Cài đặt	40
4	Kết quả sau khi cài đặt	41
5	Nhận xét	41
VIII	Genetic algorithm	42
1	Mục tiêu	42

2	Nguyên lý hoạt động	42
3	Cài đặt thuật toán	42
4	Kết quả sau khi cài đặt thuật toán	43
5	Nhận xét	43

Thành viên	Nhiệm vụ được giao	Đánh giá của nhóm trưởng
Nguyễn Thành Long	Phát triển <i>NQueen</i> .	Hoàn thành 100%
Phan Thanh Thịnh	Phát triển <i>TSP</i> , viết báo cáo.	Hoàn thành 100%
Nguyễn Trần Uyển Nhi	Thiết kế slide thuyết minh, biên tập báo cáo tổng bằng \LaTeX .	Hoàn thành 100%
Huỳnh Hoàng Huy	Viết báo cáo <i>NQueen</i> .	Hoàn thành 100%

Bảng 1: Bảng phân công công việc của nhóm

Phần A: Giải quyết vấn đề N-Queens bằng cách sử dụng Tìm kiếm cục bộ

I Giới thiệu bài toán

1 Mục tiêu

Mục tiêu của bài toán N-Queens là tìm cách sắp xếp n quân hậu trên bàn cờ kích thước $n \times n$ sao cho không có hai quân hậu nào tấn công nhau — tức là, không cùng hàng, cột, hoặc đường chéo.

2 Không gian trạng thái

Không gian trạng thái (*state space*) của bài toán gồm tất cả các cách sắp xếp các quân hậu. Ta giới hạn mỗi cột chỉ có đúng một quân hậu. Khi đó, một trạng thái có thể biểu diễn bằng vector:

$$q = \{q_1, q_2, \dots, q_n\}$$

với q_i là vị trí hàng của quân hậu trong cột thứ i . Như vậy, mỗi trạng thái tương ứng với một “bàn cờ” hợp lệ có n quân hậu.

3 Hàm mục tiêu

Hàm mục tiêu (*objective function*) là số cặp quân hậu đang tấn công lẫn nhau. Ta cần tìm cấu hình q^* sao cho:

$$\min_q \text{conflicts}(q)$$

với điều kiện mỗi cột chỉ chứa một quân hậu. Giá trị tối ưu đạt được khi $\text{conflicts}(q^*) = 0$.

4 Phép di chuyển cục bộ

Một phép di chuyển cục bộ (*local move*) là việc di chuyển một quân hậu sang một hàng khác trong cùng cột. Mỗi phép di chuyển tạo ra một trạng thái lân cận mới, có thể cải thiện hoặc làm xấu đi giá trị hàm mục tiêu.

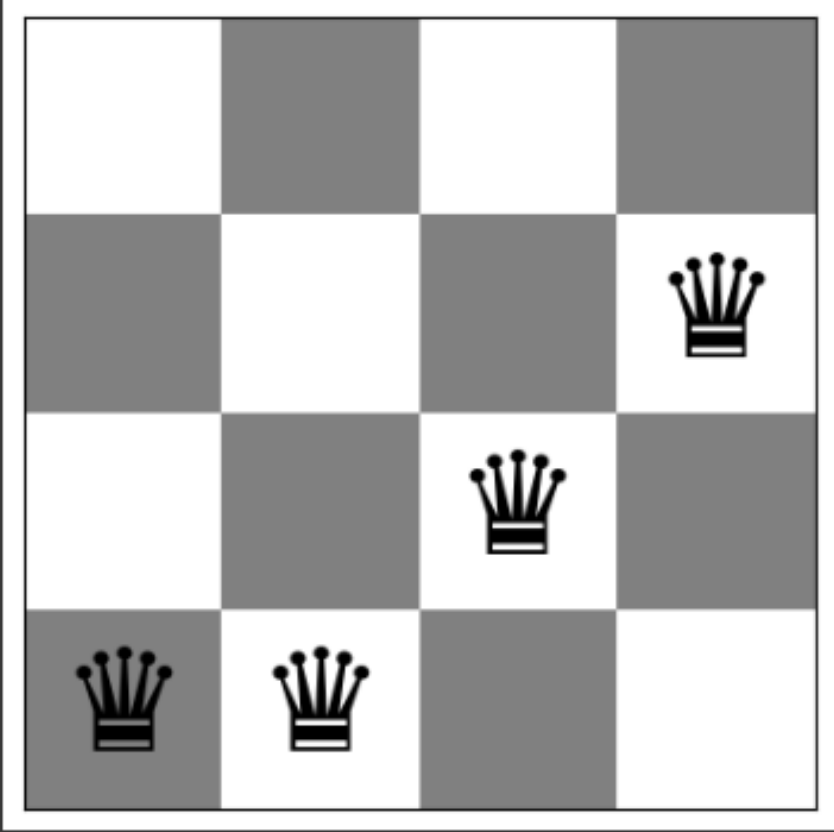
5 Điều kiện dừng

Thuật toán dừng khi đạt được lời giải không có xung đột ($\text{conflicts} = 0$), hoặc không thể tìm được nước đi nào giúp cải thiện tình hình (đạt cực tiểu cục bộ).


```
board = random_board(4)

show_board(board)
print(f"Queens (left to right) are at rows: {board}")
print(f"Number of conflicts: {conflicts(board)}")
```

Board with 4 conflicts.



Row \ Col	0	1	2	3
0	Queen			
1		Queen		
2			Queen	
3				Queen

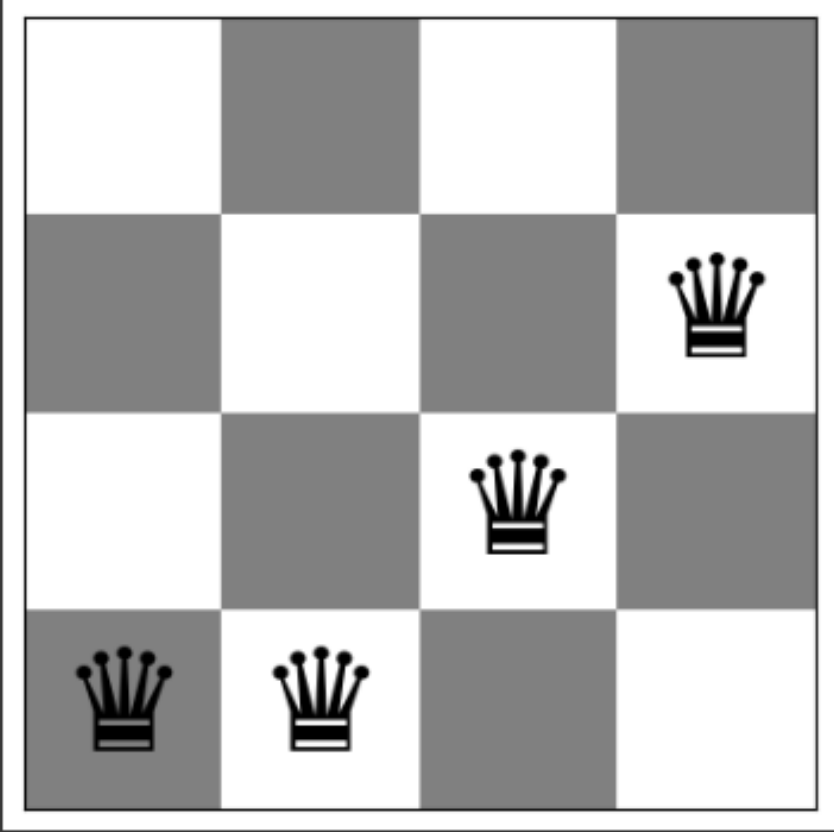
```
Queens (left to right) are at rows: [3 3 2 1]
Number of conflicts: 4
```

Hình 1: Minh họa bài toán N-Queens với $n = 4$

```
board = random_board(4)

show_board(board)
print(f"Queens (left to right) are at rows: {board}")
print(f"Number of conflicts: {conflicts(board)}")
```

Board with 4 conflicts.



			♔
		♔	
♔	♔		

```
Queens (left to right) are at rows: [3 3 2 1]
Number of conflicts: 4
```

Hình 2: Minh họa bài toán N-Queens với $n = 4$

II Thuật toán Steepest-Ascent Hill Climbing

1 Mục tiêu

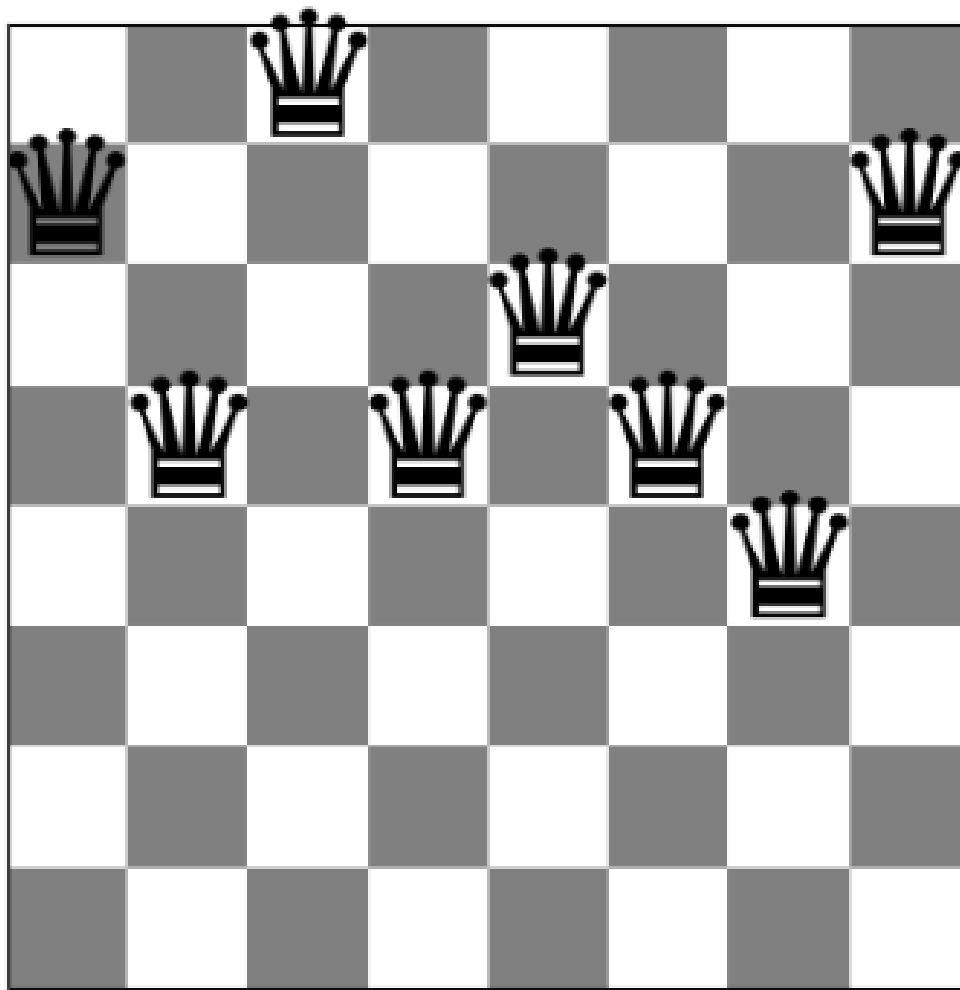
Tìm cấu hình bàn cờ có ít xung đột nhất bằng cách luôn chọn nước đi cục bộ tốt nhất.

2 Ý tưởng

1. Khởi tạo bàn cờ ngẫu nhiên.
2. Xét tất cả các nước đi hợp lệ (di chuyển một quân hậu trong cột của nó).
3. Tính giá trị hàm mục tiêu cho từng nước đi.
4. Chọn nước đi có ít xung đột nhất.
5. Nếu không còn cải thiện nào, dừng lại (đạt cực trị cục bộ).

3 Nhận xét

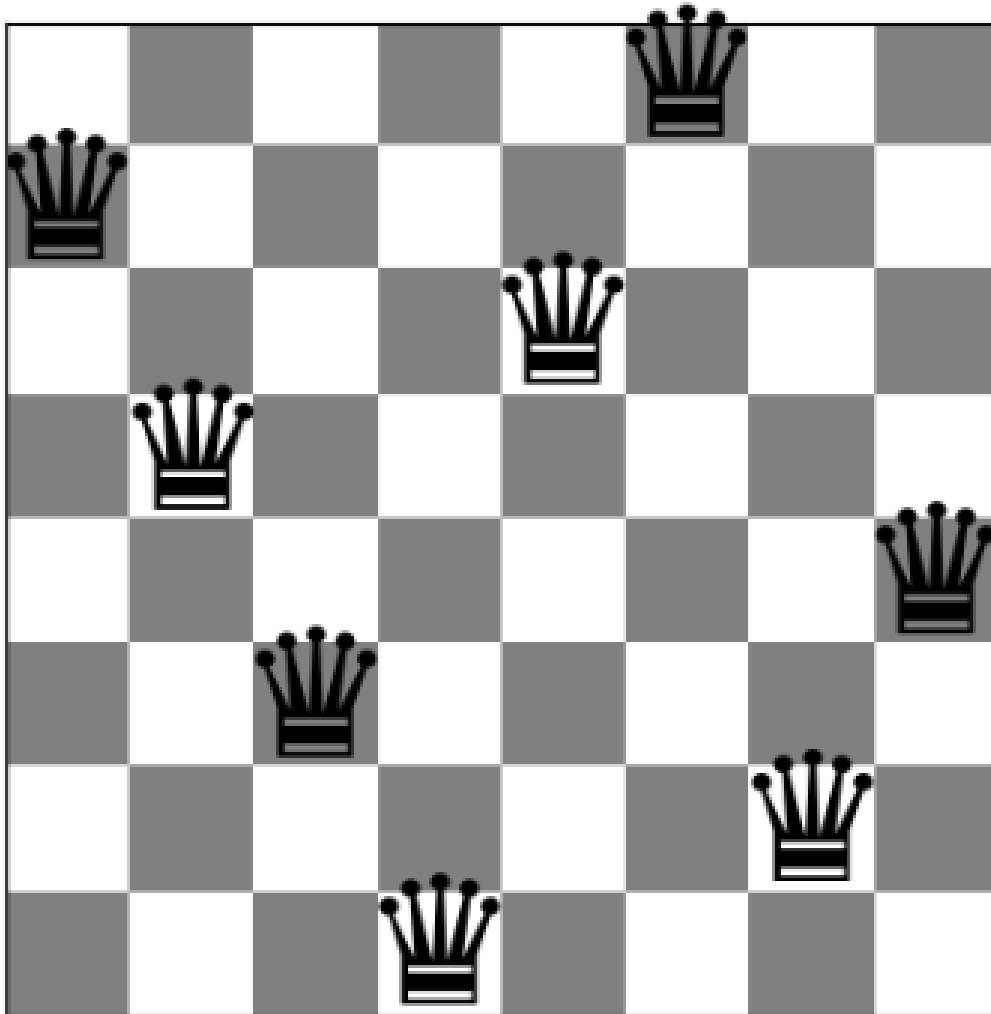
Thuật toán đơn giản, dễ cài đặt, nhưng dễ mắc kẹt ở cực trị cục bộ do luôn chọn hướng tốt nhất tại mỗi bước.



Tìm thấy nước đi tốt hơn, số xung đột giảm từ 12 xuống 7.
Tìm thấy nước đi tốt hơn, số xung đột giảm từ 7 xuống 4.
Tìm thấy nước đi tốt hơn, số xung đột giảm từ 4 xuống 3.
Tìm thấy nước đi tốt hơn, số xung đột giảm từ 3 xuống 2.
Tìm thấy nước đi tốt hơn, số xung đột giảm từ 2 xuống 1.
Tìm thấy nước đi tốt hơn, số xung đột giảm từ 1 xuống 0.
Đã đạt tối ưu cục bộ. Dừng lại.

Hình 3: Trực quan hóa thuật toán Steepest-Ascent Hill Climbing

Bàn cờ cuối cùng:
Board with 0 conflicts.



Hình 4: Trực quan hóa thuật toán Steepest-Ascent Hill Climbing

III Thuật toán Stochastic Hill Climbing

1 Ý tưởng

Thay vì chọn nước đi tốt nhất, thuật toán chọn **ngẫu nhiên một nước đi tốt hơn** so với trạng thái hiện tại. Nhờ đó, nó có thể thoát khỏi các điểm cực trị cục bộ.

2 Nguyên lý hoạt động

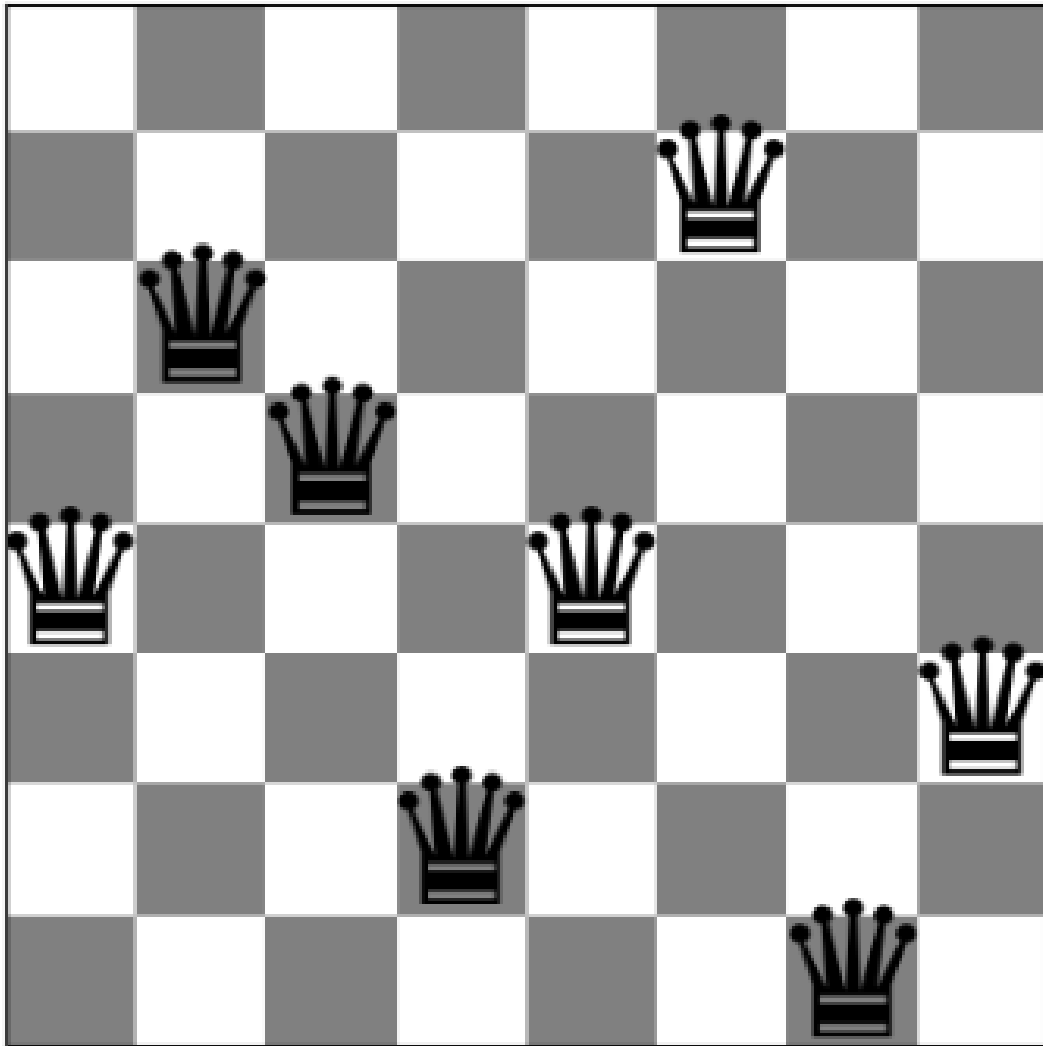
1. Bắt đầu với một trạng thái khởi tạo.
2. Xác định tất cả các nước đi giúp giảm số xung đột.
3. Chọn ngẫu nhiên một trong các nước đi này và di chuyển đến đó.
4. Lặp lại cho đến khi không còn nước đi tốt hơn.

3 Nhận xét

Stochastic Hill Climbing có khả năng thoát cực trị cục bộ tốt hơn Steepest-Ascent, nhưng kết quả phụ thuộc vào chuỗi ngẫu nhiên.

Bàn cờ ban đầu:

Board with 4 conflicts.



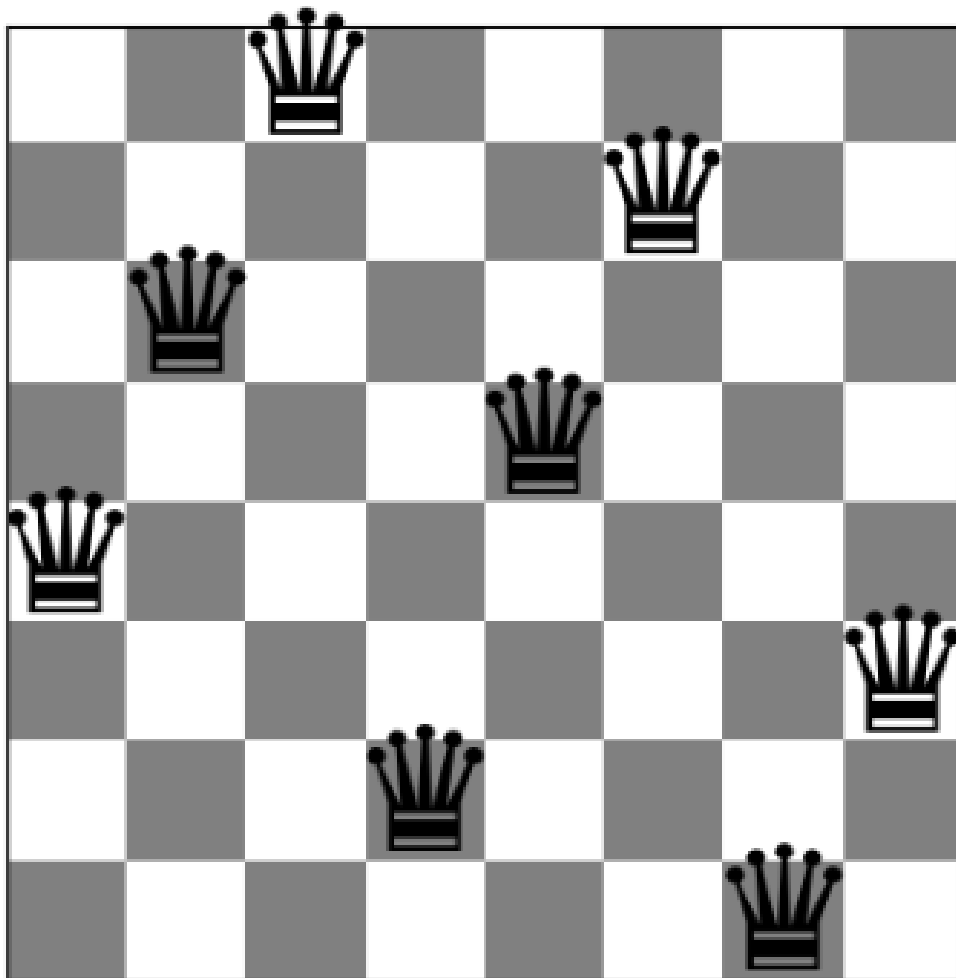
Chọn ngẫu nhiên một nước đi, số xung đột mới: 3

Chọn ngẫu nhiên một nước đi, số xung đột mới: 2

Đã đạt tối ưu cục bộ. Dừng lại.

Hình 5: Minh họa Stochastic Hill Climbing cho bài toán N-Queens

Bàn cờ cuối cùng:
Board with 2 conflicts.



Hình 6: Minh họa Stochastic Hill Climbing cho bài toán N-Queens

IV Thuật toán First-Choice Hill Climbing

1 Nguyên lý

Thay vì xét toàn bộ lân cận, thuật toán chọn **ngẫu nhiên một nước đi** và chấp nhận ngay nếu nó cải thiện lời giải.

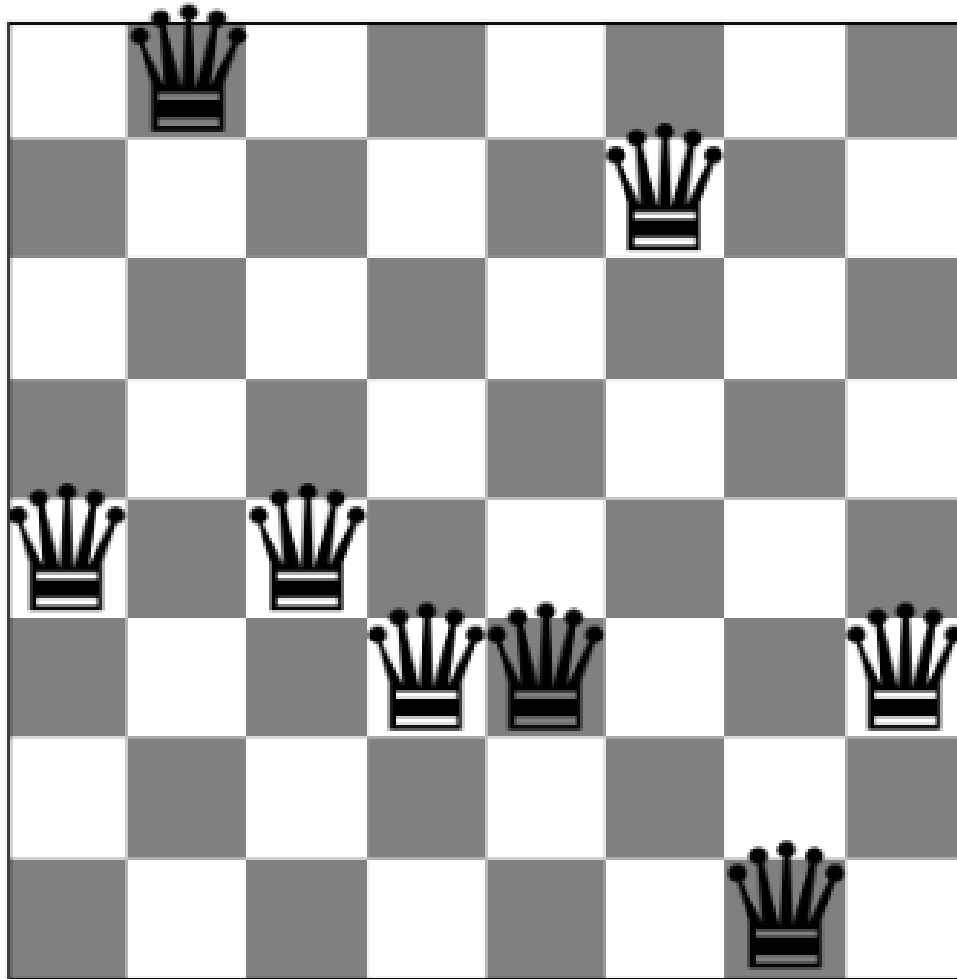
2 Bước thực hiện

1. Bắt đầu với bàn cờ ngẫu nhiên.
2. Chọn một quân hậu và di chuyển đến vị trí ngẫu nhiên trong cột của nó.
3. Nếu số xung đột giảm, chấp nhận và cập nhật trạng thái.
4. Lặp lại cho đến khi đạt cực trị hoặc đủ số lần thử.

3 Nhận xét

Ưu điểm: tốc độ nhanh hơn, giảm chi phí duyệt toàn bộ lân cận. Nhược điểm: có thể bỏ qua nước đi tối ưu nhất, kết quả không ổn định.

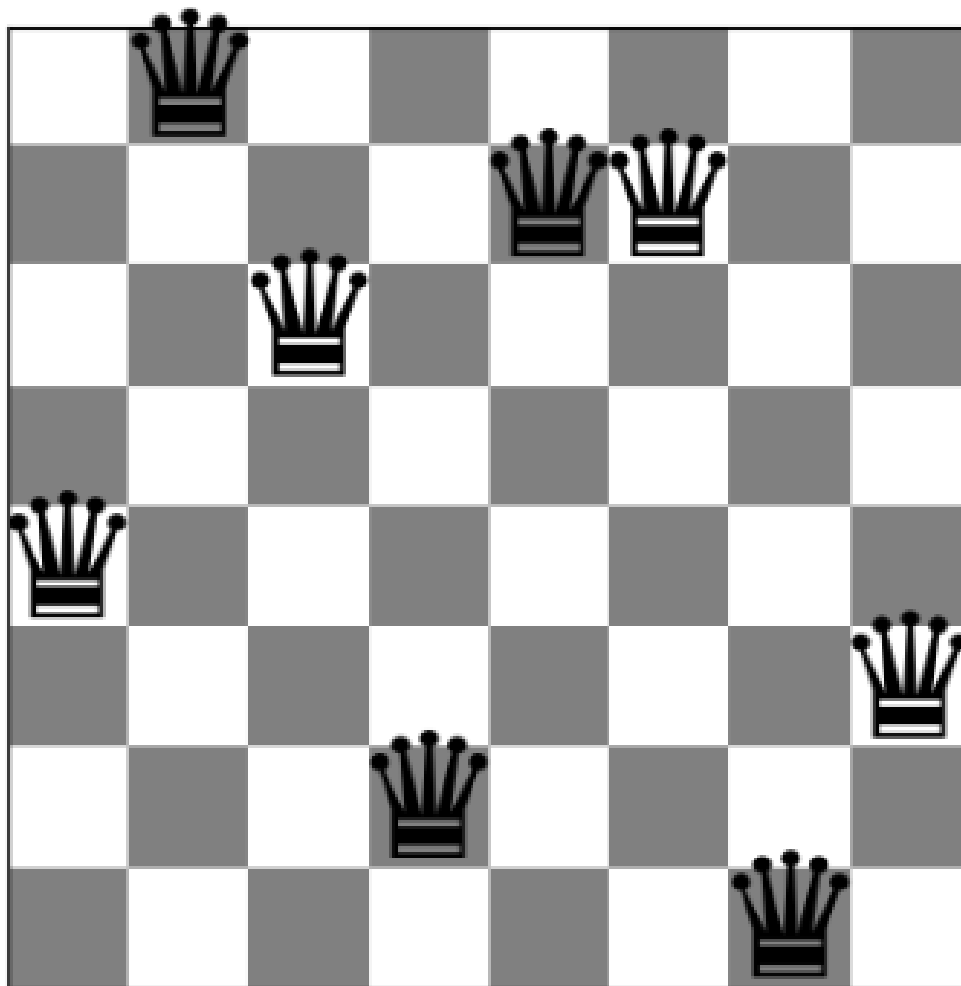
Bàn cờ ban đầu cho First-Choice HC:
Board with 7 conflicts.



Tìm thấy lựa chọn tốt hơn. Xung đột mới: 5
Tìm thấy lựa chọn tốt hơn. Xung đột mới: 3
Tìm thấy lựa chọn tốt hơn. Xung đột mới: 2
Đã đạt tối ưu cục bộ sau 100 lần thử. Dừng lại.

Hình 7: First-Choice Hill Climbing trên bài toán N-Queens

Bàn cờ cuối cùng:
Board with 2 conflicts.



Hình 8: First-Choice Hill Climbing trên bài toán N-Queens

V Thuật toán Hill Climbing với Random Restarts

1 Ý tưởng

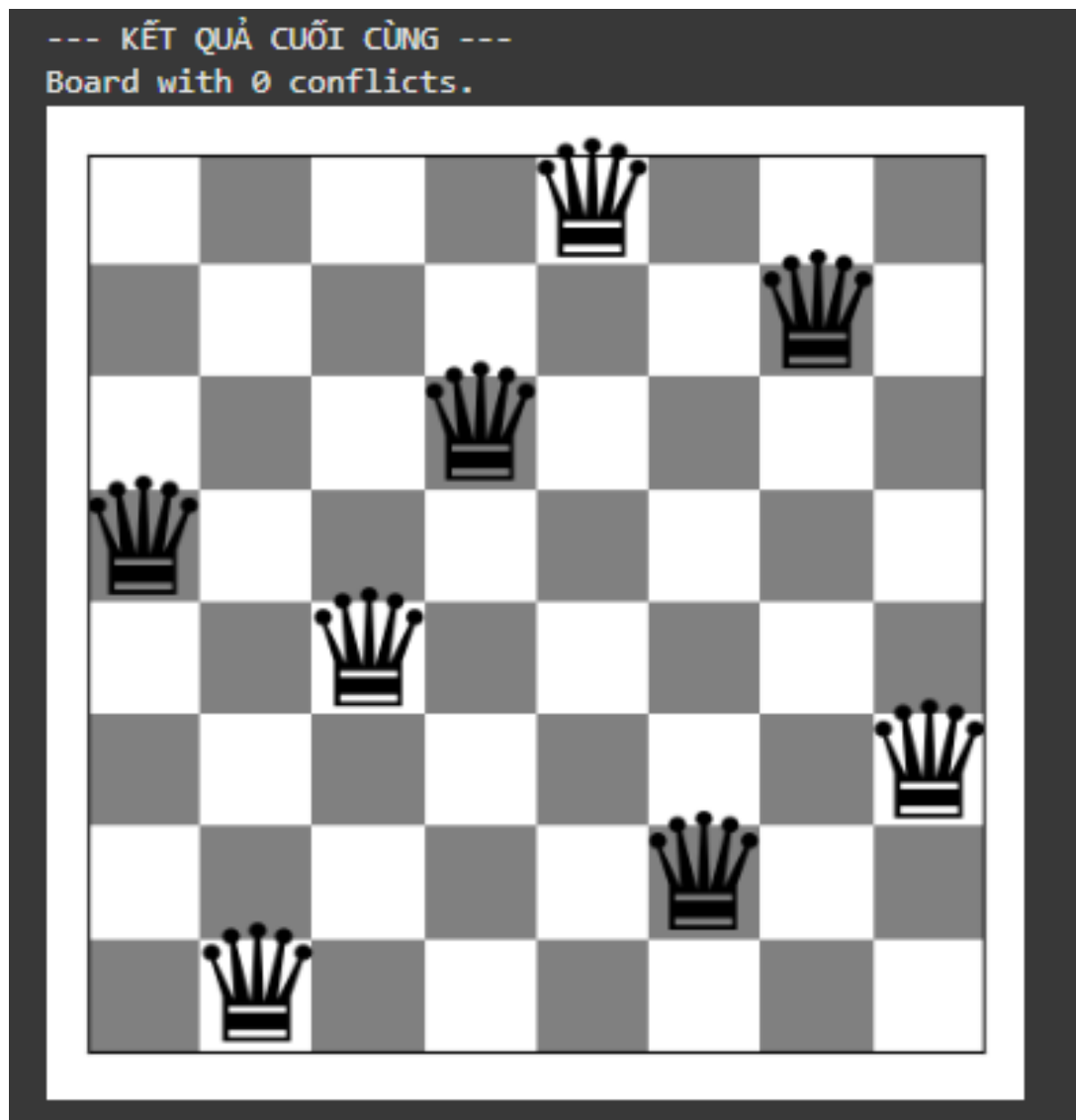
Để tránh mắc kẹt ở cực trị cục bộ, thuật toán thực hiện nhiều lần khởi tạo ngẫu nhiên. Mỗi lần khởi tạo sẽ chạy một thuật toán Hill Climbing cụ thể và lưu lại nghiệm tốt nhất.

2 Các bước

1. Chạy thuật toán Hill Climbing từ trạng thái ngẫu nhiên.
2. Ghi lại kết quả tốt nhất (số xung đột nhỏ nhất).
3. Lặp lại quá trình nhiều lần với các khởi tạo khác nhau.
4. Chọn lời giải tốt nhất trong tất cả các lần chạy.

```
Bắt đầu chạy Hill Climbing với 10 lần khởi động lại...
--- Lần khởi động thứ 1/10 ---
Tìm thấy nước đi tốt hơn, số xung đột giảm từ 10 xuống 6.
Tìm thấy nước đi tốt hơn, số xung đột giảm từ 6 xuống 3.
Tìm thấy nước đi tốt hơn, số xung đột giảm từ 3 xuống 2.
Đã đạt tối ưu cục bộ. Dừng lại.
Kết quả lần này: 2 xung đột.
*** Tìm thấy kết quả tốt hơn mới! Số xung đột: 2 ***
--- Lần khởi động thứ 2/10 ---
Tìm thấy nước đi tốt hơn, số xung đột giảm từ 6 xuống 4.
Tìm thấy nước đi tốt hơn, số xung đột giảm từ 4 xuống 3.
Tìm thấy nước đi tốt hơn, số xung đột giảm từ 3 xuống 2.
Tìm thấy nước đi tốt hơn, số xung đột giảm từ 2 xuống 1.
Đã đạt tối ưu cục bộ. Dừng lại.
Kết quả lần này: 1 xung đột.
*** Tìm thấy kết quả tốt hơn mới! Số xung đột: 1 ***
--- Lần khởi động thứ 3/10 ---
Tìm thấy nước đi tốt hơn, số xung đột giảm từ 6 xuống 4.
Tìm thấy nước đi tốt hơn, số xung đột giảm từ 4 xuống 2.
Tìm thấy nước đi tốt hơn, số xung đột giảm từ 2 xuống 0.
Đã đạt tối ưu cục bộ. Dừng lại.
Kết quả lần này: 0 xung đột.
*** Tìm thấy kết quả tốt hơn mới! Số xung đột: 0 ***
Đã tìm thấy lời giải tối ưu, dừng sớm.
```

Hình 9: Hill Climbing với nhiều lần khởi tạo ngẫu nhiên



Hình 10: Hill Climbing với nhiều lần khởi tạo ngẫu nhiên

VI Thuật toán Simulated Annealing

1 Nguyên lý

Simulated Annealing (SA) mô phỏng quá trình “tôi luyện kim loại”: ban đầu cho phép chấp nhận lời giải tệ hơn với xác suất cao, sau đó giảm dần khi “nhiệt độ” hạ xuống.

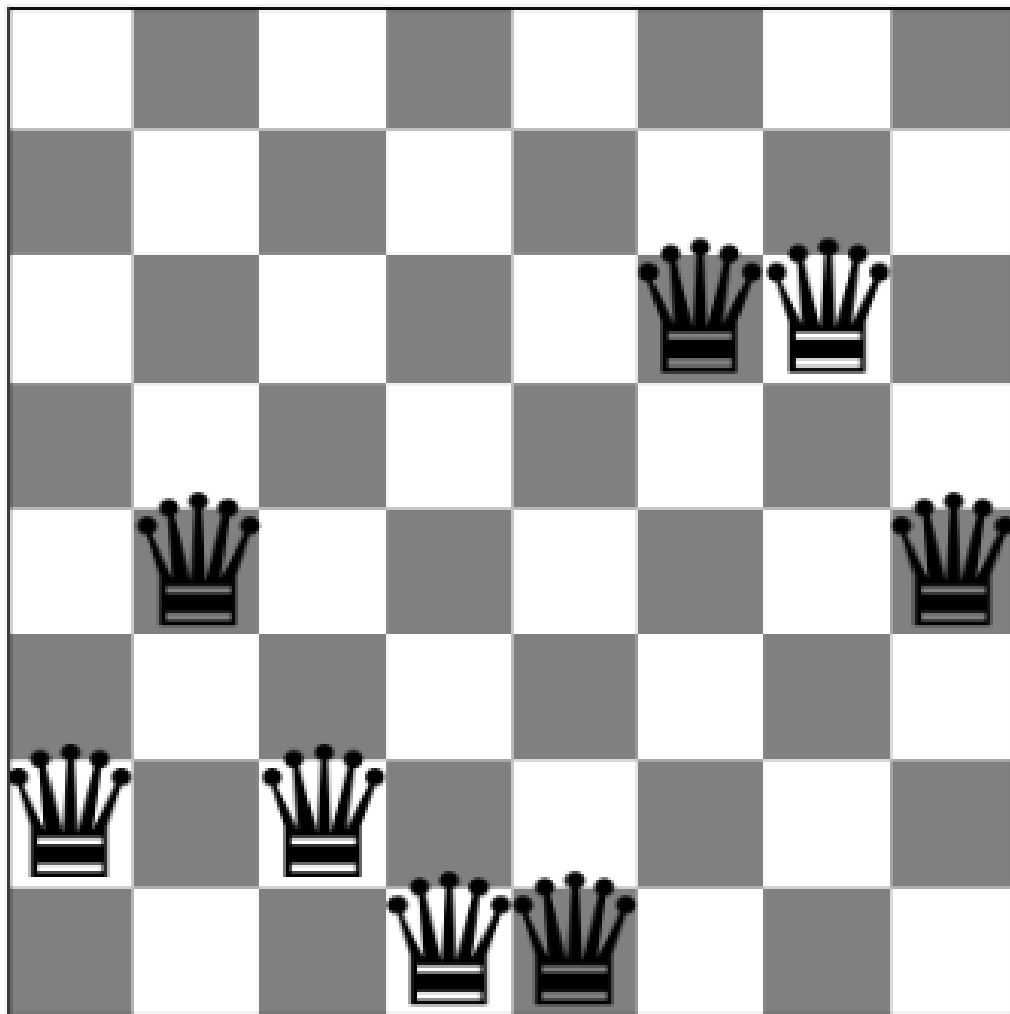
2 Cách hoạt động

1. Khởi tạo bàn cờ ngẫu nhiên và nhiệt độ T_{start} .
2. Sinh lân cận bằng cách di chuyển một quân hậu ngẫu nhiên.
3. Nếu trạng thái mới tốt hơn \rightarrow chấp nhận. Nếu tệ hơn \rightarrow chấp nhận với xác suất $e^{-\Delta/T}$.
4. Giảm nhiệt độ theo hệ số α sau mỗi vòng lặp.
5. Dừng khi $T < T_{end}$ hoặc không còn cải thiện.

3 Nhận xét

SA giúp thoát cực trị cục bộ, ổn định hơn Hill Climbing truyền thống, nhưng phụ thuộc vào tham số nhiệt độ và tốc độ làm nguội.

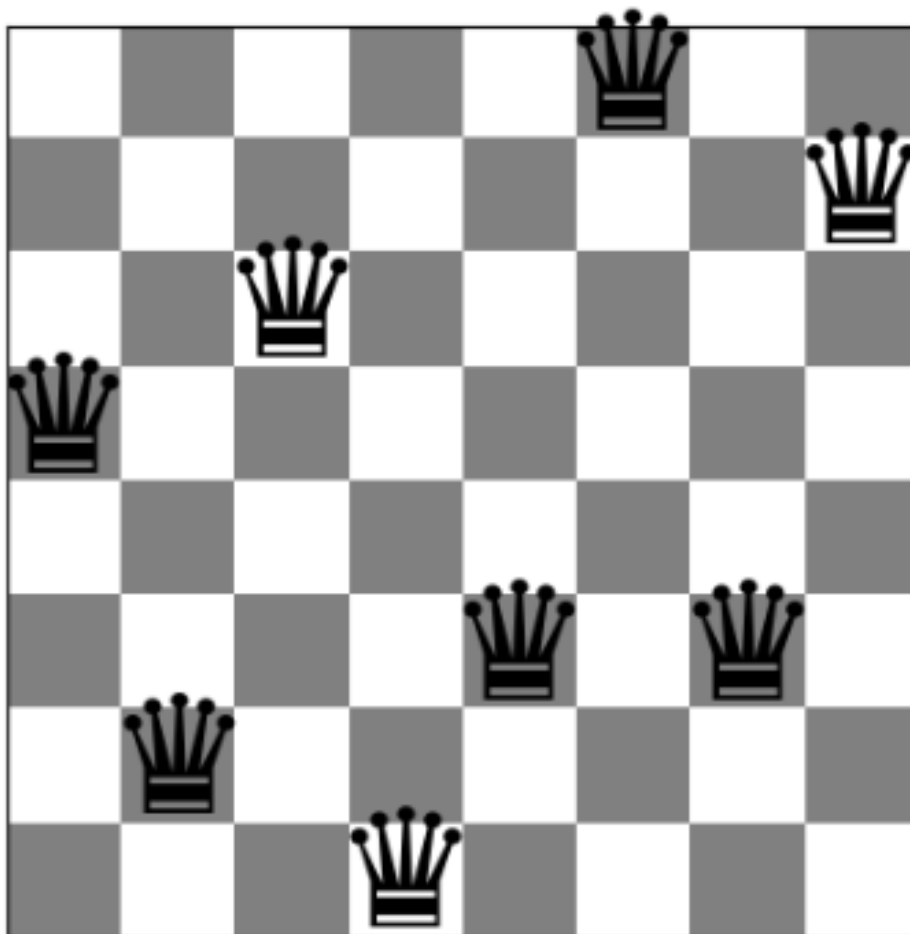
Bàn cờ ban đầu cho Simulated Annealing:
Board with 9 conflicts.



Nhiệt độ đã xuống tới mức tối thiểu. Dừng lại.

Hình 11: Kết quả Simulated Annealing trên bài toán N-Queens

Bàn cờ cuối cùng:
Board with 1 conflicts.



Hình 12: Kết quả Simulated Annealing trên bài toán N-Queens

VII Phân tích và So sánh hiệu năng

1 Mục tiêu

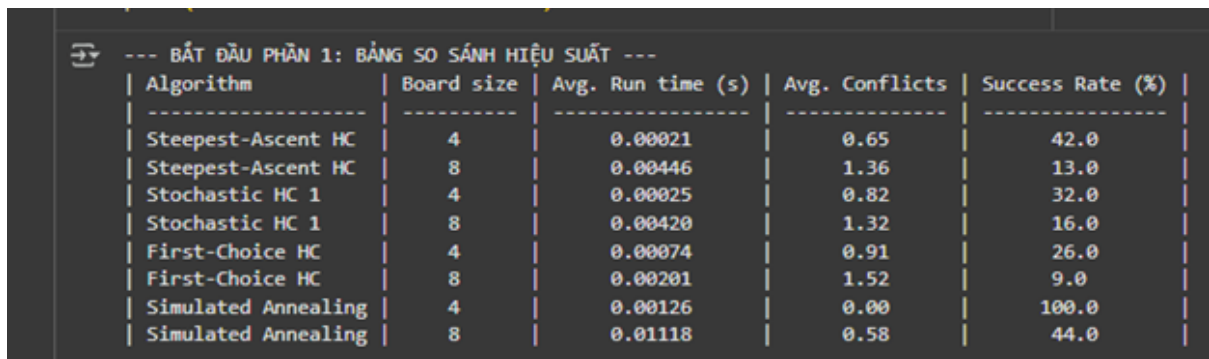
Đánh giá và so sánh bốn thuật toán tìm kiếm cục bộ:

- Steepest-Ascent Hill Climbing
- Stochastic Hill Climbing
- First-Choice Hill Climbing
- Simulated Annealing

2 Phương pháp thực nghiệm

Mỗi thuật toán được chạy nhiều lần (100 lần) trên các kích thước bàn cờ khác nhau ($n = 4, 8, 12, 16, 20$). Kết quả được ghi nhận qua các chỉ số:

- Thời gian trung bình chạy (Average Run Time)
- Số xung đột trung bình còn lại (Average Conflicts)
- Tỷ lệ thành công (Success Rate)



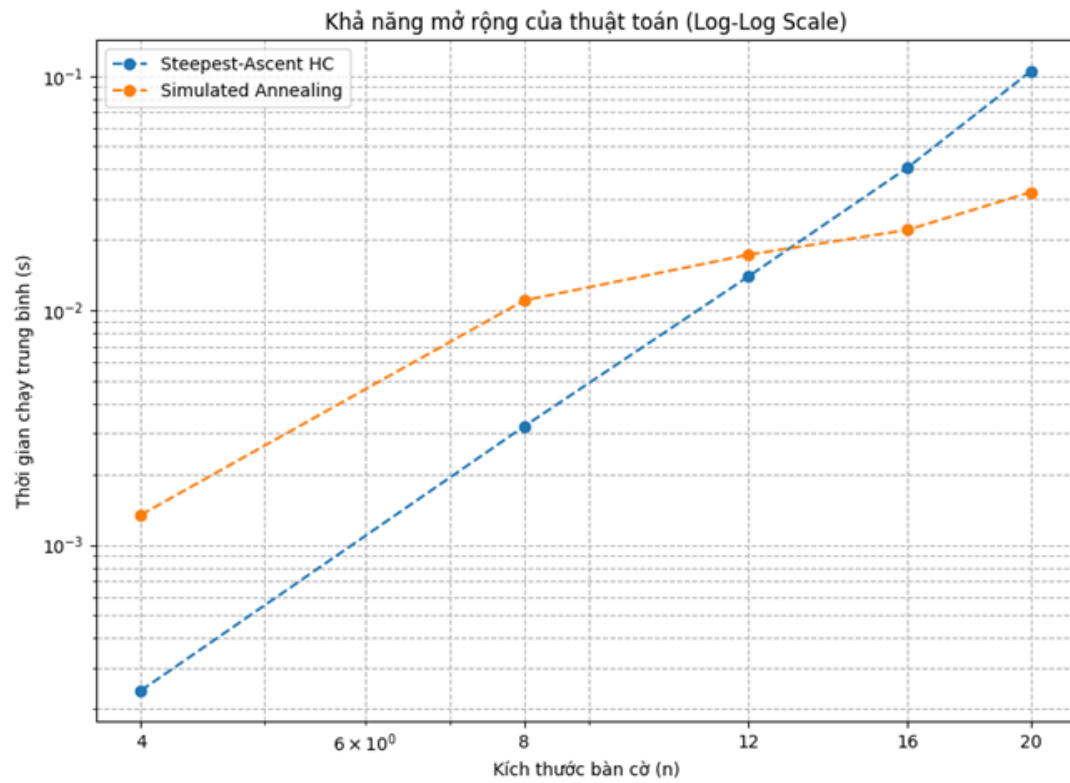
--- BẮT ĐẦU PHẦN 1: BẢNG SO SÁNH HIỆU SUẤT ---

Algorithm	Board size	Avg. Run time (s)	Avg. Conflicts	Success Rate (%)
Steepest-Ascent HC	4	0.00021	0.65	42.0
Steepest-Ascent HC	8	0.00446	1.36	13.0
Stochastic HC 1	4	0.00025	0.82	32.0
Stochastic HC 1	8	0.00420	1.32	16.0
First-Choice HC	4	0.00074	0.91	26.0
First-Choice HC	8	0.00201	1.52	9.0
Simulated Annealing	4	0.00126	0.00	100.0
Simulated Annealing	8	0.01118	0.58	44.0

Hình 13: So sánh hiệu năng các thuật toán N-Queens

3 Nhận xét

- **Steepest-Ascent Hill Climbing:** chính xác nhưng chậm, dễ mắc kẹt ở cực trị cục bộ.
- **Stochastic Hill Climbing:** kết quả dao động, đôi khi vượt qua cực trị tốt.
- **First-Choice Hill Climbing:** nhanh, hiệu quả với dữ liệu nhỏ nhưng kém ổn định.
- **Simulated Annealing:** ổn định và mở rộng tốt, tránh kẹt cực trị nhờ xác suất chấp nhận linh hoạt.



Hình 14: Độ phức tạp thời gian theo kích thước bàn cờ (log-log scale)

**Phần B: Giải quyết bài toán người đi du lịch bằng cách sử dụng
Tìm kiếm cục bộ**

I Giới thiệu bài toán

1 Mục tiêu

Tìm hành trình ngắn nhất đi qua tất cả n thành phố, mỗi thành phố đúng một lần và quay trở lại thành phố xuất phát. Cho trước khoảng cách giữa các cặp thành phố, trong đó $d_{i,j}$ là khoảng cách từ thành phố i đến thành phố j .

2 Không gian trạng thái

Không gian trạng thái (*State space*) của bài toán được xác định bởi tất cả các hành trình có thể có. Mỗi trạng thái biểu diễn một hành trình (tour), trong đó các thành phố được đánh số thứ tự, và một hành trình có thể được biểu diễn bằng một vector hoán vị π cho biết thứ tự ghé thăm các thành phố. Cụ thể, $\pi(1)$ là chỉ số của thành phố đầu tiên được ghé thăm, $\pi(2)$ là thành phố thứ hai, và tiếp tục cho đến $\pi(n)$ là thành phố cuối cùng trong hành trình.

3 Hàm mục tiêu

Hàm mục tiêu của bài toán là tối thiểu hóa tổng độ dài của hành trình, được tính theo công thức:

$$L(\pi) = \sum_{k=1}^{n-1} d_{\pi(k), \pi(k+1)} + d_{\pi(n), \pi(1)}$$

với điều kiện π là một vector hoán vị hợp lệ, tức là mỗi thành phố chỉ xuất hiện đúng một lần trong hành trình.

4 Phép biến đổi cục bộ

Các phép biến đổi cục bộ (*local moves*) được sử dụng trong các thuật toán tìm kiếm là việc hoán đổi vị trí của hai thành phố trong hành trình nhằm tạo ra một trạng thái lân cận mới để cải thiện lời giải hiện tại.

5 Các hàm hỗ trợ

Đầu tiên, ta tiến hành khai báo các thư viện cần thiết như `NumPy`, `Pandas`, `Matplotlib`, `Math`, và `Random` nhằm phục vụ cho việc sinh dữ liệu ngẫu nhiên, xử lý ma trận khoảng cách, và trực quan hóa kết quả.

Hàm `random.seed(1234)` được sử dụng để đảm bảo các kết quả ngẫu nhiên có thể được tái lập khi chạy lại chương trình.

5.1 Hàm `random_tour(n)`

Hàm này có nhiệm vụ tạo ra một hành trình (*tour*) ngẫu nhiên gồm n thành phố. Các thành phố được biểu diễn bằng các số nguyên từ 0 đến $n - 1$, sau đó được xáo trộn ngẫu nhiên bằng `random.shuffle()` để tạo thành một thứ tự ghé thăm ngẫu nhiên.

5.2 Hàm `random_tsp(n)`

Hàm này sinh ra một bài toán TSP ngẫu nhiên gồm n thành phố. Mỗi thành phố được gán một tọa độ ngẫu nhiên (x, y) trong khoảng $[0, 1]$. Sau đó, ta sử dụng hàm `pdist` và `squareform` từ thư viện `SciPy` để tính ma trận khoảng cách Euclidean giữa tất cả các cặp thành phố. Kết quả trả về là một `dictionary` gồm:

- **pos:** vị trí tọa độ của các thành phố.
- **dist:** ma trận khoảng cách giữa các thành phố.

5.3 Hàm `tour_length(tsp, tour)`

Đây là hàm đánh giá hàm mục tiêu (*objective function*) của bài toán. Hàm này nhận vào:

- **tsp:** chứa thông tin về ma trận khoảng cách.
- **tour:** thứ tự ghé thăm các thành phố.

Hàm sẽ tính tổng khoảng cách giữa các thành phố liên tiếp trong `tour`, đồng thời cộng thêm khoảng cách từ thành phố cuối cùng quay về thành phố đầu tiên để tạo thành một vòng khép kín.

5.4 Hàm `show_tsp(tsp, tour=None)`

Hàm này dùng để trực quan hóa bài toán TSP và hành trình tìm được. Nếu tham số `tour` được truyền vào, hàm sẽ:

- In ra tổng độ dài hành trình (*tour length*).
- Sắp xếp lại vị trí các thành phố theo thứ tự trong `tour`.
- Vẽ đường đi qua các thành phố theo thứ tự đó, đồng thời nối thành phố cuối cùng trở về điểm xuất phát.

Kết quả hiển thị là biểu đồ *scatter plot* thể hiện vị trí các thành phố và đường đi tương ứng của hành trình.

II Sử dụng R để tìm lời giải cho bài toán TSP

Để giải bài toán Người du lịch (Traveling Salesman Problem) một cách hiệu quả, ta có thể tận dụng gói TSP trong ngôn ngữ R, được thiết kế chuyên biệt cho các bài toán tối ưu hành trình.

Trước tiên, ta nạp phần mở rộng rpy2.ipython trong môi trường Python, cho phép tương tác trực tiếp giữa python và R.

Sau đó, tiến hành kiểm tra và cài đặt các gói R cần thiết:

- **TSP:** dùng để định nghĩa và giải bài toán người du lịch.
- **microbenchmark:** dùng để đo thời gian thực thi của đoạn mã.

```
%load_ext rpy2.ipython

%R if(!"TSP" %in% rownames(installed.packages())) install.packages("TSP", repos="http://cran.us.r-project.org")
%R if(!"microbenchmark" %in% rownames(installed.packages())) install.packages("microbenchmark", repos="http://cran.us.r-project.org")

d = tsp["dist"]
```

Sau khi chuẩn bị ma trận khoảng cách (distance matrix) giữa các thành phố, ta chuyển dữ liệu sang R để giải bài toán.

1 Giải bài toán TSP trong R

Sử dụng gói TSP, ta tạo một đối tượng tsp đại diện cho bài toán. Tiếp đó, ta dùng hàm solve_TSP() để tìm hành trình ngắn nhất. Mặc định, phương pháp giải được sử dụng là 2-opt, đây là một thuật toán Steepest Ascent Hill Climbing, trong đó hai thành phố trong hành trình được hoán đổi để giảm độ dài tour.

Tham số rep = 100 được sử dụng để thực hiện 100 lần khởi tạo ngẫu nhiên (random restarts), nhằm tránh rơi vào nghiệm cục bộ và cải thiện kết quả tối ưu.

```
%%R -i d -o tour

library("TSP")
# tạo đối tượng tsp trg R
tsp <- TSP(d)
print(tsp)

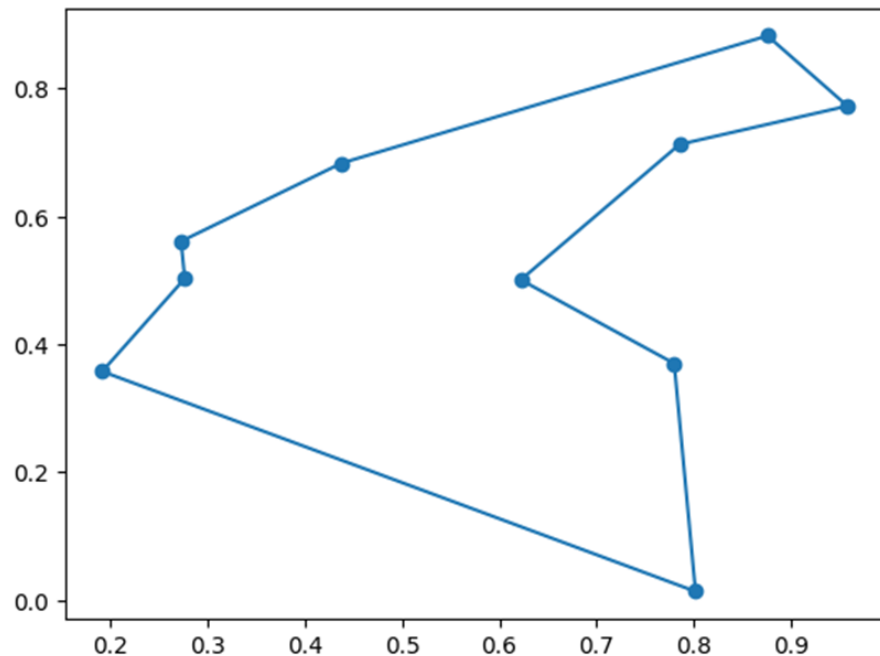
tour <- solve_TSP(tsp, rep = 100) # chạy 100 random restarts để cải thiện tour
print(tour)

# R starts index with 1, but Python starts at 0
tour <- tour - 1L

object of class 'TSP'
10 cities (distance 'unknown')
object of class 'TOUR'
result of method 'arbitrary_insertion+two_opt_rep_100' for 10 cities
tour length: 2.763574
```

Sau khi có kết quả hành trình tối ưu, ta trực quan hóa kết quả bằng hàm `show_tsp(tsp, tour)` trong Python để vẽ đường đi qua các thành phố và hiển thị tổng chiều dài hành trình.

Tour length: 2.76



2 Đo thời gian thực thi thuật toán

Để đánh giá hiệu năng của quá trình giải TSP, ta sử dụng gói `microbenchmark` trong R để đo thời gian thực thi của quá trình xây dựng đối tượng TSP và thực thi thuật toán với 100 lần khởi tạo ngẫu nhiên.

```
%R -i d
# benchmark toàn bộ quá trình giải TSP với 100 random restart
library("microbenchmark")

microbenchmark(tsp <- TSP(d))

Unit: microseconds
      expr      min       lq     mean   median       uq     max neval
tsp <- TSP(d) 343.791 370.415 400.9427 384.3895 402.288 830.021   100
```

III Thuật toán Steepest-Ascent Hill Climbing

Thuật toán Steepest-Ascent Hill Climbing là một phương pháp tìm kiếm cục bộ (local search) thường được sử dụng để giải bài toán TSP (Travelling Salesman Problem).

1 Mục tiêu

Mục tiêu là liên tục cải thiện lời giải hiện tại bằng cách xem xét tất cả các hoán vị cục bộ có thể xảy ra, sau đó chọn lời giải tốt nhất trong số đó cho đến khi không còn cải thiện nào nữa.

2 Ý tưởng

1. Bắt đầu từ một hành trình ngẫu nhiên (initial tour).
2. Sinh ra tất cả các lời giải lân cận (neighbors) bằng cách hoán đổi vị trí hai thành phố trong hành trình hiện tại.
3. Tính hàm mục tiêu (tour length) cho từng lời giải lân cận.
4. Chọn lời giải tốt nhất (có tổng quãng đường ngắn nhất) trong số các neighbors.
5. Nếu lời giải tốt nhất tốt hơn lời giải hiện tại, cập nhật và lặp lại bước 2.
6. Nếu không còn cải thiện nào, thuật toán dừng tại cực trị cục bộ (local optimum).

3 Cài đặt thuật toán

```
def all_swaps(tour):  
    """Sinh tất cả local moves bằng cách hoán đổi 2 thành phố"""  
    n = len(tour)  
    neighbors = []  
    for i in range(n-1):  
        for j in range(i+1, n):# từ kể i trở đi  
            new_tour = tour.copy()  
            new_tour[i], new_tour[j] = new_tour[j], new_tour[i] # đổi tp  
            neighbors.append(new_tour)  
    return neighbors
```



```

def steepest_ascent_hill_climbing(tsp, initial_tour):
    """Tìm tour tối ưu cục bộ bằng steepest-ascent hill climbing"""
    current_tour = initial_tour.copy()
    current_length = tour_length(tsp, current_tour)
    improved = True

    while improved:
        improved = False
        neighbors = all_swaps(current_tour)
        best_neighbor = current_tour
        best_length = current_length

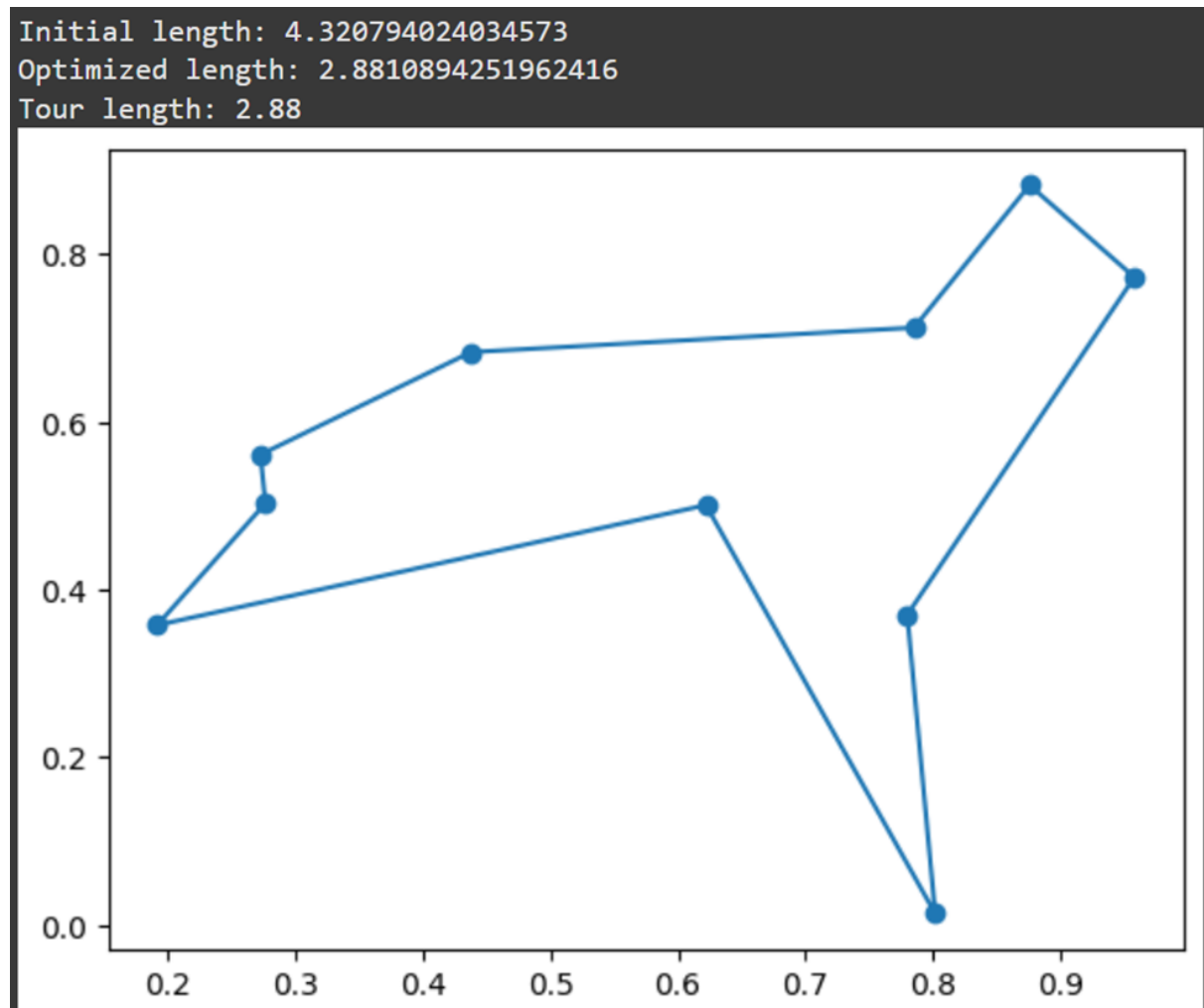
        for neighbor in neighbors:
            length = tour_length(tsp, neighbor)
            if length < best_length:
                best_length = length
                best_neighbor = neighbor
                improved = True

        current_tour = best_neighbor
        current_length = best_length

    return current_tour, current_length

```

4 Kết quả chạy thuật toán



5 Nhận xét

Thuật toán Steepest-Ascent Hill Climbing cho thấy khả năng cải thiện hành trình rõ rệt khi chiều dài đường đi giảm đáng kể so với ban đầu, lộ trình thu được ít giao cắt và hợp lý hơn, ưu điểm của thuật toán là đơn giản, dễ triển khai, hiệu quả với dữ liệu nhỏ và luôn chọn bước đi tốt nhất, tuy nhiên nó dễ mắc kẹt tại cực trị cục bộ, tốn thời gian khi số thành phố lớn và phụ thuộc vào trạng thái khởi tạo ban đầu; nhìn chung đây là phương pháp tìm kiếm cục bộ hiệu quả cho bài toán TSP nhỏ.

IV Thuật toán Stochastic Hill Climbing

Thuật toán này là một biến thể ngẫu nhiên của Hill Climbing, hoạt động theo các bước sau:

1. Bắt đầu với một tour ban đầu (`initial_tour`).
2. Sinh tất cả các hoán đổi 2 thành phố (`neighbor`).
3. Lọc ra các `neighbor` có độ dài ngắn hơn tour hiện tại (tức là tốt hơn).
4. Chọn ngẫu nhiên một trong số các `neighbor` tốt hơn này để di chuyển đến, thay vì chọn cái tốt nhất như trong Steepest-Ascent.
5. Lặp lại tối đa `max_iter` lần hoặc cho đến khi không có `neighbor` nào tốt hơn.

1 Cài đặt thuật toán

```
def stochastic_hill_climbing(tsp, initial_tour, max_iter=1000):
    """Stochastic Hill Climbing cho TSP"""
    current_tour = initial_tour.copy()
    current_length = tour_length(tsp, current_tour)

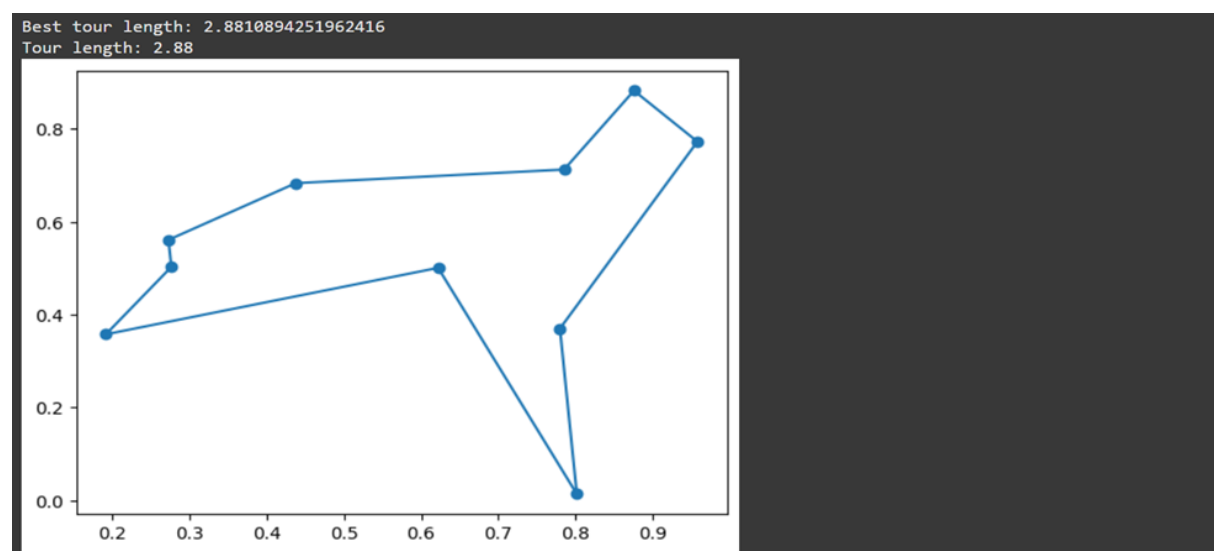
    for iteration in range(max_iter):
        neighbors = all_swaps(current_tour)
        # Chọn tất cả các neighbor tốt hơn hiện tại
        better_neighbors = [tour for tour in neighbors if tour_length(tsp, tour) < current_length]

        if not better_neighbors:
            break

        # Chọn ngẫu nhiên một neighbor tốt hơn
        current_tour = random.choice(better_neighbors)
        current_length = tour_length(tsp, current_tour)

    return current_tour, current_length
```

2 Kết quả chạy thuật toán



3 Nhận xét

Thuật toán Stochastic Hill Climbing đã tìm được lộ trình tối ưu cục bộ với độ dài tour 2.88, cho thấy sự cải thiện so với ban đầu và đường đi hợp lý giữa các thành phố. Khác với Steepest-Ascent luôn chọn hướng cải thiện tốt nhất, Stochastic Hill Climbing chọn ngẫu nhiên trong các hướng cải thiện, giúp tránh mắc kẹt tại cực trị cục bộ nhưng kết quả có thể thay đổi giữa các lần chạy.

V Thuật toán First-choice Hill Climbing

Thuật toán First-Choice Hill Climbing là một biến thể của Hill Climbing trong nhóm local search (tìm kiếm cục bộ). Mục tiêu của thuật toán là cải thiện lời giải hiện tại bằng cách di chuyển sang các trạng thái lân cận có giá trị hàm mục tiêu tốt hơn, tuy nhiên thay vì xét toàn bộ các lân cận, thuật toán này chọn ngẫu nhiên một lân cận tại mỗi bước và chấp nhận ngay nếu nó tốt hơn. Cách tiếp cận này giúp giảm đáng kể thời gian tính toán nhưng vẫn duy trì khả năng cải thiện lời giải.

1 Nguyên lý hoạt động

1. Bắt đầu với một hành trình ban đầu (tour ngẫu nhiên).
2. Tạo một cặp thành phố ngẫu nhiên và hoán đổi vị trí của chúng để tạo ra một láng giềng.
3. Nếu hành trình mới ngắn hơn hành trình hiện tại (tốt hơn), thì cập nhật lời giải.
4. Lặp lại quá trình cho đến khi đạt số lần lặp tối đa hoặc không còn cải thiện.

2 Ưu điểm

Tốc độ nhanh hơn vì không cần duyệt toàn bộ không gian láng giềng. Có khả năng thoát khỏi cực trị cục bộ nếu có yếu tố ngẫu nhiên tốt.

3 Nhược điểm

Dễ bỏ lỡ hướng cải thiện tốt nhất. Kết quả có thể không ổn định, phụ thuộc vào chuỗi ngẫu nhiên sinh ra.

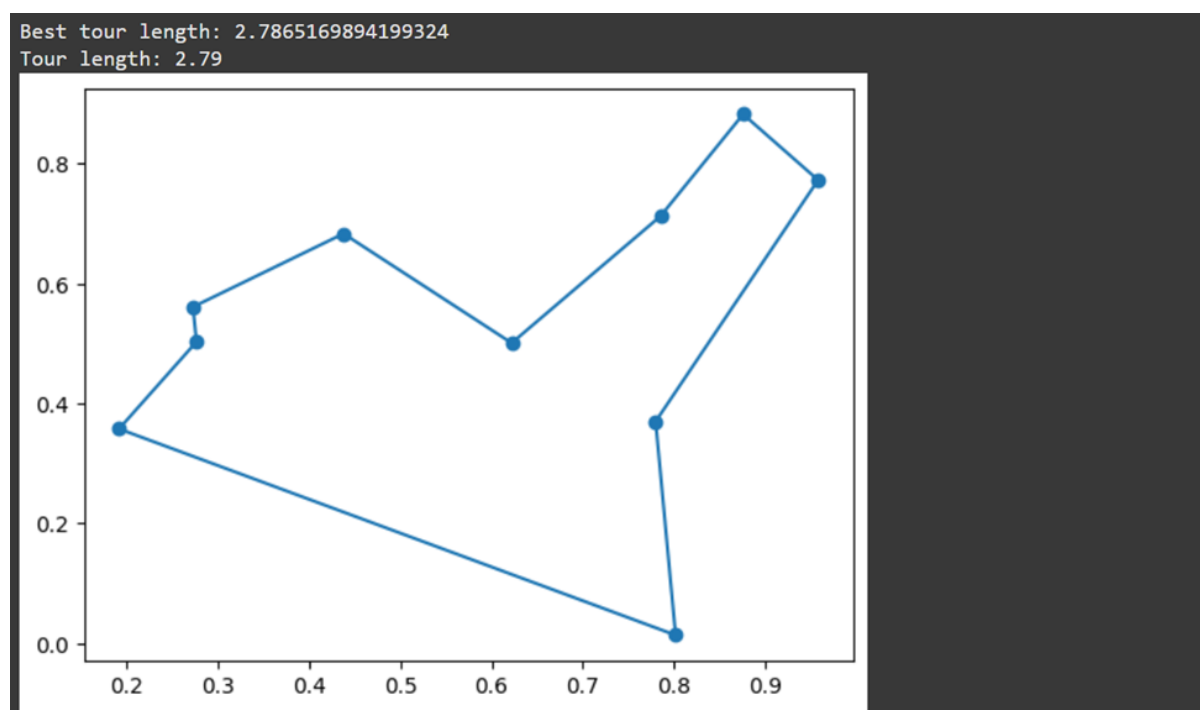
4 Cài đặt thuật toán

```
def first_choice_hill_climbing(tsp, initial_tour, max_iter=1000):
    """First-Choice Hill Climbing cho TSP"""
    current_tour = initial_tour.copy()
    current_length = tour_length(tsp, current_tour)
    n = len(current_tour)

    for i in range(max_iter):
        # lấy ngẫu nhiên 2 tp từ n( s1 tp)
        i, j = random.sample(range(n), 2)
        neighbor = current_tour.copy()
        neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
        length = tour_length(tsp, neighbor)
        # sau khi chạy chỗ mà tối ưu hơn thì thay thế
        if length < current_length:
            current_tour = neighbor
            current_length = length

    return current_tour, current_length
```

5 Kết quả chạy thuật toán



6 Nhận xét

Thuật toán First-Choice Hill Climbing cho kết quả độ dài hành trình tốt hơn hai thuật toán trước (2.79 so với 2.88), cho thấy hiệu quả cải thiện đáng kể dù chỉ kiểm tra ngẫu nhiên một lân cận tại mỗi bước. Do chọn lân cận ngẫu nhiên và cập nhật ngay khi tốt hơn, thuật toán chạy nhanh hơn Steepest-Ascent và Stochastic Hill Climbing, đồng thời có khả năng thoát khỏi cực trị cục bộ tốt hơn Steepest-Ascent. Tuy nhiên, do tính ngẫu nhiên cao nên kết quả có thể khác nhau giữa các lần chạy, không ổn định bằng Steepest-Ascent.

VI Thuật toán Simulated Annealing

Simulated Annealing (SA) là một biến thể nâng cao của Hill Climbing, mô phỏng quá trình “tôi luyện kim loại” trong vật lý. Thuật toán bắt đầu với nhiệt độ cao và dần làm nguội theo thời gian. Ở giai đoạn đầu, thuật toán cho phép chấp nhận những lời giải kém hơn (tức có giá trị mục tiêu lớn hơn) với một xác suất nhất định, giúp nó có khả năng thoát khỏi các cực trị cục bộ mà Hill Climbing thường mắc phải. Khi nhiệt độ giảm dần, thuật toán trở nên “bảo thủ” hơn, chỉ chấp nhận những lời giải tốt hơn dần hội tụ về nghiệm tối ưu cục bộ hoặc gần tối ưu toàn cục.

1 Nguyên lí hoạt động

1. Khởi tạo tour ban đầu và nhiệt độ T_{start} .
2. Ở mỗi vòng lặp, sinh một lân cận ngẫu nhiên bằng cách hoán đổi hai thành phố.
3. Nếu lân cận tốt hơn, chấp nhận ngay.
4. Nếu kém hơn, vẫn có thể chấp nhận với xác suất $e^{-\Delta/T}$.
5. Giảm dần nhiệt độ theo hệ số làm nguội alpha cho đến khi nhỏ hơn T_{end} hoặc đạt số vòng lặp tối đa.
6. Ghi nhận tour có độ dài ngắn nhất trong toàn bộ quá trình.

2 Ưu điểm

Thuật toán Simulated Annealing vượt trội hơn so với các phương pháp Hill Climbing thuần túy, vì có khả năng vượt qua các điểm cực trị cục bộ, giúp tìm ra lời giải tốt hơn gần với tối ưu toàn cục. Ngoài ra, nó đơn giản, dễ triển khai và không cần khảo sát toàn bộ lân cận, nên tốc độ vẫn khả thi với các bài toán kích thước vừa.

3 Nhược điểm

Tốc độ hội tụ phụ thuộc mạnh vào các tham số T_{start} , alpha và số vòng lặp. Nếu giảm nhiệt quá nhanh, dễ dừng sớm ở nghiệm chưa tốt; nếu giảm quá chậm, thời gian chạy tăng đáng kể. Ngoài ra, kết quả vẫn mang tính ngẫu nhiên và có thể thay đổi giữa các lần chạy.

4 Cài đặt thuật toán

```
def simulated_annealing(tsp, initial_tour, T_start=1000, T_end=1e-3, alpha=0.995, max_iter=1000):
    """Simulated Annealing cho bài toán TSP"""
    current_tour = initial_tour.copy()
    current_length = tour_length(tsp, current_tour)
    best_tour = current_tour.copy()
    best_length = current_length
    n = len(current_tour)

    T = T_start

    for iteration in range(max_iter):
        # mỗi lần lặp giảm nhiệt độ, khi nhiệt độ quá thấp + dừng lại
        T *= alpha
        if T < T_end:
            break

        # Sinh 1 neighbor ngẫu nhiên (swap 2 thành phố)
        i, j = random.sample(range(n), 2)
        neighbor = current_tour.copy()
        neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
        neighbor_length = tour_length(tsp, neighbor)

        # delta: độ chênh lệch giữa neighbor và current.
        delta = neighbor_length - current_length

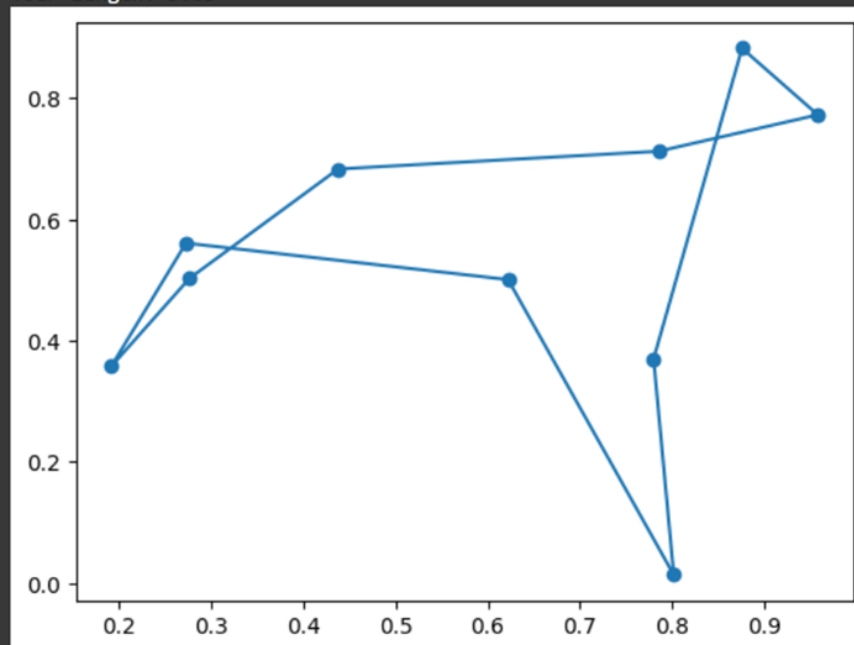
        # Xác suất tồn tại ở trạng thái năng lượng cao (kém hơn) giảm theo hàm mũ khi năng lượng tăng.
        if delta < 0 or math.exp(-delta / T) > random.random():
            current_tour = neighbor
            current_length = neighbor_length

        # Cập nhật tour tốt nhất
        if current_length < best_length:
            best_tour = current_tour.copy()
            best_length = current_length

    return best_tour, best_length
```

5 Kết quả chạy thuật toán

```
Initial tour length: 4.796953103547065
Optimized tour length (Simulated Annealing): 3.050600802958613
Tour length: 3.05
```



6 Nhận xét

Thuật toán Simulated Annealing đã cải thiện đáng kể hành trình so với ban đầu, giảm tổng độ dài tour từ 4.79 xuống còn 3.05, cho thấy khả năng tối ưu tốt. Tuy nhiên, kết

quả cuối cùng vẫn dài hơn một chút so với các thuật toán Hill Climbing trước, do đặc tính ngẫu nhiên và quá trình làm nguội ảnh hưởng đến khả năng hội tụ. Dù vậy, SA có ưu điểm nổi bật là không bị mắc kẹt tại cực trị cục bộ, giúp tìm được lời giải ổn định và có tính tổng quát cao hơn các thuật toán leo đồi truyền thống.

VII So sánh hiệu năng của các thuật toán

1 Mục tiêu

So sánh bốn thuật toán Heuristic:

- Steepest-Ascent Hill Climbing
- Stochastic Hill Climbing
- First-Choice Hill Climbing
- Simulated Annealing

2 Dựa vào 3 tiêu chí

1. Runtime (ms) — thời gian chạy trung bình.
2. Scalability — khả năng mở rộng khi tăng số lượng thành phố.
3. Best Length — giá trị hàm mục tiêu nhỏ nhất (tổng quãng đường ngắn nhất).

3 Cài đặt

```
import time

# Các thuật toán cần so sánh
algorithms = {
    "Steepest-Ascent": steepest_ascent_hill_climbing,
    "Stochastic": stochastic_hill_climbing,
    "First-Choice": first_choice_hill_climbing,
    "Simulated Annealing": simulated_annealing
}

sizes = [10, 20, 30, 40] # số thành phố để test
results = []



for n in sizes:
    tsp = random_tsp(n)
    initial_tour = random_tour(n)

    for name, algo in algorithms.items():
        # thời điểm bắt đầu
        t0 = time.time()
        best_tour, best_length = algo(tsp, initial_tour)
        t1 = time.time()

        runtime = (t1 - t0) * 1e3 # ms

        results.append({
            "Algorithm": name,
            "Cities": n,
            "Best Length": round(best_length, 2),
            "Runtime (ms)": round(runtime, 2)
        })
```

4 Kết quả sau khi cài đặt

	Algorithm	Cities	Best Length	Runtime (ms)	
0	Steepest-Ascent	10	2.79	1.70	
1	Stochastic	10	2.67	2.27	
2	First-Choice	10	3.11	5.23	
3	Simulated Annealing	10	3.33	5.33	
4	Steepest-Ascent	20	5.37	25.50	
5	Stochastic	20	4.82	49.71	
6	First-Choice	20	5.83	8.05	
7	Simulated Annealing	20	8.35	7.67	
8	Steepest-Ascent	30	7.35	78.03	
9	Stochastic	30	5.99	174.43	
10	First-Choice	30	7.30	9.64	
11	Simulated Annealing	30	12.29	10.25	
12	Steepest-Ascent	40	7.07	270.86	
13	Stochastic	40	6.70	889.29	
14	First-Choice	40	7.59	15.16	
15	Simulated Annealing	40	16.91	13.88	

5 Nhận xét

Stochastic Hill Climbing cho kết quả tốt nhất ở bài toán nhỏ, Simulated Annealing ổn định và tránh kẹt cực trị khi bài toán lớn, First-Choice nhanh nhất nhưng ít chính xác, còn Steepest-Ascent ổn định, tốn thời gian khi số thành phố tăng.

VIII Genetic algorithm

1 Mục tiêu

Tìm đường đi ngắn nhất qua tất cả các thành phố bằng cách mô phỏng quá trình tiến hóa tự nhiên (chọn lọc, lai ghép, đột biến).

2 Nguyên lý hoạt động

1. Khởi tạo quần thể: Sinh ngẫu nhiên nhiều tour (nghiệm).
2. Đánh giá độ thích nghi (fitness): Dựa trên nghịch đảo độ dài đường đi ($1 / \text{tour_length}$).
3. Chọn lọc (Selection): Dùng Roulette Wheel để chọn các cá thể tốt hơn làm “bố mẹ”.
4. Lai ghép (Crossover): Kết hợp hai cá thể bằng Ordered Crossover (OX) để tạo con.
5. Đột biến (Mutation): Đảo vị trí hai thành phố trong tour với xác suất nhỏ.
6. Cập nhật thế hệ: Thay quần thể cũ bằng quần thể con và lưu nghiệm tốt nhất.

3 Cài đặt thuật toán

```
def genetic_algorithm_tsp(tsp, pop_size=100, generations=200, mutation_rate=0.1):
    n = len(tsp["pos"])

    # --- Tạo quần thể ban đầu ---
    population = [random_tour(n) for _ in range(pop_size)]

    def fitness(tour):
        return 1 / tour_length(tsp, tour) # fitness càng cao càng tốt

    def selection(pop):
        # Chọn theo roulett (variable) t: Any
        weights = [fitness(t) for t in pop]
        return random.choices(pop, weights=weights, k=2)

    def crossover(parent1, parent2):
        # Ordered Crossover (OX)
        start, end = sorted(random.sample(range(n), 2))
        child = [None] * n
        child[start:end] = parent1[start:end]
        fill_values = [c for c in parent2 if c not in child]
        j = 0
        for i in range(n):
            if child[i] is None:
                child[i] = fill_values[j]
                j += 1
        return child

    def mutate(tour):
        if random.random() < mutation_rate:
            i, j = random.sample(range(n), 2)
            tour[i], tour[j] = tour[j], tour[i]
        return tour

    best_tour = min(population, key=lambda t: tour_length(tsp, t))
    best_length = tour_length(tsp, best_tour)
```

```

# --- Vòng lặp tiến hóa ---
for gen in range(generations):
    new_population = []
    for _ in range(pop_size):
        parent1, parent2 = selection(population)
        child = crossover(parent1, parent2)
        child = mutate(child)
        new_population.append(child)

    population = new_population

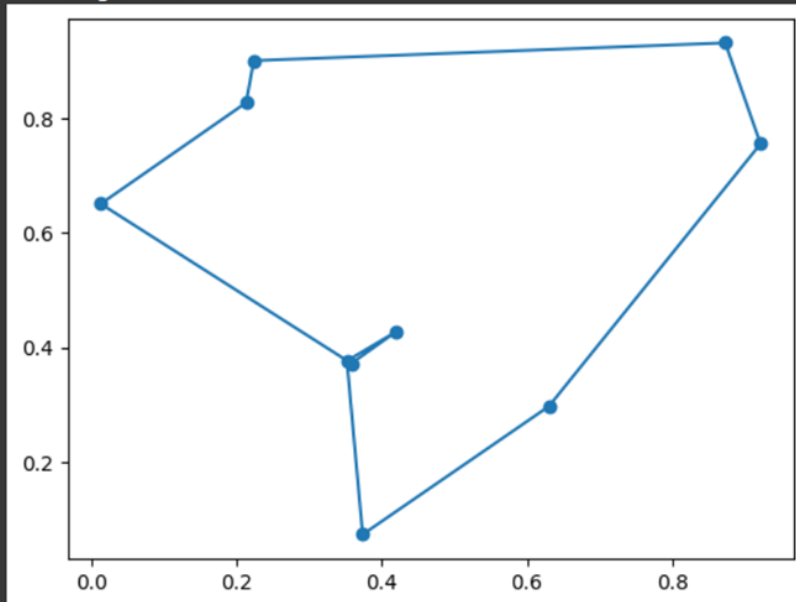
    # Cập nhật best
    current_best = min(population, key=lambda t: tour_length(tsp, t))
    current_length = tour_length(tsp, current_best)
    if current_length < best_length:
        best_tour = current_best
        best_length = current_length

return best_tour, best_length

```

4 Kết quả sau khi cài đặt thuật toán

Optimized tour length (Genetic Algorithm): 2.975497494917244
 Tour length: 2.98



5 Nhận xét

Thuật toán di truyền (GA) cho kết quả tour tối ưu có độ dài 2.98, thể hiện khả năng tìm nghiệm tốt nhờ cơ chế lai ghép (crossover) và đột biến (mutation) giúp khám phá không gian nghiệm hiệu quả, tránh được việc mắc kẹt ở cực trị cục bộ; tuy nhiên quá trình tiến hóa qua nhiều thế hệ nên thời gian chạy chậm hơn các thuật toán leo đồi. So với Steepest-Ascent, Stochastic, First-Choice và Simulated Annealing, thuật toán Genetic Algorithm cho kết quả tốt và ổn định hơn, đặc biệt ở các bài toán có số thành phố lớn; tuy nhiên nó tốn tài nguyên và thời gian hơn do cần xử lý quần thể lớn và nhiều vòng lặp tiến hóa.