KHOA KỸ THUẬT VÀ CÔNG NGHỆ
BỘ MÔN CÔNG NGHỆ THÔNG TIN

THỰC TẬP ĐỒ ÁN CƠ SỞ NGÀNH

HỌC KỲ I, NĂM HỌC 2024-2025

Viết chương trình mô phỏng bài toán người du lịch

Giảng viên hướng dẫn:
[ThS./TS.] Trầm Hoàng Nam

Sinh viên thực hiện:
Họ tên: Phan Đăng Khoa
MSSV: 110122227
Lớp: DA22TTB

Trà Vinh, tháng 12 năm 2024

KHOA KỸ THUẬT VÀ CÔNG NGHỆ
BỘ MÔN CÔNG NGHỆ THÔNG TIN

THỰC TẬP ĐỒ ÁN CƠ SỞ NGÀNH

HỌC KỲ I, NĂM HỌC 2024-2025

Viết chương trình mô phỏng bài toán người du lịch

Giảng viên hướng dẫn:
[ThS./TS.] Trầm Hoàng Nam

Sinh viên thực hiện:
Họ tên: Phan Đăng Khoa
MSSV: 110122227
Lớp: DA22TTB

Trà Vinh, tháng 12 năm 2024

NHẬN XÉT CỦA GIẢNG VIÊN HƯỚNG DẪN

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

NHẬN XÉT CỦA THÀNH VIÊN HỘI ĐỒNG

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

LỜI CẢM ƠN

Kính gửi thầy Trầm Hoàng Nam và các giảng viên bộ môn Khoa Kỹ thuật và Công nghệ, em xin gửi lời cảm ơn chân thành và sâu sắc nhất tới thầy Trầm Hoàng Nam, người đã trực tiếp hướng dẫn và hỗ trợ em trong suốt quá trình thực hiện đồ án cơ sở ngành. Những lời khuyên quý báu, sự tận tâm và kiên nhẫn của thầy đã giúp em vượt qua những khó khăn, thách thức trong công việc nghiên cứu và hoàn thiện đồ án. Em thực sự trân trọng và biết ơn thầy vì tất cả sự giúp đỡ đó.

Ngoài ra, em cũng xin gửi lời cảm ơn tới các giảng viên bộ môn Khoa Kỹ thuật và Công nghệ, những người đã luôn tận tình giảng dạy và chia sẻ kiến thức quý báu trong suốt quá trình học tập của em. Em sẽ luôn ghi nhớ những bài học, kinh nghiệm quý giá mà thầy cô đã truyền đạt.

Một lần nữa, em xin chân thành cảm ơn thầy và các giảng viên trong khoa. Em hy vọng sẽ tiếp tục nhận được sự hỗ trợ và chỉ dạy từ thầy cô trong những chặng đường học tập và nghiên cứu tiếp theo.

MỤC LỤC

DANH MỤC HÌNH ẢNH

TÓM TẮT ĐỒ ÁN CƠ SỞ NGÀNH

Tóm tắt vấn đề nghiên cứu: bài toán người du lịch (Traveling salesman problem - TSP) là một trong những bài toán nổi tiếng trong lý thuyết đồ thị và tối ưu hóa. Nó yêu cầu tìm một hành trình ngắn nhất qua tất cả các thành phố trong một danh sách, mỗi thành phố được ghé thăm đúng một lần trước khi trở lại điểm xuất phát. Bài toán này có ứng dụng rộng rãi trong logistics, lập lịch trình, và tối ưu hóa mạng.

Các hướng tiếp cận chính để giải quyết bài toán bao gồm:

1. Phương pháp vét cạn (Brute force): tính toán mọi hoán vị của các thành phố và chọn đường đi có chi phí thấp nhất.
2. Heuristic: các thuật toán gần đúng như Nearest neighbor, Christofides và Lin-Kernighan giúp tìm lời giải nhanh hơn nhưng không đảm bảo tối ưu.
3. Phương pháp tối ưu hóa: sử dụng quy hoạch tuyến tính, phân nhánh và cận (Branch and bound), hoặc quy hoạch động.
4. Giải pháp metaheuristic: Các thuật toán như Genetic algorithm, Simulated annealing, hoặc ant colony optimization.

Cách giải quyết vấn đề: chương trình mô phỏng sử dụng thuật toán Nearest neighbor như sau:

- Khởi đầu từ một thành phố, mỗi lần chọn thành phố gần nhất chưa được ghé thăm.
- Tính tổng chi phí dựa trên ma trận khoảng cách giữa các thành phố.
- Trả về thành phố khởi điểm để hoàn thành hành trình.

Chương trình được xây dựng trên Python, sử dụng thư viện như Tkinter để tạo giao diện người dùng, NumPy để xử lý ma trận khoảng cách, và JSON để quản lý log.

Một số kết quả đạt được:

- Giao diện mô phỏng trực quan: Hiển thị đồ thị của các thành phố, đường đi và kết quả tính toán.
- Tính năng linh hoạt:
  + Tạo ngẫu nhiên các thành phố.

+ Nhập tọa độ thủ công.

+ Lưu và tải log của các kết quả thực hiện.

- Kết quả cụ thể:

+ Hiển thị ma trận khoảng cách giữa các thành phố.

+ Xuất đường đi được tính toán và chi phí tổng hợp.

+ Tích hợp thời gian chạy của thuật toán, giúp người dùng đánh giá hiệu quả.

MỞ ĐẦU

Lý do chọn đề tài: bài toán người du lịch (TSP) là một trong những bài toán cổ điển nhưng đầy thách thức trong lý thuyết tối ưu hóa và ứng dụng thực tế. Việc tìm ra lời giải hiệu quả không chỉ giúp phát triển thuật toán trong lĩnh vực khoa học máy tính mà còn góp phần cải thiện hiệu quả hoạt động trong các lĩnh vực như logistics, quản lý chuỗi cung ứng, và điều phối mạng lưới giao thông. Đề tài này cũng giúp người nghiên cứu nắm vững kiến thức lý thuyết đồ thị và khả năng ứng dụng công nghệ vào việc giải quyết các vấn đề thực tiễn.

Mục đích nghiên cứu:

- Nâng cao hiểu biết về lý thuyết đồ thị và các thuật toán tối ưu hóa.
- Xây dựng một chương trình mô phỏng trực quan, minh họa bài toán Người du lịch, giúp người dùng hiểu rõ cách hoạt động của các thuật toán tìm kiếm lời giải.
- Ứng dụng công nghệ hiện đại để tối ưu hóa chi phí và thời gian, từ đó đề xuất các giải pháp áp dụng vào thực tế như trong giao thông hoặc vận chuyển hàng hóa.

Đối tượng nghiên cứu:

- Bài toán người du lịch (TSP) trong lý thuyết đồ thị.
- Các thuật toán được sử dụng để giải quyết bài toán, đặc biệt là Nearest Neighbor, một phương pháp heuristic phổ biến.
- Ứng dụng lập trình và giao diện người dùng để mô phỏng và minh họa lời giải.

Phạm vi nghiên cứu

- Phạm vi lý thuyết: tập trung vào các thuật toán tìm kiếm và tối ưu hóa, phân tích ưu nhược điểm của thuật toán heuristic.
- Phạm vi ứng dụng: chương trình mô phỏng được xây dựng trên nền tảng Python, sử dụng giao diện trực quan với thư viện Tkinter và khả năng xử lý dữ liệu bằng NumPy.

- Giới hạn: chỉ thực hiện với bài toán TSP kích thước nhỏ đến trung bình (10-100 thành phố) để đảm bảo hiệu năng tính toán và hiển thị trực quan.

CHƯƠNG 1: TỔNG QUAN

1.1 Giới thiệu tổng quan về vấn đề

Bài toán Người du lịch (Traveling Salesman Problem - TSP) là một trong những bài toán tối ưu hóa tổ hợp kinh điển và có nhiều ứng dụng thực tiễn. Trong bài toán này, người du lịch cần ghé thăm tất cả các thành phố trong danh sách đúng một lần, sau đó quay lại điểm xuất phát, sao cho tổng quãng đường hoặc chi phí là nhỏ nhất. Đây là một bài toán thuộc lớp NP-hard, nghĩa là việc tìm ra lời giải tối ưu trong thời gian ngắn là không khả thi với dữ liệu đầu vào lớn, nhưng vẫn có thể tìm được lời giải gần đúng hoặc tối ưu cục bộ thông qua các phương pháp heuristic hoặc metaheuristic.

1.2 Các yếu tố chính liên quan đến nghiên cứu

1.2.1 Lý thuyết nền tảng

- Lý thuyết đồ thị: biểu diễn bài toán TSP dưới dạng một đồ thị trọng số đầy đủ, trong đó các đỉnh là các thành phố và các cạnh là khoảng cách giữa chúng.
- Các thuật toán tối ưu hóa và heuristic như Nearest neighbor hoặc giải pháp bằng phân nhánh và cận.

1.2.2 Thách thức

- Số lượng thành phố tăng khiến số hoán vị có thể xảy ra tăng theo cấp số nhân, dẫn đến độ phức tạp tính toán rất lớn.
- Đảm bảo sự cân bằng giữa độ chính xác của lời giải và thời gian thực thi thuật toán.

1.2.3 Ứng dụng thực tiễn

- Trong logistics: tối ưu hóa tuyến đường vận chuyển hàng hóa để giảm chi phí.
- Trong lập lịch trình: lên kế hoạch di chuyển hoặc phân phối nguồn lực.
- Trong khoa học dữ liệu: sử dụng bài toán này làm cơ sở để nghiên cứu các bài toán tối ưu hóa khác.

1.3 Nghiên cứu và giải pháp được tập trung

Nghiên cứu tập trung vào:

- Phân tích và cài đặt thuật toán Nearest neighbor: một thuật toán heuristic đơn giản và hiệu quả cho các bài toán TSP kích thước nhỏ đến trung bình.
- Xây dựng mô hình và giao diện: chương trình mô phỏng được phát triển bằng Python với thư viện Tkinter, cho phép người dùng dễ dàng nhập dữ liệu, thực thi thuật toán, và xem kết quả trực quan.
- Đánh giá hiệu suất thuật toán: đo lường thời gian thực thi, chi phí đường đi, và khả năng áp dụng của Nearest neighbor. Đồng thời, phân tích ưu nhược điểm của thuật toán khi áp dụng cho các bài toán thực tế.

CHƯƠNG 2: NGHIÊN CỨU LÝ THUYẾT

2.1 Cơ sở lý thuyết

Bài toán Người du lịch (TSP) được mô hình hóa bằng lý thuyết đồ thị, với các yếu tố chính sau:

Lý thuyết đồ thị:

- Đồ thị trọng số đầy đủ: đồ thị $G = (V, E)$, trong đó:
  + V: tập các đỉnh, biểu diễn các thành phố cần ghé thăm.
  + E: tập các cạnh, biểu diễn các con đường giữa các thành phố.
  + Mỗi cạnh $e_{ij} \in E$ được gán trọng số w_{ij}, đại diện cho khoảng cách hoặc chi phí giữa thành phố i và j.
- Đặc điểm: đồ thị đầy đủ có cạnh giữa mọi cặp đỉnh.

Đường đi Hamiltonian là một chu trình qua tất cả các đỉnh của đồ thị, trong đó mỗi đỉnh được ghé thăm đúng một lần và quay lại đỉnh ban đầu.

Tính NP-hard: bài toán TSP không có thuật toán đa thức để tìm lời giải tối ưu cho các trường hợp tổng quát, nhưng có thể sử dụng các thuật toán gần đúng hoặc heuristic để tìm lời giải khả dĩ.

2.2 Đầu vào và đầu ra của bài toán

2.2.1 Đầu vào

Số thành phố: $n, với\ i\ n \geq 2$

Tọa độ của các thành phố: Mỗi thành phố được biểu diễn bằng cặp tọa độ $(x_i, y_i), i\ =\ 1, 2, \ldots, n.$

Ma trận khoảng cách: ma trận vuông D, trong đó phần tử d_{ij} là khoảng cách giữa thành phố i và j, được tính bằng công thức:

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

2.2.2 Đầu ra

Chu trình tối ưu (hoặc gần tối ưu): một chuỗi các đỉnh biểu diễn thứ tự ghé thăm các thành phố.

Chi phí tổng cộng: tổng khoảng cách của chu trình.

2.3 Lý luận và giả thiết khoa học

2.3.1 Lý luận

Cấu trúc bài toán: bài toán người du lịch là một bài toán tối ưu hóa toàn cục trên đồ thị. Mục tiêu là tìm chu trình ngắn nhất có thể.

Phương pháp heuristic: sử dụng thuật toán Nearest neighbor làm phương pháp tiếp cận chính để đảm bảo tốc độ tính toán cho các bài toán kích thước vừa và nhỏ.

2.3.2 Giả thiết khoa học

- Khoảng cách giữa các thành phố được tính bằng công thức Euclid và không thay đổi.
- Đồ thị đầu vào luôn đầy đủ, nghĩa là luôn có cạnh nối giữa bất kỳ hai thành phố.
- Số thành phố nhỏ đến trung bình, đảm bảo thời gian tính toán khả thi khi sử dụng các thuật toán heuristic như Nearest neighbor.

2.4 Phương pháp nghiên cứu

2.4.1 Phương pháp lý thuyết

- Tìm hiểu và phân tích thuật toán Nearest neighbor:
  + Xuất phát từ một thành phố ban đầu.
  + Mỗi bước chọn thành phố gần nhất chưa được ghé thăm.
  + Hoàn thành chu trình bằng cách quay lại thành phố ban đầu.
- Phân tích độ phức tạp: Thuật toán Nearest neighbor có độ phức tạp $O(n^2)$, thích hợp cho các bài toán nhỏ.

2.4.2 Phương pháp thực nghiệm

Xây dựng chương trình mô phỏng:

- Sử dụng Python và thư viện Tkinter để tạo giao diện đồ họa cho người dùng.
- Thư viện NumPy để tính toán ma trận khoảng cách và trọng số.

2.4.3 Quy trình thực nghiệm

1. Xác định đầu vào: nhập số thành phố và tọa độ hoặc tạo ngẫu nhiên.
2. Tính toán ma trận khoảng cách: tạo ma trận dựa trên tọa độ thành phố.
3. Áp dụng thuật toán NN: tìm đường đi và chi phí gần tối ưu.
4. Đánh giá kết quả: hiển thị trực quan chu trình trên giao diện và log thông tin chi tiết.

2.5 Thuật toán Nearest neighbor

Bước 1: Bắt đầu từ một thành phố ngẫu nhiên v_0.

Bước 2: Trong mỗi bước, chọn thành phố gần nhất v_{i+1} chưa được ghé thăm.

Bước 3: Khi tất cả các thành phố đã được ghé thăm, quay lại thành phố khởi điểm để hoàn thành chu trình.

Bước 4: Tính chi phí tổng cộng và lưu kết quả.

CHƯƠNG 3: HIỆN THỰC HÓA NGHIÊN CỨU

3.1 Đặc tả nhu cầu

3.1.1 Yêu cầu chức năng

- Nhập dữ liệu: cho phép người dùng nhập số lượng thành phố và tọa độ từng thành phố, hoặc tạo ngẫu nhiên.
- Hiển thị đồ thị: trực quan hóa các thành phố và kết nối giữa chúng trên giao diện.
- Chạy thuật toán: tính toán đường đi bằng thuật toán Nearest neighbor và hiển thị kết quả.
- Quản lý log: lưu trữ, hiển thị, và tải lại kết quả tính toán.
- Ma trận: hiển thị ma trận khoảng cách và trọng số.

3.1.2 Yêu cầu phi chức năng

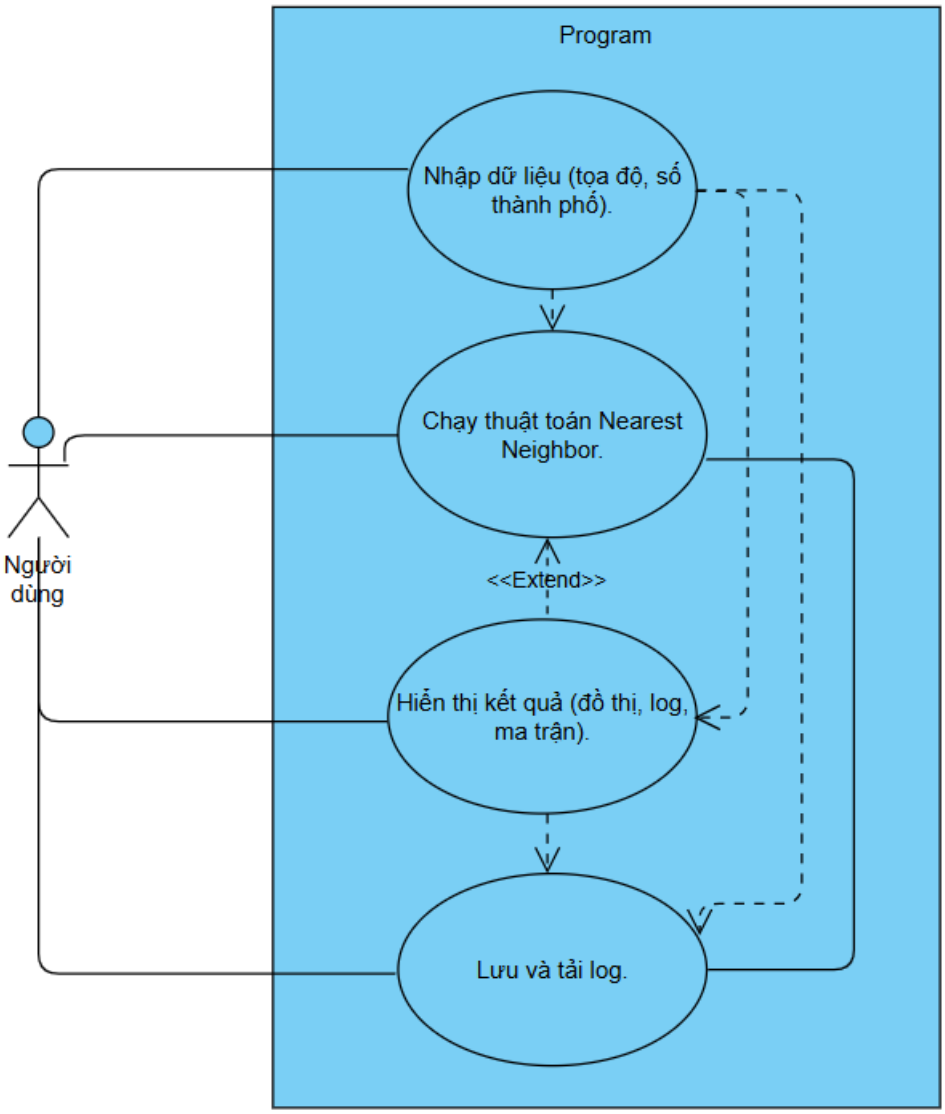
- Hiệu năng: ứng dụng chạy ổn định với số lượng thành phố nhỏ đến trung bình (10-100).
- Thân thiện người dùng: giao diện đơn giản, dễ sử dụng.
- Tính mở rộng: có thể thêm các thuật toán khác trong tương lai.

3.2 Phân tích thiết kế hệ thống

3.2.1 Kiến trúc tổng quát

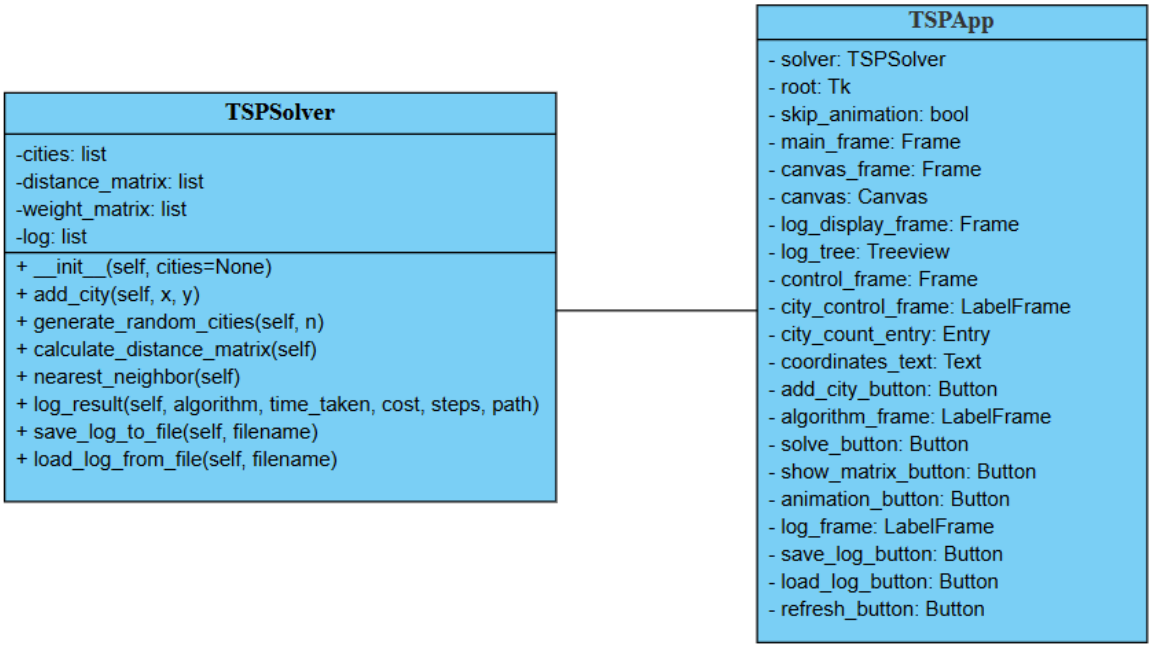
- Giao diện người dùng (UI): được phát triển bằng Tkinter, cung cấp các chức năng nhập liệu, hiển thị đồ thị, và tương tác.
- Lớp xử lý logic: chứa các thuật toán giải bài toán TSP, tính toán ma trận khoảng cách, và quản lý log.
- Lớp quản lý dữ liệu: xử lý và lưu trữ dữ liệu đầu vào, đầu ra, và log tính toán.

3.2.2 Biểu đồ Use Case



Hình 1 Biểu đồ use case

3.2.3 Biểu đồ Class



Hình 2 Biểu đồ Class

3.2.4 Các bước hoạt động

Quy trình chạy thuật toán Nearest Neighbor:

1. Người dùng nhập hoặc tạo ngẫu nhiên các thành phố.
2. Hệ thống tính toán ma trận khoảng cách.
3. Thuật toán Nearest Neighbor được thực thi để tìm đường đi ngắn nhất.
4. Hiển thị kết quả: đường đi, chi phí, thời gian tính toán.

3.3 Cách thức cài đặt chương trình

3.3.1 Công nghệ sử dụng

- Ngôn ngữ lập trình: Python.
- Thư viện:
  + Tkinter: tạo giao diện đồ họa người dùng.
  + messagebox: hiển thị các hộp thoại thông báo như lỗi, cảnh báo hoặc xác nhận.

+ filedialog: quản lý các tệp (mở hoặc lưu).
+ ttk: cung cấp các widget nâng cao.
+ tertools: hỗ trợ làm việc với tổ hợp và hoán vị.
+ random: tạo giá trị ngẫu nhiên.
+ math: thực hiện các phép toán cơ bản.
+ time: đo thời gian thực hiện thuật toán.
+ json: lưu và tải dữ liệu ở định dạng JSON.
+ numpy: tính toán các ma trận, sử dụng cho việc tính khoảng cách.

```
import itertools
import random
import math
import time
import json
import tkinter as tk
from tkinter import messagebox, filedialog, ttk
import numpy as np
```

Hình 3 Các thư viện

3.3.2 Quy trình hiện thực hóa

Bước 1: chèn thư viện

Bước 2: xây dựng cấu trúc chương trình

Lớp TSPSolver: lớp chính xử lý bài toán người du lịch. Chứa các thuộc tính và phương thức để quản lý dữ liệu, tính toán khoảng cách, và chạy thuật toán.

- Thuộc tính:
  + self.cities: danh sách tọa độ của các thành phố.
  + self.distance_matrix: ma trận khoảng cách giữa các thành phố.
  + self.weight_matrix: ma trận trọng số (nghịch đảo khoảng cách).
  + self.log: lưu kết quả các lần chạy thuật toán.
- Các phương thức chính:
  + add_city(x, y): thêm một thành phố với tọa độ (x, y) vào danh sách.

```
def add_city(self, x, y):
    self.cities.append((x, y))
```

Hình 4 Phương thức add_city(self, x, y)

- self: tham chiếu đến đối tượng hiện tại của lớp mà hàm này thuộc về.
- x, y: các tham số đầu vào, đại diện cho tọa độ của thành phố.
- self.cities: một danh sách thuộc đối tượng, nơi lưu trữ các thành phố đã thêm vào.
- append((x, y)): thêm một tuple chứa tọa độ (x, y) vào danh sách self.cities.

+ generate_random_cities(n): sinh n thành phố ngẫu nhiên trong giới hạn khung vẽ.

```
def generate_random_cities(self, n):
    canvas_width, canvas_height = 1220, 550
    self.cities = [(random.randint(50, canvas_width - 50), random.randint(50, canvas_height - 50)) for _ in range(n)]
    self.calculate_distance_matrix()
```

Hình 5 Phương thức generate_random_cities(self, n)

- self: tham chiếu đến đối tượng hiện tại của lớp mà hàm này thuộc về.
- n: số lượng thành phố cần tạo ra (số thành phố ngẫu nhiên).
- canvas_width, canvas_height: kích thước của "canvas" (khung vẽ), giới hạn tọa độ của các thành phố trong phạm vi này. Ở đây, chiều rộng là 1220 và chiều cao là 550.
- self.cities: danh sách lưu trữ các thành phố, mỗi thành phố là một tuple với tọa độ (x, y).
- random.randint(50, canvas_width - 50): tạo một giá trị ngẫu nhiên cho tọa độ x trong phạm vi từ 50 đến canvas_width - 50 (giới hạn khoảng cách để thành phố không bị ra ngoài cạnh của khung vẽ).
- random.randint(50, canvas_height - 50): tạo một giá trị ngẫu nhiên cho tọa độ y trong phạm vi từ 50 đến canvas_height - 50 (tương tự như trên).
- for _ in range(n): duyệt qua vòng lặp tạo ra n thành phố.
- self.calculate_distance_matrix(): sau khi tạo các thành phố ngẫu nhiên, hàm này gọi một hàm khác (calculate_distance_matrix) để tính toán ma trận khoảng cách giữa các thành phố.

+ calculate_distance_matrix(): tính toán ma trận khoảng cách và trọng số dựa trên tọa độ.

```
def calculate_distance_matrix(self):
    self.distance_matrix = []
    self.weight_matrix = []
    for i in range(len(self.cities)):
        distances = []
        weights = []
        for j in range(len(self.cities)):
            dist = np.linalg.norm(np.array(self.cities[i]) - np.array(self.cities[j]))
            distances.append(dist)
            weights.append(1 / dist if dist != 0 else 0)
        self.distance_matrix.append(distances)
        self.weight_matrix.append(weights)
```

Hình 6 Phương thức calculate_distance_matrix(self)

- self: tham chiếu đến đối tượng hiện tại của lớp mà hàm này thuộc về.
- self.distance_matrix: danh sách 2 chiều (ma trận) để lưu trữ khoảng cách giữa các thành phố.
- self.weight_matrix: danh sách 2 chiều (ma trận) để lưu trữ trọng số giữa các thành phố, được tính từ khoảng cách.
- self.distance_matrix = [] và self.weight_matrix = []: tạo các ma trận rỗng để lưu trữ kết quả.
- for i in range(len(self.cities)): lặp qua từng thành phố trong danh sách self.cities.
- distances = [] và weights = []: tạo các danh sách rỗng để lưu trữ khoảng cách và trọng số cho thành phố thứ i.
- for j in range(len(self.cities)): lặp qua các thành phố còn lại (bao gồm cả thành phố thứ i).
- dist = np.linalg.norm(np.array(self.cities[i]) - np.array(self.cities[j])): tính khoảng cách Euclid (khoảng cách thẳng) giữa thành phố i và thành phố j bằng cách sử dụng hàm np.linalg.norm, chuyển tọa độ của các thành phố thành mảng NumPy.
- weights.append(1 / dist if dist != 0 else 0): tính trọng số là nghịch đảo của khoảng cách (trừ khi khoảng cách là 0, trong trường hợp đó trọng số sẽ là 0).
- self.distance_matrix.append(distances): thêm danh sách distances vào ma trận khoảng cách.

- self.weight_matrix.append(weights): thêm danh sách weights vào ma trận trọng số.

  + nearest_neighbor(): thuật toán "người hàng xóm gần nhất" để tìm đường đi tối ưu.

```
def nearest_neighbor(self):
    start = 0
    unvisited = set(range(len(self.cities)))
    unvisited.remove(start)
    path = [start]
    total_cost = 0
    start_time = time.time()

    while unvisited:
        last = path[-1]
        nearest = min(unvisited, key=lambda city: self.distance_matrix[last][city])
        total_cost += self.distance_matrix[last][nearest]
        path.append(nearest)
        unvisited.remove(nearest)

    total_cost += self.distance_matrix[path[-1]][path[0]]  # Return to start
    time_taken = time.time() - start_time
    self.log_result("Nearest Neighbor", time_taken, total_cost, len(path), path)
    return path, total_cost
```

Hình 7 Phương thức nearest_neighbor(self)

- self: tham chiếu đến đối tượng hiện tại của lớp mà hàm này thuộc về.
- start = 0: chọn thành phố đầu tiên (thành phố số 0) làm điểm xuất phát.
- unvisited = set(range(len(self.cities))): tạo một tập hợp chứa tất cả các thành phố chưa được thăm.
- unvisited.remove(start): loại bỏ thành phố xuất phát khỏi tập hợp các thành phố chưa thăm.
- path = [start]: danh sách lưu trữ đường đi (chứa các chỉ số thành phố đã được thăm), bắt đầu từ thành phố xuất phát.
- total_cost = 0: biến lưu trữ tổng chi phí (tổng khoảng cách đã di chuyển).
- start_time = time.time(): lưu lại thời gian bắt đầu để tính toán thời gian chạy của thuật toán.
- while unvisited:: tiếp tục lặp cho đến khi không còn thành phố nào chưa thăm.
- last = path[-1]: lấy thành phố cuối cùng trong path (thành phố hiện tại).

- nearest = min(unvisited, key=lambda city: self.distance_matrix[last][city]): tìm thành phố chưa thăm gần nhất với thành phố hiện tại, bằng cách tìm thành phố có khoảng cách nhỏ nhất trong self.distance_matrix (ma trận khoảng cách).
- total_cost += self.distance_matrix[last][nearest]: cộng chi phí di chuyển từ thành phố hiện tại đến thành phố gần nhất vào tổng chi phí.
- path.append(nearest): thêm thành phố gần nhất vào danh sách path.
- unvisited.remove(nearest): loại bỏ thành phố gần nhất khỏi tập các thành phố chưa thăm.
- total_cost += self.distance_matrix[path[-1]][path[0]]: cộng chi phí di chuyển từ thành phố cuối cùng quay lại thành phố xuất phát vào tổng chi phí.
- time_taken = time.time() - start_time: tính toán thời gian thực hiện thuật toán.
- self.log_result("Nearest Neighbor", time_taken, total_cost, len(path), path): ghi lại kết quả của thuật toán, bao gồm tên thuật toán, thời gian chạy, tổng chi phí, số thành phố và lộ trình.
- return path, total_cost: trả về đường đi (path) và tổng chi phí (total_cost).

    + log_result(algorithm, time_taken, cost, steps, path): lưu kết quả chạy thuật toán.

```
def log_result(self, algorithm, time_taken, cost, steps, path):
    self.log.append({
        "Thuật toán": algorithm,
        "Thời gian": time_taken,
        "Chi phí": cost,
        "Bước": steps,
        "Đường đi": path
    })
```

Hình 8 Phương thức log_result(self, algorithm, time_taken, cost, steps, path)

- self: tham chiếu đến đối tượng hiện tại của lớp mà hàm này thuộc về.
- algorithm: tên của thuật toán đã được sử dụng (ví dụ: "Nearest Neighbor").
- time_taken: thời gian chạy của thuật toán (tính bằng giây).
- cost: tổng chi phí (hoặc tổng khoảng cách) của lộ trình đã tìm được.

- steps: số lượng bước (hoặc thành phố) trong đường đi (số lượng các thành phố trong lộ trình).
- path: danh sách các thành phố trong lộ trình (được thể hiện bằng các chỉ số thành phố).
- self.log.append({...}): thêm một từ điển (dictionary) vào danh sách self.log. Từ điển này chứa các thông tin về thuật toán, thời gian, chi phí, số bước và đường đi.
- "Thuật toán": lưu tên thuật toán đã sử dụng.
- "Thời gian": lưu thời gian thực hiện thuật toán.
- "Chi phí": lưu tổng chi phí (hoặc tổng khoảng cách).
- "Bước": lưu số lượng bước (hoặc thành phố) trong đường đi.
- "Đường đi": lưu danh sách các thành phố trong đường đi.

    + save_log_to_file(filename) và load_log_from_file(filename): lưu và tải log từ file JSON.

```
def save_log_to_file(self, filename):
    with open(filename, 'w') as file:
        json.dump(self.log, file)

def load_log_from_file(self, filename):
    with open(filename, 'r') as file:
        self.log = json.load(file)
```

Hình 9 Phương thức save_log_to_file(self, filename) và load_log_to_file(self, filename)

- self: Tham chiếu đến đối tượng hiện tại của lớp mà phương thức này thuộc về. Nó cho phép truy cập vào các thuộc tính và phương thức của đối tượng.
- filename: Tên tệp tin mà dữ liệu sẽ được lưu vào. Tham số này được truyền vào từ bên ngoài khi phương thức được gọi.
- open(filename, 'w'): Mở tệp tin với chế độ ghi ('w'), có nghĩa là nếu tệp tin đã tồn tại, nó sẽ bị ghi đè. Nếu tệp tin chưa tồn tại, một tệp mới sẽ được tạo ra.
- json.dump(self.log, file): Dùng phương thức json.dump() để chuyển đổi đối tượng self.log (giả sử là một danh sách hoặc từ điển) thành chuỗi JSON và lưu vào tệp tin file. Dữ liệu trong self.log sẽ được ghi vào tệp.

- open(filename, 'r'): Mở tệp tin với chế độ đọc ('r'), nghĩa là chương trình sẽ chỉ đọc dữ liệu từ tệp này.
- self.log = json.load(file): Dùng phương thức json.load() để đọc dữ liệu JSON từ tệp file và chuyển đổi nó thành đối tượng Python (ví dụ: danh sách hoặc từ điển). Dữ liệu này sau đó được gán lại cho thuộc tính self.log, thay thế dữ liệu hiện tại trong self.log.

Lớp TSPApp: xây dựng giao diện người dùng với các chức năng điều khiển chương trình.

- Thành phần giao diện:

+ Khung điều khiển (Control frame): nhập số lượng thành phố, thêm tọa độ, và chọn thuật toán.

```
self.control_frame = tk.Frame(self.main_frame, padx=10, pady=10)
self.control_frame.pack(side=tk.LEFT, fill=tk.Y)
```

Hình 10 Khung điều khiển

- self: tham chiếu đến đối tượng hiện tại.
- self.main_frame: widget cha chứa self.control_frame.
- tk.Frame(self.main_frame, padx=10, pady=10): tạo một Frame con trong self.main_frame với khoảng cách padding 10 pixel cả hai chiều.
- self.control_frame.pack(side=tk.LEFT, fill=tk.Y): đặt self.control_frame vào bên trái của self.main_frame và kéo dài nó theo chiều dọc.

+ Khung đồ họa (Canvas): vẽ các thành phố và đường đi.

```
self.canvas_frame = tk.Frame(self.main_frame)
self.canvas_frame.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True, padx=5, pady=5)
```

Hình 11 Khung đồ họa

- self: tham chiếu đến đối tượng hiện tại.
- self.main_frame: widget cha chứa self.canvas_frame.
- tk.Frame(self.main_frame): tạo một Frame con trong self.main_frame.
- side=tk.RIGHT: đặt self.canvas_frame vào bên phải của self.main_frame.
- fill=tk.BOTH: kéo dài self.canvas_frame theo cả chiều ngang và dọc.

- expand=True: cho phép self.canvas_frame mở rộng để chiếm không gian trống còn lại trong self.main_frame.
- padx=5, pady=5: thêm khoảng cách 5 pixel xung quanh self.canvas_frame.

+ Khung hiển thị log (Log display frame): hiển thị thông tin log kết quả.

```
self.log_display_frame = tk.Frame(self.canvas_frame)
self.log_display_frame.pack(fill=tk.X, padx=5, pady=5)
```

Hình 12 Khung hiển thị log

- self: tham chiếu đến đối tượng hiện tại.
- self.canvas_frame: widget cha chứa self.log_display_frame.
- tk.Frame(self.canvas_frame): tạo một Frame con trong self.canvas_frame.
- fill=tk.X: kéo dài self.log_display_frame theo chiều ngang (chiếm toàn bộ chiều rộng của self.canvas_frame).
- padx=5, pady=5: thêm khoảng cách 5 pixel xung quanh self.log_display_frame.

- Các hàm giao diện chính:

+ generate_cities(): tạo thành phố ngẫu nhiên.

```
def generate_cities(self):
    try:
        n = int(self.city_count_entry.get())
        if n <= 0:
            raise ValueError("Số thành phố phải lớn hơn 0.")
        self.solver.generate_random_cities(n)
        self.draw_cities()
    except ValueError as e:
        messagebox.showerror("Lỗi", str(e))
```

Hình 13 Hàm generate_cities(self)

- self: tham chiếu đến đối tượng hiện tại của lớp mà phương thức này thuộc về.
- n: số lượng thành phố mà người dùng muốn tạo ra, được lấy từ trường nhập liệu (entry box) trong giao diện người dùng.
- self.city_count_entry.get(): lấy giá trị mà người dùng đã nhập vào trong trường nhập liệu.
- int(): chuyển đổi giá trị chuỗi (string) nhận được từ trường nhập liệu thành kiểu dữ liệu số nguyên (integer).

- if n <= 0: kiểm tra xem giá trị n có hợp lệ không. Nếu n nhỏ hơn hoặc bằng 0, một lỗi sẽ được ném ra, yêu cầu người dùng nhập số lớn hơn 0.
- raise ValueError("Số thành phố phải lớn hơn 0."): ném ra lỗi ValueError nếu giá trị n không hợp lệ, với thông báo lỗi là "Số thành phố phải lớn hơn 0."
- self.solver.generate_random_cities(n): gọi phương thức generate_random_cities từ đối tượng solver để tạo ra n thành phố ngẫu nhiên.
- self.draw_cities(): vẽ các thành phố đã được tạo ra lên giao diện người dùng (có thể là bản đồ hoặc không gian đồ họa).
- except ValueError as e: bắt lỗi ValueError nếu có sự cố trong việc lấy giá trị hoặc xử lý đầu vào không hợp lệ.
- messagebox.showerror("Lỗi", str(e)): hiển thị một hộp thoại thông báo lỗi với tiêu đề "Lỗi", chứa thông báo lỗi cụ thể từ ngoại lệ e.

+ add_city_from_input(): thêm thành phố từ tọa độ người dùng nhập.

```
def add_city_from_input(self):
    try:
        coordinates = self.coordinates_text.get("1.0", tk.END).strip().split("\n")
        for line in coordinates:
            x, y = map(int, line.split(","))
            self.solver.add_city(x, y)
        self.solver.calculate_distance_matrix()
        self.draw_cities()
    except Exception as e:
        messagebox.showerror("Lỗi", "Tọa độ không hợp lệ.")
```

Hình 14 Hàm add_city_from_input(self)

- self: tham chiếu đến đối tượng hiện tại của lớp mà phương thức này thuộc về.
- coordinates_text: trường nhập liệu chứa các tọa độ thành phố dưới dạng văn bản trong giao diện người dùng.
- get("1.0", tk.END): lấy toàn bộ văn bản từ vị trí bắt đầu (1.0) đến kết thúc (tk.END) trong trường nhập liệu coordinates_text.
- strip(): loại bỏ khoảng trắng thừa ở đầu và cuối văn bản.
- split("\n"): chia văn bản thành các dòng riêng biệt dựa trên ký tự xuống dòng.
- for line in coordinates:: duyệt qua từng dòng (mỗi dòng chứa tọa độ của một thành phố).

- x, y = map(int, line.split(",")): chia mỗi dòng thành hai phần (tọa độ x và y), rồi chuyển chúng thành kiểu số nguyên.
- self.solver.add_city(x, y): thêm một thành phố với tọa độ (x, y) vào đối tượng solver.
- self.solver.calculate_distance_matrix(): tính toán ma trận khoảng cách giữa các thành phố.
- self.draw_cities(): vẽ các thành phố lên giao diện.
- except Exception as e:: bắt lỗi nếu có sự cố trong quá trình xử lý.
- messagebox.showerror("Lỗi", "Tọa độ không hợp lệ."): hiển thị hộp thoại lỗi nếu tọa độ không hợp lệ.

+ solve(): chạy thuật toán và hiển thị kết quả.

```
def solve(self):
    if not self.solver.cities:
        messagebox.showwarning("Cảnh báo", "Chưa có dữ liệu thành phố.")
        return
    path, cost = self.solver.nearest_neighbor()
    self.animate_path(path)
    self.update_log_display()
    messagebox.showinfo("Kết quả", f"Đường đi: {path}\nChi phí: {cost}")
```

Hình 15 Hàm solve(self)

- self: tham chiếu đến đối tượng hiện tại của lớp mà phương thức này thuộc về.
- self.solver.cities: kiểm tra danh sách các thành phố trong đối tượng solver. Nếu không có thành phố nào, tiếp tục thực hiện cảnh báo.
- messagebox.showwarning("Cảnh báo", "Chưa có dữ liệu thành phố."): hiển thị hộp thoại cảnh báo nếu không có thành phố nào được thêm vào.
- return: dừng thực thi phương thức nếu không có thành phố nào.
- self.solver.nearest_neighbor(): gọi phương thức nearest_neighbor() của đối tượng solver để tính toán đường đi gần nhất và chi phí tương ứng. Trả về hai giá trị: path (đường đi) và cost (chi phí).
- self.animate_path(path): hiển thị hoặc vẽ đường đi (path) trong giao diện người dùng (có thể là hoạt ảnh).
- self.update_log_display(): cập nhật thông tin trong giao diện người dùng (có thể là một bảng log hoặc bảng thông tin).

- messagebox.showinfo("Kết quả", f"Đường đi: {path}\nChi phí: {cost}"): hiển thị hộp thoại thông báo kết quả, bao gồm đường đi và chi phí.

+ show_matrices(): hiển thị ma trận khoảng cách và trọng số.

```
def show_matrices(self):
    if not self.solver.distance_matrix:
        messagebox.showwarning("Cảnh báo", "Chưa có dữ liệu thành phố.")
        return

    distance_text = "Ma trận khoảng cách:\n" + "\n".join(["\t".join(map(lambda x: f"{x:.2f}", row)) for row in self.solver.distance_matrix])
    weight_text = "\nMa trận trọng số:\n" + "\n".join(["\t".join(map(lambda x: f"{x:.2f}", row)) for row in self.solver.weight_matrix])
    messagebox.showinfo("Ma trận", distance_text + weight_text)
```

Hình 16 Hàm show_matrices(self)

- self: tham chiếu đến đối tượng hiện tại của lớp mà phương thức này thuộc về.
- self.solver.distance_matrix: kiểm tra nếu ma trận khoảng cách chưa được tính toán (danh sách rỗng hoặc None), tiếp tục thực hiện cảnh báo.
- messagebox.showwarning("Cảnh báo", "Chưa có dữ liệu thành phố."): hiển thị hộp thoại cảnh báo nếu chưa có ma trận khoảng cách.
- return: dừng thực thi phương thức nếu không có ma trận khoảng cách.
- distance_text = "Ma trận khoảng cách:\n" + "\n".join([...]): tạo một chuỗi văn bản để hiển thị ma trận khoảng cách. Mỗi hàng của ma trận sẽ được chuyển thành chuỗi, với các giá trị được định dạng dưới dạng số thực với 2 chữ số thập phân ({x:.2f}).
- weight_text = "\nMa trận trọng số:\n" + "\n".join([...]): tạo một chuỗi văn bản tương tự để hiển thị ma trận trọng số.
- messagebox.showinfo("Ma trận", distance_text + weight_text): hiển thị hộp thoại thông báo chứa thông tin về cả ma trận khoảng cách và ma trận trọng số.

+ refresh_app(): làm mới ứng dụng, xóa dữ liệu cũ.

```
def refresh_app(self):
    self.solver = TSPSolver()
    self.canvas.delete("all")
    self.city_count_entry.delete(0, tk.END)
    self.coordinates_text.delete("1.0", tk.END)
    self.log_tree.delete(*self.log_tree.get_children())
```

Hình 17 Hàm refresh_app(self)

- self: Tham chiếu đến đối tượng hiện tại của lớp mà phương thức này thuộc về.

- self.solver = TSPSolver(): tạo lại một đối tượng mới của lớp TSPSolver, khởi tạo lại giải pháp cho bài toán TSP.
- self.canvas.delete("all"): xóa tất cả các đối tượng vẽ trên canvas (bảng vẽ) trong giao diện người dùng.
- self.city_count_entry.delete(0, tk.END): xóa nội dung đã nhập trong trường nhập liệu city_count_entry (để nhập lại số lượng thành phố).
- self.coordinates_text.delete("1.0", tk.END): xóa toàn bộ nội dung trong trường nhập liệu chứa tọa độ các thành phố.
- self.log_tree.delete(*self.log_tree.get_children()): xóa tất cả các mục con trong cây (tree) hiển thị log thông tin.

Bước 3: triển khai thuật toán "người hàng xóm gần nhất"

Hàm nearest_neighbor() trong lớp TSPSolver thực hiện như sau:

1. Bắt đầu từ một thành phố cố định (ví dụ, thành phố 0).
2. Tạo danh sách các thành phố chưa ghé thăm.
3. Tìm thành phố gần nhất chưa ghé thăm và thêm vào đường đi.
4. Lặp lại cho đến khi ghé thăm tất cả các thành phố.
5. Quay lại thành phố ban đầu, tính tổng chi phí đường đi.

Bước 4: tạo giao diện

Chương trình sử dụng Tkinter để tạo giao diện trực quan với các thành phần chính:

- Khung nhập liệu (Control frame):
  + Nhập số lượng thành phố hoặc tọa độ cụ thể.
  + Chọn thuật toán và chạy giải pháp.
- Khung đồ họa (Canvas):
  + Hiển thị vị trí các thành phố.
  + Vẽ đường đi tối ưu bằng các đường nối.

- Khung kết quả (Log display frame): hiển thị các kết quả chạy thuật toán như tên thuật toán, thời gian thực hiện, chi phí, và đường đi.

Bước 5: xử lý log và hoạt hình

- Lưu và tải log: log được lưu dưới dạng JSON, giúp người dùng dễ dàng xem lại kết quả.
- Hoạt hình: hàm animate_path(path) vẽ từng bước đường đi trên canvas, tạo hiệu ứng trực quan.

Bước 6: chạy chương trình

Khi chạy file, chương trình sẽ hiển thị giao diện trực quan. Người dùng có thể:

1. Nhập số lượng thành phố hoặc tọa độ.
2. Chạy thuật toán "Người hàng xóm gần nhất".
3. Xem kết quả trực tiếp trên đồ họa.

CHƯƠNG 4: KẾT QUẢ NGHIÊN CỨU

4.1 Hiệu năng

4.1.1 Thuật toán sử dụng

- Thuật toán Nearest neighbor có thời gian thực thi nhanh do tính đơn giản, phù hợp với các tập dữ liệu vừa và nhỏ.
- Với bộ dữ liệu có kích thước lớn, thuật toán có thể không tìm được đường đi tối ưu nhưng vẫn đảm bảo kết quả chấp nhận được trong thời gian ngắn.

4.1.2 Thời gian thực thi

- Thời gian thực thi được đo đạc và hiển thị, giúp người dùng đánh giá được hiệu năng.
- Ma trận khoảng cách và trọng số được tính toán một cách nhanh chóng, sử dụng các thư viện tối ưu như NumPy.

4.2 Trải nghiệm người dùng

Dễ sử dụng:

- Giao diện thân thiện, dễ hiểu, với các nút chức năng rõ ràng.
- Người dùng có thể thêm tọa độ thủ công hoặc tự động tạo ngẫu nhiên các thành phố để kiểm tra.

Trực quan:

- Hiển thị rõ ràng các kết quả, bao gồm đường đi, chi phí, và thời gian thực thi.
- Tích hợp bảng log để lưu lại thông tin các lần chạy trước, thuận tiện cho việc phân tích.

Tương tác tốt:

- Tính năng hoạt hình mô phỏng trực quan, sinh động quá trình tìm đường đi.

- Người dùng có thể bật/tắt hoạt hình để tăng tốc độ thực thi hoặc giữ lại yếu tố mô phỏng.

4.3 Giao diện chức năng

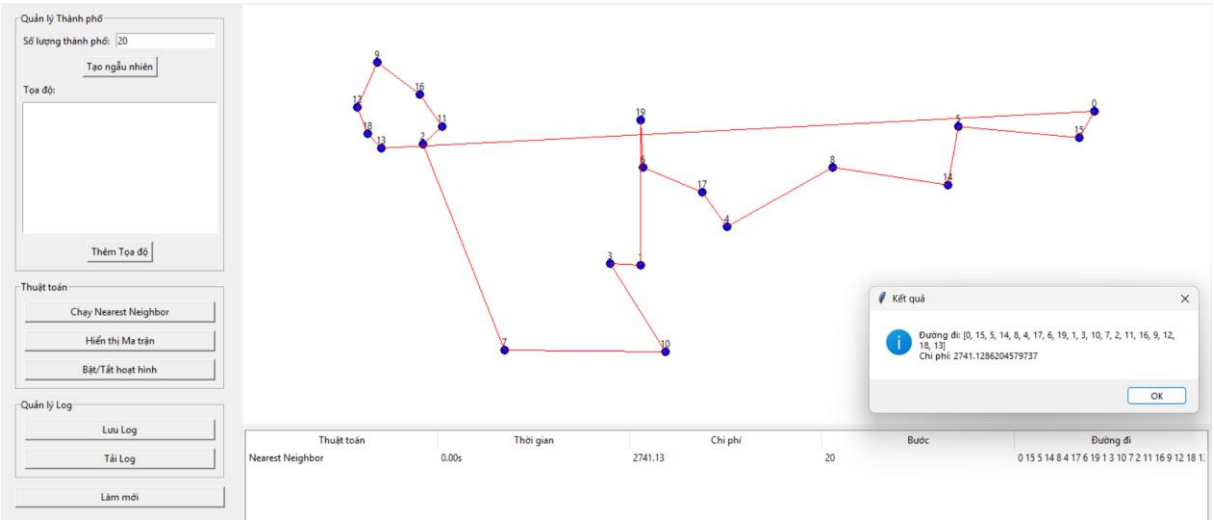
Quản lý thành phố:

- Nhập tọa độ thành phố thủ công hoặc tạo ngẫu nhiên.
- Hiển thị danh sách các thành phố trên giao diện canvas với vị trí tương ứng.

Hình 18 Giao diện Quản lý thành phố

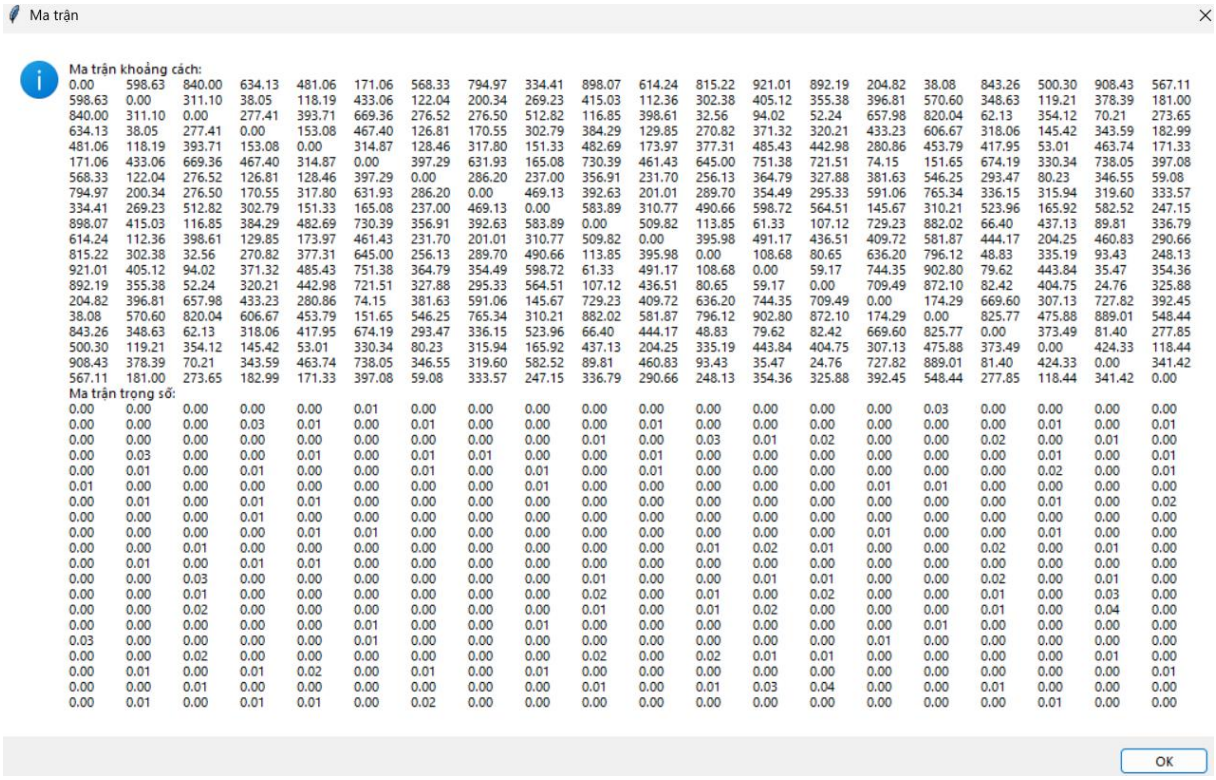
Chạy thuật toán:

- Nút "Chạy Nearest Neighbor" thực hiện tìm đường đi và hiển thị kết quả bằng cả đồ họa và văn bản.



Hình 19 Giao diện sau khi nhấn nút "Chạy Nearest Neighbor"

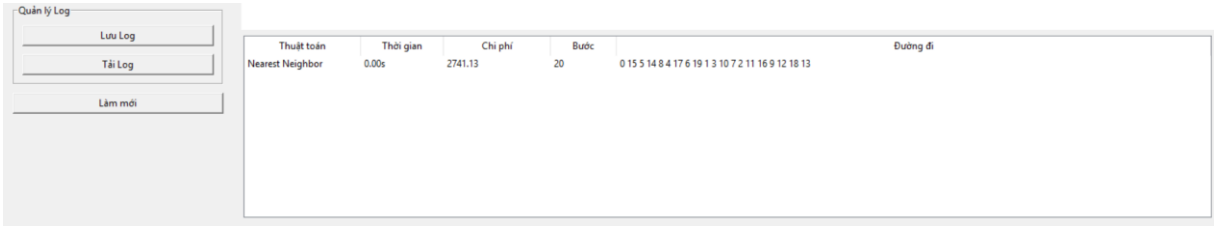
- Các ma trận khoảng cách và trọng số cũng có thể được hiển thị.



Hình 20 Giao diện hiển thị ma trận

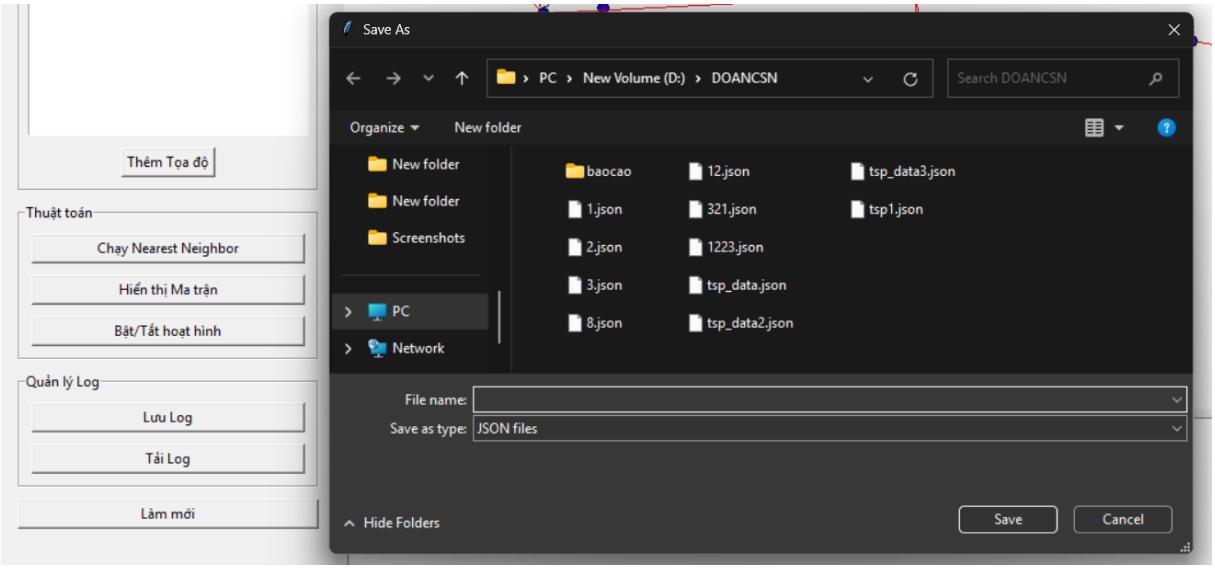
Quản lý log:

- Ghi lại chi tiết các lần chạy: tên thuật toán, thời gian, chi phí, bước đi, và đường đi.

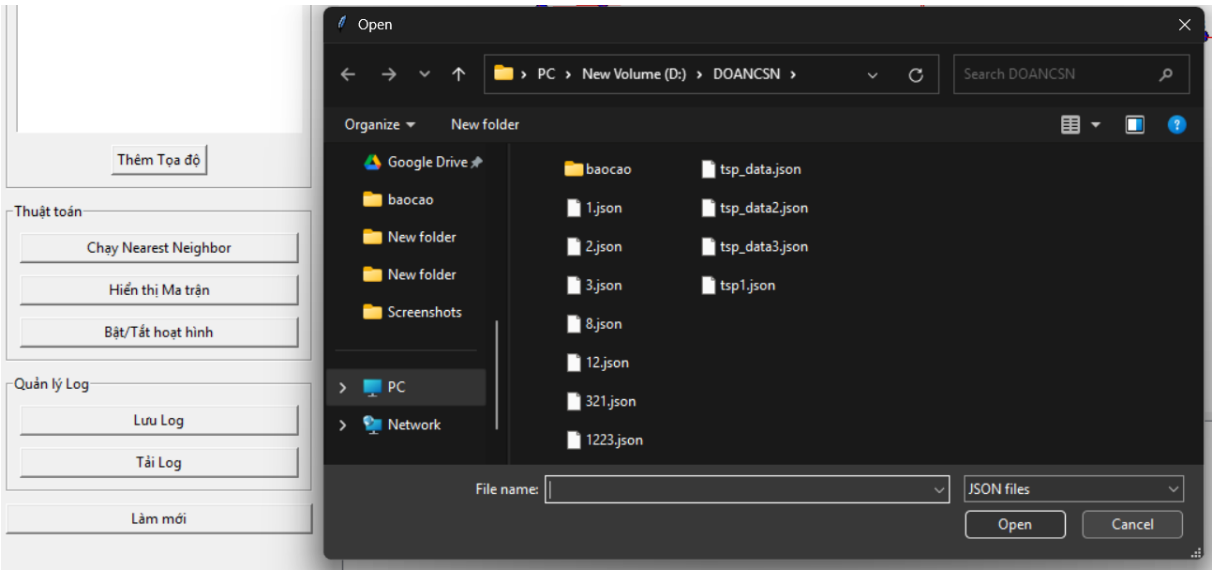


Hình 21 Giao diện log

- Tính năng lưu và tải log giúp quản lý dữ liệu hiệu quả.



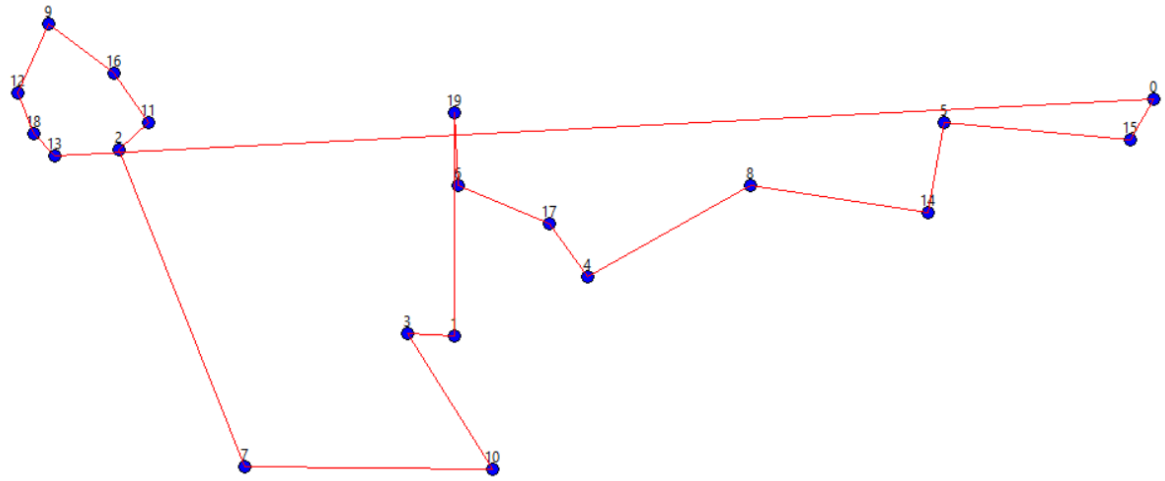
Hình 22 Giao diện lưu log



Hình 23 Giao diện tải log

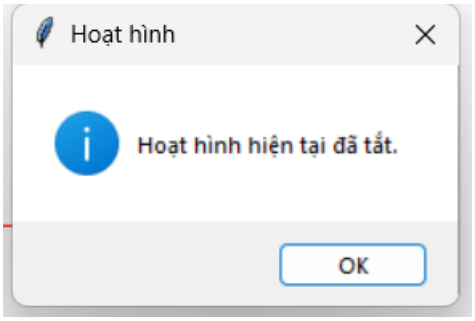
Tùy chỉnh hiển thị:

- Chế độ hoạt hình cho phép xem trực quan đường đi được xây dựng.

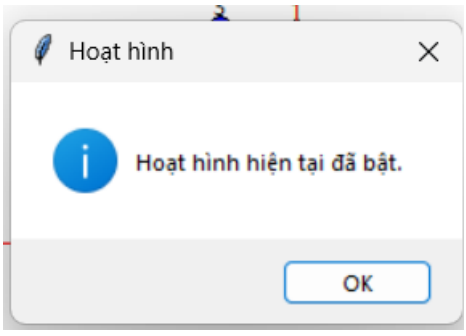


Hình 24 Giao diện sau khi đã được thực hiện hoàn tất

- Chuyển đổi nhanh giữa bật/tắt hoạt hình.



Hình 25 Thông báo đã tắt hoạt hình

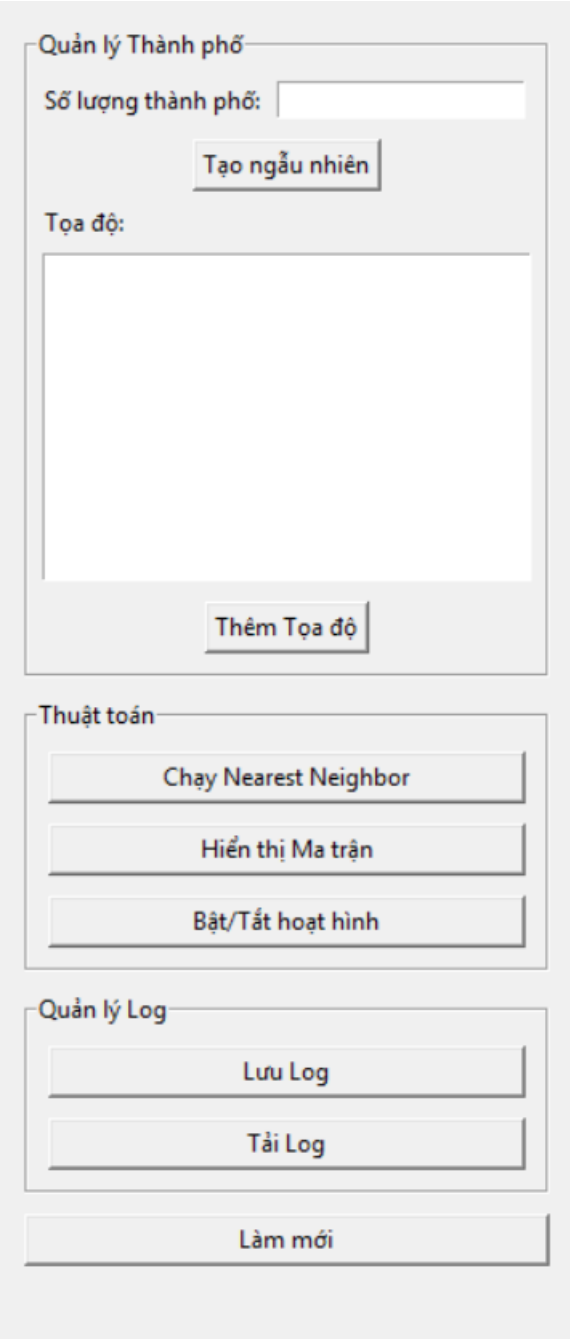


Hình 26 Thông báo đã bật hoạt hình

4.4 Hình minh họa

Giao diện chính:

- Khung bên trái: Các chức năng quản lý thành phố, chạy thuật toán và hiển thị log.



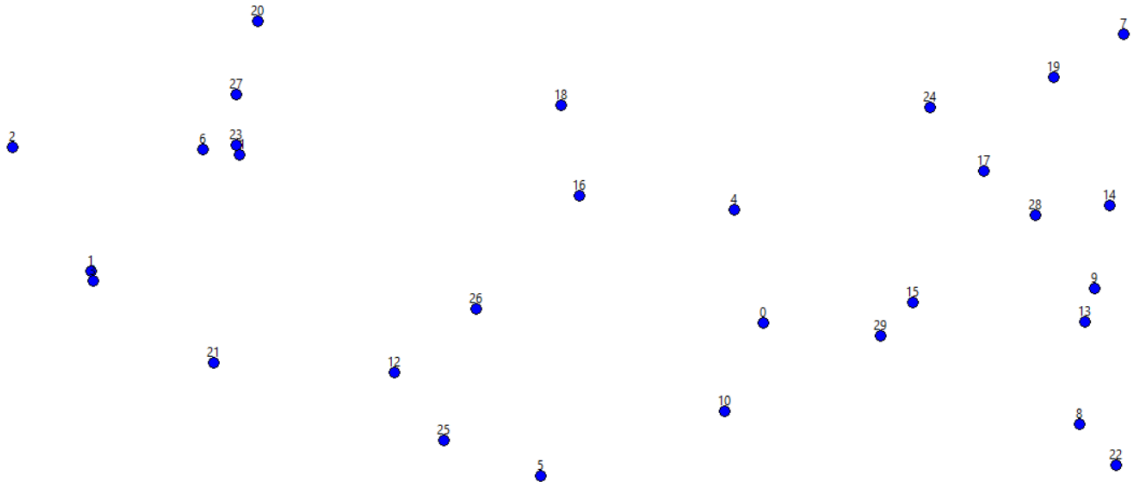
Hình 27 Khung bên trái

- Khung bên phải: Vùng canvas hiển thị đồ họa, gồm các thành phố và đường đi được vẽ.

Hình 28 Khung bên phải

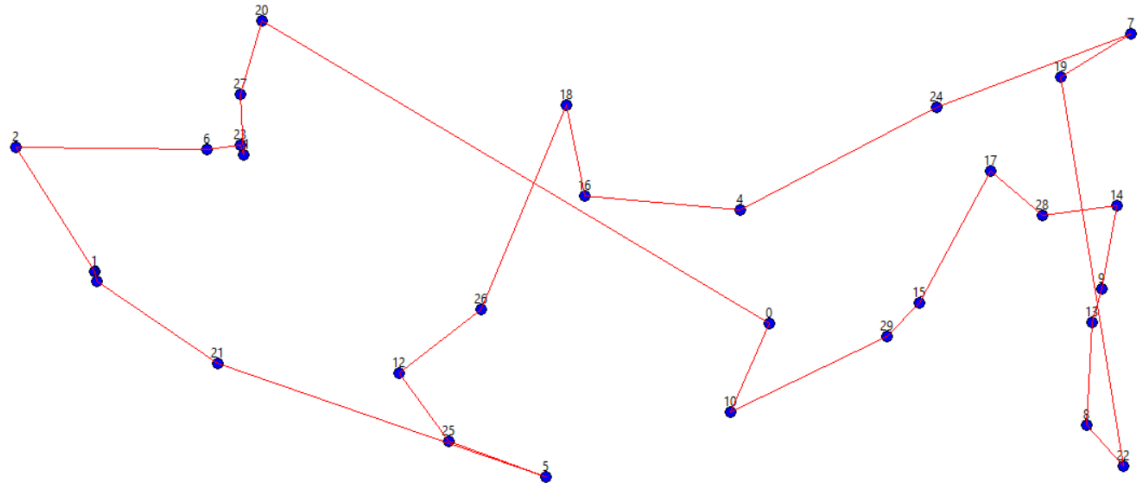
Minh họa đường đi:

- Các thành phố được biểu diễn bằng các vòng tròn nhỏ màu xanh.



Hình 29 Các thành phố được biểu diễn

- Đường đi được nối bằng các đoạn thẳng màu đỏ, mô phỏng quá trình di chuyển.



Hình 30 Đường đi được biểu diễn

CHƯƠNG 5: KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

5.1 Kết luận

Kết quả đạt được: đồ án đã xây dựng thành công một ứng dụng mô phỏng bài toán người du lịch (TSP) với giao diện đồ họa trực quan, hỗ trợ nhập dữ liệu linh hoạt và hiển thị kết quả chi tiết.

Đóng góp mới: ứng dụng tích hợp tính năng hoạt hình mô phỏng quá trình tìm đường đi, khả năng quản lý log kết quả, và cung cấp công cụ hỗ trợ phân tích ma trận khoảng cách và trọng số.

Đề xuất mới: chương trình cung cấp nền tảng để áp dụng thuật toán Nearest neighbor và mở rộng hỗ trợ các thuật toán khác cho bài toán TSP trong tương lai.

5.2 Hướng phát triển

Mở rộng thuật toán: tích hợp thêm các thuật toán tối ưu như Genetic algorithm, Simulated annealing, hoặc Ant colony optimization để cải thiện chất lượng kết quả.

Cải thiện giao diện:

- Phát triển giao diện web hoặc ứng dụng đa nền tảng để tăng tính tiện lợi.
- Tối ưu giao diện người dùng với đồ họa tương tác, ví dụ kéo-thả thành phố.

Hỗ trợ dữ liệu thực tế: mở rộng tính năng nhập dữ liệu từ các tệp bản đồ thực tế hoặc kết nối API để tải dữ liệu địa lý.

Phân tích nâng cao:

- Cung cấp báo cáo phân tích về hiệu năng thuật toán với các bộ dữ liệu có kích thước khác nhau.
- Tích hợp công cụ so sánh hiệu quả giữa các thuật toán.

Tăng hiệu suất:

- Sử dụng các thư viện tính toán hiệu năng cao để tối ưu thời gian thực thi trên dữ liệu lớn.
- Hỗ trợ xử lý đa luồng hoặc GPU để cải thiện tốc độ tính toán.

- Khả năng ứng dụng và hỗ trợ người dùng trong các bài toán tối ưu hóa thực tế.

DANH MỤC TÀI LIỆU THAM KHẢO

[1] R. Diestel, Graph Theory: An Introduction to Proofs, Algorithms, and Applications, New York: McGraw Hill, 2021.

[2] K. H. Rosen, Discrete Mathematics and Its Applications, New York: McGraw Hill, 2011.

[3] W3Schools, "Traveling Salesman Problem (TSP) Reference," [Trực tuyến]. Available: https://www.w3schools.com/dsa/dsa_ref_traveling_salesman.php. [Đã truy cập 23 12 2024].

[4] P. S. Foundation, "tkinter — Python interface to Tcl/Tk," [Trực tuyến]. Available: https://docs.python.org/3.12/library/tkinter.html. [Đã truy cập 2 12 2024].