

BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC SƯ PHẠM TP. HỒ CHÍ MINH

Nhóm học viên thực hiện:

TRẦN THANH NHÃ

KHMT-17-005

NGUYỄN PHƯƠNG NAM

KHMT-17-004

DƯƠNG XUÂN HUY

KHMT-17-002

LÝ THUYẾT ĐỒ THỊ

(GRAPH THEORY)

Chuyên ngành: **Khoa học máy tính**

Môn học: **Phương pháp toán trong tin học**

Người hướng dẫn: **TS. HUỖNH VĂN ĐỨC**

Tp. Hồ Chí Minh, tháng 8 năm 2018

BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC SƯ PHẠM TP. HỒ CHÍ MINH

LÝ THUYẾT ĐỒ THỊ (GRAPH THEORY)

Chuyên ngành: **Khoa học máy tính**

Môn học: **Phương pháp toán trong tin học**

Tp. Hồ Chí Minh, tháng 8 năm 2018

MỤC LỤC

MỤC LỤC	I
KẾT CẤU TIỂU LUẬN.....	3
NHIỆM VỤ THÀNH VIÊN NHÓM.....	3
MỞ ĐẦU.....	4
CHƯƠNG 1. CÁC KHÁI NIỆM CƠ BẢN VỀ LÝ THUYẾT ĐỒ THỊ.....	5
1.1 Khái niệm	5
1.1.1 Định nghĩa	5
1.1.2 Biểu diễn đồ thị.....	6
1.1.3 Đồ thị có hướng và đồ thị vô hướng	6
1.1.4 Đơn đồ thị (simple graph), đa đồ thị (Multigraph)	6
1.1.5 Một số dạng đơn đồ thị đặc biệt.....	7
1.1.6 Bậc của một đỉnh (degree).....	10
1.2 Đường đi, chu trình và liên thông	10
1.2.1 Đường đi.....	11
1.2.2 Chu trình.....	11
1.2.3 Liên thông.....	12
CHƯƠNG 2. BÀI TOÁN ĐƯỜNG ĐI NGẮN NHẤT	14
2.1 Giới thiệu.....	14
2.2 Đồ thị có trọng số.....	15
2.3 Bài toán đường đi ngắn nhất	16
2.3.1 Đồ thị không có chu trình âm - thuật toán Ford Bellman	16
2.3.2 Trường hợp trọng số trên các cung không âm - thuật toán dijkstra.....	18
2.3.3 Đường đi ngắn nhất giữa mọi cặp đỉnh - thuật toán Floyd-Warshall.....	20
2.3.4 Trường hợp đồ thị không có chu trình- sắp xếp Tô pô	21
2.4 Ứng dụng của bài toán tìm đường đi ngắn nhất	23
2.4.1 GPS.....	23
2.4.2 Phân tích và xử lý ảnh.....	23
2.4.3 Công cụ tìm kiếm qua mạng Internet (Social Search)	24
2.4.4 Hệ thống tin nhắn.....	25
Tiểu kết chương 2	26
CHƯƠNG 3. LUỒNG TRÊN MẠNG (NETWORK FLOW)	27
3.1 Giới thiệu chung	27
3.2 Bài toán luồng trên mạng (Network flow problem).....	27
3.2.1 Định nghĩa	27
3.2.2 Bài toán luồng cực đại (Maximum-Flow).....	29
3.2.3 Thuật toán Ford-Fulkerson	33

CHƯƠNG 4. CÂY KHUNG NHỎ NHẤT (MINIMUM SPANNING TREE)	37
4.1 Định nghĩa	37
4.2 Bài toán tìm cây khung nhỏ nhất	38
4.2.1 Bài toán xây dựng hệ thống đường sắt.....	38
4.2.2 Bài toán nối mạng máy tính	38
4.3 Thuật toán Kruskal	38
4.3.1 Mô tả thuật toán	39
4.3.2 Cài đặt thuật toán Kruskal	42
4.3.3 Chứng minh tính đúng đắn	43
4.3.4 Độ phức tạp của thuật toán Kruskal.....	43
4.4 Thuật toán Prim	44
4.4.1 Mô tả thuật toán Prim:	44
4.4.2 Thuật toán Prim	44
4.4.3 Cài đặt thuật toán Prim	48
4.4.4 Độ phức tạp thuật toán Prim	49
4.5 SO SÁNH HAI THUẬT TOÁN Prim và Kruskal	49
4.5.1 Xét các cạnh đưa vào cây	49
4.5.2 . Kiểm tra tính liên thông của đồ thị	49
4.5.3 . Chi phí cho việc kiểm tra chu trình.....	49
KẾT LUẬN	51
TÀI LIỆU THAM KHẢO.....	52

KẾT CẤU TIỂU LUẬN

Ngoài phần mở đầu và kết luận, bài tiểu luận gồm có 4 chương:

- Chương 1: Các khái niệm cơ bản về lý thuyết đồ thị.
- Chương 2: Bài toán đường đi ngắn nhất.
- Chương 3: Luồng trên mạng (network flow problem).
- Chương 4: Cây khung nhỏ nhất (Minimum spanning tree).

NHIỆM VỤ THÀNH VIÊN NHÓM

- Chương 1: toàn nhóm
- Chương 2: Nguyễn Phương Nam
- Chương 3: Dương Xuân Huy
- Chương 4: Trần Thanh Nhã

MỞ ĐẦU

Lý thuyết đồ thị là một ngành khoa học được phát triển từ lâu nhưng lại có nhiều ứng dụng hiện đại. Những ý tưởng cơ bản của nó được đưa ra từ thế kỷ 18 bởi nhà toán học Thụy Sĩ tên là Leonhard Euler. Ông đã dùng đồ thị để giải quyết bài toán 7 chiếc cầu Königsberg nổi tiếng. Đồ thị được dùng để giải các bài toán trong nhiều lĩnh vực khác nhau. Ví dụ, ta dùng đồ thị để: - Xác định xem có thực hiện một mạch điện trên một bảng điện phẳng được không. - Xác định xem hai máy tính có được nối với nhau bằng một đường truyền thông hay không thông qua mô hình đồ thị mạng máy tính. - Giải các bài toán như bài toán tìm đường đi ngắn nhất giữa hai thành phố trong một mạng giao thông (sau khi đã gán các trọng số cho các cạnh của nó). - Lập sơ đồ khối tính toán của một thuật toán. - Giải các bài toán như bài toán tính số các tổ hợp khác nhau của các chuyến bay giữa hai thành phố trong một mạng hàng không. - Tìm số các màu cần thiết để tô các vùng khác nhau của một bản đồ...

Trên thực tế có nhiều bài toán liên quan tới một tập các đối tượng và những mối liên hệ giữa chúng, đòi hỏi toán học phải đặt ra một mô hình biểu diễn một cách chặt chẽ và tổng quát bằng ngôn ngữ ký hiệu, đó là đồ thị. Đặc biệt trong khoảng vài mươi năm trở lại đây, cùng với sự ra đời của máy tính điện tử và sự phát triển nhanh chóng của Tin học, Lý thuyết đồ thị càng được quan tâm đến nhiều hơn. Các thuật toán trên đồ thị đã có nhiều ứng dụng trong nhiều lĩnh vực khác nhau như: Mạng máy tính, Lý thuyết mã, Tối ưu hoá, Kinh tế học v.v.... Hiện nay, môn học này là một trong những kiến thức cơ sở của ngành khoa học máy tính nói chung và bộ môn Phương pháp toán trong tin học nói riêng. Chính vì những ứng dụng có tầm quan trọng đó, nhóm học viên đã lựa chọn đề tài “Lý thuyết đồ thị” để nghiên cứu, tìm hiểu cho bài tập của môn học.

CHƯƠNG 1. CÁC KHÁI NIỆM CƠ BẢN VỀ LÝ THUYẾT ĐỒ THỊ

1.1 Khái niệm

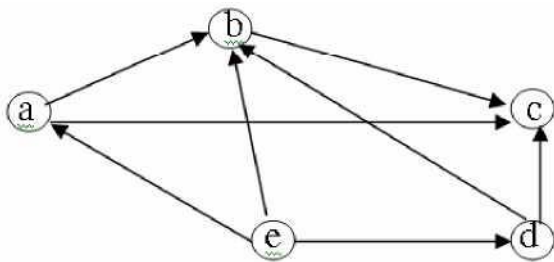
Đồ thị là một cấu trúc rời rạc gồm các đỉnh và các cạnh (vô hướng hoặc có hướng) nối các đỉnh đó. Người ta phân loại đồ thị tùy theo đặc tính và số các cạnh nối các cặp đỉnh của đồ thị.

1.1.1 Định nghĩa

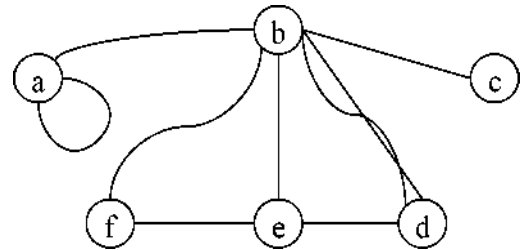
Đồ thị là một cặp $G = (V, E)$, trong đó:

- V là tập hợp các đỉnh (Vertex/Vertices),
- $E \subseteq V \times V$ là tập hợp các cạnh (Edges).

Trong khuôn khổ bài viết, chỉ xét các đồ thị hữu hạn (*finite graph*) và khác rỗng, nghĩa là các đồ thị có tập đỉnh là hữu hạn. Ví dụ:



Hình 1.1: Đồ thị có hướng (G1)



Hình 1.2: Đồ thị vô hướng (G2)

Đồ thị G1 ở trên có tập các đỉnh $V = \{a, b, c, d, e\}$ và tập các cạnh $E = \{(a, b), (a, c), (b, c), (d, b), (d, c), (e, a), (e, b), (e, d)\}$. Nếu (a, b) là một cạnh của đồ thị thì ta nói rằng đỉnh b kề với đỉnh a và cả hai đỉnh a và b kề với cạnh (a, b) .

Cạnh khuyên: một cạnh aa tương ứng với 2 đỉnh trùng nhau (a) gọi là cạnh khuyên (hình 1.2, có cạnh khuyên tại đỉnh a).

Hai cạnh song song (*parallel edges*): là 2 cạnh phân biệt cùng tương ứng với 1 cặp đỉnh (hình 1.2, có 2 cạnh song song vì cùng tương ứng với 2 đỉnh b và d).

1.1.2 Biểu diễn đồ thị

Ta có thể biểu diễn hình học cho đồ thị trên mặt phẳng như sau:

- Đỉnh: biểu diễn bằng các vòng tròn nhỏ, chứa tên của đỉnh.
- Cạnh:
 - Cạnh vô hướng: biểu diễn bằng đoạn thẳng.
 - Cạnh có hướng: biểu diễn bằng mũi tên nối hai đỉnh của đồ thị.

1.1.3 Đồ thị có hướng và đồ thị vô hướng

Cho đồ thị $G = (V, E)$, G được gọi là đồ thị:

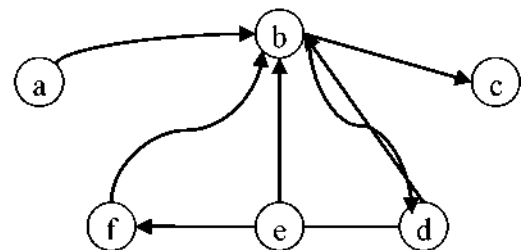
- Vô hướng: khi đồ thị chỉ chứa các cạnh vô hướng (hình 1.2).
- Có hướng: khi đồ thị chỉ chứa các cạnh có hướng (hình 1.1).

1.1.4 Đơn đồ thị (simple graph), đa đồ thị (Multigraph)

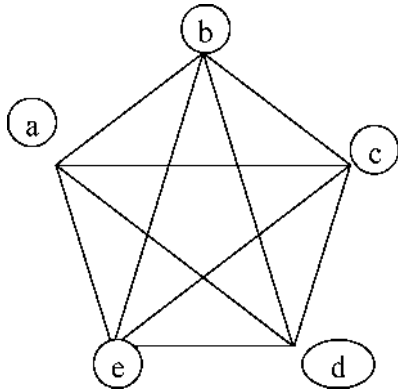
- Cho đồ thị $G = (V, E)$, G được gọi là:

- Đơn đồ thị: mà mỗi cặp đỉnh được nối với nhau bởi không quá một cạnh (thường được gọi tắt là đồ thị - hình 1.1).
- Đa đồ thị: khi đồ thị có những cặp đỉnh được nối với nhau nhiều hơn một cạnh thì được gọi là đa đồ thị (hình 1.2).
- Từ 1.3 và 1.4, ta có thể có các dạng đồ thị sau:

- Đơn đồ thị vô hướng (hình 1.3)
- Đơn đồ thị có hướng (hình 1.1)
- Đa đồ thị vô hướng (hình 1.2)
- Đa đồ thị có hướng (hình 1.4)



Hình 1.4 Đa đồ thị có hướng



Hình 1.3 Đơn đồ thị vô hướng

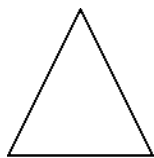
1.1.5 Một số dạng đơn đồ thị đặc biệt

1.1.5.1 Đồ thị đầy đủ (complete graph)

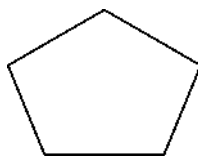
Đồ thị $G = (V, E)$ được gọi là đồ thị đầy đủ khi mọi cặp đỉnh đều kề nhau, hay nói cách khác là đồ thị mà mọi cặp đỉnh đều có cạnh nối với nhau (hình 1.3).

1.1.5.2 Đồ thị vòng

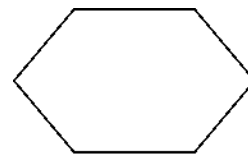
Đồ thị $G = (V, E)$ được gọi là đồ thị vòng khi số lượng đỉnh của đồ thị ≥ 3 , bậc của tất cả các đỉnh đều bằng 2 và các cạnh nối với nhau thành 1 vòng khép kín (hình 1.5). Ký hiệu: C_n



C_3



C_5

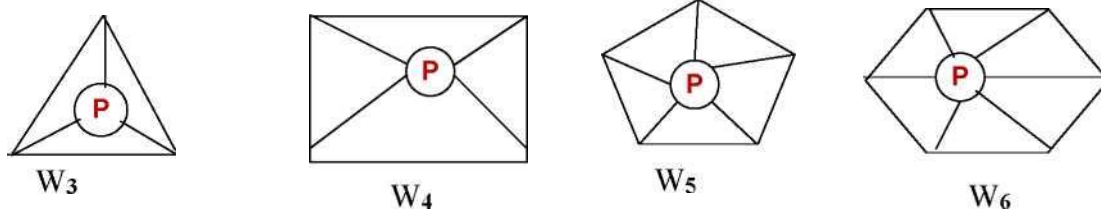


C_6

Hình 1.5. Đồ thị vòng

1.1.5.3 Đồ thị bánh xe

Đồ thị $G = (V, E)$ được gọi là đồ thị bánh xe khi đó là đồ thị vòng và có bổ sung thêm một đỉnh mới P , đỉnh này được nối với tất cả các đỉnh còn lại (hình 1.6). Ký hiệu W_n .



Hình 1.6. Đồ thị bánh xe

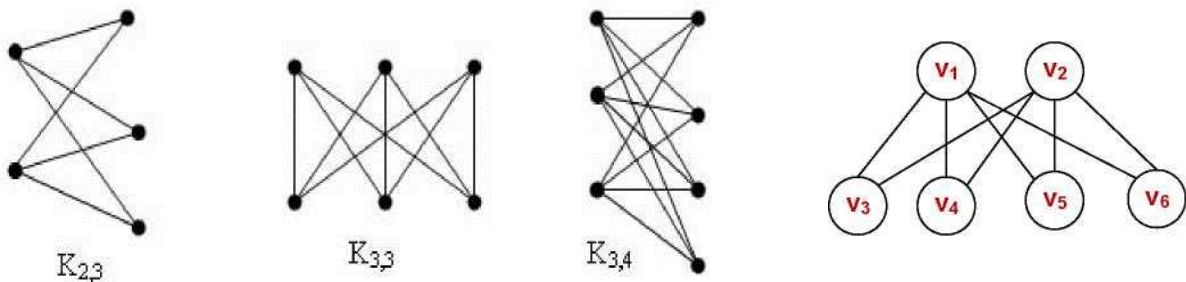
1.1.5.4 Đồ thị lập phương

Đơn đồ thị $2n$ đỉnh, tương ứng với $2n$ xâu nhị phân độ dài n và hai đỉnh kề nhau khi và chỉ khi 2 xâu nhị phân tương ứng với hai đỉnh này chỉ khác nhau đúng một bit được gọi là đồ thị lập phương, ký hiệu là Q_n . Như vậy, mỗi đỉnh của Q_n có bậc là n và số cạnh của Q_n là $n \cdot 2^{n-1}$ (từ công thức $2|E| = \sum_{v \in V} \deg(v)$)

1.1.5.5 Đồ thị phân đôi

Đơn đồ thị $G=(V,E)$ sao cho $V=V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$, $V_1 \neq \emptyset$, $V_2 \neq \emptyset$ và mỗi cạnh của G được nối một đỉnh trong V_1 và một đỉnh trong V_2 được gọi là đồ thị phân đôi.

Nếu đồ thị phân đôi $G=(V_1 \cup V_2, E)$ sao cho với mọi $v_1 \in V_1$, $v_2 \in V_2$, $(v_1, v_2) \in E$ thì G được gọi là đồ thị phân đôi đầy đủ. Nếu $|V_1|=m$, $|V_2|=n$ thì đồ thị phân đôi đầy đủ G ký hiệu là $K_{m,n}$. Như vậy $K_{m,n}$ có $m \cdot n$ cạnh, các đỉnh của V_1 có bậc n và các đỉnh của V_2 có bậc m .



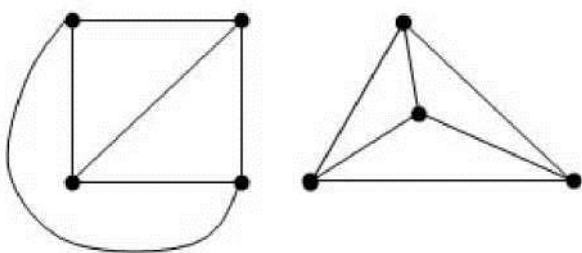
Hình 1.8. Đồ thị phân đôi

$K_{2,4}$

1.1.5.6 Đồ thị phẳng

Đồ thị được gọi là đồ thị phẳng nếu ta có thể vẽ nó trên mặt phẳng sao cho các cạnh của nó không cắt nhau ngoài ở đỉnh. Cách vẽ như vậy sẽ được gọi là biểu diễn phẳng của đồ thị.

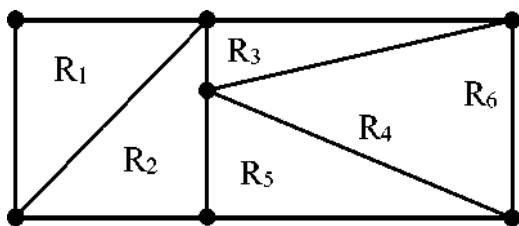
Ví dụ đồ thị K_4 là phẳng, vì có thể vẽ nó trên mặt phẳng sao cho các cạnh của nó không cắt nhau ngoài ở đỉnh (hình 1.9).



Hình 1.9. Đồ thị K_4 là đồ thị phẳng

Công thức Euler:

Biểu diễn phẳng của đồ thị sẽ chia mặt phẳng ra thành các miền. Ví dụ, biểu diễn phẳng của đồ thị cho trong hình 1.10 chia mặt phẳng ra thành 6 miền R_1, R_2, \dots, R_6 .



Hình 1.10. Các miền tương ứng với biểu diễn phẳng của đồ thị

Euler đã chứng minh được mối liên hệ giữa số miền, số đỉnh của đồ thị và số cạnh của đồ thị phẳng qua định lý sau:

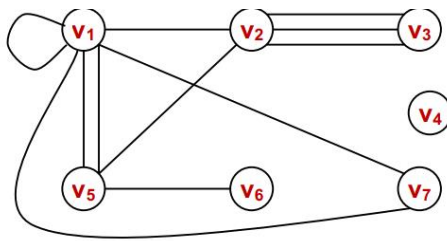
Định lý về Công thức Euler. Giả sử G là đồ thị phẳng liên thông với n đỉnh, m cạnh. Gọi r là số miền của mặt phẳng bị chia bởi biểu diễn phẳng của G . Khi đó

$$r = m - n + 2$$

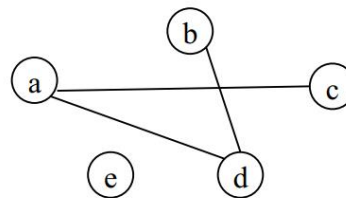
Ví dụ: Cho G là đồ thị phẳng liên thông với 20 đỉnh, mỗi đỉnh đều có bậc là 3. Hỏi mặt phẳng bị chia làm bao nhiêu phần bởi biểu diễn phẳng của đồ thị G ?
 Giải: Do mỗi đỉnh của đồ thị đều có bậc là 3, nên tổng bậc của các đỉnh là $3 \times 20 = 60$. Từ đó suy ra số cạnh của đồ thị $m = 60/2 = 30$. Vì vậy, theo công thức Euler, số miền cần tìm là $r = 30 - 20 + 2 = 12$.

1.1.6 Bậc của một đỉnh (degree)

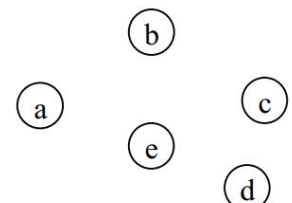
- Xét một đỉnh v bất kỳ của đồ thị $G = (V, E)$, số lượng cạnh nối tới đỉnh v gọi là bậc của đỉnh v . Cứ mỗi cạnh khuyên tại đỉnh v được tính là 2.
- Ký hiệu $d(v)$.
- Đỉnh cô lập (*isolated vertex*): là đỉnh có bậc = 0 (đỉnh d trong hình 1.13; đỉnh e trong hình 1.14; đỉnh a, b, c, d, e trong hình 1.15).
- Đỉnh treo (*pendant vertex*): là đỉnh có bậc = 1 (đỉnh f trong hình 1.13; đỉnh b, c trong hình 1.14).
- Đồ thị rỗng: là đồ thị có tất cả các đỉnh đều là đỉnh cô lập (hình 1.15).



Hình 1.13



Hình 1.14



Hình 1.15 Đồ thị rỗng

1.1.1.1 Đồ thị vô hướng

- Định lý 1: với mọi đồ thị $G = (V, E)$, số cạnh E của G được xác định bởi công thức:

$$\sum_{v \in V} d(v) = 2|E|$$

- Hệ luận 1: Trong đồ thị vô hướng, số lượng đỉnh bậc lẻ là một số chẵn.

1.1.1.2 Đồ thị có hướng

- Định lý 2: Cho đồ thị có hướng $G = (V, E)$, ta gọi bậc trong của một đỉnh là số cung có hướng đi ra khỏi đỉnh (ký hiệu $d^+(v)$); tương tự, ta gọi bậc ngoài của một đỉnh là số cung có hướng đi vào đỉnh (ký hiệu $d^-(v)$). Ta có: tổng bậc ngoài của tất cả các đỉnh bằng với tổng bậc trong của tất cả các đỉnh

$$\sum_{v \in V} d^+(v) = \sum_{v \in V} d^-(v)$$

1.2 Đường đi, chu trình và liên thông

Giả sử $G = (V, E)$ là một đồ thị.

1.2.1 Đường đi

Đường đi trong đồ thị là một dãy các đỉnh:

$$\langle x_1, x_2, \dots, x_i, x_{i+1}, \dots, x_{k-1}, x_k \rangle$$

sao cho, mỗi đỉnh trong dãy (không kể đỉnh đầu tiên) kề với đỉnh trước nó bằng một cạnh

nào đó, nghĩa là: $\forall i = 2, 3, \dots, k-1, k : (x_{i-1}, x_i) \in E$.

Vậy:

- Đường đi này đi từ đỉnh đầu x_1 đến đỉnh cuối x_k .
- Số cạnh của đường đi được gọi là *độ dài* của đường đi đó.
- Đường đi đơn là đường đi mà các đỉnh trên nó khác nhau từng đôi.

1.2.2 Chu trình

Chu trình là một đường đi khép kín (tức là đỉnh cuối của đường trùng với đỉnh đầu của đường).

Ta thường ký hiệu chu trình là:

$$[x_1, x_2, \dots, x_i, x_{i+1}, \dots, x_{k-1}, x_k], \text{ trong đó } x_1 = x_k$$

Để cho gọn, trong ký hiệu của chu trình thường không viết đỉnh cuối:

$$[x_1, x_2, \dots, x_i, x_{i+1}, \dots, x_{k-1}]$$

- Khi nói đến một chu trình, ta cũng không cần xác định đỉnh đầu và đỉnh cuối của chu trình

đó.

- Chu trình được gọi là *chu trình đơn* nếu các đỉnh trên nó khác nhau từng đôi.

- *Một số tính chất về chu trình trên đồ thị vô hướng:*

• Đồ thị vô hướng với n đỉnh ($n \geq 3$), không có đỉnh nút và bậc của mỗi đỉnh đều không nhỏ

hơn 2, luôn có chu trình đơn.

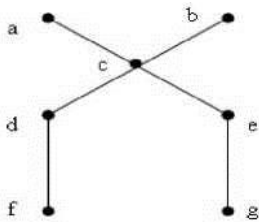
• Đồ thị vô hướng với n đỉnh ($n \geq 4$) và bậc của mỗi đỉnh đều không nhỏ hơn 3, luôn có chu

trình đơn độ dài chẵn.

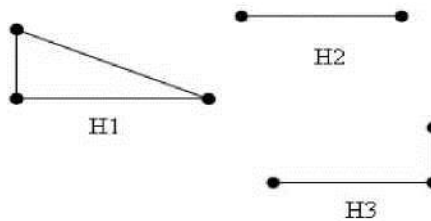
1.2.3 Liên thông

1.2.3.1 Đồ thị vô hướng

- *Đồ thị liên thông*: Đồ thị vô hướng $G = (V, E)$ được gọi là liên thông nếu luôn tìm được đường đi giữa hai đỉnh bất kỳ trong đồ thị (hình 1.23).
- *Thành phần liên thông*: Trong trường hợp đồ thị là không liên thông, nó sẽ rã ra thành một số đồ thị con liên thông không có đỉnh chung. Những đồ thị con liên thông như vậy ta sẽ gọi là các *thành phần liên thông* của đồ thị (hình 1.24).
- *Ứng dụng*: xác định hai máy tính bất kỳ trong mạng có thể trao đổi thông tin được với nhau khi và chỉ khi đồ thị tương ứng với mạng này là đồ thị liên thông.



Hình 1.23. Đồ thị liên thông



Hình 1.24. Đồ thị không liên thông gồm 3 thành phần liên thông (H_1, H_2, H_3)

- *Mệnh đề*: Mọi đơn đồ thị n đỉnh ($n \geq 2$) có tổng bậc của hai đỉnh tùy ý không nhỏ hơn n đều là đồ thị liên thông.

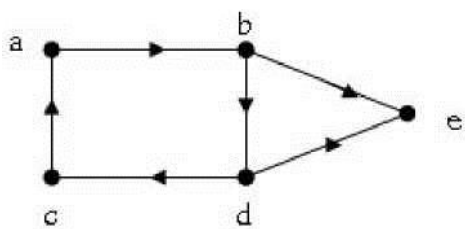
Hệ quả: Đơn đồ thị mà bậc của mỗi đỉnh của nó không nhỏ hơn một nửa số đỉnh là đồ thị liên thông.

- *Mệnh đề*: Nếu một đồ thị có đúng hai đỉnh bậc lẻ thì hai đỉnh này phải liên thông, tức là có một đường đi nối chúng.

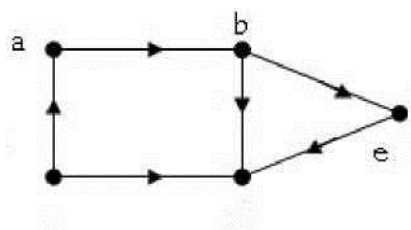
1.2.3.2 Đồ thị có hướng

- *Liên thông mạnh*: Đồ thị có hướng $G = (V, A)$ được gọi là liên thông mạnh nếu luôn tìm được đường đi giữa hai đỉnh bất kỳ của nó.
- *Liên thông yếu*: Đồ thị có hướng $G = (V, A)$ được gọi là liên thông

yếu nếu đồ thị vô hướng tương ứng với nó là vô hướng liên thông.



Hình 1.25. Đồ thị liên thông mạnh



Hình 1.26. Đồ thị liên thông yếu

CHƯƠNG 2. BÀI TOÁN ĐƯỜNG ĐI NGẮN NHẤT

2.1 Giới thiệu

Trong các ứng dụng thực tế, chẳng hạn trong mạng lưới giao thông đường bộ, đường thủy hoặc đường không. Người ta không chỉ quan tâm đến việc tìm đường đi giữa hai địa điểm mà còn phải lựa chọn một hành trình tiết kiệm nhất (theo tiêu chuẩn không gian, thời gian hay chi phí). Khi đó phát sinh yêu cầu tìm đường đi ngắn nhất giữa hai đỉnh của đồ thị. Bài toán đó phát biểu dưới dạng tổng quát như sau: Cho đồ thị có trọng số $G = (V, E)$, hãy tìm một đường đi ngắn nhất từ đỉnh xuất phát $s \in V$ đến đỉnh đích $f \in V$. Độ dài của đường đi này ta sẽ ký hiệu là $d[s, f]$ và gọi là *khoảng cách* (distance) từ s đến f . Nếu như không tồn tại đường đi từ s tới f thì ta sẽ đặt khoảng cách đó $= +\infty$.

Nếu như đồ thị có chu trình âm (chu trình với độ dài âm) thì khoảng cách giữa một số cặp đỉnh nào đó có thể không xác định, bởi vì bằng cách đi vòng theo chu trình này một số lần đủ lớn, ta có thể chỉ ra đường đi giữa hai đỉnh nào đó trong chu trình này nhỏ hơn bất kỳ một số cho trước nào. Trong trường hợp như vậy, có thể đặt vấn đề tìm **đường đi cơ bản** (đường đi không có đỉnh lặp lại) ngắn nhất. Vấn đề đó là một vấn đề hết sức phức tạp mà ta sẽ không bàn tới ở đây.

Nếu như đồ thị không có chu trình âm thì ta có thể chứng minh được rằng một trong những đường đi ngắn nhất là đường đi cơ bản. Và nếu như biết được khoảng cách từ s tới tất cả những đỉnh khác thì đường đi ngắn nhất từ s tới f có thể tìm được một cách dễ dàng qua thuật toán sau:

Gọi $c[u, v]$ là trọng số của cạnh $[u, v]$. Qui ước $c[v, v] = 0$ với mọi $v \in V$ và $c[u, v] = +\infty$ nếu như $(u, v) \notin E$. Đặt $d[s, v]$ là khoảng cách từ s tới v . Để tìm đường đi từ s tới f , ta có thể nhận thấy rằng luôn tồn tại đỉnh $f_1 \neq f$ sao cho:

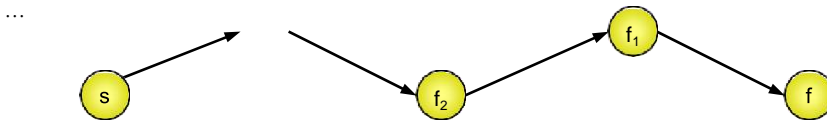
$$d[s, f] = d[s, f_1] + c[f_1, f]$$

(Độ dài đường đi ngắn nhất $s \rightarrow f = \text{Độ dài đường đi ngắn nhất } s \rightarrow f_1 + \text{Chi phí đi từ } f_1 \rightarrow f$)

Đỉnh f_1 đó là đỉnh liền trước f trong đường đi ngắn nhất từ s tới f . Nếu $f_1 \equiv s$ thì đường đi ngắn nhất là đường đi trực tiếp theo cung (s, f) . Nếu không thì vấn đề trở thành tìm đường đi ngắn nhất từ s tới f_1 . Và ta lại tìm được một đỉnh f_2 khác f và f_1 để:

$$d[s, f_1] = d[s, f_2] + c[f_2, f_1]$$

Cứ tiếp tục như vậy, sau một số hữu hạn bước, ta suy ra rằng dãy f, f_1, f_2, \dots không chứa đỉnh lặp lại và kết thúc ở s . Lật ngược thứ tự dãy cho ta đường đi ngắn nhất từ s tới f .



Tuy nhiên, người ta thường không sử dụng phương pháp này mà sẽ kết hợp lưu vết đường đi ngay trong quá trình tìm kiếm.

Dưới đây ta sẽ xét một số thuật toán tìm đường đi ngắn nhất từ đỉnh s tới đỉnh f trên đơn đồ thị có hướng $G = (V, E)$ có n đỉnh và m cung. Trong trường hợp đơn đồ thị vô hướng với trọng số không âm, bài toán tìm đường đi ngắn nhất có thể dẫn về bài toán trên đồ thị có hướng bằng cách thay mỗi cạnh của nó bằng hai cung có hướng ngược chiều nhau. Bạn có thể đưa vào một số sửa đổi nhỏ trong thủ tục nhập liệu để giải quyết bài toán trong trường hợp đa đồ thị

2.2 Đồ thị có trọng số

Đồ thị mà mỗi cạnh của nó được gán cho tương ứng với một số (nguyên hoặc thực) được gọi là đồ thị có trọng số. Số gán cho mỗi cạnh của đồ thị được gọi là trọng số của cạnh. Tương tự như đồ thị không trọng số, có nhiều cách biểu diễn đồ thị có trọng số trong máy tính. Đối với đơn đồ thị thì cách dễ dùng nhất là sử dụng ma trận trọng số:

Giả sử đồ thị $G = (V, E)$ có n đỉnh. Ta sẽ dựng ma trận vuông C kích thước $n \times n$. Ở đây:

- Nếu $(u, v) \in E$ thì $C[u, v] =$ trọng số của cạnh (u, v)
- Nếu $(u, v) \notin E$ thì tùy theo trường hợp cụ thể, $C[u, v]$ được gán một giá trị nào đó để có thể nhận biết được (u, v) không phải là cạnh (Chẳng hạn có thể gán bằng $+\infty$, hay bằng 0, bằng $-\infty$, v.v...)
- Quy ước $c[v, v] = 0$ với mọi đỉnh v .

Đường đi, chu trình trong đồ thị có trọng số cũng được định nghĩa giống như trong trường hợp không trọng số, chỉ có khác là độ dài đường đi không phải tính bằng số cạnh đi qua, mà được tính bằng tổng trọng số của các cạnh đi qua.

2.3 Bài toán đường đi ngắn nhất

2.3.1 Đồ thị không có chu trình âm - thuật toán Ford Bellman

Thuật toán Ford-Bellman có thể phát biểu rất đơn giản:

Với đỉnh xuất phát s . Gọi $d[v]$ là khoảng cách từ s tới v với các giá trị khởi tạo là:

$$d[s] := 0$$

$$d[v] := +\infty \text{ nếu } v \neq s$$

Sau đó ta tối ưu hoá dần các $d[v]$ như sau: Xét mọi cặp đỉnh u, v của đồ thị, nếu có một cặp đỉnh u, v mà $d[v] > d[u] + c[u, v]$ thì ta đặt lại $d[v] := d[u] + c[u, v]$. Tức là nếu độ dài đường đi từ s tới v lại **lớn hơn** tổng độ dài đường đi từ s tới u cộng với chi phí đi từ u tới v thì ta sẽ huỷ bỏ đường đi từ s tới v đang có và coi đường đi từ s tới v chính là đường đi từ s tới u sau đó đi tiếp từ u tới v . Chú ý rằng ta đặt $c[u, v] = +\infty$ nếu (u, v) không là cung. Thuật toán sẽ kết thúc khi không thể tối ưu thêm bất kỳ một nhãn $d[v]$ nào nữa.

```

for ( $\forall v \in V$ ) do  $d[v] := +\infty$ ;
 $d[s] := 0$ ;
repeat
  Stop := True;
  for ( $\forall u \in V$ ) do
    for ( $\forall v \in V: (u, v) \in E$ ) do
      if  $d[v] > d[u] + c[u, v]$  then
        begin
           $d[v] := d[u] + c[u, v]$ ;
          Stop := False;
        end;
until Stop;

```

Tính đúng đắn của thuật toán:

Tại bước khởi tạo thì mỗi $d[v]$ chính là độ dài ngắn nhất của đường đi từ s tới v qua không quá 0 cạnh.

Giả sử khi bắt đầu bước lặp thứ i ($i \geq 1$), $d[v]$ đã bằng độ dài đường đi ngắn nhất từ s tới v qua không quá $i - 1$ cạnh. Bởi đường đi từ s tới v qua không quá i cạnh sẽ phải thành lập bằng cách: lấy một đường đi từ s tới một đỉnh u nào đó qua không quá $i - 1$ cạnh, rồi đi tiếp tới v bằng cung (u, v) , nên độ dài đường đi ngắn nhất từ s tới v qua không quá i cạnh sẽ được tính bằng giá trị nhỏ nhất trong các giá trị (Nguyên lý tối ưu Bellman):

Độ dài đường đi ngắn nhất từ s tới v qua không quá $i - 1$ cạnh

Độ dài đường đi ngắn nhất từ s tới u qua không quá $i - 1$ cạnh cộng với trọng số cạnh (u, v) ($\forall u$)

Vì vậy, sau bước lặp tối ưu các $d[v]$ bằng công thức

$$d[v]_{\text{bước } i} = \min(d[v]_{\text{bước } i-1}, d[u]_{\text{bước } i-1} + c[u, v]) \quad (\forall u)$$

thì các $d[v]$ sẽ bằng độ dài đường đi ngắn nhất từ s tới v qua không quá i cạnh.

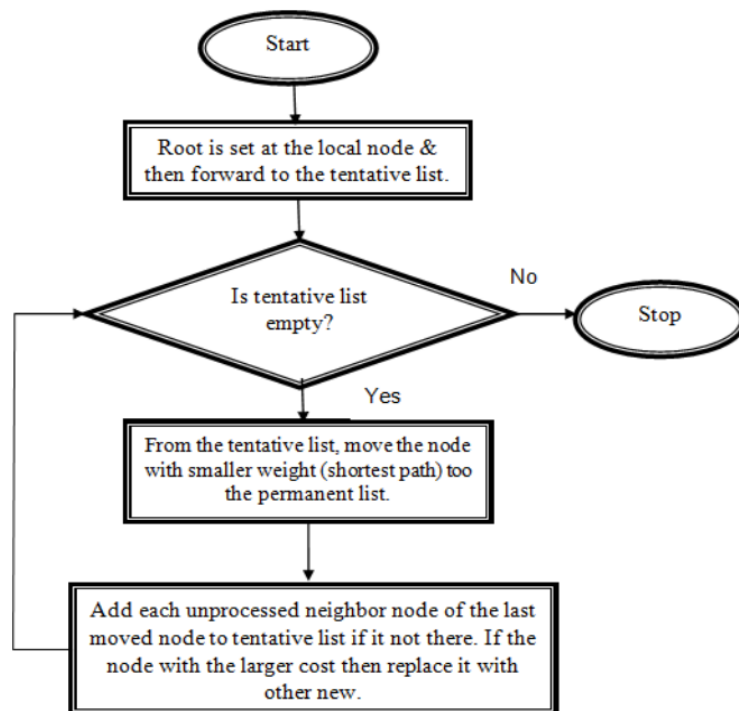
Sau bước lặp tối ưu thứ $n - 1$, ta có $d[v] =$ độ dài đường đi ngắn nhất từ s tới v qua không quá $n - 1$ cạnh. Vì đồ thị không có chu trình âm nên sẽ có một đường đi ngắn nhất từ s tới v là đường đi cơ bản (qua không quá $n - 1$ cạnh).

Tức là $d[v]$ sẽ là độ dài đường đi ngắn nhất từ s tới v . Vậy thì số bước lặp tối ưu hoá sẽ không quá $n - 1$ bước.

2.3.2 Trường hợp trọng số trên các cung không âm - thuật toán dijkstra

Trong trường hợp trọng số trên các cung không âm, thuật toán do Dijkstra đề xuất dưới đây hoạt động hiệu quả hơn nhiều so với thuật toán Ford-Bellman. Ta hãy xem trong trường hợp này, thuật toán Ford-Bellman thiếu hiệu quả ở chỗ nào:

Với đỉnh $v \in V$, gọi $d[v]$ là độ dài đường đi ngắn nhất từ s tới v . Thuật toán Ford-Bellman khởi gán $d[s] = 0$ và $d[v] = +\infty$ với $\forall v \neq s$, sau đó tối ưu hoá dần các nhãn $d[v]$ bằng cách sửa nhãn theo công thức: $d[v] := \min(d[v], d[u] + c[u, v])$ với $\forall u, v \in V$. Như vậy nếu như ta dùng đỉnh u sửa nhãn đỉnh v , sau đó nếu ta lại tối ưu được $d[u]$ thêm nữa thì ta cũng phải sửa lại nhãn $d[v]$ dẫn tới việc $d[v]$ có thể phải chỉnh đi chỉnh lại rất nhiều lần. Vậy nên chẳng, tại mỗi bước *không phải ta xét mọi cặp đỉnh (u, v) để dùng đỉnh u sửa nhãn đỉnh v mà sẽ chọn đỉnh u là đỉnh mà không thể tối ưu nhãn $d[u]$ thêm được nữa.*



Flow diagram of Dijkstra algorithm.

Thuật toán Dijkstra (E.Dijkstra - 1959) có thể mô tả như sau:

Bước 1: Khởi tạo

Với đỉnh $v \in V$, gọi nhãn $d[v]$ là độ dài đường đi ngắn nhất từ s tới v . Ban đầu $d[v]$ được khởi gán như trong thuật toán Ford-Bellman ($d[s] = 0$ và $d[v] = \infty$ với $\forall v \neq s$). Nhãn của mỗi đỉnh có hai trạng thái tự do hay cố định, nhãn tự do có nghĩa là có thể còn tối ưu hơn được nữa và nhãn cố định tức là $d[v]$ đã bằng độ dài đường đi ngắn nhất từ s tới v nên không thể tối ưu thêm. Để làm điều này ta có thể sử dụng kỹ thuật đánh dấu: $\text{Free}[v] = \text{TRUE}$ hay FALSE tùy theo $d[v]$ tự do hay cố định. Ban đầu các nhãn đều tự do.

Bước 2: Lặp

Bước lặp gồm có hai thao tác:

1. Cố định nhãn: Chọn trong các đỉnh có nhãn tự do, lấy ra đỉnh u là đỉnh có $d[u]$ nhỏ nhất, và cố định nhãn đỉnh u .
2. Sửa nhãn: Dùng đỉnh u , xét tất cả những đỉnh v và sửa lại các $d[v]$ theo công thức: $d[v] := \min(d[v], d[u] + c[u, v])$

Bước lặp sẽ kết thúc khi mà đỉnh đích f được cố định nhãn (tìm được đường đi ngắn nhất từ s tới f); hoặc tại thao tác cố định nhãn, tất cả các đỉnh tự do đều có nhãn là $+\infty$ (không tồn tại đường đi). Có thể đặt câu hỏi, ở thao tác 1, tại sao đỉnh u như vậy được cố định nhãn, giả sử $d[u]$ còn có thể tối ưu thêm được nữa thì tất phải có một đỉnh t mang nhãn tự do sao cho $d[u] > d[t] + c[t, u]$. Do trọng số $c[t, u]$ không âm nên $d[u] > d[t]$, trái với cách chọn $d[u]$ là nhỏ nhất. Tất nhiên trong lần lặp đầu tiên thì s là đỉnh được cố định nhãn do $d[s] = 0$.

Bước 3: Kết hợp với việc lưu vết đường đi trên từng bước sửa nhãn, thông báo đường đi ngắn nhất tìm được hoặc cho biết không tồn tại đường đi ($d[f] = +\infty$).

```
for ( $\forall v \in V$ ) do  $d[v] := +\infty$ ;  $d[s] := 0$ ;
```

```
repeat
```

```
 $u := \arg \min(d[v] \mid \forall v \in V)$ ; {Lấy  $u$  là đỉnh có nhãn  $d[u]$  nhỏ nhất}
```

if ($u = f$) **or** ($d[u] = +\infty$) **then Break**; {Hoặc tìm ra đường đi ngắn nhất từ s tới f , hoặc kết luận không có đường}

for ($\forall v \in V: (u, v) \in E$) **do** {Dùng u tối ưu nhân những đỉnh v kề với u }
 $d[v] := \min(d[v], d[u] + c[u, v]);$ **until False**;

2.3.3 Đường đi ngắn nhất giữa mọi cặp đỉnh - thuật toán Floyd-Warshall

Cho đơn đồ thị có hướng, có trọng số $G = (V, E)$ với n đỉnh và m cạnh. Bài toán đặt ra là hãy tính tất cả các $d(u, v)$ là khoảng cách từ u tới v . Rõ ràng là ta có thể áp dụng thuật toán tìm đường đi ngắn nhất xuất phát từ một đỉnh với n khả năng chọn đỉnh xuất phát. Nhưng ta có cách làm gọn hơn nhiều, cách làm này rất giống với thuật toán Warshall mà ta đã biết: Từ ma trận trọng số c , thuật toán Floyd tính lại các $c[u, v]$ thành độ dài đường đi ngắn nhất từ u tới v :

Với mọi đỉnh k của đồ thị được xét theo thứ tự từ 1 tới n , xét mọi cặp đỉnh u, v . Cực tiểu hoá $c[u, v]$ theo công thức:

$$c[u, v] := \min(c[u, v], c[u, k] + c[k, v])$$

Tức là nếu như đường đi từ u tới v đang có lại dài hơn đường đi từ u tới k cộng với đường đi từ k tới v thì ta huỷ bỏ đường đi từ u tới v hiện thời và coi đường đi từ u tới v sẽ là nối của hai đường đi từ u tới k rồi từ k tới v (Chú ý rằng ta còn có việc lưu lại vết):

```
for k := 1
  to n do
    for u := 1
      to n do
        for v := 1 to n do
           $c[u, v] := \min(c[u, v], c[u, k] + c[k, v]);$ 
```

Tính đúng của thuật toán:

Gọi $c^k[u, v]$ là độ dài đường đi ngắn nhất từ u tới v mà chỉ đi qua các đỉnh trung gian thuộc tập $\{1, 2, \dots, k\}$. Rõ ràng khi $k = 0$ thì $c^0[u, v] = c[u, v]$ (đường đi ngắn nhất là đường đi trực tiếp).

Giả sử ta đã tính được các $c^{k-1}[u, v]$ thì $c^k[u, v]$ sẽ được xây dựng như sau:

Nếu đường đi ngắn nhất từ u tới v mà chỉ qua các đỉnh trung gian thuộc tập $\{1, 2, \dots, k\}$ lại:

- ❖ Không đi qua đỉnh k thì tức là chỉ qua các đỉnh trung gian thuộc tập $\{1, 2, \dots, k - 1\}$ thì

$$c^k[u, v] = c^{k-1}[u, v]$$

- ❖ Có đi qua đỉnh k thì đường đi đó sẽ là nối của một đường đi từ u tới k và một đường đi từ k tới v , hai đường đi này chỉ đi qua các đỉnh trung gian thuộc tập $\{1, 2, \dots, k - 1\}$.

$$c^k[u, v] = c^{k-1}[u, k] + c^{k-1}[k, v]$$

Vì ta muốn $c^k[u, v]$ là cực tiểu nên suy ra:

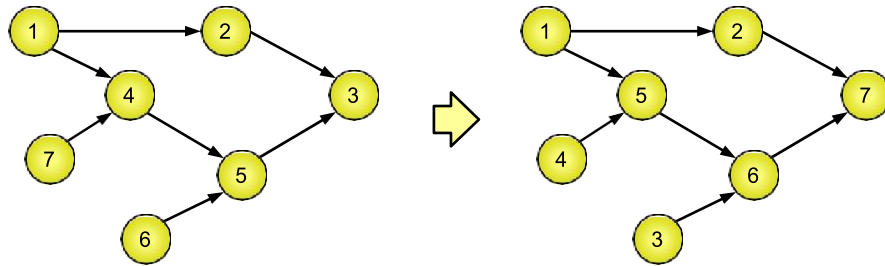
$$c^k[u, v] = \min(c^{k-1}[u, v], c^{k-1}[u, k] + c^{k-1}[k, v]).$$

Và cuối cùng, ta quan tâm tới $c^n[u, v]$: Độ dài đường đi ngắn nhất từ u tới v mà chỉ đi qua các đỉnh trung gian thuộc tập $\{1, 2, \dots, n\}$.

Khi cài đặt, thì ta sẽ không có các khái niệm $c^k[u, v]$ mà sẽ thao tác trực tiếp trên các trọng số $c[u, v]$. $c[u, v]$ tại bước tối ưu thứ k sẽ được tính toán để tối ưu qua các giá trị $c[u, v]$; $c[u, k]$ và $c[k, v]$ tại bước thứ $k - 1$. Tính chính xác của cách cài đặt dưới dạng ba vòng lặp for lồng nhau như trên có thể thấy được do sự tối ưu bắc cầu chỉ làm tăng tốc độ tối ưu các $c[u, v]$ trong mỗi bước.

2.3.4 Trường hợp đồ thị không có chu trình - sắp xếp Tô pô

Xét trường hợp đồ thị có hướng, không có chu trình (Directed Acyclic Graph - DAG), ta có một thuật toán hiệu quả dựa trên kỹ thuật sắp xếp Tô pô (Topological Sorting), cơ sở của thuật toán dựa vào định lý: Nếu $G = (V, E)$ là một DAG thì các đỉnh của nó có thể đánh số sao cho mỗi cung của G chỉ nối từ đỉnh có chỉ số nhỏ hơn đến đỉnh có chỉ số lớn hơn.



Hình: Phép đánh lại chỉ số theo thứ tự tôpô

Để đánh số lại các đỉnh theo điều kiện trên, ta có thể dùng thuật toán tìm kiếm theo chiều sâu và đánh số ngược lại với thứ tự duyệt xong.

Việc kiểm tra đồ thị không được có chu trình cũng rất đơn giản (có thể tham khảo các kỹ thuật trình bày trong thuật toán Tarjan)

Nếu các đỉnh được đánh số sao cho mỗi cung phải nối từ một đỉnh tới một đỉnh khác mang chỉ số lớn hơn thì thuật toán tìm đường đi ngắn nhất có thể mô tả rất đơn giản:

Gọi $d[v]$ là độ dài đường đi ngắn nhất từ s tới v .

Khởi tạo $d[s] = 0$ và $d[v] = +\infty$ với $\forall v \neq s$.

Ta sẽ tính các $d[v]$ như sau:

```
for u := 1 to
  n - 1 do
    for v := u
      + 1 to n do
         $d[v] := \min(d[v], d[u] + c[u, v]);$ 
```

(Giả thiết rằng $c[u, v] = +\infty$ nếu như (u, v) không là cung).

Tức là dùng đỉnh u , tối ưu nhận $d[v]$ của những đỉnh v nối từ u , với u được xét lần lượt từ 1 tới $n - 1$. Có thể làm tốt hơn nữa bằng cách chỉ cần cho u chạy từ đỉnh xuất phát tới đỉnh kết thúc. Bởi hễ u chạy tới đâu thì nhận $d[u]$ là không thể cực tiểu hoá thêm nữa.

2.4 Ứng dụng của bài toán tìm đường đi ngắn nhất

Tương ứng với các vấn đề trong thực tế đã có rất nhiều các ứng dụng của bài toán tìm đường đi ngắn nhất trong từng mạng lưới đã nêu trên. Chúng ta có thể liệt kê các ứng dụng phổ biến nhất hiện nay:

2.4.1 GPS

GPS (Global Positioning System): hệ thống định vị toàn cầu - là hệ thống xác định vị trí dựa trên vị trí của các vệ tinh nhân tạo. Trong cùng một thời điểm, ở một vị trí trên mặt đất nếu xác định được khoảng cách đến ba vệ tinh (tối thiểu) thì sẽ tính được tọa độ của vị trí đó. GPS được thiết kế và quản lý bởi Bộ Quốc phòng Mỹ, nhưng chính phủ Hoa Kỳ cho phép mọi người sử dụng nó miễn phí, bất kể quốc tịch từ năm 1980, GPS hoạt động trong mọi điều kiện thời tiết, mọi nơi trên Trái Đất, 24 giờ một ngày.

GPS là hệ dẫn đường dựa trên một mạng lưới 24 quả vệ tinh được đặt trên quỹ đạo không gian, hoạt động dựa trên các trạm phát tín hiệu vô tuyến điện. Được biết nhiều nhất là các hệ thống có tên gọi LORAN - hoạt động ở giải tần 90-100 kHz chủ yếu dùng cho hàng hải, hay TACAN - dùng cho quân đội Mỹ và biến thể với độ chính xác thấp VOR/DME - VHF dùng cho hàng không dân dụng.

2.4.2 Phân tích và xử lý ảnh

Phân tích vùng ảnh là một phần không thể thiếu của các nghiệp vụ cần xử lý ảnh như những công việc xử lý ảnh tai nạn, phân tích hình ảnh qua thiết bị y tế, và chỉnh sửa ảnh.

Vấn đề của việc phân vùng ảnh là việc gom các nhóm pixel gần nhau có tính chất giống nhau. Quá trình gom nhóm thường dựa trên tính toán đường đi ngắn nhất. Bề mặt của ảnh là những chiều liên tục. Đường đi ngắn nhất được tính toán với các phương pháp so sánh, một biến thể của thuật toán toán Dijkstra.

Các thuật toán dựa trên đồ thị xem các hình ảnh như là một đồ thị, trong đó mỗi pixel là một nút, kết nối bởi một cạnh có 4 (hoặc 8 hoặc nhiều hơn) pixel lân cận. Một phần quan trọng là gán trọng số thích hợp để các cạnh, dựa trên hình ảnh "tiềm năng" của hai điểm ảnh và khoảng cách Euclide giữa các pixel. Người sử dụng cung cấp điểm đầu cuối; các thuật toán tính toán kết quả của truy vấn con đường ngắn nhất tương ứng point-to-point và các đường dẫn kết quả được hiểu như là một phần của các đối tượng ranh giới. Nói cách khác, đưa ra hai điểm đầu cuối của một đường viền, các thuật toán xác định các đường viền tối đa khả năng kết nối chúng. Các trọng số cạnh là xác suất và bằng cách lấy logarit tiêu cực, các thuật toán chỉ có thể tổng hợp các trọng trên con đường để lấy đường viền tối ưu. Đối với các tính toán ranh giới trong 3 chiều, các thuật toán thông minh kéo phải được thích nghi. Một ranh giới có thể được tìm thấy bằng cách kết hợp nhiều con đường ngắn nhất cho đến khi một mặt kín thu được.

Trong tầm nhìn máy tính, thuật toán đường đi ngắn nhất trên đồ thị có trọng số đã tìm thấy rất nhiều ứng dụng khác với phân khúc như phát hiện trung tâm, xạ trị, lưới morphing, video tổng hợp, và việc tìm kiếm những con đường và những con đường mòn trên hình ảnh vệ tinh. Các ứng dụng nổi tiếng: Dò tìm theo hình ảnh, Google Image...

2.4.3 Công cụ tìm kiếm qua mạng Internet (Social Search)

Công cụ tìm kiếm qua mạng Internet (Social search hay social search engine) là loại hình tìm kiếm web dựa trên các đồ thị quan hệ xã hội (Social Graph) của người tìm kiếm. Việc tìm kiếm qua Social Search sẽ tiến hành thông qua việc phân tích các đoạn văn bản hoặc các cấu trúc liên kết văn bản bằng thuật toán hoặc phép tính toán gần đúng.

Social search có rất nhiều dạng từ dạng bookmark chia sẻ đơn giản cho đến việc gán nhãn mô tả bằng các thuật toán.

Hiện nay trên thế giới có rất nhiều công cụ tìm kiếm: Google, Yahoo, Facebook Graph Search, Bing... Trong đó, nổi bật lên là công cụ tìm kiếm Google search.

2.4.4 Hệ thống tin nhắn

Chuyển tiếp một tin nhắn từ người gửi đến người nhận thông qua một hệ thống mạng lưới được gọi là routing. Các thuật toán về routing đều là các biến thể, ở dạng này hay cách khác của các thuật toán đường đi ngắn nhất (tuyến đường các gói tin truyền có chi phí tối thiểu). Hệ thống mạng có thể mô tả như một đồ thị, chi phí giữa các cạnh phản ánh khả năng truyền tải, độ trễ, tắc nghẽn, tỷ lệ lỗi, và các đặc tính khác.

Trong hệ thống mạng, router phải biết chính xác nơi để chuyển tiếp một gói tin đến. Việc này được thực hiện dựa trên các thông tin trong các gói dữ liệu và các bảng định tuyến tại các router. Số lượng các mạng lưới được kết nối ngày càng lớn, bộ nhớ cần thiết để lưu trữ các bảng định tuyến phát triển. Yêu cầu bộ nhớ này thay đổi rất nhiều với các giao thức định tuyến và với kiến trúc của router.

Trong một số hệ thống nhỏ, một thiết bị trung tâm quyết định thời gian hoàn thành của tất cả các gói tin. Trong các hệ thống chuyển tin nhanh, có rất nhiều gói tin truyền đi mỗi giây cho nên việc chỉ có một thiết bị duy nhất để tính toán đường dẫn đầy đủ cho mỗi gói tin là không khả thi. Ban đầu việc xử lý gửi gói tin hệ thống tính toán đường đi ngắn nhất sau đó nếu có các gói tin khác có các nguồn giống nhau và đến cùng đích tiếp tục đi theo đường đi tương tự mà không tính toán lại cho đến khi không còn gói tin được gửi.

Trong các hệ thống lớn, có rất nhiều các kết nối giữa các thiết bị, và các kết nối này thay đổi rất thường xuyên, việc biết tất cả các thiết bị kết nối với nhau hay tính toán một đường đi ngắn nhất thông qua là không khả thi đối với bất kỳ

một thiết bị. Hệ thống như vậy thường sử dụng next-hop routing. Các ứng dụng: Hệ thống tin nhắn điện thoại, ...

Tiểu kết chương 2

Bài toán đường đi dài nhất trên đồ thị trong một số trường hợp có thể giải quyết bằng cách đổi dấu trọng số tất cả các cung rồi tìm đường đi ngắn nhất, nhưng hãy cẩn thận, có thể xảy ra trường hợp có chu trình âm. Trong tất cả các cài đặt trên, vì sử dụng ma trận trọng số chứ không sử dụng danh sách cạnh hay danh sách kề có trọng số, nên ta đều đưa về đồ thị đầy đủ và đem trọng số $+\infty$ gán cho những cạnh không có trong đồ thị ban đầu. Trên máy tính không có khái niệm trừu tượng $+\infty$ nên phải chọn một số dương đủ lớn để thay. Như thế nào là đủ lớn? số đó phải đủ lớn hơn tất cả trọng số của các đường đi cơ bản để cho dù đường đi thật có tồi tệ đến đâu vẫn tốt hơn đường đi trực tiếp theo cạnh tưởng tượng ra đó.

Xét về độ phức tạp tính toán, nếu cài đặt như trên, thuật toán Ford-Bellman có độ phức tạp là $O(n^3)$, thuật toán Dijkstra là $O(n^2)$, thuật toán tối ưu nhãn theo thứ tự tôpô là $O(n^2)$ còn thuật toán Floyd là $O(n^3)$. Tuy nhiên nếu sử dụng danh sách kề, thuật toán tối ưu nhãn theo thứ tự Tôpô sẽ có độ phức tạp tính toán là $O(m)$. Thuật toán Dijkstra kết hợp với cấu trúc dữ liệu Heap có độ phức tạp $O(\max(n, m) \cdot \log n)$.

Khác với một bài toán đại số hay hình học có nhiều cách giải thì chỉ cần nắm vững một cách cũng có thể coi là đạt yêu cầu, những thuật toán tìm đường đi ngắn nhất bộc lộ rất rõ ưu, nhược điểm trong từng trường hợp cụ thể (Ví dụ như số đỉnh của đồ thị quá lớn làm cho không thể biểu diễn bằng ma trận trọng số thì thuật toán Floyd sẽ gặp khó khăn, hay thuật toán Ford-Bellman làm việc khá chậm). Vì vậy yêu cầu trước tiên là phải hiểu bản chất và thành thạo trong việc cài đặt tất cả các thuật toán trên để có thể sử dụng chúng một cách linh hoạt trong từng trường hợp cụ thể.

CHƯƠNG 3. LƯỒNG TRÊN MẠNG (NETWORK FLOW)

3.1 Giới thiệu chung

Nhiều bài toán quy hoạch tuyến tính có thể quy về bài toán làm cực tiểu phí tổn vận chuyển hàng trong một lộ trình (hay mạng lưới) sao cho đảm bảo được các nhu cầu ở một số điểm đích khi đã biết nguồn cung cấp tại các điểm nguồn. Các bài toán như vậy được gọi là các **bài toán luồng trên mạng** (network flow problem) hoặc **bài toán chuyển vận** (transshipment problem). Đây là dạng bài toán quan trọng và hay gặp nhất trong quy hoạch tuyến tính. Dạng này bao gồm các bài toán quen thuộc trong thực tế như: bài toán vận tải, các bài toán mạng điện và mạng giao thông, các bài toán quản lý và phân bổ vật tư, bài toán bổ nhiệm, bài toán kế hoạch tài chính, bài toán đường ngắn nhất, bài toán luồng cực đại...

Bài toán luồng cực đại trong mạng cũng là một trong số những bài toán tối ưu trên đồ thị tìm được đề xuất vào đầu những năm 1950 và gắn liền với tên tuổi của hai nhà bác học Mỹ là **L.R.Ford** và **D.R.Fulkerson**. Bài toán luồng cực đại trong mạng có nhiều ứng dụng trong thực tế như: Bài toán xác định cường độ dòng lớn nhất của dòng vận tải giữa hai nút của một bản đồ giao thông, bài toán tìm luồng dầu lớn nhất có thể bơm từ tàu chở dầu vào bể chứa của một hệ thống đường ống dẫn dầu... Ngoài ra, ứng dụng của bài toán còn để giải các bài toán như: Bài toán đám cưới vùng quê, bài toán về hệ thống đại diện chung, bài toán phân nhóm sinh hoạt, bài toán lập lịch cho hội nghị...

3.2 Bài toán luồng trên mạng (Network flow problem)

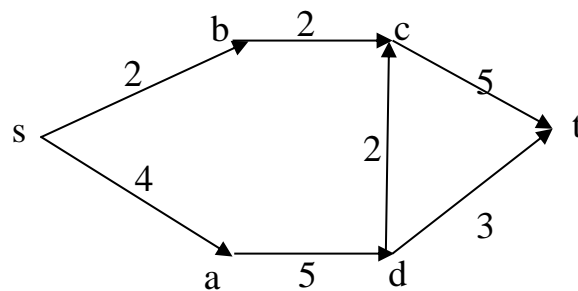
3.2.1 Định nghĩa

Mạng (Network)

Mạng (Network) là đơn đồ thị có hướng $G = (V, E)$, trong đó **V** (vertex) là các đỉnh, **E** (edge) là các cung, thoã mãn:

- Tồn tại duy nhất một đỉnh, gọi là đỉnh nguồn (source), đỉnh xuất phát. Tại đó chỉ có một cung đi ra, không có cung đi vào.
- Tồn tại duy nhất một đỉnh, gọi là đỉnh đích (sink), đỉnh kết thúc. Ngược lại với đỉnh nguồn, tại đó chỉ có một cung đi vào, không có cung đi ra.
- Mỗi cung được gán một giá trị không âm là $C_{i,j}$ của cung (i,j) gọi là trọng số của cung hay là khả năng cho đi qua của cung, sức chứa của cung (capacity).
- Nếu không tồn tại một cung giữa hai đỉnh a và b thì trọng số của cung bằng 0.
- Đồ thị G là đồ thị liên thông yếu.

Ví dụ 1: Đồ thị sau là mạng với nguồn là đỉnh s và đích là t , các đỉnh trung gian a,b,c,d



Hình 1: Đồ thị $G(V,E)$.

Luồng trong mạng (Flow)

Luồng là dòng chảy của giá trị, thông tin, dữ liệu... truyền từ đỉnh này sang đỉnh khác trong mạng thông qua các cung.

Cho mạng $G = (V, E)$, với trọng số $C_{i,j}$ $(i,j) \in G$. Tập các giá trị $\{f_{i,j} \mid (i,j) \in G\}$, gọi là luồng trên mạng G nếu thỏa mãn:

- Luồng trên cung không vượt quá khả năng thông qua của cung.

$$0 < f_{i,j} < C_{i,j} \quad \forall (i,j) \in G$$

- Với mỗi đỉnh k không là đỉnh nguồn, cũng không là đỉnh đích ($k \neq s$ và $k \neq t$) thì tổng luồng trên các cung đi vào k bằng tổng luồng trên các cung từ k đi ra

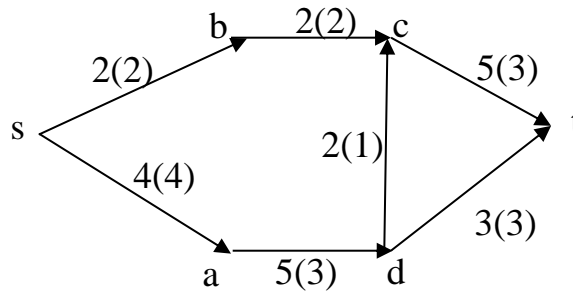
$$\sum_{(i,k) \in G} f(i,k) = \sum_{(k,j) \in G} f(k,j)$$

- Tổng luồng trên các cung phát ra từ điểm nguồn bằng với tổng luồng trên các cung thu vào tại điểm đích.

$$\sum_{(s,i) \in G} f(s,i) = \sum_{(j,t) \in G} f(j,t)$$

Ví dụ 2: Với đồ thị ở ví dụ 1, tập $\{ f_{i,j} \}$ bao gồm các luồng được biểu diễn bằng các số trong ngoặc đơn trong mạng

$$f_{sb} = 2, f_{sa} = 4, f_{bc} = 2, f_{ad} = 3, f_{dc} = 1, f_{ct} = 3, f_{dt} = 3$$



Hình 2: Đồ thị luồng dữ liệu.

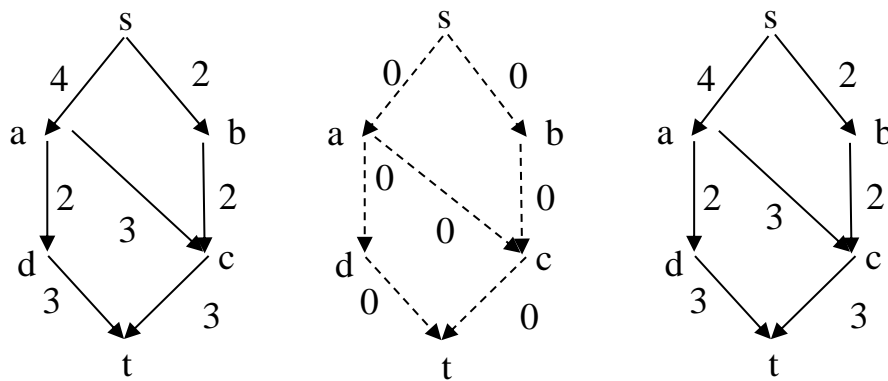
3.2.2 Bài toán luồng cực đại (Maximum-Flow)

Trong thực tế ta thường gặp bài toán gọi là bài toán tìm luồng cực đại như sau: Cho mạng G với nguồn s , đích t và khả năng thông qua $C_{i,j} \forall (i,j) \in G$. Trong số các luồng trên mạng G , tìm luồng có giá trị lớn nhất.

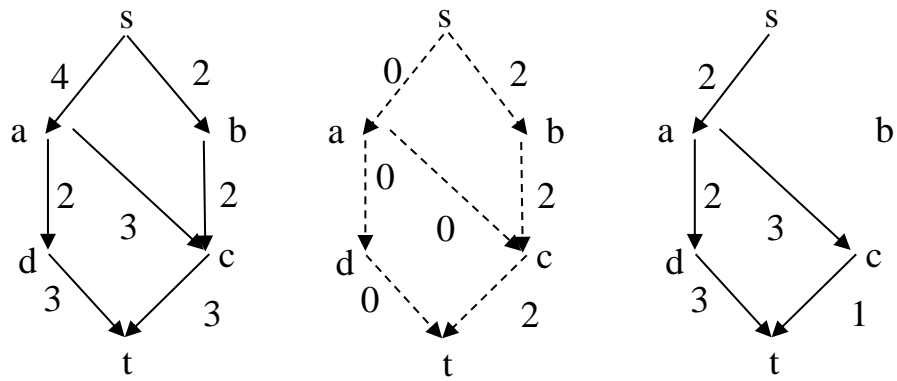
Ý tưởng xây dựng luồng cực đại như sau: xuất phát từ luồng nào đó, ta tìm đường đi từ s đến t sao cho lượng thông tin truyền qua luồng lớn nhất có thể, cho phép hiệu chỉnh giá trị luồng trên đường đi đó sao cho luồng mới có giá trị lớn hơn. Nếu không tìm được đường đi nào khác thì ta có đường đi đó là luồng cực đại.

Hướng tiếp cận đầu tiên là giải quyết vấn đề diễn ra theo từng giai đoạn. Bắt đầu với đồ thị G cho trước tiến hành xây dựng một biểu đồ dòng chảy G_f . G_f thể hiện giá trị của luồng đã đạt được ở các giai đoạn trong thuật toán. Ban đầu tất cả các cung trong G_f không có luồng và lý tưởng là thuật toán khi chấm dứt, G_f chứa một luồng tối đa. Bên cạnh đó xây dựng thêm một đồ thị G_r , được gọi **đồ thị thặng dư** (hay **mạng thặng dư**). G_r thể hiện rằng với mỗi cung, có thể thêm bao nhiêu luồng. G_r được tính bằng cách lấy sức chứa của cung trừ đi số luồng hiện tại đang chảy qua. Một cung trong G_r được gọi là **đường tăng luồng**.

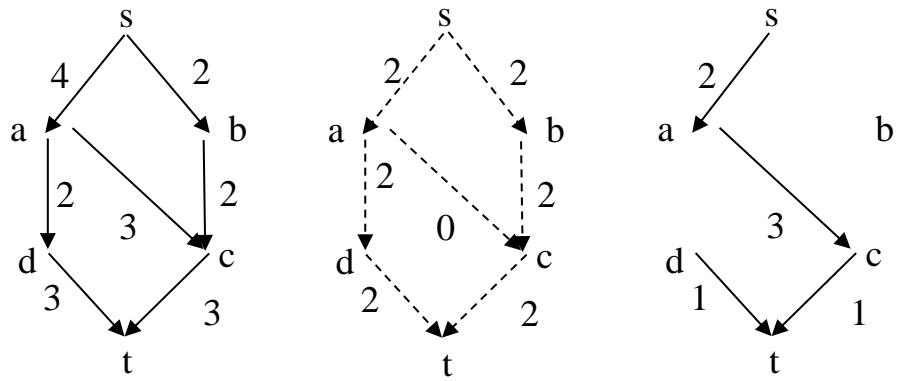
Ở mỗi giai đoạn của thuật toán, chúng ta tìm đường đi từ s đến t trong đồ thị G_r . Đường đi này như đã nhắc đến ở trên là đường tăng luồng. Mỗi lần ta thực hiện việc thay đổi đường đi trong G_f và cập nhật lại G_r , đến khi không còn đường đi từ s đến t trong đồ thị G_r thì kết thúc. Thuật toán này rất khó xác định vì chúng ta ngẫu nhiên lựa chọn đường đi từ s đến t , do đó sẽ xuất hiện rất nhiều đường mới tốt hơn đường đã chọn. Nội dung ở phần này không giải quyết vấn đề này mà chỉ giới thiệu một cách sơ nét về các thực hiện việc tìm luồng cực đại bằng phương pháp đơn giản nhất.



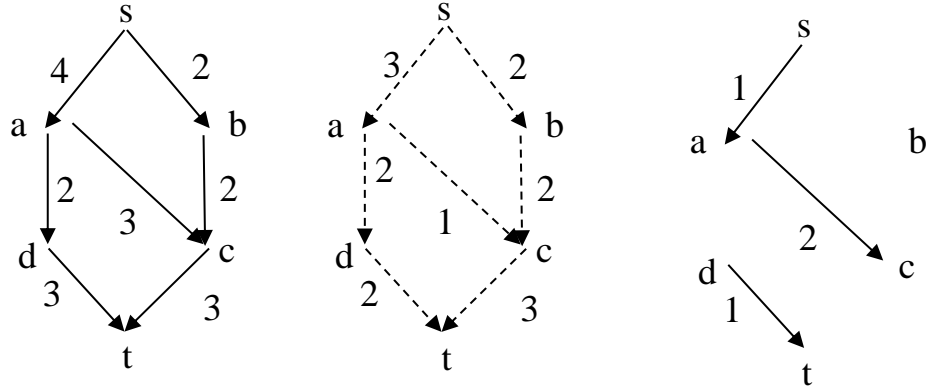
Hình 3: Khởi tạo đồ thị G , G_f , G_r .



Hình 4: Đồ thị G, G_f, G_r sau khi 2 giá trị truyền theo đường s, b, c, t .



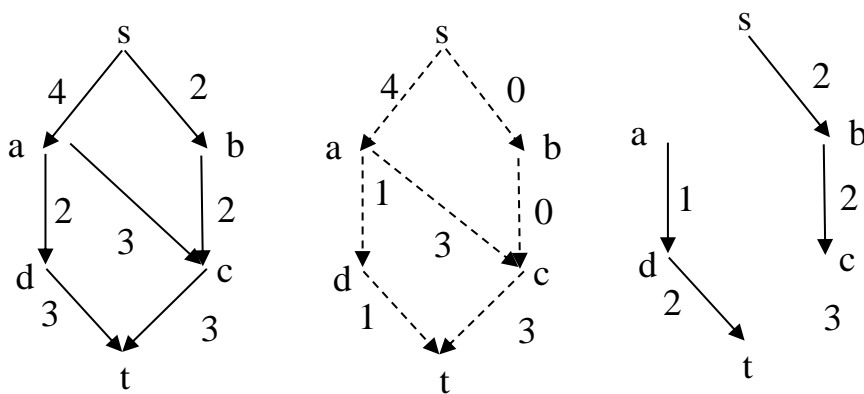
Hình 5: Đồ thị G, G_f, G_r sau khi 2 giá trị truyền theo đường s, a, d, t .



Hình 6: Đồ thị G , G_f , G_r sau khi 1 giá trị truyền theo đường s, a, c, t ; thuật toán kết thúc do không còn đường đi từ s đến t .

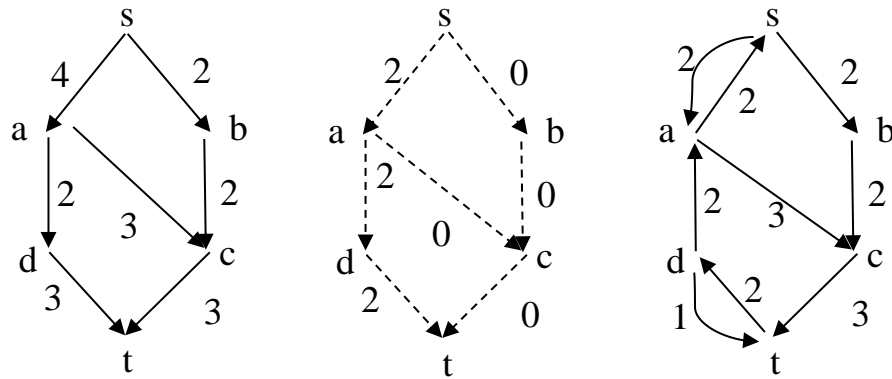
Các đường tăng luồng chỉ có thể đi qua các cung chưa bão hòa ($f_{i,j} < C_{i,j}$). Một cung gọi là bão hòa khi nó đã cho tối đa số luồng chạy qua ($f_{i,j} = C_{i,j}$). Một khi cung đã bão hòa thì sẽ bị xóa ra khỏi đồ thị thặng dư. Ở hình 4, cung sb và bc đã bão hòa cho tối đa 2 luồng chạy qua nên bị xóa, cung ct có sức chứa là 3, khi cho 2 luồng đi qua vẫn còn sức chứa là 1. Tương tự với hình 5 và 6.

Thuật toán kết thúc khi không còn đường đi từ điểm s đến t (hình 6). Số luồng đi theo con đường này là 5 và đạt giá trị cực đại.



Hình 7: Đồ thị G , G_f , G_r sau khi truyền 3 giá trị truyền theo đường s, a, c, t và 1 giá trị theo đường s, a, d, t ; thuật toán kết thúc do không còn đường đi từ s đến t .

Giả sử nếu thuật toán chọn một đường đi khác (như hình 7). Thuật toán kết thúc vì không còn đường đi từ điểm s đến t nhưng số luồng đi qua chỉ đạt được giá trị là 4. Đây là một ví dụ về thuật toán tham lam không hoạt động. Để giải quyết vấn đề này ta phải thay đổi một chút định hướng của thuật toán, ta cho phép thuật toán lùi lại một bước bằng cách trả lại các luồng đã cho qua theo hướng ngược lại ban đầu đã truyền. Mục đích việc đi ngược lại này là để tìm ra các đường tăng luồng để tìm được luồng cực đại.



Hình 8: Đồ thị G , G_f , G_r sau khi truyền ngược lại giá trị trên đường đi s, a, d, t .

3.2.3 Thuật toán Ford-Fulkerson

Thuật toán Ford-Fulkerson cũng xây dựng một đồ thị luồng thặng dư G_r (V, E) (residual graph). Sau đó, tìm một đường đi từ s tới t trong đồ thị luồng dư này, gọi là đường tăng luồng (augmenting path), và sau đó, tăng luồng của G dọc theo luồng này.

Ý tưởng của thuật toán:

- Khởi tạo đồ thị luồng thặng dư với các giá trị ban đầu bằng 0.
- Tìm các đường tăng luồng từ điểm nguồn s tới điểm đích t . Thêm các đường tăng luồng này vào luồng.
- Kết thúc thuật toán trả về luồng cực đại.

Để tìm một đường tăng luồng, chúng ta có thể sử dụng giải thuật tìm kiếm theo chiều rộng (Breadth First Search - BFS) hoặc tìm kiếm theo chiều sâu (Depth First Search - DFS) trên đồ thị luồng thặng dư. Giả thiết sau đây sử dụng giải thuật BFS. Khi áp dụng BFS, chúng ta có thể biết được có tồn tại một đường đi từ s tới t hay không. Trong quá trình tìm kiếm bằng BFS, giải thuật cũng xây dựng mảng lưu các điểm trước đó trên đường đi. Bằng việc sử dụng mảng này, chúng ta duyệt qua các đường đi đã tìm được và tìm số luồng có thể đi qua trên các đường này sao cho các luồng trên đồ thị thặng dư có giá trị nhỏ nhất. Sau đó chúng ta thêm luồng tìm thấy vào luồng tổng thể.

Điều quan trọng là, chúng ta cần cập nhật lại sức chứa trên đồ thị luồng thặng dư. Chúng ta trừ số luồng đã đi qua tất cả các cung trên đường đi từ s đến t và cộng thêm số luồng trên các cung đi ngược trở lại. Đây là điểm khác biệt cơ bản của thuật toán Ford-Fulkerson so với ý tưởng trong phần trước đã nêu. Với mỗi cung $u \rightarrow v \in G$ ta có luồng $f(u \rightarrow v) > 0$, trên một cung đó ta tạo ra 2 cung song song, cung xuôi $u \rightarrow v$ và cung ngược lại $v \rightarrow u$, và ta gán cho mỗi cung đó một trọng $c_f()$ như sau:

$$c_f(u \rightarrow v) = c_f(u \rightarrow v) - f(u \rightarrow v) \quad c_f(v \rightarrow u) = f(u \rightarrow v)$$

Việc tạo ra thêm một cung ngược $v \rightarrow u$ với lượng giảm luồng lớn nhất bằng với luồng cung xuôi là $f(u \rightarrow v)$ là vì nếu giảm quá mức này thì đồ thị sẽ nhận giá trị âm. Việc tạo ra các cung ngược là để từ điểm t ta truyền ngược lại được số lượng luồng tối đa mà t có thể nhận được, số luồng tối đa mà t nhận được chính là luồng cực đại của cả đồ thị.

Độ phức tạp của thuật toán: Chúng ta phải chạy một vòng lặp khi có một đường tăng luồng xuất hiện. Trong trường hợp xấu nhất, ta có thể tạo đường tăng luồng với giá trị là 1 trong mỗi lần lặp lại. Do đó độ phức tạp thời gian trở thành $O(\text{số luồng cực đại} * \text{số cạnh})$.

Cài đặt thuật toán:

- Thiết lập lớp đồ thị.

```

from collections import defaultdict
# Lớp class đại diện cho một đồ thị có hướng biểu diễn bằng ma trận 2 chiều
class Graph:
    def __init__(self, graph):
        self.graph = graph # Đồ thị thẳng du
        self.ROW = len(graph)

```

- Thiết lập hàm tìm kiếm theo chiều rộng BFS.

```

'''Hàm BFS trả về True khi có đường đi từ 's' đến 't' trong đồ thị thẳng du
Bên cạnh đó cũng lưu trữ mảng các đỉnh trước đó trong đường đi (parent[])'''

def BFS(self, s, t, parent):
    # Đánh dấu tất cả các đỉnh chưa đi qua
    visited = [False] * (self.ROW)
    # Tạo một ngăn xếp chứa BFS
    queue = []
    # Đánh dấu đỉnh trước và nối thêm nó vào đường đi
    queue.append(s)
    visited[s] = True
    # Chạy giải thuật BFS
    while queue:
        # Lấy ra 'u' là đỉnh sau cùng được đưa vào ngăn xếp
        u = queue.pop(0)
        # Tìm tất cả các đỉnh liền kề với 'u'
        # Nếu đỉnh liền kề chưa được đi qua thì đánh dấu
        # và nối nó vào đường đi
        for ind, val in enumerate(self.graph[u]):
            if visited[ind] == False and val > 0:
                queue.append(ind)
                visited[ind] = True
                parent[ind] = u
        # Nếu đường đi trong BFS dẫn đến đỉnh cuối
        # thì trả về true, ngược lại trả về false
    return True if visited[t] else False

```

- Thiết lập giải thuật tìm luồng cực đại Ford - Fulkerson.

```
# Trả về luồng cực đại từ 's' đến 't' trong đồ thị
def FordFulkerson(self, source, sink):
    parent = [-1] * (self.ROW)
    max_flow = 0 # Khởi tạo luồng cực đại
    # Tăng luồng khi tồn tại đường đi từ 's' đến 't'
    while self.BFS(source, sink, parent):
        # Tìm cạnh có sức chứa nhỏ nhất còn lại trong mạng thặng dư
        # được tính bằng BFS. Nói ngược lại đó là ta đang tìm luồng
        # cực đại trên cạnh đó.
        path_flow = float("Inf")
        s = sink
        while (s != source):
            path_flow = min(path_flow, self.graph[parent[s]][s])
            s = parent[s]
        # Thêm luồng vào luồng tổng thể
        max_flow += path_flow
        # cập nhật cung xuôi và ngược của đồ thị thặng dư
        # trên cung đang xét
        v = sink
        while (v != source):
            u = parent[v]
            self.graph[u][v] -= path_flow
            self.graph[v][u] += path_flow
            v = parent[v]
    return max_flow
```

- Xây dựng ma trận biểu diễn đồ thị và chạy thử.

```
# Tạo ma trận biểu diễn đồ thị có hướng
graph = [[0, 16, 13, 0, 0, 0],
          [0, 0, 10, 12, 0, 0],
          [0, 4, 0, 0, 14, 0],
          [0, 0, 9, 0, 0, 20],
          [0, 0, 0, 7, 0, 4],
          [0, 0, 0, 0, 0, 0]]

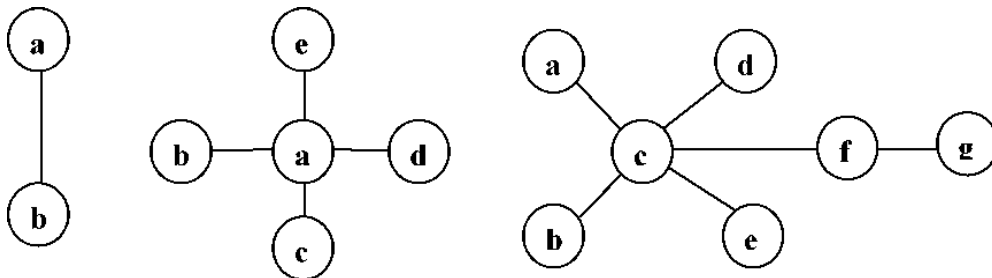
g = Graph(graph)
source = 0
sink = 5
print ("Luồng cực đại: %d " % g.FordFulkerson(source, sink))
```

CHƯƠNG 4. CÂY KHUNG NHỎ NHẤT (MINIMUM SPANNING TREE)

Một đồ thị liên thông và không có chu trình được gọi là cây. Cây được dùng từ năm 1857, khi nhà toán học Anh tên là Arthur Cayley dùng cây để xác định những dạng khác nhau của hợp chất hóa học. Từ đó cây đã được dùng để giải nhiều bài toán trong nhiều lĩnh vực khác nhau. Trong tin học cây được dùng để tìm kiếm các phần tử trong danh sách hoặc bài toán xây dựng các mạng máy tính với chi phí rẻ với các máy phân tán.

Định nghĩa và tính chất cơ bản của cây.

Cây là một đồ thị vô hướng liên thông, không chứa chu trình và có ít nhất hai đỉnh. Một đồ thị vô hướng không chứa chu trình và có ít nhất hai đỉnh gọi là một rừng. Trong một rừng, mỗi thành phần liên thông là một cây.



Sơ đồ hình cây

Định lý 1. Nếu T là một cây có n đỉnh thì T có ít nhất hai đỉnh treo.

Định lý 2. Cho đồ thị $T=(V,E)$ có $n \geq 2$ đỉnh. Sáu mệnh đề là tương đương:

- T là một cây.
- T liên thông và có $n-1$ cạnh.
- T không chứa chu trình và có $n-1$ cạnh.
- T liên thông và mỗi cạnh là cầu.
- Giữa hai đỉnh phân biệt của T luôn có duy nhất một đường đi sơ cấp.
- T không chứa chu trình nhưng khi bổ sung vào một cạnh nối hai đỉnh không kề nhau thì xuất hiện một chu trình.

4.1 Định nghĩa

Trong đồ thị liên thông G , nếu ta loại bỏ cạnh nằm trên chu trình nào đó thì ta sẽ được đồ thị vẫn là liên thông. Nếu cứ loại bỏ các cạnh ở các chu trình khác cho đến khi nào đồ thị không còn chu trình (vẫn liên thông) thì ta thu được một cây nối các đỉnh của G . Cây đó gọi là cây khung hay cây bao trùm của đồ thị G .

Tổng quát, nếu G là đồ thị có n đỉnh, m cạnh và k thành phần liên thông thì áp dụng thủ tục vừa mô tả đối với mỗi thành phần liên thông của G , ta thu được đồ thị gọi là

rừng khung của G . Số cạnh bị loại bỏ trong thủ tục này bằng $m-n+k$, số này ký hiệu là $v(G)$ và gọi là *chu số* của đồ thị G .

4.2 Bài toán tìm cây khung nhỏ nhất

Bài toán tìm cây khung nhỏ nhất của đồ thị là một trong số những bài toán tối ưu trên đồ thị tìm được ứng dụng trong nhiều lĩnh vực khác nhau của đời sống. Trong phần này ta sẽ có hai thuật toán cơ bản để giải bài toán này. Trước hết, nội dung của bài toán được phát biểu như sau.

Cho $G=(V,E)$ là đồ thị vô hướng liên thông có trọng số, mỗi cạnh E có trọng số $m(e)>0$. Giả sử $T=(V_T,E_T)$ là cây khung của đồ thị G ($V_T=V$). Ta gọi độ dài $m(T)$ của cây khung T là tổng trọng số của các cạnh của nó:

$$m(T) = \sum_{e \in E_T} m(e).$$

4.2.1 Bài toán xây dựng hệ thống đường sắt

Giả sử ta muốn xây dựng một hệ thống đường sắt nối n thành phố sao cho hành khách có thể đi từ bất cứ một thành phố nào đến bất kỳ một trong số các thành phố còn lại. Mặt khác, trên quan điểm kinh tế đòi hỏi là chi phí về xây dựng hệ thống đường phải là nhỏ nhất. Rõ ràng là đồ thị mà đỉnh là các thành phố còn các cạnh là các tuyến đường sắt nối các thành phố tương ứng, với phương án xây dựng tối ưu phải là cây. Vì vậy, bài toán đặt ra dẫn về bài toán tìm cây khung nhỏ nhất trên đồ thị đầy đủ n đỉnh, mỗi đỉnh tương ứng với một thành phố với độ dài trên các cạnh chính là chi phí xây dựng hệ thống đường sắt nối hai thành phố.

4.2.2 Bài toán nối mạng máy tính

Cần nối mạng một hệ thống gồm n máy tính đánh số từ 1 đến n . Biết chi phí nối máy i với máy j là $m(i,j)$ (thông thường chi phí này phụ thuộc vào độ dài cáp nối cần sử dụng). Hãy tìm cách nối mạng sao cho tổng chi phí là nhỏ nhất. Bài toán này cũng dẫn về bài toán tìm cây khung nhỏ nhất.

Bài toán tìm cây khung nhỏ nhất đã có những thuật toán rất hiệu quả để giải chúng. Trong tài liệu này sẽ xét hai trong số những thuật toán như vậy: thuật toán Kruskal và thuật toán Prim.

4.3 Thuật toán Kruskal

Thuật toán Kruskal dựa trên mô hình xây dựng cây khung nhỏ nhất bằng thuật toán hợp nhất.

- Thuật toán không xét các cạnh với thứ tự tùy ý.
- Thuật toán xét các cạnh theo thứ tự đã sắp xếp theo trọng số.

Để xây dựng tập $n-1$ cạnh của cây khung nhỏ nhất tạm gọi là tập K , Kruskal đề nghị cách kết nạp lần lượt các cạnh vào tập đó theo nguyên tắc như sau:

- Ưu tiên các cạnh có trọng số nhỏ hơn.
- Kết nạp cạnh khi nó không tạo chu trình với tập cạnh đã kết nạp trước đó.

Đó là một nguyên tắc chính xác và đúng đắn, đảm bảo tập K nếu thu đủ $n - 1$ cạnh sẽ là cây khung nhỏ nhất.

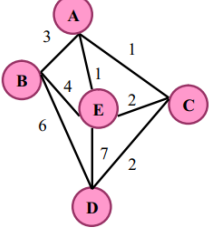
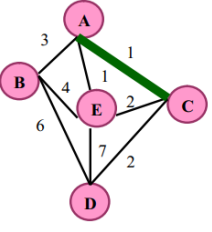
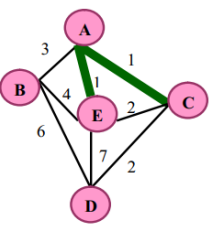
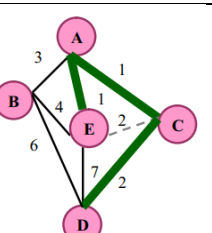
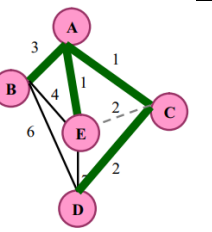
4.3.1 Mô tả thuật toán

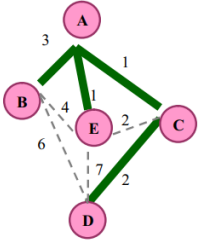
Giải thuật Kruskal không dựa trên tư tưởng của các thuật toán tìm kiếm theo chiều rộng hoặc chiều sâu. Trong các thuật toán này, tại từng bước của quá trình xây dựng T luôn là một cây, chỉ có điều kiện về số đỉnh của T phải đến bước cuối cùng mới thỏa mãn. Còn trong thuật toán Kruskal, trong quá trình xây dựng T có thể chưa là cây, nó chỉ thỏa mãn điều kiện không có chu trình.

Thuật toán sẽ xây dựng tập cạnh E_t của cây khung nhỏ nhất $T=(V, E_t)$ theo từng bước. Trước hết sắp xếp các cạnh của đồ thị G theo thứ tự không giảm của trọng số. Bắt đầu từ $E_t = 0$, ở mỗi bước ta sẽ lần lượt duyệt trong danh sách cạnh đã sắp xếp, từ cạnh có độ dài nhỏ đến cạnh có độ dài lớn hơn, để tìm ra cạnh mà việc bổ sung nó vào tập E_t không tạo thành chu trình trong tập này. Thuật toán sẽ kết thúc khi ta thu được tập E_t gồm $n-1$ cạnh. Cụ thể có thể mô tả như sau:

1. Bắt đầu từ đồ thị rỗng T có n đỉnh.
2. Sắp xếp các cạnh của G theo thứ tự không giảm của trọng số.
3. Bắt đầu từ cạnh đầu tiên của dãy này, ta cứ thêm dần các cạnh của dãy đã được xếp vào T theo nguyên tắc cạnh thêm vào không được tạo thành chu trình trong T .
4. Lặp lại Bước 3 cho đến khi nào số cạnh trong T bằng $n-1$, ta thu được cây khung nhỏ nhất cần tìm.

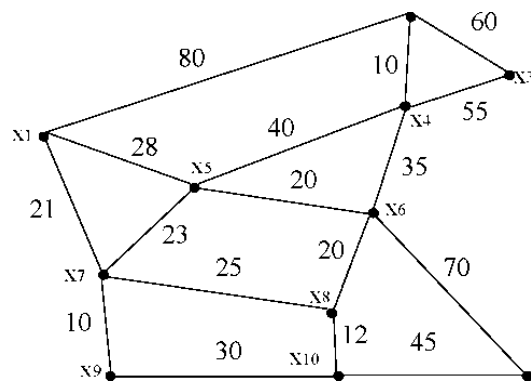
Từ ý tưởng của thuật toán Kruskal, ta có mô hình minh họa tìm cây khung nhỏ nhất cho đồ thị sau: $G = (V, E)$ vô hướng, $n = 5$ đỉnh

	<p>Khởi tạo $T = 0$ và sắp xếp các cạnh theo thứ tự trọng số tăng dần, ta có :</p> <p>$(A, C) = 1, (A, E) = 1, (C, D) = 2, (C, E) = 2, (A, B) = 3, (B, E) = 4, (B, D) = 6, (D, E) = 7$</p>
	<p>Bước 2: Lần lượt lấy từng cạnh trong danh sách đã sắp xếp vào T sao cho không tạo thành chu trình đơn.</p> <p>Δ Cạnh (A, C) bổ sung vào T ($T = 0$) không tạo chu trình nên:</p> <p>$T = \{(A, C)\}.$</p>
	<p>Bước 2 (lần 1):</p> <p>A Cạnh (A, E) bổ sung vào T không tạo chu trình nên:</p> <p>$T = \{(A, C); (A, E)\}$</p> <p>Bước 3 (lần 1): Vì T chưa đủ $n-1$ phần tử \rightarrow Tiếp tục Bước 2.</p>
	<p>Bước 2 (lần 2):</p> <p>A Cạnh (C, D) bổ sung vào T không tạo chu trình nên:</p> <p>$T = \{(A, C); (A, E); (C, D)\}$</p> <p>Bước 3 (lần 2): Vì T chưa đủ $n-1$ phần tử \rightarrow Tiếp tục Bước 2.</p> <p>Bước 2 (lần 3):</p> <p>A Cạnh (C, E) bổ sung vào T tạo chu trình \rightarrow Loại bỏ.</p>
	<p>Bước 2 (lần 4):</p> <p>A Cạnh (A, B) bổ sung vào T không tạo chu trình nên:</p> <p>$T = \{(A, C); (A, E); (C, D); (A, B)\}$</p> <p>Bước 3 (lần 3): Lúc này ta thấy T đã có đúng $n-1$ cạnh. Quá trình lặp kết thúc.</p>

	<p>Vậy $T = \{(A, C); (A, E); (C, D); (A, B)\}$ chính là tập cạnh của cây khung nhỏ nhất cần tìm</p>
---	---

Bảng 4.1. Thuật toán Kruskal

Ví dụ 1: Tìm cây khung nhỏ nhất của đồ thị cho trong hình bên dưới, đồ thị có 11 đỉnh, 17 cạnh như vậy cây khung có 10 cạnh.



Hình 4.2 Cây khung nhỏ nhất của đồ thị

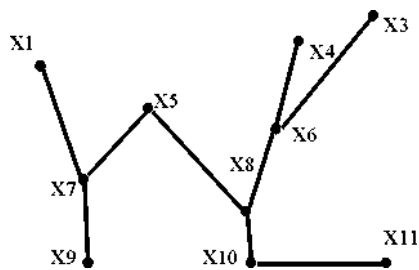
Bắt đầu từ đồ thị rỗng T có 11 đỉnh.

Bước sắp thứ tự các cạnh			Bước chọn các cạnh cho cây k hung nhỏ nhất		
Thứ tự	Cạnh	Trọng số	Bước chọn	Cạnh	Trọng số
1	x2 x4	10	1	x2 x4	10
2	x7 x9	10	2	x7 x9	10
3	x8 x10	12	3	x8	12
4	x5 x6	20	4	x5 x6	20
5	x6 x8	20	5	x6 x8	20
6	x1 x7	21	6	x1 x7	21
7	x5 x7	23	7	x5 x7	23
8	x1 x5	25	8	x4 x6	35
9	x7 x8	25	9	x10	45
10	x9 x10	30	1	x3 x4	55

11	x4 x6	35	Tổng trọng số các cạnh	251
12	x4 x5	40	<p>Ghi chú</p> <p>Sau bước chọn 7 ta không thể chọn các cạnh (x1,x5), (x7,x8) và (x9, x10) vì các cạnh này tạo thành chu trình với các cạnh đã chọn. Tình huống tương tự nếu chọn cạnh (x4,x5)</p>	
13	x10 x11	45		
14	x3 x4	55		
15	x2 x3	60		
16	x6 x11	70		
17	x1 x2	80		

Bảng 4.2. Kết quả chạy ví dụ 1

Kết thúc thuật toán được cây khung nhỏ nhất là cây. Trọng số của cây khung nhỏ nhất thu được là (T) = 251.



Hình 4.3 Cây khung nhỏ nhất

4.3.2 Cài đặt thuật toán Kruskal

```

1  from collections import defaultdict
2  class Graph:
3
4      def __init__(self, vertices):
5          self.V= vertices
6          self.graph = []
7      def addEdge(self,u,v,w):
8          self.graph.append([u,v,w])
9      def find(self, parent, i):
10         if parent[i] == i:
11             return i
12         return self.find(parent, parent[i])
13     def union(self, parent, rank, x, y):
14         xroot = self.find(parent, x)
15         yroot = self.find(parent, y)
16         if rank[xroot] < rank[yroot]:
17             parent[xroot] = yroot
18         elif rank[xroot] > rank[yroot]:
19             parent[yroot] = xroot
20
21         # If ranks are same, then make one as root
22         # and increment its rank by one
23         else :
24             parent[yroot] = xroot
25             rank[xroot] += 1

```

```

27     def KruskalMST(self):
28         result = []
29         i = 0
30         e = 0
31         self.graph = sorted(self.graph, key=lambda item: item[2])
32         parent = [] ; rank = []
33         for node in range(self.V):
34             parent.append(node)
35             rank.append(0)
36
37         while e < self.V - 1 :
38             u,v,w = self.graph[i]
39             i = i + 1
40             x = self.find(parent, u)
41             y = self.find(parent, v)
42             if x != y:
43                 e = e + 1
44                 result.append([u,v,w])
45                 self.union(parent, rank, x, y)
46         print( "Following are the edges in the constructed MST")
47         for u,v,weight in result:
48             print("%d -- %d == %d" % (u,v,weight))
49
50 g = Graph(4)
51 g.addEdge(0, 1, 10)
52 g.addEdge(0, 2, 6)
53 g.addEdge(0, 3, 5)
54 g.addEdge(1, 3, 15)
55 g.addEdge(2, 3, 4)
56 g.KruskalMST()

```

Hình 4.4 Cài đặt thuật toán bằng ngôn ngữ Python

4.3.3 Chứng minh tính đúng đắn

Đồ thị thu được theo thuật toán có $n-1$ cạnh và không có chu trình, vì vậy theo định lý 2 nó là cây khung của đồ thị G . Như vậy, chỉ còn phải chỉ ra rằng T có độ dài nhỏ nhất. Giả sử tồn tại cây S của đồ thị mà $c(S) < c(T)$. Ký hiệu là e_k là cạnh đầu tiên trong dãy các cạnh của T không thuộc S . Khi đó đồ thị con của G sinh bởi cây S được bổ sung cạnh e_k sẽ chứa một chu trình duy nhất C đi qua e_k . Do chu trình C phải chứa cạnh e thuộc S nhưng không thuộc T nên đồ thị con thu được từ S bằng cách thay cạnh e của nó bởi e_k (ký hiệu đồ thị này là S') sẽ là cây khung. Theo cách xây dựng $c(e_k) < c(e)$ do đó $c(S') < c(S)$, đồng thời số cạnh chung của S' và T đã tăng thêm một so với số cạnh chung của S và T . Lặp lại quá trình trên từng bước một ta có thể biến đổi S thành T và trong mỗi bước tổng độ dài không tăng, tức là $c(T) < c(S)$. Ta thu được chứng tỏ T là cây khung nhỏ nhất.

4.3.4 Độ phức tạp của thuật toán Kruskal

Trước tiên, việc sắp xếp các cạnh của G theo thứ tự trọng số tăng dần có độ phức tạp $O(m^2)$, với m là số cạnh của G . Người ta chứng minh được rằng việc chọn cạnh u_{k+1} không tạo nên chu trình với k cạnh đã chọn trước đó có độ phức tạp $O(n^2)$. Do $m \leq \frac{n(n-1)}{2}$ nên độ phức tạp thuật toán là $O(n^2)$.

4.4 Thuật toán Prim

Thuật toán Prim là một thuật toán tham lam để tìm cây bao trùm nhỏ nhất của một đồ thị vô hướng có trọng số liên thông. Nghĩa là nó tìm một tập hợp các cạnh của đồ thị tạo thành một cây chứa tất cả các đỉnh, sao cho tổng trọng số các cạnh của cây là nhỏ nhất.

Thuật toán được tìm ra năm 1930 bởi nhà toán học người Séc Vojtěch Jarník và sau đó bởi nhà nghiên cứu khoa học máy tính Robert C. Prim năm 1957 và một lần nữa độc lập bởi Edsger Dijkstra năm 1959.

4.4.1 Mô tả thuật toán Prim:

Thuật toán xuất phát từ một cây chỉ chứa đúng một đỉnh và mở rộng từng bước một, mỗi bước thêm một cạnh mới vào cây, cho tới khi bao trùm được tất cả các đỉnh của đồ thị.

- Dữ liệu vào: Một đồ thị có trọng số liên thông với tập hợp đỉnh V và tập hợp cạnh E (trọng số có thể âm). Đồng thời cũng dùng V và E để ký hiệu số đỉnh và số cạnh của đồ thị.
- Khởi tạo: $V_{\text{mới}} = \{x\}$, trong đó x là một đỉnh bất kì (đỉnh bắt đầu) trong V , $E_{\text{mới}} = \{\}$
- Lặp lại cho tới khi $V_{\text{mới}} = V$:
 - Chọn cạnh (u, v) có trọng số nhỏ nhất thỏa mãn u thuộc $V_{\text{mới}}$ và v không thuộc $V_{\text{mới}}$ (nếu có nhiều cạnh như vậy thì chọn một cạnh bất kì trong chúng)
 - Thêm v vào $V_{\text{mới}}$, và thêm cạnh (u, v) vào $E_{\text{mới}}$
- Dữ liệu ra: $V_{\text{mới}}$ và $E_{\text{mới}}$ là tập hợp đỉnh và tập hợp cạnh của một cây bao trùm nhỏ nhất

4.4.2 Thuật toán Prim

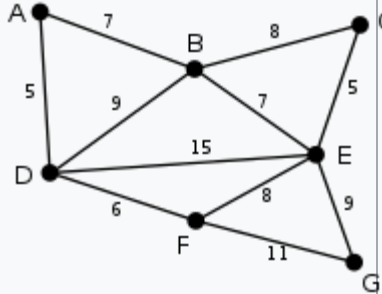
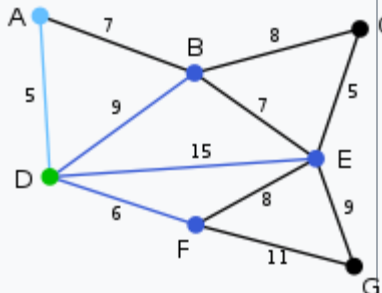
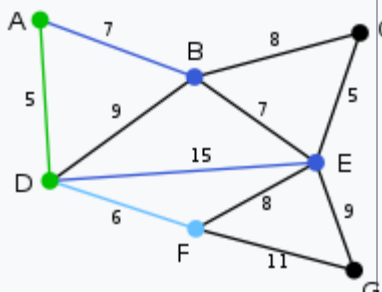
Cho $G = (X, E)$ là một đồ thị liên thông có trọng số gồm n đỉnh. Thuật toán Prim được dùng để tìm ra cây khung nhỏ nhất của G .

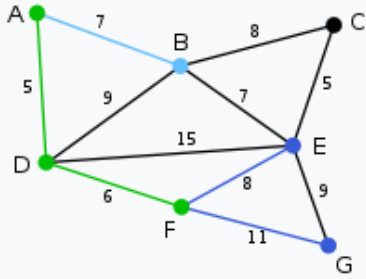
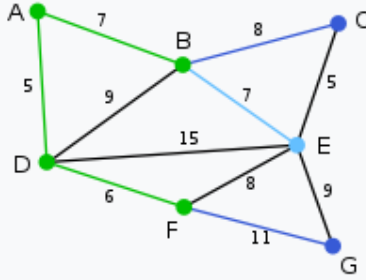
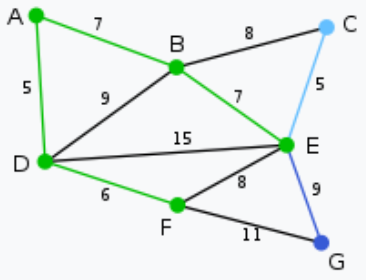
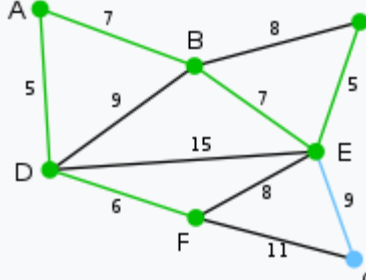
• *Bước 1:* Chọn tùy ý x_0 thuộc X và khởi tạo $V := \{x_0\}$; $T := \emptyset$. Trong đó X là tập các đỉnh của đồ thị, V là tập các đỉnh được chọn vào cây khung nhỏ nhất và T là tập các cạnh của cây này.

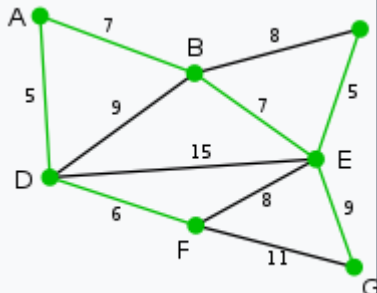
• *Bước 2:* Trong số những cạnh nối đỉnh x với đỉnh y mà $x \in V$ và $y \in X \setminus V$ ta chọn cạnh e có trọng số nhỏ nhất. Nếu không có cạnh e thỏa yêu cầu: **DỪNG (1)**

- *Bước 3:* Thêm đỉnh y vào tập V và thêm cạnh e vào tập T .
- *Bước 4:* Nếu T đủ $n - 1$ phần tử thì **DỪNG (2)**, ngược lại làm tiếp tục bước 2.

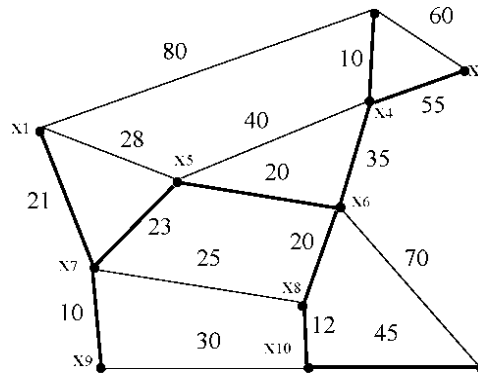
Ví dụ minh họa: Tìm khung cây nhỏ nhất của bài toán sau:

Hình minh họa	U	Cạnh (u,v)	$V \setminus U$	Mô tả
	$\{\}$		$\{A,B,C,D,E,F,G\}$	Đây là đồ thị có trọng số ban đầu. Các số là các trọng số của các cạnh.
	$\{D\}$	$(D,A) = 5$ $(D,B) = 9$ $(D,E) = 15$ $(D,F) = 6$	$\{A,B,C,E,F,G\}$	Chọn một cách tùy ý đỉnh D là đỉnh bắt đầu. Các đỉnh A, B, E và F đều được nối trực tiếp tới D bằng cạnh của đồ thị. A là đỉnh gần D nhất nên ta chọn A là đỉnh thứ hai của cây và thêm cạnh AD vào cây.
	$\{A,D\}$	$(D,B) = 9$ $(D,E) = 15$ $(D,F) = 6$ $(A,B) = 7$	$\{B,C,E,F,G\}$	Đỉnh được chọn tiếp theo là đỉnh gần D hoặc A nhất. B có khoảng cách tới D bằng 9 và tới A bằng 7, E có khoảng cách tới cây hiện tại bằng 15, và F có khoảng cách bằng 6. F là đỉnh gần cây hiện tại nhất nên chọn đỉnh F và

				cạnh DF .
	{A,D, F}	$(D,B) = 9$ $(D,E) = 15$ $(A,B) = 7$ $(F,E) = 8$ $(F,G) = 11$	{B,C,E,G }	Thuật toán tiếp tục tương tự như bước trước. Chọn đỉnh B có khoảng cách tới A bằng 7.
	{A,B, D,F}	$(B,C) = 8$ $(B,E) = 7$ $(D,B) = 9$ chu trình $(D,E) = 15$ $(F,E) = 8$ $(F,G) = 11$	{C,E,G}	Ở bước này ta chọn giữa C , E , và G . C có khoảng cách tới B bằng 8, E có khoảng cách tới B bằng 7, và G có khoảng cách tới F bằng 11. E là đỉnh gần nhất, nên chọn đỉnh E và cạnh BE .
	{A,B, D,E,F}	$(B,C) = 8$ $(D,B) = 9$ chu trình $(D,E) = 15$ chu trình $(E,C) = 5$ V $(E,G) = 9$ $(F,E) = 8$ chu trình $(F,G) = 11$	{C,G}	Ở bước này ta chọn giữa C và G . C có khoảng cách tới E bằng 5, và G có khoảng cách tới E bằng 9. Chọn C và cạnh EC .
	{A,B, C,D,E, F}	$(B,C) = 8$ chu trình $(D,B) = 9$ chu trình $(D,E) = 15$ chu trình $(E,G) = 9$ V $(F,E) = 8$ chu trình $(F,G) = 11$	{G}	Đỉnh G là đỉnh còn lại duy nhất. Nó có khoảng cách tới F bằng 11, và khoảng cách tới E bằng 9. E ở gần hơn nên chọn đỉnh G và cạnh EG .

		11		
	{A,B, C,D,E ,F,G}	(B,C) = 8 chu trình (D,B) = 9 chu trình (D,E) = 15 chu trình (F,E) = 8 chu trình (F,G) = 11 chu trình	{}	Hiện giờ tất cả các đỉnh đã nằm trong cây và cây bao trùm nhỏ nhất được tô màu xanh lá cây. Tổng trọng số của cây là 39.

Bảng 4.3. Minh họa thuật toán Prim

Ví dụ 1:

Hình 4.5 Cây khung có trọng số

Bước	Cạnh	Trọng số
1	(x7,x9)	10
2	(x7,x1)	21
3	(x7,x5)	23
4	(x5,x6)	20
5	(x6,x8)	20
6	(x8,x10)	12
7	(x6,x4)	35
8	(x4,x2)	10
9	(x10,x11)	45
10	(x4,x3)	55
l(T)		251

Bảng 4.2. Kết quả chạy ví dụ

Cây khung nhỏ nhất thu được thể hiện bằng nét đậm trên hình và trọng số của nó là $l(T) = 251$

4.4.3 Cài đặt thuật toán Prim

```

1  import sys
2  class Graph():
3
4      def __init__(self, vertices):
5          self.V = vertices
6          self.graph = [[0 for column in range(vertices)]
7                        for row in range(vertices)]
8
9      def printMST(self, parent):
10         print( "Edge \tWeight")
11         for i in range(1,self.V):
12             print( parent[i],"-",i,"\t",self.graph[i][ parent[i] ])
13
14     def minKey(self, key, mstSet):
15         min = sys.maxsize
16         for v in range(self.V):
17             if key[v] < min and mstSet[v] == False:
18                 min = key[v]
19                 min_index = v
20
21         return min_index
22
23     def primMST(self):
24         key = [sys.maxsize] * self.V
25         parent = [None] * self.V
26         key[0] = 0
27         mstSet = [False] * self.V
28         parent[0] = -1
29
30         for cout in range(self.V):
31             u = self.minKey(key, mstSet)
32             mstSet[u] = True
33             for v in range(self.V):
34                 if self.graph[u][v] > 0 and mstSet[v] == False and key[v] > self.graph[u][v]:
35                     key[v] = self.graph[u][v]
36                     parent[v] = u
37
38         self.printMST(parent)
39
40 g = Graph(5)
41 g.graph = [ [0, 2, 0, 6, 0],
42             [2, 0, 3, 8, 5],
43             [0, 3, 0, 0, 7],
44             [6, 8, 0, 0, 9],
45             [0, 5, 7, 9, 0],
46             ]
47
48 g.primMST()

```

Hình 4.6 Cài đặt thuật toán Prim bằng ngôn ngữ Python

4.4.4 Độ phức tạp thuật toán Prim

Cấu trúc dữ liệu tìm cạnh có trọng số nhỏ nhất	Độ phức tạp thời gian (tổng cộng)
Tìm kiếm trên ma trận kề	$O(V ^2)$
Đồng nhị phân và danh sách kề	$O((V + E) \log V) = O(E \log V)$
Đồng Fibonacci và danh sách kề	$O(E + V \log V)$

Bảng 4.2. Liệt kê độ phức tạp thuật toán

4.5 SO SÁNH HAI THUẬT TOÁN Prim và Kruskal

4.5.1 Xét các cạnh đưa vào cây

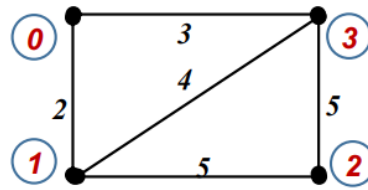
- *Prime*: thực hiện việc mở rộng tập đã xét (ban đầu chỉ gồm một đỉnh, 0 cạnh thành n đỉnh, n-1 cạnh) dựa trên các cạnh ngắn nhất nối giữa tập đã xét và tập chưa xét. Tư tưởng chính là thêm vào các cạnh ngắn nhất sao cho không tạo ra chu trình. Như vậy, có trường hợp một cạnh sẽ phải xét đi xét nhiều lần rồi mới được chọn, thậm chí không hề được chọn.
- *Kruskal*: cũng thêm lần lượt các cạnh vào đồ thị, theo thứ tự từ trọng nhỏ nhất đến trọng lớn nhất (như vậy mỗi cạnh sẽ chỉ được duyệt một lần duy nhất). Ta chỉ bổ sung cạnh vào cây khung nếu việc thêm cạnh này không làm phát sinh ra chu trình.

4.5.2 . Kiểm tra tính liên thông của đồ thị

- *Prim*: cần kiểm tra đồ thị liên thông trước khi thi hành thuật toán.
- *Kruskal*: không cần kiểm tra đồ thị liên thông trước khi thi hành thuật toán. Nếu quá trình thi hành thuật toán tìm được cây khung/bao trùm thì đồ thị liên thông và ngược lại là đồ thị không liên thông.

4.5.3 . Chi phí cho việc kiểm tra chu trình

- *Prime*: mở rộng tập đỉnh đã xét (mỗi lần 1 đỉnh nên không cần kiểm tra chu trình).
- *Kruskal*: kiểm tra nếu khi thêm cạnh đang xét vào cây khung mà không làm phát sinh chu trình thì sẽ chọn cạnh này. Việc kiểm tra chu trình có khả năng gây tốn chi phí. Gợi ý cải tiến giảm chi phí kiểm tra chu trình:
 - Dùng một mảng Nhãn có kích thước bằng số đỉnh của đồ thị. Giá trị Nhãn[i] tại đỉnh i được khởi tạo bằng với chính chỉ số i của đỉnh.



Ví dụ: Với đồ thị G ở trên ta có

Đỉnh	0	1	2	3
Nhãn	0	1	2	3

- Khi chọn một cạnh (u,v) được thêm vào cây khung/cây bao trùm, ta sửa nhãn của tất cả các đỉnh có cùng giá trị với nhãn của đỉnh v thành nhãn của đỉnh u .

Ví dụ:

- Sau khi thêm cạnh $(0,1)$ vào cây khung/cây bao trùm hay tập T , ta sửa nhãn của tất cả các đỉnh có cùng giá trị với nhãn của đỉnh 1 (là 1) thành nhãn của đỉnh 0 (là 0).

Đỉnh	0	1	2	3
Nhãn	0	0	2	3

- Sau đó, khi chọn tiếp cạnh $(0, 3)$, ta sửa nhãn của đỉnh 3 (đang có giá trị là 3) để có cùng giá trị với nhãn của đỉnh 0 (là 0).

Đỉnh	0	1	2	3
Nhãn	0	0	2	0

- Sau đó, khi chọn cạnh để thêm vào cây khung/cây bao trùm thì chỉ chọn cạnh có nhãn hai đỉnh là khác nhau sẽ không tạo thành chu trình.

Ví dụ:

- Khi ta xét tới cạnh $(1,3)$: ta không chọn cạnh này vì đỉnh 1 và đỉnh 3 có cùng nhãn là 0
- Chọn tiếp cạnh $(1,2)$: 2 đỉnh 1 và 2 có nhãn khác nhau nên cạnh $(1,2)$ sẽ được chọn đưa vào cây khung/cây bao trùm.

Đỉnh	0	1	2	3
Nhãn	0	0	0	0

Nhận xét: Sau khi ta đã chọn đủ $n-1$ cạnh, mảng nhãn chỉ chứa một giá trị duy nhất.

KẾT LUẬN

Lý thuyết đồ thị được nghiên cứu và phát triển do nảy sinh từ nhu cầu giải quyết các vấn đề thực tiễn, có nhiều ứng dụng trong các ngành khoa học kỹ thuật khác nhau. Lý thuyết đồ thị nói riêng và môn học Phương pháp toán trong tin học nói chung đóng vai trò làm cơ sở toán cho Tin học và Khoa học máy tính vì đồ thị là một bộ phận của Toán rời rạc, bản chất và cấu trúc của đồ thị mang tính rời rạc mà công cụ chính trong Tin học là máy tính, các quá trình xử lý thông tin trong máy cũng mang tính rời rạc, nên điều này tương hợp giữa đồ thị và máy tính. Bài tiểu luận của nhóm trên đây đã phần nào đạt được mục tiêu đề ra là mang lại cái nhìn tổng thể về lý thuyết đồ thị, những ứng dụng quan trọng trong việc giải các bài toán rất thực tiễn, tạo tiền đề cơ bản rất cần thiết cho quá trình nghiên cứu chuyên ngành Khoa học máy tính. Tuy nhiên, lý thuyết đồ thị có cả một khối lượng kiến thức đồ sộ, ứng dụng rất rộng nên bài viết mới chỉ mang lại những nội dung cơ sở và trọng tâm của đồ thị và không tránh khỏi những hạn chế. Hướng nghiên cứu mở rộng của bài viết là sử dụng các ngôn ngữ lập trình hiện đại như Python, R... để cài đặt các thuật toán lý thuyết đồ thị, ứng dụng vào máy học (machine learning), deeplearning, khoa học dữ liệu (data science), khai phá dữ liệu (data mining) và nhiều lĩnh vực, chuyên ngành khác./.

TÀI LIỆU THAM KHẢO

- [1] Kenneth H. Rosen, *Discrete Mathematics and Its Applications*, Mc Graw-Hill, 2000.
- [2] Kenneth H. Rosen. *Toán học rời rạc và Ứng dụng trong tin học*, Nhà xuất bản lao động 2010, người dịch Bùi Xuân Toại.
- [3] Mark Allen Weiss, *Data structures and algorithm analysis in Java*, 3rd Edition, Addison-Wesley 2012.
- [4] Srinu Devadas and Eric Lehman, *Generating Functions*, Lectures Notes, April 2005.
- [5] Wikipedia: Generating Functions, Probability Generating Functions
- [6] Robert Sedgewick and Kevin Wayne, *Algorithms*, 4th Edition, Addison-Wesley 2011.
- [7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, third edition, The MIT Press 2009.
- [8] Robert Sedgewick, Philippe Flajolet, *An introduction to the analysis of algorithms*, Addison-Wesley 2013.
- [9] Michael H. Goldwasser, Michael T. Goodrich, and Roberto Tamassia, *Data Structures and Algorithms in Python*, Wiley 2013.
- [10] Mathematics 113. Analysis I: Complex Function Theory, *Complex Analysis*, Harvard University, Spring 2013.
- [11] Ian Parberry, *Lecture Notes on Algorithm Analysis and Computational Complexity*, fourth edition, University of North Texas 2001
- [12] Ian Parberry and William Gasarch, *Problems on Algorithms*, second edition, PrenticeHall 2002.