ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA TOÁN TIN HỌC

※※※※※

ĐỒ ÁN CUỐI KỲ

Lý thuyết đồ thị (Chu trình Euler và chu trình Hamilton)

Thành phố Hồ Chí Minh, ngày 11 tháng 06 năm 2025

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA TOÁN TIN HỌC

※※※※※

Lý thuyết đồ thị (Chu trình Euler và chu trình Hamilton)

Tiểu luận môn: Phương Pháp Số Cho Khoa Học Dữ Liệu

Giảng viên hướng dẫn: Nguyễn Thị Hoài Thương

Nhóm thực hiện: 9

Lê Hoàng An	22110002
Trần Duy An	22110008
Phạm Hoàng Dũng	22110043
Nguyễn Kỳ Anh	19110263

Thành phố Hồ Chí Minh, ngày 11 tháng 06 năm 2025

Mục lục

PHẦN MỞ ĐẦU

Lời đầu tiên, nhóm xin gửi đến quý Thầy/Cô và các bạn sinh viên Trường Đại học Khoa học Tự nhiên lời chào trân trọng và lời chúc sức khỏe. Bài tiểu luận này được thực hiện trong khuôn khổ học phần *Phương pháp số cho khoa học dữ liệu*, với mong muốn tìm hiểu và vận dụng kiến thức đã học vào một chủ đề nền tảng trong khoa học dữ liệu – đó là **Lý thuyết đồ thị**, cụ thể là hai khái niệm quan trọng: **Chu trình Euler** và **Chu trình Hamilton**.

Trong bối cảnh các bài toán trên đồ thị ngày càng đóng vai trò quan trọng trong xử lý dữ liệu, mạng lưới thông tin, logistics và nhiều ứng dụng thực tiễn khác, việc hiểu rõ các cấu trúc đặc biệt trong đồ thị như chu trình Euler và Hamilton không chỉ có giá trị lý thuyết mà còn hỗ trợ định hướng cho các phương pháp tối ưu hóa và mô hình hóa dữ liệu phức tạp.

Thông qua việc thực hiện đề tài này, nhóm không chỉ có cơ hội củng cố lại kiến thức về lý thuyết đồ thị và thuật toán, mà còn rèn luyện thêm các kỹ năng nghiên cứu tài liệu, làm việc nhóm, trình bày và lập luận vấn đề một cách logic và khoa học. Đây cũng là bước khởi đầu để làm quen với phương pháp tiếp cận bài toán một cách có hệ thống trong lĩnh vực khoa học dữ liệu.

Việc hiểu rõ bản chất của các chu trình trong đồ thị, cũng như cách biểu diễn và đánh giá tính liên thông, sẽ là tiền đề quan trọng để học viên có thể tiếp cận tốt hơn với các thuật toán phức tạp hơn sau này trong khoa học dữ liệu và công nghệ thông tin.

Nhóm hy vọng rằng bài viết này sẽ mang đến một cái nhìn tổng quan, dễ tiếp cận và hữu ích cho những ai đang quan tâm đến ứng dụng của lý thuyết đồ thị trong khoa học dữ liệu và tin học ứng dụng. Dù đã có nhiều cố gắng trong quá trình thực hiện, nhóm không tránh khỏi những thiếu sót. Kính mong nhận được ý kiến đóng góp quý báu từ quý Thầy/Cô để hoàn thiện hơn trong những lần sau.

Xin chân thành cảm ơn!

1 Hình thành ý tưởng:

1.1 Bối cảnh ra đời

Trong lĩnh vực khoa học dữ liệu, việc tối ưu hóa đường đi và mạng lưới là một trong những bài toán cốt lõi, xuất hiện rộng rãi trong logistics, quy hoạch giao thông, và tối ưu hóa hệ thống. Tuy nhiên, để giải quyết hiệu quả các bài toán này, cần có một nền tảng lý thuyết vững chắc và các thuật toán phù hợp. Nhận thấy tầm quan trọng của việc áp dụng kiến thức lý thuyết vào thực tiễn, đặc biệt là trong khuôn khổ môn học "Phương pháp số cho Khoa học dữ liệu", chủ đề về Chu trình Euler và Chu trình Hamilton đã được lựa chọn để cung cấp một cách tiếp cận toán học và thuật toán để giải quyết các vấn đề tối ưu hóa đường đi.

1.2 Sự mô hình hóa bài toán

Nhiều vấn đề thực tiễn, như bài toán giao hàng hay tìm đường đi, có thể được mô hình hóa một cách hiệu quả dưới dạng đồ thị. Khi tổng hợp các điểm cần ghé thăm và các tuyến đường giữa chúng, chúng ta có thể biểu diễn chúng thành một đồ thị, trong đó:

- **Mỗi đỉnh (node)** đại diện cho một địa điểm cụ thể (ví dụ: kho hàng, điểm giao hàng).
- **Mỗi cạnh (edge)** là một tuyến đường hoặc kết nối giữa hai địa điểm.

Ví dụ: nếu có một tuyến đường từ điểm A đến điểm B, trên đồ thị sẽ có một cạnh nối A và B. Mô hình hóa này giúp chúng ta hình dung và áp dụng các công cụ của lý thuyết đồ thị để phân tích và tìm kiếm giải pháp.

1.3 Nguyên lý ứng dụng và giải quyết bài toán

Tuy nhiên, việc chỉ mô hình hóa là chưa đủ; cần có các nguyên lý và thuật toán để tìm ra giải pháp tối ưu. Đây là lúc Chu trình Euler và Chu trình Hamilton trở nên quan trọng.

- **Chu trình Euler** được áp dụng khi chúng ta muốn đi qua *tất cả các cạnh* của đồ thị đúng một lần và quay về điểm xuất phát (ví dụ: một người đưa thư cần đi qua tất cả các con đường trong một khu vực).
- **Chu trình Hamilton** được áp dụng khi mục tiêu là đi qua *tất cả các đỉnh* của đồ thị đúng một lần và quay về điểm xuất phát (ví dụ: một nhân viên giao hàng cần ghé thăm tất cả các địa điểm cụ thể).

Mặc dù có những điểm khác biệt và độ phức tạp riêng, cả hai loại chu trình này đều cung cấp khuôn khổ để giải quyết bài toán tìm đường đi tối ưu, đảm bảo rằng tất cả các yêu cầu về điểm đến hoặc tuyến đường đều được đáp ứng một cách hiệu quả nhất. Việc tìm kiếm các chu trình này không chỉ giúp tối ưu hóa quãng đường mà còn đảm bảo tính khả thi của lộ trình trong các bài toán thực tiễn.

Việc toán học hóa các nguyên lý này được thực hiện thông qua việc phân tích cấu trúc đồ thị, sử dụng các thuộc tính của ma trận kề và áp dụng các thuật toán chuyên biệt để xác định sự tồn tại và tìm kiếm các chu trình mong muốn, từ đó đưa ra lời giải cho các bài toán tối ưu hóa phức tạp.

2 Cơ sở lý thuyết

2.1 Ma trận kề

2.1.1 Định nghĩa

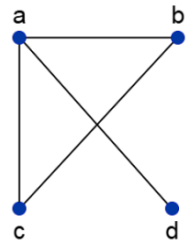
Định nghĩa

Ma trận kề là một ma trận vuông dùng để biểu diễn một đồ thị, trong đó:

- Các hàng và cột tương ứng với các đỉnh của đồ thị.
- Phần tử tại hàng i, cột j (ký hiệu là $A[i][j]$) thể hiện mối liên kết giữa đỉnh i và đỉnh j:

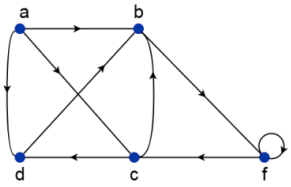
$$A[i][j] = \begin{cases} 1 & \text{nếu có cạnh nối từ } i \text{ đến } j \\ 0 & \text{nếu không có cạnh} \end{cases}$$

Ví Dụ - Ma trận kề đồ thị vô hướng



	a	b	c	d
a	0	1	1	1
b	1	0	1	0
c	1	1	0	0
d	1	0	0	0

Ví Dụ - Ma trận kề đồ thị có hướng



	a	b	c	d	f
a	0	1	1	1	0
b	0	0	0	0	1
c	0	1	0	1	0
d	0	1	0	0	0
f	0	0	1	0	1

2.1.2 Tính chất

Tính chất

Với một **đồ thị vô hướng** G có n đỉnh và m cạnh, ma trận kề $A = [a_{ij}]$ thỏa các tính chất sau:

- Ma trận A là đối xứng, tức là $a_{ij} = a_{ji}$ với mọi i, j, do các cạnh không có hướng.
- Tổng tất cả các phần tử trong ma trận bằng **2 lần số cạnh**:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij} = 2m$$

- Bậc của một đỉnh u được tính bằng tổng các phần tử trên hàng tương ứng:

$$\deg(u) = \sum_{j=1}^{n} a_{uj}$$

- Do tính đối xứng, bậc của đỉnh u cũng có thể được tính bằng tổng các phần tử trên cột tương ứng:

$$\deg(u) = \sum_{i=1}^{n} a_{iu}$$

Tính chất

Với một **đồ thị có hướng** G gồm n đỉnh và m cung, ma trận kề $A = [a_{ij}]$ có các tính chất sau:

- Tổng tất cả các phần tử trong ma trận bằng **số cung** của đồ thị:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij} = m$$

  vì mỗi cung chỉ được ghi nhận một lần từ đỉnh đầu đến đỉnh cuối.

- Tổng các phần tử trên hàng ứng với đỉnh u chính là **bán bậc ra** của đỉnh đó:

$$\deg^{+}(u) = \sum_{j=1}^{n} a_{uj}$$

- Tổng các phần tử trên cột ứng với đỉnh u chính là **bán bậc vào** của đỉnh đó:

$$\deg^{-}(u) = \sum_{i=1}^{n} a_{iu}$$

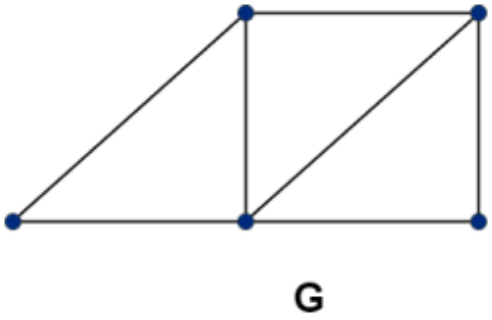
2.2 Đồ thị liên thông

2.2.1 Định nghĩa

Đồ thị liên thông (vô hướng)

Một **đồ thị vô hướng** được gọi là **liên thông** nếu tồn tại một **đường đi** giữa mọi cặp đỉnh bất kỳ trong đồ thị.
Nói cách khác: Với mọi hai đỉnh u và v trong đồ thị, luôn tồn tại một dãy các cạnh liên tiếp nối từ u đến v.

Ví Dụ - Đồ thị liên thông

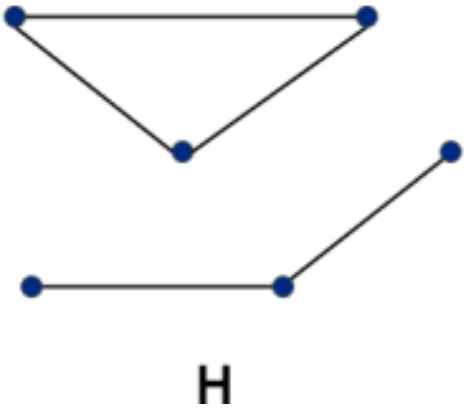


Hình 1: Đồ thị liên thông

Đồ thị G liên thông vì

- Mỗi đỉnh đều được nối với ít nhất một đỉnh khác bằng cạnh.
- Từ bất kỳ đỉnh nào, bạn đều có thể tìm được một chuỗi các cạnh để đến mọi đỉnh còn lại.

Ví Dụ - Đồ thị không liên thông



Hình 2: Đồ thị không liên thông

Đồ thị H không liên thông vì:

- Đồ thị được chia thành hai nhóm đỉnh tách rời nhau:
  - Nhóm thứ nhất gồm 3 đỉnh tạo thành một tam giác ở phía trên.
  - Nhóm thứ hai gồm 3 đỉnh nối với nhau thành một đường gấp khúc ở phía dưới.
- Không tồn tại cạnh nào nối giữa hai nhóm đỉnh này.
- Do đó, tồn tại các cặp đỉnh không có đường đi giữa chúng.

Kết luận: Đồ thị H không liên thông vì giữa một số cặp đỉnh không tồn tại đường đi.

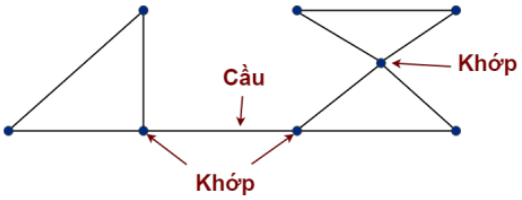
2.2.2 Tính liên thông

Định nghĩa

Cho đồ thị $G = (V, E)$ là đồ thị vô hướng liên thông.

a) Đỉnh v được gọi là **đỉnh khớp** nếu $G - v$ không liên thông ($G - v$ là đồ thị con của G có được bằng cách xóa v và các cạnh kề với v).

b) Cạnh e được gọi là **cầu** nếu $G - e$ không liên thông ($G - e$ là đồ thị con của G có được bằng cách xóa cạnh e).

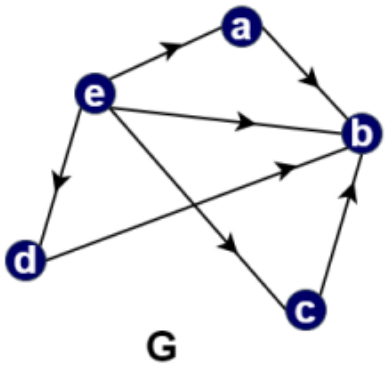


2.2.3 Liên thông mạnh - yếu

Đồ thị có hướng: Liên thông mạnh và yếu

- **Liên thông mạnh (Strongly connected)**: Đồ thị có hướng được gọi là liên thông mạnh nếu tồn tại **đường đi có hướng** từ mỗi đỉnh đến mọi đỉnh khác (nghĩa là tồn tại đường đi từ $u \to v$ và từ $v \to u$ với mọi cặp u, v).
- **Liên thông yếu (Weakly connected)**: Nếu khi bỏ hướng của các cạnh thì đồ thị trở thành một đồ thị **vô hướng liên thông**, thì đồ thị ban đầu được gọi là liên thông yếu.

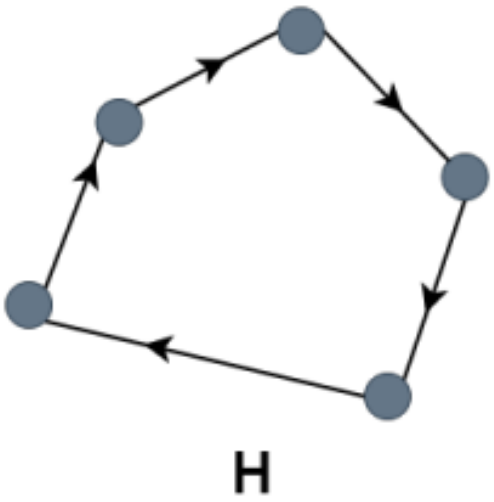
Ví Dụ - Đồ thị liên thông yếu



Hình 3: Đồ thị liên thông yếu

Đồ thị G liên thông yếu do đỉnh C không thể đi ngược về lại các đỉnh khác nên G liên thông yếu

Ví Dụ - Đồ thị liên thông mạnh



Hình 4: Đồ thị liên thông mạnh

Đồ thị H liên thông mạnh do từ bất kỳ đỉnh nào đều có thể tới mọi đỉnh khác theo hướng

2.3 Chu Trình

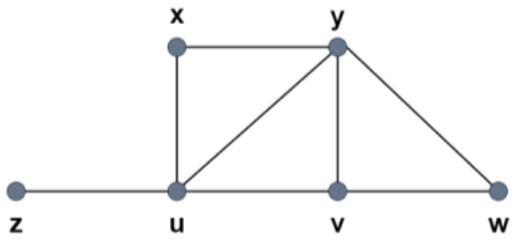
2.3.1 Định nghĩa

Định nghĩa

Cho $G = (V, E)$ là một đồ thị.

- Một **chu trình** (còn gọi là **mạch**) là một **đường đi** trong đó đỉnh đầu trùng với đỉnh cuối, tức là $x_0 = x_k$.
- Chu trình được gọi là **chu trình đơn** nếu không có cạnh/cung nào trong chu trình xuất hiện quá một lần.
- Chu trình được gọi là **chu trình sơ cấp** nếu:
  - Bắt đầu và kết thúc tại cùng một đỉnh: $x_0 = x_k$
  - Không có đỉnh nào (ngoại trừ x_0) xuất hiện quá một lần trong chu trình.

Ví Dụ - Chu Trình



- (u, y, w, v) là một đường đi độ dài 3.
- (z, u, y, y, v, u) là một đường đi đơn nhưng không sơ cấp.
- (u, y, w, v, u) là một chu trình. Có thể xem chu trình này như chu trình (w, v, u, y, w).

2.4 Đồ thị Euler

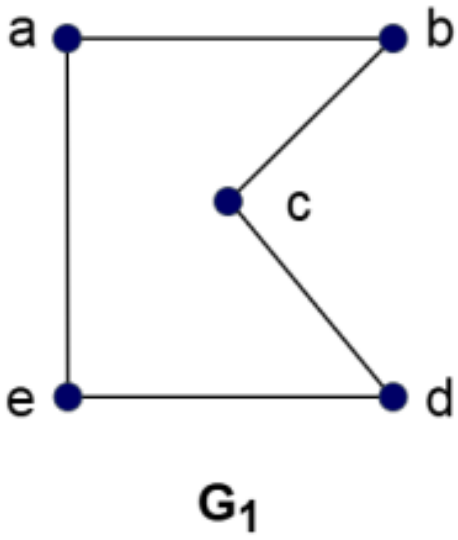
2.4.1 Định nghĩa

Định nghĩa: Đồ thị Euler

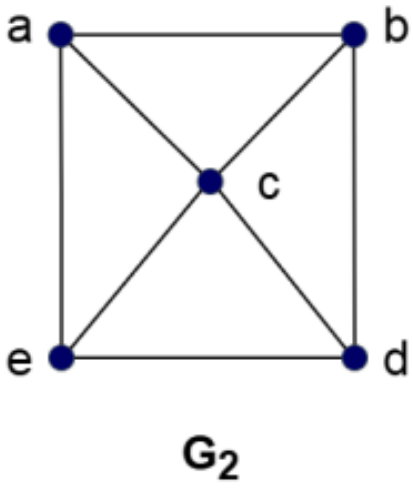
Một **đồ thị Euler** là một đồ thị (vô hướng hoặc có hướng) chứa một **chu trình Euler**, tức là một **chu trình đi qua mỗi cạnh đúng một lần**.

- Nếu đồ thị là **vô hướng**, thì chu trình Euler phải bắt đầu và kết thúc tại cùng một đỉnh, và mỗi cạnh chỉ được đi qua đúng một lần.
- Nếu đồ thị là **có hướng**, thì chu trình Euler là một dãy các cung sao cho:
  - Mỗi cung xuất hiện đúng một lần.
  - Chu trình bắt đầu và kết thúc tại cùng một đỉnh.
  - Hướng của các cung phải được tôn trọng.
- Một đồ thị có thể không có chu trình Euler nhưng vẫn có **đường đi Euler** nếu tồn tại một đường đi qua mỗi cạnh đúng một lần nhưng không cần quay lại đỉnh xuất phát.

Ví Dụ



Hình 5: Đồ thị Euler



Hình 6: Đồ thị không Euler

- **Đồ thị** G_1 là **Euler** vì tất cả các đỉnh đều có bậc chẵn (mỗi đỉnh có số cạnh kề là chẵn). Do đó, tồn tại một chu trình đi qua tất cả các cạnh đúng một lần và quay lại đỉnh xuất phát.
- **Đồ thị** G_2 **không phải Euler** vì có đỉnh có bậc lẻ (ví dụ: đỉnh c có bậc 4, nhưng các đỉnh còn lại có bậc lẻ). Để đồ thị là Euler, tất cả các đỉnh phải có bậc chẵn.

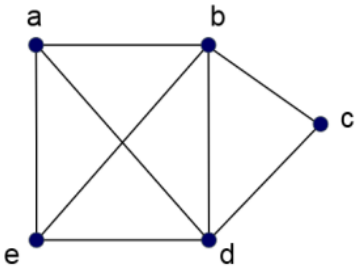
2.4.2 Đồ thị nửa Euler

Định nghĩa: Đồ thị nửa Euler

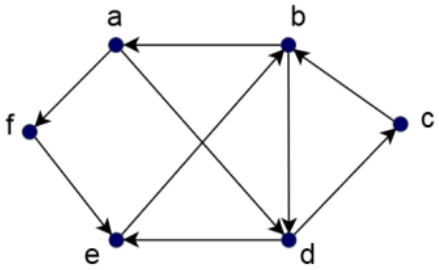
Một **đồ thị nửa Euler** là một đồ thị (vô hướng hoặc có hướng) có chứa một **đường đi Euler**, tức là một đường đi đi qua **mỗi cạnh đúng một lần** nhưng không cần quay về đỉnh xuất phát.

- Với **đồ thị vô hướng**: Đồ thị liên thông là **nửa Euler** khi và chỉ khi có đúng **hai đỉnh có bậc lẻ**, các đỉnh còn lại có bậc chẵn. Đường đi Euler sẽ bắt đầu tại một trong hai đỉnh bậc lẻ và kết thúc tại đỉnh còn lại.
- Với **đồ thị có hướng**: Đồ thị liên thông là **nửa Euler** khi:
  - Mọi đỉnh đều có số cung vào bằng số cung ra, **trừ** hai đỉnh:
    - Một đỉnh có số cung ra lớn hơn số cung vào đúng 1 (đỉnh bắt đầu).
    - Một đỉnh có số cung vào lớn hơn số cung ra đúng 1 (đỉnh kết thúc).

Ví Dụ - nửa Euler



H_1



H_2

- **Đồ thị** H_1 là **nửa Euler** vì có đúng hai đỉnh bậc lẻ đỉnh a và đỉnh e đều có bậc là 3 và các đỉnh còn lại là bậc chẵn
- **Đồ thị** H_2 **nửa Euler** vì $\deg^+(a) - \deg^-(a) = \deg^-(e) - \deg^+(e) = 1$ và các bán bậc còn lại đều bằng nhau nên đồ thị nửa Euler .

2.4.3 Ví dụ minh họa cách tìm chu trình Euler

1. Đồ thị và các đỉnh

Giả sử chúng ta có một đồ thị vô hướng $G = (V, E)$ với 5 đỉnh và 6 cạnh:

- **Đỉnh:** $V = \{A, B, C, D, E\}$
- **Cạnh:** $E = \{(A, B), (B, C), (C, A), (C, D), (D, E), (E, C)\}$

Đồ thị này có thể đi qua tất cả các cạnh một lần duy nhất vì mỗi đỉnh đều có bậc chẵn (bậc của A, B, D, E là 2; bậc của C là 4).

2. Các bước tìm chu trình bằng tay

Chúng ta sẽ bắt đầu từ một đỉnh bất kỳ, ví dụ đỉnh A, và thực hiện các bước sau:

1. **Tìm một chu trình bất kỳ:** Bắt đầu tại đỉnh A và đi theo các cạnh chưa sử dụng để tạo một chu trình con.
   - Từ A đi đến B (sử dụng cạnh (A, B)).
   - Từ B đi đến C (sử dụng cạnh (B, C)).
   - Từ C đi đến A (sử dụng cạnh (C, A)).

   Chúng ta có chu trình con đầu tiên: $C_1 = (A \to B \to C \to A)$. Ba cạnh $(A, B), (B, C), (C, A)$ đã được sử dụng.

2. **Kiểm tra và tìm đường đi mới:** Hiện tại, chu trình C_1 đã hoàn thành nhưng vẫn còn các cạnh chưa được sử dụng trong đồ thị (các cạnh $(C, D), (D, E), (E, C)$). Ta cần tìm một đỉnh trên chu trình C_1 mà còn cạnh chưa đi qua. * Đỉnh A và B không còn cạnh nào. * Đỉnh C vẫn còn hai cạnh (C, D) và (E, C) chưa sử dụng.

   Chúng ta sẽ tiếp tục tìm một đường đi mới bắt đầu từ đỉnh C.

3. **Tìm một chu trình con khác:** Từ đỉnh C, ta đi theo các cạnh còn lại để tạo một chu trình con mới.
   - Từ C đi đến D (sử dụng cạnh (C, D)).
   - Từ D đi đến E (sử dụng cạnh (D, E)).
   - Từ E đi đến C (sử dụng cạnh (E, C)).

   Chúng ta có chu trình con thứ hai: $C_2 = (C \to D \to E \to C)$. Tất cả các cạnh trong đồ thị bây giờ đã được sử dụng.

4. **Ghép các chu trình:** Ghép chu trình C_2 vào chu trình C_1 tại điểm gặp nhau là đỉnh C. * Chu trình ban đầu: $(A \to B \to \mathbf{C} \to A)$ * Khi đến C, ta sẽ đi theo chu trình con C_2 rồi mới tiếp tục chu trình ban đầu: $(A \to B \to \mathbf{C} \to D \to E \to \mathbf{C} \to A)$

3. Kết quả
Chu trình Euler cuối cùng của đồ thị là: $(A \to B \to C \to D \to E \to C \to A)$.

2.5 Hamilton

2.5.1 Định nghĩa

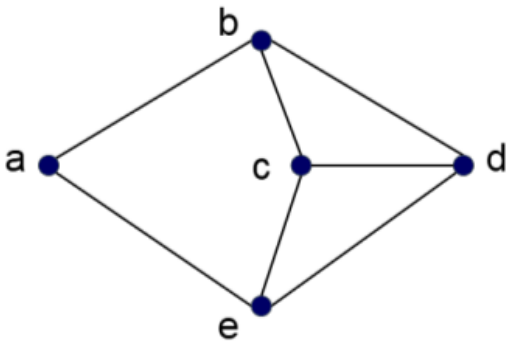
Định nghĩa: Đường đi Hamilton

Một **đường đi Hamilton** là một **đường đi đi qua mỗi đỉnh đúng một lần** (không lặp lại đỉnh), nhưng không cần phải quay lại đỉnh xuất phát.

- Nếu đường đi bắt đầu và kết thúc tại hai đỉnh khác nhau, ta gọi đó là **đường đi Hamilton**.
- Nếu đường đi bắt đầu và kết thúc tại cùng một đỉnh, thì đó là một **chu trình Hamilton**.

Lưu ý: Đường đi Hamilton chỉ quan tâm đến việc đi qua các đỉnh (không lặp), không quan trọng số lần đi qua cạnh.

Ví Dụ



- **Đường đi**: e, d, c, b, a.
- **Đường đi**: c, b, a, e, d.
- **Chu trình**: b, c, d, e, a, b.
- **Chu trình**: d, e, a, b, c, d.

2.5.2 Điều kiện để Hamilton

Điều kiện đủ (cho đồ thị đơn vô hướng).

Định lý Ore (1960). Cho đồ thị G có n đỉnh. Nếu $\deg(i) + \deg(j) \geq n \geq 3$ với i và j là hai đỉnh không kề nhau tùy ý thì G là Hamilton.

Định lý Dirac (1952). Cho đồ thị G có n đỉnh. Nếu $\deg(i) \geq n/2$ với i tùy ý thì G là Hamilton.

Điều kiện đủ cho đồ thị có hướng, đơn (không có khuyên và không có cạnh song song cùng chiều).

Định lý Camion (1959). Nếu G là đơn đồ thị đủ, liên thông mạnh thì G Hamilton.

Định lý Ghouila-Houri (1960). Nếu G là đơn đồ thị liên thông mạnh sao cho mọi đỉnh đều có bậc không nhỏ hơn n thì G Hamilton.

2.5.3 Ví dụ minh họa cách tìm chu trình Hamilton

1. Đồ thị và mục tiêu
Giả sử chúng ta có một đồ thị $G = (V, E)$ với 6 đỉnh:

- **Đỉnh:** $V = \{A, B, C, D, E, F\}$
- **Cạnh:** $E = \{(A, B), (B, C), (C, D), (D, E), (E, F), (F, A), (A, C)\}$

Mục tiêu là tìm một đường đi bắt đầu từ một đỉnh, đi qua tất cả các đỉnh khác mỗi đỉnh đúng một lần, và sau đó quay trở lại đỉnh ban đầu.

2. Các bước tìm chu trình bằng tay (thử và sai)
Chúng ta sẽ bắt đầu tại đỉnh A và thực hiện các bước thử nghiệm, nếu gặp ngõ cụt thì quay lại:

1. **Thử nghiệm 1 (đi từ A đến C):** Bắt đầu tại A, chúng ta chọn đi theo cạnh (A, C).
   - Đường đi hiện tại: $(A \to C)$
   - Từ C, chúng ta có thể đi đến B hoặc D. Thử đi đến B.
   - Đường đi: $(A \to C \to B)$
   - Từ B, các đỉnh lân cận là A và C đều đã được thăm. Không thể đi tiếp được. **Đây là ngõ cụt.**

   => Quay lại (backtracking): Chúng ta quay lại đỉnh C. Đã thử đi đến B, bây giờ chúng ta thử đi đến D.

2. **Thử nghiệm 2 (tiếp tục từ C):**
   - Đường đi hiện tại: $(A \to C)$

- Từ C, đi đến D.
- Đường đi: $(A \to C \to D)$
- Từ D, đi đến E.
- Đường đi: $(A \to C \to D \to E)$
- Từ E, đi đến F.
- Đường đi: $(A \to C \to D \to E \to F)$

Chúng ta đã thăm tất cả các đỉnh trừ B. Liệu có cạnh nào nối từ F đến B không? Không có. **Đây là một ngõ cụt khác.** => **Quay lại (backtracking):** Chúng ta phải quay lại, nhưng tất cả các lựa chọn từ F, E, D, C đều đã dẫn đến ngõ cụt. Cuối cùng, chúng ta quay lại đỉnh xuất phát A.

3. **Thử nghiệm 3 (đi từ A đến B):** Bắt đầu lại từ A, lần này chúng ta chọn đi theo cạnh (A, B) thay vì (A, C).

- Đường đi hiện tại: $(A \to B)$
- Từ B, đi đến C.
- Đường đi: $(A \to B \to C)$
- Từ C, đi đến D.
- Đường đi: $(A \to B \to C \to D)$
- Từ D, đi đến E.
- Đường đi: $(A \to B \to C \to D \to E)$
- Từ E, đi đến F.
- Đường đi: $(A \to B \to C \to D \to E \to F)$

Chúng ta đã đi qua tất cả 6 đỉnh của đồ thị. Bây giờ, chúng ta kiểm tra xem có cạnh nào nối từ đỉnh cuối cùng (F) quay về đỉnh ban đầu (A) không. Có cạnh (F, A) tồn tại.

3. Kết quả
Chúng ta đã tìm thấy một chu trình Hamilton: $(A \to B \to C \to D \to E \to F \to A)$.

3 Điều kiện đồ thị có chu trình Euler, chu trình Hamilton, thuật toán tìm chu trình Euler, chu trình Hamilton.

3.1 Điều kiện tồn tại chu trình

Việc xác định điều kiện tồn tại chu trình là bước đầu tiên và quan trọng để biết liệu bài toán có thể giải được bằng các thuật toán tương ứng hay không.

- **Điều kiện tồn tại chu trình Euler:**
  - **Đối với đồ thị vô hướng:** Một đồ thị liên thông có chu trình Euler khi và chỉ khi **tất cả các đỉnh của nó đều có bậc chẵn**.

    - *Giải thích trực quan:* Khi bạn vào một đỉnh bằng một cạnh, bạn phải có một cạnh khác để rời khỏi đỉnh đó. Nếu có một đỉnh bậc lẻ, bạn sẽ "mắc kẹt" hoặc không thể quay lại đỉnh đó sau khi đã sử dụng hết các cạnh.
  - **Đối với đồ thị có hướng:** Một đồ thị liên thông mạnh có chu trình Euler khi và chỉ khi với mọi đỉnh u, **bán bậc vào bằng bán bậc ra** ($deg^+(u) = deg^-(u)$).
    - *Giải thích trực quan:* Mỗi khi bạn đi vào một đỉnh, bạn phải có một đường để đi ra khỏi đỉnh đó mà không dùng lại cung đã vào.

- **Điều kiện tồn tại đường đi Euler (Đồ thị nửa Euler):**
  - **Đối với đồ thị vô hướng:** Một đồ thị liên thông có đường đi Euler khi và chỉ khi có **đúng hai đỉnh có bậc lẻ**, các đỉnh còn lại có bậc chẵn. Đường đi Euler sẽ bắt đầu tại một trong hai đỉnh bậc lẻ và kết thúc tại đỉnh còn lại.
  - **Đối với đồ thị có hướng:** Một đồ thị liên thông có đường đi Euler khi tất cả các đỉnh đều có bán bậc vào bằng bán bậc ra, trừ hai đỉnh: một đỉnh có bán bậc ra lớn hơn bán bậc vào đúng 1 (đỉnh bắt đầu của đường đi), và một đỉnh có bán bậc vào lớn hơn bán bậc ra đúng 1 (đỉnh kết thúc của đường đi).

- **Điều kiện tồn tại chu trình Hamilton:**
  - Không có điều kiện "khi và chỉ khi" đơn giản và dễ kiểm tra như chu trình Euler. Các định lý sau đây cung cấp điều kiện đủ (nếu thỏa mãn thì chắc chắn có Hamilton, nhưng không thỏa mãn thì chưa chắc không có):
    - **Định lý Ore (1960):** Cho đồ thị G có $n \geq 3$ đỉnh. Nếu với mọi cặp đỉnh không kề nhau $u, v \in V(G)$, ta có $deg(u) + deg(v) \geq n$, thì G là Hamilton.
      - *Ý nghĩa:* Định lý này nói rằng nếu đồ thị "đủ dày đặc" (tổng bậc của hai đỉnh bất kỳ không kề nhau đủ lớn), thì chắc chắn có chu trình Hamilton.
    - **Định lý Dirac (1952):** Cho đồ thị G có $n \geq 3$ đỉnh. Nếu $deg(u) \geq n/2$ với mọi đỉnh $u \in V(G)$, thì G là Hamilton.
      - *Ý nghĩa:* Đây là một trường hợp đặc biệt của định lý Ore. Nếu mọi đỉnh đều có bậc đủ cao (ít nhất một nửa số đỉnh), thì sẽ có chu trình Hamilton.
    - **Định lý Camion (1959) (đồ thị có hướng):** Nếu G là đơn đồ thị đủ, liên thông mạnh thì G Hamilton.
    - **Định lý Ghouila-Houri (1960) (đồ thị có hướng):** Nếu G là đơn đồ thị liên thông mạnh sao cho mọi đỉnh đều có bậc không nhỏ hơn n thì G Hamilton.

3.2 Thuật toán tìm chu trình Euler: Thuật toán Hierholzer

3.2.1 Tổng quan về Thuật toán Hierholzer

Chu trình Euler là một đường đi trong đồ thị đi qua **mỗi cạnh đúng một lần** và kết thúc tại đỉnh bắt đầu. Đây là một bài toán quan trọng trong tối ưu hóa lộ trình bao phủ.

Thuật toán Hierholzer là một phương pháp cổ điển, được phát triển bởi Carl Hierholzer vào thế kỷ 19, dùng để xây dựng một chu trình Euler trong đồ thị khi nó thỏa mãn các điều kiện tồn tại (đối với đồ thị vô hướng: liên thông và tất cả các đỉnh có bậc chẵn; đối với đồ thị có hướng: liên thông mạnh và bán bậc vào bằng bán bậc ra cho mọi đỉnh).

Nền tảng xây dựng thuật toán xuất phát trực tiếp từ chính **định lý Euler về điều kiện tồn tại chu trình Euler**. Định lý này khẳng định rằng nếu một đồ thị liên thông và tất cả các đỉnh đều có bậc chẵn (hoặc bán bậc vào bằng bán bậc ra đối với đồ thị có hướng), thì chu trình Euler chắc chắn tồn tại. Nhận thức này dẫn đến ý tưởng rằng:

- Mỗi khi ta đi vào một đỉnh bằng một cạnh, do bậc của đỉnh là chẵn, luôn có một cạnh khác để đi ra khỏi đỉnh đó (trừ khi tất cả các cạnh của đỉnh đã được thăm). Điều này đảm bảo rằng quá trình duyệt theo các cạnh chưa thăm sẽ không bị kẹt vĩnh viễn và luôn có thể tạo thành một chu trình đóng.
- Nếu sau khi hình thành một chu trình, vẫn còn các cạnh chưa thăm trong đồ thị, thì do tính liên thông và tính chất bậc chẵn, chắc chắn phải có một đỉnh trên chu trình vừa tìm được mà vẫn còn các cạnh kề chưa được thăm. Điều này cho phép thuật toán "quay lui" để tìm đến đỉnh đó, bắt đầu một "chu trình con" mới từ đó, và sau đó "nối" chu trình con này vào chu trình lớn hơn đã hình thành.

Có thể tóm tắt **ý tưởng cốt lõi** của thuật toán Hierholzer là: "Đi theo các cạnh cho đến khi không thể đi nữa, ghi lại đường đi đó, sau đó quay lại các điểm dừng để tìm thêm những con đường chưa được đi qua và nối chúng vào lộ trình ban đầu." Thuật toán tận dụng đặc tính "bậc chẵn" của các đỉnh trong đồ thị Euler để đảm bảo rằng mỗi khi đi vào một đỉnh, luôn có một đường để đi ra, cho phép tạo thành các chu trình con và nối chúng lại thành một chu trình lớn bao phủ toàn bộ đồ thị.

3.2.2 Cấu trúc và Nguyên lý hoạt động của Thuật toán Hierholzer

Thuật toán Hierholzer không biểu diễn dưới dạng công thức toán học mà là một quy trình lặp dựa trên cấu trúc dữ liệu ngăn xếp. Luồng hoạt động chính của thuật toán có thể được mô tả như sau:

Các bước khái quát:

1. **Khởi tạo:**
   - Chọn một đỉnh bất kỳ làm điểm bắt đầu và đẩy nó vào một ngăn xếp (stack).
   - Tạo một danh sách rỗng để lưu trữ chu trình Euler cuối cùng (Euler Circuit List).
2. **Duyệt và Xây dựng Chu trình:**
   - Trong khi ngăn xếp không rỗng:
     - Lấy đỉnh hiện tại từ đầu ngăn xếp.
     - **Nếu đỉnh hiện tại vẫn còn cạnh kề chưa được thăm:**
       * Chọn một cạnh chưa thăm từ đỉnh hiện tại và di chuyển đến đỉnh kề tiếp theo.
       * Đẩy đỉnh kề này vào ngăn xếp.
       * Loại bỏ cạnh vừa đi qua khỏi đồ thị (hoặc đánh dấu đã thăm).

- **Nếu đỉnh hiện tại không còn cạnh kề chưa được thăm:**
  - Loại bỏ đỉnh hiện tại khỏi ngăn xếp.
  - Thêm đỉnh này vào **đầu** danh sách chu trình Euler.

3. **Hoàn thành:**

   - Sau khi ngăn xếp rỗng, danh sách chu trình Euler sẽ chứa các đỉnh theo thứ tự ngược. Đảo ngược danh sách này để có được chu trình Euler hoàn chỉnh.

Trong đó:

- **`stack` (Ngăn xếp):** Đóng vai trò là "lịch sử đường đi tạm thời". Nó lưu giữ các đỉnh mà chúng ta đã đi qua và có thể quay lại để tìm các nhánh đường khác nếu cần. Nó hỗ trợ cơ chế "quay lui" của thuật toán.
- **`euler_circuit` (Danh sách chu trình Euler):** Là nơi lưu trữ kết quả cuối cùng. Việc thêm đỉnh vào đầu danh sách khi "pop" khỏi ngăn xếp là một kỹ thuật thông minh để tự động "nối" các chu trình con vào chu trình chính khi thuật toán quay lui, mà không cần thao tác nối thủ công phức tạp.
- **Việc loại bỏ/đánh dấu cạnh đã thăm:** Đảm bảo mỗi cạnh chỉ được đi qua đúng một lần, tuân thủ định nghĩa của chu trình Euler.

Nguyên lý hoạt động có thể hiểu theo hai giai đoạn chính:

1. **Giai đoạn "khám phá sâu":**

   - Thuật toán liên tục di chuyển sâu vào đồ thị theo một con đường chưa được thăm. Nhờ tính chất bậc chẵn của các đỉnh, ta luôn có thể đi ra khỏi một đỉnh nếu ta đi vào (trừ khi không còn cạnh nào từ đỉnh đó), đảm bảo không bị mắc kẹt vĩnh viễn trên một con đường. Giai đoạn này đẩy các đỉnh vào ngăn xếp.

2. **Giai đoạn "hoàn thiện và nối mạch":**

   - Khi thuật toán đi đến một đỉnh mà không còn cạnh kề chưa thăm nào, điều đó có nghĩa là một chu trình con đã được hoàn thành. Đỉnh này được loại bỏ khỏi ngăn xếp và thêm vào đầu danh sách `euler_circuit`. Việc này lặp lại theo cơ chế quay lui của ngăn xếp. Nhờ đó, nếu có các "chu trình con" nằm bên trong hoặc rẽ nhánh từ chu trình chính, chúng sẽ được tìm thấy và "nối" một cách tự động vào chu trình lớn hơn thông qua thứ tự nghịch đảo của ngăn xếp, cuối cùng tạo thành một chu trình bao phủ toàn bộ các cạnh.

Như vậy, thuật toán Hierholzer không tính toán một "điểm" mà là xây dựng một "lộ trình" cụ thể, tận dụng cấu trúc của đồ thị để đảm bảo mọi cạnh đều được thăm chính xác một lần.

3.2.3 Triển khai thuật toán Hierholzer

Dưới đây là mã giả chi tiết của thuật toán Hierholzer:

Euler_Cycle(u)

Bước 1: Khởi tạo

stack $= \emptyset$; — Khởi tạo stack là $\emptyset$
CE $= \emptyset$; — Khởi tạo mảng CE là $\emptyset$
push(stack, u); — Đưa đỉnh u vào ngăn xếp

Bước 2: Lặp

while (stack $\neq \emptyset$) **do**
  s = get(stack); — Lấy đỉnh ở đầu ngăn xếp (không loại bỏ)
  if (Ke(s) $\neq \emptyset$)
    $t = \langle$đỉnh đầu tiên trong Ke(s)$\rangle$;
    push(stack, t); — Đưa đỉnh t vào ngăn xếp
    $E = E \setminus \{(s,t)\}$; — Loại bỏ cạnh (s,t) khỏi đồ thị
  else
    s = pop(stack); — Loại đỉnh s khỏi ngăn xếp
    $s \Rightarrow$ CE; — Thêm s vào đầu danh sách CE
  endif
endwhile

Bước 3: Trả lại kết quả

$\langle$Lật ngược lại các đỉnh trong CE ta được chu trình Euler hoàn chỉnh$\rangle$

3.2.4 Ví dụ minh họa (Đồ thị vô hướng):

Giả sử đồ thị $G = (V, E)$ với $V = \{1, 2, 3, 4\}$ và các cạnh $E = \{(1,2), (2,3), (3,4), (4,1), (1,3), (3,1), (2,4)$ (là một đa đồ thị có 4 đỉnh và 8 cạnh). Kiểm tra bậc: $deg(1) = 4, deg(2) = 4, deg(3) = 4, deg(4) = 4$. Tất cả đều chẵn, nên có chu trình Euler.

Các bước thực hiện:

1. **Khởi tạo:** `stack = [1]`, `euler_circuit = []`.
2. **Lặp:**
   - `current_node` = 1. Cạnh kề chưa thăm: `(1,2)`. Chọn 2. `stack = [1, 2]`. Xóa `(1,2)`.
   - `current_node` = 2. Cạnh kề chưa thăm: `(2,3)`. Chọn 3. `stack = [1, 2, 3]`. Xóa `(2,3)`.
   - `current_node` = 3. Cạnh kề chưa thăm: `(3,4)`. Chọn 4. `stack = [1, 2, 3, 4]`. Xóa `(3,4)`.
   - `current_node` = 4. Cạnh kề chưa thăm: `(4,1)`. Chọn 1. `stack = [1, 2, 3, 4, 1]`. Xóa `(4,1)`.
   - `current_node` = 1. Còn cạnh `(1,3)` chưa thăm. Chọn 3. `stack = [1, 2, 3, 4, 1, 3]`. Xóa `(1,3)`.
   - `current_node` = 3. Còn cạnh `(3,2)` chưa thăm. Chọn 2. `stack = [1, 2, 3, 4, 1, 3, 2]`. Xóa `(3,2)`.
   - `current_node` = 2. Còn cạnh `(2,4)` chưa thăm. Chọn 4. `stack = [1, 2, 3, 4, 1, 3, 2, 4]`. Xóa `(2,4)`.

- `current_node = 4`. Không còn cạnh kề nào của 4 chưa thăm.
  - Pop 4. `euler_circuit = [4]`. `stack = [1, 2, 3, 4, 1, 3, 2]`.
- `current_node = 2`. Không còn cạnh kề nào của 2 chưa thăm.
  - Pop 2. `euler_circuit = [2, 4]`. `stack = [1, 2, 3, 4, 1, 3]`.
- `current_node = 3`. Không còn cạnh kề nào của 3 chưa thăm.
  - Pop 3. `euler_circuit = [3, 2, 4]`. `stack = [1, 2, 3, 4, 1]`.
- `current_node = 1`. Không còn cạnh kề nào của 1 chưa thăm.
  - Pop 1. `euler_circuit = [1, 3, 2, 4, 1]`. `stack = [1, 2, 3, 4]`. (Lưu ý: Để đơn giản hóa ví dụ, phần này chỉ minh họa cách các chu trình con được nối khi quay lui. Trên thực tế, có thể có thêm các bước pop đỉnh khác trước khi quay về đỉnh đầu tiên.)

3. **Đảo ngược:** Sau khi hoàn tất, `euler_circuit` sẽ được đảo ngược để có chu trình đúng. Ví dụ: `[1, 4, 2, 3, 1, 4, 3, 2, 1]`.

3.2.5 Độ phức tạp:

$O(|V| + |E|)$. Thuật toán này rất hiệu quả vì mỗi cạnh và mỗi đỉnh được thăm một số lần hữu hạn (thường là hằng số lần).

3.3 Thuật toán tìm đường đi Hamilton: Thuật toán Backtracking

3.3.1 Tổng quan về Đường đi Hamilton và Thuật toán Backtracking

Đường đi Hamilton là một dãy các đỉnh trong đồ thị đi qua **mỗi đỉnh đúng một lần**. Nếu đường đi đó bắt đầu và kết thúc tại cùng một đỉnh, ta gọi là **chu trình Hamilton**.

Không giống như chu trình Euler (với điều kiện xác định rõ ràng), bài toán Hamilton là một trong những bài toán thuộc lớp **NP-đầy đủ** – nghĩa là không có thuật toán giải tổng quát nào hiệu quả đã biết. Do đó, với các đồ thị nhỏ, thuật toán **quay lui (backtracking)** là một phương pháp khả thi, giúp thử tất cả các cách đi qua đỉnh để tìm ra một lời giải.

3.3.2 Cơ sở lý thuyết và Ý tưởng thuật toán

Ý tưởng xây dựng thuật toán Backtracking dựa trên việc xây dựng đường đi dần dần:

- Bắt đầu từ một đỉnh, lần lượt chọn một đỉnh kề chưa thăm.
- Mỗi khi thêm đỉnh mới vào đường đi, đánh dấu đỉnh đó là đã thăm.
- Nếu không thể tiếp tục (không còn đỉnh nào hợp lệ để đi tiếp), ta quay lui (backtrack), loại bỏ đỉnh đã thêm, và thử nhánh khác.

Điều kiện dừng:

- Nếu đường đi chứa đủ $|V|$ đỉnh:

- Nếu yêu cầu là chu trình Hamilton: kiểm tra xem đỉnh cuối có nối lại đỉnh đầu không.
- Nếu chỉ cần đường đi Hamilton: trả về kết quả ngay.

3.3.3 Cấu trúc và Nguyên lý hoạt động của thuật toán Backtracking

Thuật toán Backtracking thường được cài đặt bằng **đệ quy**, với hai cấu trúc dữ liệu chính:

- `path`: danh sách lưu đường đi hiện tại.
- `visited`: mảng/biến theo dõi trạng thái đã thăm của các đỉnh.

Các bước khái quát:

1. **Khởi tạo:**
   - Chọn một đỉnh bắt đầu, thêm vào `path`, đánh dấu là đã thăm.
2. **Duyệt và mở rộng đường đi:**
   - Tại mỗi bước, thử tất cả các đỉnh kề của đỉnh hiện tại:
     - Nếu đỉnh đó chưa được thăm, thêm vào `path`, đánh dấu đã thăm, gọi đệ quy tiếp theo.
     - Nếu không thành công, quay lui: xóa đỉnh khỏi `path`, bỏ đánh dấu.
3. **Kết thúc:**
   - Nếu `path` chứa $|V|$ đỉnh:
     - Nếu yêu cầu chu trình: kiểm tra xem đỉnh cuối nối được về đỉnh đầu không.
     - Nếu yêu cầu đường đi: trả về kết quả ngay.

Vai trò các thành phần:

- `visited`: đảm bảo mỗi đỉnh chỉ được đi qua đúng một lần.
- `path`: lưu trữ thứ tự duyệt.
- Đệ quy + quay lui giúp kiểm tra mọi khả năng hợp lệ mà không lặp lại trạng thái sai.

3.3.4 Mã giả của thuật toán Backtracking tìm chu trình Hamilton

Hamiltonian_Cycle(current_node)

Bước 1: Điều kiện dừng

if len(path) == $|V|$ **then**
  if current_node kề với path[0] **then**
    return True (Tìm thấy chu trình Hamilton)
  else return False

Bước 2: Duyệt các đỉnh kề

for next_node in Kề(current_node):
  if not visited[next_node]:
    visited[next_node] = True
    path.append(next_node)
    if Hamiltonian_Cycle(next_node):
      return True
    Quay lui:
    visited[next_node] = False
    path.pop()

return False

3.3.5 Ví dụ minh họa (Đồ thị vô hướng)

Xét đồ thị $G = (V, E)$ với $V = \{A, B, C, D\}$ và $E = \{(A, B), (A, C), (A, D), (B, C), (C, D)\}$.

Bước thực hiện (tìm chu trình Hamilton):

1. Bắt đầu tại `A`, `path = [A]`, `visited = {A:True, B:False, C:False, D:False}`.
2. Thử `B`, `path = [A, B]`, đánh dấu `B`.
3. Thử `C`, `path = [A, B, C]`, đánh dấu `C`.
4. Thử `D`, `path = [A, B, C, D]`, đánh dấu `D`.
5. Đã đi qua 4 đỉnh. Kiểm tra xem `D` kề với `A`. Vì có cạnh `(D,A)`, tìm được chu trình `A-B-C-D-A`.

3.3.6 Độ phức tạp

- **Thời gian**: $O(V!)$ trong trường hợp xấu nhất – thử tất cả các hoán vị đỉnh.
- **Không gian**: $O(V)$ cho mảng đánh dấu và đường đi hiện tại.
- Thích hợp với đồ thị nhỏ, thường $V \leq 15$.

4 Áp dụng Chu trình Euler và Hamilton trong giải quyết bài toán tìm đường đi, bài toán phân phát hàng hóa.

4.1 Mô hình hóa bài toán thực tế (giao hàng, phân phát hàng hóa)

Lý thuyết đồ thị là một công cụ toán học mạnh mẽ để biểu diễn và giải quyết các vấn đề tối ưu hóa trong thế giới thực, đặc biệt trong lĩnh vực logistics, vận tải và lập kế hoạch.

- **Bài toán giao hàng/Phủ sóng mạng lưới (ví dụ: bưu tá giao thư, xe quét rác, kiểm tra đường ống):**
  - **Mô hình hóa:** Trong các bài toán này, các con đường, tuyến đường, hoặc các liên kết vật lý được biểu diễn bằng các **cạnh** của đồ thị. Các giao lộ, ngã ba, hoặc điểm kết nối của các tuyến đường là các **đỉnh**. Đồ thị có thể là vô hướng (đường hai chiều) hoặc có hướng (đường một chiều).
  - **Mục tiêu:** Mục tiêu chính là đi qua *tất cả các cạnh* của mạng lưới ít nhất một lần (hoặc lý tưởng là đúng một lần) một cách hiệu quả nhất, sau đó quay trở lại điểm xuất phát. Ví dụ: một người đưa thư cần đi qua mọi con phố để giao thư, hoặc một xe quét rác cần đi qua mọi con đường để dọn dẹp.
  - **Liên hệ với lý thuyết đồ thị:** Bài toán này mô hình hóa việc tìm kiếm **chu trình Euler** hoặc **đường đi Euler**.
    - Nếu bưu tá cần bắt đầu và kết thúc tại cùng một bưu cục, và đi qua mỗi con đường đúng một lần, đó là một chu trình Euler.
    - Nếu có thể kết thúc ở một điểm khác điểm xuất phát (ví dụ: giao một bưu kiện lớn ở cuối tuyến), đó có thể là một đường đi Euler.
  - **Ví dụ cụ thể:**
    - **Bưu điện Hạnh Phúc** cần tối ưu hóa lộ trình cho người đưa thư tại khu vực A. Khu vực A có các con phố được nối với nhau qua các ngã tư. Mỗi con phố cần được đi qua để giao thư. Bưu điện muốn tìm một lộ trình mà người đưa thư đi qua *mọi con phố chỉ một lần* và quay về bưu điện, để tiết kiệm thời gian và nhiên liệu. Đây chính là một bài toán tìm chu trình Euler trên đồ thị biểu diễn các con phố và ngã tư.
- **Bài toán phân phát hàng hóa/Lập kế hoạch tuyến đường (ví dụ: xe tải giao hàng đến nhiều cửa hàng, nhân viên bán hàng thăm các khách hàng, tour du lịch):**
  - **Mô hình hóa:** Các địa điểm cần ghé thăm (như kho hàng, cửa hàng, điểm đến, thành phố) được biểu diễn bằng các **đỉnh** của đồ thị. Các con đường nối giữa các địa điểm này là các **cạnh**, và các cạnh này thường có **trọng số** (ví dụ: khoảng cách, thời gian di chuyển, chi phí nhiên liệu). Đồ thị có thể là vô hướng (đi hai chiều trên đường) hoặc có hướng (đường một chiều).
  - **Mục tiêu:** Xe tải hoặc nhân viên cần ghé thăm *tất cả các địa điểm* (đỉnh) một lần duy nhất và quay trở về điểm xuất phát, với tổng trọng số của các cạnh trên lộ trình là *nhỏ nhất*.

- **Liên hệ với lý thuyết đồ thị:** Bài toán này liên quan trực tiếp đến việc tìm **chu trình Hamilton** hoặc, phổ biến hơn và phức tạp hơn, **Bài toán người du lịch (Traveling Salesperson Problem - TSP)**.
  - Tìm chu trình Hamilton chỉ đảm bảo mọi địa điểm được thăm đúng một lần.
  - TSP mở rộng bằng cách tìm chu trình Hamilton tối ưu nhất về chi phí hoặc quãng đường.
- **Ví dụ cụ thể:**
  - **Công ty Giao Hàng Nhanh** có một xe tải cần giao hàng từ kho trung tâm (A) đến các cửa hàng B, C, D, E. Sau khi giao hàng xong ở tất cả các cửa hàng, xe phải quay về kho A. Công ty muốn tìm lộ trình ngắn nhất để tiết kiệm chi phí nhiên liệu. Nếu khoảng cách giữa các cửa hàng được biết, đây là một bài toán TSP. Mỗi cửa hàng là một đỉnh, khoảng cách là trọng số của cạnh.

4.2 Liên hệ với chu trình Euler, Hamilton và TSP

Việc hiểu rõ sự khác biệt và mối liên hệ giữa các khái niệm này là rất quan trọng để áp dụng chúng vào giải quyết bài toán thực tế:

- **Chu trình Euler:**
  - **Định nghĩa:** Một chu trình trong đồ thị đi qua **mỗi cạnh đúng một lần** và kết thúc tại đỉnh bắt đầu.
  - **Ứng dụng thực tế:** Tối ưu hóa việc "bao phủ" hoặc "quét" một khu vực.
    - *Ví dụ:* Lập kế hoạch lộ trình cho xe quét đường, xe bôi sơn vạch kẻ đường, kiểm tra đường dây điện thoại/cáp quang, hoặc các hoạt động bảo trì yêu cầu đi qua mọi liên kết trong mạng lưới.
  - **Đặc điểm:** Tập trung vào việc thăm các *cạnh*.
- **Chu trình Hamilton:**
  - **Định nghĩa:** Một chu trình trong đồ thị đi qua **mỗi đỉnh đúng một lần** và kết thúc tại đỉnh bắt đầu.
  - **Ứng dụng thực tế:** Lập kế hoạch lộ trình cho việc thăm các địa điểm riêng lẻ.
    - *Ví dụ:* Lập lịch trình xe buýt đi qua tất cả các trạm, thiết kế mạch tích hợp (đảm bảo tín hiệu đi qua mọi cổng logic một lần), lập kế hoạch tham quan các địa điểm du lịch.
  - **Đặc điểm:** Tập trung vào việc thăm các *đỉnh*.
- **Bài toán người du lịch (TSP - Traveling Salesperson Problem):**
  - **Định nghĩa:** Là một phiên bản mở rộng của bài toán chu trình Hamilton trên đồ thị có trọng số. Mục tiêu là tìm chu trình Hamilton có tổng trọng số các cạnh là *nhỏ nhất*.

- **Mối quan hệ với Hamilton:** Mọi lời giải của TSP đều là một chu trình Hamilton, nhưng không phải mọi chu trình Hamilton đều là lời giải của TSP. TSP tìm kiếm chu trình Hamilton *tối ưu*.
- **Tính chất và độ phức tạp:** TSP là một bài toán NP-hard. Điều này có nghĩa là, với số lượng đỉnh lớn, không có thuật toán hiệu quả nào (thời gian đa thức) được biết đến để tìm ra lời giải chính xác trong mọi trường hợp. Do đó, trong thực tế, thường phải sử dụng các thuật toán xấp xỉ (heuristic) để tìm lời giải gần tối ưu.
- **Ứng dụng thực tế:** Lập kế hoạch tuyến đường cho đội xe giao hàng, lập trình máy khoan PCB (Printed Circuit Board) để khoan tất cả các lỗ với quãng đường ngắn nhất, lập kế hoạch chuyến đi cho các hãng hàng không.

4.3 Các Thuật Toán Giải Bài Toán Người Du Lịch (TSP)

Bài toán Người Du lịch (Traveling Salesperson Problem - TSP) là một trong những bài toán tối ưu hóa nổi tiếng và khó trong khoa học máy tính, có nhiều ứng dụng thực tế trong logistics, quy hoạch tuyến đường, và sản xuất. Mục tiêu của TSP là tìm ra chu trình ngắn nhất đi qua mỗi đỉnh (thành phố) đúng một lần và quay trở lại đỉnh xuất phát. Dưới đây là các phương pháp tiếp cận để giải quyết bài toán TSP, từ đơn giản đến phức tạp hơn:

1.Thuật toán Brute-force ("Vét cạn") Thuật toán vét cạn là phương pháp đơn giản nhất để giải quyết TSP, bằng cách kiểm tra mọi khả năng có thể. Nó đảm bảo tìm ra giải pháp tối ưu. Thuật toán vét cạn là phương pháp đơn giản nhất để giải quyết TSP, bằng cách kiểm tra mọi khả năng có thể. Nó đảm bảo tìm ra giải pháp tối ưu.

Cơ sở lý thuyết: Thuật toán Brute-force dựa trên nguyên lý của sự tìm kiếm đầy đủ (exhaustive search). Đối với Bài toán Người Du lịch, mục tiêu là tìm một chu trình Hamilton có tổng trọng số (chi phí) nhỏ nhất. Cơ sở lý thuyết của phương pháp vét cạn là: nếu một giải pháp tối ưu tồn tại, thì nó phải nằm trong tập hợp tất cả các giải pháp khả thi. Bằng cách liệt kê và đánh giá *tất cả* các chu trình Hamilton có thể có trên đồ thị và so sánh chi phí của chúng, thuật toán này đảm bảo sẽ tìm thấy chu trình có chi phí nhỏ nhất, tức là giải pháp tối ưu toàn cục.

- **Ý tưởng chính:** Liệt kê tất cả các chu trình Hamilton có thể có trên đồ thị (có trọng số), tính tổng chi phí (trọng số) của mỗi chu trình và chọn chu trình có chi phí nhỏ nhất.
- **Các bước chi tiết:**
  1. **Chọn đỉnh xuất phát:** Chọn một đỉnh bất kỳ làm điểm xuất phát (ví dụ: đỉnh 0). Chu trình sẽ bắt đầu và kết thúc tại đỉnh này.
  2. **Liệt kê hoán vị:** Tạo tất cả các hoán vị có thể của các đỉnh còn lại (tức là $n - 1$ đỉnh).
  3. **Xây dựng và tính toán chu trình:** Đối với mỗi hoán vị của các đỉnh còn lại:

- Xây dựng một chu trình bằng cách nối đỉnh xuất phát với đỉnh đầu tiên trong hoán vị, sau đó nối các đỉnh theo thứ tự trong hoán vị, và cuối cùng nối đỉnh cuối cùng trong hoán vị với đỉnh xuất phát.
- Tính tổng trọng số của tất cả các cạnh trong chu trình này. Nếu có cạnh không tồn tại giữa hai đỉnh liên tiếp trong hoán vị, chu trình đó không hợp lệ và bị loại bỏ (hoặc gán chi phí vô hạn).

4. **So sánh và tìm tối ưu:** So sánh tổng trọng số của tất cả các chu trình hợp lệ đã tính được và chọn chu trình có tổng trọng số nhỏ nhất.

- **Ví dụ minh họa:** Giả sử có 4 thành phố A, B, C, D với ma trận khoảng cách (trọng số) sau:

```
   A  B  C  D
A  0 10 15 20
B 10  0 35 25
C 15 35  0 30
D 20 25 30  0
```

Chọn A là đỉnh xuất phát. Các đỉnh còn lại là B, C, D. Số hoán vị của (B, C, D) là $3! = 6$. Các hoán vị và chi phí tương ứng:

1. **(B, C, D):** Chu trình A -> B -> C -> D -> A Chi phí: $dist(A, B) + dist(B, C) + dist(C, D) + dist(D, A) = 10 + 35 + 30 + 20 = 95$
2. **(B, D, C):** Chu trình A -> B -> D -> C -> A Chi phí: $dist(A, B) + dist(B, D) + dist(D, C) + dist(C, A) = 10 + 25 + 30 + 15 = 80$ (Hiện tại là nhỏ nhất)
3. **(C, B, D):** Chu trình A -> C -> B -> D -> A Chi phí: $dist(A, C)+dist(C, B)+dist(B, D) + dist(D, A) = 15 + 35 + 25 + 20 = 95$
4. **(C, D, B):** Chu trình A -> C -> D -> B -> A Chi phí: $dist(A, C) + dist(C, D) + dist(D, B) + dist(B, A) = 15 + 30 + 25 + 10 = 80$ (Cũng 80)
5. **(D, B, C):** Chu trình A -> D -> B -> C -> A Chi phí: $dist(A, D) + dist(D, B) + dist(B, C) + dist(C, A) = 20 + 25 + 35 + 15 = 95$
6. **(D, C, B):** Chu trình A -> D -> C -> B -> A Chi phí: $dist(A, D) + dist(D, C) + dist(C, B) + dist(B, A) = 20 + 30 + 35 + 10 = 95$

Chu trình ngắn nhất là: A -> B -> D -> C -> A hoặc A -> C -> D -> B -> A, với chi phí 80.

- **Độ phức tạp:** Số lượng hoán vị của $n - 1$ đỉnh là $(n - 1)!$. Đối với mỗi hoán vị, cần thực hiện n phép cộng. Do đó, độ phức tạp là $O(n \cdot (n - 1)!)$ hay $O(n!)$. Điều này khiến thuật toán vét cạn trở nên không khả thi nhanh chóng khi số lượng đỉnh tăng lên (ví dụ: với 20 đỉnh, 20! là một con số khổng lồ).

Pseudocode (Giải mã):

```
function BruteForceTSP(khoangCach):
 n = so luong dinh trong do thi
 dinh_bat_dau = 0 // Chon dinh 0 lam diem bat dau mac dinh

 cac_dinh_con_lai = mot danh sach chua tat ca cac dinh tru
    dinh_bat_dau

 min_chi_phi_toan_cuc = vo_cuc
 chu_trinh_toi_uu = null

 // Duyet qua tat ca cac hoan vi cua (n-1) dinh con lai
 for moi_hoan_vi trong tat_ca_cac_hoan_vi(cac_dinh_con_lai):
  // Xay dung chu trinh day du: bat dau -> hoan_vi -> ket thuc ve
      bat_dau
  current_path = [dinh_bat_dau] + moi_hoan_vi + [dinh_bat_dau]
  current_cost = 0
  chu_trinh_hop_le = true

  // Tinh chi phi cho chu trinh hien tai
  for i tu 0 den n-1: // Duyet tu dinh thu i den i+1 trong
     current_path
   u = current_path[i]
   v = current_path[i+1]
   if khoangCach[u][v] == vo_cuc: // Kiem tra canh co ton tai
      khong
    chu_trinh_hop_le = false
    break // Chu trinh khong hop le, bo qua
   current_cost = current_cost + khoangCach[u][v]

  // So sanh va cap nhat neu tim thay chu trinh ngan hon va hop
     le
  if chu_trinh_hop_le and current_cost < min_chi_phi_toan_cuc:
   min_chi_phi_toan_cuc = current_cost
   chu_trinh_toi_uu = current_path

 return chu_trinh_toi_uu, min_chi_phi_toan_cuc
```

Listing 1: Pseudocode Thuật toán Brute-force (Vét cạn) cho TSP

2. Thuật toán Quy hoạch Động (Dynamic Programming) - Thuật toán Held-Karp : Thuật toán Held-Karp là một phương pháp Quy hoạch Động kinh điển để giải bài toán TSP một cách chính xác. Mặc dù có độ phức tạp thời gian mũ, nó hiệu quả hơn phương pháp vét cạn (brute-force) đối với các đồ thị có số đỉnh vừa phải.

Cơ sở lý thuyết: Thuật toán Held-Karp dựa trên nguyên lý Quy hoạch Động (Dynamic Programming), cụ thể là các tính chất của **cấu trúc con tối ưu (optimal substructure)** và **các bài toán con gối nhau (overlapping subproblems)**. Nó nhận thấy rằng đường đi tối ưu đến một đỉnh i qua một tập con các đỉnh S phải bao gồm một đường đi tối ưu đến một đỉnh $j \in S$ (khác i) qua tập con $S \setminus \{i\}$ trước khi đi đến i. Bằng cách lưu trữ và

tái sử dụng kết quả của các bài toán con nhỏ hơn, thuật toán tránh việc tính toán lặp lại, từ đó cải thiện hiệu suất so với vét cạn.

Ý tưởng chính: Thuật toán Held-Karp sử dụng một hàm trạng thái $C(S, i)$ biểu thị chi phí đường đi ngắn nhất bắt đầu từ đỉnh xuất phát (giả sử đỉnh 0), đi qua tất cả các đỉnh trong tập S (trừ đỉnh 0), và kết thúc tại đỉnh $i \in S$.

Các bước chi tiết: Thuật toán Held-Karp tiến hành theo các bước sau, xây dựng bảng Quy hoạch Động (DP table) từ các bài toán con nhỏ nhất đến bài toán lớn hơn:

1. **Khởi tạo bảng DP:** * Tạo một bảng 'C[mask][i]' để lưu trữ chi phí đường đi ngắn nhất. 'mask' là một bitmask biểu diễn tập hợp các đỉnh đã được thăm (bao gồm đỉnh xuất phát), và 'i' là đỉnh cuối cùng trong đường đi. * Ban đầu, tất cả các giá trị trong bảng được gán là vô cực.

2. **Bước cơ sở (Base Case):** * Đối với mỗi đỉnh i khác đỉnh xuất phát (giả sử đỉnh 0), chi phí để đi từ đỉnh 0 đến i mà chỉ đi qua hai đỉnh này là khoảng cách trực tiếp $d_{0,i}$. * Công thức: $C(\{0, i\}, i) = d_{0,i}$ (Tập S ở đây chỉ gồm đỉnh 0 và i).

3. **Tính toán đệ quy (Bước lặp):** * Lặp từ các tập con có kích thước nhỏ nhất đến các tập con lớn hơn. * Đối với mỗi tập con S của các đỉnh (luôn bao gồm đỉnh 0) và mỗi đỉnh $i \in S$ (khác đỉnh 0): * Tính $C(S, i)$ bằng cách xem xét tất cả các đỉnh $j \in S$ (khác i). Đường đi đến i qua S phải đi qua một đỉnh j trước đó. * Công thức đệ quy:
$$C(S, i) = \min_{j \in S, j \neq i} \{C(S \setminus \{i\}, j) + d_{j,i}\}$$
Trong đó:
   - S là một tập con các đỉnh đã được thăm.
   - i là đỉnh cuối cùng trong đường đi.
   - $d_{j,i}$ là trọng số cạnh từ đỉnh j đến đỉnh i.

4. **Kết quả cuối cùng:** * Sau khi tính toán hết tất cả các trạng thái $C(S, i)$, chi phí của chu trình TSP ngắn nhất sẽ là chi phí nhỏ nhất khi đi từ đỉnh xuất phát, qua tất cả các đỉnh còn lại, kết thúc tại một đỉnh i bất kỳ, và sau đó quay về đỉnh xuất phát. * Công thức: $\min_{i \neq 0}\{C(V \setminus \{0\}, i) + d_{i,0}\}$, với V là tập hợp tất cả các đỉnh.

Ví dụ minh họa: Giả sử có 3 thành phố A, B, C (đỉnh 0, 1, 2) với A (đỉnh 0) là đỉnh xuất phát. Ma trận khoảng cách $d_{u,v}$:

	0	1	2
0	0	10	15
1	12	0	20
2	18	25	0

Chúng ta cần tính $C(S, i)$ với S là tập con các đỉnh đã thăm (luôn bao gồm đỉnh 0), và i là đỉnh cuối cùng của đường đi.

1. Bước cơ sở (Đường đi gồm 2 đỉnh: 0 và 1 đỉnh khác): * $C(\{0, 1\}, 1) = d_{0,1} = 10$ * $C(\{0, 2\}, 2) = d_{0,2} = 15$

2. Bước lặp (Đường đi gồm 3 đỉnh: 0 và 2 đỉnh khác): * **Tập con $S = \{0, 1, 2\}$** (tất cả các đỉnh) * **Tính $C(\{0, 1, 2\}, 1)$:** Đỉnh kết thúc là 1. Đỉnh trước 1 phải là 2. * $C(\{0, 1, 2\}, 1) = C(\{0, 2\}, 2) + d_{2,1} = 15 + 25 = 40$ * **Tính $C(\{0, 1, 2\}, 2)$:**

Đỉnh kết thúc là 2. Đỉnh trước 2 phải là 1. * $C(\{0,1,2\},2) = C(\{0,1\},1) + d_{1,2} = 10+20 = 30$

3. Kết quả cuối cùng: Chi phí chu trình ngắn nhất là $\min_{i\in\{1,2\}}\{C(V,i) + d_{i,0}\}$ * $C(\{0,1,2\},1) + d_{1,0} = 40 + 12 = 52$ * $C(\{0,1,2\},2) + d_{2,0} = 30 + 18 = 48$

Chi phí tối ưu là $\min(52, 48) = 48$. Một chu trình tối ưu tương ứng có thể là 0 -> 1 -> 2 -> 0.

Độ phức tạp: $O(n^2 \cdot 2^n)$, với n là số đỉnh. Đây là độ phức tạp cao, làm cho thuật toán chỉ khả thi với n nhỏ (thường $n \leq 20$).

Pseudocode (Giải mã):

```
function HeldKarpTSP(khoangCach):
    n = so luong dinh (bao gom dinh bat dau 0)

    // Khoi tao bang luu tru ket qua (DP table)
    // dp[mask][i] = chi phi toi thieu
    // mask: bitmask dai dien cho tap con cac dinh da duoc tham (bit thu j = 1 neu dinh j da tham)
    // i: dinh cuoi cung trong duong di
    // Kich thuoc bang dp: 2^n x n
    dp = mot bang 2^n x n chua vo_cuc

    // Buoc co so: duong di tu dinh 0 den moi dinh i (tap con chi co dinh 0 va i)
    // mask = (1 << 0) | (1 << i)
    for i tu 1 den n-1:
        dp[(1 << 0) | (1 << i)][i] = khoangCach[0][i]

    // Lap qua kich thuoc tap con (tu 2 den n-1 dinh khac 0)
    // k la so luong cac dinh ngoai dinh 0 trong tap con hien tai
    for k tu 1 den n-1: // k = so luong dinh trung gian
        // Lap qua tat ca cac tap con S cua cac dinh (su dung bitmask)
        // ma chua dinh 0 (bit 0 luon duoc set)
        // va co kich thuoc la k+1 (bao gom dinh 0)
        for mask tu 0 den (1 << n) - 1:
            if (mask & (1 << 0)) == 0: continue // Tap con phai chua dinh 0
            if __builtin_popcount(mask) == k + 1: // Neu kich thuoc tap con la k+1
                // Lap qua cac dinh ket thuc i trong tap_con S (i khac 0)
                for i tu 1 den n-1:
                    if (mask & (1 << i)) != 0: // Neu dinh i co trong tap con hien tai (mask)
                        // min_chi_phi se la C(S, i)
                        min_chi_phi_cho_S_i = vo_cuc
                        // Lap qua cac dinh j trong tap_con S_tru_i
                        // (S ma khong co i)
                        for j tu 0 den n-1:
                            if j == i: continue
```

```
                            if (mask & (1 << j)) != 0: // Neu dinh j co trong tap con hien tai (mask)
                                // Chi phi tu j den i
                                chi_phi_tu_j = dp[mask ^ (1 << i)][j] + khoangCach[j][i] // mask ^ (1 << i) la S \ {i}
                                if chi_phi_tu_j < min_chi_phi_cho_S_i:
                                    min_chi_phi_cho_S_i = chi_phi_tu_j
                        dp[mask][i] = min_chi_phi_cho_S_i

    // Buoc cuoi cung: tim chu trinh ngan nhat quay ve dinh 0
    final_min_chi_phi = vo_cuc
    // mask_tat_ca_cac_dinh_tru_0 = (1 << n) - 1
    full_mask = (1 << n) - 1 // Bitmask bieu dien tat ca cac dinh

    for i tu 1 den n-1:
        chi_phi_quay_ve_0 = dp[full_mask][i] + khoangCach[i][0]
        if chi_phi_quay_ve_0 < final_min_chi_phi:
            final_min_chi_phi = chi_phi_quay_ve_0

    return final_min_chi_phi
```

Listing 2: Pseudocode Thuật toán Held-Karp (Quy hoạch Động)

3. Thuật toán láng giềng gần nhất (Nearest Neighbor Algorithm) Vì TSP là bài toán NP-hard, trong nhiều ứng dụng thực tế, người ta sử dụng các thuật toán heuristic (thuật toán xấp xỉ) để tìm kiếm các giải pháp *gần tối ưu* trong một khoảng thời gian chấp nhận được, thay vì tìm kiếm giải pháp tối ưu tuyệt đối. Thuật toán láng giềng gần nhất là một trong những heuristic đơn giản và phổ biến nhất.

Cơ sở lý thuyết: Thuật toán láng giềng gần nhất là một thuật toán tham lam (greedy algorithm). Cơ sở lý thuyết của nó dựa trên việc đưa ra lựa chọn tối ưu cục bộ tại mỗi bước với hy vọng rằng chuỗi các lựa chọn cục bộ này sẽ dẫn đến một giải pháp tốt (mặc dù không nhất thiết là tối ưu toàn cục). Đối với bài toán TSP, lựa chọn cục bộ tối ưu ở đây là luôn di chuyển đến đỉnh chưa được thăm gần nhất từ vị trí hiện tại. Mặc dù không có đảm bảo về tính tối ưu, sự đơn giản và tốc độ của thuật toán này làm cho nó trở thành một lựa chọn phù hợp khi cần một giải pháp nhanh chóng cho các bài toán lớn.

Ý tưởng chính: Tại mỗi bước, thuật toán sẽ di chuyển đến đỉnh chưa được thăm gần nhất từ đỉnh hiện tại.

Các bước chi tiết: Thuật toán láng giềng gần nhất hoạt động theo các bước tuần tự sau:

1. **Khởi tạo:** Chọn một đỉnh bất kỳ làm điểm bắt đầu (thường là đỉnh 0). Thêm đỉnh này vào danh sách lộ trình và đánh dấu nó là đã thăm.

2. **Lặp để xây dựng lộ trình:** * Từ đỉnh hiện tại, tìm tất cả các đỉnh chưa được thăm. * Chọn đỉnh chưa được thăm có khoảng cách ngắn nhất đến đỉnh hiện tại. * Thêm đỉnh được chọn vào lộ trình và đánh dấu nó là đã thăm. * Cập nhật đỉnh hiện tại là đỉnh vừa được thăm. * Lặp lại bước này cho đến khi tất cả các đỉnh đã được thăm.

3. **Hoàn thành chu trình:** Nối đỉnh cuối cùng trong lộ trình với đỉnh xuất phát ban đầu để hoàn thành một chu trình khép kín.

4. **Tính tổng chi phí:** Tính tổng trọng số của tất cả các cạnh trong chu trình đã xây dựng.

Ví dụ minh họa: Sử dụng ma trận khoảng cách tương tự ví dụ Brute-force:

```
   A  B  C  D
A  0 10 15 20
B 10  0 35 25
C 15 35  0 30
D 20 25 30  0
```

Bắt đầu từ A:

1. `path = [A]`, `visited = {A:T, B:F, C:F, D:F}`. Đỉnh hiện tại: A.

2. Từ A, đỉnh gần nhất chưa thăm là B (khoảng cách 10). `path = [A,B]`, `visited = {A:T, B:T, C:F, D:F}`. Đỉnh hiện tại: B.

3. Từ B, các đỉnh chưa thăm là C (khoảng cách 35) và D (khoảng cách 25). D gần hơn. `path = [A, B, D]`, `visited = {A:T, B:T, C:F, D:T}`. Đỉnh hiện tại: D.

4. Từ D, đỉnh gần nhất chưa thăm là C (khoảng cách 30). `path = [A, B, D, C]`, `visited = {A:T, B:T, C:T, D:T}`. Đỉnh hiện tại: C.

5. Tất cả đỉnh đã thăm. Hoàn thành chu trình bằng cách nối C với A. `path = [A, B, D, C, A]`.

Tổng chi phí của chu trình A -> B -> D -> C -> A: $= dist(A,B) + dist(B,D) + dist(D,C) + dist(C,A) = 10 + 25 + 30 + 15 = 80$. (Trong ví dụ này, thuật toán láng giềng gần nhất đã tìm được kết quả tối ưu. Tuy nhiên, điều này không phải lúc nào cũng đúng do bản chất tham lam của thuật toán.)

Độ phức tạp: $O(n^2)$. Tại mỗi bước, thuật toán duyệt qua $O(n)$ đỉnh để tìm đỉnh gần nhất chưa thăm. Có $n-1$ bước như vậy để thăm tất cả các đỉnh. Do đó, tổng độ phức tạp là $O(n \cdot n) = O(n^2)$. Đây là một thuật toán rất nhanh so với các thuật toán tối ưu.

Ưu điểm:

- Đơn giản, dễ hiểu và dễ triển khai.
- Tốc độ thực thi rất nhanh, phù hợp cho các bài toán lớn với số lượng đỉnh lớn khi các thuật toán tối ưu không khả thi.
- Luôn tìm được một chu trình hợp lệ.

Nhược điểm:

- **Không đảm bảo tối ưu:** Thuật toán này là một heuristic, nghĩa là nó không đảm bảo tìm được chu trình tối ưu toàn cục. Nó có thể bị mắc kẹt trong một "tối ưu cục bộ" (local optimum), bỏ lỡ các giải pháp tổng thể tốt hơn.
- **Phụ thuộc vào điểm bắt đầu:** Kết quả có thể khác nhau đáng kể tùy thuộc vào đỉnh bắt đầu được chọn. Để cải thiện, có thể chạy thuật toán này với mỗi đỉnh làm điểm bắt đầu và chọn kết quả tốt nhất (khi đó độ phức tạp tăng lên $O(n^3)$).
- **Chi phí bỏ lỡ cơ hội:** Việc chọn đỉnh gần nhất tại mỗi bước có thể dẫn đến việc bỏ lỡ một đường đi tổng thể ngắn hơn về sau, vì nó không xem xét các lựa chọn dài hơn ở hiện tại có thể dẫn đến lợi ích lớn hơn trong tương lai.

Pseudocode (mã giả):

```
function NearestNeighborTSP(khoangCach):
    n = so luong dinh (bao gom ca dinh bat dau)
    path = danh sach rong
    visited = mot mang boolean (ban dau tat ca la false) // Danh dau dinh da tham

    // Buoc 1: Khoi tao - Chon dinh bat dau
    current_node = chon_dinh_bat_dau_bat_ky() // Thuong la dinh 0
    path.them(current_node)
    visited[current_node] = true

    // Buoc 2: Lap de tim cac dinh con lai
    for count tu 1 den n-1: // Lap n-1 lan de tham n-1 dinh con lai
        next_node = null
        min_distance = vo_cuc

        // Tim dinh chua duoc tham gan nhat tu dinh hien tai
        for neighbor tu 0 den n-1:
            if not visited[neighbor]:
                if khoangCach[current_node][neighbor] < min_distance:
                    min_distance = khoangCach[current_node][neighbor]
                    next_node = neighbor

        // Them dinh gan nhat vao lo trinh va danh dau da tham
        path.them(next_node)
        visited[next_node] = true
        current_node = next_node

    // Buoc 3: Hoan thanh chu trinh - Noi ve dinh bat dau
    path.them(path[0]) // Quay ve dinh dau tien de hoan thanh chu trinh

    // Buoc 4: Tinh tong chi phi cua chu trinh
```

```
    total_cost = 0
    for i tu 0 den n-1: // Duyet qua n canh cua chu trinh
        total_cost = total_cost + khoangCach[path[i]][path[i+1]]

    return path, total_cost
```

Listing 3: Pseudocode Thuật toán láng giềng gần nhất (Nearest Neighbor)

4. Thuật toán 2-opt Thuật toán 2-opt là một thuật toán heuristic cải tiến, được sử dụng để tìm một lời giải tốt cho TSP (thường là tối ưu cục bộ) trong thời gian hợp lý, đặc biệt phù hợp với các đồ thị lớn mà các thuật toán tối ưu như Held-Karp không thể áp dụng được.

Cơ sở lý thuyết: Thuật toán 2-opt dựa trên nguyên lý tìm kiếm cục bộ (local search) hay còn gọi là cải tiến lặp (iterative improvement). Ý tưởng là bắt đầu với một chu trình hợp lệ bất kỳ (ví dụ, một chu trình được tạo bởi thuật toán Nearest Neighbor), sau đó lặp đi lặp lại việc tìm kiếm các hoán đổi "2-opt" có thể cải thiện chu trình. Mỗi hoán đổi 2-opt cố gắng phá vỡ hai cạnh của chu trình và nối lại chúng theo một cách khác để tạo ra một chu trình ngắn hơn. Quá trình này tiếp tục cho đến khi không tìm thấy bất kỳ hoán đổi nào có thể làm giảm tổng chiều dài của chu trình, tại điểm đó thuật toán hội tụ về một tối ưu cục bộ.

Ý tưởng chính: Thuật toán 2-opt hoạt động bằng cách lặp đi lặp lại cải thiện một chu trình TSP hiện có. Ở mỗi bước, nó cố gắng gỡ bỏ hai cạnh không kề nhau từ chu trình và nối lại các đỉnh còn lại theo một cách khác để tạo ra một chu trình mới ngắn hơn. Quá trình này được lặp lại cho đến khi không có sự hoán đổi 2-opt nào có thể làm giảm tổng chiều dài của chu trình.

Các bước chi tiết: Giả sử có một chu trình ban đầu $P = v_1 \to v_2 \to \cdots \to v_n \to v_1$.

1. **Khởi tạo:** Chọn một chu trình TSP ban đầu. Chu trình này có thể được tạo ngẫu nhiên hoặc bằng một heuristic khác (ví dụ: Nearest Neighbor).

2. **Lặp cải tiến:** Lặp lại các bước sau cho đến khi không thể tìm thấy bất kỳ hoán đổi 2-opt nào làm giảm chi phí chu trình:

   - Duyệt qua tất cả các cặp cạnh không kề nhau (v_i, v_{i+1}) và (v_j, v_{j+1}) trong chu trình hiện tại (với $i < j - 1$, và (v_n, v_1) là một cạnh).
   - **Kiểm tra cải tiến:** Giả sử hai cạnh bị loại bỏ là $A = (v_i, v_{i+1})$ và $B = (v_j, v_{j+1})$. Kiểm tra xem nếu thay thế chúng bằng hai cạnh mới $A' = (v_i, v_j)$ và $B' = (v_{i+1}, v_{j+1})$ có làm giảm tổng chi phí của chu trình không (tức là $dist(v_i, v_j) + dist(v_{i+1}, v_{j+1}) < dist(v_i, v_{i+1}) + dist(v_j, v_{j+1})$).
   - **Thực hiện hoán đổi (nếu có cải tiến):** Nếu việc thay thế làm giảm chi phí, thực hiện hoán đổi bằng cách:
     - Tách chu trình tại v_i và v_{j+1}.
     - Đảo ngược phần đường đi giữa v_{i+1} và v_j (tức là đoạn $v_{i+1} \to v_{i+2} \to \cdots \to v_j$ trở thành $v_j \to v_{j-1} \to \cdots \to v_{i+1}$).
     - Nối lại các phần để tạo thành chu trình mới: $v_1 \to \cdots \to v_i \to v_j \to v_{j-1} \to \cdots \to v_{i+1} \to v_{j+1} \to \cdots \to v_n \to v_1$.

- Đánh dấu rằng đã có cải tiến và tiếp tục vòng lặp chính.
- **Kết thúc:** Khi không tm thy hoán i 2-opt nào na có th ci thin chu trnh, thut toán dng li và tr v chu trnh hin ti.

Ví dụ minh họa: Giả sử có 4 thành phố (0, 1, 2, 3) với ma trận khoảng cách sau:

```
    0   1   2   3
0   0  10  10  10
1  10   0 100  10
2  10 100   0  10
3  10  10  10   0
```

(Lưu ý: Khoảng cách $d_{1,2} = 100$ là một cạnh dài bất thường, giúp minh họa rõ ràng tác dụng của 2-opt).

Giả sử chu trình ban đầu (tạo ngẫu nhiên hoặc bởi Nearest Neighbor) là: $0 \to 1 \to 2 \to 3 \to 0$. Tổng chi phí ban đầu: $d_{0,1} + d_{1,2} + d_{2,3} + d_{3,0} = 10 + 100 + 10 + 10 = 130$.

Tìm kiếm hoán đổi 2-opt: Xét cặp cạnh (v_i, v_{i+1}) và (v_j, v_{j+1}) không kề nhau. Chọn $i = 1$ (cạnh $(1, 2)$) và $j = 3$ (cạnh $(3, 0)$, trong chu trình thực tế là (v_n, v_1)). Hai cạnh hiện tại bị loại bỏ là $(1, 2)$ và $(3, 0)$. Chi phí của hai cạnh này: $d_{1,2} + d_{3,0} = 100 + 10 = 110$.

Đề xuất hoán đổi: Thử nối lại các đỉnh bằng hai cạnh mới: $(1, 3)$ và $(2, 0)$. Chi phí của hai cạnh mới: $d_{1,3} + d_{2,0} = 10 + 10 = 20$.

So sánh: $20 < 110$. Việc hoán đổi này sẽ cải thiện chu trình!

Thực hiện hoán đổi: Chu trình hiện tại: $0 \to \mathbf{1} \to \mathbf{2} \to \mathbf{3} \to 0$. Hai cạnh được chọn là $(1, 2)$ (tức $v_i = 1, v_{i+1} = 2$) và $(3, 0)$ (tức $v_j = 3, v_{j+1} = 0$). Để hoán đổi, ta đảo ngược phần đường đi giữa v_{i+1} và v_j, tức là đoạn từ 2 đến 3. Đoạn này là $2 \to 3$. Đảo ngược thành $3 \to 2$. Chu trình mới sẽ là: $0 \to v_i \to v_j \to \cdots \to v_{i+1} \to v_{j+1} \to \cdots \to 0$. $0 \to \mathbf{1} \to \mathbf{3} \to \mathbf{2} \to 0$.

Tổng chi phí của chu trình mới: $d_{0,1} + d_{1,3} + d_{3,2} + d_{2,0} = 10 + 10 + 10 + 10 = 40$. Chu trình đã được cải thiện từ 130 xuống 40. Thuật toán sẽ tiếp tục tìm kiếm các hoán đổi khác cho đến khi không có cải tiến nào nữa.

Độ phức tạp: * Mỗi lần duyệt qua tất cả các cặp cạnh để tìm một hoán đổi cải thiện cần $O(n^2)$ thời gian (vì có $O(n^2)$ cặp cạnh để kiểm tra). * Thao tác đảo ngược một đoạn đường đi cũng mất $O(n)$ thời gian. * Số lần lặp lại vòng lặp chính ('while improved') là không xác định trước, nhưng trong trường hợp xấu nhất có thể rất lớn. Tuy nhiên, trong thực tế, nó thường hội tụ khá nhanh. * Do đó, độ phức tạp điển hình của 2-opt là $**O(n^2)$ cho mỗi pass**, và tổng độ phức tạp phụ thuộc vào số pass cần thiết để hội tụ (thường được coi là hiệu quả cho các bài toán lớn).

Ưu điểm:

- Đơn giản để cài đặt và hiểu.
- Khá hiệu quả trong việc tìm kiếm các giải pháp tốt (gần tối ưu) trong thời gian hợp lý cho các bài toán TSP lớn, nơi các thuật toán tối ưu không khả thi.
- Linh hoạt, có thể được sử dụng để cải thiện một chu trình ban đầu được tạo bởi bất kỳ phương pháp nào khác.

Nhược điểm:

- **Dễ bị mắc kẹt tại tối ưu cục bộ:** Đây là một heuristic, nó không đảm bảo tìm được chu trình ngắn nhất tuyệt đối. Nó sẽ dừng lại khi không có hoán đổi 2-opt nào có thể cải thiện thêm, ngay cả khi tồn tại một giải pháp tốt hơn mà cần nhiều hơn một hoán đổi để đạt được. Kết quả phụ thuộc vào chu trình ban đầu được cung cấp.
- **Có thể mất nhiều thời gian để hội tụ:** Mặc dù mỗi pass là $O(n^2)$, tổng thời gian chạy phụ thuộc vào số pass cần thiết để hội tụ, có thể lớn đối với một số trường hợp.

Pseudocode (mã giả):

```
function twoOptSwap(route, i, j):
// Thuc hien hoan doi 2-opt tren 'route' giua cac chi so i va j.
// route = [.., route[i], route[i+1], .., route[j], route[j+1], ..]
// Ket qua: [.., route[i], route[j], route[j-1], .., route[i+1], route[j+1], ..]
// Phan 1: tu dau den route[i]
segment1 = route[0..i]
// Phan 2: tu route[i+1] den route[j], dao nguoc
segment2_reversed = reversed(route[i+1..j])
// Phan 3: tu route[j+1] den cuoi chu trinh
segment3 = route[j+1..length(route)-1]
return segment1 + segment2_reversed + segment3

function twoOpt(initial_route, khoangCach):
best_route = initial_route
improved = true
n_cities = length(best_route) - 1 // So luong thanh pho thuc te
while improved:
    improved = false
    current_best_distance = calculate_distance(best_route, khoangCach)
    // Duyet qua tat ca cac cap canh khong ke nhau
    // (route[i], route[i+1]) va (route[j], route[j+1])
    for i from 0 to n_cities - 1:
        for j from i + 1 to n_cities - 1:
            // Bo qua cac cap canh ke nhau hoac trung lap
            if j == i + 1: continue
            if i == 0 and j == n_cities - 1: continue
            candidate_route = twoOptSwap(best_route, i, j)
            candidate_distance = calculate_distance(
                candidate_route, khoangCach)
            if candidate_distance < current_best_distance:
                best_route = candidate_route
                current_best_distance = candidate_distance
                improved = true

return best_route, current_best_distance

function calculate_distance(route, khoangCach):
```

```
distance = 0
// route = [v0, v1, .., v(N-1), v0], length(route) = N+1
// Tinh tong chi phi cua N canh trong chu trinh
for i from 0 to length(route) - 2:
    distance = distance + khoangCach[route[i]][route[i+1]]
return distance
```

Listing 4: Pseudocode Thuật toán 2-opt (Ngắn gọn)

So sánh các thuật toán Để có cái nhìn tổng quan và đánh giá hiệu quả của các thuật toán đã nghiên cứu, nhóm tiến hành so sánh chúng dựa trên các tiêu chí quan trọng. Các nhận định và số liệu trong bảng so sánh này được tổng hợp dựa trên kiến thức phổ quát trong lĩnh vực Lý thuyết đồ thị và Giải thuật, tham khảo từ các giáo trình và tài liệu học thuật uy tín đã được liệt kê trong mục của báo cáo này.

Tiêu chí so sánh	**Hierholzer (Euler)**	**Backtracking (Hamilton)**	**Brute-force (TSP)**	**Nearest Neighbor (TSP Heuristic)**	**Held-Karp (TSP DP)**	**2-opt (TSP Heuristic)**
Mục tiêu	Tìm chu trình/đường đi qua tất cả các **cạnh** một lần.	Tìm chu trình/đường đi qua tất cả các **đỉnh** một lần.	Tìm chu trình qua tất cả các **đỉnh** một lần với **chi phí tối thiểu**.	Tìm chu trình qua tất cả các **đỉnh** một lần với **chi phí gần tối ưu**.	Tìm chu trình qua tất cả các **đỉnh** một lần với **chi phí tối thiểu** (chính xác).	Cải thiện chu trình TSP hiện có để tìm một chu trình với **chi phí gần tối ưu**.
Loại bài toán	Tồn tại và xây dựng.	Tồn tại và xây dựng.	Tối ưu (tìm Min/Max).	Tối ưu (tìm xấp xỉ Min/Max).	Tối ưu (tìm Min/Max) - chính xác.	Tối ưu (tìm xấp xỉ Min/Max) - heuristic.
Độ phức tạp thời gian	Rất hiệu quả: $O(\|V\| + \|E\|)$	Rất kém hiệu quả: $O(V!)$	Cực kỳ kém hiệu quả: $O(n!)$	Hiệu quả cho đồ thị lớn: $O(n^2)$ hoặc $O(n^2 \log n)$	$O(n^2 \cdot 2^n)$	Tốt (thường $O(n^2)$ cho mỗi lần cải tiến, lặp đến khi ổn định)
Đảm bảo tối ưu	Có (nếu chu trình tồn tại)	Có (nếu chu trình tồn tại)	Có (tìm ra chính xác giá trị tối thiểu)	Không (chỉ là giải pháp gần tối ưu)	Có (tìm ra chính xác giá trị tối thiểu)	Không (chỉ là giải pháp gần tối ưu/tối ưu cục bộ)
Khả năng mở rộng	Tốt (cho đồ thị lớn)	Kém (chỉ cho đồ thị nhỏ $<$ 15-20 đỉnh)	Rất kém (chỉ cho đồ thị rất nhỏ $<$ 10-12 đỉnh)	Tốt (cho đồ thị lớn hàng ngàn đỉnh)	Kém (chỉ cho đồ thị nhỏ ≤ 20 đỉnh)	Tốt (cho đồ thị lớn)

Tiếp theo trên trang sau

Tiêu chí so sánh	**Hierholzer (Euler)**	**Backtracking (Hamilton)**	**Brute-force (TSP)**	**Nearest Neighbor (TSP Heuristic)**	**Held-Karp (TSP DP)**	**2-opt (TSP Heuristic)**
Phạm vi áp dụng	Các bài toán phủ sóng mạng lưới (quét dọn, kiểm tra).	Các bài toán cần thăm tất cả các địa điểm mà không lặp lại (ít phổ biến trong thực tế do TSP).	Chỉ các bài toán TSP với số đỉnh rất nhỏ.	Các bài toán TSP thực tế với số đỉnh lớn.	Các bài toán TSP với số đỉnh vừa phải.	Các bài toán TSP thực tế với số đỉnh lớn.
Ưu điểm	Đơn giản, nhanh, chính xác.	Đảm bảo tìm được nếu có.	Đảm bảo tìm được lời giải tối ưu toàn cục.	Nhanh, dễ thực hiện, cung cấp giải pháp hợp lý trong thời gian ngắn.	Đảm bảo tìm được lời giải tối ưu toàn cục, hiệu quả hơn Brute-force cho n vừa.	Đơn giản, dễ cài đặt, nhanh chóng tìm được giải pháp tốt cho đồ thị lớn.
Nhược điểm	Chỉ áp dụng khi thỏa mãn điều kiện bậc chẵn.	Rất chậm, không thực tế với đồ thị lớn.	Rất chậm, không thực tế với đồ thị lớn.	Không đảm bảo tối ưu, có thể bị mắc kẹt ở tối ưu cục bộ. Kết quả có thể khác nhau tùy điểm bắt đầu.	Độ phức tạp thời gian mũ, không thực tế với đồ thị lớn.	Không đảm bảo tối ưu toàn cục, có thể bị mắc kẹt ở tối ưu cục bộ.

Kết luận:

- **Hierholzer** là thuật toán lý tưởng và hiệu quả để giải quyết các bài toán liên quan đến chu trình Euler, miễn là đồ thị thỏa mãn các điều kiện tồn tại.
- **Backtracking** cho chu trình Hamilton mặc dù đảm bảo tìm được lời giải nếu có, nhưng độ phức tạp giai thừa khiến nó không thực tế cho hầu hết các ứng dụng thực tế. Trong thực tế, chu trình Hamilton thường được giải quyết thông qua TSP.
- **TSP Brute-force** chỉ mang tính lý thuyết hoặc ứng dụng trong những trường hợp cực kỳ nhỏ.
- **Held-Karp** cung cấp một giải pháp chính xác cho TSP hiệu quả hơn vét cạn, nhưng vẫn bị giới hạn bởi độ phức tạp mũ, chỉ phù hợp với số lượng đỉnh nhỏ đến vừa.
- Đối với **TSP** trong thực tế với số lượng đỉnh lớn, các **thuật toán heuristic** (như Nearest Neighbor, **2-opt** và các thuật toán phức tạp hơn như Ant Colony Optimization, Genetic Algorithms, Simulated Annealing) là lựa chọn tối ưu, cân bằng giữa tốc độ và chất lượng giải pháp, mặc dù không đảm bảo tối ưu toàn cục.

5 Sản phẩm(Code)

5.1 Phân tích Bài toán thực tế và Áp dụng

5.1.1 Bài toán giao hàng

Bài toán giao hàng là một vấn đề phổ biến trong logistics, nơi một xe hoặc nhân viên cần đi từ một kho hàng, ghé thăm một tập hợp các điểm giao hàng cụ thể và sau đó quay trở lại kho ban đầu. Mục tiêu thường là tối thiểu hóa tổng quãng đường di chuyển hoặc thời gian.

5.1.2 Mô hình hóa bài toán

Để giải quyết bài toán giao hàng bằng lý thuyết đồ thị, chúng ta mô hình hóa:

- Các điểm giao hàng: Là các đỉnh của đồ thị (ví dụ: điểm 0 là kho, các điểm 1, 2, ... là các địa điểm cần giao hàng).
- Các tuyến đường giữa các điểm: Là các cạnh của đồ thị.
- Khoảng cách/chi phí giữa các điểm: Là trọng số của các cạnh.

Trong đề tài này, chúng tôi sử dụng đồ thị vô hướng. Lý do là ma trận kề được xây dựng từ khoảng cách Euclidean giữa các tọa độ điểm, và khoảng cách này là đối xứng (khoảng cách từ A đến B bằng khoảng cách từ B đến A). Điều này đơn giản hóa mô hình và phù hợp với nhiều tình huống giao hàng thực tế khi đường đi là hai chiều và không có sự khác biệt về chi phí di chuyển giữa hai hướng.

5.1.3 Áp dụng Chu trình Euler

Chu trình Euler được áp dụng trong các bài toán cần đi qua tất cả các tuyến đường (cạnh) của một mạng lưới đúng một lần.

- **Ví dụ ứng dụng:** Xe quét rác cần đi qua mọi con đường trong khu phố, nhân viên kiểm tra đường ống cần đi dọc theo tất cả các đường ống, hoặc xe tải chở thư cần ghé thăm mọi con phố.
- **Cách áp dụng:** Khi cần đảm bảo mọi "tuyến đường" đều được đi qua một lần và quay về điểm xuất phát, chúng ta tìm chu trình Euler trên đồ thị biểu diễn các tuyến đường đó.

5.1.4 Áp dụng Chu trình Hamilton/TSP

Chu trình Hamilton và bài toán TSP được áp dụng trong các bài toán cần ghé thăm tất cả các địa điểm (đỉnh) đúng một lần.

- **Ví dụ ứng dụng:** Nhân viên giao hàng cần ghé thăm một danh sách các khách hàng cụ thể, nhân viên bảo trì cần kiểm tra tất cả các trạm, hoặc lập kế hoạch tuyến đường cho nhân viên bán hàng.
- **Cách áp dụng:** Khi cần tìm đường đi ngắn nhất hoặc hiệu quả nhất để ghé thăm một tập hợp các địa điểm, chúng ta giải bài toán TSP trên đồ thị biểu diễn các địa điểm và khoảng cách giữa chúng.

5.2 Triển khai và Kết quả

Chương trình được triển khai bằng **Python**, sử dụng các thư viện `numpy` cho tính toán ma trận, `matplotlib` cho trực quan hóa, và `collections` cho cấu trúc dữ liệu.

5.2.1 Kiến trúc chương trình

Chương trình được xây dựng xung quanh lớp `VisualizedDeliveryProblemSolver`, bao gồm:

- `__init__(self, adjacency_matrix, coordinates=None)`: Khởi tạo lớp với ma trận kề và tọa độ các điểm. Nếu tọa độ không được cung cấp, hệ thống sẽ tự động tạo ngẫu nhiên.
- `generate_coordinates(self)`: Tạo tọa độ ngẫu nhiên cho các điểm giao hàng.
- `create_matrix_from_coordinates(coordinates)`: Tạo ma trận kề từ các tọa độ điểm (sử dụng khoảng cách Euclidean).
- `floyd_warshall(self)`: Triển khai thuật toán Floyd-Warshall để tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh.
- `calculate_path_cost(self, path)`: Tính tổng chi phí của một đường đi cho trước.

5.2.2 Các chức năng chính

- **Tìm Chu trình Euler (`find_eulerian_cycle`)**:
  - Thuật toán Hierholzer được sử dụng để tìm chu trình Euler.
  - **Cải tiến:** Để đảm bảo tính đúng đắn cho đồ thị vô hướng, phần xây dựng danh sách kề đã được điều chỉnh để thêm cạnh ở cả hai chiều (`(i, j)` và `(j, i)`). Đồng thời, một bước kiểm tra tính liên thông của đồ thị (không tính các đỉnh cô lập) đã được bổ sung trước khi chạy thuật toán Hierholzer, đảm bảo chu trình tìm được bao phủ toàn bộ các cạnh của thành phần liên thông.
  - **Điều kiện:** Chương trình kiểm tra điều kiện cần và đủ: tất cả các đỉnh phải có bậc chẵn và đồ thị phải liên thông.
- **Tìm Chu trình Hamilton/TSP**:
  - `tsp_dynamic_programming(self)`: Triển khai thuật toán Held-Karp. Đây là phương pháp tối ưu, đảm bảo tìm được chu trình Hamilton ngắn nhất.
  - `tsp_nearest_neighbor(self, start=0)`: Triển khai thuật toán heuristic láng giềng gần nhất, nhanh chóng nhưng không đảm bảo tối ưu.
  - `tsp_2opt(self, initial_path=None, max_iterations=1000)`: Triển khai thuật toán cải tiến cục bộ 2-opt, được sử dụng để tối ưu chu trình ban đầu (ví dụ từ Nearest Neighbor).
- **Trực quan hóa (`visualize_graph` và `animate_path`)**:
  - `visualize_graph(self, path=None, title='Graph Visualization', filename=None)` Vẽ đồ thị với các đỉnh (số hiệu và tọa độ) và các cạnh. Có thể vẽ thêm một đường đi cụ thể trên đồ thị.
  - `animate_path(self, path, title='Path Animation', interval=500, filename=None)` Tạo hoạt ảnh mô phỏng quá trình di chuyển theo một đường đi nhất định, giúp người dùng dễ dàng hình dung lộ trình.

5.3 Kết quả thực nghiệm và đánh giá

Nhóm đã thực hiện các thử nghiệm trên các đồ thị có số lượng đỉnh khác nhau (ví dụ: 5 điểm, 8 điểm, 12 điểm) để đánh giá hoạt động của các thuật toán.

Mô tả chương trình triển khai

- Kiểm tra điều kiện tồn tại chu trình Euler, chu trình Hamilton.
- Tìm chu trình Euler bằng thuật toán Hierholzer.
- Tìm chu trình Hamilton bằng thuật toán Backtracking.
- Giải bài toán TSP bằng các phương pháp: Held-Karp (quy hoạch động), láng giềng gần nhất (Nearest Neighbor), và 2-opt.

5.4 Ví dụ thử nghiệm

Xét một đồ thị 5 đỉnh với ma trận trọng số:

- **Chu trình Euler:**
  - Với đồ thị đầy đủ (K5), chương trình đã tìm được một chu trình Euler hợp lệ, đi qua tất cả 10 cạnh của đồ thị.
    Ví dụ đường đi: `0 -> 1 -> 2 -> 0 -> 3 -> 1 -> 4 -> 2 -> 3 -> 4 -> 0`.
    Chu trình này chứng minh khả năng của thuật toán Hierholzer trong việc bao phủ toàn bộ mạng lưới.
- **Chu trình Hamilton/TSP:**
  - **TSP với Quy hoạch động (DP):** Đối với các đồ thị nhỏ (ví dụ 5-10 điểm), thuật toán DP luôn tìm được đường đi ngắn nhất (chu trình Hamilton tối ưu) với chi phí thấp nhất. Tuy nhiên, thời gian tính toán tăng lên rất nhanh khi số lượng đỉnh tăng, làm cho nó không thực tế với hơn 20 đỉnh.

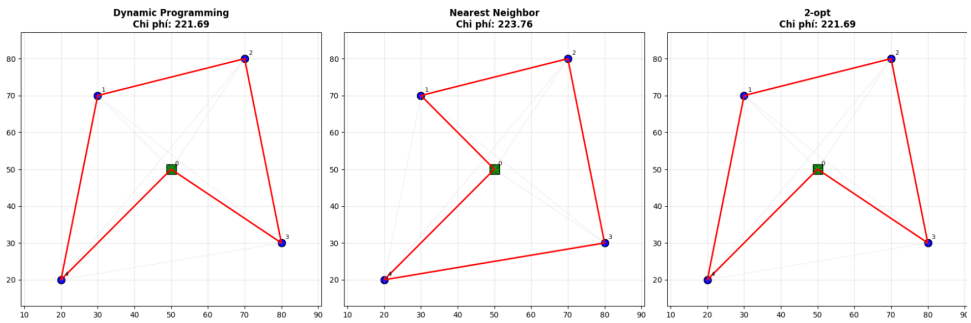
    ```
    Đường đi (DP): 0 -> 1 -> 2 -> 3 -> 4 -> 0
    Chi phí (DP): 221.69
    ```

  - **TSP với Nearest Neighbor (NN):** Cung cấp một giải pháp nhanh chóng cho chu trình Hamilton. Tuy nhiên, chi phí đường đi thường cao hơn so với DP do bản chất tham lam.

    ```
    Đường đi (NN): 0 -> 1 -> 4 -> 2 -> 3 -> 0
    Chi phí (NN): 223.76
    ```

  - **TSP với 2-opt:** Đã cho thấy khả năng cải thiện đáng kể đường đi được tạo bởi Nearest Neighbor. Chi phí thường gần hơn với kết quả của DP so với NN ban đầu, đặc biệt sau nhiều vòng lặp.

    ```
    Đường đi (2-opt): 0 -> 3 -> 2 -> 1 -> 4 -> 0
    Chi phí (2-opt): 221.69
    ```



Hình 7: So sánh đường đi của các thuật toán TSP

- **Trực quan hóa:** Các chức năng `visualize_graph` và `animate_path` đã chứng minh hiệu quả trong việc giúp người dùng hình dung cấu trúc đồ thị và quá trình di chuyển trên đường đi được tìm thấy, tăng tính tương tác và dễ hiểu cho giải pháp.

6 Nhận xét

6.1 Ưu điểm của các phương pháp sử dụng

Dựa trên quá trình triển khai và thử nghiệm, chúng tôi nhận thấy các ưu điểm nổi bật của từng thuật toán như sau:

- **Thuật toán Hierholzer (Tìm chu trình Euler):**
  - **Hiệu quả:** Thuật toán này có độ phức tạp thời gian tuyến tính ($O(E)$), làm cho nó rất hiệu quả để tìm chu trình Euler ngay cả trên các đồ thị lớn.
  - **Chắc chắn tìm được:** Nếu chu trình Euler tồn tại, thuật toán Hierholzer đảm bảo sẽ tìm thấy nó.
  - **Đơn giản trong nguyên lý:** Ý tưởng cơ bản là đi theo các cạnh và xóa chúng, sau đó nối các chu trình con, khá trực quan.
- **Thuật toán Backtracking (Tìm chu trình Hamilton):**
  - **Tính đầy đủ:** Đảm bảo tìm được một chu trình Hamilton nếu nó tồn tại (hoặc xác định nếu không tồn tại) bằng cách duyệt toàn bộ không gian lời giải.
  - **Tính tổng quát:** Là một kỹ thuật giải quyết vấn đề tổng quát cho nhiều bài toán tổ hợp, có thể áp dụng cho các biến thể khác.
- **Thuật toán Brute-force (TSP):**
  - **Đảm bảo tối ưu:** Luôn tìm ra chu trình Hamilton có chi phí thấp nhất (tức là lời giải tối ưu cho TSP).
  - **Đơn giản về ý tưởng:** Nguyên lý cơ bản là thử tất cả các hoán vị của các đỉnh, dễ hiểu.
- **Thuật toán Quy hoạch Động - Held-Karp (TSP):**
  - **Đảm bảo tối ưu:** Giống như Brute-force, Held-Karp luôn tìm ra lời giải tối ưu cho bài toán TSP.
  - **Hiệu quả hơn Brute-force:** Mặc dù vẫn có độ phức tạp theo cấp số mũ, nó hiệu quả hơn đáng kể so với Brute-force cho các đồ thị nhỏ.
- **Thuật toán láng giềng gần nhất (Nearest Neighbor Algorithm - TSP):**
  - **Tốc độ cao:** Rất nhanh để tìm ra một đường đi, phù hợp cho các bài toán có số lượng đỉnh lớn khi cần một giải pháp gần đúng nhanh chóng.
  - **Đơn giản để cài đặt:** Thuật toán tham lam này có logic đơn giản, dễ dàng triển khai.

- **Thuật toán 2-opt (TSP):**
  - **Khả năng cải thiện:** Có khả năng cải thiện đáng kể chất lượng của các giải pháp ban đầu (ví dụ từ Nearest Neighbor), đưa chi phí đường đi gần hơn đến mức tối ưu.
  - **Hiệu quả cục bộ:** Dù không đảm bảo tối ưu toàn cục, nó là một phương pháp cải tiến cục bộ hiệu quả.
  - **Tương đối dễ cài đặt:** Ý tưởng hoán đổi 2 cạnh khá trực quan để lập trình.

6.2 Nhược điểm của các phương pháp sử dụng

Bên cạnh những ưu điểm, mỗi thuật toán cũng tồn tại những hạn chế và nhược điểm cần được xem xét:

- **Thuật toán Hierholzer (Tìm chu trình Euler):**
  - **Yêu cầu nghiêm ngặt về đồ thị:** Chỉ áp dụng được khi tất cả các đỉnh đều có bậc chẵn và đồ thị liên thông. Không thể tìm chu trình nếu đồ thị không thỏa mãn điều kiện này.
- **Thuật toán Backtracking (Tìm chu trình Hamilton):**
  - **Độ phức tạp tính toán cao:** Là một thuật toán vét cạn có độ phức tạp theo cấp số mũ, khiến nó không thực tế cho các đồ thị có số lượng đỉnh lớn. Thời gian chạy tăng rất nhanh khi số đỉnh tăng.
  - **Không hiệu quả cho bài toán tối ưu:** Bản thân Backtracking chỉ tìm một chu trình Hamilton, không tối ưu về chi phí.
- **Thuật toán Brute-force (TSP):**
  - **Độ phức tạp cực kỳ cao:** Có độ phức tạp là $O(N!)$, làm cho nó hoàn toàn không thực tế ngay cả với số lượng đỉnh nhỏ (ví dụ, với 20 đỉnh, số hoán vị là cực lớn).
  - **Không thể sử dụng trong thực tế:** Do độ phức tạp, nó không có giá trị thực tiễn cho các bài toán lớn.
- **Thuật toán Quy hoạch Động - Held-Karp (TSP):**
  - **Độ phức tạp theo cấp số mũ:** Mặc dù tốt hơn Brute-force, độ phức tạp $O(N^2 \cdot 2^N)$ vẫn khiến nó chỉ khả thi cho đồ thị có tối đa khoảng 20-25 đỉnh.
  - **Yêu cầu bộ nhớ lớn:** Đòi hỏi một lượng bộ nhớ đáng kể để lưu trữ các bảng con của quy hoạch động.
- **Thuật toán láng giềng gần nhất (Nearest Neighbor Algorithm - TSP):**
  - **Không đảm bảo tối ưu:** Là một thuật toán tham lam, nó có thể mắc kẹt ở một tối ưu cục bộ và không tìm được đường đi ngắn nhất toàn cục.
  - **Phụ thuộc điểm bắt đầu:** Kết quả có thể khác nhau tùy thuộc vào đỉnh bắt đầu được chọn.

- **Thuật toán 2-opt (TSP):**
  - **Chỉ tìm tối ưu cục bộ:** Không đảm bảo tìm được chu trình tối ưu toàn cục; có thể bị mắc kẹt tại một tối ưu cục bộ.
  - **Phụ thuộc vào đường đi ban đầu:** Chất lượng của kết quả cuối cùng phụ thuộc nhiều vào chất lượng của đường đi ban đầu được cung cấp cho nó.
  - **Không phải là thuật toán tạo chu trình:** 2-opt là một thuật toán cải tiến, không phải là thuật toán để tự tạo ra một chu trình.

7 Kết luận

7.1 Tóm tắt kết quả và Đánh giá

Thông qua việc thực hiện đề tài, nhóm đã đạt được những kết quả quan trọng như:

- Hiểu rõ bản chất và điều kiện tồn tại của các chu trình Euler, Hamilton, cùng với các ứng dụng thực tiễn trong bài toán tìm đường đi và phân phát hàng hóa.
- Cài đặt và vận hành các thuật toán cơ bản (Hierholzer, Held-Karp, Nearest Neighbor, 2-opt) trên đồ thị, có khả năng trực quan hóa kết quả.
- Nâng cao kỹ năng xử lý dữ liệu, tổ chức mã lệnh và trực quan hóa kết quả, đồng thời phân tích được ưu nhược điểm của từng thuật toán về độ chính xác và thời gian chạy. Cụ thể, các thuật toán tối ưu (Hierholzer, Held-Karp) cho kết quả chính xác nhưng hạn chế với đồ thị lớn, trong khi các thuật toán heuristic (Nearest Neighbor, 2-opt) mang lại giải pháp nhanh chóng cho quy mô lớn.

Tuy nhiên, nhóm cũng gặp phải một số khó khăn và nhận thấy các hạn chế như:

- Khả năng tối ưu hóa thời gian chạy cho các thuật toán vét cạn và backtracking là rất thấp với đồ thị có kích thước lớn.
- Việc trực quan hóa kết quả vẫn chưa được tích hợp hoàn toàn vào một giao diện tương tác người dùng thân thiện.

7.2 Hướng phát triển

Để nâng cao chất lượng và tính ứng dụng của đề tài, có thể xem xét các hướng phát triển sau trong tương lai:

- **Hỗ trợ đồ thị có hướng:** Mở rộng các thuật toán để xử lý các đồ thị có hướng, phù hợp với các tuyến đường một chiều hoặc chi phí di chuyển không đối xứng.
- **Thuật toán TSP nâng cao:** Triển khai thêm các thuật toán metaheuristic khác (ví dụ: Simulated Annealing, Genetic Algorithms, Ant Colony Optimization) để giải quyết bài toán TSP trên đồ thị lớn hơn và phức tạp hơn.
- **Tích hợp dữ liệu thực tế:** Phát triển khả năng nhập và xử lý dữ liệu bản đồ thực tế (ví dụ từ OpenStreetMap, Google Maps API) để mô hình hóa các mạng lưới giao thông phức tạp hơn.

- **Giao diện người dùng (GUI):** Xây dựng một giao diện đồ họa thân thiện và trực quan hơn để người dùng có thể tương tác dễ dàng với chương trình và các thuật toán.
- **Mở rộng bài toán:** Mở rộng mô hình để giải quyết các biến thể phức tạp hơn của bài toán định tuyến, chẳng hạn như bài toán định tuyến phương tiện (VRP) với nhiều kho, nhiều xe, hoặc các ràng buộc về năng lực và thời gian.

Tài liệu tham khảo

1. Kenneth H. Rosen, *Discrete Mathematics and Its Applications*, 7th Edition, McGraw-Hill, 2012.
2. Reinhard Diestel, *Graph Theory*, Springer, 5th edition, 2017.
3. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, MIT Press, 3rd edition, 2009.
4. Croes, G. A. (1958). *A method for solving traveling-salesman problems*. Operations Research, 6(6), 791–812.
5. Bellman, R. (1962). *Dynamic Programming*. Princeton University Press.
6. Held, M., & Karp, R. M. (1962). *A dynamic programming approach to sequencing problems*. Journal of the Society for Industrial and Applied Mathematics, 10(2), 196–210.
7. Wikipedia contributors, "Eulerian path," *Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/wiki/Eulerian_path.
8. Wikipedia contributors, "Hamiltonian path," *Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/wiki/Hamiltonian_path.
9. Cormen et al. (2009). Introduction to Algorithms. MIT Press.

Phân công công việc

Họ tên	**Công việc phụ trách**	**Tiến độ**
Lê Hoàng An	- Trình bày các khái niệm: + Ma trận kề, đồ thị liên thông + Chu trình, khái niệm Euler và Hamilton	100%
Trần Duy An	- Viết Điều kiện đồ thị có chu trình Euler, chu trình Hamilton, thuật toán tìm chu trình Euler, chu trình Hamilton + Xây dựng slide Powerpoint và nội dung thuyết trình; Thuyết trình.	100%
Phạm Hoàng Dũng	Làm sản phẩm: Viết chương trình (matlab, python) nhập vào ma trận kề mô tả bài toán giao hàng + Viết báo cáo sản phẩm + Sửa lại báo cáo	120%
Nguyễn Kỳ Anh	- Tìm kiếm tài liệu; nhận xét ưu nhược điểm của thuật toán và ví dụ; đưa ra kết luận của thuật toán + Kết luận + Tổng hợp nội dung các phần	80%