TRƯỜNG KỸ THUẬT VÀ CÔNG NGHỆ

KHOA CÔNG NGHỆ THÔNG TIN

THỰC TẬP ĐỒ ÁN CƠ SỞ NGÀNH

HỌC KỲ I, NĂM HỌC 2025-2026

VIẾT CHƯƠNG TRÌNH

MÔ PHỎNG GIẢI THUẬT FLOYD WARSHALL

Giảng viên hướng dẫn:
ThS. Trầm Hoàng Nam

Sinh viên thực hiện:
Họ tên: Võ Nhật Duy Nam
MSSV: 110122119

Vĩnh Long, tháng 1 năm 2026

NHẬN XÉT CỦA GIẢNG VIÊN HƯỚNG DẪN

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Vĩnh Long, ngày 5 tháng 1 năm 2026

Giáo viên hướng dẫn

(Ký tên và ghi rõ họ tên)

NHẬN XÉT CỦA THÀNH VIÊN HỘI ĐỒNG

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Vĩnh Long, ngày ..... tháng …… năm ……

Thành viên hội đồng

(Ký tên và ghi rõ họ tên)

LỜI CẢM ƠN

Báo cáo đồ án cơ sở ngành với chủ đề "Viết chương trình mô phỏng thuật toán Floyd Warshal" là kết quả của sự nỗ lực không ngừng của bản thân tôi và sự giúp đỡ, động viên của thầy cô, bạn bè. Thông qua báo cáo này, tôi xin gửi lời cảm ơn đến những người đã giúp đỡ tôi trong quá trình nghiên cứu và hoàn thành đồ án. Tôi xin bày tỏ sự kính trọng và biết ơn sâu sắc tới thầy Trầm Hoàng Nam vì đã trực tiếp hướng dẫn và cung cấp những thông tin cần thiết cho đồ án này. Tôi xin chân thành cảm ơn lãnh đạo Trường Đại học Trà Vinh, Khoa Kỹ thuật và Công nghệ, bộ môn Công nghệ thông tin đã tạo điều kiện để tôi hoàn thành thành công đồ án. Trong quá trình thực hiện đồ án, do kiến thức còn hạn chế nên còn nhiều thiếu sót mong các thầy cô có thể bổ sung thêm để đồ án hoàn thiện hơn.

Xin chân thành cám ơn!

Người thực hiện

Võ Nhật Duy Nam

MỤC LỤC

DANH MỤC HÌNH ẢNH

TÓM TẮT ĐỒ ÁN CƠ SỞ NGÀNH

Đồ án "Viết chương trình mô phỏng giải thuật Floyd-Warshall" nhằm cung cấp cho sinh viên cái nhìn tổng quan và sâu sát về một trong những giải thuật quan trọng của lý thuyết đồ thị trong ngành khoa học máy tính.

Trong bài toán này, tôi đã nghiên cứu các nguyên lý hoạt động của giải thuật Floyd-Warshall, áp dụng các nguyên tắc để xây dựng chương trình mô phỏng nhằm tìm đường đi ngắn nhất trong đồ thị có trọng số. Để tối ưu hoá việc xử lý, chương trình được cài đặt bằng Python và sử dụng giao diện tkinter giúp người dùng dễ dàng tương tác.

Kết quả đạt được bao gồm:

- Mô phỏng chính xác hoạt động của giải thuật Floyd-Warshall.
- Minh hoạ giao diện đồ thị và đường đi ngắn nhất.

Các hướng tiếp cận bao gồm nghiên cứu về lý thuyết đồ thị, xây dựng thuật toán và mô phỏng trên giao diện trực quan.

MỞ ĐẦU

- **Lý do chọn đề tài**

Trong bối cảnh phát triển mạnh mẽ của công nghệ thông tin, việc giải quyết các bài toán tối ưu trên đồ thị ngày càng trở nên quan trọng và phổ biến trong nhiều lĩnh vực như giao thông, mạng máy tính, logistics và trí tuệ nhân tạo. Trong đó, bài toán tìm đường đi ngắn nhất giữa các cặp đỉnh trong đồ thị đóng vai trò quan trọng, đặc biệt trong việc tối ưu hóa chi phí, thời gian và tài nguyên.

Giải thuật Floyd-Warshall, với khả năng tính toán hiệu quả đường đi ngắn nhất giữa tất cả các cặp đỉnh trong một đồ thị có trọng số, là một trong những giải thuật quan trọng nhất trong lý thuyết đồ thị. Để hiểu rõ hơn cách thức hoạt động cũng như ứng dụng thực tiễn của giải thuật này, việc xây dựng một chương trình mô phỏng trực quan là cần thiết.

- **Mục đích nghiên cứu**
    + Hiểu rõ lý thuyết và cơ chế hoạt động của giải thuật Floyd-Warshall thông qua việc lập trình và mô phỏng.
    + Xây dựng một ứng dụng trực quan, thân thiện, giúp người dùng có thể tương tác và trải nghiệm cách giải thuật hoạt động trên các đồ thị cụ thể.
    + Ứng dụng Python và Tkinter để minh họa đồ thị, thể hiện đường đi ngắn nhất và quá trình tính toán của giải thuật.
    + Cung cấp một công cụ hỗ trợ giảng dạy, học tập và nghiên cứu giải thuật Floyd-Warshall.
- **Đối tượng nghiên cứu:**
    + Giải thuật Floyd-Warshall và các ứng dụng liên quan đến bài toán tìm đường đi ngắn nhất trong đồ thị.
    + Các thành phần chính của ngôn ngữ Python, thư viện Tkinter và các công cụ hỗ trợ vẽ đồ thị.
- **Phạm vi nghiên cứu:**

+ Xây dựng chương trình xử lý đồ thị, với khả năng nhập ma trận trọng số tùy chỉnh.
+ Giao diện đồ họa cho phép người dùng chỉnh sửa đồ thị, nhập điểm bắt đầu và kết thúc để tìm đường đi ngắn nhất.
+ Ứng dụng chỉ tập trung vào đồ thị có trọng số (có thể dương hoặc âm) nhưng không có chu trình âm.

CHƯƠNG 1: TỔNG QUAN

1.1. Cơ sở lý thuyết

1.1.1. Đồ thị

Đồ thị là một cấu trúc rời rạc bao gồm các đỉnh và các cạnh nối các đỉnh này. Chúng ta phân biệt các loại đồ thị khác nhau bởi kiểu và số lượng cạnh nối hai đỉnh nào đó của đồ thị.[1]

1.1.1.1. Định nghĩa 1

Đồ thị vô hướng G = (V,E) bao gồm V là tập các đỉnh, và E là tập các cặp không có.Thứ tự gồm hai phần tử khác nhau của V gọi là các cạnh.[1]

1.1.1.2. Định nghĩa 2

Hai đỉnh được gọi là liền kề nhau nếu có cạnh nối hai đỉnh đó với nhau. [1]

Cạnh nối 2 đỉnh lại với nhau được gọi là cạnh liên thuộc.Hai cạnh được gọi là kề nhau nếu giữa chúng có đỉnh chung.Nếu e=(v,v) là một cạnh của đồ thị thì e được gọi là một khuyên.[1]

1.1.1.3. Định nghĩa 3

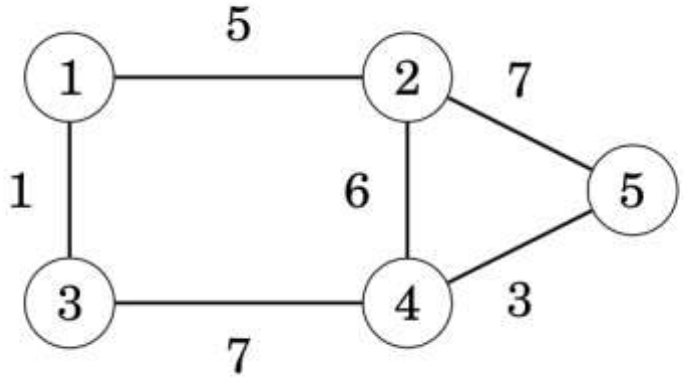
- Nếu mỗi cạnh e=(u,v) là không phân biệt thứ tự của các đỉnh u và v

(từ u v tới v không kể hướng) thì ta nói đồ thị G=(V,E) là đồ thị vô hướng.[1]

- Nếu mỗi cạnh e = (u,v) có phân biệt thứ tự của các đỉnh u và v (tức là từ u tới v khác từ v tới u) thì ta nói đồ thị G=(V,E) là đồ thị có hướng. Cạnh của đồ thị có hướng được gọi là cung.[1]

1.1.2. Đồ thị có trọng số

- Đồ thị có trọng số (weighted) là đồ thị có các cạnh được gán một trọng số. Các trọng số thường thể hiện chiều dài của cạnh.[2]
- Ví dụ, đồ thị sau là đồ thị có trọng số:



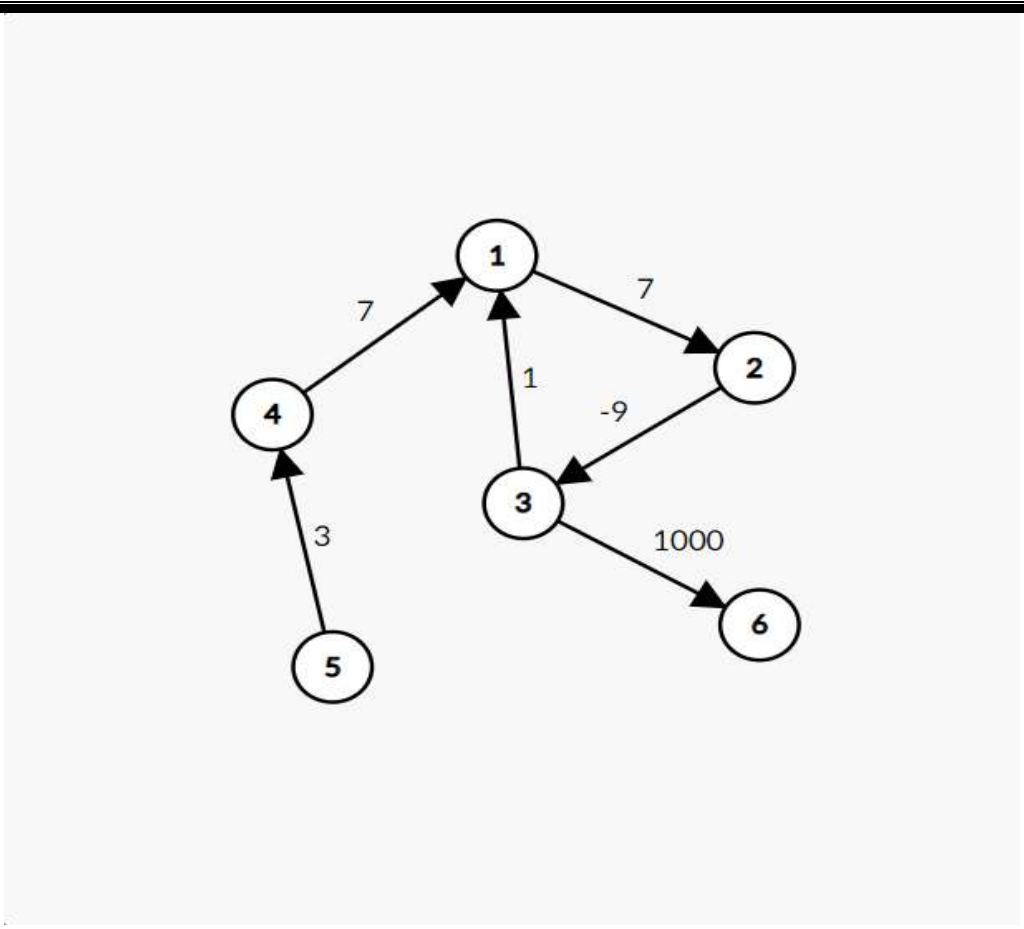
Hình 1: Đồ thị có trọng số

- Chiều dài của con đường trong đồ thị trọng số là tổng trọng số của các cạnh trên đường đi. Ví dụ, trong đồ thị trên, chiều dài của đường đi 1 → 2 → 5 là 12, và chiều dài của đường đi 1 → 3 → 4 → 5 là 11. Đường đi sau là đường đi ngắn nhất từ đỉnh 1 đến đỉnh 5.

1.1.3. Chu trình âm

- Chu trình âm là một chu trình trong đó tổng trọng số các cạnh là số âm. [4]
- Ví dụ: trong bài toán trên, ta có một chu trình âm 1–2–3–1 có tổng trọng số là

7–9+1= −1. Nếu trên đường đi từ u đến v chứa chu trình âm thì độ dài đường đi ngắn nhất từ u đến v sẽ là âm vô cực. Vì vậy nên một số cặp đỉnh không tồn tại đường đi ngắn nhất do có chu trình âm trên đường đi giữa chúng (chỉ tồn tại đường đi có độ dài âm vô cực).Ở đồ thị này, đường đi ngắn nhất từ 4 đến 6 sẽ có cách đi là vô hạn lần qua chu trình âm đã nhắc đến, sau đó mới đi đến 6. Như vậy không có đường đi ngắn nhất.



Hình 2: Chu trình âm trong đồ thị

CHƯƠNG 2: NGHIÊN CỨU LÝ THUYẾT

2.1. Bài toán Floyd-Warshall

2.1.1 Định nghĩa

Thuật toán Floyd-Warshall còn được gọi là thuật toán Floyd được Robert Floyd tìm ra năm 1962 là thuật toán để tìm đường đi ngắn nhất giữa mọi cặp đỉnh. Floyd hoạt động được trên đồ thị có hướng, có thể có trọng số âm, tuy nhiên không có chu trình âm.[2]

2.1.2 Giả thuyết khoa học

Thuật toán Floyd-Warshall khẳng định rằng thuật toán này có thể tìm ra đường đi ngắn nhất giữa tất cả các cặp đỉnh trong một đồ thị có trọng số, miễn là không có chu trình âm. Điều này làm cho thuật toán trở thành một công cụ hữu ích trong nhiều lĩnh vực khác nhau, từ mạng máy tính đến hệ thống giao thông

2.1.3 Một số định lý của thuật toán Floyd-Warshall

2.1.3.1. Tính chất đường đi ngắn nhất

Đường đi ngắn nhất của Floyd-Warshall dựa trên nguyên tắc rằng mọi đường đi có thể được chia nhỏ, và việc đi qua các đỉnh trung gian k sẽ cải thiện hoặc giữ nguyên khoảng cách ngắn nhất từ i đến j. Công thức đệ quy đảm bảo tính chính xác và hiệu quả của thuật toán.

2.1.3.2. Hội tụ

Thuật toán hội tụ sau n bước (với n là số đỉnh)

2.1.3.3. Tính chính xác

Kết quả đúng nếu đồ thị không có chu trình âm.

2.1.3.4. Bất biến

Thuật toán sẽ luôn kết thúc sau n bước.

2.1.4 Giải thuật Floyd-Warshall

2.1.4.1. Cách tính các ma trận trong giải thuật Floyd-Warshall

- Giải thuật Floyd-Warshall sử dụng hai ma trận chính:

  - **Ma trận khoảng cách (dist):** Lưu trữ khoảng cách ngắn nhất giữa các cặp đỉnh.

  - Ma **trận truy vết (next_node):** Lưu thông tin về đỉnh tiếp theo trên đường đi ngắn nhất từ đỉnh i đến j.

- **Quy trình tính toán ma trận**:

- **Khởi tạo ma trận** dist **và** next_node**:**

  + dist[i][j]: Bằng trọng số cạnh i→j nếu có cạnh, và bằng vô cực (INF) nếu không có cạnh nối.

  + next_node[i][j]: Ban đầu, nếu tồn tại cạnh i→j, giá trị sẽ là j. Nếu không, gán giá trị None.

- **Cập nhật ma trận với đỉnh trung gian** k**:**

  + Dùng ba vòng lặp lồng nhau để duyệt qua tất cả các cặp đỉnh i,j kiểm tra xem việc đi qua đỉnh trung gian k có rút ngắn được khoảng cách hay không:

  + Nếu dist[i][k] + dist[k][j] < dist[i][j]:

  + dist[i][j] = dist[i][k] + dist[k][j]

  + next_node[i][j] = next_node[i][k]

2.1.4.2. Xác định đường đi ngắn nhất

- Sau khi tính toán, sử dụng ma trận next_node để truy vết đường đi ngắn nhất giữa hai đỉnh i và j.

- **Quy trình tái dựng đường đi:**

  - đầu từ đỉnh i, dùng next_node[i][j] để tìm đỉnh kế tiếp.

  - Lặp lại quá trình đến khi đạt đến đỉnh j.

2.1.4.3. Ví dụ minh họa

Giả sử đồ thị có 5 đỉnh với ma trận trọng số ban đầu:

$$\begin{bmatrix} 0 & 4 & \infty & 7 & \infty \\ 4 & 0 & 3 & \infty & 6 \\ \infty & 3 & 0 & 2 & \infty \\ 7 & \infty & 2 & 0 & 5 \\ \infty & 6 & \infty & 5 & 0 \end{bmatrix}$$

Hình 3: Ma trận ví dụ 1

Khởi tạo ma trận disk và next_node:

- dist:

$$\begin{bmatrix} 0 & 4 & \infty & 7 & \infty \\ 4 & 0 & 3 & \infty & 6 \\ \infty & 3 & 0 & 2 & \infty \\ 7 & \infty & 2 & 0 & 5 \\ \infty & 6 & \infty & 5 & 0 \end{bmatrix}$$

Hình 4: Ma trận ví dụ 2

- next_node:

$$\begin{bmatrix} None & 1 & None & 3 & None \\ 0 & None & 2 & None & 4 \\ None & 1 & None & 3 & None \\ 0 & None & 2 & None & 4 \\ None & 1 & None & 3 & None \end{bmatrix}$$

Hình 5: Ma trận ví dụ 3

Cập nhật qua các đỉnh trung gian: Ví dụ: Sử dụng đỉnh trung gian k=2:

- dist[0][3]=min(dist[0][3],dist[0][2]+dist[2][3])
- Nếu có cải thiện, cập nhật giá trị dist và next_node.

Kết quả cuối cùng (sau khi cập nhật):

- dist:

$$\begin{bmatrix} 0 & 4 & 7 & 7 & 11 \\ 4 & 0 & 3 & 5 & 6 \\ 7 & 3 & 0 & 2 & 7 \\ 7 & 5 & 2 & 0 & 5 \\ 11 & 6 & 7 & 5 & 0 \end{bmatrix}$$

Hình 6: Ma trận ví dụ 4

- Đường đi ngắn nhất từ đỉnh 0 đến đỉnh 4:
  - Truy vết từ ma trận next_node: 0→1→4
  - Khoảng cách ngắn nhất: dist[0][4]=11

2.1.4.4. Biểu diễn các bước trên giao diện

- Hiển thị ma trận trọng số ban đầu.
- Tính toán và hiển thị ma trận dist và next_node theo từng bước.
- Vẽ đồ thị trực quan với đường đi ngắn nhất được tô đỏ.
- Cho phép người dùng nhập hoặc chỉnh sửa đồ thị để kiểm tra các kết quả khác.

CHƯƠNG 3: HIỆN THỰC HÓA NGHIÊN CỨU

3.1Cài đặt chương trình

3.1.1. Khai báo các thư viện cần thiết

```
import tkinter as tk
from tkinter import ttk, messagebox, scrolledtext
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure
import time
```

Hình 7: Khai báo thư viện

Trong đó:

Tkinter & Ttk: Xây dựng giao diện người dùng (GUI).

Numpy: Xử lý ma trận số học.

NetworkX: Quản lý cấu trúc đồ thị và tính toán vị trí đỉnh.

Matplotlib: Vẽ đồ thị và nhúng vào giao diện Tkinter.

3.1.2. Khởi tạo đồ thị

```
class FloydWarshallApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Mô phỏng Giải thuật Floyd-Warshall")
        self.root.geometry("1600x900")
        self.root.configure(bg="#2c3e50")
        self.root.state('zoomed')

        self.INF = 999
        self.current_step = 0
        self.steps = []
        self.is_running = False
        self.speed = 1000
        self.graph_history = []
        self.current_example_index = 0
```

Hình 8: Giá trị vô cực inf

Giao diện được chia thành 3 phần chính (Panel):

Panel Trái: Nhập liệu số đỉnh, tạo ma trận và các nút điều khiển (Bắt đầu, Tạm dừng).

Panel Giữa: Hiển thị đồ thị trực quan và thanh cuộn xem lịch sử.

Panel Phải: Hiển thị ma trận số và nhật ký thực hiện (Log).

3.1.2.1. Hàm thực hiện giải thuật Floyd-Warshall

```
def compute_floyd_warshall(self, graph):
    n = len(graph)
    dist = np.array(graph, dtype=float)
    next_node = np.full((n, n), -1, dtype=int)

    # Khởi tạo ma trận next_node
    for i in range(n):
        for j in range(n):
            if i != j and dist[i][j] < self.INF:
                next_node[i][j] = j

    # Trạng thái ban đầu
    self.steps.append({
        'matrix': dist.copy(),
        'next_matrix': next_node.copy(),
        'k': -1, 'i': -1, 'j': -1,
        'message': 'Khởi tạo ma trận ban đầu D⁰',
        'explanation': 'Ma trận ban đầu chứa khoảng cách trực tiếp giữa các đỉnh. Nếu không c
        'updated': False,
        'negative_cycle': False
    })
```

Hình 9: Hàm thực hiện giải thuật Floyd-Warshall

n = len(graph): Xác định số lượng đỉnh của đồ thị (kích thước *n* \times *n* của ma trận).

dist = np.array(graph, dtype=float): Khởi tạo ma trận khoảng cách (dist) dựa trên dữ liệu đồ thị đầu vào.

Ma trận này sẽ lưu trữ khoảng cách ngắn nhất giữa các cặp đỉnh.

Ban đầu, nó chứa trọng số trực tiếp giữa các đỉnh (nếu không có cạnh nối thì giá trị là vô cực INF).

next_node = np.full((n, n), -1, dtype=int): Khởi tạo ma trận truy vết đường đi (next_node) kích thước *n* \times *n*.

Ban đầu tất cả các giá trị được gán là -1 (nghĩa là chưa xác định được đỉnh kế tiếp).

Vòng lặp khởi tạo next_node (Dòng 688-691):

Duyệt qua các cặp đỉnh (*i, j*).

Nếu tồn tại cạnh nối trực tiếp từ *i* đến *j* (tức là dist[i][j] < INF), ta gán next_node[i][j] = j.

Ý nghĩa: Để đi từ *i* đến *j*, đỉnh kế tiếp cần đi chính là *j*.

self.steps.append({...}) (Dòng 694-702):

Lưu lại trạng thái ban đầu của ma trận vào danh sách self.steps.

Mục đích: Để hiển thị "Bước 0" (Trạng thái khởi tạo) lên giao diện mô phỏng, giúp người dùng so sánh sự thay đổi trước và sau khi chạy thuật toán.

3.1.2.2. Xác định các đỉnh kế tiếp trên đường đi trực tiếp từ đỉnh đến đích

```
else:  # random
    edge_probability = 0.4
    for i in range(n):
        for j in range(n):
            if i != j and random.random() < edge_probability:
                matrix[i][j] = random.randint(1, 12)
```

Hình 10: Xác định các đỉnh

next_node[i][j] = j: Nếu giữa đỉnh *i* và đỉnh *j* có cạnh nối trực tiếp (khoảng cách nhỏ hơn vô cực INF), thì đỉnh kế tiếp để đi từ *i* sang *j* chính là *j*.

Vòng lặp for i, for j: Duyệt qua tất cả các cặp đỉnh trong đồ thị để thiết lập giá trị khởi tão này trước khi chạy thuật toán chính.

Ý nghĩa: Ma trận next_node này đóng vai trò quan trọng trong việc **truy vết (trace back)** đường đi sau khi thuật toán hoàn tất. Thay vì chỉ biết độ dài, ta biết được chính xác cần đi qua những đỉnh nào.

3.1.2.3. Tính toán giải thuật Floyd-Warshall

```
for k in range(n):
    if algorithm_stopped:
        break

    # Thêm bước bắt đầu xét đỉnh k
    self.steps.append({
        'matrix': dist.copy(),
        'next_matrix': next_node.copy(),
        'k': k, 'i': -1, 'j': -1,
        'message': f'Bắt đầu xét đỉnh trung gian k = {k}',
        'explanation': f'Xét đỉnh {k} làm đỉnh trung gian. Kiểm tra xem có thể cải thiện đường đi từ i đến j bằng các
        'updated': False,
        'negative_cycle': False
    })

    # Thực hiện các cập nhật cho k hiện tại
    for i in range(n):
        for j in range(n):
            if dist[i][k] + dist[k][j] < dist[i][j]:
                old_value = dist[i][j]
                dist[i][j] = dist[i][k] + dist[k][j]
                next_node[i][j] = next_node[i][k]

                # Tạo giải thích chi tiết cho việc cập nhật
                if old_value >= self.INF:
                    explanation = f'Tìm thấy đường đi mới từ {i} đến {j} qua đỉnh {k}:\n' \
                                  f'• Trước: không có đường đi (∞)\n' \
                                  f'• Qua đỉnh {k}: {dist[i][k]:.0f} + {dist[k][j]:.0f} = {dist[i][j]:.0f}\n' \
                                  f'• Kết quả: cập nhật d[{i}][{j}] = {dist[i][j]:.0f}'
```

Hình 11: Tính toán giải thuật

Vòng lặp ngoài (for k in range(n)): duyệt qua từng đỉnh *k* trong đồ thị để xem xét nó như một đỉnh trung gian. Tại mỗi bước lặp *k*, thuật toán sẽ kiểm tra xem: "Liệu đi qua đỉnh *k* này có giúp đường đi giữa các cặp đỉnh khác ngắn hơn không?".

Hai vòng lặp trong (for i và for j): duyệt qua tất cả các cặp đỉnh nguồn (*j*) và đỉnh đích (*j*) trong ma trận để thực hiện so sánh.

Điều kiện nới lỏng (Relaxation): if dist[i][k] + dist[k][j] < dist[i][j]:

Chương trình so sánh hai giá trị:

- Khoảng cách hiện tại từ *i* đến *j* (dist[i][j]).
- Khoảng cách đi vòng qua trung gian *k* (dist[i][k] + dist[k][j]).

Cập nhật kết quả: nếu đi qua *k* ngắn hơn, chương trình thực hiện 2 việc:

- Cập nhật khoảng cách: dist[i][j] = ... (Lưu lại độ dài đường đi mới ngắn hơn).
- Cập nhật truy vết: next_node[i][j] = next_node[i][k] (Lưu lại thông tin để sau này vẽ được mũi tên đường đi trên giao diện đồ họa).

3.1.2.4. Tái dựng đường đi ngắn nhất

```python
def get_path(self, next_matrix, start, end):
    if next_matrix[start][end] == -1:
        return []

    path = [start]
    current = start
    while current != end:
        current = next_matrix[current][end]
        if current == -1:
            return []
        path.append(current)
    return path
```

Hình 12: Tái dựng đường đi ngắn nhất

Cơ chế hoạt động của hàm get_path:

- **Đầu vào:** Hàm nhận vào ma trận truy vết next_matrix, điểm bắt đầu start và điểm kết thúc end.
- **Kiểm tra tồn tại:** Đầu tiên, kiểm tra next_matrix[start][end]. Nếu giá trị là -1, nghĩa là không có đường đi, trả về danh sách rỗng [].
- **Vòng lặp truy vết (while):**

+ Bắt đầu từ đỉnh current = start.

+ Liên tục tìm đỉnh tiếp theo bằng cách tra cứu: current = next_matrix[current][end].

+ Thêm đỉnh tìm được vào danh sách path.

+ Lặp lại cho đến khi current trùng với end (đã đến đích).

Kết quả: Trả về một danh sách các đỉnh theo đúng thứ tự di chuyển (Ví dụ: [0, 2, 4] nghĩa là đi từ 0 - 2 - 4).

3.1.2.5. Hàm tạo đồ thị mẫu

```
def create_graph_by_type(self, n, graph_type):
    try:
        import random

        matrix = [[self.INF if i != j else 0 for j in range(n)] for i in range(n)]

        if graph_type == "complete":
            for i in range(n):
                for j in range(n):
                    if i != j:
                        matrix[i][j] = random.randint(1, 10)

        elif graph_type == "cycle":
            for i in range(n):
                next_vertex = (i + 1) % n
                matrix[i][next_vertex] = random.randint(1, 8)
                if random.random() < 0.4:
                    matrix[next_vertex][i] = random.randint(1, 8)
            for _ in range(min(2, n // 2)):
                i, j = random.sample(range(n), 2)
                matrix[i][j] = random.randint(5, 12)

        elif graph_type == "path":
            for i in range(n - 1):
                matrix[i][i + 1] = random.randint(1, 6)
                if random.random() < 0.5:
                    matrix[i + 1][i] = random.randint(1, 6)
            for _ in range(min(2, n // 2)):
                i, j = random.sample(range(n), 2)
                if abs(i - j) > 1:
                    matrix[i][j] = random.randint(8, 15)
```

```
elif graph_type == "star":
    center = 0
    for i in range(1, n):
        matrix[center][i] = random.randint(1, 5)
        matrix[i][center] = random.randint(1, 5)
    if n > 3:
        for _ in range(min(2, n // 3)):
            i, j = random.sample(range(1, n), 2)
            matrix[i][j] = random.randint(6, 12)

elif graph_type == "negative_cycle":
    # Tạo chu trình cơ bản
    for i in range(n - 1):
        matrix[i][i + 1] = random.randint(1, 5)
    matrix[n - 1][0] = random.randint(1, 5)
```

Hình 13: Hàm tạo đồ thị mẫu

Khởi tạo ma trận: matrix = [[self.INF ...]]. Tạo ma trận $n \times n$ với toàn bộ giá trị là vô cực, đường chéo chính là 0.

Tham số graph_type: Hàm nhận vào loại đồ thị người dùng muốn tạo để chạy logic sinh cạnh tương ứng:

- "complete": Tạo cạnh nối giữa tất cả các cặp đỉnh (đồ thị đầy đủ).
- "cycle": Tạo các cạnh nối vòng tròn khép kín.
- "star": Tạo một đỉnh trung tâm kết nối với tất cả đỉnh còn lại.

- "negative_cycle": Tạo đồ thị chứa chu trình âm để kiểm thử tính năng cảnh báo.

Gán trọng số: Sử dụng random.randint(1, 10) để sinh trọng số ngẫu nhiên cho các cạnh, tạo sự đa dạng cho dữ liệu thử nghiệm.

3.1.2.6. Hàm cập nhật đồ thị khi thay đổi số đỉnh

```
def create_matrix(self):
    try:
        n = int(self.node_entry.get())
        if n < 2 or n > 10:
            messagebox.showerror("Lỗi", "Số đỉnh phải từ 2 đến 10!")
            return

        self.current_example_index = 0

        # Xóa ma trận trước đó
        for widget in self.matrix_frame.winfo_children():
            widget.destroy()

        self.matrix_entries = []

        # Tạo tiêu đề
        tk.Label(self.matrix_frame, text="", width=3, bg="white").grid(row=0, column=0)
        for j in range(n):
            tk.Label(self.matrix_frame, text=str(j), width=6,
                     font=("Segoe UI", 10, "bold"), bg="white", fg="#2c3e50").grid(row=0, column=j+1)
```

```
        # Tạo các ô nhập ma trận
        for i in range(n):
            tk.Label(self.matrix_frame, text=str(i), width=3,
                     font=("Segoe UI", 10, "bold"), bg="white", fg="#2c3e50").grid(row=i+1, column=0)
            row_entries = []
            for j in range(n):
                entry = tk.Entry(self.matrix_frame, width=6, font=("Segoe UI", 10),
                                 justify=tk.CENTER, relief=tk.FLAT, bd=2)
                if i == j:
                    entry.insert(0, "0")
                    entry.config(state=tk.DISABLED, bg="#ecf0f1", fg="#7f8c8d")
                else:
                    entry.insert(0, str(self.INF))
                    entry.config(bg="white", fg="#2c3e50")
                    entry.bind('<KeyRelease>', self.on_matrix_change)
                    entry.bind('<FocusOut>', self.on_matrix_change)
                entry.grid(row=i+1, column=j+1, padx=3, pady=3)
                row_entries.append(entry)
            self.matrix_entries.append(row_entries)
```

Hình 14: Hàm cập nhật khi thay đổi số đỉnh

Lấy dữ liệu đầu vào: Hàm đọc giá trị từ ô nhập "Số đỉnh" (node_entry).

Kiểm tra hợp lệ (Validation): Đảm bảo số đỉnh *n* nằm trong khoảng cho phép (từ 2 đến 10). Nếu sai, hiện thông báo lỗi.

Làm sạch giao diện: Sử dụng vòng lặp destroy() để xóa toàn bộ các ô nhập liệu của ma trận cũ (nếu có).

Tạo lưới ma trận mới:

- Sử dụng 2 vòng lặp lồng nhau để tạo ra *n* \times *n* ô nhập liệu (tk.Entry).

- Tự động điền giá trị 0 cho đường chéo chính (khoảng cách từ đỉnh đến chính nó).
- Tự động điền giá trị INF (vô cực) cho các ô còn lại để người dùng tiện nhập liệu.

3.1.2.7. Tạo và vẽ đồ thị mẫu

```
def load_example(self):
    try:
        import random

        n = len(self.matrix_entries)
        if n < 2:
            messagebox.showinfo("Thông báo", "Cần tạo ma trận trước!")
            return

        # Xóa ma trận hiện tại
        for i in range(n):
            for j in range(n):
                if i != j:
                    self.matrix_entries[i][j].delete(0, tk.END)

        # Các dạng đồ thị khác nhau
        graph_types = [
            "complete", "cycle", "path", "star", "negative_cycle", "random"
        ]

        graph_type = graph_types[self.current_example_index % len(graph_types)]
        matrix = self.create_graph_by_type(n, graph_type)

        if not matrix or len(matrix) != n:
            raise Exception("Ma trận không hợp lệ")

        # Cập nhật giao diện
        for i in range(n):
            for j in range(n):
                if i != j:
                    value = matrix[i][j] if matrix[i][j] != self.INF else self.INF
                    self.matrix_entries[i][j].insert(0, str(value))
```

Hình 15: Tạo và vẽ đồ thị mẫu

Chọn mẫu đồ thị: Hàm sử dụng biến current_example_index để xoay vòng qua các loại đồ thị khác nhau (Đầy đủ -> Vòng -> Đường đi -> Sao -> ...). Mỗi lần bấm nút sẽ ra một kiểu khác nhau.

Sinh dữ liệu: Gọi hàm create_graph_by_type (đã mô tả ở mục 3.2.5) để lấy về ma trận trọng số tương ứng.

Đồng bộ giao diện:

- Xóa dữ liệu cũ trong các ô nhập liệu.
- Điền giá trị trọng số mới vào ma trận nhập liệu (matrix_entries) để người dùng nhìn thấy các con số cụ thể.

Kích hoạt vẽ: Sau khi điền số liệu, hàm sẽ gọi lệnh cập nhật để vẽ đồ thị mới lên màn hình.

3.1.2.8. Hàm cập nhật trọng số của cạnh khi chỉnh sửa

```
def get_matrix(self):
    if not hasattr(self, 'matrix_entries') or not self.matrix_entries:
        messagebox.showwarning("Cảnh báo", "Vui lòng tạo ma trận trước!")
        return None

    n = len(self.matrix_entries)
    matrix = []
    try:
        for i in range(n):
            row = []
            for j in range(n):
                value_str = self.matrix_entries[i][j].get().strip()
                if not value_str:
                    messagebox.showerror("Lỗi", f"Ô [{i}][{j}] không được để trống!")
                    return None

                try:
                    value = int(value_str)
                    row.append(value)
                except ValueError:
                    messagebox.showerror("Lỗi", f"Giá trị tại [{i}][{j}] phải là số nguyên!\nGiá trị hiện tại: '{value_str}'")
                    return None

            matrix.append(row)
        return matrix
    except Exception as e:
        messagebox.showerror("Lỗi", f"Có lỗi khi đọc ma trận: {str(e)}")
        return None
```

Hình 16: Hàm cập nhật trọng số của cạnh

Thu thập dữ liệu: Hàm duyệt qua toàn bộ lưới các ô nhập liệu (self.matrix_entries) trên giao diện để lấy giá trị trọng số mà người dùng đã nhập.

Chuyển đổi dữ liệu: Sử dụng int(value_str) để chuyển đổi chuỗi ký tự người dùng nhập thành số nguyên để máy tính có thể tính toán.

Kiểm tra tính hợp lệ (Validation):

- Kiểm tra xem ô có bị để trống hay không.
- Sử dụng khối try...except ValueError để bắt lỗi nếu người dùng nhập ký tự không phải là số (ví dụ: nhập chữ "a" hoặc để trống).

Kết quả: Nếu tất cả dữ liệu hợp lệ, hàm trả về một ma trận 2 chiều (list of lists) sẵn sàng cho thuật toán Floyd-Warshall xử lý

3.1.2.9. Cập nhật,vẽ ma trận và hiển thị đồ thị

```
def draw_graph(self, matrix, k=-1, i=-1, j=-1):
    try:
        # Xóa canvas trước đó
        for widget in self.canvas_frame.winfo_children():
            widget.destroy()

        # Tạo figure
        fig = Figure(figsize=(7, 6), dpi=100, facecolor='white')
        ax = fig.add_subplot(111)

        G = nx.DiGraph()
        n = len(matrix)

        # Thêm tất cả các đỉnh trước
        for node in range(n):
            G.add_node(node)

        # Thêm cạnh - bỏ qua các giá trị -∞
        for row in range(n):
            for col in range(n):
                if (row != col and
                    matrix[row][col] < self.INF and
                    matrix[row][col] != float('-inf')):
                    try:
                        weight = int(matrix[row][col])
                        G.add_edge(row, col, weight=weight)
                    except (ValueError, OverflowError):
                        continue
```

Hình 17: Cập nhật,vẽ ma trận và hiển thị đồ thị

Khởi tạo môi trường vẽ:

- Xóa các widget cũ (widget.destroy()) để tránh hình ảnh bị chồng chéo khi cập nhật liên tục.
- Sử dụng Figure của thư viện Matplotlib thay vì Canvas thường để hình ảnh sắc nét và chuyên nghiệp hơn.

Xây dựng cấu trúc đồ thị:

- Chuyển đổi dữ liệu từ ma trận kề sang đối tượng đồ thị nx.DiGraph của thư viện NetworkX.
- Chỉ thêm các cạnh có trọng số nhỏ hơn vô cực (INF).

Trực quan hóa thông minh:

- **Tô màu động:** Các đỉnh đang tham gia vào quá trình tính toán (đỉnh trung gian *k*, đỉnh đầu *i*, đỉnh cuối *j*) sẽ được tô màu nổi bật (Đỏ/Vàng/Cam) để người dùng dễ theo dõi luồng thuật toán.

- **Hiển thị trọng số:** Các giá trị trọng số được vẽ lên cạnh, có khung nền bao quanh để dễ đọc.

Tích hợp giao diện: Sử dụng FigureCanvasTkAgg để nhúng biểu đồ Matplotlib vào cửa sổ ứng dụng Tkinter.

3.1.2.10. Hàm hiển thị kết quả và xử lý ngoại lệ

```
def update_paths_display(self, matrix, next_matrix):
    self.paths_display.delete(1.0, tk.END)
    n = len(matrix)

    # Kiểm tra chu trình âm
    has_negative_cycle = any(matrix[i][i] < 0 for i in range(n))

    if has_negative_cycle:
        self.paths_display.insert(tk.END, "⚠️ CẢNH BÁO CHU TRÌNH ÂM!\n")
        self.paths_display.insert(tk.END, "=" * 45 + "\n\n")

        negative_vertices = []
        for i in range(n):
            if matrix[i][i] < 0:
                negative_vertices.append(i)

        self.paths_display.insert(tk.END, f"Các đỉnh có chu trình âm: {negative_vertices}\n\n")

        start_idx = "1.0"
        end_idx = "3.0"
        self.paths_display.tag_add("warning", start_idx, end_idx)
        self.paths_display.tag_config("warning", foreground="#e74c3c", font=("Consolas", 10, "bold"))

        self.paths_display.insert(tk.END, "Một số khoảng cách có thể là -∞\n")
        self.paths_display.insert(tk.END, "do ảnh hưởng của chu trình âm.")

    else:
        self.paths_display.insert(tk.END, "ĐƯỜNG ĐI NGẮN NHẤT XA NHẤT:\n")
        self.paths_display.insert(tk.END, "=" * 45 + "\n\n")

        max_distance = -1
        best_start = -1
        best_end = -1
        best_path = []
```

Hình 18: Hàm hiển thị kết quả và xử lý ngoại lệ

Tự động hóa hiển thị: Thay vì yêu cầu người dùng nhập thủ công điểm bắt đầu và kết thúc như các chương trình cũ, hàm này tự động quét toàn bộ ma trận kết quả để tìm ra các đường đi tiêu biểu.

Phát hiện Chu trình âm (Negative Cycle):

- Hàm kiểm tra đường chéo chính của ma trận: if matrix[i][i] < 0.
- Nếu phát hiện giá trị âm, chương trình lập tức in cảnh báo "CẢNH BÁO CHU TRÌNH ÂM" và liệt kê danh sách các đỉnh bị lỗi. Đây là tính năng an toàn giúp người dùng biết kết quả tính toán có thể không chính xác (do lỗi vô cực âm).

Hiển thị Đường đi tiêu biểu:

- Trong trường hợp dữ liệu hợp lệ (else), hàm sẽ tìm đường đi ngắn nhất có tổng trọng số lớn nhất ("Đường đi ngắn nhất xa nhất") để hiển thị làm mẫu.
- Thông tin được xuất ra khung Text bên phải giao diện gồm: Đỉnh bắt đầu, Đỉnh kết thúc, Tổng trọng số và lộ trình chi tiết.

3.1.2.11. Tìm đường đi ngắn nhất

```
        return None
def start_algorithm(self):
    if not hasattr(self, 'matrix_entries') or not self.matrix_entries:
        messagebox.showwarning("Cảnh báo", "Vui lòng tạo ma trận trước khi bắt đầu!")
        return

    matrix = self.get_matrix()
    if matrix is None:
        return

    if not self.validate_matrix(matrix):
        return

    self.steps = []
    self.current_step = 0
    self.compute_floyd_warshall(matrix)

    if len(self.steps) > 0:
        self.is_running = True
        self.start_btn.config(state=tk.DISABLED)
        self.pause_btn.config(state=tk.NORMAL)
        self.next_btn.config(state=tk.DISABLED)
        self.log_message("Bắt đầu mô phỏng giải thuật Floyd-Warshall")
        self.run_next_step()
```

Hình 19: Tìm đường đi ngắn nhất

Xác thực dữ liệu (Validation): Trước khi chạy, hàm gọi validate_matrix để đảm bảo ma trận đầu vào là hợp lệ (ma trận vuông, đường chéo chính bằng 0, có ít nhất một cạnh...). Điều này giúp tránh lỗi crash chương trình khi đang chạy.

Kết nối thuật toán: Gọi hàm compute_floyd_warshall(matrix) để thực hiện toàn bộ việc tính toán toán học và lưu kết quả vào danh sách self.steps.

Quản lý trạng thái ứng dụng:

- Chuyển trạng thái is_running = True.
- Khóa nút "Bắt đầu", mở nút "Tạm dừng" để người dùng không bấm nhầm khi đang chạy.

Kích hoạt mô phỏng: Gọi hàm run_next_step() để bắt đầu quá trình hiển thị trực quan từng bước (animation) lên màn hình.

3.1.2.12. Hàm đặt lại trạng thái ban đầu

```
def reset_algorithm(self):
    self.is_running = False
    self.current_step = 0
    self.steps = []

    # Xóa lịch sử đúng cách
    for item in self.graph_history:
        if item['widget'].winfo_exists():
            item['widget'].destroy()
    self.graph_history = []

    self.start_btn.config(state=tk.NORMAL, bg="#27ae60")
    self.pause_btn.config(state=tk.DISABLED)
    self.next_btn.config(state=tk.DISABLED)

    self.update_start_button_state()

    self.matrix_display.delete(1.0, tk.END)
    self.paths_display.delete(1.0, tk.END)
    self.log_text.delete(1.0, tk.END)
    self.status_label.config(text="Đã đặt lại")

    # Xóa canvas chính
    for widget in self.canvas_frame.winfo_children():
        widget.destroy()
```

Hình 20: Tạo khung

```
# Xóa container các bước
for widget in self.steps_container.winfo_children():
    widget.destroy()

# Tạo lại thông báo ban đầu
initial_container = tk.Frame(self.steps_container, bg="#f8f9fa", relief=tk.FLAT, bd=1)
initial_container.pack(fill=tk.X, padx=20, pady=20)

self.initial_message = tk.Label(initial_container,
                                text="🚀 Nhấn 'Bắt đầu' để xem các bước giải chi tiết...",
                                font=("Segoe UI", 12), bg="#f8f9fa", fg="#7f8c8d",
                                pady=30)
self.initial_message.pack()

# Đặt lại vị trí cuộn
self.main_canvas.yview_moveto(0.0)

self.log_message("Đã đặt lại ứng dụng")
self.draw_initial_placeholder()
```

Hình 21: Hiện kết quả

Dừng quy trình: Đặt biến cờ is_running = False để ngắt vòng lặp hoạt hình (animation) nếu chương trình đang chạy dở.

Dọn dẹp bộ nhớ: Xóa toàn bộ danh sách các bước (steps) và lịch sử hiển thị (graph_history) để giải phóng tài nguyên.

Làm sạch giao diện (UI):

- Xóa trắng các khung hiển thị: Nhật ký (Log), Ma trận số và Kết quả đường đi.
- Xóa đồ thị hiện tại trên Canvas.

Khôi phục trạng thái:

- Bật lại nút "Bắt đầu" (Enable).
- Vô hiệu hóa nút "Tạm dừng" và "Bước tiếp".
- Đưa màn hình về trạng thái chờ (Placeholder) để người dùng sẵn sàng nhập dữ liệu cho bài toán mới.

3.1.2.13. Hàm chính

```
def main():
    root = tk.Tk()
    app = FloydWarshallApp(root)
    root.mainloop()

if __name__ == "__main__":
    main()
```

Hình 22: Khởi tạo đồ thị

Khởi tạo môi trường (tk.Tk()): Tạo ra cửa sổ gốc (root window) của hệ thống giao diện Tkinter. Đây là khung nền tảng để gắn các widget lên.

Khởi tạo ứng dụng: Gọi class FloydWarshallApp(root) để bắt đầu thiết lập giao diện (setup_ui), biến số và logic chương trình như đã định nghĩa ở các mục trước.

Vòng lặp sự kiện (mainloop):

- Lệnh root.mainloop() đưa chương trình vào trạng thái "lắng nghe".

- Nó giữ cho cửa sổ luôn hiển thị và chờ đợi các thao tác từ người dùng (như click chuột, nhập phím) để phản hồi lại. Chương trình sẽ chạy mãi cho đến khi người dùng tắt cửa sổ.

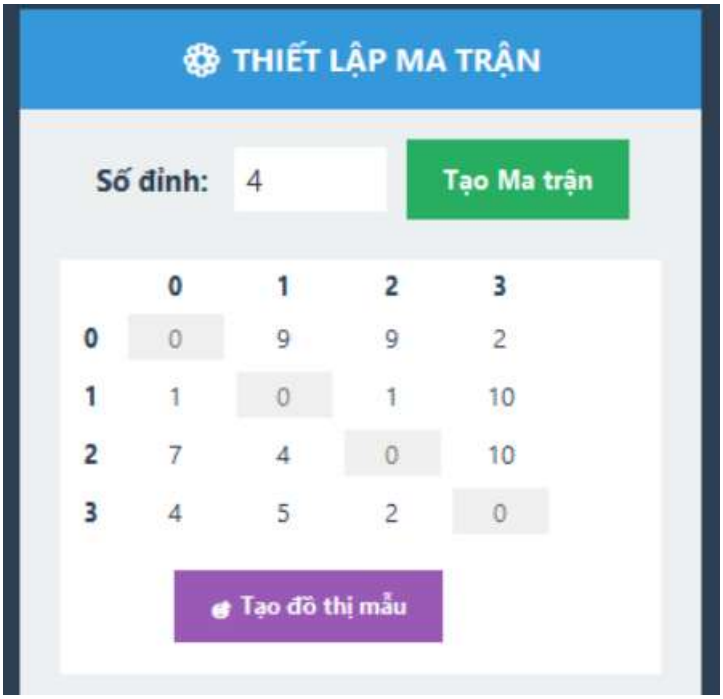
CHƯƠNG 4: KẾT QUẢ NGHIÊN CỨU

4.1. Giao diện người dùng

4.1.1. Giao diện chính

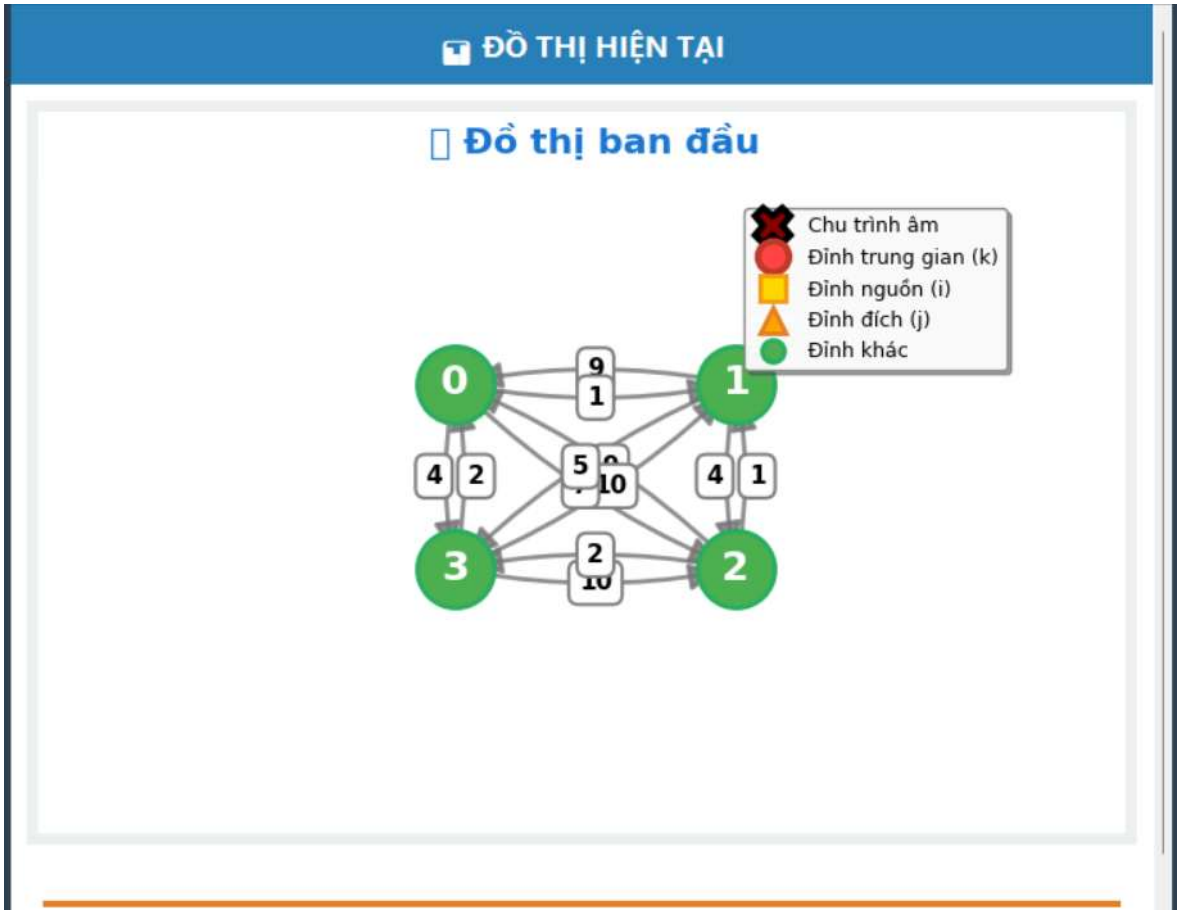
Hình 23: Màn hình giao diện

4.1.2. Khung nhập khoảng cách giữa các đỉnh trong ma trận



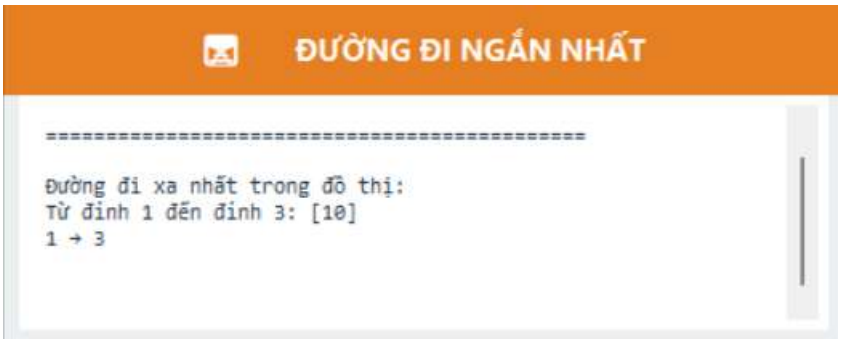
Hình 24: Khung nhập ma trận

4.1.3. Hiển thị đồ thị trực quan



Hình 25: Vị trí nhập đỉnh, điểm đầu và điểm đích

4.1.4. Xem đường đi ngắn nhất



Hình 26: đường đi ngắn nhất

4.1.5. Xem nhật kí thực hiện

Hình 27: Xem nhật kí thực hiện

4.2. Hiệu năng

- **Tính toán hiệu quả:**
    - Giải thuật Floyd-Warshall được triển khai thành công với thời gian chạy O(V^3), phù hợp với các đồ thị nhỏ (tối đa 10 đỉnh trong giao diện này).
    - Ma trận trọng số được cập nhật động khi người dùng chỉnh sửa, và kết quả luôn phản ánh chính xác các thay đổi.
- **Trực quan hóa:**
    - Đồ thị được vẽ bằng thư viện NetworkX kết hợp Matplotlib, cho phép hiển thị các cạnh và trọng số một cách trực quan.
    - Đường đi ngắn nhất được tô màu đỏ, giúp người dùng dễ dàng nhận biết.

CHƯƠNG 5: KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

5.1. Kết Luận

Qua quá trình nghiên cứu và thực hiện đề tài "Mô phỏng giải thuật Floyd-Warshall tìm đường đi ngắn nhất", đồ án đã đạt được các kết quả quan trọng sau:

- **Về mặt lý thuyết:** Đã tìm hiểu và nắm vững nguyên lý hoạt động của giải thuật Floyd-Warshall, hiểu rõ tư tưởng quy hoạch động và cách thức giải quyết bài toán tìm đường đi ngắn nhất giữa mọi cặp đỉnh.
- **Về mặt thực tiễn:**
    - Đã xây dựng thành công ứng dụng minh họa trực quan bằng ngôn ngữ Python, sử dụng các thư viện đồ họa mạnh mẽ như Tkinter, NetworkX và Matplotlib.
    - Giao diện người dùng được thiết kế hiện đại (Modern UI), thân thiện và dễ sử dụng với chế độ hiển thị tối (Dark Mode) giúp nâng cao trải nghiệm người dùng.
    - Chương trình hỗ trợ đầy đủ các tính năng từ cơ bản đến nâng cao: nhập liệu ma trận động, sinh đồ thị mẫu ngẫu nhiên, mô phỏng từng bước (Step-by-step), và tự động phát hiện các trường hợp ngoại lệ như chu trình âm.
- **Về giá trị giáo dục:** Ứng dụng hoạt động ổn định, chính xác và trực quan, có thể được sử dụng làm công cụ hỗ trợ giảng dạy và học tập hiệu quả, giúp sinh viên dễ dàng hình dung quy trình tính toán trừu tượng của thuật toán.

5.2. Hướng phát triển

Mặc dù ứng dụng đã đáp ứng tốt các yêu cầu đề ra ban đầu, nhưng để sản phẩm hoàn thiện hơn và có khả năng ứng dụng rộng rãi trong thực tế, một số hướng phát triển tiềm năng trong tương lai bao gồm:

- **Nâng cấp khả năng tương tác đồ họa:**

- Cho phép người dùng kéo thả (Drag & Drop) các đỉnh trực tiếp trên màn hình để thay đổi vị trí hoặc cấu trúc đồ thị thay vì chỉ nhập số liệu qua ma trận.
- Bổ sung tính năng Undo/Redo (Quay lại bước trước) để người dùng dễ dàng kiểm tra lại nếu lỡ thao tác sai hoặc muốn xem lại bước giải vừa qua.

- **Mở rộng quy mô và hiệu năng:**
  - Tối ưu hóa thuật toán vẽ để hỗ trợ các đồ thị có kích thước lớn hơn (trên 20 đỉnh) mà không gây giật lag.
  - Cải tiến thuật toán layout để hiển thị các đồ thị phức tạp một cách rõ ràng, hạn chế việc các cạnh bị chồng chéo lên nhau.

DANH MỤC TÀI LIỆU THAM KHẢO

[1] T. H. Nam, TÀI LIỆU GIẢNG DẠY MÔN LÝ THUYẾT ĐỒ THỊ, Trà Vinh, 2013.

[2] Viblo algorithm ,Thuật toán Floyd-Warshall,Việt Nam,2021.

Nguồn: https://viblo.asia/p/thuat-toan-floyd-warshall-GrLZDBng5k0

[3] Michael Sambol, Thuật toán Floyd–Warshall trong 4 phút, Hoa Kỳ,2016.

Nguồn: https://www.youtube.com/watch?v=4OQeCuLYj-4

[4] Dương Thế Hưng,Các thuật toán tìm đường đi ngắn nhất trên đồ thị có trọng số (Phần 3) – Bellman-Ford,Việt Nam, 2023.

Nguồn: https://codedream.edu.vn/bellman-ford/