

BỘ GIÁO DỤC VÀ ĐÀO TẠO
ĐẠI HỌC KINH TẾ TP HỒ CHÍ MINH
TRƯỜNG CÔNG NGHỆ VÀ THIẾT KẾ



ĐỒ ÁN MÔN HỌC

ĐỀ TÀI 10:

GIẢI BÀI NGƯỜI DU LỊCH (TRAVELLING SALESMAN
PROBLEM)

BẰNG GIẢI THUẬT SIMULATED ANNEALING

Học phần: Trí Tuệ Nhân Tạo

Nhóm Sinh Viên:

1. ĐỖ THÁI GIA HY
2. HÀ QUANG ĐẠI
3. THÁI THỦY ĐỨC
4. TRẦN HUỖNH HUY THÔNG

Chuyên Ngành: KHOA HỌC DỮ LIỆU

Khóa: K49

Giảng Viên: TS. Đặng Ngọc Hoàng Thành

TP. Hồ Chí Minh, Ngày 26 tháng 5 năm 2025

MỤC LỤC

MỤC LỤC	1
CHƯƠNG 1. TỔNG QUAN	3
1.1. Giới Thiệu Về Traveling Salesman Problem (TSP)	3
1.2. Phát Biểu Bài Toán	4
1.3. Một Số Hướng Tiếp Cận Giải Quyết Bài Toán	5
2.1. Giới Thiệu Về Simulated Annealing	7
2.2.1 Ứng Dụng Giải thuật Simulated Annealing Cho Bài Toán Người du lịch	10
2.2.a. Các hàm khởi tạo/tính toán cần thiết	10
2.2.b Hàm tạo láng giềng (Neighbor Methods)	11
2.2.c. Hàm chính (Simulated Annealing)	11
2.2.d. Hàm vẽ đồ thị (Mathplotlib)	14
2.2.2 Ứng Dụng Giải thuật Simulated Annealing Cho Bài Toán Người du lịch mở rộng 15	
2.2.2.1 Bài toán mở rộng:	15
2.2.2.2 Các hàm sử dụng:	15
CHƯƠNG 3. CÁC KẾT QUẢ THỰC NGHIỆM	21
3.1. Các Tình Huống Trong TSP cơ bản	21
3.2. Phân Tích và Đánh Giá TSP cơ bản	25
3.3 Các tình huống trong bài toán TSP mở rộng	26
3.4 Phân tích và đánh giá kết quả của bài toán TSP mở rộng	38
3.4.1. Tập 15 khách hàng (num_customers = 15, num_vehicles = 2, seed = 15)	38
3.4.2. Tập 50 khách hàng (num_customers = 50, num_vehicles = 4, seed = 50)	39
3.4.3. Tập 100 khách hàng (num_customers = 100, num_vehicles = 7, seed = 100)	39
3.4.1.4 Một số suy nghĩ và kết luận	40
Tác động của phương pháp tự động điều chỉnh nhiệt độ khởi đầu	40
Tác động của phương pháp tạo lời giải khởi đầu	40
Tác động của phương pháp tạo hàng xóm	41
CHƯƠNG 4. KẾT LUẬN	42
4.1. Các Kết Quả Đạt Được	42
Tác động của phương pháp khởi tạo lời giải:	42
4.2. Những Hạn Chế và Hướng Phát Triển	42
TÀI LIỆU THAM KHẢO	44

PHỤ LỤC	45
Hướng dẫn cài đặt	45
UI và hướng dẫn sử dụng	46
PHÂN CÔNG	47

CHƯƠNG 1. TỔNG QUAN

1.1. Giới Thiệu Về Traveling Salesman Problem (TSP)

Hãy tưởng tượng bạn đang lên kế hoạch cho một chuyến đi toàn nước Mỹ, bắt đầu từ San Francisco và muốn ghé qua tất cả các thành phố lớn một lần duy nhất, trước khi quay trở lại điểm xuất phát. Câu hỏi đặt ra: Lộ trình ngắn nhất để làm điều đó là gì? Đây chính là cốt lõi của bài toán **Traveling Salesman Problem (TSP)**.

Bài toán **Traveling Salesman Problem (TSP)** là một bài toán tối ưu tổ hợp nổi tiếng, yêu cầu tìm hành trình **ngắn nhất** đi qua mỗi thành phố **đúng một lần** và quay về điểm xuất phát. Đây là một bài toán **NP-hard** trong lý thuyết độ phức tạp tính toán, nghĩa là không có thuật toán giải chính xác nào có thể chạy trong thời gian đa thức cho mọi trường hợp.

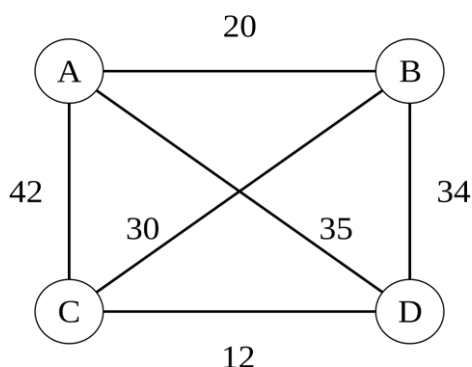
Giả định quan trọng:

- Đồ thị được mô hình hóa đầy đủ (complete graph), mỗi cặp thành phố đều có đường đi trực tiếp.
- Trong bài toán TSP, ta giả định rằng khoảng cách từ thành phố A đến B **luôn bằng** khoảng cách từ B đến A (bài toán đối xứng)
- Giả định rằng buộc bất đẳng thức tam giác: đường đi trực tiếp từ A \rightarrow B **luôn ngắn hơn** đường đi A \rightarrow C \rightarrow B.

Ứng dụng thực tế của TSP:

- Lập kế hoạch lộ trình giao hàng: Tối ưu hóa đường đi cho xe giao hàng để giảm chi phí nhiên liệu và thời gian.
- Thiết kế mạch in: Tối ưu hóa đường đi của đầu hàn trong quá trình sản xuất mạch điện tử.
- Chuỗi DNA: Tìm thứ tự sắp xếp các đoạn DNA sao cho tổng độ tương đồng giữa các đoạn là lớn nhất.
- Quan sát thiên văn: Tối ưu hóa thứ tự quan sát các thiên thể để giảm thời gian di chuyển kính thiên văn.

Minh họa:



Với:

- $d(A,D) = 35$
- $d(A,B) = 20$; $d(B,D) = 34$

$$\Rightarrow A \rightarrow B \rightarrow D = 54 > A \rightarrow D = 35$$

\Rightarrow Đi trực tiếp là ngắn nhất.

1.2. Phát Biểu Bài Toán

Cho trước một tập hợp gồm n thành phố, cùng với chi phí di chuyển giữa mỗi cặp thành phố (có thể là khoảng cách, thời gian hoặc chi phí thực tế), mục tiêu của bài toán là tìm một hành trình khép kín xuất phát từ một thành phố, đi qua **mỗi thành phố đúng một lần** và quay trở lại điểm xuất phát, sao cho **tổng chi phí di chuyển là nhỏ nhất**.

Dữ liệu đầu vào:

- n : số lượng thành phố.
- $d(i, j)$: chi phí hoặc khoảng cách từ thành phố i đến thành phố j
- Đồ thị đầy đủ $G = (V, E)$ với tập đỉnh V gồm n thành phố, và tập cạnh E nối mọi cặp thành phố.

Biến quyết định:

- $x_{ij} \in \{0, 1\}$: biến nhị phân, bằng 1 nếu hành trình đi trực tiếp từ thành phố i đến j , ngược lại bằng 0.

Hàm mục tiêu:

Tối thiểu tổng chi phí di chuyển:

$$\min \sum_{i=1}^n \sum_{\substack{j=1 \\ j \neq i}}^n d(i, j) \cdot x_{ij}$$

Ràng buộc:

1. Mỗi thành phố chỉ được đến đúng một lần:

$$\sum_{\substack{i=1 \\ i \neq j}}^n x_{ij} = 1, \quad \forall j = 1, \dots, n$$

2. Mỗi thành phố chỉ được rời đi đúng một lần:

$$\sum_{\substack{j=1 \\ j \neq i}}^n x_{ij} = 1, \quad \forall i = 1, \dots, n$$

3. Loại bỏ chu trình con (subtours): Đảm bảo rằng hành trình là duy nhất và đi qua toàn bộ các thành phố mà không bị tách thành nhiều chu trình con nhỏ. Với mọi tập con S của tập thành phố (sao cho $2 \leq |S| \leq n-1$):

$$\sum_{i \in S} \sum_{\substack{j \in S \\ j \neq i}} x_{ij} \leq |S| - 1$$

4. Ràng buộc biến:

$$x_{ij} \in \{0, 1\}, \quad \forall i, j = 1, \dots, n, \quad i \neq j$$

1.3. Một Số Hướng Tiếp Cận Giải Quyết Bài Toán

1.3.1. Brute Force

Là phương pháp đơn giản và trực quan nhất để giải bài toán TSP: **liệt kê tất cả các hoán vị có thể của các thành phố**, tính tổng chi phí của từng tour, và chọn ra hành trình có tổng chi phí nhỏ nhất.

- Xác định tất cả các thứ tự sắp xếp của n thành phố (trừ thành phố xuất phát).
- Với mỗi hoán vị, tính tổng chi phí của tour đi qua các thành phố và quay lại điểm xuất phát.
- So sánh tất cả các tour và chọn tour có tổng chi phí nhỏ nhất.

Số lượng hoán vị cần kiểm tra là $(n-1)!$ đối với bài toán đối xứng, dẫn đến thời gian xử lý tăng rất nhanh theo n . Brute Force đảm bảo tìm được lời giải tối ưu, nhưng chỉ phù hợp cho các bài toán nhỏ do chi phí tính toán lớn.

1.3.2. Nearest Neighbor

Tại mỗi bước, ta luôn chọn thành phố gần nhất trong số các thành phố chưa được thăm để di chuyển tới.

- Chọn một thành phố làm điểm khởi đầu.
- Từ thành phố hiện tại, tìm thành phố gần nhất chưa được đi qua và di chuyển đến đó.
- Lặp lại quá trình cho đến khi tất cả các thành phố đã được thăm.
- Quay trở lại thành phố xuất phát để hoàn tất chu trình.

Việc chỉ ưu tiên khoảng cách ngắn nhất ở từng bước có thể dẫn đến một hành trình tổng thể không tối ưu.

1.3.3. Greedy Algorithm

Lựa chọn các cạnh có chi phí nhỏ nhất để tạo dần một chu trình qua tất cả các thành phố.

- Xét tất cả các đoạn đường giữa các thành phố và sắp xếp theo thứ tự chi phí tăng dần.

- Duyệt qua danh sách này và chọn các đoạn đường ngắn nhất để thêm vào tour, với điều kiện:
 - Không được tạo chu trình con trước khi có đủ n đỉnh.
 - Mỗi thành phố chỉ được nối với tối đa hai cạnh.
- Khi tour đã bao gồm toàn bộ thành phố, thuật toán kết thúc.

Việc lựa chọn các đoạn đường ngắn ban đầu có thể dẫn đến việc hoàn thành chu trình một cách kém hiệu quả, do các thành phố còn lại buộc phải kết nối với các đoạn đường dài hơn.

1.3.4. K-opt

Một hướng tiếp cận hiệu quả khác trong việc cải thiện lời giải TSP là sử dụng k-opt, trong đó k là số cạnh bị cắt và thay thế tại mỗi bước nhằm tạo ra lời giải mới tốt hơn. Trong thực tế, các phiên bản phổ biến là 2-opt và 3-opt.

- **2-opt:** cắt và kết nối lại hai cạnh.
- **3-opt:** cắt và kết nối lại ba cạnh.

Cách tiếp cận k-opt được sử dụng rộng rãi để cải thiện các lời giải ban đầu giúp loại bỏ các đoạn đường không hợp lý. Tuy nhiên, k-opt vẫn là tìm kiếm cục bộ: nó chỉ duyệt qua các lời giải gần kề và chỉ chấp nhận các cải tiến.

CHƯƠNG 2. SIMULATED ANNEALING

2.1. Giới Thiệu Về Simulated Annealing

Simulated Annealing (SA) là một giải thuật tối ưu hoá gần đúng, lấy cảm hứng từ quá trình **làm nguội kim loại** trong vật lý (annealing). Khi một kim loại được nung nóng và làm nguội dần từ từ, các nguyên tử sẽ có đủ năng lượng để tái tổ chức lại, giúp hệ thống đạt đến **trạng thái có năng lượng thấp nhất** – tức là trạng thái ổn định nhất. Giải thuật SA mô phỏng chính quá trình này trong việc tìm kiếm lời giải tốt cho các bài toán tối ưu hoá tổ hợp, điển hình như **bài toán TSP**.

Nguyên lý hoạt động:

SA bắt đầu với một lời giải ban đầu (có thể là ngẫu nhiên hoặc sinh từ thuật toán đơn giản như Nearest Neighbor). Trong mỗi vòng lặp:

1. SA tạo ra một **lời giải mới** bằng cách thực hiện một phép biến đổi nhỏ từ lời giải hiện tại.
2. **SA đánh giá lời giải mới:**
 - Để tạo ra lời giải lân cận, SA thường sử dụng các phép biến đổi đơn giản nhưng hiệu quả, tiêu biểu là 2-opt: Trong mô hình, chọn hai cạnh bất kỳ, loại bỏ chúng và tạo một cặp cạnh mới bằng cách nối lại các phần còn lại.
 - Việc sử dụng 2-opt đảm bảo rằng mỗi lời giải lân cận vẫn hợp lệ (đi qua đúng mỗi thành phố một lần), đồng thời có khả năng cải thiện đáng kể lời giải.
 - Nếu lời giải mới tốt hơn (tức là có tổng chi phí ngắn hơn), nó được chấp nhận.
 - Nếu lời giải mới tệ hơn, nó vẫn có thể được chấp nhận với một xác suất xác định:

$$P = e^{-\frac{\Delta E}{T}}$$

Nếu $r \leq P$ thì chấp nhận ($r \in [0,1]$ là một số ngẫu nhiên lấy từ phân phối đều) trong đó:

- ΔE là độ chênh lệch chi phí giữa lời giải mới và lời giải hiện tại (khi lời giải mới tệ hơn).
- T là nhiệt độ hiện tại, giảm dần theo thời gian.
- Nhiệt độ T sẽ **giảm dần sau mỗi vòng lặp** theo cooling schedule:

$$T_{\text{new}} = \alpha \cdot T_{\text{old}} \quad \text{với } 0 < \alpha < 1$$

- α là hệ số làm nguội, thường nằm trong khoảng từ **0.8 đến 0.99** tùy bài toán, quy định mức độ giảm nhiệt sau mỗi vòng lặp.

Xác suất này càng lớn khi:

- ΔE nhỏ (lời giải mới không tệ nhiều)
- T còn cao (giai đoạn đầu quá trình)

Xác suất giảm nhanh khi:

- ΔE lớn
 - T nhỏ (giai đoạn sau của thuật toán)
3. Khi nhiệt độ giảm xuống rất thấp, thuật toán hoạt động giống như một tìm kiếm cục bộ, chỉ chấp nhận lời giải tốt hơn.
 4. Khi nhiệt độ đã nhỏ (dưới ngưỡng Min đã thiết lập) hoặc đạt số vòng lặp tối đa \rightarrow thuật toán dừng.

Độ phức tạp tính toán:

Giả sử:

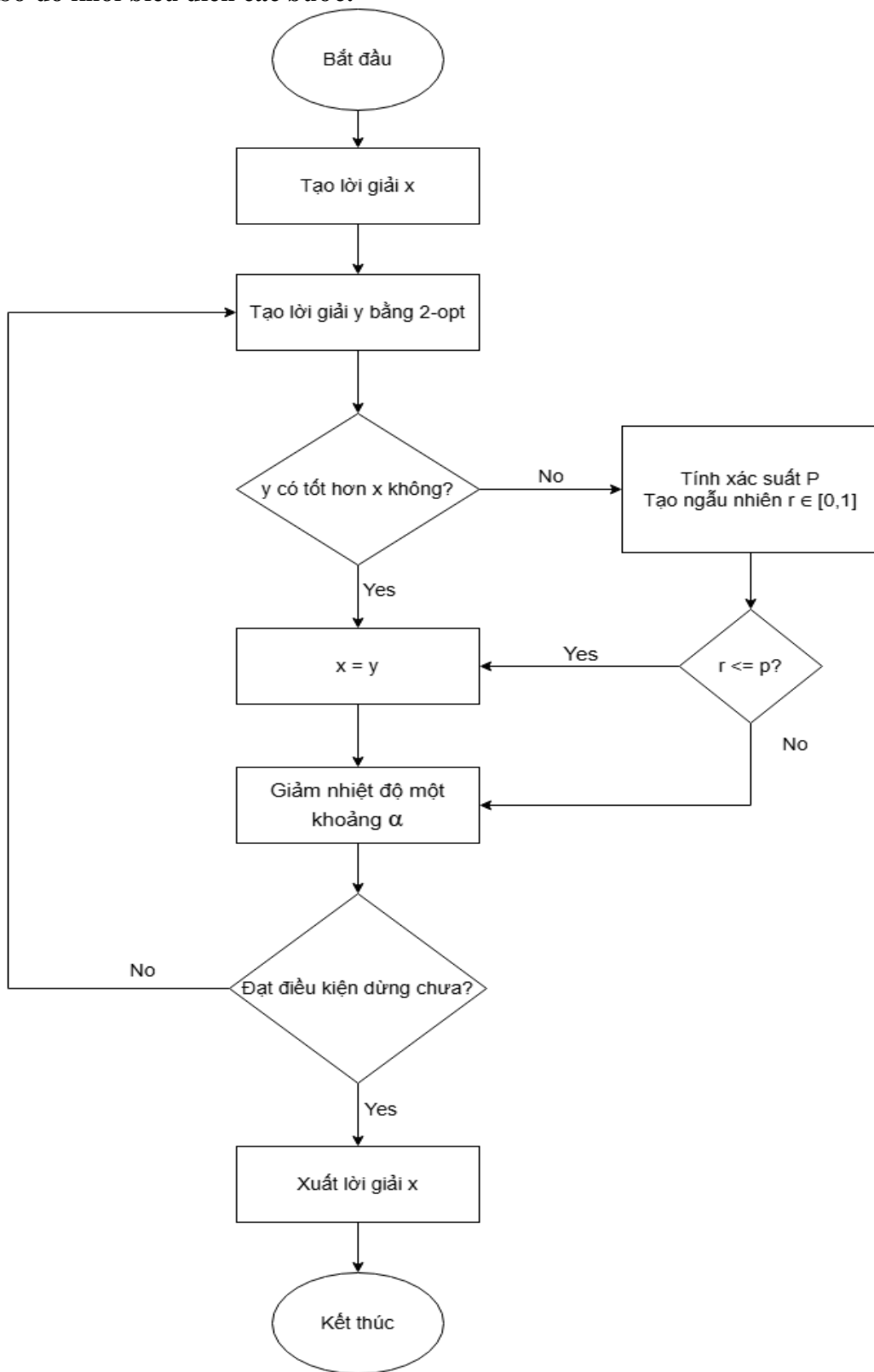
- n : số thành phố
- I : số vòng lặp tối đa (hoặc số lần sinh lời giải mới)

Thì độ phức tạp của SA ước lượng là:

$$O(I.n)$$

Mỗi bước thực hiện phép biến đổi 2-opt và đánh giá chi phí mất khoảng $O(n)$

Sơ đồ khối biểu diễn các bước:



Pseudocode của giải thuật SA có thể được biểu diễn như sau:

```
T= T0; x= initialization; xbest= x; α= alpha

While stopping criterion is not met do
  for it= 1 to MaxIt do
    s= move x by an operator
    if F(s)< F(x) then
      x=s
      if F(s)<F(xbest) then
        xbest= s
      endif
    else
      if random < exp{-(F(s)-F(x))/t} then
        x=s
      endif
    endif
  endfor

  t=α.t

endwhile
```

2.2.1 Ứng Dụng Giải thuật Simulated Annealing Cho Bài Toán Người du lịch

2.2.a. Các hàm khởi tạo/tính toán cần thiết

- Hàm **generate_cities(num_cities)** được dùng để khởi tạo danh sách tọa độ của các thành phố. Tham số đầu vào là **num_cities** tương ứng với số thành phố muốn tạo. Bằng cách sử dụng phương thức `random.uniform(0, 10)`, hàm tạo ra tọa độ x và y ngẫu nhiên trong khoảng [0, 10]. Hàm sẽ trả về một danh sách các tọa độ (x, y) của các thành phố, mỗi tọa độ là một tuple dạng (x, y). Ví dụ: [(2,2) , (3, 7), (6, 8), ...]

- Hàm **distance(city1, city2)** tính toán khoảng cách giữa hai thành phố trong không gian hai chiều. Tham số đầu vào là **city1, city2**: Hai thành phố, mỗi thành phố là một tuple tọa độ (x, y). Giá trị trả về là khoảng cách **Euclid** giữa hai thành phố, được tính bằng công thức:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- Hàm **total_distance(tour, cities)** tính tổng khoảng cách của một hành trình ghé thăm tất cả các thành phố trong một tour. Tham số đầu vào là **tour**: một danh sách các chỉ số đại diện cho thứ tự ghé thăm các thành phố và **cities**: danh sách các tọa độ thành phố. Nhờ `cities[tour[(i + 1) % len(tour)]]`, ta đảm bảo quay về thành phố xuất phát ở cuối chuyến đi (nếu thứ tự thành phố cuối = tổng số thành phố thì phần dư là 0). Giá trị trả về tổng khoảng cách của hành trình.

- Hàm **nearest_neighbor** xây dựng một hành trình bắt đầu từ một thành phố và luôn chọn thành phố gần nhất chưa ghé thăm để tiếp tục. Tham số đầu vào là **cities**: danh sách các tọa độ thành phố và trả về một tour. Đây là phương thức giải thuộc nhóm heuristic và là một cách giải khá đơn giản. Mục đích của hàm này là để khởi tạo một tour đẹp để sử dụng cho thuật

toán chính là Simulated Annealing. Ở phần sau này sẽ có một mục riêng so sánh 2 phương thức khởi tạo tour khác nhau: Random và Nearest Neighbor.

2.2.b Hàm tạo láng giềng (Neighbor Methods)

- Hàm **random_swap** đơn giản là chỉ hoán đổi ngẫu nhiên hai thành phố trong tour. Tham số đầu vào là tour hiện tại và trả về một tour mới với hai thành phố đã hoán đổi vị trí.

- Hàm **two_opt** lựa chọn 2 điểm ngẫu nhiên trong tour và đảo ngược đoạn giữa hai điểm đó để tạo ra một hành trình mới. Tham số đầu vào là tour hiện tại và trả về một tour mới với hai đoạn trong tour được đảo ngược.

- Hàm **three_opt** lựa chọn ba điểm a, b, c ngẫu nhiên trong tour và tạo ra các phương án hoán đổi khác nhau giữa ba đoạn này. Sau đó, nó lựa chọn phương án có tổng khoảng cách ngắn nhất. Các phương án mà hàm three_opt cân nhắc là:

- **Không thay đổi:** giữ nguyên ban đầu.
- **Đảo ngược đoạn đầu tiên:** đoạn giữa a và b bị đảo ngược.
- **Đảo ngược đoạn thứ hai:** đoạn giữa b và c bị đảo ngược.
- **Đảo ngược cả hai đoạn:** đảo ngược cả đoạn đầu và đoạn thứ hai.
- **Hoán đổi hai đoạn:** đoạn từ b đến c được hoán đổi lên trước đoạn từ a đến b.
- **Hoán đổi và đảo ngược đoạn đầu tiên:** hoán đổi và đoạn từ a đến b bị đảo ngược.
- **Hoán đổi và đảo ngược đoạn thứ hai:** hoán đổi và đoạn từ b đến c bị đảo ngược.
- **Hoán đổi và đảo ngược cả hai đoạn**

2.2.c. Hàm chính (Simulated Annealing)

Áp dụng thuật toán Simulated Annealing để giải bài toán TSP.

Tham số đầu vào:

- **num_cities:** Số lượng thành phố.
- **initial_temp:** Nhiệt độ ban đầu.
- **cooling_rate:** Tốc độ làm lạnh.
- **stopping_temp:** Nhiệt độ dừng.
- **max_iter:** Số vòng lặp tối đa.
- **neighbor_method:** Phương pháp tạo láng giềng (2-opt, 3-opt, hoặc random swap).
- **initialization_method:** Phương pháp khởi tạo (random hoặc nearest neighbor).
- **cooling_strategy:** Chiến lược làm lạnh (exponential, linear, logarithmic).
- **OnlyNearestNeighbor:** Chỉ dùng phương pháp nearest neighbor (so sánh kết quả).
- **CustomCity:** Cho phép người dùng tự nhập tọa độ các thành phố.

Mô tả chi tiết:

Đầu tiên, thuật toán tạo một danh sách các thành phố ở những tọa độ ngẫu nhiên. Số lượng thành phố được tạo tương ứng biến **num_cities** nhận vào. Tuy nhiên, người dùng cũng có thể tự nhập tọa độ của các thành phố vào biến **CustomCity**. Sau đó thuật toán kiểm tra biến **initialization_method** xem phương thức khởi tạo là gì. Nếu là **'nn'** (Nearest Neighbor) thì tour được đưa vào SA là một tour đã được giải bằng phương thức Nearest Neighbor. Nếu là **'random'**, SA sẽ cải thiện một tour ngẫu nhiên với chi phí ban đầu khá cao. Tiếp đến, hàm sẽ tính tổng chi phí của tour khởi tạo và gán nó vào một biến lưu trữ là **best_cost**. Tour đó cũng sẽ được lưu vào biến **best_tour**.

Sau khi khởi tạo lộ trình và tính toán chi phí ban đầu, thuật toán bắt đầu vòng lặp Simulated Annealing. Trong mỗi vòng lặp, thuật toán kiểm tra nhiệt độ hiện tại so với nhiệt độ dừng (**stopping_temp**); nếu nhiệt độ đã thấp hơn giá trị này, thuật toán sẽ dừng. Một trạng thái lân cận (tức là một lộ trình mới) được tạo ra từ lộ trình hiện tại thông qua phương pháp tạo lân cận (**neighbor_method**), có thể là **two_opt**, **three_opt**, hoặc **random_swap**. Chi phí của lộ trình mới được tính và so sánh với lộ trình hiện tại. Nếu lộ trình mới tốt hơn (chi phí thấp hơn), nó sẽ được chấp nhận ngay lập tức. Nếu không, nó vẫn có thể được chấp nhận với một **xác suất** nhất định, dựa trên giá trị **delta** (sự chênh lệch giữa chi phí lộ trình mới và hiện tại) chia cho nhiệt độ hiện tại. Quy tắc này giúp thuật toán có khả năng "thoát" khỏi các cực trị cục bộ. Công thức tính xác suất chấp nhận lộ trình tệ hơn:

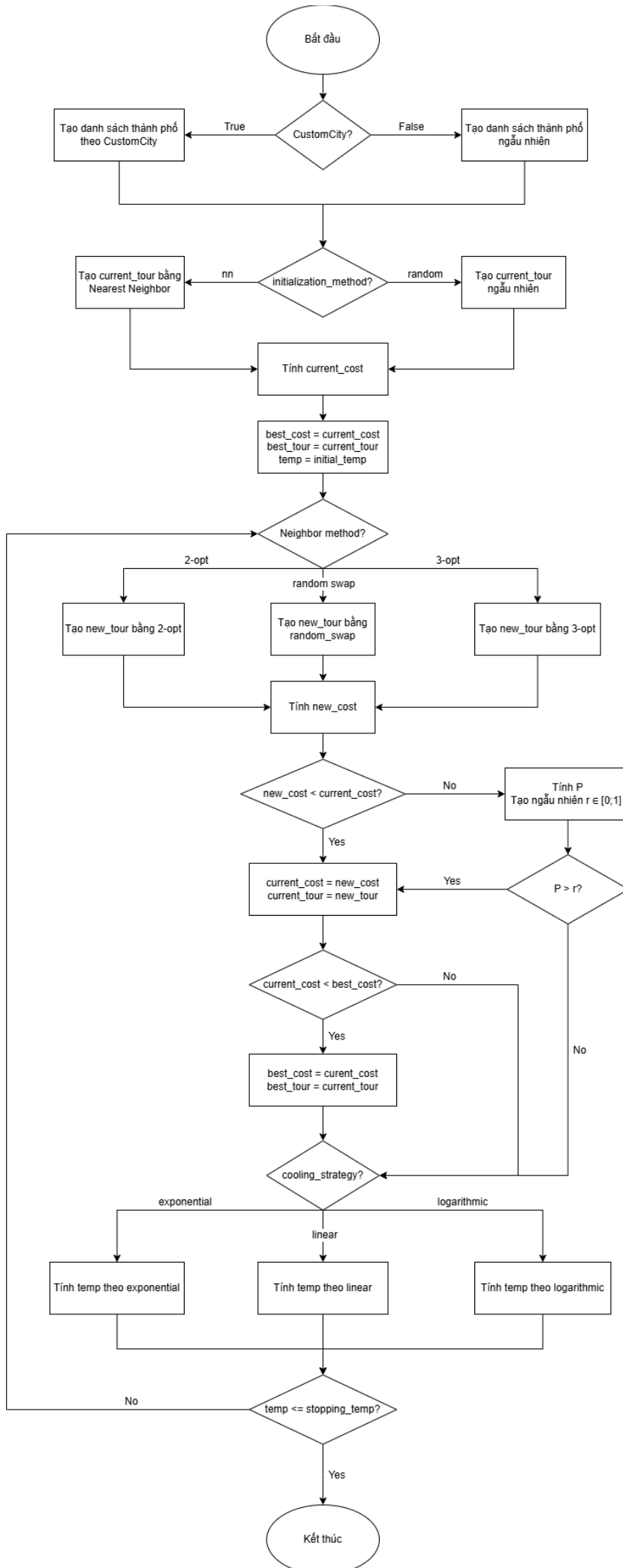
$$P = e^{\frac{-\Delta}{T}}$$

Sau mỗi vòng lặp, nhiệt độ được điều chỉnh dựa trên chiến lược làm lạnh được chọn: **exponential** (giảm theo hàm mũ), **linear** (giảm tuyến tính), hoặc **logarithmic** (giảm theo logarit). Do nhiệt độ giảm dần sau mỗi vòng lặp, xác suất chấp nhận tour xấu cũng giảm dần. Ở những vòng lặp gần cuối, thuật toán gần như chỉ chấp nhận những tour có chi phí tốt hơn. Khi vòng lặp kết thúc, hàm trả về lộ trình tốt nhất tìm được, tổng chi phí của lộ trình, danh sách thành phố, số lần lặp đã thực hiện, tên phương pháp lân cận đã dùng, và phương pháp khởi tạo ban đầu.

Trường hợp đặc biệt (OnlyNearestNeighbor = True):

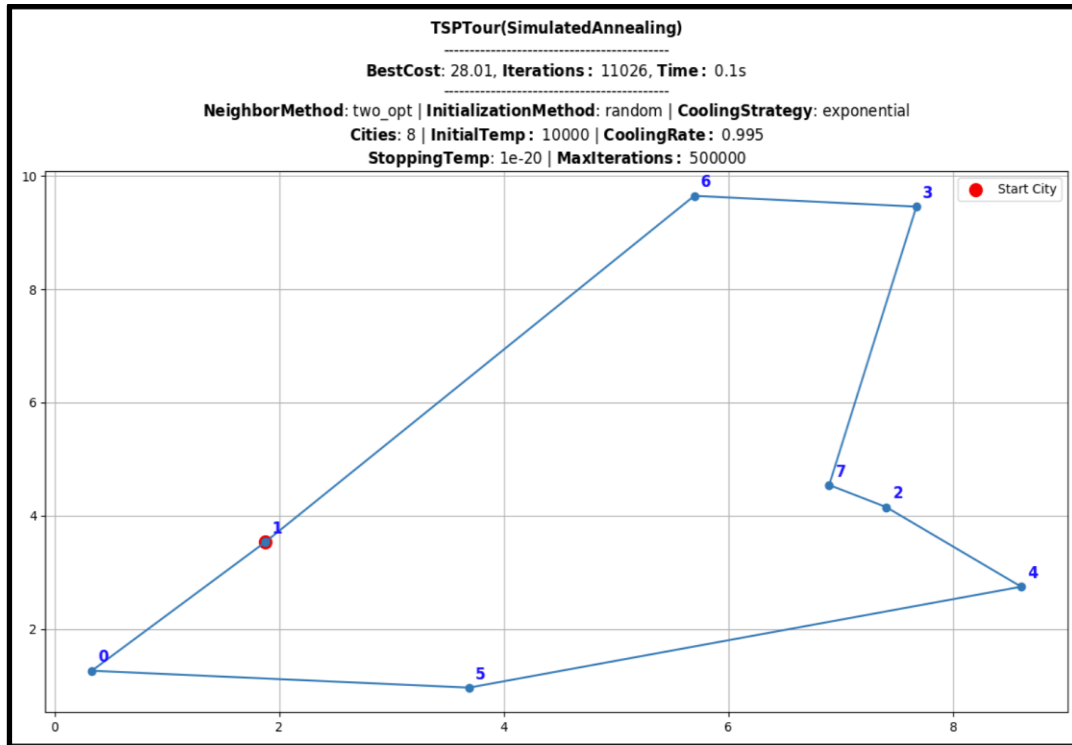
Lúc này, phương thức khởi tạo (**initialization_method**) sẽ tự được điều chỉnh thành 'nn' dù biến nhập vào là gì. Sau khi chi phí lộ trình được tính, hàm sẽ trả về ngay kết quả của tour mà không đi vào vòng lặp Stimulated Annealing.

Sơ đồ khối:

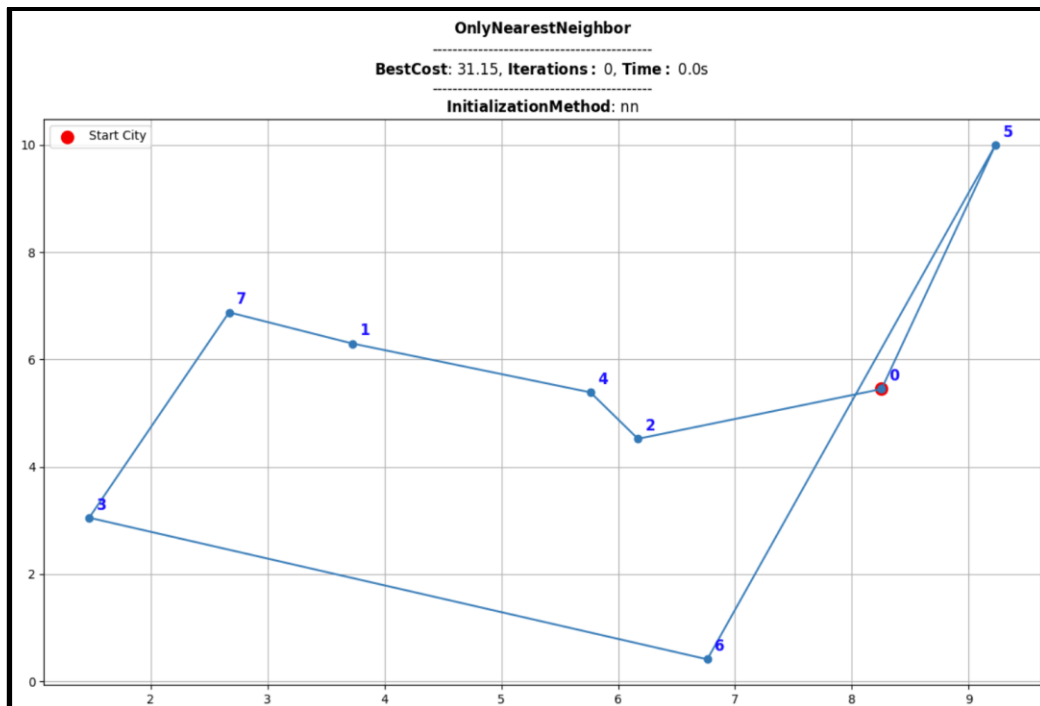


2.2.d. Hàm vẽ đồ thị (Matplotlib)

Các biến được nhập vào trong hàm Simulated Annealing sẽ được hiển thị đầy đủ ở phần trên của đồ thị. Bên dưới và bên trái của đồ thị là trục x, y để biểu diễn tọa độ của các thành phố. Điểm xuất phát sẽ được đánh dấu bằng vòng tròn màu đỏ.



Nếu chỉ chọn in phương án Nearest Neighbor (OnlyNearestNeighbor = True):



2.2.2 Ứng Dụng Giải thuật Simulated Annealing Cho Bài Toán Người du lịch mở rộng

2.2.2.1 Bài toán mở rộng:

Để ứng dụng vào thực tế, bài toán TSP còn cần phải phức tạp hơn nhiều với sự thêm vào các điều kiện, ràng buộc như đường một chiều, đường cắt, thời gian giờ cao điểm, tải trọng,... Trong khuôn khổ bài tiểu luận này, nhóm đã thực hiện mở rộng bài toán TSP cơ bản thành dạng Người du lịch với chi phí bất đối xứng (Asymmetric Travelling Salesman Problem - ATSP) kết hợp với Nhiều người du lịch (mTSP) và lấy bối cảnh như sau: Một kho hàng cần cử một số xe tải đi đến các điểm nhất định để giao hàng cho khách hàng và cuối cùng về kho xuất phát. Bài toán đặt ra yêu cầu tìm đường đi cho mỗi xe sao cho tổng chi phí là nhỏ nhất, biết trên một số đoạn đường, chi phí đi xuôi sẽ khác với chi phí khi đi ngược chiều vì các yếu tố như gió, đường xấu,...

Một số giả định:

- Các xe tải giống nhau về mọi mặt.
- Các xe tải có thể chứa toàn bộ số hàng cần để giao cho khách hàng trên hành trình nó được chia.
- Chi phí khi đi trên cùng đoạn đường và chiều là giống nhau với mọi xe và mọi lần.
- Mỗi xe chỉ đi qua một khách hàng một lần, và mỗi khách hàng chỉ có một xe đi qua.
- Không yêu cầu toàn bộ các xe phải được sử dụng.

Mô hình hóa bài toán:

Dữ liệu đầu vào:

- Danh sách $n + 1$ tuple gồm 02 phần tử, là tọa độ của khách hàng trên bản đồ, với tọa độ đầu tiên quy ước là điểm xuất phát.
- Ma trận $C((n+1) \times (n+1))$ với giá trị $C(i,j)$ là chi phí khi đi từ điểm i tới điểm j ($C(i,j)$ có thể khác $C(j,i)$)
- Một số tham số phục vụ điều chỉnh tính toán của các hàm.

Biểu diễn lời giải:

Mỗi lời giải là một danh sách các chuỗi con, mỗi chuỗi là hành trình của một xe tải (phải bắt đầu và kết thúc tại điểm kho), chẳng hạn như:

[[0, 1, 4, 6, 0], [0, 2, 3, 0], [0, 5, 7, 0]]

Lời giải trên nói rằng:

- Có tổng cộng 3 xe tương ứng 3 cung đường được vạch ra.
- Có tổng cộng 7 khách hàng.
- 0 chính là kho xuất phát.
- Mỗi xe chỉ đi qua một khách hàng một lần, và mỗi khách hàng chỉ có một xe đi qua.

2.2.2.2 Các hàm sử dụng:

Tên hàm	Tham số	Chức năng
generate_customers	num_customers	Tạo danh sách khách hàng với tọa độ ngẫu nhiên

distance	customer1, customer2	Tính khoảng cách Euclid giữa hai khách hàng
create_symmetric_cost_matrix	customers	Tạo ma trận chi phí đối xứng
create_asymmetric_cost_matrix	cost_matrix, asymmetric_percentage, max_factor, use_asymmetric	Tạo ma trận chi phí bất đối xứng
create_problem	custom_customer_map, custom_customer_map_mode, custom_cost_matrix, custom_cost_matrix_mode, num_customers, asymmetric_percentage, use_asymmetric, max_factor, random_seed	Tạo bài toán gồm tọa độ và ma trận chi phí
total_distance	solution, cost_matrix, depot	Tính tổng chi phí của một route (đường đi của một xe) trong lời giải
initial_solution_generation_random	num_customers, num_vehicles	Tạo lời giải ban đầu theo phương pháp ngẫu nhiên
initial_solution_generation_NN	cost_matrix, num_customers, num_vehicles, depot	Tạo lời giải ban đầu theo phương pháp Nearest Neighbor
swap_within_route	solution	Đổi vị trí 2 khách hàng trong cùng một route
two_opt_within_route	solution	Đảo ngược một đoạn ngẫu nhiên trong một route
swap_between_routes	solution	Đổi vị trí 2 khách hàng giữa 2 route
move_customer	solution	Chọn và đưa ngẫu nhiên một khách hàng từ route này sang route khác
generate_positive_transitions	sample_size, cost_matrix, num_customers, num_vehicles, depot, initialization_method, chosen_operator	Tạo tập các cặp chuyển trạng thái trong đó có chi phí của lời giải mới lớn hơn chi phí lời giải hiện tại
compute_initial_temp	transitions, desired_accept_prob, acceptable_error, p, initial_temp, max_iter_temp	Tính nhiệt độ khởi đầu phù hợp
plot_solution	solution, customers, best_cost, iterations, execution_time, num_vehicles, initial_temp, cooling_rate, stopping_temp, max_iter, depot, initialization_method,	Trực quan hóa lời giải cuối

	cooling_strategy	
plot_time_and_cost	compare_cost,avg_cost,best_cost,worst_cost,std_cost, compare_time,avg_time,best_time,worst_time,std_time	Trực quan hóa sự thay đổi của thời gian chạy thuật toán và tổng chi phí các lần lặp
simulated_annealing_mtsp	num_customers, num_vehicles, cost_matrix, initial_temp, cooling_rate, stopping_temp, max_iter, chosen_operator, depot, initialization_method, cooling_strategy	Chạy giải thuật Simulated Annealing chính
run_simulation	custom_customer_map, custom_customer_map_mode, custom_cost_matrix, custom_cost_matrix_mode, num_customers, num_vehicles, asymmetric_percentage, use_asymmetric, max_factor, auto_initial_temp, sample_size, desired_accept_prob, acceptable_error, p, max_iter_temp, initial_temp, cooling_rate, stopping_temp, max_iter, depot, initialization_method, cooling_strategy, chosen_operator, random_seed	Chạy toàn bộ chương trình

Bảng xx: Tóm tắt các hàm sử dụng

Các tham số sử dụng

Tên tham số	Ý nghĩa
custom_customer_map	Danh sách các tọa độ của khách hàng do người dùng tự nhập
custom_customer_map_mode	Chế độ tự nhập tọa độ (True = On)
custom_cost_matrix	Mã trận chi phí do người dùng tự nhập
custom_cost_matrix_mode	Chế độ tự nhập ma trận chi phí (True = On)
num_customers	Số khách hàng (không bao gồm depot)
num_vehicles	Số xe
asymmetric_percentage	Tỷ lệ các đoạn đường có chi phí khác nhau ở hai chiều
use_asymmetric	Chế độ bài toán có ma trận chi phí bất đối xứng (True = On)
max_factor	Tỷ lệ biến đổi của chi phí bất đối xứng so với chi phí gốc

auto_initial_temp	Chế độ tự động tinh chỉnh nhiệt độ khởi đầu (True = On)
sample_size	Cỡ mẫu của tập các cặp chuyển trạng thái
desired_accept_prob	Tỷ lệ chấp nhận lời giải xấu khởi đầu mong muốn
acceptable_error	Sai số chấp nhận được giữa tỷ lệ chấp nhận mong muốn và tỷ lệ chấp nhận hiện tại
p	tham số của công thức tinh chỉnh nhiệt độ khởi đầu (mặc định là 1 để nhiệt độ thay đổi theo hàm log)
max_iter_temp	Số lần lặp tối đa để ngưng thuật toán tinh chỉnh nhiệt độ khởi đầu nếu không hội tụ sớm
initial_temp	Nhiệt độ khởi đầu
cooling_rate	Tốc độ làm lạnh
stopping_temp	Nhiệt độ ngừng (giải thuật ngưng khi nhiệt độ \leq nhiệt độ ngừng)
max_iter	Số vòng lặp tối đa để ngưng giải thuật Simulated Annealing nếu chi phí không hội tụ sớm
depot	điểm xuất phát (mặc định có chỉ số index là 0 trong chuỗi các điểm cần đi)
initialization_method	Lựa chọn phương pháp tạo lời giải khởi đầu
cooling_strategy	Lựa chọn phương pháp làm lạnh
chosen_operator	Lựa chọn phương pháp tạo hàng xóm
random_seed	Số để đánh dấu và tái sử dụng các kết quả của hàm random
customers	Danh sách tọa độ các điểm cần đi (giá trị có index = 0 là depot, còn lại là khách hàng)
cost_matrix	Mã trận chi phí giữa các điểm cần đi (bao gồm cả depot)
solution	Lời giải cho bài toán
compare_cost	Tập giá trị chi phí mỗi lần chạy giải thuật
avg_cost	Chi phí trung bình sau 100 lần chạy giải thuật
best_cost	Chi phí thấp nhất sau 100 lần chạy giải thuật

worst_cost	Chi phí cao nhất sau 100 lần chạy giải thuật
std_cost	Độ lệch chuẩn của 100 lần chạy giải thuật
compare_time	Tập giá trị thời gian mỗi lần chạy giải thuật
avg_time	Thời gian trung bình sau 100 lần chạy giải thuật
best_time	Thời gian ít nhất sau 100 lần chạy giải thuật
worst_time	Thời gian nhiều nhất sau 100 lần chạy giải thuật
std_time	Độ lệch chuẩn thời gian chạy của 100 lần chạy giải thuật

Bảng xx: Tóm tắt các tham số sử dụng

Mô tả các hàm chính:

- **create_problem(custom_customer_map, custom_customer_map_mode, ..., random_seed)**: được dùng để khởi tạo danh sách tọa độ và ma trận chi phí của các khách hàng.
- Nếu custom_customer_map_mode và/hoặc custom_cost_matrix_mode có giá trị là True, sẽ tương ứng dùng tọa độ và ma trận chi phí do người dùng tự nhập vào.
- Nếu không, hàm mặc định sẽ gọi generate_customers và create_symmetric_cost_matrix để tạo tọa độ và ma trận chi phí.
- Trường hợp muốn biến bài toán trở thành ATSP, người dùng có thể thực hiện bằng cách đặt giá trị của use_asymmetric là True, và đặt max_factor để kiểm soát độ biến động của chi phí theo ý.
- Người dùng có thể đặt seed cho thuật toán ngẫu nhiên bằng random_seed để giữ một kết quả danh sách tọa độ và ma trận chi phí dùng trong nhiều lần lặp tiếp theo nhằm mục đích so sánh.
- **initial_solution_generation_random(num_customers, num_vehicles)**: được dùng để khởi tạo lời giải ban đầu theo phương pháp ngẫu nhiên bằng cách: tạo một danh sách các chỉ số, tương ứng với khách hàng, xáo trộn, sau đó chia lần lượt vào các route của các xe.
- **initial_solution_generation_NN(cost_matrix, num_customers, num_vehicles, depot=0)**: được dùng để khởi tạo lời giải ban đầu theo phương pháp Nearest Neighbor bằng cách: tạo một set các chỉ số đại diện cho các khách hàng. Duyệt qua từng xe, chọn khách hàng gần nhất với xe đó, thêm vào route của xe đó và tiếp tục qua xe khác cho đến khi đã đi qua toàn bộ các khách hàng.
- **swap_within_route(solution)**: Đổi vị trí hai khách hàng trong cùng một route bằng cách: chọn ngẫu nhiên một route không rỗng, tiếp tục chọn ngẫu nhiên hai điểm trong route, đổi vị trí cho nhau.
- **two_opt_within_route(solution)**: Đảo ngược một đoạn ngẫu nhiên trong cùng một route bằng cách: chọn một route ngẫu nhiên không rỗng, chọn hai điểm ngẫu nhiên trên route và đảo đoạn giữa hai điểm đó.

- **swap_between_routes(solution)**: Đổi hai khách hàng giữa hai route bằng cách: chọn ngẫu nhiên hai route không rỗng, chọn ngẫu nhiên một điểm trên mỗi route, đổi chỗ chúng với nhau.
- **move_customer(solution)**: Đưa ngẫu nhiên khách hàng từ route này sang route khác bằng cách: chọn một khách hàng ngẫu nhiên từ một route ngẫu nhiên không rỗng, chọn một vị trí ngẫu nhiên của một route không rỗng ngẫu nhiên khác và chèn điểm đó vào vị trí đã chọn.
- **generate_positive_transitions(sample_size, cost_matrix, num_customers, num_vehicles, depot, initialization_method, chosen_operator)**: tạo ra một số lượng cụ thể các cặp lời giải đầu và hàng xóm của nó sao cho chi phí lời giải hàng xóm lớn hơn lời giải khởi đầu (positive transitions), nhằm phục vụ cho việc tính toán của phương pháp tự chọn nhiệt độ khởi đầu, bằng cách: sử dụng các hàm tạo lời giải khởi đầu và tạo hàng xóm mà sẽ được sử dụng trong thuật toán Simulated Annealing chính để tạo ra các cặp chuyển trạng thái. Sau đó so sánh chi phí của chúng và thêm vào danh sách transitions các cặp chuyển trạng thái có chi phí tăng lên.
- **compute_initial_temp(transitions, desired_accept_prob, acceptable_error, p, initial_temp, max_iter_temp)**: dùng tính chỉnh nhiệt độ khởi tạo cho phù hợp với thước đo của ma trận chi phí của bài toán bằng cách: dùng các cặp giá trị chi phí trong transitions truyền vào tính xác suất chấp nhận theo công thức:

$$\hat{\chi}(T_n) = \frac{\sum_{t \in S} \exp(-\frac{E_{\max_t}}{T_n})}{\sum_{t \in S} \exp(-\frac{E_{\min_t}}{T_n})}$$

- Nếu xác suất chấp nhận này nằm trong khoảng **desired_accept_prob** \pm **acceptable_error** thì trả về T là initial_temp cho các bước tiếp theo. Ngược lại thì cập nhật T theo công thức:

$$- T_{n+1} = T_n \left(\frac{\ln(\hat{\chi}(T_n))}{\ln(\chi_0)} \right)^{\frac{1}{p}}$$

và tăng số vòng đã lặp lên 1 cho tới khi đạt max_iter_temp thì ngưng và trả về T hiện tại.

- **simulated_annealing_mtsp(num_customers, num_vehicles, ..., cooling_strategy)**: hàm chạy thuật toán Simulated Annealing chính. Thực hiện tương tự như hàm giải bài toán TSP cơ bản đã đề cập.
- **run_simulation(custom_customer_map, custom_customer_map_mode, ..., random_seed)**: hàm thực thi toàn bộ quy trình: đầu tiên sẽ gọi hàm tạo bài toán gồm danh sách tọa độ các khách hàng và ma trận chi phí. Trước khi đến bước tiếp theo, sẽ cài lại seed ngẫu nhiên để tránh hàm chạy giải thuật dùng chung seed với hàm
- **create_problem**. Sau đó, nếu lựa chọn auto_initial_temp được truyền vào là True, sẽ thực hiện tính chỉnh nhiệt độ khởi đầu. Tiếp theo, hàm cho lặp một số lượng nhất định (ở đây cài mặc định là 100) lần giải thuật Simulated Annealing trên bài toán đã tạo, và tính các chỉ số thống kê cơ bản như trung bình, tối đa, tối thiểu, và độ lệch của thời gian chạy và chi phí của các lời giải. Cuối cùng là gọi hàm plot_solution và plot_time_and_cost để trực quan hóa các chỉ số thống kê trên.

CHƯƠNG 3. CÁC KẾT QUẢ THỰC NGHIỆM

3.1. Các Tình Huống Trong TSP cơ bản

Trong phần sau đây, ta sẽ so sánh ảnh hưởng của các lựa chọn khác nhau lên kết quả thuật toán. Cụ thể, ta sẽ cho thuật toán chạy 100 lần rồi **tính chi phí trung bình** và **thời gian trung bình** cho 100 lần đó. Ta **tự nhập tọa độ** của thành phố và chọn **Initialization Method** là NearestNeighbor để đầu vào mỗi lần chạy là **cố định**. Ta cũng sẽ vẽ đồ thị biểu diễn kết quả của mỗi lần chạy bằng Matplotlib.

```
all_cost = []
all_time = []

for i in range(100):

    start_time = time.time() #Đếm thời gian chạy

    best_tour, best_cost, cities, iterations, neighbor_name, initialization_name = simulated_annealing(
        num_cities=num_cities,
        initial_temp=initial_temp,
        cooling_rate=cooling_rate,
        stopping_temp=stopping_temp,
        max_iter=max_iter,
        neighbor_method=neighbor_method,
        initialization_method=initialization_method,
        cooling_strategy=cooling_strategy,
        OnlyNearestNeighbor=OnlyNearestNeighbor,
        CustomCity = CustomCity
    )

    end_time = time.time() #Đếm thời gian chạy
    execution_time = round(end_time - start_time, 2)

    print("Best cost:", round(best_cost, 2))
    print("Time:", round(execution_time, 2))
    print("Iteration:", iterations)
    print("-----")

    all_cost.append(best_cost)
    all_time.append(execution_time)
```

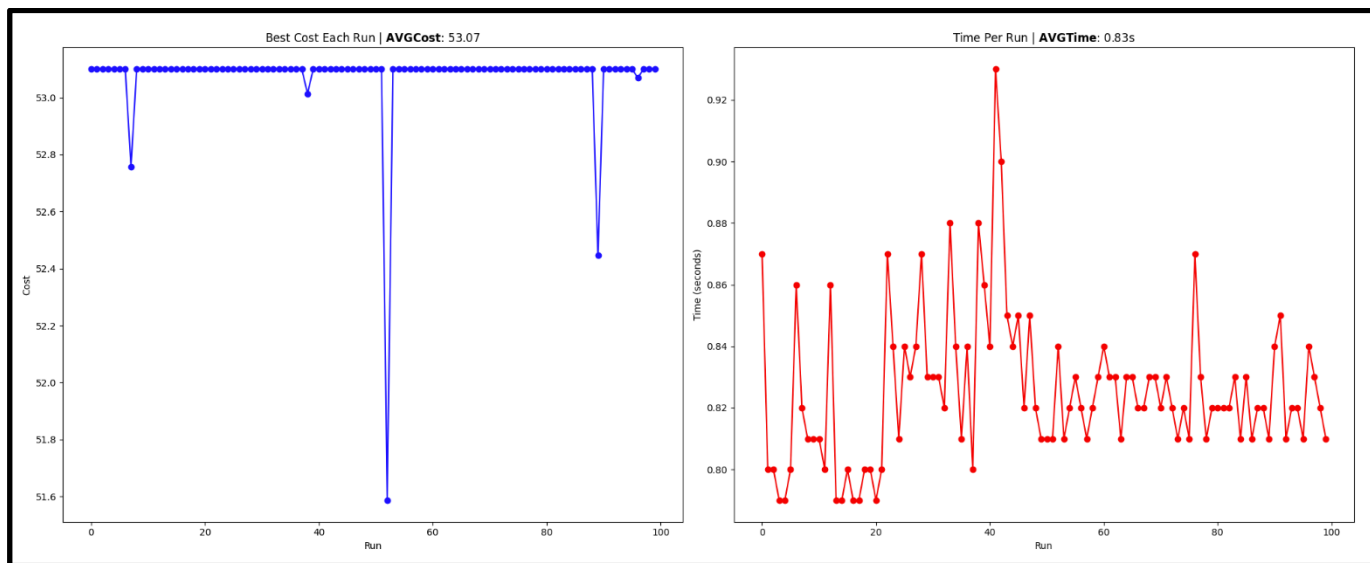
3.1.a. So sánh two_opt vs three_opt vs random_swap vs NearestNeighborOnly

Cho giá trị của các biến:

- **CustomCity:** [(8.5, 8.42), (7.07, 9.29), (7.82, 7.0), (4.44, 6.65), (3.03, 8.68), (8.49, 7.95), (7.96, 7.89), (6.51, 2.08), (6.04, 6.96), (8.56, 1.42), (2.39, 7.11), (3.91, 8.89), (3.27, 3.79), (3.47, 5.7), (8.42, 2.09), (7.06, 4.85), (1.36, 9.64), (4.61, 4.15), (1.74, 5.86), (4.33, 5.03), (6.32, 2.47), (8.15, 7.69), (9.98, 4.02), (3.66, 3.48), (4.26, 1.24)] (**25 thành phố**)
- **initial_temp:** 1000.
- **cooling_rate:** 0.995.
- **stopping_temp:** 1e-200.
- **max_iter:** 300000.
- **initialization_method:** nn.
- **cooling_strategy:** exponential.
- **OnlyNearestNeighbor:** False

Với 25 thành phố, việc duyệt toàn bộ các trường hợp $((n-1)!/2)$ là không khả thi. Tuy nhiên, sử dụng các công cụ tối ưu như **Concorde** (dùng kỹ thuật nhánh cắt - branch-and-cut), ta có thể tìm được **lộ trình tối ưu**: [0, 5, 6, 21, 1, 22, 15, 14, 9, 20, 7, 23, 17, 19, 13, 3, 18, 10, 4, 11, 16, 12, 24, 8, 2, 0] có chi phí là **36.62**

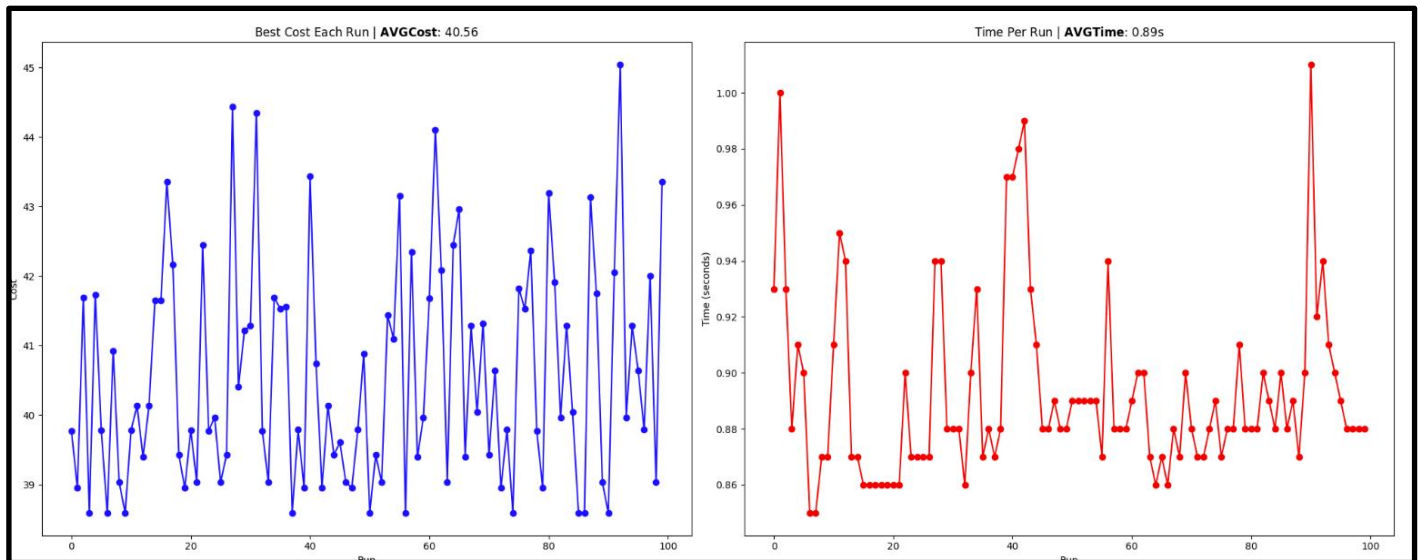
Random_swap:



Trung bình chi phí hành trình: 53.07. So với chi phí tối ưu (36.62) thì đắt hơn khoảng 45%.

Trung bình thời gian chạy: 0.83s

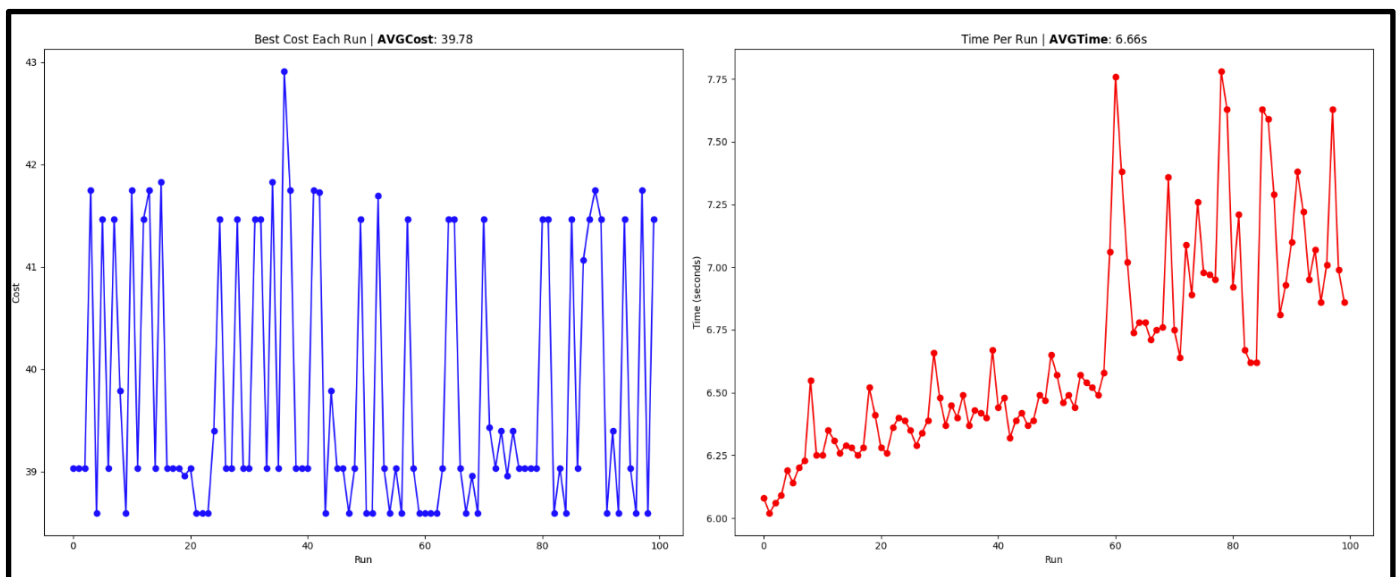
Two_opt:



Trung bình chi phí hành trình: 40.56. So với chi phí tối ưu (36.62) thì đắt hơn khoảng 10%.

Trung bình thời gian chạy: 0.89s

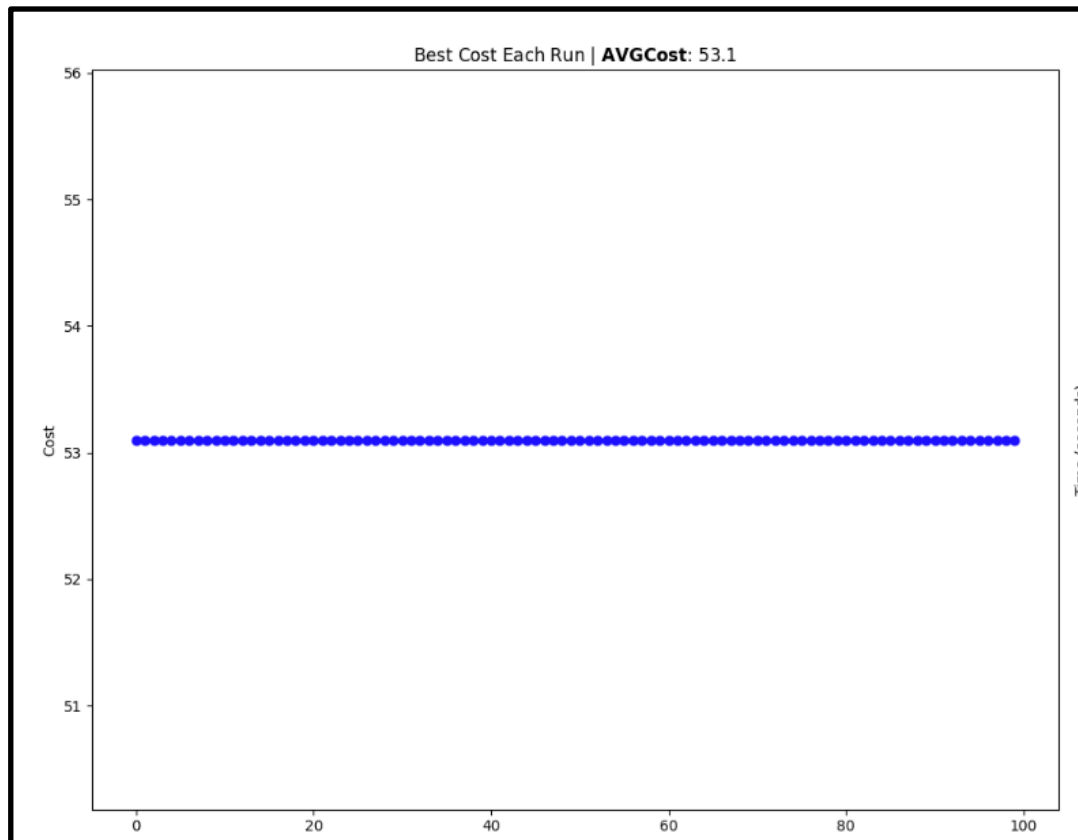
Three_opt:



Trung bình chi phí hành trình: 39.78. So với chi phí tối ưu (36.62) thì đắt hơn khoảng 6.7%.

Trung bình thời gian chạy: 6.66s

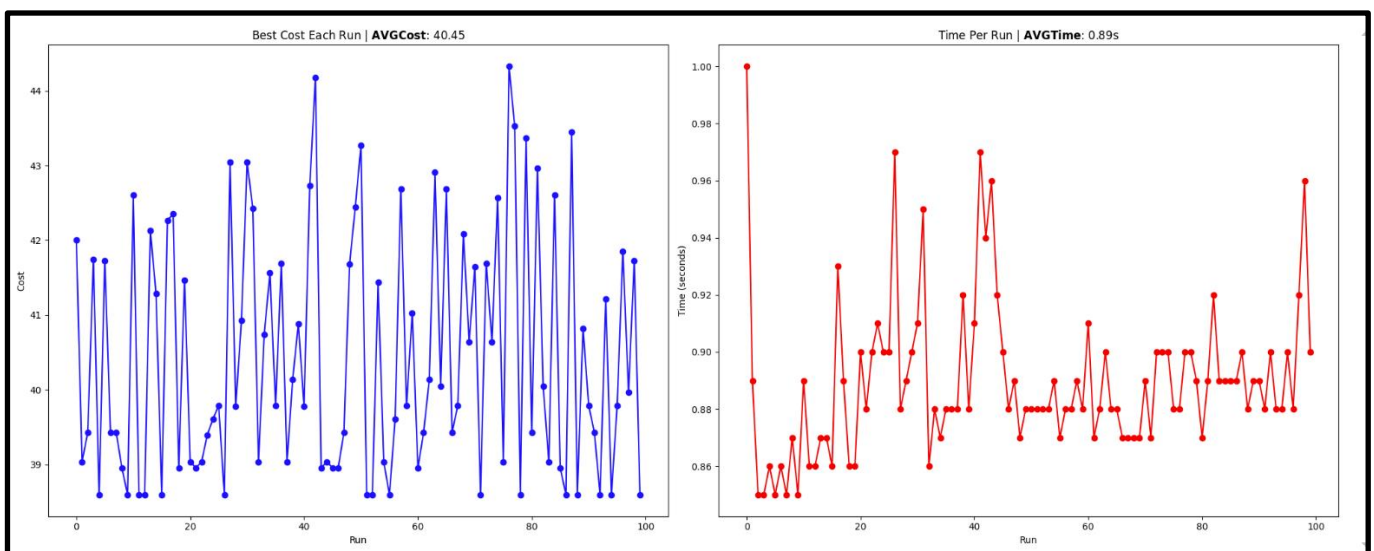
Nearest Neighbor Only:



Chỉ 1 chi phí hành trình: 53.1

3.1.b. Nếu phương thức khởi tạo tour (Initialization Method) là random thì sao?

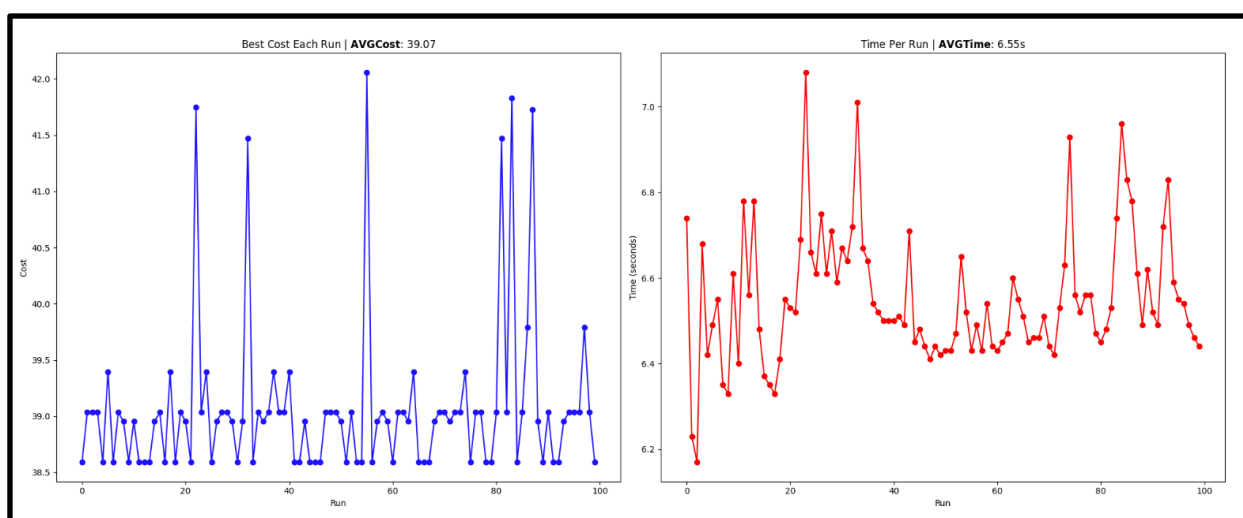
Two-opt:



Trung bình chi phí hành trình: 40.45

Trung bình thời gian chạy: 0.89s

Three-opt:



Trung bình chi phí hành trình: 39.07

Trung bình thời gian chạy: 6.55s

3.2. Phân Tích và Đánh Giá TSP cơ bản

Dựa vào kết quả trên, ta rút ra được:

Phương pháp **Random Swap** có chi phí hành trình trung bình cao nhất trong ba phương pháp Neighbor. Điều này là do phương pháp này chỉ hoán đổi ngẫu nhiên hai thành phố trong hành trình, dẫn đến việc không khai thác tối ưu cấu trúc hành trình. Tuy nhiên, phương pháp này nhanh nhất do độ phức tạp thấp. So với chi phí tối ưu (36.62) thì đắt hơn khoảng 45%.

Two-Opt cải thiện đáng kể chi phí hành trình so với Random Swap nhờ khả năng đảo ngược các đoạn trong hành trình để giảm khoảng cách. Thời gian chạy tăng nhẹ do phải tính lại khoảng cách của các hành trình sau mỗi lần đảo. So với chi phí tối ưu (36.62) thì đắt hơn khoảng 10%.

Three-Opt cho kết quả tốt nhất về chi phí hành trình, chứng tỏ hiệu quả của phương pháp này trong việc tối ưu hóa hành trình. Tuy nhiên, cái giá phải trả là thời gian chạy cao nhất trong ba phương pháp do độ phức tạp cao hơn, vì Three-Opt thử nhiều cách hoán đổi các đoạn của hành trình để tối ưu. So với chi phí tối ưu (36.62) thì đắt hơn khoảng 6.7%.

Nearest Neighbor vs Stimulated Annealing: Như có thể thấy, trừ Random Swap ra thì 2 phương thức còn lại của Stimulated Annealing đều cho ra hành trình có chi phí thấp hơn đáng kể so với Nearest Neighbor (chỉ đến điểm gần nhất ở mỗi thành phố).

Vậy Stimulated Annealing là một phương pháp giải bài toán TSP khá hiệu quả.

Initialization Method = Random vs NN:

Two-Opt: Chi phí gần như không thay đổi (40.45 vs 40.56) và thời gian chạy bằng nhau.

Three-Opt: Chi phí và thời gian không có sự khác biệt không đáng kể (39.07 vs 39.78)

=> Vậy phương thức khởi tạo không có ảnh hưởng đáng kể đến kết quả của thuật toán.

3.3 Các tình huống trong bài toán TSP mở rộng

Trong phần này, ta sẽ tiếp tục so sánh ảnh hưởng của các lựa chọn phương pháp khác nhau lên kết quả chạy giải thuật tương tự với phần trước, cho bài toán TSP mở rộng. Ta cũng sẽ cho thuật toán của mỗi tổ hợp các phương pháp (2 phương pháp tạo lời giải đầu, 5 phương pháp tạo hàng xóm, và lựa chọn tự động tinh chỉnh nhiệt độ) chạy 100 lần rồi tính các thống kê (chi phí/ thời gian trung bình, tốt nhất, tệ nhất, và độ lệch chuẩn của chi phí và thời gian) cho 100 lần đó. Đồng thời ta sẽ thực hiện chạy giải thuật trên các kích cỡ tập khách hàng khác nhau, và đặt seed là 10, 15, 50, 100 lần lượt cho các kích cỡ 15, 50, 100 để đảm bảo các tọa độ và ma trận chi phí là như nhau ở mỗi cấp. Các kết quả sẽ được biểu diễn trực quan trên biểu đồ nhờ thư viện Matplotlib.

Các điều kiện khác ngoài các phương pháp đang so sánh phải được giữ nguyên qua mọi lần lặp, ta sẽ sử dụng cùng các tham số chung như sau:

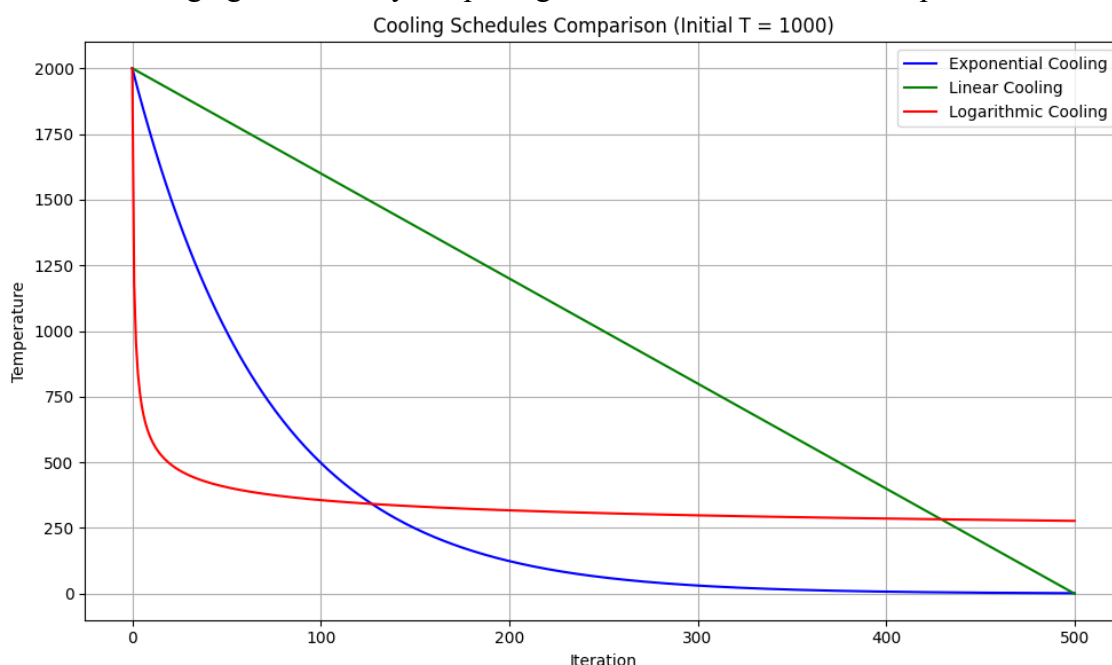
```
custom_customer_map = [],  
custom_customer_map_mode = False,  
custom_cost_matrix=[],  
custom_cost_matrix_mode=False,  
asymmetric_percentage = 0.25,  
use_asymmetric=True,  
max_factor = 2.0,  
sample_size = 100,  
desired_accept_prob=0.8,  
acceptable_error=0.01,  
p=1.0,  
max_iter_temp = 100,  
initial_temp=1000,  
cooling_rate=0.995,  
stopping_temp=1e-50,  
max_iter=100000,  
depot=0,  
cooling_strategy='exponential'
```

```
custom_customer_map_mode = False, custom_cost_matrix_mode=False,  
use_asymmetric=True: thuật toán sẽ sử dụng danh sách tọa độ và ma trận chi phí bất  
đối xứng hoàn toàn ngẫu nhiên tạo bởi hàm generate_customers() theo các thông số  
trên (seed ngẫu nhiên sẽ được lưu lại và sử dụng cho các lần chạy thử cùng kích cỡ  
tập khách hàng).
```

num_vehicles: số lượng xe (k) có thể đi từ depot là một tham số quan trọng, nó ảnh hưởng đến bản chất bài toán (để đảm bảo đây là bài toán mTSP thì: $k \in \mathbb{Z}$, $k > 1$), và

mức độ phức tạp (tăng số lượng xe trong bài toán mTSP không chỉ làm gia tăng số lượng tổ hợp phân chia khách hàng mà còn làm tăng số lượng hoán vị bên trong mỗi vòng tour. Không gian tìm kiếm vì vậy tăng theo tốc độ nhanh hơn cấp số nhân, khiến bài toán trở nên phức tạp hơn đáng kể về mặt tính toán). Do đó, nhóm sẽ lựa chọn một số lượng xe sao cho đảm bảo rằng mỗi xe trung bình phụ trách ko quá 15 khách.

cooling_strategy='exponential': Việc lựa chọn tham số cooling_rate phù hợp phụ thuộc chặt chẽ vào bản chất của từng chiến thuật làm lạnh, do mỗi chiến thuật có cơ chế giảm nhiệt độ khác nhau (hình xx). Điều này đòi hỏi một quá trình nghiên cứu và thử nghiệm chuyên sâu, vượt ra ngoài phạm vi của bài tiểu luận hiện tại. Do đó, nhóm quyết định chỉ sử dụng một chiến thuật làm lạnh duy nhất. Sau khi xem xét đặc điểm của các chiến thuật phổ biến, nhóm nhận thấy rằng chiến thuật làm lạnh theo cấp số nhân là lựa chọn hợp lý nhất. Cụ thể, chiến thuật logarit có xu hướng giảm nhiệt quá nhanh trong những vòng lặp đầu tiên, dễ dẫn đến hội tụ sớm, trong khi chiến thuật cấp số cộng lại làm giảm nhiệt độ quá chậm và đều, khó thích nghi với các không gian nghiệm lớn. Chiến thuật cấp số nhân, với tốc độ giảm nhiệt ổn định và linh hoạt, được đánh giá là phù hợp hơn với các bài toán có quy mô khác nhau. Vì vậy, toàn bộ các thuật toán trong nghiên cứu này sẽ áp dụng chiến thuật làm lạnh theo cấp số nhân.



hình x.x

3.3.1 Tập 15 khách hàng (num_customers = 15, num_vehicles = 2, seed = 15)

Kết quả chạy so sánh các thuật toán có thể được tóm tắt theo bảng sau:

Phương pháp tạo hàng xóm	Phương pháp tạo lời giải khởi đầu	auto_initial_temp = False	auto_initial_temp = True
swap_within_route	Random	Best cost: 448889.42 Worst cost: 663331.23 Avg cost: 574900.05 Std cost: 41319.57	Best cost: 475042.19 Worst cost: 644629.03 Avg cost: 570875.33 Std cost: 32623.04

		Best time: 0.1467 s Worst time: 0.1858 s Avg time: 0.1641 s Std time: 0.0087 s	Best time: 0.1511 s Worst time: 0.3772 s Avg time: 0.177 s Std time: 0.0233 s
	Nearest Neighbor	Best cost: 436775.11 Worst cost: 470721.0 Avg cost: 470381.54 Std cost: 3394.59 Best time: 0.1529 s Worst time: 0.8036 s Avg time: 0.2062 s Std time: 0.1241 s	Best cost: 403417.28 Worst cost: 470620.29 Avg cost: 424385.05 Std cost: 18974.31 Best time: 0.154 s Worst time: 0.1899 s Avg time: 0.1717 s Std time: 0.0084 s
two_opt_within_route	Random	Best cost: 463129.52 Worst cost: 619654.26 Avg cost: 560563.9 Std cost: 34013.26 Best time: 0.1678 s Worst time: 0.7897 s Avg time: 0.309 s Std time: 0.2112 s	Best cost: 443411.06 Worst cost: 620333.48 Avg cost: 558766.06 Std cost: 34422.24 Best time: 0.1717 s Worst time: 0.2 s Avg time: 0.1835 s Std time: 0.0049 s
	Nearest Neighbor	Best cost: 413232.89 Worst cost: 463476.52 Avg cost: 442711.59 Std cost: 14569.35 Best time: 0.1647 s Worst time: 0.1985 s Avg time: 0.1761 s Std time: 0.0055 s	Best cost: 403417.28 Worst cost: 453995.28 Avg cost: 417659.2 Std cost: 15352.5 Best time: 0.1706 s Worst time: 0.2113 s Avg time: 0.185 s Std time: 0.0061 s
swap_between_routes	Random	Best cost: 440299.19 Worst cost: 696874.75 Avg cost: 542596.69 Std cost: 57341.92 Best time: 0.1499 s	Best cost: 426477.17 Worst cost: 580473.69 Avg cost: 508044.31 Std cost: 34375.05 Best time: 0.1558 s

		Worst time: 0.1821 s Avg time: 0.162 s Std time: 0.0059 s	Worst time: 0.1818 s Avg time: 0.1691 s Std time: 0.0047 s
	Nearest Neighbor	Best cost: 475460.4 Worst cost: 475460.4 Avg cost: 475460.4 Std cost: 0.0 Best time: 0.1544 s Worst time: 0.1841 s Avg time: 0.1633 s Std time: 0.005 s	Best cost: 405434.81 Worst cost: 484802.99 Avg cost: 477248.66 Std cost: 16923.4 Best time: 0.1599 s Worst time: 0.1911 s Avg time: 0.1709 s Std time: 0.0044 s
move_customer	Random	Best cost: 403417.28 Worst cost: 656767.69 Avg cost: 497976.95 Std cost: 49916.69 Best time: 0.1491 s Worst time: 0.2471 s Avg time: 0.1618 s Std time: 0.0142 s	Best cost: 393178.91 Worst cost: 637898.18 Avg cost: 517352.02 Std cost: 53909.34 Best time: 0.1496 s Worst time: 0.2174 s Avg time: 0.1601 s Std time: 0.0071 s
	Nearest Neighbor	Best cost: 425355.15 Worst cost: 466464.23 Avg cost: 444446.87 Std cost: 10190.57 Best time: 0.144 s Worst time: 0.1829 s Avg time: 0.157 s Std time: 0.0065 s	Best cost: 402008.56 Worst cost: 484802.99 Avg cost: 475723.77 Std cost: 18970.05 Best time: 0.1527 s Worst time: 0.1694 s Avg time: 0.1597 s Std time: 0.0041 s
random (mỗi lần lặp chọn ngẫu nhiên 1 trong 4 phương pháp trên)	Random	Best cost: 324507.29 Worst cost: 487330.54 Avg cost: 401777.0 Std cost: 29903.18 Best time: 0.1556 s Worst time: 0.2364 s Avg time: 0.1818 s	Best cost: 324507.29 Worst cost: 426714.01 Avg cost: 366364.51 Std cost: 28433.21 Best time: 0.1613 s Worst time: 0.7908 s Avg time: 0.2066 s

		Std time: 0.0131 s	Std time: 0.1154 s
	Nearest Neighbor	Best cost: 397852.93 Worst cost: 458221.76 Avg cost: 416934.53 Std cost: 12952.62 Best time: 0.1726 s Worst time: 0.7774 s Avg time: 0.3775 s Std time: 0.2137 s	Best cost: 324507.29 Worst cost: 433765.63 Avg cost: 372026.13 Std cost: 26647.28 Best time: 0.1608 s Worst time: 0.2178 s Avg time: 0.1774 s Std time: 0.0119 s

bảng x.x

3.3.1 Tập 50 khách hàng (num_customers = 50, num_vehicles = 4, seed = 50)

Phương pháp tạo hàng xóm	Phương pháp tạo lời giải khởi đầu	auto_initial_tem p = False	auto_initial_tem p = True
swap_within_route	Random	Best cost: 1344489.54 Worst cost: 1753042.13 Avg cost: 1561951.76 Std cost: 87303.0 Best time: 0.343 s Worst time: 0.395 s Avg time: 0.3563 s Std time: 0.0088 s	Best cost: 1325275.69 Worst cost: 1675902.13 Avg cost: 1506628.92 Std cost: 73910.07 Best time: 0.355 s Worst time: 0.4401 s Avg time: 0.3699 s Std time: 0.0101 s
	Nearest Neighbor	Best cost: 972875.73 Worst cost: 981613.06 Avg cost: 975037.39 Std cost: 3758.23	Best cost: 954778.59 Worst cost: 1018362.37 Avg cost: 1013036.22 Std cost: 12580.87

		Best time: 0.3403 s Worst time: 0.3797 s Avg time: 0.3555 s Std time: 0.0072 s	Best time: 0.358 s Worst time: 0.4396 s Avg time: 0.3715 s Std time: 0.0112 s
two_opt_within_route	Random	Best cost: 1225683.6 Worst cost: 1508334.52 Avg cost: 1381930.71 Std cost: 65504.44 Best time: 0.3759 s Worst time: 1.2857 s Avg time: 1.0039 s Std time: 0.2394 s	Best cost: 1247164.79 Worst cost: 1559798.02 Avg cost: 1371323.38 Std cost: 68425.21 Best time: 0.383 s Worst time: 1.4742 s Avg time: 0.9955 s Std time: 0.3084 s
	Nearest Neighbor	Best cost: 945196.66 Worst cost: 957248.57 Avg cost: 952237.63 Std cost: 4532.51 Best time: 1.0217 s Worst time: 1.2447 s Avg time: 1.0897 s Std time: 0.0428 s	Best cost: 933093.78 Worst cost: 1018362.37 Avg cost: 969158.15 Std cost: 24423.15 Best time: 0.3797 s Worst time: 1.6403 s Avg time: 0.4592 s Std time: 0.1935 s

swap_between_routes	Random	Best cost: 946396.51 Worst cost: 1356139.86 Avg cost: 1129699.45 Std cost: 79161.24 Best time: 0.3359 s Worst time: 0.3684 s Avg time: 0.349 s Std time: 0.0075 s	Best cost: 969896.29 Worst cost: 1305230.03 Avg cost: 1116146.53 Std cost: 65644.75 Best time: 0.3511 s Worst time: 1.4852 s Avg time: 0.4707 s Std time: 0.2707 s
	Nearest Neighbor	Best cost: 885542.38 Worst cost: 964305.67 Avg cost: 935337.89 Std cost: 20051.62 Best time: 0.3381 s Worst time: 1.4824 s Avg time: 0.4993 s Std time: 0.3111 s	Best cost: 959137.16 Worst cost: 1018362.37 Avg cost: 1016266.52 Std cost: 8720.73 Best time: 0.3501 s Worst time: 1.2296 s Avg time: 0.4066 s Std time: 0.1548 s
move_customer	Random	Best cost: 776092.55 Worst cost: 1050556.04 Avg cost: 892847.16 Std cost: 57075.54 Best time:	Best cost: 779399.06 Worst cost: 1311006.19 Avg cost: 932011.08 Std cost: 94592.15 Best time:

		0.3317 s Worst time: 0.3767 s Avg time: 0.3448 s Std time: 0.0093 s	0.3334 s Worst time: 1.391 s Avg time: 0.4698 s Std time: 0.2773 s
	Nearest Neighbor	Best cost: 748242.89 Worst cost: 891702.77 Avg cost: 853126.22 Std cost: 25598.37 Best time: 0.3296 s Worst time: 0.4654 s Avg time: 0.3622 s Std time: 0.0291 s	Best cost: 725618.81 Worst cost: 1018362.37 Avg cost: 921161.49 Std cost: 80660.72 Best time: 0.3347 s Worst time: 0.4616 s Avg time: 0.365 s Std time: 0.0201 s
random (mỗi lần lập chọn ngẫu nhiên 1 trong 4 phương pháp trên)	Random	Best cost: 697542.91 Worst cost: 936009.67 Avg cost: 825910.26 Std cost: 50879.93 Best time: 0.3499 s Worst time: 1.3286 s Avg time: 0.6777 s Std time: 0.3638 s	Best cost: 719414.95 Worst cost: 975111.24 Avg cost: 806166.21 Std cost: 50735.13 Best time: 0.3646 s Worst time: 0.3941 s Avg time: 0.3753 s Std time: 0.0063 s
	Nearest Neighbor	Best cost: 775716.45	Best cost: 692133.09

		Worst cost: 875948.6 Avg cost: 820857.37 Std cost: 18951.2 Best time: 0.3549 s Worst time: 1.3434 s Avg time: 0.587 s Std time: 0.3418 s	Worst cost: 985225.91 Avg cost: 785855.04 Std cost: 50641.77 Best time: 0.3649 s Worst time: 0.4101 s Avg time: 0.3799 s Std time: 0.0079 s
--	--	--	--

bảng x.x

3.3.1 Tập 100 khách hàng (num_customers = 100, num_vehicles = 7, seed = 100)

Phương pháp tạo hàng xóm	Phương pháp tạo lời giải khởi đầu	auto_initial_tem p = False	auto_initial_tem p = True
swap_within_route	Random	Best cost: 2528901.9 Worst cost: 3144699.27 Avg cost: 2840393.86 Std cost: 130183.35 Best time: 0.6058 s Worst time: 0.6763 s Avg time: 0.6249 s Std time: 0.0117 s	Best cost: 2430499.01 Worst cost: 3112356.89 Avg cost: 2820112.72 Std cost: 128922.69 Best time: 0.6199 s Worst time: 2.1749 s Avg time: 0.7244 s Std time: 0.2818 s
	Nearest Neighbor	Best cost: 1592559.33 Worst cost: 1615089.23 Avg cost:	Best cost: 1622352.74 Worst cost: 1626467.13 Avg cost:

		1603923.06 Std cost: 5049.93 Best time: 0.6042 s Worst time: 2.0746 s Avg time: 0.7522 s Std time: 0.3544 s	1626425.27 Std cost: 411.43 Best time: 0.6346 s Worst time: 2.0112 s Avg time: 0.6955 s Std time: 0.1938 s
two_opt_within _route	Random	Best cost: 2243794.18 Worst cost: 2834918.41 Avg cost: 2437742.08 Std cost: 97637.89 Best time: 0.6298 s Worst time: 1.988 s Avg time: 0.8627 s Std time: 0.4427 s	Best cost: 2187978.65 Worst cost: 2643711.04 Avg cost: 2422815.72 Std cost: 98531.69 Best time: 0.6483 s Worst time: 0.9605 s Avg time: 0.6854 s Std time: 0.0452 s
	Nearest Neighbor	Best cost: 1538589.77 Worst cost: 1593023.62 Avg cost: 1568122.32 Std cost: 10314.06 Best time: 0.6254 s Worst time: 2.1792 s Avg time: 1.0401 s Std time: 0.5674	Best cost: 1569007.98 Worst cost: 1626467.13 Avg cost: 1621164.67 Std cost: 11080.65 Best time: 0.6543 s Worst time: 2.1214 s Avg time: 0.7232 s Std time: 0.2347

		s	s
swap_between_routes	Random	Best cost: 1614271.57 Worst cost: 2038883.35 Avg cost: 1769563.4 Std cost: 81591.5 Best time: 0.6047 s Worst time: 0.6845 s Avg time: 0.6226 s Std time: 0.0161 s	Best cost: 1607244.87 Worst cost: 2006562.69 Avg cost: 1771997.48 Std cost: 87150.61 Best time: 0.6308 s Worst time: 0.748 s Avg time: 0.6638 s Std time: 0.0257 s
	Nearest Neighbor	Best cost: 1509796.02 Worst cost: 1589697.35 Avg cost: 1550038.32 Std cost: 18199.33 Best time: 0.6026 s Worst time: 0.6872 s Avg time: 0.6184 s Std time: 0.0099 s	Best cost: 1571306.8 Worst cost: 1626467.13 Avg cost: 1625418.83 Std cost: 6754.65 Best time: 0.6319 s Worst time: 2.256 s Avg time: 0.7133 s Std time: 0.2601 s
move_customer	Random	Best cost: 1276694.61 Worst cost: 1685811.86 Avg cost: 1452486.73 Std cost: 78343.4	Best cost: 1241124.78 Worst cost: 1723060.6 Avg cost: 1461397.05 Std cost:

		Best time: 0.593 s Worst time: 0.6678 s Avg time: 0.6158 s Std time: 0.0105 s	85377.62 Best time: 0.6039 s Worst time: 0.6672 s Avg time: 0.6263 s Std time: 0.0084 s
	Nearest Neighbor	Best cost: 1211810.8 Worst cost: 1420660.34 Avg cost: 1333561.42 Std cost: 46624.11 Best time: 0.6009 s Worst time: 0.7231 s Avg time: 0.6216 s Std time: 0.018 s	Best cost: 1218010.88 Worst cost: 1626467.13 Avg cost: 1459884.85 Std cost: 81566.91 Best time: 0.6182 s Worst time: 2.0365 s Avg time: 0.701 s Std time: 0.2187 s
random (mỗi lần lập chọn ngẫu nhiên 1 trong 4 phương pháp trên)	Random	Best cost: 1391311.38 Worst cost: 1717380.48 Avg cost: 1539363.28 Std cost: 67780.84 Best time: 0.6217 s Worst time: 0.6992 s Avg time: 0.6396 s Std time: 0.0113 s	Best cost: 1329870.68 Worst cost: 1669320.36 Avg cost: 1512921.77 Std cost: 72408.19 Best time: 0.65 s Worst time: 2.121 s Avg time: 1.2169 s Std time: 0.5548 s

	Nearest Neighbor	Best cost: 1329700.08 Worst cost: 1457019.71 Avg cost: 1398518.68 Std cost: 27998.25 Best time: 0.6291 s Worst time: 2.1729 s Avg time: 1.4865 s Std time: 0.5154 s	Best cost: 1274988.95 Worst cost: 1626467.13 Avg cost: 1485306.08 Std cost: 70207.41 Best time: 0.6522 s Worst time: 2.3159 s Avg time: 1.0196 s Std time: 0.5387 s
--	-------------------------	--	--

bảng x.x

3.4 Phân tích và đánh giá kết quả của bài toán TSP mở rộng

3.4.1. Tập 15 khách hàng (num_customers = 15, num_vehicles = 2, seed = 15)

Phương pháp tự động điều chỉnh nhiệt độ khởi đầu

Với số lượng điểm cần đi qua tương đối ít, việc sử dụng hàm tự động điều chỉnh nhiệt độ khởi đầu giúp giảm chi phí trung bình, tuy mức giảm không quá lớn. Một số trường hợp cho thấy chi phí giảm tới 30.000 (tương đương 10%). Nhìn chung, kết quả có phần ổn định hơn khi cả chi phí tốt nhất và tệ nhất đều thấp hơn so với khi không áp dụng điều chỉnh nhiệt độ. Đặc biệt, tổ hợp giữa phương pháp tạo hàng xóm ngẫu nhiên và điều chỉnh nhiệt độ tự động đạt chi phí thấp nhất là 366.364,51 — thấp hơn 29% so với tổ hợp move_customer kết hợp điều chỉnh nhiệt (517.352,02).

Tuy nhiên, phương pháp này khiến thời gian chạy tăng nhẹ ở hầu hết các tổ hợp.

Phương pháp tạo lời giải khởi đầu

Khi số điểm ít, Nearest Neighbor tỏ ra vượt trội hơn rõ rệt so với phương pháp khởi tạo ngẫu nhiên. Trong hầu hết các trường hợp, Nearest Neighbor cho chi phí thấp hơn từ 25.000 đến 100.000. Ngoại lệ duy nhất là tổ hợp với hàng xóm ngẫu nhiên, nhưng chênh lệch cũng chỉ khoảng 10.000. Phương pháp này cũng cho kết quả ổn định hơn, với phần lớn chi phí nằm trong khoảng 46.000–48.000.

Thời gian chạy của Nearest Neighbor không đáng kể hơn so với ngẫu nhiên, đôi khi còn nhanh hơn.

Phương pháp tạo hàng xóm

Hai phương pháp **swap_within_route** và **two_opt_within_route** có kết quả tệ nhất khi kết hợp với khởi tạo ngẫu nhiên, với chi phí đều trên 560.000. Đáng chú ý, **move_customer** lại cho kết quả kém hơn khi kết hợp với điều chỉnh nhiệt độ tự động (so với khi không dùng). Ngược lại, phương pháp kết hợp cả bốn kỹ thuật hàng xóm mang lại kết quả tổng thể tốt nhất, đặc biệt khi có áp dụng điều chỉnh nhiệt.

Về thời gian, hầu hết tổ hợp đều xấp xỉ 0,16s, trừ tổ hợp trộn + Nearest Neighbor và **two_opt_within_route** + ngẫu nhiên, có thời gian trung bình ~0,3s.

3.4.2. Tập 50 khách hàng (num_customers = 50, num_vehicles = 4, seed = 50)

Phương pháp tự động điều chỉnh nhiệt độ khởi đầu

Hiệu quả cải thiện chi phí khi sử dụng điều chỉnh nhiệt độ giảm rõ rệt, chỉ khoảng 10.000 (0,7%) so với không dùng. Một số tổ hợp (ví dụ với Nearest Neighbor) còn khiến chi phí tăng. Độ ổn định cũng giảm, với độ lệch chi phí dao động 5–10%.

Thời gian chạy vẫn tăng lên tương tự như với tập nhỏ.

Phương pháp tạo lời giải khởi đầu

Sự khác biệt giữa Nearest Neighbor và khởi tạo ngẫu nhiên càng rõ rệt hơn ở tập dữ liệu này. Hơn một nửa số tổ hợp với phương pháp ngẫu nhiên có chi phí trung bình vượt 1.000.000; cao nhất là tổ hợp **swap_within_route** + ngẫu nhiên với chi phí 1.561.951,76 — cao hơn hơn 500.000 so với tổ hợp tương ứng dùng Nearest Neighbor (~37,5%). Ngoài ra, khởi tạo ngẫu nhiên còn cho kết quả rất thiếu ổn định.

Thời gian chạy phụ thuộc vào phương pháp tạo hàng xóm đi kèm.

Phương pháp tạo hàng xóm

swap_within_route và **two_opt_within_route** tiếp tục cho kết quả tệ nhất. **Ngẫu nhiên** lại là phương pháp tốt nhất trong tập này, với chi phí trung bình ~82.000, và có thể giảm xuống ~79.000 khi kết hợp điều chỉnh nhiệt độ. **move_customer** và **swap_between_routes** cho kết quả trung bình.

Về thời gian, **two_opt_within_route** là chậm nhất (~1s), tiếp theo là ngẫu nhiên. Các phương pháp còn lại chạy quanh mức 0,35s.

3.4.3. Tập 100 khách hàng (num_customers = 100, num_vehicles = 7, seed = 100)

Phương pháp tự động điều chỉnh nhiệt độ khởi đầu

Ở quy mô lớn hơn, hàm điều chỉnh nhiệt độ bắt đầu thể hiện rõ sự kém hiệu quả, thậm chí gây tăng chi phí trong hầu hết trường hợp. Tệ nhất là tổ hợp với phương pháp trộn + Nearest Neighbor (tăng 6%) và **move_customer** + Nearest Neighbor (tăng 9,5%). Một vài trường hợp cải thiện chi phí thì cũng chỉ khoảng 20.000 (0,7%).

Phần lớn thời gian chạy đều tăng khi dùng phương pháp này, ngoại lệ là tổ hợp với

hàng xóm ngẫu nhiên + Nearest Neighbor hoặc two_opt_within_route + Nearest Neighbor/Random, khi thời gian giảm nhẹ.

Phương pháp tạo lời giải khởi đầu

Nearest Neighbor tiếp tục áp đảo rõ rệt so với ngẫu nhiên ở mọi tổ hợp. Cách biệt lớn nhất là khi kết hợp với swap_within_route: Nearest Neighbor đạt 1.603.923,06 trong khi ngẫu nhiên lên tới 2.840.393,86 (chênh lệch 43%).

Tuy nhiên, thời gian chạy trung bình khi dùng Nearest Neighbor thường chậm hơn một chút so với ngẫu nhiên, đặc biệt trong tổ hợp với hàng xóm ngẫu nhiên và two_opt_within_route.

Phương pháp tạo hàng xóm

Với 100 điểm, move_customer trở thành phương pháp tốt nhất, vượt ngẫu nhiên 5,8% (khi dùng khởi tạo ngẫu nhiên) và 4,6% (khi dùng Nearest Neighbor). Dù vậy, độ ổn định của nó chưa cao. Ngược lại, two_opt_within_route và ngẫu nhiên là hai phương pháp khiến thời gian chạy tăng nhiều nhất, nhất là khi đi kèm Nearest Neighbor.

3.4.1.4 Một số suy nghĩ và kết luận

Tác động của phương pháp tự động điều chỉnh nhiệt độ khởi đầu

Thử nghiệm so sánh cho thấy: Khi sử dụng nhiệt độ khởi đầu tự động, các lời giải thu được có chi phí tốt hơn so với khi sử dụng nhiệt độ cố định, trong nhiều trường hợp. Nhưng khi không gian tìm kiếm mở rộng, tác dụng của việc tự động điều chỉnh này giảm đi, đôi khi có thể phản tác dụng tùy vào cách kết hợp với các phương pháp khác.

Tác động của phương pháp tạo lời giải khởi đầu

Một xu hướng nổi bật được ghi nhận là: Phương pháp khởi tạo có ảnh hưởng rất lớn đến hiệu quả cuối cùng của giải thuật, đặc biệt rõ ở các tập dữ liệu có quy mô lớn.

- Ở tập dữ liệu 100 khách hàng, giải pháp khởi tạo bằng phương pháp Nearest Neighbor (NN) giúp giảm chi phí đến 43% so với khởi tạo ngẫu nhiên.
- Với quy mô 50 khách hàng, sự khác biệt vẫn rõ ràng: khởi tạo bằng NN kết hợp với swap_within_route cho kết quả tốt hơn khởi tạo ngẫu nhiên đến 37,5%.
- Tuy nhiên, ở tập nhỏ (15 khách hàng), khoảng cách hiệu quả giữa các phương pháp khởi tạo thu hẹp đáng kể, đôi khi chỉ ở mức 10% hoặc ít hơn. Điều này cho thấy khởi tạo thông minh đóng vai trò quan trọng hơn khi bài toán trở nên phức tạp.

Khởi tạo tốt giúp giải thuật SA bắt đầu từ một điểm có chất lượng cao hơn trong không gian lời giải, từ đó dẫn đến việc hội tụ nhanh hơn và giảm rủi ro rơi vào cực trị cục bộ không mong muốn.

Một lời giải khởi đầu tốt (chi phí thấp) đòi hỏi nhiệt độ khởi đầu không quá cao, nếu không sẽ dễ dẫn đến việc nhận các lời giải xấu hơn một cách không cần thiết. Ngược lại, nếu lời giải khởi đầu rất kém (chi phí cao), một nhiệt độ khởi đầu quá thấp sẽ khiến thuật toán khó thoát khỏi cực trị cục bộ, vì hầu hết các lời giải xấu hơn đều bị từ chối.

Tác động của phương pháp tạo hàng xóm

Khi so sánh các chiến lược hoán đổi, kết quả cho thấy:

- Move_customer thường cho kết quả tốt hơn swap_within_route, đặc biệt khi kết hợp với khởi tạo ngẫu nhiên. Trong một số thử nghiệm, move_customer giúp giảm thêm 4.6% – 5.8% so với swap_within_route.
- Tuy nhiên, mức cải thiện này giảm dần khi chất lượng khởi tạo ban đầu được nâng cao, ví dụ như khi dùng Nearest Neighbor. Trong một số trường hợp, sự khác biệt gần như không đáng kể ($\approx 0.7\%$ ở tập 50 khách hàng).

Khi khởi tạo đã tốt (ví dụ Nearest Neighbor), không gian lân cận của lời giải khởi đầu đã đủ gần cực trị toàn cục, do đó chiến lược hoán đổi đóng vai trò ít quan trọng hơn. Ngược lại, nếu khởi tạo kém, một chiến lược hoán đổi hiệu quả như move_customer có thể giúp thuật toán khám phá không gian tốt hơn và thoát khỏi cực trị cục bộ.

Kết luận:

- Khởi tạo thông minh là yếu tố quan trọng hàng đầu, đặc biệt với bài toán có quy mô lớn. Phương pháp Nearest Neighbor là lựa chọn hiệu quả, đáng được ưu tiên trong nhiều trường hợp so với ngẫu nhiên. Khi đã có khởi tạo tốt, có thể cân nhắc giảm mức độ phức tạp trong chiến lược tạo hàng xóm để tiết kiệm thời gian và tài nguyên.
- Phương pháp nhiệt độ khởi đầu tự động có tác dụng đáng cân nhắc trong bài toán có mẫu các điểm cần đi qua nhỏ, với mẫu tầm trung (khoảng 50 điểm) trở lên, hiệu quả của phương pháp giảm dần. Cần xem xét xây dựng tốt hàm tự điều chỉnh nếu vẫn muốn sử dụng.
- Cần cân nhắc mối quan hệ giữa các phương pháp tạo lời giải khởi đầu, phương pháp tạo lời giải hàng xóm, và phương pháp nhiệt độ khởi đầu tự động để lựa chọn các phương pháp phù hợp với mục đích - kết quả mong muốn từ việc chạy giải thuật.

CHƯƠNG 4. KẾT LUẬN

4.1. Các Kết Quả Đạt Được

TSP cơ bản:

- **Random Swap:** Có chi phí hành trình cao nhất (cao hơn ~45% so với tối ưu), do chỉ hoán đổi ngẫu nhiên, nhưng tốc độ nhanh nhất vì độ phức tạp thấp.
- **Two-Opt:** Cải thiện đáng kể chi phí hành trình (chỉ đắt hơn ~10% so với tối ưu). Tốc độ chạy chậm hơn Random do cần tính lại quãng đường sau mỗi lần đảo.
- **Three-Opt:** Cho kết quả tốt nhất (gần tối ưu, chỉ đắt hơn ~6.7%) nhưng có thời gian xử lý lâu nhất do độ phức tạp cao.
- **So sánh với Nearest Neighbor:** Ngoại trừ Random Swap, các phương pháp trong SA đều vượt trội về chất lượng lời giải so với Nearest Neighbor.
- **So sánh hai phương thức khởi tạo (Random vs Nearest Neighbor):** Ảnh hưởng không đáng kể đến kết quả cả về chi phí lẫn thời gian chạy.

TSP mở rộng:

Tác động của nhiệt độ khởi đầu tự động:

- Tăng chất lượng lời giải so với nhiệt độ cố định, đặc biệt ở các bài toán nhỏ.
- Hiệu quả giảm dần khi bài toán phức tạp hơn; có thể phản tác dụng nếu không kết hợp tốt với các thành phần khác.

Tác động của phương pháp khởi tạo lời giải:

- Khởi tạo Nearest Neighbor (NN) cho kết quả tốt hơn rõ rệt so với ngẫu nhiên, đặc biệt ở tập dữ liệu lớn:
 - + Giảm chi phí đến 43% ở bài toán 100 khách hàng.
 - + Tác động yếu hơn ở bài toán nhỏ (~10% chênh lệch).
- Khởi tạo tốt giúp giải thuật SA:
 - + Bắt đầu từ điểm chất lượng cao → hội tụ nhanh hơn.
 - + Giảm nguy cơ rơi vào cực trị cục bộ.

Tác động của phương pháp tạo hàng xóm:

- Move_customer hiệu quả hơn swap_within_route, nhất là khi khởi tạo ngẫu nhiên (cải thiện ~4.6% – 5.8%).
- Nếu khởi tạo đã tốt (NN), sự khác biệt giữa các chiến lược hàng xóm giảm (~0.7%).
- Khi lời giải khởi đầu tốt, chiến lược hoán đổi có thể đơn giản hóa để tiết kiệm thời gian.

4.2. Những Hạn Chế và Hướng Phát Triển

4.2.1. Hạn chế

- Thuật toán rất nhạy cảm với các tham số như nhiệt độ ban đầu (initial_temp), hệ số làm nguội (cooling_rate), và nhiệt độ dừng (stopping_temp), khiến việc lựa chọn cấu hình phù hợp trở thành một thách thức.
- Mặc dù SA có khả năng tránh được các nghiệm cục bộ nhờ chấp nhận lời giải xấu có xác suất, nhưng nếu không được cấu hình tốt (đặc biệt là trong giai đoạn nhiệt độ

cao), thuật toán có thể tiêu tốn nhiều thời gian mà không cải thiện được nghiệm. Ngoài ra, thuật toán không đảm bảo tìm được nghiệm tối ưu toàn cục, đặc biệt ở các bài toán có không gian tìm kiếm lớn như TSP với hàng trăm thành phố.

- Kết quả phụ thuộc nhiều vào cách chọn lân cận (2-opt, 3-opt, swap). Nếu dùng các kỹ thuật đơn giản như hoán đổi ngẫu nhiên, chất lượng lời giải thường thấp. Trong khi đó, các kỹ thuật phức tạp hơn như 3-opt tuy cải thiện chất lượng nhưng lại làm tăng thời gian tính toán đáng kể.
- Khoảng cách chỉ tính bằng phương thức Euclid, chưa thể chỉnh khoảng cách cụ thể giữa các thành phố.
- Còn nhiều hướng mở rộng bài toán TSP chưa khám phá như:
 - + Time Window TSP (TW-TSP): thêm thời gian vào mô hình: mỗi thành phố chỉ có thể tới vào thời gian cụ thể
 - + Prize Collecting TSP (PCTSP): đổi mục tiêu thành tối ưu điểm thưởng từ việc ghé thăm các thành phố và tổng chi phí.
 - + Stochastic TSP (STSP): chỉ chọn k điểm đến trong n điểm ($k \leq n$) theo phân phối xác suất

4.2.1. Hướng phát triển

- Thay vì sử dụng tốc độ làm nguội cố định như trong đoạn mã (exponential/ linear/ logarithmic), ta có thể thiết kế hệ thống tự điều chỉnh tốc độ làm nguội dựa trên hiệu suất hiện tại của thuật toán (ví dụ, nếu nhiều bước liên tiếp không cải thiện lời giải thì chậm lại tốc độ làm nguội).
- Thêm tính năng tự set khoảng cách giữa các thành phố, ví dụ như sử dụng một ma trận khoảng cách
- So sánh thuật toán với nhiều thuật toán khác hơn như GA, Hillclimbing,...
- Khám phá thêm nhiều dạng bài toán TSP
- Tìm phương pháp tốt hơn để đánh giá độ hiệu quả của thuật toán

TÀI LIỆU THAM KHẢO

1. Brownlee, J. (2024). Chapter 3: Generating solutions and acceptance criteria. In *Simulated Annealing Afternoon: A Practical Guide for Software Developers*. AlgorithmAfternoon.com.
2. Albrecht, A., & Wong, C.-K. (2001). On logarithmic simulated annealing. In J. Karhumäki, H. Maurer, G. Paun, & G. Rozenberg (Eds.), *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics* (pp. 301–314). Springer.
https://doi.org/10.1007/3-540-44929-9_23

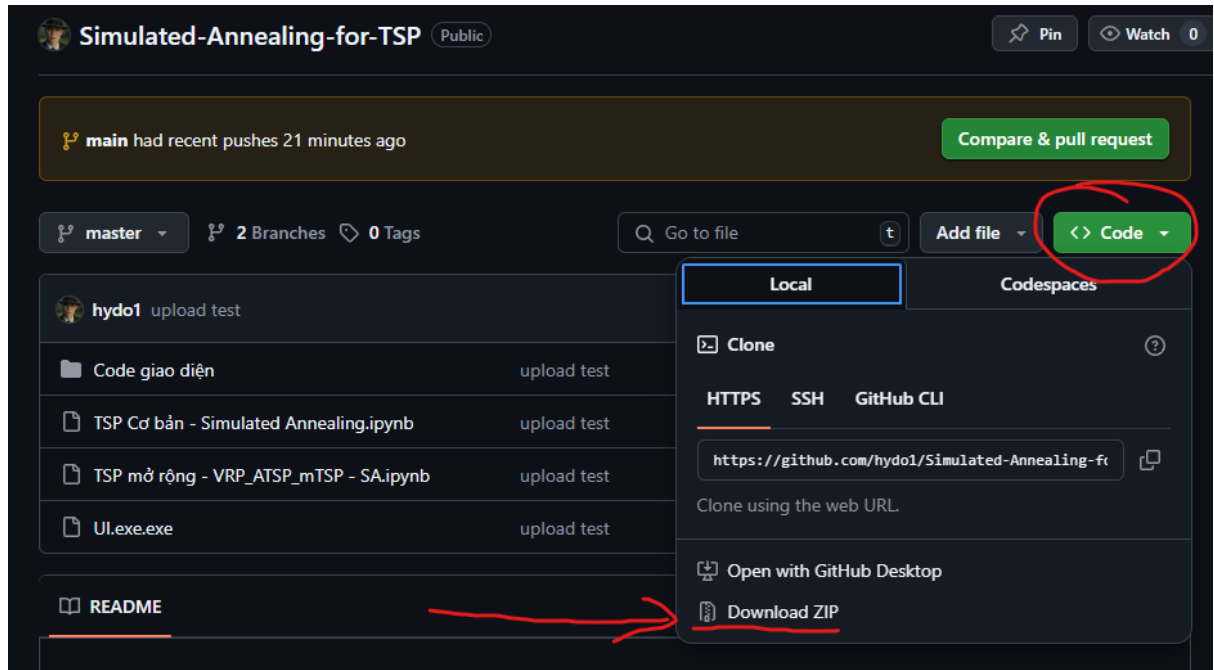
PHỤ LỤC

Mã nguồn push lên Github và dẫn link vào đây





<https://github.com/hydo1/Simulated-Annealing-for-TSP>

Hướng dẫn cài đặt/sử dụng chương trình

Hướng dẫn cài đặt



Truy cập đường link, nhấn vào nút code và Download Zip.

 UI.exe	26/5/2025 10:26 pm	Application	85,901 KB
 TSP Cơ bản - Simulated Annealing	26/5/2025 10:26 pm	Jupyter Source File	129 KB
 TSP mở rộng - VRP_ATSP_mTSP - SA	26/5/2025 10:26 pm	Jupyter Source File	274 KB
 Code giao diện	26/5/2025 10:27 pm	File folder	

- Giải nén và ấn vào file UI.exe để khởi động giao diện
- TSP cơ bản và TSP mở rộng là hai file mã nguồn
- Folder code giao diện là mã nguồn cho UI

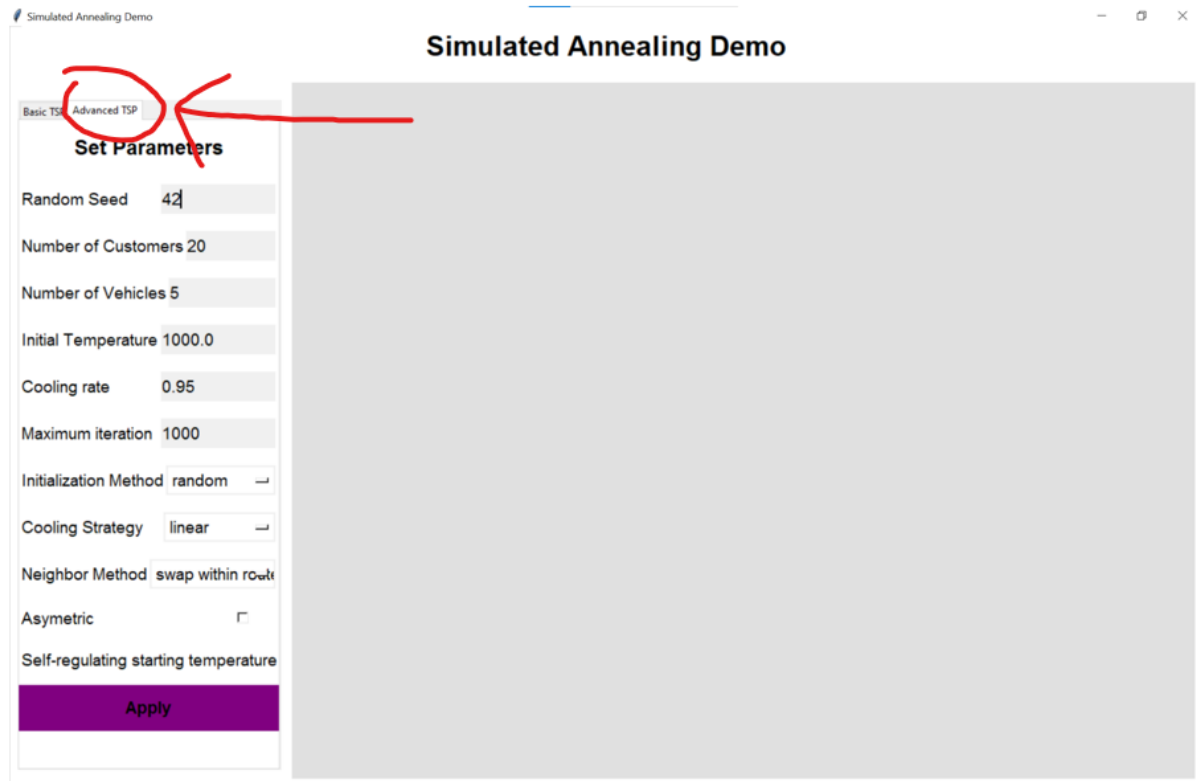
UI và hướng dẫn sử dụng

The screenshot shows a web application titled "Simulated Annealing Demo". On the left is a control panel with two tabs: "Basic TSP" (selected) and "Advanced TSP". Under "Add Cities", there are input fields for "1" and "3", and a purple "Add" button. Below this is the "Set Parameters" section with the following settings: "Numbers of Cities" (20), "Initial Temperature" (1000.0), "Cooling Rate" (0.995), "Maximum Iteration" (200000), "Stopping Temperature" (1e-200), "Neighbor Method" (two opt), "Initialization Method" (random), "Cooling Strategy" (exponential), and "Only Nearest Neighbor" (unchecked). At the bottom of the panel are two purple buttons: "Apply" and "Clear Cities". The main area on the right is a large, empty gray rectangle representing the map.

- Nhập và chọn các biến tham gia rồi nhấn apply để chạy chương trình

Lưu ý:

- Hai ô vuông trên đầu nhận tọa độ x (ô trái) và tọa độ y (ô phải). Sau khi nhập, ấn nút add thành phố sẽ hiển thị trên bản đồ
- Nhấn nút Clear Cities để xóa các thành phố đã nhập trên bản đồ.
- **Chương trình sẽ chỉ tạo random số thành phố (dựa vào biến Numbers of Cities) khi đã clear hết tọa độ tự nhập.**



- Chọn Advanced TSP ở trên để chuyển qua bài toán mở rộng
- Nhập và chọn các biến tham gia rồi nhấn apply để chạy chương trình

PHÂN CÔNG

Nhiệm vụ	Thành viên
2.2.1 Ứng Dụng Giải thuật Simulated Annealing Cho Bài Toán Người du lịch 2.2.c. Hàm chính (Simulated Annealing) 2.2.d. Hàm vẽ đồ thị (Mathplotlib) 3.1. Các Tình Huống Trong TSP cơ bản 3.2. Phân Tích và Đánh Giá TSP cơ bản 4.1. Các Kết Quả Đạt Được 4.2. Những Hạn Chế và Hướng Phát Triển	Đỗ Thái Gia Hy MSSV: 31231021575
1.1. Giới Thiệu Về Traveling Salesman Problem (TSP) 1.2. Phát Biểu Bài Toán 1.3. Một Số Hướng Tiếp Cận Giải Quyết Bài Toán 2.1. Giới Thiệu Về Simulated Annealing 2.2.1 Ứng Dụng Giải thuật Simulated Annealing Cho Bài Toán Người du lịch 2.2.a. Các hàm khởi tạo/tính toán cần thiết 2.2.b. Hàm tạo láng giềng (Neighbor Methods)	Trần Huỳnh Huy Thông MSSV: 31231023784
2.2.2 Ứng Dụng Giải thuật Simulated Annealing Cho Bài Toán Người du lịch mở rộng	Hà Quang Đại MSSV: 31231026075

<p>3.4.1. Tập 15 khách hàng (num_customers = 15, num_vehicles = 2, seed = 15)</p> <p>3.4.2. Tập 50 khách hàng (num_customers = 50, num_vehicles = 4, seed = 50)</p> <p>3.4.1.4 Tác động của phương pháp tạo lời giải khởi đầu</p>	
<p>3.3 Các tình huống trong bài toán TSP mở rộng</p> <p>3.4.3. Tập 100 khách hàng (num_customers = 100, num_vehicles = 7, seed = 100)</p> <p>3.4.1.4 Tác động của phương pháp tự động điều chỉnh nhiệt độ khởi đầu</p> <p>3.4.1.4 Tác động của phương pháp tạo hàng xóm</p> <p>Xây dựng giao diện chương trình</p>	<p>Thái Thủy Đức</p> <p>MSSV: 31231026287</p>