

NHẬN XÉT CỦA GIẢNG VIÊN HƯỚNG DẪN

This image shows a full page of white paper with horizontal dotted lines. The lines are evenly spaced and run across the width of the page, providing a guide for handwriting or typing. There are no margins, text, or other markings on the page.

This image shows a full page of white paper with horizontal dotted lines. The lines are evenly spaced and run across the width of the page, providing a guide for handwriting practice. There are no margins, text, or other markings on the page.

LỜI CẢM ƠN

Kính gửi thầy Trần 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 Trần 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

MỤC LỤC	5
DANH MỤC HÌNH ẢNH.....	8
DANH MỤC BẢNG BIỂU.....	8
TÓM TẮT ĐỒ ÁN CƠ SỞ NGÀNH.....	9
MỞ ĐẦU	11
CHƯƠNG 1: TỔNG QUAN	13
1.1 Mô tả bài toán người du lịch.....	13
1.2 Đặc điểm bài toán người du lịch.....	13
1.3 Các phương pháp giải quyết bài toán người du lịch.....	14
1.4 Mục tiêu và phạm vi nghiên cứu	14
1.5 Ý nghĩa và ứng dụng.....	15
CHƯƠNG 2: NGHIÊN CỨU LÝ THUYẾT	16
2.1 Cơ sở lý thuyết.....	16
2.2 Lý luận	16
2.3 Giả thuyết khoa học	17
2.4 Phương pháp nghiên cứu	17
2.4.1 Mô hình hóa bài toán	17
2.4.2 Lựa chọn thuật toán.....	17
2.4.3 Xây dựng giao diện người dùng.....	18
2.4.4 Thử nghiệm và phân tích kết quả.....	18
2.4.5 Đánh giá và tổng kết	18
CHƯƠNG 3: HIỆN THỰC HÓA NGHIÊN CỨU	19
3.1 Mô tả các bước nghiên cứu đã tiến hành	19
3.1.1 Xác định bài toán và mô hình hóa.....	19

3.1.2	Lựa chọn thuật toán.....	19
3.1.3	Thiết kế hệ thống.....	19
3.1.4	Cài đặt chương trình.....	19
3.1.5	Thử nghiệm và đánh giá.....	19
3.2	Các bản thiết kế.....	20
3.2.1	Lược đồ Use case	20
3.2.2	Lược đồ Class.....	20
3.2.3	Lược đồ hoạt động	21
3.3	Cách cài đặt chương trình	22
3.3.1	Cấu trúc thư mục dự án	22
3.3.2	Mô tả các tệp tin và chức năng.....	22
3.3.3	Nội dung tệp tin tsp_algorithms.py	22
3.3.4	Nội dung tệp tin tsp_gui.py	29
3.3.5	Nội dung tệp tin main.py.....	41
3.4	Hồ sơ thiết kế và cài đặt.....	41
3.4.1	Thiết kế hệ thống.....	41
3.4.2	Cài đặt chương trình.....	42
3.4.3	Thử nghiệm và đánh giá.....	42
CHƯƠNG 4: KẾT QUẢ NGHIÊN CỨU		43
4.1	Hiệu năng của các thuật toán	43
4.2	Trải nghiệm người dùng	44
4.3	Giao diện chức năng	44
4.4	Hình ảnh minh họa.....	47
CHƯƠNG 5: KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN		49
5.1	Kết luận.....	49

5.1.1	Kết quả đạt được	49
5.1.2	Đóng góp mới.....	49
5.1.3	Đề xuất mới.....	49
5.2	Hướng phát triển	49
DANH MỤC TÀI LIỆU THAM KHẢO		51

DANH MỤC HÌNH ẢNH

Hình 1 Giao diện chính của chương trình	45
Hình 2 Khung nhập liệu	45
Hình 3 Khung chọn thuật toán.....	46
Hình 4 Khung kết quả log	46
Hình 5 Các nút chức năng	47
Hình 6 Giao diện chính của chương trình khi khởi động	47
Hình 7 Kết quả mô phỏng thuật toán Nearest Neighbor	48
Hình 8 Kết quả mô phỏng thuật toán 2-opt	48

DANH MỤC BẢNG BIỂU

Bảng 1 Nội dung tệp tin tsp_algorithms.py.....	29
Bảng 2 Nội dung tệp tin tsp_gui.py.....	41
Bảng 3 Nội dung tệp tin main.py.....	41

TÓM TẮT ĐỒ ÁN CƠ SỞ NGÀNH

1. Vấn đề nghiên cứu

Bài toán người du lịch (TSP - Travelling Salesman Problem) là một bài toán tối ưu trong lý thuyết đồ thị, yêu cầu tìm ra một chu trình đi qua tất cả các thành phố một lần và chỉ một lần, quay lại thành phố xuất phát sao cho tổng chi phí (hoặc khoảng cách) là nhỏ nhất. Đây là một bài toán NP-hard, tức là không có thuật toán tối ưu hóa nhanh chóng cho tất cả các trường hợp.

2. Các hướng tiếp cận

Để giải quyết bài toán TSP, đồ án sử dụng 4 thuật toán phổ biến:

- Brute Force (Tất cả hoán vị): Duyệt qua tất cả các hoán vị của các thành phố để tính toán tổng chi phí, tìm ra chu trình có chi phí nhỏ nhất.
- Nearest Neighbor (Láng giềng gần nhất): Chọn thành phố gần nhất chưa được thăm và tiếp tục cho đến khi tất cả thành phố được thăm.
- 2-opt: Cải tiến đường đi ban đầu bằng cách hoán đổi các đoạn đường để giảm chi phí.
- Backtracking (Quay lui): Thử tất cả các cách đi, sử dụng phương pháp quay lui và cắt tỉa để giảm số lượng khả năng phải thử.

3. Cách giải quyết vấn đề

Các thuật toán trên được triển khai trong một ứng dụng giao diện đồ họa sử dụng Tkinter, cho phép người dùng mô phỏng bài toán với các thành phố ngẫu nhiên hoặc tự nhập tọa độ các thành phố. Các bước giải quyết cụ thể như sau:

- Tạo thành phố: Người dùng có thể nhập số lượng thành phố, hoặc tải danh sách thành phố từ tệp tin.
- Tạo ma trận khoảng cách: Tính toán ma trận khoảng cách giữa các thành phố.
- Chọn thuật toán: Người dùng chọn thuật toán giải quyết từ các tùy chọn có sẵn.

- Mô phỏng và hiển thị kết quả: Chạy thuật toán đã chọn và hiển thị kết quả trên giao diện, bao gồm chi phí và thời gian tính toán. Đặc biệt, có thể bật/tắt hoạt hình để theo dõi quá trình đi qua các thành phố.
- Lưu và tải kết quả: Người dùng có thể lưu lại kết quả và lịch sử các lần chạy thuật toán vào tệp JSON, hoặc tải lại dữ liệu đã lưu.

4. Một số kết quả đạt được

- Thành công trong việc triển khai giao diện đồ họa: Người dùng có thể tương tác dễ dàng với phần mềm, nhập liệu, chọn thuật toán và xem kết quả.
- Tính năng mô phỏng và hoạt hình: Chương trình không chỉ tính toán kết quả mà còn cho phép mô phỏng quá trình di chuyển giữa các thành phố, hỗ trợ cả việc hiển thị lộ trình và tiết kiệm thời gian cho người dùng.
- Lưu và tải kết quả: Chương trình hỗ trợ lưu kết quả dưới dạng tệp JSON, giúp người dùng có thể tiếp tục công việc từ điểm đã lưu.

MỞ ĐẦU

1. Lý do chọn đề tài:

Đề tài "Viết chương trình mô phỏng bài toán người du lịch (TSP - Traveling Salesman Problem)" được chọn vì bài toán người du lịch là một trong những bài toán cổ điển trong lĩnh vực tối ưu hóa và lý thuyết đồ thị. Bài toán này không chỉ có ý nghĩa lý thuyết mà còn có ứng dụng rộng rãi trong nhiều lĩnh vực thực tiễn như logistics, giao thông, quản lý chuỗi cung ứng và các vấn đề tối ưu hóa lộ trình. Việc xây dựng một chương trình mô phỏng để giải quyết bài toán TSP giúp cải thiện kỹ năng lập trình, hiểu rõ hơn về các thuật toán tối ưu hóa, đồng thời áp dụng lý thuyết vào thực tế để giải quyết vấn đề phức tạp.

2. Mục đích nghiên cứu:

Mục đích của đồ án là thiết kế và phát triển một chương trình mô phỏng bài toán người du lịch nhằm:

- Nghiên cứu và áp dụng các thuật toán giải quyết bài toán người du lịch: Brute Force, Thuật toán gần nhất (Nearest Neighbor), 2-opt và Backtracking.
- Cung cấp công cụ mô phỏng để người dùng có thể thử nghiệm và so sánh hiệu quả của các thuật toán này.
- Xây dựng giao diện người dùng dễ sử dụng, trực quan, hỗ trợ người dùng nhập liệu, xem kết quả và lưu trữ thông tin.
- Phân tích hiệu quả của các thuật toán trong các trường hợp khác nhau thông qua các thống kê và log kết quả.

3. Đối tượng nghiên cứu:

Đối tượng nghiên cứu của đồ án là bài toán người du lịch, trong đó một người du lịch cần phải đi qua tất cả các thành phố mà không quay lại một thành phố nào và quay về điểm xuất phát, sao cho tổng quãng đường di chuyển là ngắn nhất. Các thuật toán tối ưu hóa áp dụng trong nghiên cứu bao gồm:

- Brute Force: Thử tất cả các hoán vị của các thành phố để tìm ra lộ trình ngắn nhất.
- Nearest Neighbor: Chọn thành phố gần nhất chưa được thăm để tiếp tục chuyển đi.
- 2-opt: Tối ưu hóa lộ trình ban đầu bằng cách đảo ngược các đoạn đường.
- Backtracking: Khám phá các lộ trình có thể một cách đệ quy và áp dụng cắt tỉa để loại bỏ các lộ trình không khả thi.

4. Phạm vi nghiên cứu:

- Phạm vi về thuật toán: Các thuật toán được áp dụng để giải quyết bài toán TSP trong phạm vi bài toán có số lượng thành phố nhỏ đến trung bình (thường từ 5 đến 50 thành phố). Các thuật toán sẽ được triển khai đầy đủ và tối ưu hóa cho từng phương pháp.
- Phạm vi về ứng dụng: Xây dựng một chương trình giao diện người dùng (GUI) bằng Python và thư viện Tkinter, cho phép người dùng nhập liệu, chọn thuật toán, thực hiện mô phỏng và xem kết quả. Chương trình cũng hỗ trợ người dùng lưu trữ và tải lại dữ liệu về thành phố và kết quả mô phỏng dưới định dạng JSON.
- Phạm vi về mô phỏng: Mô phỏng lộ trình của người du lịch, hiển thị các thành phố và các kết nối giữa chúng trên một bản đồ trực quan. Người dùng có thể quan sát quá trình tìm kiếm lộ trình ngắn nhất của các thuật toán thông qua đồ thị.

Với phạm vi này, đồ án sẽ cung cấp một cái nhìn tổng quan về cách giải quyết bài toán người du lịch bằng các thuật toán khác nhau, cũng như tạo ra một công cụ hữu ích cho việc nghiên cứu và ứng dụng trong các tình huống thực tế.

CHƯƠNG 1: TỔNG QUAN

Bài toán người du lịch (hay còn gọi là Traveling Salesman Problem - TSP) là một trong những bài toán cổ điển và nổi bật trong lĩnh vực tối ưu hóa và lý thuyết đồ thị. Đây là một bài toán có tính ứng dụng rộng rãi trong nhiều lĩnh vực, đặc biệt trong logistics, quản lý chuỗi cung ứng, giao thông vận tải, và các bài toán tối ưu hóa lộ trình khác.

1.1 Mô tả bài toán người du lịch

Bài toán người du lịch đặt ra một vấn đề đơn giản nhưng lại rất phức tạp khi giải quyết: một người du lịch cần phải thăm tất cả các thành phố trong một tập hợp, mỗi thành phố chỉ được thăm một lần, và sau cùng phải quay lại thành phố xuất phát sao cho tổng quãng đường di chuyển là ngắn nhất. Với mỗi thành phố, có một khoảng cách nhất định giữa chúng, và mục tiêu là tìm ra một lộ trình có tổng quãng đường ngắn nhất.

Trong bài toán người du lịch, bài toán có thể được mô tả như một đồ thị, trong đó các thành phố là các đỉnh, và các đoạn đường nối các thành phố là các cạnh với trọng số là khoảng cách giữa các thành phố. Mục tiêu của bài toán là tìm ra chu trình Hamilton (một chu trình thăm tất cả các đỉnh đúng một lần) có tổng trọng số nhỏ nhất.

1.2 Đặc điểm bài toán người du lịch

Bài toán người du lịch có một số đặc điểm quan trọng:

- **Tính NP-hard:** Bài toán người du lịch là một bài toán NP-hard, có nghĩa là không có thuật toán chính xác hiệu quả để giải quyết bài toán trong thời gian đa thức khi số lượng thành phố tăng lên.
- **Tối ưu hóa toàn cục:** Mục tiêu là tìm ra lời giải tối ưu toàn cục, tức là một lộ trình có tổng quãng đường di chuyển ngắn nhất, thay vì chỉ tìm ra một giải pháp gần đúng hoặc hài lòng với một giải pháp tạm chấp nhận.
- **Ứng dụng thực tế:** Dù có tính chất lý thuyết cao, bài toán người du lịch lại có rất nhiều ứng dụng trong thực tế. Ví dụ, các công ty giao hàng cần tối ưu hóa tuyến đường vận chuyển, hay các vấn đề trong việc lập kế hoạch sản xuất và phân phối hàng hóa.

1.3 Các phương pháp giải quyết bài toán người du lịch

Vì bài toán người du lịch rất khó giải quyết đối với số lượng thành phố lớn, các phương pháp giải quyết thường được chia thành hai nhóm chính:

- Phương pháp chính xác: Các thuật toán này đảm bảo tìm ra được lời giải tối ưu, ví dụ như thuật toán brute-force, thuật toán quay lui (backtracking), và thuật toán chiếu tìm kiếm (branch and bound). Tuy nhiên, những thuật toán này không khả thi khi số lượng thành phố quá lớn vì số lượng các khả năng cần kiểm tra là quá lớn.
- Phương pháp xấp xỉ: Các thuật toán này không đảm bảo tìm ra lời giải tối ưu nhưng có thể tìm được các lời giải gần tối ưu trong một khoảng thời gian hợp lý. Một số phương pháp điển hình là:
 - + Thuật toán Nearest Neighbor (hàng xóm gần nhất): Chọn thành phố gần nhất chưa được thăm.
 - + Thuật toán 2-opt: Tối ưu hóa lộ trình ban đầu bằng cách hoán đổi các đoạn đường.
 - + Thuật toán di truyền (Genetic Algorithm), Thuật toán duyệt sâu (Simulated Annealing), và thuật toán tìm kiếm Tabu: Đây là các phương pháp tìm kiếm heuristics có thể đưa ra lời giải gần đúng với chi phí tính toán hợp lý.

1.4 Mục tiêu và phạm vi nghiên cứu

Trong phạm vi nghiên cứu này, em tập trung vào việc mô phỏng và giải quyết bài toán người du lịch thông qua một chương trình với các thuật toán nổi bật để tìm kiếm giải pháp cho bài toán này. Các thuật toán mà em sẽ áp dụng và so sánh bao gồm:

- Brute Force: Thử tất cả các hoán vị của các thành phố để tìm ra lộ trình ngắn nhất.
- Nearest Neighbor: Sử dụng phương pháp chọn thành phố gần nhất để xây dựng lộ trình.
- 2-opt: Tối ưu hóa lộ trình ban đầu bằng cách thay đổi thứ tự các điểm thăm.

- Backtracking: Tìm kiếm giải pháp bằng cách thử các lựa chọn và quay lại khi gặp trường hợp không khả thi.

Mục tiêu là xây dựng một chương trình giúp người dùng có thể nhập vào tập hợp các thành phố, chọn thuật toán và nhận được lộ trình tối ưu hoặc gần tối ưu. Chương trình sẽ trực quan hóa kết quả dưới dạng đồ thị, đồng thời cung cấp các công cụ để so sánh hiệu quả của các thuật toán.

1.5 Ý nghĩa và ứng dụng

Việc giải quyết bài toán người du lịch không chỉ giúp nâng cao hiểu biết về các thuật toán tối ưu hóa mà còn có ứng dụng thực tế trong các vấn đề như:

- Tối ưu hóa lộ trình giao hàng: Giúp các công ty giao nhận tối ưu hóa lộ trình để tiết kiệm chi phí vận chuyển.
- Lập kế hoạch sản xuất: Tối ưu hóa việc sắp xếp các công đoạn trong sản xuất, nhằm giảm thiểu thời gian di chuyển hoặc thời gian sản xuất.
- Ứng dụng trong robot di động: Các robot cần tối ưu hóa lộ trình di chuyển trong môi trường không gian ba chiều, chẳng hạn như robot giao hàng trong các kho.

Thông qua nghiên cứu và giải quyết bài toán người du lịch, đề án này không chỉ giúp nâng cao kỹ năng lập trình mà còn đóng góp vào việc phát triển các ứng dụng phần mềm có ích trong thực tế.

CHƯƠNG 2: NGHIÊN CỨU LÝ THUYẾT

2.1 Cơ sở lý thuyết

- Định nghĩa bài toán người du lịch: Cho một số thành phố, tìm con đường ngắn nhất sao cho người bán hàng đi qua tất cả các thành phố và quay lại điểm xuất phát.
- Mô hình toán học:

$$\text{Minimize } \sum_{i=1}^{n-1} d_{i,j} \quad \text{với } d_{i,j} \text{ là khoảng cách giữa thành phố } i \text{ và thành phố } j.$$

Các thuật toán giải quyết bài toán này có thể được chia thành các nhóm như:

- Giải pháp chính xác (Exact Algorithms): Tìm ra lời giải tối ưu, bao gồm các thuật toán như Brute Force và Backtracking.
- Giải pháp xấp xỉ (Approximation Algorithms): Tìm lời giải gần đúng với độ chính xác không tuyệt đối nhưng rất hiệu quả về mặt tính toán, ví dụ như thuật toán Nearest Neighbor và 2-opt.

2.2 Lý luận

Bài toán người du lịch là bài toán NP-hard, tức là không có thuật toán hiệu quả nào để giải quyết bài toán này trong thời gian đa thức đối với mọi trường hợp. Tuy nhiên, có nhiều thuật toán và phương pháp có thể đưa ra giải pháp gần đúng hoặc giải quyết các trường hợp nhỏ một cách chính xác.

- Brute Force: Phương pháp này kiểm tra tất cả các hoán vị của các thành phố để tìm ra đường đi ngắn nhất. Tuy nhiên, do số lượng hoán vị tăng nhanh theo số lượng thành phố, phương pháp này chỉ phù hợp với các bài toán có quy mô nhỏ.
- Nearest Neighbor: Đây là một thuật toán tham lam, bắt đầu từ một thành phố và luôn chọn thành phố gần nhất chưa được thăm để tiếp tục hành trình. Mặc dù không đảm bảo giải pháp tối ưu, nhưng phương pháp này đơn giản và nhanh chóng.

- 2-opt: Thuật toán này cải thiện một giải pháp ban đầu bằng cách thử thay đổi các đoạn đường con (swap) để giảm tổng chi phí.
- Backtracking: Là phương pháp đệ quy, tìm kiếm tất cả các khả năng của bài toán và sử dụng cách tiếp cận loại bỏ các giải pháp không khả thi ngay từ đầu để tiết kiệm thời gian tính toán.

2.3 Giải thuyết khoa học

- Giả thuyết 1: Phương pháp Brute Force sẽ cho kết quả chính xác nhưng không thể áp dụng cho bài toán có số lượng thành phố lớn do độ phức tạp tính toán quá cao.
- Giả thuyết 2: Thuật toán Nearest Neighbor, mặc dù không tối ưu, sẽ tìm được giải pháp gần đúng nhanh chóng cho bài toán người du lịch với một số lượng thành phố nhất định.
- Giả thuyết 3: Thuật toán 2-opt có thể cải thiện kết quả của thuật toán Nearest Neighbor, giúp rút ngắn đường đi và đưa ra một giải pháp gần tối ưu hơn.
- Giả thuyết 4: Phương pháp Backtracking sẽ tìm ra được giải pháp tối ưu nhưng với độ phức tạp tính toán lớn, do đó chỉ khả thi khi số lượng thành phố không quá nhiều.

2.4 Phương pháp nghiên cứu

Phương pháp nghiên cứu của đề án chủ yếu là nghiên cứu thực nghiệm kết hợp với mô phỏng và phân tích dữ liệu thực tế. Các bước thực hiện bao gồm:

2.4.1 Mô hình hóa bài toán

Mô phỏng bài toán người du lịch bằng cách tạo ra ma trận khoảng cách giữa các thành phố, trong đó mỗi thành phố là một đỉnh và khoảng cách giữa các thành phố là trọng số của cạnh.

2.4.2 Lựa chọn thuật toán

Áp dụng và so sánh các thuật toán giải quyết người du lịch (Brute Force, Nearest Neighbor, 2-opt, Backtracking) để tìm ra giải pháp hiệu quả cho bài toán.

2.4.3 Xây dựng giao diện người dùng

Sử dụng thư viện tkinter để xây dựng một ứng dụng GUI cho phép người dùng nhập vào các thành phố, chọn thuật toán, và xem kết quả của thuật toán.

2.4.4 Thử nghiệm và phân tích kết quả

- So sánh thời gian chạy và độ chính xác của các thuật toán trên các bài toán có quy mô khác nhau.
- Đo lường và phân tích hiệu quả của thuật toán Nearest Neighbor so với thuật toán chính xác như Brute Force.
- Cải thiện kết quả của các thuật toán bằng cách sử dụng phương pháp tối ưu như 2-opt.

2.4.5 Đánh giá và tổng kết

- Đánh giá tính hiệu quả và độ chính xác của từng thuật toán.
- Phân tích mức độ phù hợp của từng thuật toán với các bài toán có số lượng thành phố khác nhau.

CHƯƠNG 3: HIỆN THỰC HÓA NGHIÊN CỨU

3.1 Mô tả các bước nghiên cứu đã tiến hành

Quá trình nghiên cứu và phát triển ứng dụng giải bài toán người du lịch được thực hiện qua các bước sau:

3.1.1 Xác định bài toán và mô hình hóa

- Định nghĩa bài toán người du lịch và yêu cầu của nó.
- Mô hình hóa bài toán dưới dạng đồ thị với các thành phố là các đỉnh và khoảng cách giữa các thành phố là trọng số của các cạnh.

3.1.2 Lựa chọn thuật toán

- Chọn các thuật toán thích hợp để giải quyết bài toán người du lịch, bao gồm Brute Force, Nearest Neighbor, 2-opt và Backtracking.
- Tìm hiểu và phân tích độ phức tạp, ưu điểm và nhược điểm của từng thuật toán.

3.1.3 Thiết kế hệ thống

- Thiết kế giao diện người dùng (GUI) cho phép người dùng nhập vào dữ liệu, chọn thuật toán và xem kết quả.
- Xây dựng các lược đồ thiết kế hệ thống như lược đồ use case, lược đồ lớp, và lược đồ hoạt động.

3.1.4 Cài đặt chương trình

- Viết mã nguồn cho các thuật toán Brute Force, Nearest Neighbor, 2-opt và Backtracking.
- Phát triển giao diện người dùng sử dụng thư viện tkinter trong Python.

3.1.5 Thử nghiệm và đánh giá

- Chạy thử nghiệm các thuật toán trên các tập dữ liệu khác nhau để so sánh kết quả và hiệu năng.
- Đánh giá kết quả và cải thiện thuật toán nếu cần thiết.

3.2 Các bản thiết kế

3.2.1 Lược đồ Use case

Người dùng:

- Nhập dữ liệu thành phố: Có thể nhập tọa độ thành phố hoặc tạo ngẫu nhiên.
- Chọn thuật toán: Brute Force, Nearest Neighbor, 2-opt, hoặc Backtracking.
- Chạy mô phỏng: Xem kết quả trực tiếp trên giao diện đồ họa.
- Lưu/Tải dữ liệu: Lưu kết quả hoặc nạp dữ liệu từ tệp JSON.

Các thành phần:

- Tác nhân (Actor): Người dùng.
- Các trường hợp sử dụng (Use Case):
 - + Nhập dữ liệu thành phố.
 - + Chọn thuật toán.
 - + Xem kết quả.
 - + Lưu hoặc tải dữ liệu.

3.2.2 Lược đồ Class

Lớp chính:

- TSPSolver:

+ Thuộc tính:

cities: Danh sách tọa độ các thành phố.

distance_matrix: Ma trận khoảng cách giữa các thành phố.

log: Nhật ký lưu các kết quả tính toán.

+ Phương thức:

generate_random_cities(n): Tạo ngẫu nhiên n thành phố.

brute_force(): Tìm đường đi ngắn nhất bằng thuật toán thử tất cả các hoán vị.

nearest_neighbor(): Áp dụng thuật toán gần nhất.

`two_opt()`: Tối ưu đường đi bằng thuật toán 2-opt.

`backtracking()`: Áp dụng thuật toán quay lui.

- `TSPApp`:

- + Thuộc tính:

`solver`: Đối tượng của lớp `TSPSolver`.

`canvas`: Giao diện đồ họa để hiển thị thành phố và đường đi.

`algorithm_var`: Biến lưu thuật toán được chọn.

- + Phương thức:

`generate_cities()`: Sinh dữ liệu thành phố từ giao diện.

`solve()`: Chọn và chạy thuật toán.

`plot_cities()`: Vẽ các thành phố trên giao diện.

`plot_path(path)`: Hiển thị đường đi ngắn nhất trên giao diện.

- Quan hệ: Lớp `TSPApp` sử dụng lớp `TSPSolver` để thực hiện các thuật toán tính toán.

3.2.3 Lược đồ hoạt động

Quy trình hoạt động:

- Bắt đầu: Người dùng khởi động ứng dụng.
- Nhập thông tin thành phố: Người dùng nhập tọa độ hoặc chọn số lượng để tạo thành phố ngẫu nhiên.
- Tính toán ma trận khoảng cách: Hệ thống tính toán khoảng cách giữa các thành phố và lưu trong `distance_matrix`.
- Chọn thuật toán: Người dùng chọn một trong các thuật toán: Brute Force, Nearest Neighbor, 2-opt, Backtracking.
- Thực thi thuật toán: Hệ thống tính toán đường đi ngắn nhất và chi phí tương ứng.
- Hiển thị kết quả: Kết quả được hiển thị qua giao diện đồ họa và bảng log.
- Lưu hoặc tải dữ liệu (tùy chọn): Người dùng có thể lưu kết quả hoặc nạp dữ liệu từ tệp JSON.
- Kết thúc.

3.3 Cách cài đặt chương trình

3.3.1 Cấu trúc thư mục dự án

TSP_project/

```
|— README.md
|— setup/
|   |— INSTALL.md
|— scr/
|   |— main.py
|   |— tsp_algorithms.py
|   |— tsp_gui.py
|— progress-report/
|— thesis/
|   |— doc/
|   |— pdf/
|   |— html/
|   |— abs/
|   |— refs/
```

3.3.2 Mô tả các tệp tin và chức năng

main.py: Tệp tin chính để khởi động chương trình.

tsp_algorithms.py: Chứa các thuật toán giải quyết bài toán TSP.

tsp_gui.py: Chứa mã nguồn xây dựng giao diện người dùng.

3.3.3 Nội dung tệp tin tsp_algorithms.py

```
import itertools

import math
```

```
import time

import json

import numpy as np

class TSPSolver:

    def __init__(self, cities=None):

        self.cities = cities if cities else []

        self.distance_matrix = []

        self.log = []

    def add_city(self, x, y):

        self.cities.append((x, y))

    def generate_random_cities(self, n):

        canvas_width, canvas_height = 1200, 600

        self.cities = [(np.random.randint(50, canvas_width - 50), np.random.randint(50,
        canvas_height - 50)) for _ in range(n)]

        self.calculate_distance_matrix()

    def calculate_distance_matrix(self):

        self.distance_matrix = []

        for i in range(len(self.cities)):

            distances = []

            for j in range(len(self.cities)):
```

```
        dist = np.linalg.norm(np.array(self.cities[i]) - np.array(self.cities[j]))

        distances.append(dist)

    self.distance_matrix.append(distances)

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

    })

def brute_force(self):

    min_cost = float('inf')

    best_path = None

    start_time = time.time()

    for perm in itertools.permutations(range(len(self.cities))):

        cost = sum(self.distance_matrix[perm[i]][perm[i + 1]] for i in

range(len(perm) - 1))

        cost += self.distance_matrix[perm[-1]][perm[0]]

        if cost < min_cost:

            min_cost = cost

            best_path = perm
```



```
time_taken = time.time() - start_time

self.log_result("Brute Force", time_taken, min_cost,
len(list(itertools.permutations(range(len(self.cities))))), best_path)

return best_path, min_cost

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]]

    time_taken = time.time() - start_time

    self.log_result("Nearest Neighbor", time_taken, total_cost, len(path), path)

    return path, total_cost
```

```
def two_opt(self, max_iterations=100):  
    def calculate_cost(route):  
        return sum(self.distance_matrix[route[i]][route[i + 1]] for i in  
range(len(route) - 1)) + \  
        self.distance_matrix[route[-1]][route[0]]  
  
    route = list(range(len(self.cities)))  
    best_cost = calculate_cost(route)  
    start_time = time.time()  
  
    for _ in range(max_iterations):  
        improved = False  
        for i in range(1, len(route) - 1):  
            for j in range(i + 1, len(route)):  
                new_route = route[:i] + route[i:j][::-1] + route[j:]  
                new_cost = calculate_cost(new_route)  
                if new_cost < best_cost:  
                    route, best_cost = new_route, new_cost  
                    improved = True  
                    break  
            if improved:  
                break  
        if not improved:
```

```
        break

    time_taken = time.time() - start_time

    self.log_result("2-opt", time_taken, best_cost, max_iterations, route)

    return route, best_cost

def backtracking(self):
    def backtrack(current_city, visited, current_cost):
        nonlocal min_cost, best_path

        if len(visited) == len(self.cities):
            total_cost = current_cost + self.distance_matrix[current_city][visited[0]]

            if total_cost < min_cost:
                min_cost = total_cost
                best_path = visited[:]

            return

        for next_city in range(len(self.cities)):
            if next_city not in visited:
                visited.append(next_city)

                backtrack(next_city, visited, current_cost +
self.distance_matrix[current_city][next_city])

                visited.pop()

        min_cost = float('inf')
```

```
best_path = None

start_time = time.time()

backtrack(0, [0], 0)

time_taken = time.time() - start_time

self.log_result("Backtracking", time_taken, min_cost, len(self.cities),
best_path)

return best_path, min_cost

def save_to_file(self, filename):

    with open(filename, 'w') as file:

        json.dump({"cities": self.cities, "distance_matrix": self.distance_matrix,
"log": self.log}, file)

def load_from_file(self, filename):

    with open(filename, 'r') as file:

        data = json.load(file)

        self.cities = data["cities"]

        self.distance_matrix = data["distance_matrix"]

        self.log = data.get("log", [])

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)
```

Bảng 1 Nội dung tệp tin tsp_algorithms.py

3.3.4 Nội dung tệp tin tsp_gui.py

```
import tkinter as tk  
  
from tkinter import messagebox, filedialog, ttk  
  
from tsp_algorithms import TSPSolver  
  
import time  
  
class TSPApp:  
  
    def __init__(self, root):  
  
        self.root = root  
  
        self.root.title("Mô phỏng bài toán người du lịch")  
  
        self.root.geometry("1600x900") # Điều chỉnh kích thước  
  
        self.solver = TSPSolver()  
  
        self.skip_animation = False  
  
        # Khởi tạo giao diện  
  
        self.main_frame = tk.Frame(root, padx=10, pady=10)  
  
        self.main_frame.pack(fill=tk.BOTH, expand=True)  
  
        # Phần Trên  
  
        self.top_frame = tk.Frame(self.main_frame)
```

```
self.top_frame.pack(side=tk.TOP, fill=tk.BOTH, expand=True)

# Phần Trái

self.left_panel = tk.Frame(self.top_frame, width=300)

self.left_panel.pack(side=tk.LEFT, fill=tk.Y, padx=10, pady=10)

# Phần Phải (Canvas + Log)

self.right_panel = tk.Frame(self.top_frame)

self.right_panel.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True)

# Canvas

self.canvas = tk.Canvas(self.right_panel, bg="white")

self.canvas.pack(fill=tk.BOTH, expand=True)

# Khu vực Log Dưới

self.log_tree = ttk.Treeview(self.right_panel, columns=("Thuật toán", "Thời gian", "Chi phí", "Bước", "Đường đi"), show="headings", height=10)

self.log_tree.heading("Thuật toán", text="Thuật toán")

self.log_tree.heading("Thời gian", text="Thời gian")

self.log_tree.heading("Chi phí", text="Chi phí")

self.log_tree.heading("Bước", text="Bước")

self.log_tree.heading("Đường đi", text="Đường đi")

self.log_tree.pack(fill=tk.X, pady=5)
```

```
# Khu vực Thành phố

self.city_frame = tk.LabelFrame(self.left_panel, text="Thành phố", padx=10,
pady=10)

self.city_frame.pack(fill=tk.X, padx=5, pady=5)

self.city_count_label = tk.Label(self.city_frame, text="Số lượng thành phố:")
self.city_count_label.grid(row=0, column=0, sticky="w")

self.city_count_entry = tk.Entry(self.city_frame)
self.city_count_entry.grid(row=0, column=1, padx=5, pady=5)

self.generate_button = tk.Button(self.city_frame, text="Tạo",
command=self.generate_cities)
self.generate_button.grid(row=1, column=0, columnspan=2, pady=5)

self.coordinates_label = tk.Label(self.city_frame, text="Tọa độ:")
self.coordinates_label.grid(row=2, column=0, columnspan=2)

self.coordinates_text = tk.Text(self.city_frame, height=10, width=25)
self.coordinates_text.grid(row=3, column=0, columnspan=2, pady=5)

self.generate_matrix_button = tk.Button(self.city_frame, text="Tạo ma trận",
command=self.generate_distance_matrix)
self.generate_matrix_button.grid(row=4, column=0, columnspan=2, pady=5)
```

```
# Khu vực Thuật toán

self.algorithm_frame = tk.LabelFrame(self.left_panel, text="Thuật toán",
padx=10, pady=10)

self.algorithm_frame.pack(fill=tk.X, padx=5, pady=5)

self.algorithm_var = tk.StringVar(value="Brute Force")

self.algorithms = {
    "Brute Force": "Thử tất cả các hoán vị để tìm đường ngắn nhất.",
    "Nearest Neighbor": "Chọn thành phố chưa thăm gần nhất lần lượt.",
    "2-opt": "Tối ưu hóa một đường ban đầu bằng cách đảo ngược các đoạn.",
    "Backtracking": "Khám phá tất cả các đường đi đệ quy với cắt tỉa."
}

for algo, desc in self.algorithms.items():
    tk.Radiobutton(self.algorithm_frame, text=algo, variable=self.algorithm_var,
value=algo,
                    command=lambda desc=desc:
self.show_algorithm_description(desc)).pack(anchor="w")

    self.algorithm_description = tk.Label(self.algorithm_frame, text="",
wraplength=250, justify="left")

    self.algorithm_description.pack(pady=5)

# Khu vực Điều khiển
```



```
self.control_frame = tk.LabelFrame(self.left_panel, text="Điều khiển",
padx=10, pady=10)

self.control_frame.pack(fill=tk.X, padx=5, pady=5)

self.solve_button = tk.Button(self.control_frame, text="Mô phỏng",
command=self.solve)

self.solve_button.grid(row=0, column=0, columnspan=2, pady=5, sticky="ew")

self.save_button = tk.Button(self.control_frame, text="Lưu kết quả",
command=self.save_results)

self.save_button.grid(row=1, column=0, pady=5, sticky="ew")

self.load_button = tk.Button(self.control_frame, text="Tải thành phố",
command=self.load_cities)

self.load_button.grid(row=1, column=1, pady=5, sticky="ew")

self.save_log_button = tk.Button(self.control_frame, text="Lưu Log",
command=self.save_log)

self.save_log_button.grid(row=2, column=0, pady=5, sticky="ew")

self.load_log_button = tk.Button(self.control_frame, text="Tải Log",
command=self.load_log)

self.load_log_button.grid(row=2, column=1, pady=5, sticky="ew")

self.clearlog_button = tk.Button(self.control_frame, text="Xóa log",
command=self.clear_log)
```

```
self.clearlog_button.grid(row=3, column=0, pady=5, sticky="ew")

self.refresh_button = tk.Button(self.control_frame, text="Làm mới",
command=self.refresh_app)

self.refresh_button.grid(row=3, column=1, pady=5, sticky="ew")

self.show_plot_button = tk.Button(self.control_frame, text="Xóa đường đi",
command=self.show_plot)

self.show_plot_button.grid(row=4, column=0, pady=5, sticky="ew")

self.skip_animation_button = tk.Button(self.control_frame, text="Bật/Tắt hoạt
hình mô phỏng", command=self.toggle_animation)

self.skip_animation_button.grid(row=4, column=1, pady=5, sticky="ew")

def toggle_animation(self):

    self.skip_animation = not self.skip_animation

    status = "đã tắt" if self.skip_animation else "đã bật"

    messagebox.showinfo("Hoạt hình", f"Hoạt hình hiện tại {status}.")

def show_algorithm_description(self, desc):

    self.algorithm_description.config(text=desc)

def generate_cities(self):

    try:

        n = int(self.city_count_entry.get())
```

```
self.solver.generate_random_cities(n)

self.coordinates_text.delete(1.0, tk.END)

for x, y in self.solver.cities:

    self.coordinates_text.insert(tk.END, f"{x},{y}\n")

except ValueError:

    messagebox.showerror("Lỗi", "Vui lòng nhập số hợp lệ.")

def generate_distance_matrix(self):

    try:

        self.solver.cities = []

        for line in self.coordinates_text.get(1.0, tk.END).strip().split("\n"):

            x, y = map(int, line.split(","))

            if 0 <= x <= 1200 and 0 <= y <= 600:

                self.solver.add_city(x, y)

            else:

                raise ValueError(f"Tọa độ thành phố ({x}, {y}) vượt quá giới hạn khu vực tọa độ mô phỏng được.")

        self.solver.calculate_distance_matrix()

        self.plot_cities()

        messagebox.showinfo("Thành công", "Ma trận khoảng cách đã được tạo và thành phố đã được mô phỏng thành công.")

    except Exception as e:

        messagebox.showerror("Lỗi", f"Dữ liệu không hợp lệ: {e}")
```

```
def plot_cities(self):

    self.canvas.delete("all")

    radius = 5

    for i, city in enumerate(self.solver.cities):

        x, y = city

        self.canvas.create_oval(x - radius, y - radius, x + radius, y + radius,
fill="blue")

        self.canvas.create_text(x + 10, y, text=str(i), anchor="w")

def solve(self):

    algorithm = self.algorithm_var.get()

    if not self.solver.cities:

        messagebox.showerror("Lỗi", "Không có thành phố để mô phỏng.")

        return

    start_time = time.time()

    if algorithm == "Brute Force":

        path, cost = self.solver.brute_force()

    elif algorithm == "Nearest Neighbor":

        path, cost = self.solver.nearest_neighbor()

    elif algorithm == "2-opt":

        path, cost = self.solver.two_opt()

    elif algorithm == "Backtracking":
```

```
        path, cost = self.solver.backtracking()

    else:

        messagebox.showerror("Lỗi", "Thuật toán không hợp lệ.")

        return

    elapsed_time = time.time() - start_time

    self.log_tree.insert("", "end", values=(algorithm, f"{elapsed_time:.2f}", cost,
len(path), path))

    if self.skip_animation:

        self.plot_path(path)

    else:

        self.animate_path(path)

    messagebox.showinfo("Kết quả", f"Chi phí: {cost}\nThời gian:
{elapsed_time:.2f} giây")

def plot_path(self, path):

    self.canvas.delete("all")

    radius = 5

    for i, city in enumerate(self.solver.cities):

        x, y = city

        self.canvas.create_oval(x - radius, y - radius, x + radius, y + radius,
fill="blue")

        self.canvas.create_text(x + 10, y, text=str(i), anchor="w")
```

```
for i in range(len(path)):

    x1, y1 = self.solver.cities[path[i]]

    x2, y2 = self.solver.cities[path[(i + 1) % len(path)]]

    self.canvas.create_line(x1, y1, x2, y2, fill="red", width=2)

def animate_path(self, path):

    self.canvas.delete("all")

    radius = 5

    for i, city in enumerate(self.solver.cities):

        x, y = city

        self.canvas.create_oval(x - radius, y - radius, x + radius, y + radius,
fill="blue")

        self.canvas.create_text(x + 10, y, text=str(i), anchor="w")

    for i in range(len(path)):

        x1, y1 = self.solver.cities[path[i]]

        x2, y2 = self.solver.cities[path[(i + 1) % len(path)]]

        self.canvas.create_line(x1, y1, x2, y2, fill="red", width=2)

        self.canvas.update()

        time.sleep(0.3)

def show_plot(self):

    self.canvas.delete("all")
```

```
radius = 5

for i, city in enumerate(self.solver.cities):

    x, y = city

    self.canvas.create_oval(x - radius, y - radius, x + radius, y + radius,
fill="blue")

    self.canvas.create_text(x + 10, y, text=str(i), anchor="w")


def save_results(self):

    filename = filedialog.asksaveasfilename(defaultextension=".json",
filetypes=[("Tập JSON", "*.json")])

    if filename:

        self.solver.save_to_file(filename)

        messagebox.showinfo("Thành công", "Kết quả đã được lưu.")


def load_cities(self):

    filename = filedialog.askopenfilename(filetypes=[("Tập JSON", "*.json")])

    if filename:

        self.solver.load_from_file(filename)

        self.coordinates_text.delete(1.0, tk.END)

        for x, y in self.solver.cities:

            self.coordinates_text.insert(tk.END, f"{x},{y}\n")

        messagebox.showinfo("Thành công", "Thành phố đã được tải.")


def save_log(self):
```

```
filename = filedialog.asksaveasfilename(defaultextension=".json",
filetypes=[("Tập JSON", "*.json")])

if filename:

    self.solver.save_log_to_file(filename)

    messagebox.showinfo("Thành công", "Log đã được lưu.")

def load_log(self):

    filename = filedialog.askopenfilename(filetypes=[("Tập JSON", "*.json")])

    if filename:

        self.solver.load_log_from_file(filename)

        self.log_tree.delete(*self.log_tree.get_children())

        for entry in self.solver.log:

            self.log_tree.insert("", "end", values=(entry["Thuật toán"], f'{entry["Thời
gian"]:.2f}', entry["Chi phí"], entry["Bước"], entry["Đường đi"]))

            messagebox.showinfo("Thành công", "Log đã được tải.")

def refresh_app(self):

    self.solver = TSPSolver()

    self.city_count_entry.delete(0, tk.END)

    self.coordinates_text.delete(1.0, tk.END)

    self.canvas.delete("all")

    messagebox.showinfo("Thành công", "Đã làm mới thành công.")

def clear_log(self):
```



```
self.log_tree.delete(*self.log_tree.get_children())

messagebox.showinfo("Thông báo", "Log đã được xóa.")
```

Bảng 2 Nội dung tệp tin tsp_gui.py

3.3.5 Nội dung tệp tin main.py

```
from tsp_gui import TSPApp

import tkinter as tk

if __name__ == "__main__":

    root = tk.Tk()

    app = TSPApp(root)

    root.mainloop()
```

Bảng 3 Nội dung tệp tin main.py

3.4 Hồ sơ thiết kế và cài đặt

3.4.1 Thiết kế hệ thống

- **Lược đồ Use Case:** Minh họa các trường hợp sử dụng chính của hệ thống, bao gồm việc quản lý dữ liệu thành phố, lựa chọn thuật toán tối ưu, và hiển thị kết quả thông qua giao diện đồ họa.
- **Lược đồ Class:** Hệ thống được thiết kế dựa trên các class chính:
 - + **TSPSolver:** Xử lý các thuật toán tìm đường đi ngắn nhất.
 - + **TSPApp:** Quản lý giao diện người dùng, tương tác với người dùng, và gọi các thuật toán từ lớp TSPSolver.
- **Lược đồ Hoạt động:** Quy trình hoạt động chính:
 - + Người dùng nhập số lượng thành phố hoặc tọa độ thành phố.
 - + Hệ thống tính toán ma trận khoảng cách giữa các thành phố.
 - + Người dùng chọn thuật toán tối ưu.

- + Kết quả đường đi và chi phí tối ưu được hiển thị trên giao diện đồ họa.

3.4.2 Cài đặt chương trình

Yêu cầu hệ thống:

- Python 3.x: Phiên bản 3.8 trở lên được khuyến nghị.
- Thư viện numpy: Hỗ trợ các phép toán số học.
- Thư viện tkinter: Thư viện tích hợp để xây dựng giao diện đồ họa (thường đi kèm với Python).

Hướng dẫn cài đặt

- Cài đặt thư viện: Mở terminal hoặc command prompt và chạy lệnh sau để cài đặt thư viện numpy: `pip install numpy`
- Tạo một thư mục mới trên máy tính của bạn.
- Sao chép hoặc tải các tệp sau vào thư mục:
 - + `main.py`
 - + `tsp_algorithms.py`
 - + `tsp_gui.py`
 - + `README.md`
- Chạy chương trình:
 - + Mở terminal hoặc command prompt trong thư mục chứa dự án.
 - + Chạy lệnh: `python main.py`
- Giao diện ứng dụng sẽ được khởi động, sẵn sàng để sử dụng.

Ghi chú: Nếu gặp lỗi liên quan đến thư viện tkinter, hãy kiểm tra lại cài đặt Python của bạn hoặc tham khảo tài liệu Python để đảm bảo thư viện này đã được tích hợp.

3.4.3 Thử nghiệm và đánh giá

- Thử nghiệm chương trình với các tập dữ liệu khác nhau để đảm bảo tính chính xác và hiệu năng.
- So sánh kết quả của các thuật toán để đánh giá tính hiệu quả và đưa ra kết luận.

CHƯƠNG 4: KẾT QUẢ NGHIÊN CỨU

4.1 Hiệu năng của các thuật toán

Sau khi thực hiện và thử nghiệm đồ án, các kết quả và đánh giá về hiệu năng của các thuật toán đã được thu thập. Dưới đây là kết quả và phân tích chi tiết cho từng thuật toán:

- Brute Force:
 - + Ưu điểm: Luôn tìm được lời giải tối ưu.
 - + Nhược điểm: Thời gian thực hiện tăng nhanh theo số lượng thành phố ($n!$). Do đó, không khả thi cho số lượng thành phố lớn.
 - + Thời gian chạy: Chỉ khả thi với số lượng thành phố nhỏ (dưới 10).
- Nearest Neighbor:
 - + Ưu điểm: Thời gian thực hiện nhanh, dễ cài đặt.
 - + Nhược điểm: Không đảm bảo lời giải tối ưu. Đôi khi có thể cho kết quả xa so với tối ưu.
 - + Thời gian chạy: Nhanh, phù hợp với số lượng thành phố lớn.
- 2-opt:
 - + Ưu điểm: Cải thiện lời giải từ thuật toán Nearest Neighbor, giúp kết quả gần với tối ưu hơn.
 - + Nhược điểm: Vẫn không đảm bảo lời giải tối ưu. Hiệu năng phụ thuộc vào số lần lặp.
 - + Thời gian chạy: Chấp nhận được với số lượng thành phố vừa phải.
- Backtracking:
 - + Ưu điểm: Tìm được lời giải tối ưu thông qua cách tiếp cận đệ quy.
 - + Nhược điểm: Độ phức tạp cao, chỉ phù hợp với số lượng thành phố nhỏ.
 - + Thời gian chạy: Khá chậm với số lượng thành phố lớn.

4.2 Trải nghiệm người dùng

Giao diện đồ họa (GUI) của ứng dụng đã được thiết kế để tối ưu hóa trải nghiệm người dùng:

- Giao diện thân thiện và dễ sử dụng:
 - + Người dùng có thể dễ dàng nhập số lượng thành phố, tọa độ thành phố, chọn thuật toán, và xem kết quả mô phỏng trực quan trên canvas.
 - + Các nút chức năng được sắp xếp hợp lý, giúp người dùng dễ dàng lưu, tải dữ liệu và log kết quả.
- Tính năng mô phỏng và hoạt hình:
 - + Ứng dụng cung cấp tùy chọn bật/tắt hoạt hình mô phỏng, giúp người dùng có thể quan sát trực quan quá trình tìm kiếm đường đi của các thuật toán.
 - + Kết quả mô phỏng hiển thị trực quan, với các thành phố được đánh dấu và đường đi giữa các thành phố được vẽ rõ ràng.

4.3 Giao diện chức năng

Dưới đây là một số giao diện chức năng của chương trình:

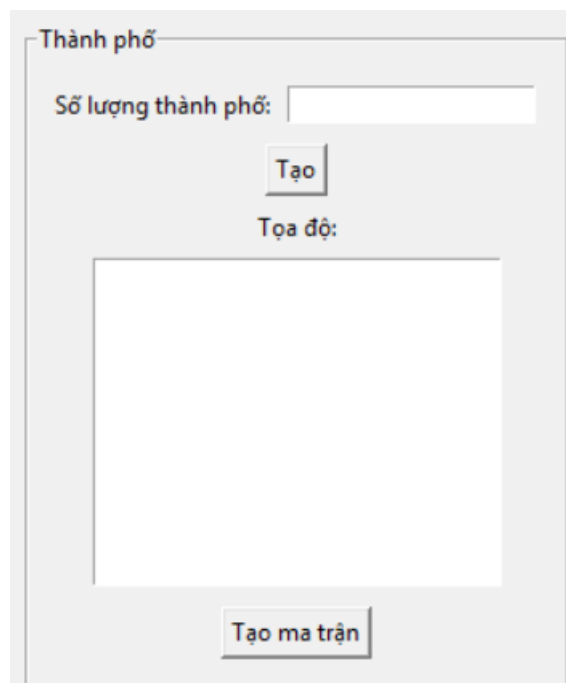
- Giao diện chính:
 - + Bao gồm khung nhập liệu cho số lượng thành phố và tọa độ thành phố, khung chọn thuật toán, và khung hiển thị kết quả log.
 - + Canvas trung tâm hiển thị các thành phố và đường đi được tìm kiếm.

Viết chương trình mô phỏng bài toán người du lịch



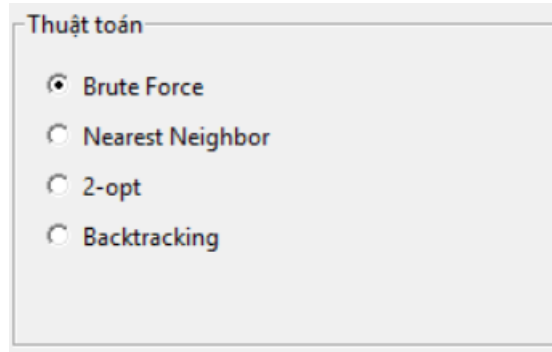
Hình 1 Giao diện chính của chương trình

- Khung nhập liệu: Cho phép người dùng nhập số lượng thành phố và tọa độ thành phố. Người dùng cũng có thể tạo ngẫu nhiên các thành phố trong giới hạn của canvas.



Hình 2 Khung nhập liệu

- Khung chọn thuật toán: Cho phép người dùng chọn một trong các thuật toán: Brute Force, Nearest Neighbor, 2-opt, và Backtracking. Mỗi thuật toán đều có mô tả ngắn gọn về phương pháp hoạt động.



Hình 3 Khung chọn thuật toán

- Khung kết quả log: Hiển thị kết quả của các lần mô phỏng, bao gồm thuật toán được sử dụng, thời gian thực hiện, chi phí, số bước và đường đi.

Thuật toán	Thời gian	Chi phí	Bước	Đường đi

Hình 4 Khung kết quả log

- Các nút chức năng:
 - + Nút "Mô phỏng" để chạy thuật toán và hiển thị kết quả.
 - + Nút "Lưu kết quả", "Tải thành phố", "Lưu Log", "Tải Log" để lưu và tải dữ liệu từ file JSON.
 - + Nút "Xóa log" để xóa log kết quả hiện tại.
 - + Nút "Làm mới" để làm mới giao diện và dữ liệu.
 - + Nút "Bật/Tắt hoạt hình mô phỏng" để bật hoặc tắt hoạt hình mô phỏng.

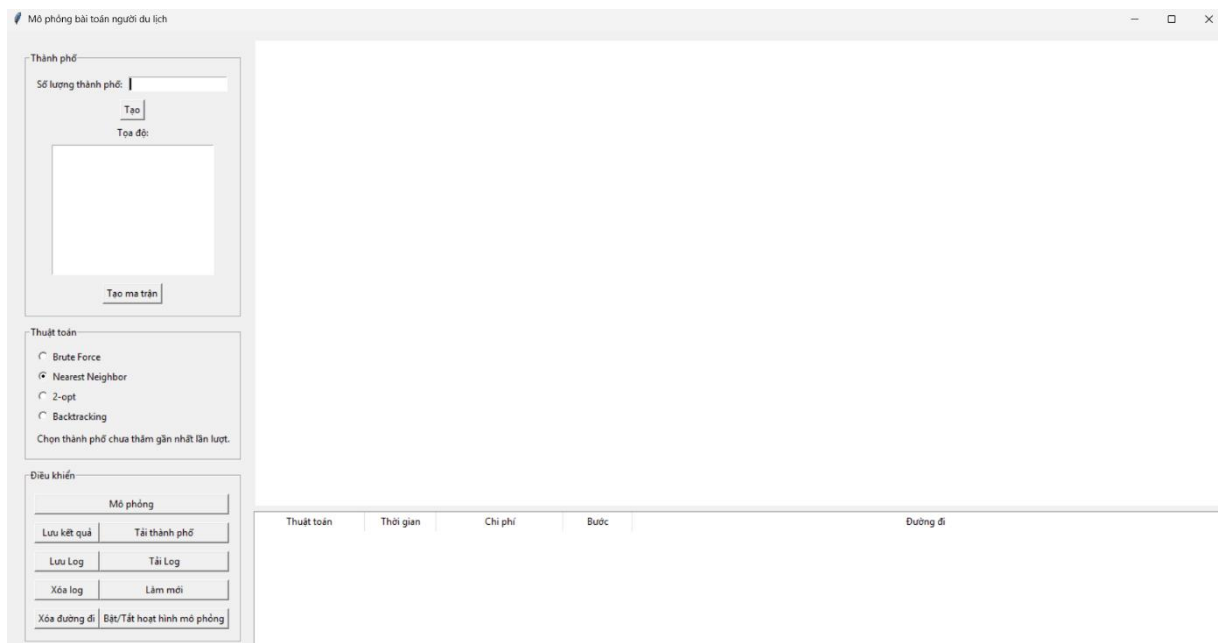


Hình 5 Các nút chức năng

4.4 Hình ảnh minh họa

Dưới đây là một số hình ảnh minh họa giao diện và kết quả mô phỏng của ứng dụng:

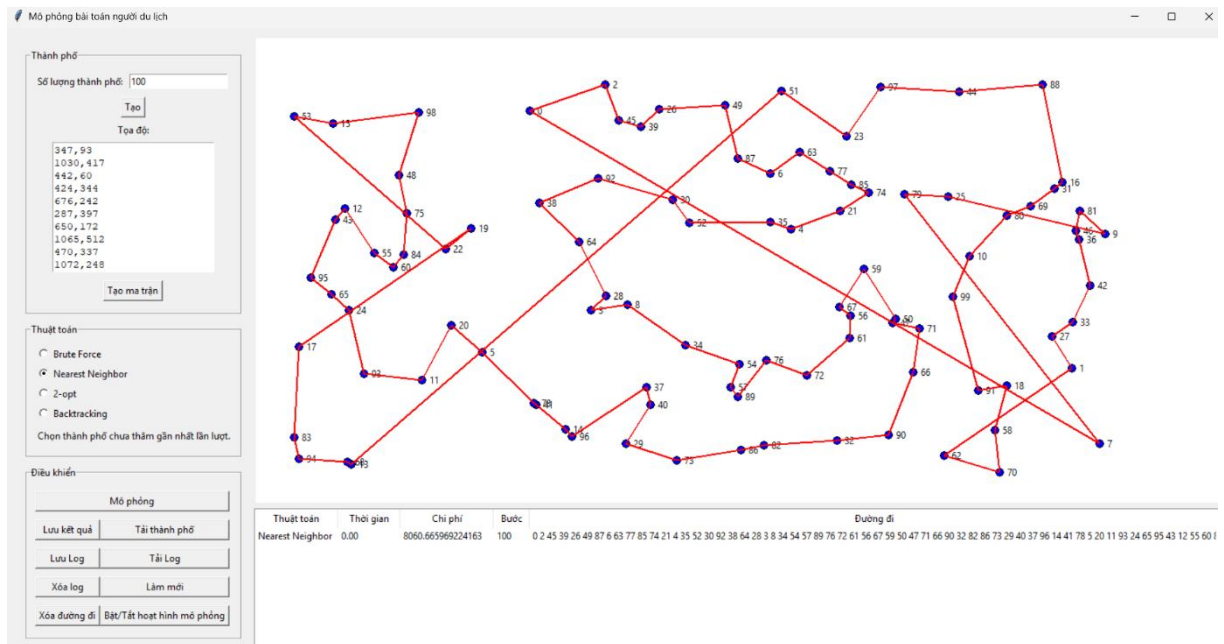
- Giao diện chính khi khởi động:



Hình 6 Giao diện chính của chương trình khi khởi động

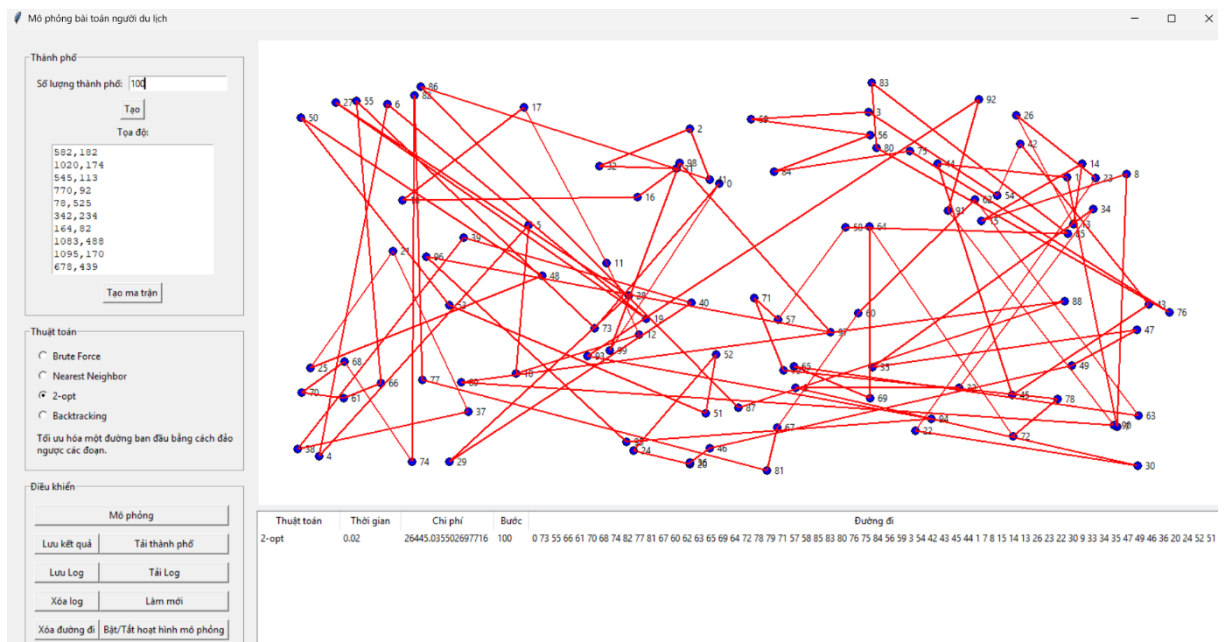
Viết chương trình mô phỏng bài toán người du lịch

- Kết quả mô phỏng thuật toán Nearest Neighbor:



Hình 7 Kết quả mô phỏng thuật toán Nearest Neighbor

- Kết quả mô phỏng thuật toán 2-opt:



Hình 8 Kết quả mô phỏng thuật toán 2-opt

CHƯƠNG 5: KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

5.1 Kết luận

5.1.1 Kết quả đạt được

Ứng dụng đã thành công trong việc xây dựng một hệ thống mô phỏng bài toán người du lịch (TSP) với giao diện đồ họa người dùng (GUI) trực quan và dễ sử dụng.

Các thuật toán Brute Force, Nearest Neighbor, 2-opt và Backtracking đã được triển khai và thử nghiệm thành công, với các kết quả về hiệu năng và thời gian chạy được ghi nhận.

Giao diện đồ họa cho phép người dùng nhập liệu, chọn thuật toán, và xem kết quả mô phỏng một cách dễ dàng và trực quan.

Chức năng lưu và tải dữ liệu từ các file JSON giúp người dùng có thể tiếp tục công việc từ nơi đã dừng lại.

5.1.2 Đóng góp mới

Cung cấp một công cụ mô phỏng bài toán người du lịch toàn diện với nhiều thuật toán khác nhau, giúp người dùng có thể so sánh và đánh giá hiệu năng của các phương pháp.

Giao diện người dùng được thiết kế thân thiện, hỗ trợ tốt cho việc nhập liệu và hiển thị kết quả, giúp người dùng có trải nghiệm tốt hơn khi làm việc với bài toán người du lịch.

5.1.3 Đề xuất mới

Đề xuất tích hợp thêm các thuật toán tối ưu hóa khác như Genetic Algorithm, Simulated Annealing để cung cấp nhiều lựa chọn hơn cho người dùng.

Phát triển thêm các tính năng phân tích và báo cáo kết quả chi tiết hơn, giúp người dùng hiểu rõ hơn về các thuật toán và kết quả của chúng.

5.2 Hướng phát triển

Kiến nghị về những hướng nghiên cứu tiếp theo:

- Tích hợp thêm thuật toán: Nghiên cứu và tích hợp thêm các thuật toán hiện đại và hiệu quả như Ant Colony Optimization, Genetic Algorithm, Simulated Annealing để mở rộng lựa chọn cho người dùng.
- Tối ưu hóa giao diện người dùng: Cải thiện giao diện đồ họa để nâng cao trải nghiệm người dùng, thêm các tính năng tương tác như kéo thả thành phố, zoom và pan trên canvas.

- Phân tích và báo cáo kết quả: Phát triển các module phân tích và báo cáo chi tiết, cung cấp thông tin sâu hơn về hiệu năng của các thuật toán, đồ thị phân tích và so sánh kết quả.
- Ứng dụng thực tế: Mở rộng ứng dụng vào các lĩnh vực thực tế như logistics, lập kế hoạch vận chuyển, và các vấn đề tối ưu hóa khác.

Qua đó, nghiên cứu và phát triển thêm các hướng mới sẽ giúp hoàn thiện hơn công cụ mô phỏng bài toán Người du lịch, tăng cường 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].