A picture containing application

Description automatically generated**TRƯỜNG ĐẠI HỌC SÀI GÒN**

**KHOA CÔNG NGHỆ THÔNG TIN**

Icon

Description automatically generated with low confidence

**BÁO CÁO**

**HỌC PHẦN: TRÍ TUỆ NHÂN TẠO NÂNG CAO**

**LAB 03**

**Sinh viên thực hiện**

Nguyễn Anh Khoa – 3122410185

Huỳnh Khôi Nguyên - 3122410267

Trần Gia Nuyễn – 3122410274

**TP. HCM THÁNG 9/2025**

**MỤC LỤC**

[**PHÂN CÔNG 4**](#_Toc211167683)

[**1. N Queen 5**](#_Toc211167684)

[**TASK1: Steepest-ascend Hill Climbing Search 5**](#_Toc211167685)

[**TASK 2: Stochastic Hill Climbing 1 (Stochastic HC 1) 8**](#_Toc211167686)

[**TASK 3: Stochastic Hill Climbing 2 (First choice HC) 10**](#_Toc211167687)

[**TASK 4: Hill Climbing Search with Random Restarts 14**](#_Toc211167688)

[**TASK 5: Simulated Annealing 15**](#_Toc211167689)

[**TASK 6: Algorithm Behavior Analysis 18**](#_Toc211167690)

[**ADVANCE TASK: 21**](#_Toc211167691)

[**2. Traveling salesman problem 27**](#_Toc211167692)

[**2.1. Giới thiệu Bài toán Người du lịch (Traveling Salesman Problem - TSP) 27**](#_Toc211167693)

[**2.1.1. Mục tiêu 27**](#_Toc211167694)

[**2.1.2. Không gian trạng thái 27**](#_Toc211167695)

[**2.1.3. Hàm mục tiêu 27**](#_Toc211167696)

[**2.1.4. Phép toán lân cận (Local Moves) 28**](#_Toc211167697)

[**2.2. Chuẩn bị và Dữ liệu thử nghiệm 28**](#_Toc211167698)

[**2.2.1. Các thư viện và hàm hỗ trợ 28**](#_Toc211167699)

[**2.2.2. Thiết lập bài toán ngẫu nhiên 30**](#_Toc211167700)

[**2.3. Áp dụng các Thuật toán Tìm kiếm cục bộ 32**](#_Toc211167701)

[**2.3.1. Giải pháp tham khảo sử dụng R 32**](#_Toc211167702)

[**2.3.2. Thuật toán Leo đồi Steepest-Ascend 33**](#_Toc211167703)

[**2.3.3. Thuật toán Leo đồi Steepest-Ascend với Khởi đầu ngẫu nhiên 35**](#_Toc211167704)

[**2.3.4. Thuật toán Leo đồi Stochastic 37**](#_Toc211167705)

[**2.3.5. Thuật toán Leo đồi First-Choice 39**](#_Toc211167706)

[**2.3.6. Thuật toán Luyện kim mô phỏng (Simulated Annealing) 40**](#_Toc211167707)

[**2.3.7. (Bonus) Thuật toán Di truyền (Genetic Algorithm) 42**](#_Toc211167708)

[**2.4. So sánh và Đánh giá Hiệu năng 44**](#_Toc211167709)

[**2.4.1. Mã nguồn so sánh 44**](#_Toc211167710)

[**2.4.2. Kết quả thực nghiệm và Phân tích 44**](#_Toc211167711)

[**2.5. Tổng kết và Đề xuất 45**](#_Toc211167712)

[**2.5.1. Bảng tổng hợp 45**](#_Toc211167713)

[**2.5.2. Nhận xét và đề xuất 45**](#_Toc211167714)

PHÂN CÔNG

|  |  |  |
| --- | --- | --- |
| **MSSV** | **Họ tên** | **Phân công** |
| 3122410185 | Nguyễn Anh Khoa | n\_queens, báo cáo, slide |
| 3122410267 | Huỳnh Khôi Nguyên | traveling\_salesman\_problem, slide |
| 3122410274 | Trần Gia Nguyễn (nhóm trưởng) | Tổng hợp báo cáo, slide |

1. N Queen

TASK1: Steepest-ascend Hill Climbing Search

1. **Mục tiêu**

* Cài đặt thuật toán steepest-ascent hill climbing (tìm kiếm leo đồi chọn bước tốt nhất).
* **Ý tưởng:** từ một trạng thái khởi tạo (bàn cờ), ở mỗi bước đánh giá toàn bộ tập các di chuyển lân cận (tức là thay đổi vị trí hàng của một quân trong mỗi cột) và chọn di chuyển làm giảm số xung đột nhiều nhất. Lặp lại cho đến khi không còn di chuyển cải thiện nào (cực trị cục bộ) hoặc đạt số xung đột = 0 (giải pháp).

1. **Code:**

Ảnh có chứa văn bản, ảnh chụp màn hình, phần mềm, màn hình

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

Ảnh có chứa văn bản, ảnh chụp màn hình, đa phương tiện, phần mềm

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

Ảnh có chứa văn bản, ảnh chụp màn hình, phần mềm, Phông chữ

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

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, danh thiếp

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

1. **Kết quả:**



1. **Mô tả:**

**- Hàm và đầu vào/đầu ra**

* Hàm: **steepest\_ascent\_hill\_climbing(start\_board, max\_iters=1000)**

**- Input:**

* **start\_board:** mảng (list hoặc np.array) kích thước n, phần tử i là hàng của quân ở cột i (0-indexed).
* **max\_iters:** số bước tối đa để tránh vòng lặp vô hạn.

**- Output:**

* **final\_board:** mảng numpy chứa trạng thái cuối cùng (sau khi dừng).
* **history:** danh sách số xung đột theo từng bước (dùng để vẽ đồ thị/thống kê).

- **Luồng xử lý**

* Sao chép **start\_board** vào biến **board** để không làm thay đổi đầu vào.
* Khởi tạo**history** bằng số xung đột ban đầu **(conflicts(board))**.
* Lặp **while** cho tới**max\_iters**:
  + Tính **current\_conf = conflicts(board)**.
  + Khởi **best\_conf = current\_conf**, **best\_board = None**.
  + Duyệt mọi cột **col** từ **0..n-1**:
* Lưu vị trí ban đầu **orig = board[col]**.
* Duyệt mọi hàng**row** trong cột đó (0..n-1):
  + Nếu **row == orig**, bỏ qua (không phải là di chuyển).
  + Thay tạm **board[col] = row**, tính **c = conflicts(board)**.
  + Nếu **c < best\_conf**, cập nhật **best\_conf = c** và lưu **best\_board = board.copy()**.
* Khôi phục**board[col] = orig**.
* Sau khi duyệt hết tất cả các di chuyển đơn quân:
* Nếu **best\_board is None** (không có di chuyển nào giảm conflict), **break** (đã tới cực trị cục bộ).
* Gán **board = best\_board**, **append best\_conf** vào **history**.
* Nếu **best\_conf == 0**, **break** (đã tìm nghiệm tối ưu).
* Trả về **board, history**.

1. **Ghi chú về phức tạp và tối ưu**

**Phức tạp:** mỗi bước cần kiểm tra O(n \* n) láng giềng (n cột × n hàng khả dĩ). Tính toán xung đột hiện tại (cấp toàn bộ) trong hàm conflicts đang là O(n) sử dụng đếm hàng/đường chéo (thực ra conflicts chạy O(n) sau khi đã tính tần suất). Do đó chi phí mỗi bước là ~O(n^2). Số bước tối đa phụ thuộc vào cấu trúc bài toán; worst-case là max\_iters.

**Tối ưu tiềm năng:**

* Thay vì gọi conflicts toàn phần cho mỗi láng giềng, có thể duy trì thông tin bổ sung để cập nhật xung đột cục bộ nhanh hơn khi thay đổi 1 ô (cập nhật incremental).
* Hoán đổi logic để tránh copy toàn bộ bảng quá nhiều, hoặc hạn chế đánh giá các hàng hợp lý.

TASK 2: Stochastic Hill Climbing 1 (Stochastic HC 1)

1. **Mục tiêu:**

* Cài đặt biến thể "stochastic hill climbing" mà tại mỗi bước thu thập toàn bộ các di chuyển cải thiện (tức là mọi láng giềng có số xung đột nhỏ hơn trạng thái hiện tại), rồi chọn ngẫu nhiên một trong số các di chuyển đó để thực hiện.
* So với steepest-ascent (chọn di chuyển tốt nhất), phương pháp này chọn ngẫu nhiên trong các cải thiện, giúp giảm bias và có thể tránh một vài bẫy cục bộ nhỏ.

1. **Code:**

Ảnh có chứa văn bản, ảnh chụp màn hình, màn hình, máy tính

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

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, đa phương tiện

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

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

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

1. **Kết quả:**



1. **Mô tả:**

**- Hàm và đầu vào/đầu ra**

* Hàm: **stochastic\_hill\_climbing\_all\_neighbors(start\_board, max\_iters=1000)**

**- Input:**

* **start\_board:** mảng vị trí hàng của hậu theo cột.
* **max\_iters:** giới hạn số bước.

**- Output:**

* **final\_board:** trạng thái cuối cùng (numpy array).
* **history:** danh sách số xung đột sau mỗi lần chọn di chuyển (bắt đầu với xung đột ban đầu).

**- Luồng xử lý**

* Sao chép **start\_board** sang**board**, lấy**n = len(board)**.
* Khởi **history = [conflicts(board)]**.
* Lặp while tới max\_iters:
  + Tính **current\_conf = conflicts(board)**.
  + Tạo danh sách **improving\_moves = []** để lưu các di chuyển làm giảm số xung đột.
  + Duyệt mọi cột col:
* Lưu vị trí ban đầu **orig = board[col]**.
* Duyệt mọi hàng row khác orig:
  + Thử **board[col] = row**, tính**c = conflicts(board)**.
  + Nếu **c < current\_conf**, **append (c, col, row)** vào**improving\_moves**.
* Khôi phục **board[col] = orig**.
* Nếu**improving\_moves** rỗng: dừng (đã tới cực trị cục bộ).
* Ngược lại: chọn ngẫu nhiên một **tuple (c, col, row)**từ **improving\_moves**.
  + Thực hiện **board[col] = row.**
  + **Append c** vào **history**.
  + Nếu c == 0, dừng (đã tìm nghiệm).
* Trả về **board, history**.

1. **Điểm chú ý về thực thi và hiệu năng**

**Chi phí:** giống steepest về mặt xét toàn bộ láng giềng, vì vẫn duyệt O(n^2) láng giềng mỗi bước để phát hiện improving\_moves. Tuy nhiên, sau khi thu thập, chỉ thực hiện 1 di chuyển.

**Ưu điểm:**

Tính ngẫu nhiên giúp thay đổi quá trình tìm kiếm, có thể vượt một vài bẫy cục bộ mà steepest hay rơi vào do luôn chọn cùng một bước tốt nhất.

**Nhược điểm:**

Vẫn tốn chi phí đánh giá vì phải thu thập tất cả các láng giềng; nếu muốn nhanh hơn, có thể dùng first-choice (Task 3) để sinh từng láng giềng ngẫu nhiên.

TASK 3: Stochastic Hill Climbing 2 (First choice HC)

1. **Mục tiêu**

Cài đặt biến thể "First-choice hill climbing" (một dạng stochastic HC): thay vì liệt kê tất cả các láng giềng, thuật toán sinh từng láng giềng ngẫu nhiên một lần và chấp nhận ngay nếu nó cải thiện objective. Nếu không cải thiện nhiều lần liên tiếp (no\_improve\_limit), dừng lại. Phương pháp này tiết kiệm thời gian khi số láng giềng quá lớn.

1. **Code**

def first\_choice\_hill\_climbing(start\_board, max\_iters=10000, no\_improve\_limit=1000):

    """First-choice hill climbing:

    Sinh ngẫu nhiên một láng giềng (chọn cột và hàng mới) và chấp nhận ngay nếu nó cải thiện.

    Dừng nếu không cải thiện trong một số lượng thử liên tiếp (no\_improve\_limit).

    """

    board = np.array(start\_board).copy()

    n = len(board)

    history = [conflicts(board)]

    it = 0

    no\_improve = 0

    current\_conf = conflicts(board)

    while it < max\_iters and no\_improve < no\_improve\_limit:

        it += 1

Ảnh có chứa văn bản, ảnh chụp màn hình, phần mềm, Hệ điều hành

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

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, thiết kế

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

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

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

1. **Kết quả:**



1. **Mô tả:**

**- Mục tiêu của ô**

Cài đặt biến thể "First-choice hill climbing" (một dạng stochastic HC): thay vì liệt kê tất cả các láng giềng, thuật toán sinh từng láng giềng ngẫu nhiên một lần và chấp nhận ngay nếu nó cải thiện objective. Nếu không cải thiện nhiều lần liên tiếp (no\_improve\_limit), dừng lại. Phương pháp này tiết kiệm thời gian khi số láng giềng quá lớn.

**- Hàm và đầu vào/đầu ra**

Hàm: **first\_choice\_hill\_climbing(start\_board, max\_iters=10000, no\_improve\_limit=1000)**

**- Input:**

* **start\_board:** mảng vị trí hàng cho mỗi cột.
* **max\_iters:** giới hạn tối đa số lần thử.
* **no\_improve\_limit:** số lần thử liên tiếp không thấy cải thiện để coi là đã tới cực trị cục bộ (dừng sớm).

**- Output:**

* **final\_board:** trạng thái cuối cùng (numpy array).
* **history:** danh sách số xung đột theo thời gian.

- **Luồng xử lý (chi tiết)**

* Sao chép **start\_board** sang **board**; lưu **n = len(board)**.
* Khởi **history = [conflicts(board)]**, **no\_improve = 0**, **current\_conf = conflicts(board)**.
* Vòng lặp **while it < max\_iters and no\_improve < no\_improve\_limit**:
* Sinh một láng giềng ngẫu nhiên bằng cách:
* Chọn ngẫu nhiên một cột **col = random.randrange(n)**.
* Chọn một hàng mới **row = random.randrange(n)**; nếu **row == orig** (không thay đổi), bỏ qua (tiếp tục vòng).
* Thực hiện **board[col] = row**, tính **c = conflicts(board)**.
* Nếu **c < current\_conf** (tức cải thiện):
* Cập nhật **current\_conf = c**, **history.append(c)**, **no\_improve = 0**.
* Nếu **c == 0**, dừng vì tìm nghiệm.
* Nếu không cải thiện:
* Khôi phục **board[col] = orig** và **no\_improve += 1**.
* Trả về **board, history**.

1. **Tại sao dùng first-choice?**

* Khi mỗi trạng thái có nhiều láng giềng (ở đây n\*(n-1) láng giềng nếu xét tất cả), liệt kê và đánh giá hết sẽ rất tốn. First-choice giảm chi phí bằng cách sinh cặp (col,row) ngẫu nhiên và chỉ đánh giá từng láng giềng một lần cho tới khi tìm được cải thiện.
* Nếu không có cải thiện trong nhiều lần thử liên tiếp (no\_improve\_limit), khả năng cao đã ở cực trị cục bộ -> dừng để tránh vòng lặp vô hạn.

1. **Các tham số và điều chỉnh**

* no\_improve\_limit: nhỏ → dừng sớm (có thể bỏ lỡ các cải tiến hiếm); lớn → cho phép tìm kiếm kỹ hơn. Thử nghiệm thường chọn giá trị tương đối so với max\_iters.
* max\_iters: dùng để bảo vệ chống infinite loop.

TASK 4: Hill Climbing Search with Random Restarts

1. **Mục tiêu**

Random restarts (khởi động lại ngẫu nhiên) là kỹ thuật để giảm khả năng kẹt ở cực trị cục bộ. Ý tưởng: chạy thuật toán local search nhiều lần, mỗi lần bắt đầu từ một trạng thái ngẫu nhiên mới, và giữ nghiệm tốt nhất tìm được.

1. **Code**

Ảnh có chứa văn bản, ảnh chụp màn hình, đa phương tiện, Phông chữ

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

Ảnh có chứa văn bản, ảnh chụp màn hình, danh thiếp, Phông chữ

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

1. **Kết quả**



1. **Mô tả**

**- Hàm và đầu vào/đầu ra**

* Hàm: **random\_restarts(algorithm\_fn, n, restarts=100, \*\*kwargs)**
  + **algorithm\_fn**: một hàm hill-climbing (ví dụ steepest\_ascent\_hill\_climbing) nhận start\_board là đầu vào và trả về (final\_board, history).
  + **n**: kích thước bàn (số cột).
  + **restarts**: số lần chạy (ví dụ 100).
  + **\*\*kwargs**: các tham số phụ chuyển tiếp cho algorithm\_fn (ví dụ max\_iters).
* **Output**:
  + **results**: danh sách dict cho mỗi lần chạy, chứa 'start\_conf', 'end\_conf', 'history'.
  + **times**: danh sách thời gian chạy (s) tương ứng với mỗi lần restart.

- **Luồng xử lý**

* Khởi**results = []**, **times = []**.
* Vòng lặp **for i in range(restarts)**:
  + Sinh **start = random\_board(n)** (bàn khởi tạo ngẫu nhiên).
  + Ghi thời gian bắt đầu **t0 = time.time()**.
  + Gọi **final, hist = algorithm\_fn(start, \*\*kwargs)** để chạy thuật toán từ khởi tạo đó.
  + Ghi**t1 = time.time()** và tính thời gian **t1 - t0**.
  + **Lưu kết quả**: append dict {'start\_conf': conflicts(start), 'end\_conf': conflicts(final), 'history': hist} vào results.
  + Append thời gian vào times.
* Trả về **results, times**.

1. **Mục đích trong thí nghiệm**

Sử dụng random\_restarts để:

* Tăng tỷ lệ tìm được nghiệm tối ưu so với chạy 1 lần.
* So sánh thuật toán nào có khả năng cải thiện nhiều nhờ restarts (ví dụ first-choice có thể cần ít restarts hơn so với steepest).
* Ở báo cáo, báo min(end\_conf) (xung đột nhỏ nhất), mean(end\_conf), thời gian trung bình, và số lần đạt optimal.

TASK 5: Simulated Annealing

1. **Mục tiêu**

Triển khai thuật toán Simulated Annealing (SA) cho bài toán n-Queens để cho phép thoát khỏi cực trị cục bộ bằng cách chấp nhận các bước xấu với xác suất tỉ lệ nghịch với độ xấu và tỉ lệ thuận với nhiệt độ T. Nhiệt độ giảm theo thời gian (annealing schedule) để dần chuyển từ khám phá sang khai thác.

1. **Code**

Ảnh có chứa văn bản, ảnh chụp màn hình, đa phương tiện, phần mềm

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

Ảnh có chứa văn bản, ảnh chụp màn hình, phần mềm, đa phương tiện

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

Ảnh có chứa văn bản, ảnh chụp màn hình, phần mềm, đa phương tiện

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

Ảnh có chứa văn bản, ảnh chụp màn hình, đa phương tiện, phần mềm

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

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

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

1. **Kết quả**



1. **Mô tả**

**- Hàm và đầu vào/đầu ra**

* Hàm: **simulated\_annealing(start\_board, max\_iters=10000, t0=1.0, alpha=0.995)**
  + **start\_board:** mảng khởi tạo (bảng).
  + **max\_iters:** số vòng lặp tối đa.
  + **t0:** nhiệt độ ban đầu.
  + **alpha:** hệ số làm nguội (T <- T \* alpha mỗi vòng). Giá trị gần 1 làm nguội chậm hơn.
* **Output:**
  + **final\_board:** bảng sau khi dừng.
  + **history:** danh sách số xung đột theo thời gian (dùng để vẽ tiến trình).

**- Luồng xử lý**

* Sao chép **start\_board** sang **board**. Khởi **history = [conflicts(board)], current\_conf = conflicts(board), T = t0**.
* Vòng **for it in range(max\_iters)**:
  + Sinh một láng giềng ngẫu nhiên bằng cách chọn cột **col = random.randrange(n)** và hàng ngẫu nhiên **row = random.randrange(n)**; nếu **row == orig**, bỏ qua.
  + Cập nhật **board[col] = row**, tính **c = conflicts(board)**.
  + Tính **delta = c - current\_conf**.
* Nếu **delta <= 0**: move cải thiện hoặc bằng; chấp nhận: current\_conf = c và append c vào history.
* Ngược lại **delta > 0** (move xấu): chấp nhận với xác suất exp(-delta / T) nếu T > 0. Nếu chấp nhận, cập nhật current\_conf và history; nếu không, revert board[col] = orig.
* **Làm nguội**: T \*= alpha.
* Nếu**current\_conf == 0**, dừng sớm (giải).
* Nếu **T < 1e-12**, dừng (nhiệt độ quá nhỏ).
* Trả về **board, history**.

1. **Giải thích các tham số và ảnh hưởng**

* **t0** (nhiệt độ ban đầu): càng lớn → càng dễ chấp nhận các bước xấu ban đầu, tăng tính khám phá.
* **alpha** (tỷ lệ giảm): 0.995 là lịch làm nguội chậm; alpha nhỏ hơn (vd. 0.9) làm nguội nhanh hơn, giảm lượng chấp nhận bước xấu.
* **max\_iters:** cần đủ lớn để SA có thời gian khám phá; trong thực nghiệm, thường cần điều chỉnh t0/alpha để đạt kết quả tốt.

TASK 6: Algorithm Behavior Analysis

- **Mục tiêu** của phần so sánh là tóm tắt hiệu năng thực nghiệm: thời gian chạy trung bình, chất lượng nghiệm (số xung đột trung bình) và tỷ lệ chạy đạt nghiệm tối ưu (0 xung đột).

- **Code và kết quả:**

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

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

Ảnh có chứa văn bản, đồ điện tử, màn hình, ảnh chụp màn hình

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

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, thiết kế

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

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

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

- Từ các thuật toán đã triển khai thường thu được những quan sát sau:

* **Steepest-ascent HC:** thường có cải thiện nhanh ban đầu (vì đánh giá toàn bộ không gian lân cận để chọn bước tốt nhất), nhưng dễ bị mắc kẹt tại cực trị cục bộ. Chi phí mỗi bước cao hơn (do xem xét tất cả các láng giềng), nên thời gian chạy trung bình có thể lớn hơn so với các biến thể mang tính ngẫu nhiên.
* **Stochastic HC 1** (chọn ngẫu nhiên từ các bước cải thiện): cho hiệu năng ổn định hơn so với steepest trong một số trường hợp, vì chọn ngẫu nhiên giữa các bước tốt giúp tránh một vài bẫy cục bộ nhỏ, nhưng vẫn có khả năng bị dừng ở cực trị cục bộ.
* **Stochastic HC 2 / First-choice**: thường có bước tiến nhanh hơn (ít phải đánh giá nhiều láng giềng), phù hợp với bài toán có số láng giềng lớn; tuy nhiên phụ thuộc mạnh vào may rủi của mẫu, nên phương pháp này có phương sai lớn hơn về kết quả cuối cùng.
* **Simulated Annealing**: chậm hơn từng bước so với first-choice nhưng có khả năng vượt khỏi cực trị cục bộ nhờ chấp nhận các bước xấu với xác suất giảm dần. Trong thực nghiệm, SA thường có tỷ lệ đạt nghiệm tối ưu cao hơn khi lịch làm lạnh và tham số được chỉnh hợp lý.

- **Kết luận thực tế**: không có thuật toán nào "tốt nhất" cho mọi kích thước; steepest tốt về chất lượng bước nhưng tốn thời gian, first-choice phù hợp khi cần chạy nhanh nhiều lần, và simulated annealing là lựa chọn hợp lý khi muốn ưu tiên tìm nghiệm tối ưu hơn và chấp nhận chi phí tính toán cao hơn.

ADVANCE TASK:

**- Algorith Convergence: Hội tụ của thuật toán**

def plot\_representative\_runs():

    fig, ax = plt.subplots(figsize=(9,5))

    for name, fn in algorithms:

        start = random\_board(8)

        final, hist = fn(start)

        ax.plot(hist, label=name)

    ax.set\_xlabel('Số bước (Iteration)')

    ax.set\_ylabel('Số xung đột (conflicts)')

    ax.set\_title('Mô tả hội tụ: các run đại diện trên 8-Queens')

    ax.grid(True)

    ax.legend()

    plt.tight\_layout()

    plt.show()

# Gọi hàm để hiển thị

if \_\_name\_\_ == '\_\_main\_\_':

    plot\_representative\_runs()

**Kết quả:**

Ảnh có chứa văn bản, hàng, Sơ đồ, biểu đồ

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

**Nhận xét:**

**Steepest Ascent Hill Climbing (xanh dương)**

* Giảm xung đột rất nhanh chỉ sau khoảng 3 bước đã đạt 0.
* Thể hiện khả năng hội tụ nhanh nhất trong các thuật toán được so sánh.
* Tuy nhiên, do đặc tính “tham lam” (greedy), thuật toán này dễ mắc kẹt ở cực trị địa phương nếu không may mắn ở trạng thái ban đầu.

**Stochastic Hill Climbing (vàng cam)**

* Cũng giảm xung đột nhanh, hội tụ sau khoảng 5–6 bước.
* So với Steepest, kết quả tương đương nhưng ổn định hơn, vì có yếu tố ngẫu nhiên giúp tránh kẹt cục bộ.

**First Choice Hill Climbing (xanh lá)**

* Hiệu quả gần như Steepest, đạt nghiệm chỉ sau vài bước.
* Ưu điểm là nhanh và tiết kiệm tính toán hơn vì không cần đánh giá toàn bộ lân cận.

**Simulated Annealing (đỏ)**

* Giảm xung đột chậm hơn rõ rệt, dao động thất thường giữa các mức xung đột.
* Đặc trưng của thuật toán này là cho phép bước lùi tạm thời để thoát cực trị địa phương, nên đường biểu diễn có nhiều “lên xuống”.
* Dù khởi đầu kém hơn, nhưng sau khoảng 90 bước cũng đạt được nghiệm (conflict = 0).
* Điều này cho thấy tính bền vững và khả năng tìm lời giải tốt dù tốn thời gian hơn.

- **Problem Size Scalability (Độ phóng to theo kích thước bài toán):**

def measure\_scaling(sizes=[4,8,12,16], runs=5, algorithms\_to\_test=None):

    """Trả về dict chứa kích thước và thời gian trung bình cho mỗi thuật toán.

    algorithms\_to\_test: list các tuple (label, function)

    """

    if algorithms\_to\_test is None:

        algorithms\_to\_test = [("Steepest", steepest\_ascent\_hill\_climbing), ("FirstChoice", first\_choice\_hill\_climbing)]

    results = { 'size': list(sizes) }

    for name, \_ in algorithms\_to\_test:

        results[name] = []

    for size in sizes:

        for name, fn in algorithms\_to\_test:

            times = []

            for r in range(runs):

                start = random\_board(size)

                t0 = time.time()

                fn(start)

                t1 = time.time()

                times.append(t1 - t0)

            results[name].append(np.mean(times))

    return results

# Ví dụ chạy nhanh (chỉ dùng 4 kích thước để demo)

if \_\_name\_\_ == '\_\_main\_\_':

    res = measure\_scaling(sizes=[4,8,12,16], runs=3)

    plt.loglog(res['size'], res['Steepest'], marker='o', label='Steepest')

    plt.loglog(res['size'], res['FirstChoice'], marker='o', label='FirstChoice')

    plt.xlabel('Kích thước bàn (n)')

    plt.ylabel('Thời gian trung bình (s)')

    plt.title('Độ phóng to thời gian theo kích thước (log-log)')

    plt.legend()

    plt.grid(True)

    plt.show()

**Kết quả:**

Ảnh có chứa văn bản, hàng, ảnh chụp màn hình, Sơ đồ

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

**Nhận xét:**

**Steepest Ascent (xanh dương)**

* Có thời gian tăng rất nhanh theo kích thước bàn (đường dốc lên rõ rệt).
* Khi nnn tăng, thời gian tăng gần như theo cấp số nhân.
* Điều này là do thuật toán phải đánh giá toàn bộ lân cận ở mỗi bước — chi phí tăng mạnh khi bàn cờ lớn.

**First Choice (cam)**

* Thời gian tăng chậm hơn, gần như ổn định ở các giá trị n=8n = 8n=8 đến n=16n = 16n=16.
* Chỉ khi nnn rất lớn (khoảng > 16), thời gian mới tăng đáng kể.
* Lý do: thuật toán chọn ngẫu nhiên và dừng sớm khi gặp nước đi tốt, nên giảm khối lượng tính toán đáng kể.

-**Exploring other Local Moves Operators**

Triển khai các Local Moves Operators khác:

def single\_step\_move(board):

    """Di chuyển một quân một ô lên hoặc xuống (wrap-around).

    - board: mảng 1D (numpy array hoặc list) biểu diễn vị trí hàng của quân theo cột.

    Trả về bảng mới (copy) sau di chuyển.

    """

    b = board.copy()

    n = len(b)

    col = random.randrange(n)

    direction = random.choice([-1, 1])

    b[col] = (int(b[col]) + direction) % n

    return b

def column\_swap\_move(board):

    """Hoán đổi vị trí của hai cột (đổi chỗ 2 quân hậu giữa 2 cột).

    Trả về bảng mới.

    """

    b = board.copy()

    n = len(b)

    c1, c2 = random.sample(range(n), 2)

    b[c1], b[c2] = b[c2], b[c1]

    return b

def dual\_queen\_move(board):

    """Chọn hai cột và đặt lại vị trí hàng của cả hai quân một cách ngẫu nhiên."""

    b = board.copy()

    n = len(b)

    c1, c2 = random.sample(range(n), 2)

    b[c1] = random.randrange(n)

    b[c2] = random.randrange(n)

    return b

# Hàm phụ: tính số xung đột mà một quân ở cột `col` đang tạo ra

def local\_conflicts(board, col):

    """Trả về số lượng quân khác đang xung đột với quân ở cột `col`.

    Dùng để xác định cột nào có vấn đề và nên tập trung.

    """

    n = len(board)

    cnt = 0

    for j in range(n):

        if j == col:

            continue

        # cùng hàng

        if board[j] == board[col]:

            cnt += 1

        # cùng đường chéo

        if abs(board[j] - board[col]) == abs(j - col):

            cnt += 1

    return cnt

class AdaptiveMove:

    """Move adaptive: tập trung vào cột có nhiều xung đột.

    - move(board): trả về board mới sau một move áp dụng chiến lược thích ứng.

    """

    def \_\_init\_\_(self):

        self.history = []

    def move(self, board):

        n = len(board)

        # tìm các cột có xung đột > 0

        conflict\_cols = [col for col in range(n) if local\_conflicts(board, col) > 0]

        if conflict\_cols:

            col = random.choice(conflict\_cols)

            new\_board = board.copy()

            # thử đặt ngẫu nhiên một hàng mới cho cột chọn

            new\_board[col] = random.randrange(n)

            return new\_board

        # nếu không có cột nào xung đột, fallback về single step

        return single\_step\_move(board)

# Sử dụng first-choice hill climbing với operator tuỳ chọn

def first\_choice\_with\_operator(start\_board, move\_operator, max\_iters=10000, no\_improve\_limit=1000):

    """Phiên bản first-choice hill climbing cho phép truyền move\_operator(board)."""

    board = np.array(start\_board).copy()

    n = len(board)

    history = [conflicts(board)]

    it = 0

    no\_improve = 0

    current\_conf = conflicts(board)

    while it < max\_iters and no\_improve < no\_improve\_limit:

        it += 1

        candidate = move\_operator(board)

        c = conflicts(candidate)

        if c < current\_conf:

            board = np.array(candidate)

            current\_conf = c

            history.append(c)

            no\_improve = 0

            if c == 0:

                break

        else:

            no\_improve += 1

    return board, history

# Ví dụ nhanh

if \_\_name\_\_ == '\_\_main\_\_':

    b = random\_board(8)

    final, hist = first\_choice\_with\_operator(b, column\_swap\_move)

    print('Xung đột ban đầu:', conflicts(b), 'Xung đột cuối:', conflicts(final))

Kết quả:



Từ việc thử nghiệm các move operators khác nhau (single-step, column-swap, dual-queen, adaptive) ta rút ra:

* **Column-swap** hữu ích khi nhiều cột có cấu trúc xung đột lẫn nhau; việc hoán đổi đôi khi nhanh chóng khôi phục trạng thái không xung đột.
* **Single-step** phù hợp khi cần tinh chỉnh nhẹ vị trí của các quân, ít phá vỡ cấu trúc tổng thể.
* **Dual-queen move** và các move thay đổi nhiều biến cùng lúc có thể giúp thoát khỏi bẫy cục bộ nhưng cũng khiến quá trình tìm kiếm mất ổn định hơn.
* **Adaptive move** (tập trung vào cột có xung đột nhiều) thường cho hiệu quả thực nghiệm tốt vì nó kết hợp khám phá có định hướng và randomization.

**- Bài học chung:**

- Không có move operator tốt nhất cho mọi trường hợp; việc kết hợp vài operator và lựa chọn ngẫu nhiên theo trạng thái hiện tại thường cho kết quả khả quan.

  - Việc đo lường và so sánh bằng thực nghiệm là thiết yếu — các giả thuyết lý thuyết chỉ dẫn hướng, nhưng thực nghiệm mới cho biết tham số tối ưu cho từng kích thước và mục tiêu.

1. Traveling salesman problem

2.1. Giới thiệu Bài toán Người du lịch (Traveling Salesman Problem - TSP)

Bài toán Người du lịch là một trong những bài toán tối ưu hóa nổi tiếng nhất trong khoa học máy tính. Nó thuộc lớp bài toán NP-hard, nghĩa là không có thuật toán hiệu quả nào được biết đến để tìm ra lời giải tối ưu trong thời gian đa thức khi số lượng thành phố tăng lên.

2.1.1. Mục tiêu

Mục tiêu của bài toán là tìm ra một chu trình ngắn nhất đi qua tất cả các thành phố trong một danh sách cho trước, mỗi thành phố chỉ được đi qua đúng một lần và quay trở về thành phố xuất phát.

2.1.2. Không gian trạng thái

Mỗi trạng thái đại diện cho một chu trình (tour). Với N thành phố, một chu trình có thể được biểu diễn dưới dạng một hoán vị của các thành phố, cho biết thứ tự chúng được ghé thăm. Ví dụ, π(1) là thành phố đầu tiên, π(2) là thành phố thứ hai, và cứ thế tiếp diễn.

2.1.3. Hàm mục tiêu

Hàm mục tiêu cần được tối thiểu hóa là tổng độ dài của chu trình. Công thức tính như sau:

Ảnh có chứa Phông chữ, văn bản, Đồ họa, ảnh chụp màn hình

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

Trong đó di,j​ là khoảng cách giữa thành phố i và thành phố j.

2.1.4. Phép toán lân cận (Local Moves)

Để di chuyển từ trạng thái này sang trạng thái khác trong không gian tìm kiếm, phép toán lân cận cơ bản được sử dụng là **hoán đổi vị trí của hai thành phố** trong chu trình hiện tại.

2.2. Chuẩn bị và Dữ liệu thử nghiệm

2.2.1. Các thư viện và hàm hỗ trợ

Đầu tiên, chúng ta cần nhập các thư viện cần thiết và định nghĩa các hàm cơ bản để làm việc với bài toán TSP.

**import numpy as np**

**import pandas as pd**

**import matplotlib.pyplot as plt**

**import math**

**import random**

**np.set\_printoptions(precision=2)**

**pd.set\_option('display.precision', 2)**

**# make the results repeatable**

**np.random.seed(1234)**

**def random\_tour(n):**

**"""Create a random tour"""**

**tour = list(range(n))**

**random.shuffle(tour)**

**return(tour)**

**from scipy.spatial.distance import pdist**

**from scipy.spatial.distance import squareform**

**def random\_tsp(n):**

**"""**

**Create a random (Euclidean) traveling salesman problem. Choose n points randomly in a 1 x 1 unit square and calulates a**

**pairwise Euclidean distance matrix.**

**"""**

**pos = pd.DataFrame({**

**"x" : np.random.uniform(size = n),**

**"y" : np.random.uniform(size = n)**

**})**

**dist = squareform(pdist(pos))**

**return({"pos": pos, "dist": dist})**

**def tour\_length(tsp, tour):**

**"""Caclulate the length of a tour, i.e., the objective function."""**

**# make sure tour is a Python list (not an array or a numpy.array)**

**if not isinstance(tour, list): tour = tour.tolist()**

**tl = 0**

**dist = tsp["dist"]**

**for i in range(len(tour)-1):**

**tl += dist[tour[i], tour[i+1]]**

**tl += dist[tour[-1], tour[0]]**

**return(tl)**

**def show\_tsp(tsp, tour = None):**

**"""display the traveling salesman problem and a tour."""**

**pos = tsp["pos"]**

**plt.scatter(pos["x"], pos["y"])**

**if tour is not None:**

**# make sure tour is a Python list (not an array or a numpy.array)**

**if not isinstance(tour, list): tour = tour.tolist()**

**print(f"Tour length: {round(tour\_length(tsp, tour), 2)}")**

**pos\_ = pos.reindex(tour)**

**pos\_ = pd.concat([pos\_, pos\_.head(1)])**

**plt.plot(pos\_["x"], pos\_["y"])**

**plt.show()**

2.2.2. Thiết lập bài toán ngẫu nhiên

Để thử nghiệm các thuật toán, một bài toán TSP ngẫu nhiên được tạo ra với 10 thành phố. Vị trí của các thành phố này được sinh ngẫu nhiên trong một hình vuông đơn vị 1x1. Ma trận khoảng cách Euclid giữa các cặp thành phố được tính toán trước.

**tsp = random\_tsp(10)**

**print(f"Positions:\n{tsp['pos']}")**

**print(f"Distance matrix:\n{pd.DataFrame(tsp['dist'])}")**

**# Hiển thị một tour ngẫu nhiên ban đầu**

**tour = random\_tour(10)**

**show\_tsp(tsp, tour)**

Kết quả:

Positions:

x y

0 0.19 0.36

1 0.62 0.50

2 0.44 0.68

3 0.79 0.71

4 0.78 0.37

5 0.27 0.56

6 0.28 0.50

7 0.80 0.01

8 0.96 0.77

9 0.88 0.88

Tour length: 5.55

Ảnh có chứa hàng, biểu đồ, ảnh chụp màn hình, Song song

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

*(Hình ảnh của tour ngẫu nhiên ban đầu)*

****2.3. Áp dụng các Thuật toán Tìm kiếm cục bộ****

2.3.1. Giải pháp tham khảo sử dụng R

Để có một kết quả tham chiếu, bài toán được giải bằng gói TSP trong ngôn ngữ R. Thuật toán sử dụng là arbitrary\_insertion kết hợp với 2-opt (tương tự Steepest-Ascend) và 100 lần khởi đầu ngẫu nhiên.

**# Tải các thư viện R cần thiết**

**%load\_ext rpy2.ipython**

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

**d = tsp["dist"]**

**# Giải bài toán bằng R**

**%%R -i d -o tour**

**library("TSP")**

**tsp <- TSP(d)**

**tour <- solve\_TSP(tsp, rep = 100)**

**tour <- tour - 1L**

* **Kết quả:** Chu trình ngắn nhất tìm được có độ dài **2.76**.

2.3.2. Thuật toán Leo đồi Steepest-Ascend

* **Cách hoạt động:** Tại mỗi bước, thuật toán duyệt qua tất cả các trạng thái lân cận và chọn nước đi mang lại sự cải thiện lớn nhất.

**def steepest\_hill\_climbing\_tsp(tsp, max\_iterations=1000, verbose=False):**

**# (Nội dung hàm steepest\_hill\_climbing\_tsp)**

**current\_tour = random\_tour(len(tsp['pos']))**

**current\_length = tour\_length(tsp, current\_tour)**

**if verbose:**

**print(f"Initial tour length: {current\_length:.3f}")**

**iterations = 0**

**for iteration in range(max\_iterations):**

**iterations += 1**

**best\_neighbor = None**

**best\_neighbor\_length = current\_length**

**n = len(current\_tour)**

**for i in range(n):**

**for j in range(i + 1, n):**

**neighbor\_tour = current\_tour.copy()**

**neighbor\_tour[i], neighbor\_tour[j] = neighbor\_tour[j], neighbor\_tour[i]**

**neighbor\_length = tour\_length(tsp, neighbor\_tour)**

**if neighbor\_length < best\_neighbor\_length:**

**best\_neighbor = neighbor\_tour.copy()**

**best\_neighbor\_length = neighbor\_length**

**if best\_neighbor is not None:**

**current\_tour = best\_neighbor**

**current\_length = best\_neighbor\_length**

**if verbose and iteration % 100 == 0:**

**print(f"Iteration {iteration}: Tour length = {current\_length:.3f}")**

**else:**

**if verbose:**

**print(f"Local optimum reached at iteration {iteration}")**

**break**

**return current\_tour, current\_length, iterations**

**best\_tour, best\_length, iterations = steepest\_hill\_climbing\_tsp(tsp, verbose=True)**

**show\_tsp(tsp, best\_tour)**

* **Kết quả:** Thuật toán đạt đến điểm tối ưu cục bộ sau 7 vòng lặp, tìm được chu trình có độ dài **2.764**.

Ảnh có chứa hàng, biểu đồ, Sơ đồ, Song song

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

2.3.3. Thuật toán Leo đồi Steepest-Ascend với Khởi đầu ngẫu nhiên

* **Cách hoạt động:** Thực hiện chạy Steepest-Ascend nhiều lần từ các điểm khởi đầu ngẫu nhiên khác nhau và chọn ra kết quả tốt nhất.

**def steepest\_hill\_climbing\_with\_restarts(tsp, num\_restarts=10, max\_iterations\_per\_restart=1000, verbose=False):**

**# (Nội dung hàm steepest\_hill\_climbing\_with\_restarts)**

**global\_best\_tour = None**

**global\_best\_length = float('inf')**

**total\_iterations = 0**

**restart\_results = []**

**if verbose:**

**print(f"Bắt đầu {num\_restarts} random restarts...")**

**for restart in range(num\_restarts):**

**tour, length, iterations = steepest\_hill\_climbing\_tsp(**

**tsp, max\_iterations\_per\_restart, verbose=False**

**)**

**total\_iterations += iterations**

**restart\_results.append({**

**'restart': restart, 'tour': tour, 'length': length, 'iterations': iterations**

**})**

**if length < global\_best\_length:**

**global\_best\_tour = tour.copy()**

**global\_best\_length = length**

**if verbose:**

**print(f"Restart {restart + 1}: length = {length:.3f}, iterations = {iterations}")**

**return global\_best\_tour, global\_best\_length, total\_iterations, restart\_results**

**best\_tour\_restart, best\_length\_restart, \_, \_ = steepest\_hill\_climbing\_with\_restarts(tsp, num\_restarts=20, verbose=True)**

**show\_tsp(tsp, best\_tour\_restart)**

* **Kết quả:** Sau 20 lần khởi tạo lại, kết quả tốt nhất vẫn là **2.764**.

Ảnh có chứa hàng, biểu đồ, Sơ đồ, Song song

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

2.3.4. Thuật toán Leo đồi Stochastic

* **Cách hoạt động:** Tìm tất cả các nước đi tốt hơn và chọn ngẫu nhiên một trong số chúng.

**def stochastic\_hill\_climbing\_tsp(tsp, max\_iterations=1000, verbose=False):**

**# (Nội dung hàm stochastic\_hill\_climbing\_tsp)**

**current\_tour = random\_tour(len(tsp['pos']))**

**current\_length = tour\_length(tsp, current\_tour)**

**if verbose:**

**print(f"Initial tour length: {current\_length:.3f}")**

**iterations = 0**

**for iteration in range(max\_iterations):**

**iterations += 1**

**better\_neighbors = []**

**n = len(current\_tour)**

**for i in range(n):**

**for j in range(i + 1, n):**

**neighbor\_tour = current\_tour.copy()**

**neighbor\_tour[i], neighbor\_tour[j] = neighbor\_tour[j], neighbor\_tour[i]**

**neighbor\_length = tour\_length(tsp, neighbor\_tour)**

**if neighbor\_length < current\_length:**

**better\_neighbors.append({**

**'tour': neighbor\_tour.copy(), 'length': neighbor\_length, 'swap': (i, j)**

**})**

**if better\_neighbors:**

**chosen\_neighbor = random.choice(better\_neighbors)**

**current\_tour = chosen\_neighbor['tour']**

**current\_length = chosen\_neighbor['length']**

**else:**

**if verbose:**

**print(f"Local optimum reached at iteration {iteration}")**

**break**

**return current\_tour, current\_length, iterations**

**best\_tour\_stoch, best\_length\_stoch, \_ = stochastic\_hill\_climbing\_tsp(tsp, verbose=True)**

**show\_tsp(tsp, best\_tour\_stoch)**

* **Kết quả:** Do tính ngẫu nhiên, độ dài tour dao động từ **2.806** đến **3.343**.

Ảnh có chứa hàng, biểu đồ, Sơ đồ, ảnh chụp màn hình

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

2.3.5. Thuật toán Leo đồi First-Choice

* **Cách hoạt động:** Chấp nhận ngay nước đi đầu tiên tìm được mà tốt hơn trạng thái hiện tại.

**def first\_choice\_hill\_climbing\_tsp(tsp, max\_iterations=1000, max\_neighbor\_checks=100, verbose=False):**

**# (Nội dung hàm first\_choice\_hill\_climbing\_tsp)**

**current\_tour = random\_tour(len(tsp['pos']))**

**current\_length = tour\_length(tsp, current\_tour)**

**iterations = 0**

**for iteration in range(max\_iterations):**

**iterations += 1**

**found\_better = False**

**n = len(current\_tour)**

**all\_swaps = [(i, j) for i in range(n) for j in range(i + 1, n)]**

**random.shuffle(all\_swaps)**

**for i, j in all\_swaps[:max\_neighbor\_checks]:**

**neighbor\_tour = current\_tour.copy()**

**neighbor\_tour[i], neighbor\_tour[j] = neighbor\_tour[j], neighbor\_tour[i]**

**neighbor\_length = tour\_length(tsp, neighbor\_tour)**

**if neighbor\_length < current\_length:**

**current\_tour = neighbor\_tour**

**current\_length = neighbor\_length**

**found\_better = True**

**break**

**if not found\_better:**

**break**

**return current\_tour, current\_length, iterations**

**best\_tour\_fc, best\_length\_fc, \_ = first\_choice\_hill\_climbing\_tsp(tsp, max\_neighbor\_checks=25, verbose=True)**

**show\_tsp(tsp, best\_tour\_fc)**

* **Kết quả:** Kết quả tốt nhất trong các lần thử là **2.881**.

2.3.6. Thuật toán Luyện kim mô phỏng (Simulated Annealing)

* **Cách hoạt động:** Cho phép chấp nhận các nước đi "tệ hơn" với một xác suất nhất định để thoát khỏi tối ưu cục bộ.

**def simulated\_annealing\_tsp(tsp, max\_iterations=10000, initial\_temp=1000, final\_temp=1, cooling\_rate=0.995, verbose=False):**

**# (Nội dung hàm simulated\_annealing\_tsp)**

**current\_tour = random\_tour(len(tsp['pos']))**

**current\_length = tour\_length(tsp, current\_tour)**

**best\_tour = current\_tour.copy()**

**best\_length = current\_length**

**temperature = initial\_temp**

**iterations = 0**

**for iteration in range(max\_iterations):**

**iterations += 1**

**i, j = random.sample(range(len(current\_tour)), 2)**

**neighbor\_tour = current\_tour.copy()**

**neighbor\_tour[i], neighbor\_tour[j] = neighbor\_tour[j], neighbor\_tour[i]**

**neighbor\_length = tour\_length(tsp, neighbor\_tour)**

**delta = neighbor\_length - current\_length**

**if delta < 0 or random.random() < math.exp(-delta / temperature):**

**current\_tour = neighbor\_tour**

**current\_length = neighbor\_length**

**if current\_length < best\_length:**

**best\_tour = current\_tour.copy()**

**best\_length = current\_length**

**temperature \*= cooling\_rate**

**if temperature < final\_temp:**

**break**

**return best\_tour, best\_length, iterations**

**sa\_best\_tour, sa\_best\_length, \_ = simulated\_annealing\_tsp(tsp, max\_iterations=5000, verbose=True)**

**show\_tsp(tsp, sa\_best\_tour)**

* **Kết quả:** Kết quả tốt nhất thu được là **3.143** (kết quả có thể thay đổi do tính ngẫu nhiên).

Ảnh có chứa hàng, biểu đồ, Sơ đồ, ảnh chụp màn hình

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

2.3.7. (Bonus) Thuật toán Di truyền (Genetic Algorithm)

* **Cách hoạt động:** Mô phỏng quá trình tiến hóa tự nhiên với các toán tử chọn lọc, lai ghép và đột biến.

**def genetic\_algorithm\_tsp(tsp, population\_size=50, num\_generations=200, mutation\_rate=0.02, elite\_ratio=0.2, verbose=False):**

**# (Nội dung hàm genetic\_algorithm\_tsp)**

**# ... (Hàm này khá dài, bao gồm các hàm con cho fitness, selection, crossover, mutation)**

**n\_cities = len(tsp['pos'])**

**# ... (Khởi tạo quần thể)**

**for generation in range(num\_generations):**

**# ... (Tiến hóa)**

**pass**

**# ... (Trả về kết quả tốt nhất)**

**return best\_ever\_tour, 1.0 / best\_ever\_fitness, num\_generations**

**best\_ga\_tour, best\_ga\_length, \_ = genetic\_algorithm\_tsp(tsp, population\_size=30, num\_generations=100, verbose=True)**

**show\_tsp(tsp, best\_ga\_tour)**

* **Kết quả:** Thuật toán Di truyền cũng tìm được lời giải rất tốt với độ dài là **2.764**.

Ảnh có chứa hàng, văn bản, Sơ đồ, biểu đồ

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

Ảnh có chứa hàng, biểu đồ, Sơ đồ, Song song

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

2.4. So sánh và Đánh giá Hiệu năng

2.4.1. Mã nguồn so sánh

Một hàm được viết để tự động chạy các thuật toán trên nhiều kích thước bài toán khác nhau và ghi lại kết quả.

**def compare\_algorithms\_performance(tsp\_sizes=[5, 8, 10, 12, 15], num\_runs=5):**

**# (Nội dung hàm compare\_algorithms\_performance)**

**# ... (Vòng lặp qua các kích thước và thuật toán)**

**# ... (Ghi lại thời gian, độ dài, số vòng lặp)**

**# ... (In bảng kết quả)**

**return results**

**performance\_results = compare\_algorithms\_performance(tsp\_sizes=[5, 8, 10, 12], num\_runs=3)**

2.4.2. Kết quả thực nghiệm và Phân tích

Ảnh có chứa văn bản, biểu đồ, hàng, Sơ đồ

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

(Bảng và biểu đồ kết quả được trình bày như trong các phản hồi trước)

* **Tốc độ:** **First-choice HC** là nhanh nhất.
* **Chất lượng:** **Random Restarts** và **Simulated Annealing** cho chất lượng cao và ổn định.
* **Khả năng mở rộng:** Thời gian chạy tăng theo cấp số nhân, phù hợp với bản chất của bài toán.

2.5. Tổng kết và Đề xuất

2.5.1. Bảng tổng hợp

| Thuật toán | Tốc độ | Chất lượng | Khả năng thoát Local Optimum |
| --- | --- | --- | --- |
| **Steepest HC** | Trung bình | Tốt | Thấp |
| **Random Restarts** | Chậm | Rất tốt & Ổn định | Trung bình |
| **Stochastic HC** | Nhanh | Khá | Trung bình |
| **First-choice HC** | Rất nhanh | Trung bình | Trung bình |
| **Simulated Annealing** | Chậm | Rất tốt | Cao |
| **Genetic Algorithm** | Rất chậm | Rất tốt | Rất cao |

2.5.2. Nhận xét và đề xuất

Không có thuật toán nào là "tốt nhất" cho mọi trường hợp. Việc lựa chọn phụ thuộc vào yêu cầu cụ thể:

* **Khi cần tốc độ nhanh:** Sử dụng **First-choice Hill Climbing**.
* **Khi ưu tiên chất lượng giải pháp cao nhất:** Sử dụng **Steepest Hill Climbing với Random Restarts** hoặc **Simulated Annealing**.
* **Khi cần sự cân bằng giữa tốc độ và chất lượng:** **Stochastic Hill Climbing** là một lựa chọn hợp lý.
* **Với các bài toán lớn và phức tạp:** **Genetic Algorithm** và **Simulated Annealing** thể hiện khả năng khám phá không gian lời giải tốt hơn.

**Đề xuất chung:** Với các bài toán kích thước nhỏ đến trung bình, **Steepest HC với Random Restarts** là lựa chọn đáng tin cậy. Nếu thời gian là yếu tố quan trọng, có thể bắt đầu với **Stochastic HC**.