

**BỘ GIÁO DỤC VÀ ĐÀO TẠO  
ĐẠI HỌC CÔNG NGHỆ TP.HCM**

# **THỰC HÀNH LÝ THUYẾT ĐỒ THỊ**

**Biên Soạn:**

ThS.Nguyễn Kim Hưng

THỰC HÀNH LÝ THUYẾT ĐỒ THỊ



★ 1 . 2 0 1 8 . C M P 3 0 1 4 ★

---

*Các ý kiến đóng góp về tài liệu học tập này, xin gửi về e-mail của ban biên tập:  
tailieuhoctap@hutech.edu.vn*

# MỤC LỤC

MỤC LỤC .....	I
HƯỚNG DẪN .....	II
BÀI 1: ĐỒ THỊ, LIÊN THÔNG VÀ BIỂU DIỄN MA TRẬN KẼ .....	1
1.1 ĐỒ THỊ .....	1
1.2 PHÂN LOẠI ĐỒ THỊ .....	2
1.3 MA TRẬN KẼ .....	2
1.4 NHẬP XUẤT MA TRẬN KẼ TỪ FILE .....	4
1.5 ĐỒ THỊ LIÊN THÔNG .....	9
1.6 THUẬT GIẢI ĐI TÌM CÁC THÀNH PHẦN LIÊN THÔNG CỦA ĐỒ THỊ .....	11
1.7 HƯỚNG DẪN TÌM ĐỒ THỊ LIÊN THÔNG .....	14
BÀI 2: ĐƯỜNG ĐI VÀ CHU TRÌNH EULER .....	17
2.1 MỘT SỐ ĐỊNH NGHĨA .....	17
2.2 THUẬT TOÁN FLEURY .....	18
2.3 THUẬT TOÁN TÌM CHU TRÌNH VÀ ĐƯỜNG ĐI EULER .....	19
2.4 HƯỚNG DẪN TÌM CHU TRÌNH EULER .....	22
2.5 HƯỚNG DẪN TÌM ĐƯỜNG ĐI .....	25
BÀI 3: TÌM KIẾM ĐƯỜNG ĐI TRÊN ĐỒ THỊ THEO CHIỀU SÂU VÀ CHIỀU RỘNG .....	28
3.1 CÁC KHÁI NIỆM .....	28
3.2 THUẬT TOÁN TÌM KIẾM THEO CHIỀU SÂU DFS_DEPTH FIRST SEARCH .....	29
3.3 HƯỚNG DẪN TÌM KIẾM ĐƯỜNG ĐI BẰNG PHƯƠNG PHÁP ... CHIỀU SÂU (DFS) ...	35
3.4 THUẬT TOÁN TÌM KIẾM THEO CHIỀU RỘNG BFS (BREADTH FIRST SEARCH) .....	38
3.5 HƯỚNG DẪN TÌM KIẾM ĐƯỜNG ĐI BẰNG ... CHIỀU RỘNG (BFS) .....	44
BÀI 4: TÌM CÂY KHUNG NHỎ NHẤT .....	49
4.1 ĐỊNH NGHĨA CÂY KHUNG .....	49
4.2 THUẬT TOÁN PRIM .....	49
4.3 HƯỚNG DẪN THI HÀNH THUẬT TOÁN PRIM .....	52
4.4 THUẬT TOÁN KRUSKAL .....	56
4.5 SO SÁNH SỰ KHÁC BIỆT GIỮA KRUSKAL VỚI PRIM .....	60
4.6 HƯỚNG DẪN THI HÀNH THUẬT TOÁN KRUSKAL .....	61
BÀI 5: CÁC THUẬT TOÁN TÌM ĐƯỜNG ĐI NGẮN NHẤT .....	64
5.1 GIỚI THIỆU .....	64
5.2 THUẬT TOÁN DIJKSTRA .....	65
5.3 HƯỚNG DẪN THI HÀNH TÌM ĐƯỜNG ĐI NGẮN NHẤT VỚI ... DIJKSTRA .....	70
5.4 THUẬT TOÁN FLOYD .....	73
5.5 HƯỚNG DẪN THI HÀNH TÌM KIẾM ĐƯỜNG ĐI NGẮN NHẤT ... TOÁN FLOYD .....	79
TÀI LIỆU THAM KHẢO .....	82

# HƯỚNG DẪN

## MÔ TẢ MÔN HỌC

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.

Môn học cung cấp cho sinh viên kiến thức về đồ thị và những thuật toán xử lý trên đồ thị. Từ đó có thể áp dụng vào một số bài toán mang tính thực tế cao, ví dụ như: nếu dùng mô hình đồ thị mạng máy tính có thể xác định xem hai máy tính có được nối với nhau bằng một đường truyền thông hay không, đồ thị với các trọng số được gán cho các cạnh của nó có thể dùng để giải các bài toán như bài toán tìm đường đi ngắn nhất giữa hai thành phố trong một mạng giao thông, ...

## NỘI DUNG MÔN HỌC

- Bài 1: Đồ thị, liên thông và biểu diễn ma trận kề.
- Bài 2: Đường đi và chu trình Euler.
- Bài 3: Tìm kiếm đường đi trên đồ thị theo chiều rộng và chiều sâu.
- Bài 4: Tìm cây khung/cây bao trùm nhỏ nhất.
- Bài 5: Các thuật toán tìm đường đi ngắn nhất.

## KIẾN THỨC TIỀN ĐỀ

Môn học đòi hỏi sinh viên có một số kiến thức cơ bản về Toán rời rạc và Kỹ thuật lập trình trên ngôn ngữ C++ hoặc Java.

## YÊU CẦU MÔN HỌC

Người học phải dự học đầy đủ các buổi lên lớp lý thuyết, thực hành và làm bài tập đầy đủ ở nhà.

## CÁCH TIẾP NHẬN NỘI DUNG MÔN HỌC

Môn học được đánh giá gồm:

- **Điểm đánh giá quá trình** môn học, do giảng viên phụ trách quyết định bao gồm: điểm kiểm tra thường xuyên trong quá trình học tập, điểm tiểu luận, điểm làm bài tập trên lớp, hoặc điểm chuyên cần.
- **Điểm đánh giá phần thực hành**, do giảng viên dạy thực hành quyết định như chấm bài tập từng buổi, hoặc cho thi ở buổi học cuối cùng.
- **Điểm học phần** =  $50\% \times \text{điểm đánh giá quá trình} + 50\% \times \text{điểm đánh giá thực hành}$ .



# BÀI 1: ĐỒ THỊ, LIÊN THÔNG VÀ BIỂU DIỄN MA TRẬN KẼ

Học xong bài này người học sẽ:

- Hiểu được đồ thị là gì? Thế nào là đồ thị có hướng và đồ thị vô hướng.
- Hiểu được thế nào là ma trận kề. Cách nhập xuất ma trận kề từ file.
- Biết được đồ thị liên thông là như thế nào?
- Nắm được thuật giải để tìm các thành phần liên thông của đồ thị.

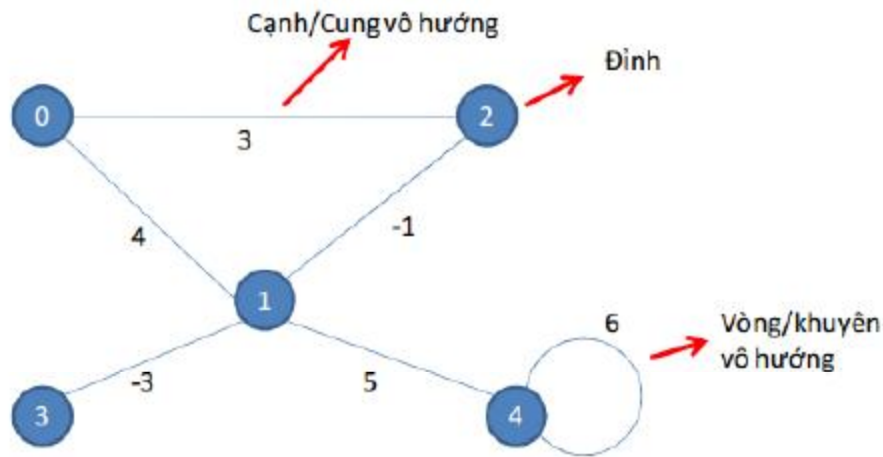
## 1.1 ĐỒ THỊ

---

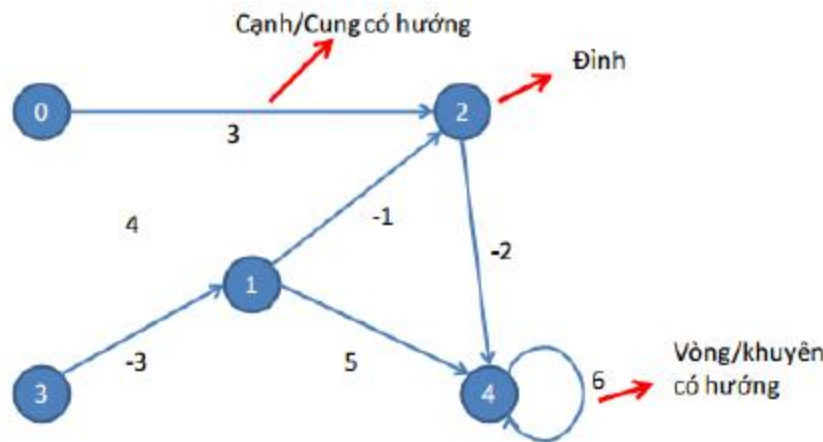
Đồ thị bao gồm hai thành phần:

- Đỉnh.
- Cung/Cạnh: Về cung/cạnh thì có các dạng
  - Cung/cạnh không có hướng.
  - Cung/cạnh có hướng.
  - Vòng/khuyên.

Ví dụ đồ thị sau



Hình 1.1: Đồ thị vô hướng



Hình 1.2: Đồ thị có hướng

## 1.2 PHÂN LOẠI ĐỒ THỊ

Đồ thị có 2 dạng:

- Đồ thị vô hướng là đồ thị mà gồm các đỉnh và các cung/cạnh/khuyên/vòng vô hướng. Ví dụ hình 1.1.
- Đồ thị có hướng là đồ thị mà gồm các đỉnh và các cung/cạnh/khuyên/vòng có hướng. Ví dụ hình 1.2.

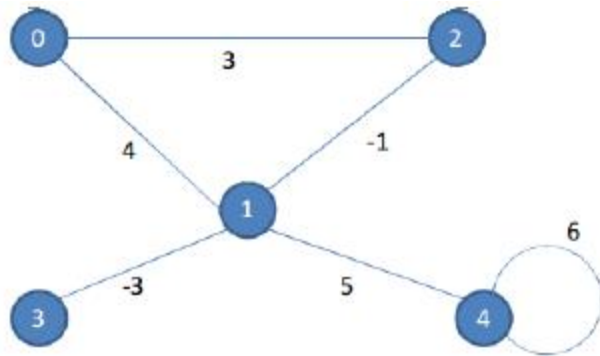
## 1.3 MA TRẬN KẼ

Trong lý thuyết đồ thị, người ta thường dùng ma trận kề để biểu diễn một đồ thị. Một giá trị  $a[i,j]$  trong ma trận (dòng  $i$ , cột  $j$  của ma trận  $A$ ) ứng với trọng số của

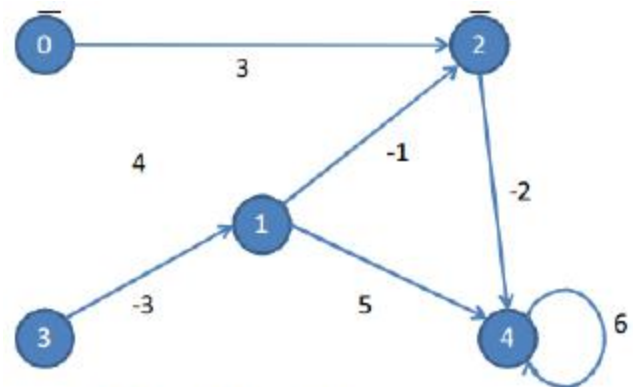


cung/cạnh từ đỉnh  $i$  đến đỉnh  $j$  trong đồ thị. Nếu giữa 2 đỉnh của đồ thị không có cung/cạnh thì phần tử  $a[i,j] = 0$ .

Ví dụ



**Đồ thị vô hướng**



**Đồ thị có hướng**

Đối với đồ thị vô hướng, ta có giá trị  $a[i,j] = a[j,i]$  (xem như có 2 cạnh có hướng nối từ đỉnh  $i$  đến đỉnh  $j$  và ngược lại từ  $j$  đến  $i$ ).

Đối với đồ thị có hướng, thông thường ta có giá trị  $a[i,j] \neq a[j,i]$ .

Ma trận kề của đồ thị vô hướng trên là:

Ma trận kề a	0	1	2	3	4
0	0	4	3	0	0
1	4	0	-1	-3	5
2	3	-1	0	0	0
3	0	-3	0	0	0
4	0	5	0	0	6

Ma trận kề của đồ thị có hướng trên là:

Ma trận kề a	0	1	2	3	4
0	0	0	3	0	0
1	0	0	-1	0	5
2	0	0	0	0	-2
3	0	-3	0	0	0
4	0	5	0	0	6

Ví dụ:

Đối với đồ thị trên (đồ thị vô hướng) ta có:  
Cạnh nối từ đỉnh 0 đến đỉnh 1 với trọng số là 4  $\rightarrow i = 0, j = 1$  và  $a[i,j] = a[0,1] = 4$ .  
Cạnh nối từ đỉnh 1 đến đỉnh 0 với trọng số là 4  $\rightarrow i = 1, j = 0$  và  $a[i,j] = a[1,0] = 4$ .

Ví dụ:

Đối với đồ thị trên (đồ thị có hướng) ta có:  
Cạnh nối từ đỉnh 0 đến đỉnh 2 với trọng số là 3  $\rightarrow i = 0, j = 2$ , và  $a[i,j] = a[0,2] = 3$ .  
Ngược lại ta không có cạnh nối từ đỉnh 2 đến đỉnh 0  $\rightarrow i = 2, j = 0$  và  $a[i,j] = a[2,0] = 0$ .

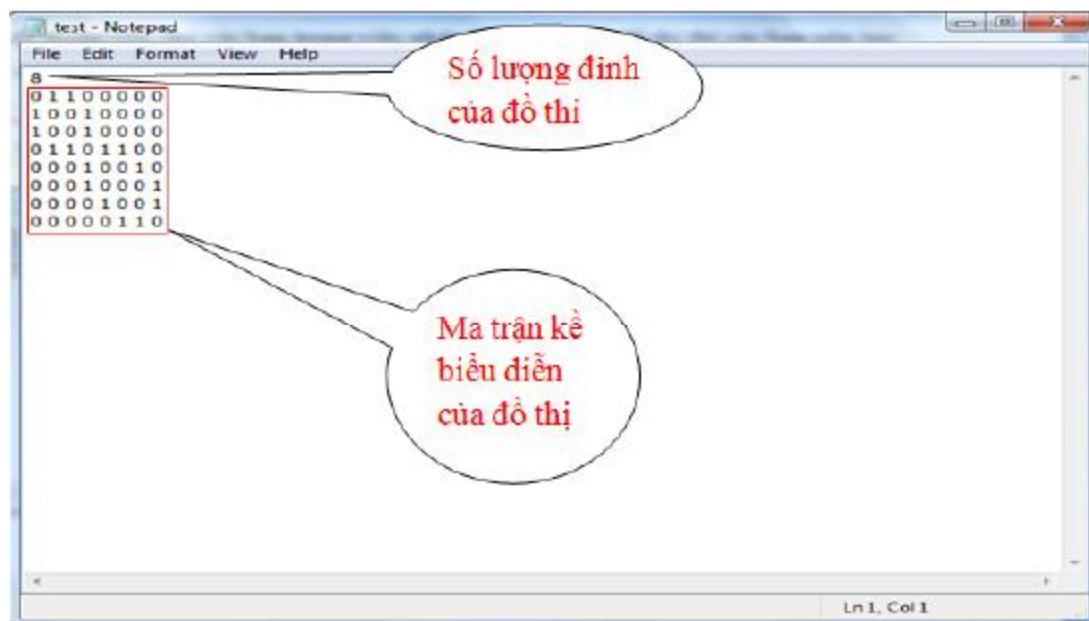
**Nhận xét:**

- Đối với đồ thị vô hướng/có hướng không có khuyên/vòng thì đường chéo chính của ma trận kề là 0 tức  $a[i,i] = 0$ .
- Đối với đồ thị vô hướng/có hướng có khuyên/vòng thì đường chéo chính của ma trận kề là khác 0 tức tồn tại tại  $i$  sao cho  $a[i,i] \neq 0$ .
- Đối với đồ thị vô hướng thì ma trận kề đối xứng qua đường chéo chính vì  $a[i,j] = a[j,i]$ .

## 1.4 NHẬP XUẤT MA TRẬN KÊ TỪ FILE

Hướng dẫn viết chương trình để:

- Đọc thông tin của đồ thị gồm số đỉnh  $n$ , ma trận kề từ một file nào đó, chẳng hạn như C:/test.txt và file đó có cấu trúc như sau:



- Xuất thông tin của đồ thị ra màn hình:
  - Dòng đầu tiên là số đỉnh  $n$  của đồ thị.
  - Những dòng tiếp theo ( $n$  dòng) là thông tin ma trận kề của đồ thị.
- Kiểm tra đồ thị được đọc vào từ file đó có hợp lệ không?
- Nếu thông tin đồ thị đó là hợp lệ, thì hãy cho biết đồ thị đó là có hướng hay vô hướng?

**Đầu tiên:** tạo một cấu trúc lưu trữ đồ thị như sau:

```
#define MAX 10 // định nghĩa giá trị MAX

#define inputfile "C:/test.txt" // định nghĩa đường dẫn tuyệt đối đến file chứa
thông tin của đồ thị

typedef struct GRAPH {
    int n; // số đỉnh của đồ thị
    int a[MAX][MAX]; // ma trận kề của đồ thị
}DOTHI;
```

**Giải thích:** struct dùng để khai một cấu trúc DOTHI (đồ thị) gồm có số đỉnh của đồ thị là *n*, và ma trận kề là mảng 2 chiều *a*. Cách sử dụng cấu trúc struct này hoàn toàn tương tự cách sử dụng đối với kiểu dữ liệu thông thường cơ bản như *int a*, *char c*, ...

Ví dụ: bạn muốn tạo một biến đồ thị thì bạn làm như sau:

```
DOTHI g;
```

Còn bạn muốn gán giá trị số đỉnh của đồ thị vào *g* (chẳng hạn đồ thị có 8 đỉnh) thì bạn làm như sau:

***g.n* = 8;** // giống như bạn sử dụng một biến *int*, *char*, *float* thông thường nhưng chỉ khác chỗ là có thêm ở đằng trước ***g***.

Điều này có nghĩa là nếu bạn muốn truy cập vào thành phần nào của đồ thị thì bạn dùng <ten\_bien\_thuoc\_kieu\_do\_thi> và dấu chấm.

Ví dụ: ***g.n*** //truy cập vào giá trị *n* của biến *g*

***g.a[i][j]*** //truy cập vào giá trị hàng *i* cột *j* của mảng *a* của biến *g*.

Còn nếu bạn muốn sử dụng các giá trị *n* (số đỉnh của đồ thị), *a[i][j]* (giá trị ma trận kề) thì bạn chỉ việc dùng <ten\_bien\_thuoc\_kieu\_do\_thi> và dấu chấm như trên.

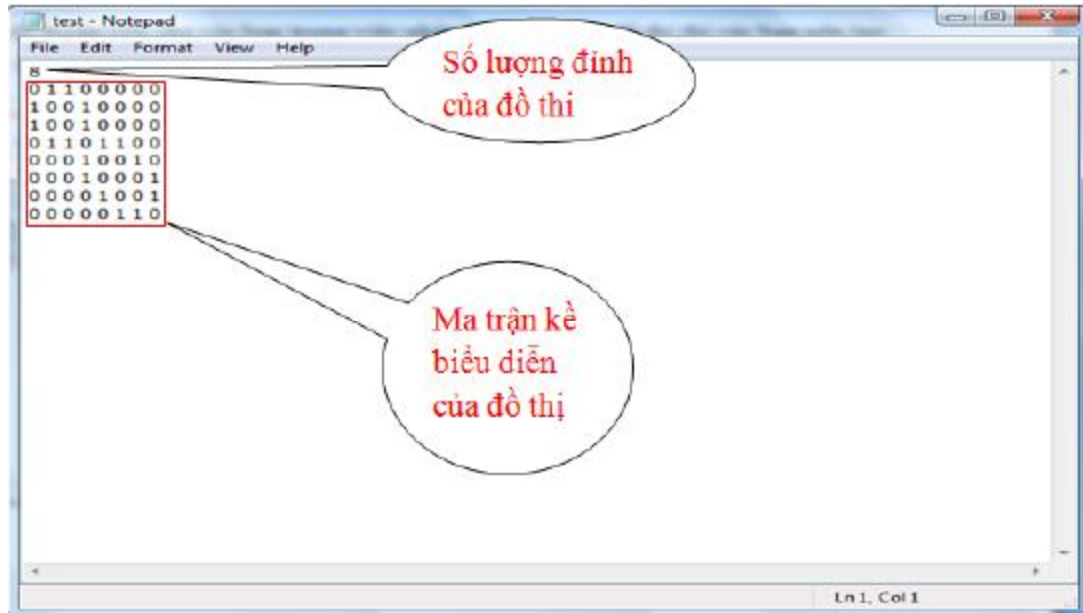
Ví dụ: lấy số đỉnh của đồ thị *g* gán vào biến *int h* và giá trị đầu tiên (chỉ số *i* = 0, *j* = 0) trong ma trận kề của đồ thị *g* vào biến *int k* thì làm như sau:

```
h = g.n;
```

```
k = g.a[0][0];
```

Do đó, để thuận tiện cho việc nhập thông tin của đồ thị thì các bạn nên tạo một cái file có cấu trúc như sau lưu trữ thông tin của đồ thị bạn cần test.

**Bước 1:** bạn mở NOTEPAD lên tạo một file có cấu trúc tổ chức như sau:



Sau đó bạn lưu lại với tên file bất kì gì đó chẳng hạn như bạn lưu với tên test.txt ở ổ C.

**Bước 2:** Lấy/đọc những thông tin đồ thị từ file bạn đã lưu ở trên (ví dụ C:/test.txt) gán vào một biến DOTHİ g. Việc đó được làm thông qua hàm DocMaTranKe như sau:

```
int DocMaTranKe(char TenFile[100], DOTHİ &g)
{
    FILE* f; // một biến FILE
    f = fopen(TenFile, "rt"); // mở một file có đường dẫn lưu ở biến TenFile
    if (f == NULL) // nếu file mở được thì biến f != NULL và file không mở được thì biến f = NULL
    {
        printf("Khong mo duoc file\n");
        return 0; // không đọc được file trả về kết quả 0
    }
    // Đọc giá trị đỉnh của đồ thị vào biến n của cấu trúc DOTHİ g
    fscanf(f, "%d", &g.n);
    // Đọc giá trị của ma trận a từ file vào (dùng 2 vòng for, dòng trước, cột sau để đọc từng phần tử của ma trận)
    // với a[i][j] là giá trị ma trận tại dòng i, cột j
    int i, j;
    for (i=0; i<g.n; i++)
```

```

{
    for (j=0; j<g.n; j++)
    {
        fscanf(f, "%d", &g.a[i][j]); // đọc từng giá trị và gán vào ma trận kề a
    }
}
// đóng file đã mở ở trên.
fclose(f);
return 1; // trả về kết quả 1 cho biết đã đọc file và xử lý nhập thông tin đồ thị
xong
}

```

Tuy nhiên, để xem kết quả đọc thông tin đồ thị từ file có đúng hay không? bạn viết hàm `XuatMaTranKe` như sau để xuất thông tin của đồ thị `g` hiện tại:

```

void XuatMaTranKe (DOTHI g)
{
    printf("So dinh cua do thi la %d\n", g.n);
    printf("Ma tran ke cua do thi la\n");
    for (int i = 0; i < g.n; i++)
    {
        printf ("\t");
        for (int j = 0; j < g.n; j++)
        {
            printf("%d ",g.a[i][j]);
        }
        printf("\n");
    }
}

```

**Bước 3:** Kiểm tra tính hợp lệ của ma trận kề nhập vào: nó có phải là biểu diễn của một trong 2 dạng đồ thị mà giáo viên đã trình bày cho các bạn không? Thì các bạn viết hàm **KiemTraMaTranHopLe** như sau:

```

int KiemTraMaTranKeHopLe(DOTHI g)
{
    int i;
    for (i=0; i<g.n; i++)
    {

```

```

    if (g.a[i][i] != 0) /* kiểm tra nếu tồn tại một giá trị trên đường chéo khác 0 nghĩa
là a[i][j] != 0 Thì trả về giá trị 0 (ma trận kề không hợp lệ) */
        return 0;
    }
    return 1; // trả về 1 nếu tất cả các giá trị trên đường chéo là 0
}

```

**Bước 4:** kiểm tra đồ thị bạn nhập vào có là **vô hướng** hay là **có hướng**, bạn dùng hàm sau KiemTraDoThiVoHuong sau.

```

int KiemTraDoThiVoHuong(DOTHI g)
{
    int i, j;
    for (i=0; i<g.n; i++)
    {
        for (j=0; j<g.n; j++)
        {
            if (g.a[i][j] != g.a[j][i]) /* kiểm tra nếu tồn tại một giá trị a[i][j] !=
a[j][i] thì tức ma trận kề không đối xứng, lúc đó đồ thị không phải là vô hướng. trả về
kết quả 0 (đồ thị không phải là vô hướng) */
                return 0;
        }
    }
    return 1; // trả về kết quả 1 nếu đồ thị là vô hướng
}

```

**Bước 5:** Code trong hàm main để gọi hàm các hàm tương ứng và chạy. Có thể làm như sau:

```

void main()
{
    DOTHI g;
    clrscr();
    if (DocMaTranKe(inputfile, g) == 1)
    {
        printf("Da lay thong tin do thi tu file thanh cong.\n\n");
        XuatMaTranKe(g);
        printf("Bam 1 phim bat ki de tien hanhkiem tra do thi ...\n\n");
        getch();
    }
}

```

```

if (KiemTraMaTranKeHopLe(g) == 1)
    printf ("Do thi hop le.\n");
else
    printf ("Do thi khong hop le.\n");

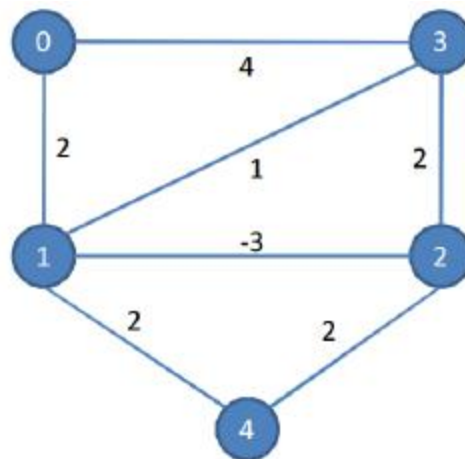
if (KiemTraDoThiVoHuong(g) == 1)
    printf ("Do thi vo huong.\n");
else
    printf ("Do thi co huong.\n");
}
getch();
}

```

## 1.5 ĐỒ THỊ LIÊN THÔNG

Đồ thị liên thông là đồ thị chỉ có 1 thành phần liên thông hay nói cách khác đồ thị liên thông là đồ thị mà ta lấy bất kì 2 đỉnh  $i, j$  nào đều có đường đi trực tiếp hay gián tiếp (thông qua các đỉnh khác trong đồ thị) nối từ đỉnh  $i$  đến đỉnh  $j$ .

Ví dụ:



**Hình 1.3: Đồ thị liên thông**

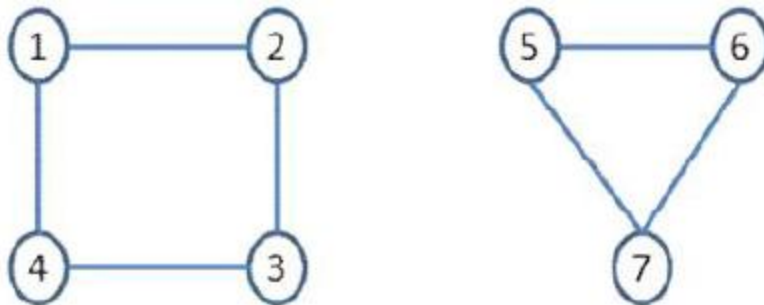
Đồ thị vô hướng hình 1.3 là đồ thị liên thông bởi vì lấy bất kì 2 đỉnh  $i, j$  nào đều có đường đi trực tiếp hay gián tiếp (thông qua các đỉnh khác trong đồ thị) nối từ đỉnh  $i$  đến đỉnh  $j$ .

Chẳng hạn, lấy 2 đỉnh: 1 và 2 thì có đường đi trực tiếp hay cung nối trực tiếp từ đỉnh 1 đến đỉnh 2.

Hoặc lấy 2 đỉnh: 0 và 2 thì có đường đi gián tiếp nối từ đỉnh 0 đến đỉnh 2 (tức  $0 \rightarrow 1 \rightarrow 2$  hoặc  $0 \rightarrow 3 \rightarrow 2$  hoặc  $0 \rightarrow 3 \rightarrow 1 \rightarrow 2$ ).

Trong trường hợp, **đồ thị không liên thông** thì sẽ tồn tại nhiều thành phần liên thông con.

Ví dụ:



**Hình 1.4: Đồ thị không liên thông**

Đồ thị hình 1.4 không phải là đồ thị liên thông vì ta lấy 2 đỉnh: 1 và 5 thì ta không có đường đi trực tiếp hay gián tiếp từ đỉnh 1 đến đỉnh 5.

Tuy nhiên, nếu ta chia đồ thị hình 1.4 thành 2 đồ thị con:

- Đồ thị con A gồm các đỉnh (1, 2, 3, 4) và
- Đồ thị con B gồm các đỉnh (5, 6, 7).

Khi đó bản thân đồ thị con A và đồ thị con B là những đồ thị liên thông. Nghĩa là lấy 2 đỉnh bất kì  $i, j$  trong 4 đỉnh (1, 2, 3, 4) của đồ thị con A đều có đường đi trực tiếp hoặc gián tiếp từ  $i$  đến  $j$ . Tương tự cho đồ thị con B.

Khi đó những đồ thị con A hay B được gọi là thành phần liên thông con của đồ thị ban đầu.

Như vậy, **thành phần liên thông con** của đồ thị là với bất kì hai đỉnh  $i, j$  nào thuộc về thành phần liên thông con đó đều có đường đi trực tiếp hoặc gián tiếp (thông qua các đỉnh khác trong đồ thị) nối từ đỉnh  $i$  đến đỉnh  $j$ .



## 1.6 THUẬT GIẢI ĐI TÌM CÁC THÀNH PHẦN LIÊN THÔNG CỦA ĐỒ THỊ

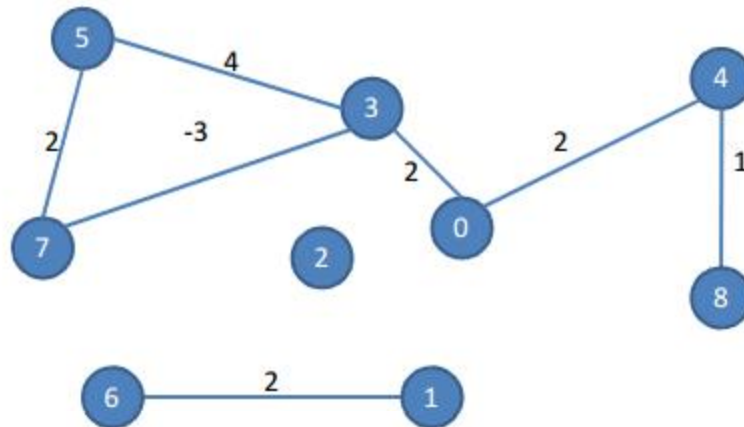
**Bước 1:** Tại thời điểm khởi tạo tất cả các đỉnh  $i$  chưa viếng thăm của đồ thị đều có  $Nhan[i] = 0$  và  $SoThanhPhanLT = 0$ .

**Bước 2:** Chọn 1 đỉnh  $i$  bất kỳ chưa được viếng thăm ( $Nhan[i] = 0$ ).

- Tăng giá trị của  $SoThanhPhanLT$  lên 1 ( $SoThanhPhanLT++$ ).
- Sử dụng hàm  $DiTimCacDinhLienThong$  để duyệt đỉnh  $i$  và tất cả các đỉnh  $j$  chưa được viếng thăm ( $Nhan[j] = 0$ ) có nối với đỉnh  $i$ . Kết thúc mỗi lần duyệt, ta được 1 thành phần liên thông. Để đánh dấu đỉnh này đã viếng thăm ta gán  $Nhan[j] = SoThanhPhanLT$ .

**Bước 3:** Lặp lại bước 2 cho đến khi không chọn được đỉnh  $i$  nào nữa.

Ví dụ thi hành thuật giải tìm các thành phần liên thông trên với đồ thị sau:

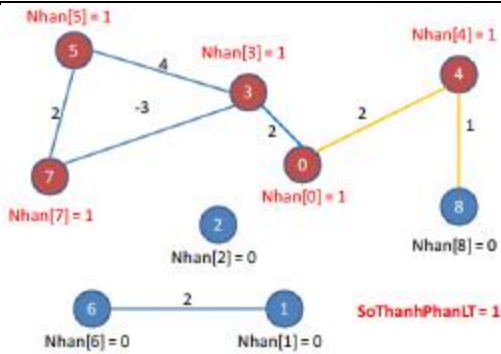


Hình 1.5: Tìm thành phần liên thông đồ thị

Bước	Trạng thái đồ thị	Xử lý
1	<p>Nhan[5] = 0, Nhan[3] = 0, Nhan[4] = 0, Nhan[7] = 0, Nhan[0] = 0, Nhan[2] = 0, Nhan[8] = 0, Nhan[6] = 0, Nhan[1] = 0. SoThanhPhanLT = 0.</p>	<p>Ban đầu mình chưa có gì, chưa làm gì.</p> <p>⇒ Tất cả các đỉnh của đồ thị có nhãn là 0. Tức gán <math>Nhan[i] = 0</math>. Điều này có nghĩa là đỉnh của đồ thị này chưa thuộc về một thành phần liên thông nào.</p> <p>⇒ Biến <math>SoThanhPhanLT = 0</math>.</p>

2		<p>Chọn một đỉnh bất kỳ có nhãn của nó là 0.</p> <p>Ở đây xin chọn đỉnh 5 có <math>Nhan[5] = 0</math>.</p> <p>Sau đó tăng <math>SoThanhPhanLT</math> lên 1 đơn vị, từ <math>SoThanhPhanLT = 1</math>.</p> <p>Gán nhãn của đỉnh 5 = <math>SoThanhPhanLT</math>, tức <math>Nhan[5] = 1</math>.</p>
3		<p>Tiến hành đi tìm các đỉnh khác mà có nối với đỉnh 5. Đó là đỉnh 3 và đỉnh 7.</p> <p>Ta thấy đỉnh 3 và đỉnh 7 có nhãn là 0 khác với nhãn của đỉnh 5 là 1.</p> <p>Tiến hành cập nhật nhãn của 2 đỉnh 3 và 7 bằng nhãn của đỉnh 5. Tức <math>Nhan[3] = 1</math>, <math>Nhan[7] = 1</math>.</p>
4		<p>Tiếp đến ta xét đỉnh 3, đỉnh 3 có 3 cạnh nối từ đỉnh 3 đến đỉnh 5, cạnh nối từ 3 – 7 và cạnh nối từ 3 – 0. Nhưng ở đây ta thấy nhãn của đỉnh 3, 5, 7 là 1. Tức <math>Nhan[3] = Nhan[5] = Nhan[7] = 1</math>. Do đó ta không cần tiến hành cập nhật giá trị nhãn của đỉnh 5 và 7. Vì 3 đỉnh 3, 5, 7 đã cùng thuộc một thành phần liên thông rồi. Tức là thành phần liên thông đầu tiên (1).</p> <p>Trong khi đó ta thấy đỉnh 0 có nhãn khác đỉnh 3, tức <math>Nhan[0] \neq Nhan[3]</math>. Ta tiến hành cập nhật nhãn của đỉnh 0 bằng nhãn của đỉnh 3. Tức gán <math>Nhan[0] = Nhan[3]</math>.</p>
5		<p>Tiếp đến ta xét đỉnh 0, đỉnh 0 có 2 cạnh nối từ đỉnh 0 đến đỉnh 3, cạnh nối từ 0 – 4. Nhưng ở đây ta thấy nhãn của đỉnh 3 là 1. Tức <math>Nhan[3] = Nhan[0] = 1</math>. Do đó ta không cần tiến hành cập nhật giá trị nhãn của đỉnh 3.</p> <p>Trong khi đó ta thấy đỉnh 4 có nhãn khác đỉnh 0, tức <math>Nhan[4] \neq Nhan[0]</math>. Ta tiến hành cập nhật nhãn của đỉnh 4 bằng nhãn của đỉnh 0. Tức gán <math>Nhan[4] = Nhan[0]</math>.</p>

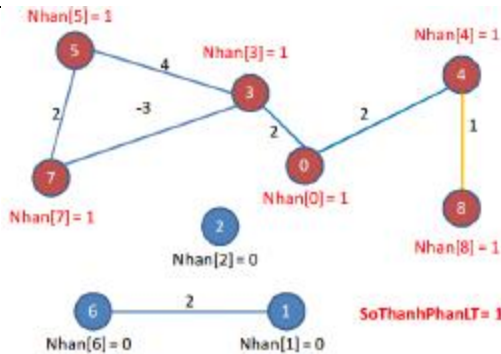
6



Tiếp đến ta xét đỉnh 4, đỉnh 4 có 2 cạnh nối từ đỉnh 8 đến đỉnh 4, cạnh nối từ 0 – 4. Nhưng ở đây ta thấy nhãn của đỉnh 0 là 1. Tức  $Nhan[0] = Nhan[4] = 1$ . Do đó ta không cần tiến hành cập nhật giá trị nhãn của đỉnh 0.

Trong khi đó ta thấy đỉnh 8 có nhãn khác đỉnh 4, tức  $Nhan[8] \neq Nhan[4]$ . Ta tiến hành cập nhật nhãn của đỉnh 8 bằng nhãn của đỉnh 4. Tức gán  $Nhan[8] = Nhan[4]$ .

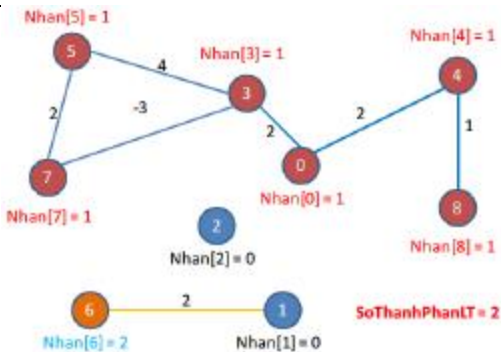
7



Tiếp đến ta xét đỉnh 8, đỉnh 8 có 1 cạnh nối từ đỉnh 8 đến đỉnh 4. Nhưng ở đây ta thấy nhãn của đỉnh 4 là 1. Tức  $Nhan[8] = Nhan[4] = 1$ . Do đó ta không cần tiến hành cập nhật giá trị nhãn của đỉnh 4.

Và tại thời điểm này ta không tìm được đỉnh mới để thay đổi nhãn tiếp. Do đó ta kết thúc thành phần liên thông đầu tiên (1) ở đây. Nó bao gồm các đỉnh 5, 7, 3, 0, 4, 8.

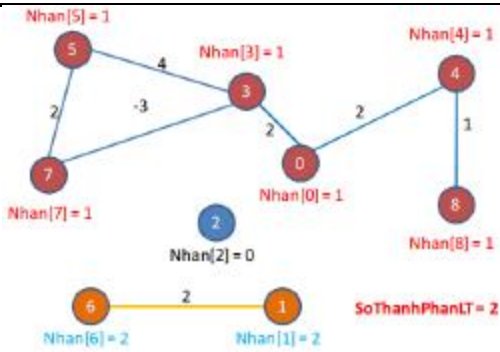
8



Tiếp đến ta xét đỉnh 6, đỉnh 6 có 1 cạnh nối từ đỉnh 6 đến đỉnh 1. Ta thấy đỉnh 1 có nhãn khác đỉnh 6, tức  $Nhan[1] \neq Nhan[6]$ . Ta tiến hành cập nhật nhãn của đỉnh 1 bằng nhãn của đỉnh 6. Tức gán  $Nhan[1] = Nhan[6]$ .



9

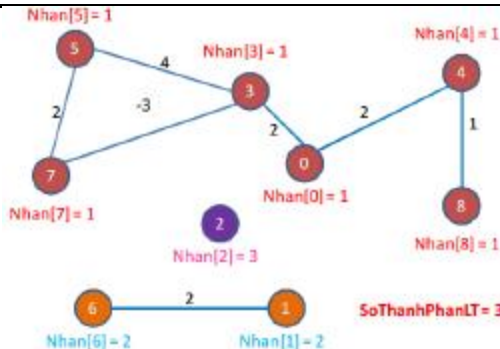


Tiếp đến ta xét đỉnh 1, đỉnh 1 có 1 cạnh nối từ đỉnh 1 đến đỉnh 6. Nhưng ở đây ta thấy nhãn của đỉnh 6 là 1. Tức  $Nhan[6] = Nhan[1] = 1$ . Do đó ta không cần tiến hành cập nhật giá trị nhãn của đỉnh 6.

Và tại thời điểm này ta không tìm được đỉnh mới để thay đổi nhãn tiếp. Do đó ta kết thúc thành phần liên thông thứ 2 ở đây. Nó bao gồm các đỉnh 6, 1.

Sau khi tìm xong thành phần liên thông thứ 2, ta thấy trong đồ thị vẫn còn đỉnh có nhãn 0. Tức đỉnh đó chưa thuộc về một thành phần liên thông nào. Do đó, ta quay lại bước đầu. Tiến hành chọn đỉnh có Nhãn là 0 rồi tăng SoThanhPhanLT lên 1 đơn vị (tức = 3), và đi tìm thành phần liên thông thứ 3. Ở đây chọn đỉnh 2 nhé. Gán  $Nhan[2] = SoThanhPhanLT$

10



Tiếp đến ta xét đỉnh 2, đỉnh 2 không có cạnh nào hết.

Và tại thời điểm này ta không tìm được đỉnh mới để thay đổi nhãn tiếp. Do đó ta kết thúc thành phần liên thông thứ 3 ở đây. Nó bao gồm các đỉnh 3.

Kết thúc quá trình, ta được 3 thành phần liên thông. Và căn cứ vào giá trị của các nhãn ta sẽ biết được đỉnh nào thuộc về thành phần liên thông nào.

Thành phần Liên Thông 1: bao gồm các đỉnh có giá trị nhãn là 1, tức đỉnh 0, 3, 4, 5, 7, 8.

Thành phần Liên Thông 2: bao gồm các đỉnh có giá trị nhãn là 2, tức đỉnh 1, 6.

Thành phần Liên Thông 3: bao gồm các đỉnh có giá trị nhãn là 3, tức đỉnh 2.

## 1.7 HƯỚNG DẪN TÌM ĐỒ THỊ LIÊN THÔNG

Viết chương trình để xác định số thành phần liên thông của một đồ thị vô hướng, và cho biết với mỗi thành phần liên thông bao gồm những đỉnh nào của đồ thị.

**Chú ý:** bạn đã làm được việc đọc thông tin của đồ thị từ một file nào đó vào chương trình của bạn rồi.

**Bước 1:** xét tính liên thông của đồ thị, viết một hàm DiTimCacDinhLienThong như sau để đi tìm các đỉnh j liên thông với đỉnh i trong đồ thị và gán nhãn cho những đỉnh liên thông j đó bằng nhãn của đỉnh i.

```
void DiTimCacDinhLienThong (DOTHI g, int nhan[MAX], int i)
{
    for (int j = 0; j < g.n; j++)
    {
        if (g.a[i][j] != 0 && nhan[j] != nhan[i]) // nếu tồn tại cạnh giữa đỉnh i và
        // đỉnh j, đồng thời nhãn của đỉnh j khác với nhãn của đỉnh i (nhãn thành phần liên
        // thông) thì thực hiện gán nhãn của đỉnh j = nhãn của đỉnh i và
        // DiTimCacDinhLienThong với đỉnh j
        {
            nhan[j] = nhan[i]; // gán nhãn cho đỉnh j
            DiTimCacDinhLienThong (g, nhan, j); // tiếp tục DiTimCacDinhLienThong với
            // đỉnh j và gán nhãn tương ứng tiếp.
        }
    }
}
```

**Bước 2:** Sau đó viết hàm XetLienThong như sau:

```
void XetLienThong(DOTHI g)
{
    int Nhan[MAX]; // tạo một mảng Nhãn để lưu lại nhãn của các đỉnh trong đồ thị
    int i;
    for (i=0; i<g.n; i++) // gán nhãn ban đầu cho tất cả các đỉnh của đồ thị g là 0
        Nhan[i] = 0;
    int SoThanhPhanLT = 0; // lưu lại số thành phần liên thông trong đồ thị g, ban
    // đầu là 0. Tức chưa có thành phần nào.
    // duyệt lần lượt tất cả các đỉnh và chọn đỉnh có nhãn là 0. Ta bắt đầu xét
    for (i=0; i<g.n; i++)
    {
        if (Nhan[i] == 0) // có một đỉnh trong đồ thị có nhãn là 0
        {
            SoThanhPhanLT ++; // tăng số thành phần liên thông lên
            Nhan[i] = SoThanhPhanLT; // gán nhãn cho đỉnh đó bởi
            SoThanhPhanLT
        }
    }
}
```

DiTimCacDinhLienThong(g, Nhan, i); // gọi hàm đi tìm các đỉnh liên thông với đỉnh i và gán nhãn cho nó. Hàm này được khai báo ở sau.

```

    }
}
printf("So thanh phan lien thong la %d\n", SoThanhPhanLT);
/*Tới đây là coi như bạn đã hoàn tất quá trình xét tính liên thông của đồ thị rồi
đó. Vấn đề còn lại là bạn chỉ cần code để hiển thị ra những thành phần liên thông của
nó ra. Gợi ý dựa vào mảng Nhãn */
for (i = 1; i <= SoThanhPhanLT; i++)
{
    printf("Thanh phan lien thong thu %d gom cac dinh ", i);
    /* các bạn code phần này để xử lý hiển thị ra các đỉnh trong thành phần
liên thông thứ i*/
    printf("\n");
}
}

```

**Bước 3:** Code trong hàm main để gọi hàm các hàm tương ứng và chạy. Có thể làm như sau:

```

void main()
{
    DOTH1 g;
    clrscr();
    if (DocMaTranKe("C:\\test2.txt",g) == 1)
    {
        printf("Da lay thong tin do thi tu file thanh cong.\n\n");
        XuatMaTranKe(g);
        printf("Bam 1 phim bat ki de bat dau xet tinh lien thong cua do thi ...\n\n");
        getch();
        XetLienThong(g);
    }
    getch();
}

```

# BÀI 2: ĐƯỜNG ĐI VÀ CHU TRÌNH EULER

Học xong bài này người học sẽ:

- Biết được chu trình euler và đường đi euler.
- Nắm được thuật toán Fleury.
- Biết cách thi hành thuật toán tìm chu trình và đường đi Euler.

## 2.1 MỘT SỐ ĐỊNH NGHĨA

Chu trình Euler là chu trình đi qua tất cả các cạnh của đồ thị, mỗi cạnh đi qua đúng một lần trong đó đỉnh đầu và đỉnh cuối trùng nhau (đồ thị có thể có các đỉnh cô lập). Tương tự với đường đi Euler, ngoại trừ điểm đầu và điểm cuối không trùng nhau.

Ví dụ:



**Hình 2. 1 Đường đi và chu trình Euler**

Đồ thị 1 có chu trình Euler, chẳng hạn như  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 5 \rightarrow 3 \rightarrow 1$ .

Còn đồ thị 2 không có chu trình Euler nhưng có đường đi Euler, chẳng hạn như  $4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow 5$ .

**Điều kiện cần** để tồn tại chu trình Euler là:

- Trong đồ thị vô hướng, bậc của tất cả các đỉnh phải là số chẵn.
- Trong đồ thị có hướng, bậc ngoài và bậc trong của mỗi đỉnh phải bằng nhau.

Trong một đồ thị, nếu không tìm ra chu trình Euler, vẫn có thể tồn tại đường đi Euler.

**Điều kiện cần** để tồn tại đường đi Euler trong trường hợp này là:

- Trong đồ thị vô hướng, tồn tại duy nhất hai đỉnh có bậc lẻ, tất cả các đỉnh còn lại là bậc chẵn.
- Trong đồ thị có hướng, bậc ngoài và bậc trong của mỗi đỉnh bằng nhau. Ngoại trừ một đỉnh tại đó bậc ngoài lớn hơn bậc trong 1 đơn vị (làm đỉnh bắt đầu) và một đỉnh tại đó bậc trong lớn hơn bậc ngoài 1 đơn vị (làm đỉnh kết thúc).

Đối với một số đồ thị thỏa mãn điều kiện cần của chu trình và đường đi Euler thì chưa chắc tồn tại chu trình Euler hay đường đi Euler. Do đó muốn chắc chắn có hay không? khi và chỉ khi chỉ ra được chu trình Euler hoặc đường đi Euler trong đồ thị đó.

Trường hợp sau chắc chắn tồn tại **chu trình Euler**:

- Đồ thị vô hướng, liên thông và bậc của tất cả các đỉnh trong đồ thị đều là bậc chẵn (không tồn tại đỉnh bậc lẻ nào).

Trường hợp sau chắc chắn tồn tại **đường đi Euler**:

- Đồ thị vô hướng, liên thông và tồn tại đúng 2 đỉnh bậc lẻ trong đồ thị.

## 2.2 THUẬT TOÁN FLEURY

---

Xuất phát từ 1 đỉnh nào đó của đồ thị  $G$  ( $G$  đã loại các đỉnh cô lập) ta đi theo các cạnh của nó một cách tùy ý chỉ cần tuân thủ 2 quy tắc sau:

- Cạnh này không phải là "cầu" (việc bỏ cạnh này không làm cho đồ thị mất tính liên thông).
- Xóa bỏ cạnh đã đi qua và đồng thời xóa cả những đỉnh cô lập tạo thành.

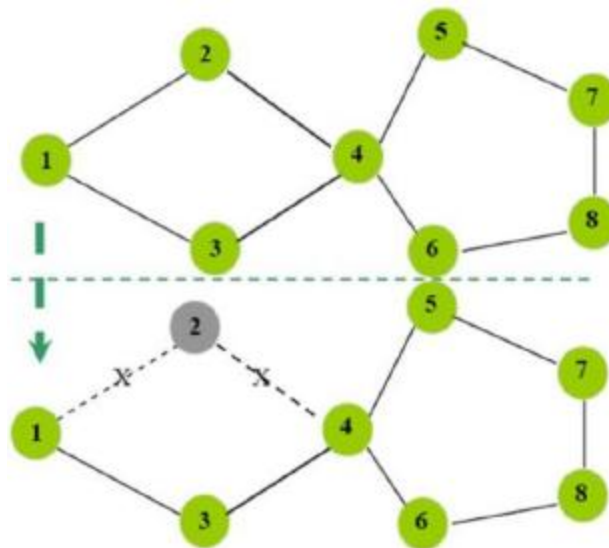
Nếu không tìm thấy cạnh nào thỏa mãn thì dừng thuật toán. Ngược lại, từ đỉnh được nối đến, lặp lại quá trình trên. Một cách tự nhiên, chu trình Euler sẽ quay lại điểm đầu, đường đi Euler sẽ đến điểm kết thúc.



**Lưu ý:** khi xóa bỏ cạnh đi qua và đỉnh cô lập, sẽ hình thành một đồ thị mới. Việc xét “cầu” tiếp theo sẽ dựa trên đồ thị mới này.

**Nhận xét:** thuật toán này tuy đơn giản nhưng thường không được sử dụng bởi vì việc xác định “cầu” trong đồ thị không đơn giản.

Ví dụ ta có đồ thị sau:



**Hình 2.2: Thuật toán Fleury**

Nếu xuất phát từ đỉnh 1, có hai cách đi tiếp: hoặc sang 2 hoặc sang 3. Giả sử ta sang 2 và xóa cạnh (1,2) vừa đi qua. Từ 2 chỉ có cách duy nhất là sang 4, nên cho dù (2,4) là cầu ta cũng phải đi qua và sau đó xóa luôn cạnh (2,4). Đến đây các cạnh còn lại của đồ thị được vẽ bằng nét liền, các cạnh đã đi bị xóa được vẽ bằng nét đứt.

Bây giờ đang đứng ở đỉnh 4 thì ta có 3 cách đi tiếp: sang 3, 5, hoặc 6. Vì (4,3) là cầu nên ta sẽ không đi theo cạnh (4,3) mà sẽ đi (4,5) hoặc (4,6). Nếu đi theo (4,5) và cứ tiếp tục đi như vậy, ta sẽ được chu trình Euler là (1,2,4,5,7,8,6,4,3,1). Còn đi theo (4,6) thì ta sẽ tìm được chu trình Euler là (1,2,3,6,8,7,5,4,3,1).

## 2.3 THUẬT TOÁN TÌM CHU TRÌNH VÀ ĐƯỜNG ĐI EULER

Từ nhận xét về thuật toán Fleury, chúng ta tìm thuật toán để thực thi một cách dễ hơn.

Cho  $G=(X,E)$  là một đồ thị gồm  $n$  đỉnh. Thuật toán tìm chu trình Euler xuất phát từ đỉnh  $u$  với  $u$  là đỉnh có bậc khác 0 như sau:

'tour' là một stack.

Find\_tour (u)

Với mỗi cạnh  $e = (u,v)$  trong  $E$

Loại cạnh  $e$  khỏi tập  $E$ .

Find\_tour (v).

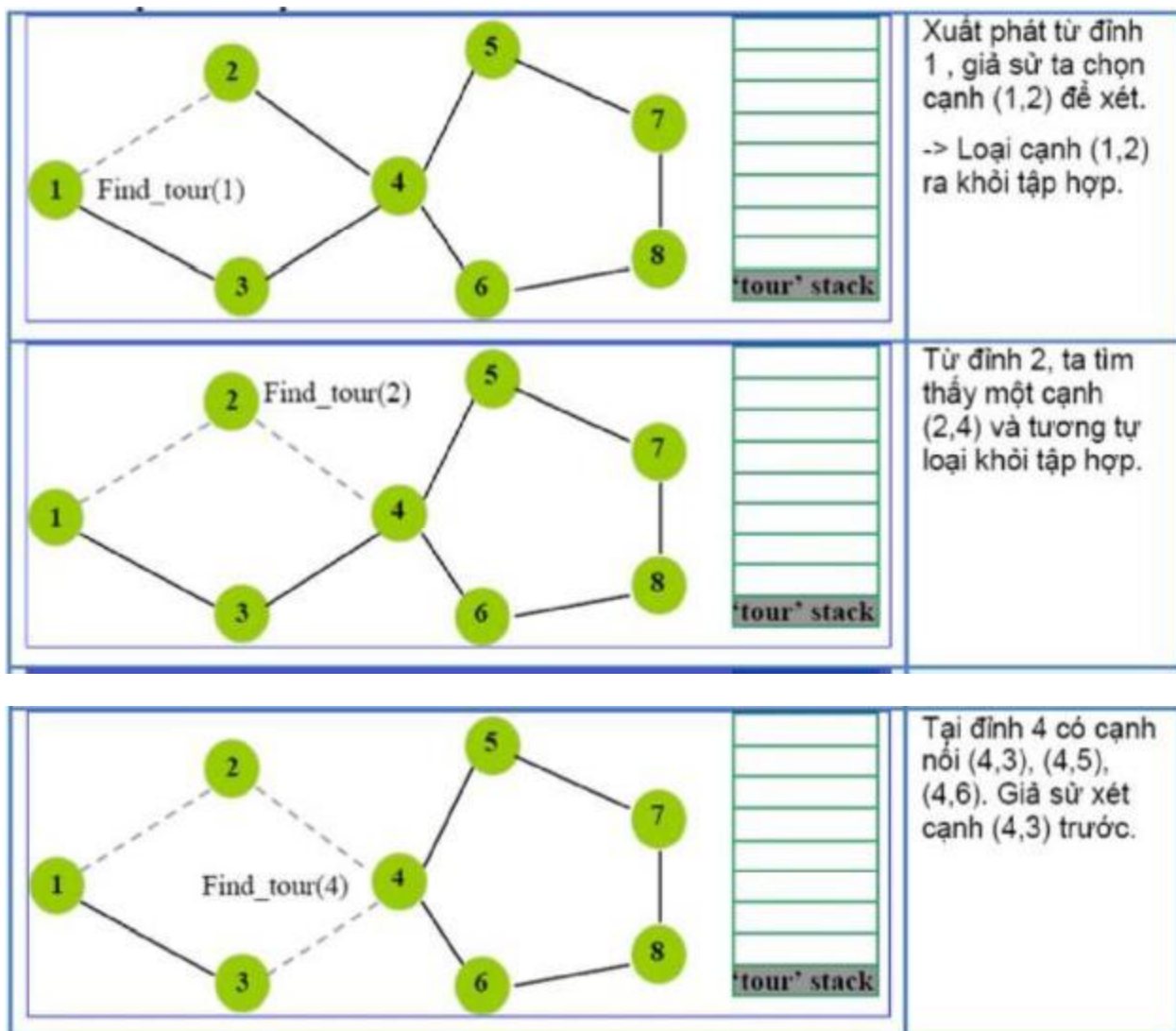
Cuối với mỗi

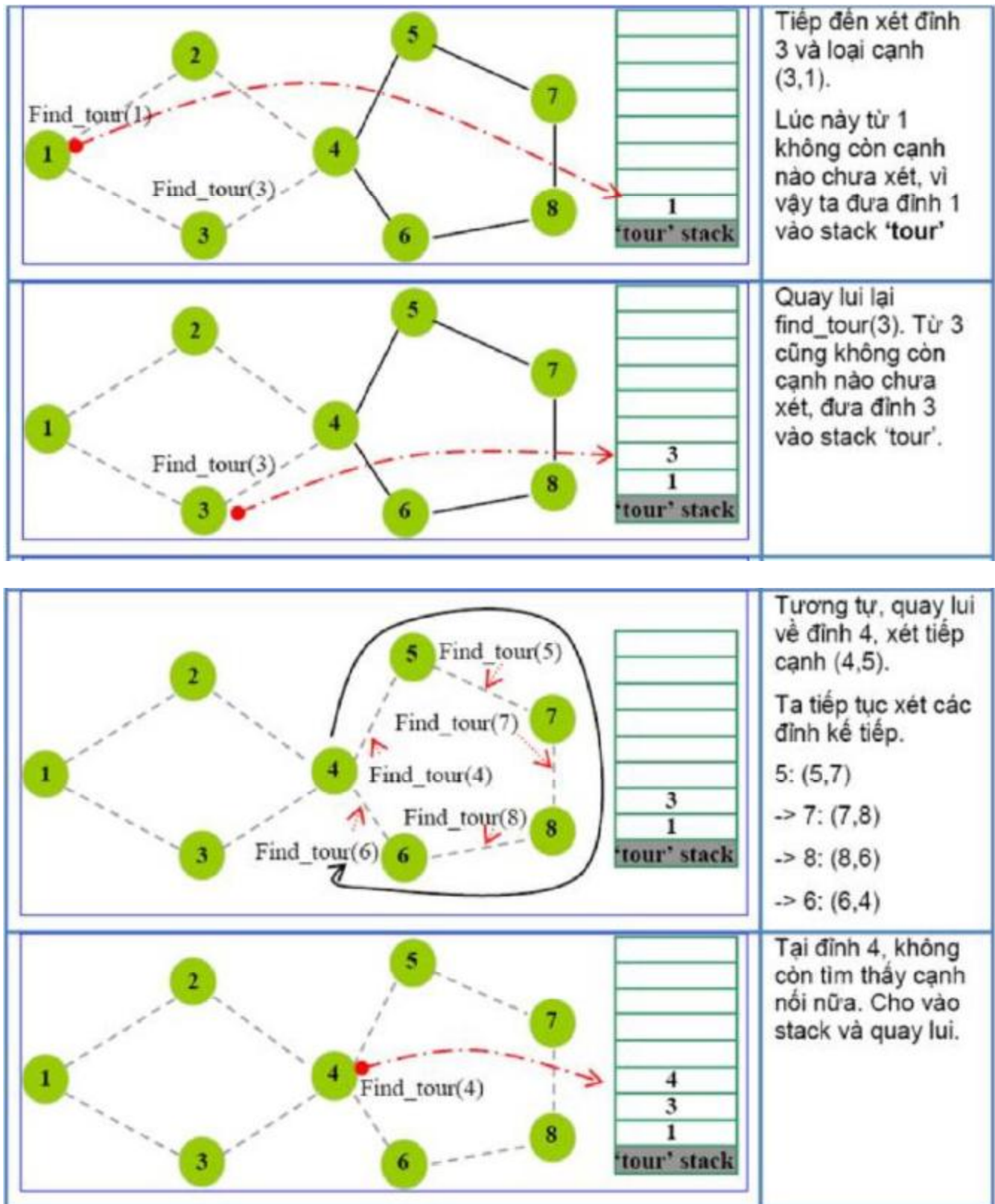
Thêm u vào stack 'tour'

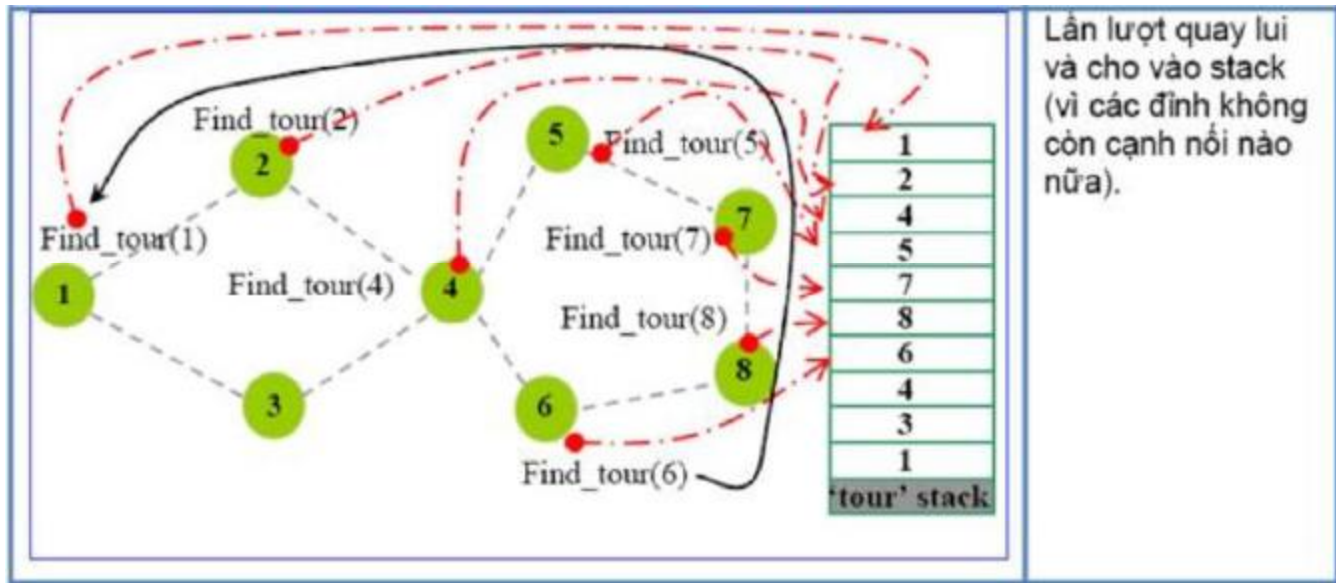
Cuối hàm.

**Lưu ý:** thuật toán này cũng có thể được sử dụng để tìm đường đi Euler bằng cách tạo một cạnh "giả" (dummy edge) nối điểm đầu và điểm cuối với nhau.

Ví dụ minh họa







Khi đó ta lấy theo nguyên tắc của stack là LIFO (Last In First Out) ta sẽ được một chu trình Euler là  $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 6 \rightarrow 4 \rightarrow 3 \rightarrow 1$ .

**Nhận xét:** Quá trình tìm đường đi Euler cũng như quá trình tìm chu trình Euler. Chỉ có điều khác nhau ở chỗ: đường đi Euler xuất phát từ đỉnh có bậc lẻ và kết thúc ở đỉnh bậc lẻ còn lại. Do đó kết quả trong stack sẽ có điểm đầu và điểm cuối khác nhau.

So sánh đường đi Euler và chu trình Euler trên đồ thị vô hướng

Tiêu chí	Chu trình Euler	Đường đi Euler
Đồ thị vô hướng	Có	Có
Đồ thị liên thông	Có	Có
Bậc của đỉnh	Không có đỉnh bậc lẻ, tức toàn bộ các đỉnh đều là bậc chẵn.	Có đúng 2 đỉnh bậc lẻ.
Điểm xuất phát	Bất kì đỉnh nào bậc chẵn của đồ thị.	Là một đỉnh bậc lẻ của đồ thị.
Điểm kết thúc	Đỉnh bậc chẵn của đồ thị.	Đỉnh bậc lẻ còn lại của đồ thị.

## 2.4 HƯỚNG DẪN TÌM CHU TRÌNH EULER

Viết chương trình để kiểm tra xem đồ thị của bạn có chu trình euler hay không? nếu có thì hãy xuất ra chu trình euler đó?

**Bước 1:** Tạo một cấu trúc Stack như sau để lưu lại các đỉnh trong chu trình euler:

```
struct STACK
{
    int array[100]; // lưu lại thứ tự các đỉnh trong chu trình euler có tối đa 100 đỉnh
    int size; // số lượng các đỉnh trong chu trình euler.
};
```

Để khởi tạo một stack rỗng, ta tạo một hàm khoitaoStack như sau:

```
void khoitaoStack (STACK &stack)
{
    stack.size = 0; // khởi tạo stack thì kích thước của stack bằng 0
}
```

Để đẩy một giá trị value vào stack, ta gọi hàm DayGiaTriVaoStack như sau:

```
void DayGiaTriVaoStack (STACK &stack, int value)
{
    if(stack.size + 1 >= 100) // nếu stack đã đầy thì không đẩy giá trị đó vào được
    vì kích thước của stack chỉ có chứa tối đa 100 phần tử.
        return; //thoát không thực hiện đẩy giá trị vào stack nữa
    stack.array[stack.size] = value; // đẩy giá trị value vào stack
    stack.size++; // tăng kích thước stack lên
}
```

**Bước 2:** Viết hàm **tìm đường đi** từ một đỉnh i đối với đồ thị g theo dạng đệ qui.

```
void TimDuongDi (int i, DOTHI &g, STACK &stack)
{
    for (int j = 0; j < g.n; j++)
    {
        if (g.a[i][j] != 0) // vì đồ thị vô hướng nên đối xứng do đó chỉ cần kiểm tra
        g.a[i][j]!=0 thôi, không cần kiểm tra g.a[j][i] != 0
        {
            g.a[i][j] = g.a[j][i] = 0; // loại bỏ cạnh nối đỉnh i tới đỉnh j khỏi đồ thị
            TimDuongDi(j,g,stack); // gọi đệ quy tìm đường đi tại đỉnh j
        }
    }
    DayGiaTriVaoStack(stack,i); // đẩy đỉnh i vào trong stack
}
```

**Bước 3:** Để kiểm tra đồ thị  $g$ , có chu trình euler không thì ta viết hàm kiểm tra chu trình Euler như sau:

```
int KiemTraChuTrinhEuler (DOTHI g)
{
    int i,j;
    int x = 0; // x là giá trị đỉnh bắt đầu xét chu trình euler, điều kiện x là đỉnh phải
    có bậc > 0

    /* bạn phải code chỗ này để tìm 1 đỉnh x bắt đầu tìm chu trình euler, đỉnh x này
    phải có bậc > 0*/

    DOTHI temp = g; // tạo ra một bản copy của đồ thị để thực hiện thuật toán, vì
    trong quá trình thi hành thuật toán ta có xóa cạnh nên ta tạo ra bản copy này để
    thực hiện việc xóa đó mà không ảnh hưởng đến đồ thị gốc (ban đầu).

    STACK stack; // tạo một stack như thuật toán đã trình bày

    khoitaoStack (stack); // ban đầu stack chưa có gì nên phải khởi tạo nó về 0.

    TimDuongDi(x,temp, stack); // bắt đầu tìm chu trình euler từ đỉnh x trong đồ thị
    temp, và thứ tự các đỉnh trong chu trình euler được lưu vào stack này.

    /* Bạn cần phải kiểm tra xem hàm TimDuongDi có tìm thấy chu trình euler trong
    đồ thị temp không? Bạn kiểm tra bằng cách nào? Vì đối với thuật toán của mình thì có
    xóa cạnh (tức cạnh đã đi qua rồi không đi lại được nữa), do đó để làm điều này, đơn
    giản bạn kiểm tra xem có tồn tại cung hay đường đi nào trong đồ thị temp không?
    Nếu tồn tại một cung hay cạnh thì đồ thị temp hay đồ thị ban đầu không có chu trình
    euler và trả về kết quả 0(sai_ tương ứng là không tìm thấy chu trình euler). Bạn viết
    code kiểm tra điều này*/

    /* Nếu có chu trình euler thì bắt buộc đỉnh đầu và đỉnh cuối trong stack phải
    bằng nhau. Điều này tương đương với việc đỉnh đầu và đỉnh cuối trùng nhau trong
    chu trình Euler. Nếu đỉnh đầu và đỉnh cuối không trùng nhau thì bạn trả về kết quả 0
    (sai_đồ thị temp hay đồ thị ban đầu không có chu trình euler). Bạn viết code kiểm tra
    điều này*/

    printf("\n Chu Trinh Euler : ");

    /* Nếu không rơi vào 2 trường hợp trên thì bạn đã có chu trình euler rồi đó, hãy
    code mà xuất ra. Dựa vào stack đó*/

    return 1; // trả về kết quả 1 tức có chu trình euler
}
```

**Bước 4:** Code trong hàm main để gọi hàm các hàm tương ứng và chạy. Có thể làm như sau:

```
void main()
{
    DOTHİ g;
    clrscr();
    if (DocMaTranKe(inputfile, g) == 1)
    {
        printf("Đã lấy thông tin đồ thị từ file thành công.\n\n");
        XuatMaTranKe(g);
        printf("Bấm 1 phím bất kỳ để bắt đầu xét tìm chu trình euler ...\n\n");
        getch();
        if (!KiemTraChuTrinhEuler(g))
        {
            printf("Không có chu trình Euler trong đồ thị của bạn\n");
            getch();
        }
    }
    getch();
}
```

## 2.5 HƯỚNG DẪN TÌM ĐƯỜNG ĐI

Viết chương trình để kiểm tra xem đồ thị của bạn có đường đi euler không? nếu có thì hãy xuất ra đường đi euler đó.

**Bước 1 và 2:** Làm tương tự ở phần hướng dẫn tìm chu trình euler.

**Bước 3:** Trường hợp nếu như đồ thị không tồn tại chu trình euler thì đồ thị đó có thể tồn tại đường đi euler không?. Để làm điều này tiến hành viết hàm KiemTraDuongDiEuler như sau:

```
int KiemTraDuongDiEuler (DOTHİ g)
{
    int i,j;

    int x = 0; // x là giá trị đỉnh bắt đầu xét đường đi euler, điều kiện x là đỉnh phải có bậc lẻ.
```

```
/* bạn phải code chỗ này để tìm 1 đỉnh x bắt đầu tìm chu trình euler, đỉnh x này phải có bậc lẻ*/
```

DOTH1 temp = g; // tạo ra một bản copy của đồ thị để thực hiện thuật toán, vì trong quá trình thi hành thuật toán ta có xóa cạnh nên ta tạo ra bản copy này để thực hiện việc xóa đó mà không ảnh hưởng đến đồ thị gốc (ban đầu).

```
STACK stack; // tạo một stack như thuật toán đã trình bày
```

```
khoitaoStack (stack); // ban đầu stack chưa có gì nên phải khởi tạo nó về 0.
```

TimDuongDi(x,temp, stack); // bắt đầu tìm đường đi euler từ đỉnh x trong đồ thị temp, và thứ tự các đỉnh trong đường đi euler được lưu vào stack này.

/\* Bạn cần phải kiểm tra xem hàm TimDuongDi có tìm thấy đường đi euler trong đồ thị temp không? Bạn kiểm tra bằng cách nào? Vì đối với thuật toán của mình thì có xóa cạnh (tức cạnh đã đi qua rồi không đi lại được nữa), do đó để làm điều này, đơn giản bạn kiểm tra xem có tồn tại cung hay đường đi nào trong đồ thị temp không? Nếu tồn tại một cung hay cạnh thì đồ thị temp hay đồ thị ban đầu không có đường đi euler và trả về kết quả 0 (sai\_ tương ứng là không tìm thấy đường đi euler). Bạn viết code kiểm tra điều này\*/

/\* Nếu có đường đi euler thì bắt buộc đỉnh đầu và đỉnh cuối trong stack không giống nhau. Điều này tương đương với việc đỉnh đầu và đỉnh cuối không trùng nhau trong đường đi Euler. Nếu đỉnh đầu và đỉnh cuối trùng nhau thì bạn trả về kết quả 0 (sai\_đồ thị temp hay đồ thị ban đầu không có đường đi euler). Bạn viết code kiểm tra điều này\*/

```
printf("\nĐường đi Euler : ");
```

/\* Nếu không rơi vào 2 trường hợp trên thì bạn đã có đường đi euler rồi đó, hãy code mà xuất ra. Dựa vào stack đó \*/

```
return 1;
```

```
}
```

**Bước 4:** Code trong hàm main để gọi hàm các hàm tương ứng và chạy. Có thể làm như sau:

```
void main()
{
    DOTH1 g;
    clrscr();
    if (DocMaTranKe(inputfile, g) == 1)
    {
```



```
printf("Da lay thong tin do thi tu file thanh cong.\n\n");
XuatMaTranKe(g);
printf("Bam 1 phim bat ki de bat dau xet tim chu trinh euler ...\n\n");
getch();
if (!KiemTraChuTrinhEuler(g))
{
    printf("Khong co chu trinh Euler trong do thi cua ban\n");
    printf("Bam 1 phim bat ki de bat dau xet tim duong di euler ...\n\n");
    getch();
    if (!KiemTraDuongDiEuler(g))
    {
        printf("Khong co duong di Euler trong do thi cua ban \n");
    }
}
}
getch();
}
```

# BÀI 3: TÌM KIẾM ĐƯỜNG ĐI TRÊN ĐỒ THỊ THEO CHIỀU SÂU VÀ CHIỀU RỘNG

Học xong bài này người học sẽ:

- Hiểu được định nghĩa như thế nào là đường đi.
- Hiểu được cách tìm kiếm đường đi trên đồ thị theo chiều rộng và chiều sâu.
- Cách thi hành thuật toán tìm kiếm đường đi theo chiều rộng và chiều sâu.

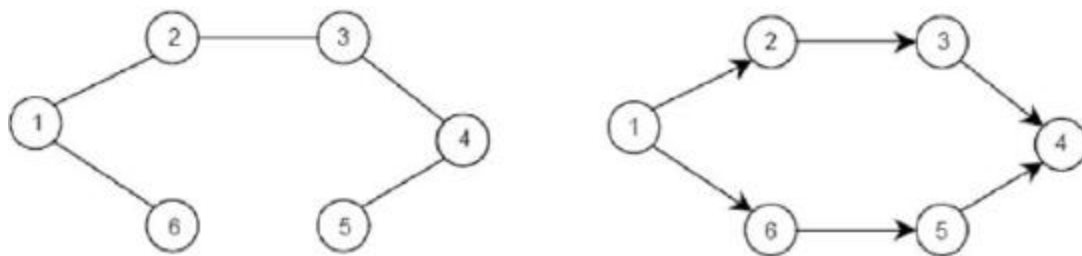
## 3.1 CÁC KHÁI NIỆM

Cho đồ thị  $G = (V, E)$ ,  $u$  và  $v$  là hai đỉnh của  $G$ . Một **đường đi** (path) độ dài  $k$  từ đỉnh  $u$  đến đỉnh  $v$  là dãy  $(u = x_0, x_1, x_2, \dots, x_k = v)$  thỏa mãn  $(x_i, x_{i+1}) \in E$  với  $\forall i (0 \leq i \leq k)$ .

Đường đi nói trên còn có thể biểu diễn bởi dãy các cạnh:  $(u = x_0, x_1), (x_1, x_2), \dots, (x_{k-1}, x_k = v)$ .

Đỉnh  $u$  được gọi là đỉnh đầu, đỉnh  $v$  được gọi là đỉnh cuối của đường đi. Đường đi có đỉnh đầu trùng với đỉnh cuối gọi là **chu trình** (Circuit), đường đi không có cạnh nào đi qua hơn 1 lần gọi là **đường đi đơn**, tương tự ta có khái niệm **chu trình đơn**.

Ví dụ xét một đồ thị vô hướng và một đồ thị có hướng dưới đây:



Hình 3.1: Đồ thị vô hướng

Trên cả hai đồ thị,  $(1,2,3,4)$  là đường đi đơn độ dài 3 từ đỉnh 1 đến đỉnh 4. Bởi  $(1,2)$ ,  $(2,3)$  và  $(3,4)$  đều là cạnh (hay cung).  $(1,6,5,4)$  không phải đường đi bởi  $(6,5)$  không phải là cạnh (hay cung).

Một bài toán quan trọng trong lý thuyết đồ thị là bài toán duyệt tất cả các đỉnh có thể đến được từ một đỉnh xuất phát nào đó. Vấn đề này đưa về một bài toán liệt kê mà yêu cầu của nó là không được bỏ sót hay lặp lại bất kỳ đỉnh nào. Chính vì vậy mà ta phải xây dựng những thuật toán cho phép **duyet một cách hệ thống** các đỉnh, những thuật toán như vậy gọi là những thuật toán **tìm kiếm trên đồ thị** và ở đây ta quan tâm đến hai thuật toán cơ bản nhất: **thuật toán tìm kiếm theo chiều sâu (DFS\_Depth First Search)** và **thuật toán tìm kiếm theo chiều rộng (BFS\_Breath First Search)** cùng với một số ứng dụng của chúng.

## 3.2 THUẬT TOÁN TÌM KIẾM THEO CHIỀU SÂU DFS\_DEPTH FIRST SEARCH

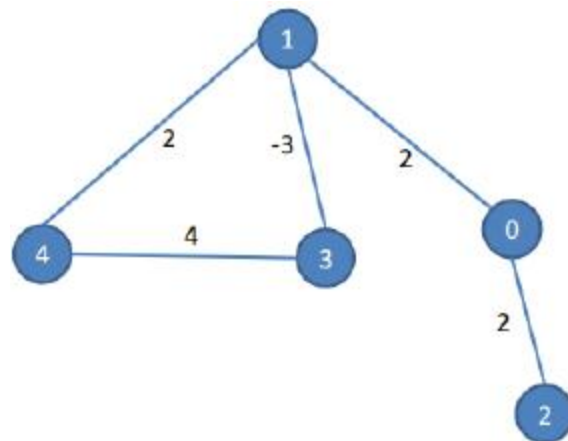
Tư tưởng của thuật toán có thể trình bày như sau: trước hết, mọi đỉnh  $x$  kề với  $S$  tất nhiên sẽ đến được từ  $S$ . Với mỗi đỉnh  $x$  kề với  $S$  đó thì tất nhiên những đỉnh  $y$  kề với  $x$  cũng đến được  $S$ .... Điều đó gợi ý cho ta viết một thủ tục đệ quy.

Cho  $G = (X,E)$  là một đồ thị gồm  $n$  đỉnh. Thuật toán tìm kiếm đường đi từ đỉnh  $S$  đến đỉnh  $F$  theo chiều sâu DFS dạng đệ quy như sau:

- **Bước 1:** Tạo ra 2 mảng 1 chiều `LuuVet` và `ChuaXet` với có kích thước là  $n$ .
- **Bước 2:** gán
  - `LuuVet[i] = -1`; //Vi ban đầu chưa chạy thuật toán nên các đỉnh  $i$  đều chưa có vết đi đến đó nên đặt giá trị là -1.
  - `ChuaXet[i] = 0`; //Vi ban đầu chưa chạy thuật toán nên các đỉnh  $i$  trong đồ thị  $g$  đều chưa được xét đến nên gán giá trị 0.
- **Bước 3:** Bắt đầu duyệt và xét 1 đỉnh  $v$  nào đó (bắt đầu bước này đầu tiên thì đỉnh  $v$  là đỉnh  $S$ ).
  - `ChuaXet[v] = 1`; //gán lại giá trị đỉnh  $v$  trong mảng `ChuaXet` là 1, điều này có nghĩa là đỉnh  $v$  đã và đang được xét hay duyệt đến theo thuật toán.

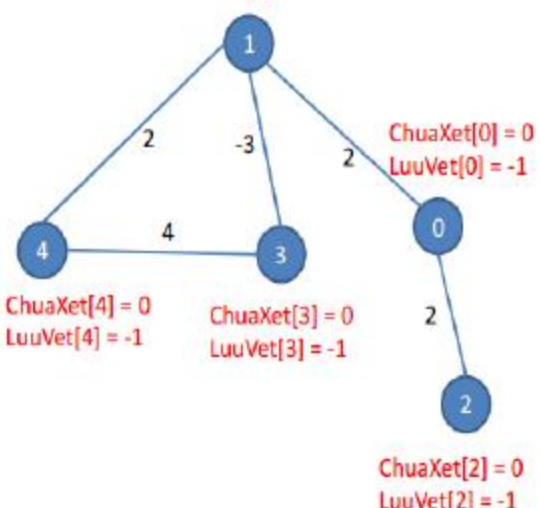
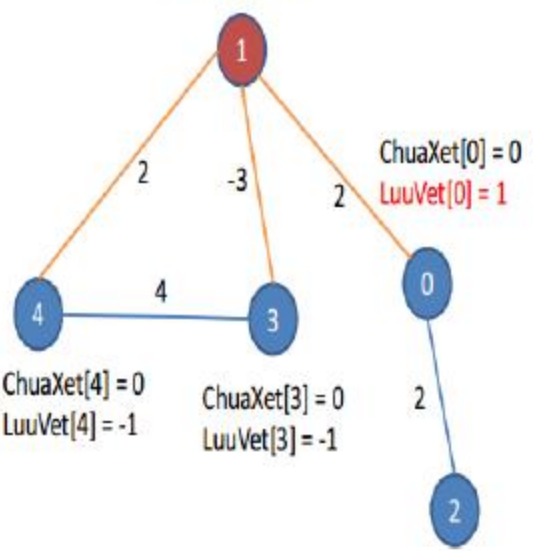
- Từ đỉnh  $v$  ta xem xét có đường đi đến đỉnh nào.
  - Nếu như tồn tại một đỉnh  $u$  mà đỉnh này chưa được xét đến tức  $ChuaXet[u] = -1$ .
    - Ta tiến hành  $LuuVet[u] = v$ ; //đánh dấu lại đỉnh  $u$  được đi đến từ đỉnh  $v$  trong quá trình duyệt theo thuật toán DFS.
    - Tiến hành nhảy đến đỉnh  $u$  ta xét và bắt đầu lại bước 3.
  - Ngược lại không tồn tại đỉnh nào thì dừng thuật toán.
- Bước 4: Sau khi tiến hành xong bước 3. Kiểm tra
  - if ( $ChuaXet[F] == 1$ ) //sau khi thuật toán duyệt xong mà đỉnh  $F$  được xét hay duyệt đến thì nhãn của nó = 1 điều này có nghĩa là có đường đi từ  $S \rightarrow F$ . Tiến hành xuất đường đi.
    - Dựa vào mảng  $LuuVet$  tiến hành truy vết mà xuất ra đường đi.
  - Ngược lại sau khi thuật toán duyệt xong mà đỉnh  $F$  chưa được xét hay duyệt đến thì nhãn của nó = 0 điều này có nghĩa là không có đường đi từ  $S \rightarrow F$ . Thông báo không có đường đi từ  $S \rightarrow F$ .

Ví dụ minh họa: Giả sử ta có đồ thị  $G = (X, E)$  gồm có 5 đỉnh như sau



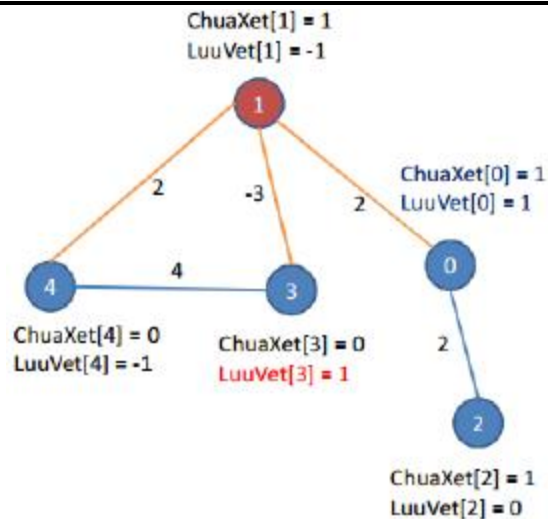
**Hình 3.2: Duyệt đồ thị DFS**

Thực hành thuật toán duyệt theo DFS tìm đường đi từ đỉnh 1 đến các đỉnh còn lại của đồ thị như sau

Bước	Trạng thái đồ thị	Xử lý
1	<p>ChuaXet[1] = 0 LuuVet[1] = -1</p>  <p>ChuaXet[4] = 0 LuuVet[4] = -1</p> <p>ChuaXet[3] = 0 LuuVet[3] = -1</p> <p>ChuaXet[2] = 0 LuuVet[2] = -1</p> <p>ChuaXet[0] = 0 LuuVet[0] = -1</p>	<p>Bước 1 và 2: Tạo ra 2 mảng 1 chiều <b>LuuVet</b> và <b>ChuaXet</b> với có kích thước là n.</p> <p>Và Gán:</p> <ul style="list-style-type: none"> <li>- <b>LuuVet[i] = -1;</b> // Vì ban đầu chưa chạy thuật toán nên các đỉnh i đều chưa có vết đi đến đó nên đặt giá trị là -1</li> <li>- <b>ChuaXet[i] = 0;</b> // Vì ban đầu chưa chạy thuật toán nên các đỉnh i trong đồ thị g đều chưa được xét đến nên gán giá trị 0</li> </ul>
2	<p>ChuaXet[1] = 1 LuuVet[1] = -1</p>  <p>ChuaXet[4] = 0 LuuVet[4] = -1</p> <p>ChuaXet[3] = 0 LuuVet[3] = -1</p> <p>ChuaXet[2] = 0 LuuVet[2] = -1</p> <p>ChuaXet[0] = 0 LuuVet[0] = 1</p>	<p>Bắt đầu duyệt và xét 1 đỉnh v nào đó (bắt đầu bước này đầu tiên thì đỉnh v là đỉnh S = 1)</p> <ul style="list-style-type: none"> <li>- <b>ChuaXet[1] = 1;</b> // gán lại giá trị đỉnh 1 trong mảng ChuaXet là 1, điều này có nghĩa là đỉnh 1 đã và đang được xét hay duyệt đến theo thuật toán.</li> <li>- Từ đỉnh 1 ta xem xét có đường đi đến đỉnh 0, 3, 4. Ở đây ta chọn theo thứ tự là đỉnh 0. <ul style="list-style-type: none"> <li>- <b>LuuVet[0] = 1;</b> // đánh dấu lại đỉnh 0 được đi đến từ đỉnh 1 trong quá trình duyệt theo thuật toán DFS</li> </ul> </li> <li>- Khi đó ta lại quay lại bước 3 và xét đỉnh 0.</li> </ul>

3	<p>ChuaXet[1] = 1 LuuVet[1] = -1</p> <p>ChuaXet[4] = 0 LuuVet[4] = -1</p> <p>ChuaXet[3] = 0 LuuVet[3] = -1</p> <p>ChuaXet[2] = 0 LuuVet[2] = 0</p> <p>ChuaXet[0] = 1 LuuVet[0] = 1</p>	<p>Bắt đầu duyệt và xét đỉnh 0</p> <ul style="list-style-type: none"> <li>- ChuaXet[0] = 1; // gán lại giá trị đỉnh 1 trong mảng ChuaXet là 1, điều này có nghĩa là đỉnh 0 đã và đang được xét hay duyệt đến theo thuật toán.</li> <li>- Từ đỉnh 0 ta xem xét có đường đi đến đỉnh 1, 2. Mà đỉnh 1 ta đã xét rồi (ChuaXet[1] = 1) nên ta chọn đỉnh 2 vì đỉnh 2 ta chưa xét (ChuaXet[2] = 0)             <ul style="list-style-type: none"> <li>- LuuVet[2] = 1; // đánh dấu lại đỉnh 2 được đi đến từ đỉnh 0 trong quá trình duyệt theo thuật toán DFS</li> </ul> </li> <li>- Khi đó ta lại quay lại bước 3 và xét đỉnh 2.</li> </ul>
4	<p>ChuaXet[1] = 1 LuuVet[1] = -1</p> <p>ChuaXet[4] = 0 LuuVet[4] = -1</p> <p>ChuaXet[3] = 0 LuuVet[3] = -1</p> <p>ChuaXet[2] = 1 LuuVet[2] = 0</p> <p>ChuaXet[0] = 1 LuuVet[0] = 1</p>	<p>Bắt đầu duyệt và xét đỉnh 2</p> <ul style="list-style-type: none"> <li>- ChuaXet[2] = 1;</li> <li>- Từ đỉnh 2 ta xem xét có đường đi đến đỉnh 0. Mà đỉnh 0 ta đã xét rồi (ChuaXet[0] = 1) và cũng ko còn có đỉnh nào từ đỉnh 2 mà chưa xét nên ta quay lại đỉnh trước đó (đỉnh 0) và xét tiếp đỉnh đó (đỉnh 0) (đệ qui_mảng lưu vết)</li> </ul>
5	<p>ChuaXet[1] = 1 LuuVet[1] = -1</p> <p>ChuaXet[4] = 0 LuuVet[4] = -1</p> <p>ChuaXet[3] = 0 LuuVet[3] = -1</p> <p>ChuaXet[2] = 1 LuuVet[2] = 0</p> <p>ChuaXet[0] = 1 LuuVet[0] = 1</p>	<p>Xét đỉnh 0</p> <ul style="list-style-type: none"> <li>- Từ đỉnh 0 ta xem xét có đường đi đến đỉnh 1, 2. Mà đỉnh 1, 2 ta đã xét rồi (ChuaXet[1] = 1, ChuaXet[2] = 1) và cũng ko còn có đỉnh nào từ đỉnh 0 mà chưa xét nên ta quay lại đỉnh trước đó (đỉnh 1) và xét tiếp đỉnh đó (đỉnh 1) (đệ qui_mảng lưu vết).</li> </ul>

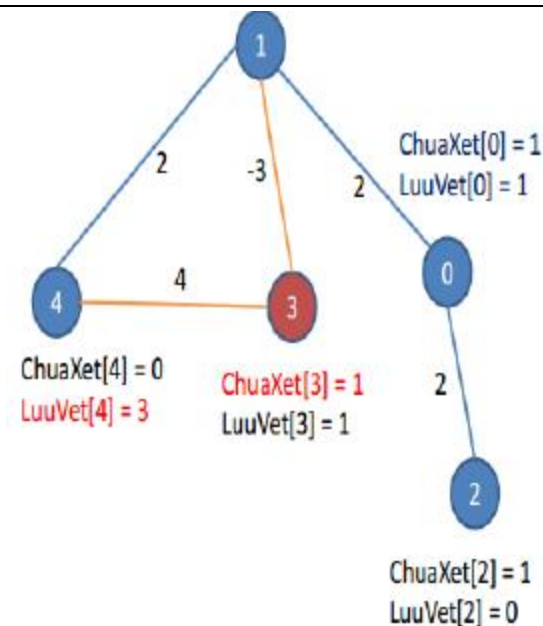
6



Xét đỉnh 1

- Từ đỉnh 1 ta xem xét có đường đi đến đỉnh 0, 3, 4. Mà đỉnh 0 ta đã xét rồi (**ChuaXet[0] = 1**) nên ta chọn đỉnh 3 (theo thứ tự) (**ChuaXet[3] = 0**)
  - **LuuVet[3] = 1;** // đánh dấu lại đỉnh 3 được đi đến từ đỉnh 1 trong quá trình duyệt theo thuật toán DFS
- Khi đó ta lại quay lại bước 3 và xét đỉnh 3

7



Bắt đầu duyệt và xét đỉnh 3

- **ChuaXet[3] = 1;** // gán lại giá trị đỉnh 3 trong mảng ChuaXet là 1, điều này có nghĩa là đỉnh 0 đã và đang được xét hay duyệt đến theo thuật toán.
- Từ đỉnh 3 ta xem xét có đường đi đến đỉnh 1, 4. Mà đỉnh 1 ta đã xét rồi (**ChuaXet[1] = 1**) nên ta chọn đỉnh 4 vì đỉnh 2 ta chưa xét (**ChuaXet[4] = 0**)
  - **LuuVet[4] = 3;** // đánh dấu lại đỉnh 4 được đi đến từ đỉnh 3 trong quá trình duyệt theo thuật toán DFS
- Khi đó ta lại quay lại bước 3 và xét đỉnh 4



8	<p>ChuaXet[1] = 1 LuuVet[1] = -1</p> <p>ChuaXet[4] = 1 LuuVet[4] = 3</p> <p>ChuaXet[3] = 1 LuuVet[3] = 1</p> <p>ChuaXet[0] = 1 LuuVet[0] = 1</p> <p>ChuaXet[2] = 1 LuuVet[2] = 0</p>	<p>Bắt đầu duyệt và xét đỉnh 4</p> <ul style="list-style-type: none"> <li>- <b>ChuaXet[4] = 1</b>; // gán lại giá trị đỉnh 4 trong mảng ChuaXet là 1, điều này có nghĩa là đỉnh 4 đã và đang được xét hay duyệt đến theo thuật toán.</li> <li>- Từ đỉnh 4 ta xem xét có đường đi đến đỉnh 1, 3. Mà đỉnh 1, 3 ta đã xét rồi (<b>ChuaXet[1] = 1</b>, <b>ChuaXet[4] = 1</b>) nên ta không còn đỉnh nào chưa xét mà đi từ 4. Do đó ta quay lại đỉnh đi tới nó là đỉnh 3 và xét đỉnh 3.</li> </ul>
9	<p>ChuaXet[1] = 1 LuuVet[1] = -1</p> <p>ChuaXet[4] = 1 LuuVet[4] = 3</p> <p>ChuaXet[3] = 1 LuuVet[3] = 1</p> <p>ChuaXet[0] = 1 LuuVet[0] = 1</p> <p>ChuaXet[2] = 1 LuuVet[2] = 0</p>	<p>Xét đỉnh 3</p> <ul style="list-style-type: none"> <li>- Từ đỉnh 3 ta xem xét có đường đi đến đỉnh 1, 4. Mà đỉnh 1, 4 ta đã xét rồi (<b>ChuaXet[1] = 1</b>, <b>ChuaXet[4] = 1</b>) nên ta không còn đỉnh nào chưa xét mà đi từ 3. Do đó ta quay lại đỉnh đi tới nó là đỉnh 1 và xét đỉnh 1.</li> </ul>
10	<p>ChuaXet[1] = 1 LuuVet[1] = -1</p> <p>ChuaXet[4] = 1 LuuVet[4] = 3</p> <p>ChuaXet[3] = 1 LuuVet[3] = 1</p> <p>ChuaXet[0] = 1 LuuVet[0] = 1</p> <p>ChuaXet[2] = 1 LuuVet[2] = 0</p>	<p>Xét đỉnh 1</p> <ul style="list-style-type: none"> <li>- Từ đỉnh 1 ta xem xét có đường đi đến đỉnh 0, 3, 4. Mà đỉnh 1, 4 ta đã xét rồi (<b>ChuaXet[0] = 1</b>, <b>ChuaXet[3] = 1</b>, <b>ChuaXet[4] = 1</b>) nên ta không còn đỉnh nào chưa xét mà đi từ 1. Do đó ta quay lại đỉnh đi tới nó mà đỉnh đi tới nó là -1 nên dừng thuật toán duyệt DFS ở đây.</li> </ul>

Sau khi duyệt DFS xong, bạn muốn biết đường đi từ đỉnh S (là đỉnh 1 trong trường hợp này) thì bạn căn cứ vào mảng LuuVet và ChuaXet.



LuuVet[0]	LuuVet[1]	LuuVet[2]	LuuVet[3]	LuuVet[4]
1	-1	0	1	3
ChuaXet[0]	ChuaXet[0]	ChuaXet[0]	ChuaXet[0]	ChuaXet[0]
1	1	1	1	1

Giả sử, bạn muốn tìm đường đi từ đỉnh 1 đến đỉnh 4. Thì đầu tiên căn cứ vào giá trị của mảng ChuaXet.

- Ta thấy  $\text{ChuaXet}[4] = 1 \rightarrow$  Có đường đi từ 1 đến 4.
- Ta sẽ truy vết đường đi dựa vào mảng LuuVet.
  - Ta thấy  $\text{LuuVet}[4] = 3 \rightarrow$  đỉnh 4 được đi đến từ đỉnh 3 trong đường đi duyệt theo DFS.
  - Rồi qua xét  $\text{LuuVet}[3] = 1 \rightarrow$  đỉnh 3 được đi đến từ đỉnh 1 trong đường đi duyệt theo DFS.
  - Ta thấy đỉnh 1 là đỉnh xuất phát đường đi nên dừng.
  - Vậy đường đi từ  $1 \rightarrow 4$  là  $4 \leftarrow 3 \leftarrow 1$ .
  - Tương tự ta có đường đi từ  $1 \rightarrow 2$  và  $1 \rightarrow 3$ .

### 3.3 HƯỚNG DẪN TÌM KIẾM ĐƯỜNG ĐI BẰNG PHƯƠNG PHÁP DUYỆT THEO CHIỀU SÂU (DFS)

Bạn có một đồ thị  $g$  (gồm đỉnh  $n$  và ma trận kề  $a$ ), bạn muốn tìm đường đi từ một đỉnh  $S$  (Start) đến một đỉnh  $F$  (Finish) trong đồ thị đó. Hãy viết chương trình tìm đường đi có thể theo thuật toán duyệt theo chiều sâu (DFS). Nếu tìm có đường đi từ đỉnh  $S$  (Start) đến đỉnh  $F$  (Finish) trong đồ thị  $g$  này thì bạn xuất ra đường đi, còn nếu không có đường đi thì bạn thông báo không có đường đi từ đỉnh  $S$  (Start) đến đỉnh  $F$  (Finish).

**Bước 1:** Tạo 1 mảng 1 chiều  $\text{LuuVet}$  dùng để lưu vết đường đi từ  $S \rightarrow F$ , và 1 mảng 1 chiều khác với tên là  $\text{ChuaXet}$  dùng để đánh dấu đỉnh nào trong đồ thị đã xét rồi, đỉnh nào chưa xét trong quá trình tìm đường đi từ  $S \rightarrow F$ .

```
int LuuVet[MAX]; // LuuVet[i] = đỉnh liên trước i trên đường đi từ S → i
```

int ChuaXet[MAX]; // ChuaXet[i] = 0 là đỉnh i chưa được xét đến trong quá trình tìm đường đi, còn ChuaXet[i] = 1 là đỉnh i được xét đến rồi trong quá trình tìm đường đi.

**Bước 2: Code phần duyệt theo chiều sâu (DFS) theo dạng đệ qui.**

```
void DFS(int v, GRAPH g) // hàm xét tại đỉnh v của đồ thị g
```

```
{
```

ChuaXet[v] = 1; // gán lại giá trị đỉnh v trong mảng ChuaXet là 1, điều này có nghĩa là đỉnh v đã và đang được xét hay duyệt đến theo thuật toán.

```
int u;
```

```
for(u = 0; u < g.n ; u++)
```

```
{
```

if(g.a[v][u] != 0 && ChuaXet[u] == 0) // Xem xét có cạnh nào nối từ đỉnh v đến đỉnh u trong đồ thị g không (điều này tương ứng với g.a[v][u] != 0) và đỉnh u đã được xét hay duyệt đến hay chưa? Nếu có cạnh nối từ đỉnh v đến đỉnh u và đỉnh u chưa được xét hay duyệt đến thì tiến hành duyệt đỉnh u

```
{
```

LuuVet[u] = v; // đánh dấu lại đỉnh u được đi đến từ đỉnh v trong quá trình duyệt theo thuật toán DFS

```
DFS(u,g); // tiến hành nhảy tới đỉnh u và xét duyệt đỉnh u
```

```
}
```

```
}
```

```
}
```

**Bước 3:** Trước khi tiến hành duyệt DFS để tìm đường đi  $S \rightarrow F$  thì cần phải khởi tạo các giá trị thích hợp cho các mảng LuuVet và ChuaXet. Quá trình đó như sau:

```
void duyetttheoDFS (int S, int F, GRAPH g) // hàm này dùng để tìm đường đi từ S → F trong đồ thị g theo thuật toán DFS
```

```
{
```

```
int i;
```

```
// khởi tạo lại các giá trị thích hợp cho mảng LuuVet và ChuaXet
```

/\* các bạn tự viết code khởi tạo các giá trị cho mảng LuuVet và ChuaXet. Gợi ý vì ban đầu chưa chạy thuật toán nên các đỉnh i đều chưa có vết đi đến đó nên đặt giá trị -1. Và ban đầu chưa chạy thuật toán nên đỉnh i trong đồ thị g đều chưa được xét nên giá trị là 0 \*/

```
DFS(S,g); // tiến hành thuật toán duyệt theo chiều sâu
```

```
// xuất đường đi
```

if (ChuaXet[F] == 1) // sau khi thuật toán duyệt xong mà đỉnh F được xét hay duyệt đến thì nhãn của nó = 1 điều này có nghĩa là có đường đi từ  $S \rightarrow F$ . Tiến hành xuất đường đi.

```
{
```

```
    printf("Duong di tu dinh %d den dinh %d la: \n\t",S,F);
```

```
    i = F;
```

```
    printf("%d ", F);
```

/\* các bạn tự viết code xuất ra đường đi từ  $F \leftarrow S$  hén. Gợi ý dựa vào mảng LuuVet để tiến hành truy vết ra đường đi từ  $S \rightarrow F$ \*/

```
}
```

Else // sau khi thuật toán duyệt xong mà đỉnh F chưa được xét hay duyệt đến thì nhãn của nó = 0 điều này có nghĩa là không có đường đi từ  $S \rightarrow F$ . Thông báo không có đường đi từ  $S \rightarrow F$

```
{
```

```
    printf("Khong co duong di tu dinh %d den dinh %d \n",S,F);
```

```
}
```

```
}
```

**Bước 4:** Code trong hàm main để gọi hàm các hàm tương ứng và chạy. Có thể làm như sau:

```
void main()
```

```
{
```

```
    DOTH1 g;
```

```
    clrscr();
```

```
    if (DocMaTranKe(inputfile, g) == 1)
```

```
    {
```

```
        printf("Da lay thong tin do thi tu file thanh cong.\n\n");
```

```

    XuatMaTranKe(g);
    printf("Bam 1 phim bat ki de bat dau duyett theo DFS ...\n\n");
    getch();
    duyetttheoDFS(0,2,g);
}
getch();
}

```

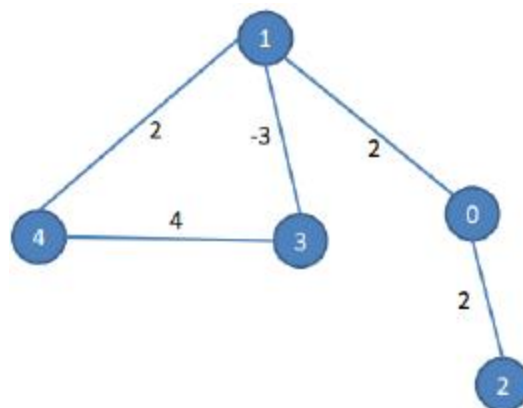
### 3.4 THUẬT TOÁN TÌM KIẾM THEO CHIỀU RỘNG BFS (BREADTH FIRST SEARCH)

Cho  $G = (X, E)$  là một đồ thị gồm  $n$  đỉnh. Thuật toán tìm kiếm đường đi từ đỉnh  $S$  đến đỉnh  $F$  theo chiều rộng BFS dùng Queue như sau:

- **Bước 1:** Tạo ra 2 mảng 1 chiều `LuuVet` và `ChuaXet` với có kích thước là  $n$ .
- **Bước 2:** Gán
  - `LuuVet[i] = -1`; //Vi ban đầu chưa chạy thuật toán nên các đỉnh  $i$  đều chưa có vết đi đến đó nên đặt giá trị là  $-1$ .
  - `ChuaXet[i] = 0`; //Vi ban đầu chưa chạy thuật toán nên các đỉnh  $i$  trong đồ thị  $g$  đều chưa được xét đến nên gán giá trị  $0$ .
- **Bước 3:** xây dựng với QUEUE các hàm cần thiết như
  - `KhoiTaoQueue`.
  - `DayGiaTriVaoQueue`.
  - `LayGiaTriRaKhoiQueue`.
  - `KiemTraQueueRong`.
- **Bước 4:** Bắt đầu duyệt và xét 1 đỉnh  $v$  nào đó (bắt đầu bước này đầu tiên thì đỉnh  $v$  là đỉnh  $S$ ).
  - `KhoiTaoQueue(Q)`; //khởi tạo hàng đợi vì ban đầu thuật toán hàng đợi ta chưa có gì.
  - `DayGiaTriVaoQueue(Q,v)`; //Đẩy đỉnh  $v$  vào hàng đợi theo.

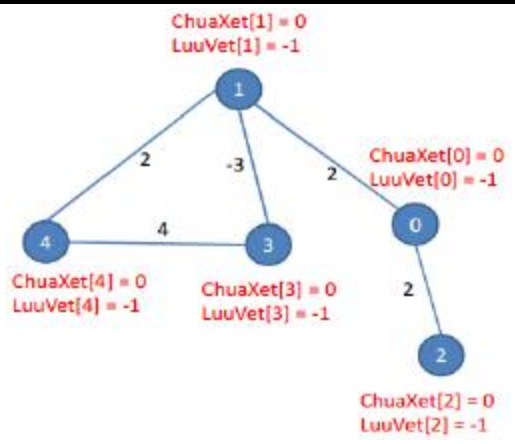
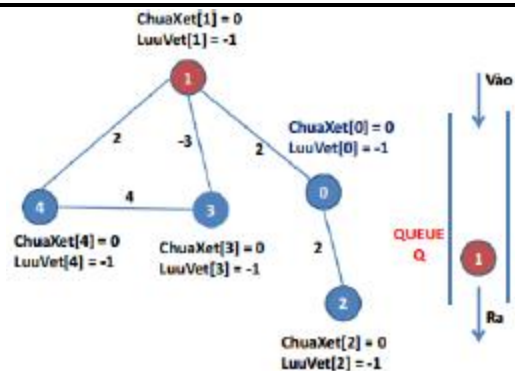
- Trong khi hàm đợi Q chưa rỗng thì tiến hành các bước sau
    - LayGiaTriRaKhoiQueu(Q,v); //lấy phần tử đầu tiên trong hàng đợi ra và gán giá trị đó vào v.
    - ChuaXet[v] = 1; //gán lại giá trị đỉnh v trong mảng ChuaXet là 1, điều này có nghĩa là đỉnh v đã và đang được xét hay duyệt đến theo thuật toán.
    - Từ đỉnh v ta xem xét có đường đi đến đỉnh nào.
      - Nếu như tồn tại một đỉnh u mà đỉnh này chưa được xét đến tức ChuaXet[u] = -1.
        - Ta tiến hành LuuVet[u] = v; //đánh dấu lại đỉnh u được đi đến từ đỉnh v trong quá trình duyệt theo thuật toán BFS.
        - DayGiaTriVaoQueue(Q,u); //đẩy đỉnh u vào trong hàng đợi Q.
  - Ngược lại hàng đợi Q rỗng thì dừng thuật toán.
- Bước 5: sau khi tiến hành xong bước 4. Kiểm tra
- if (ChuaXet[F] == 1) //sau khi thuật toán duyệt xong mà đỉnh F được xét hay duyệt đến thì nhãn của nó = 1 điều này có nghĩa là có đường đi từ  $S \rightarrow F$ . Tiến hành xuất đường đi.
    - Dựa vào mảng LuuVet tiến hành truy vết để xuất ra đường đi.
  - Ngược lại sau khi thuật toán duyệt xong mà đỉnh F chưa được xét hay duyệt đến thì nhãn của nó = 0 điều này có nghĩa là không có đường đi từ  $S \rightarrow F$ . Thông báo không có đường đi từ  $S \rightarrow F$ .

Ví dụ minh họa: Giả sử ta có đồ thị  $G = (X,E)$  gồm có 5 đỉnh như sau

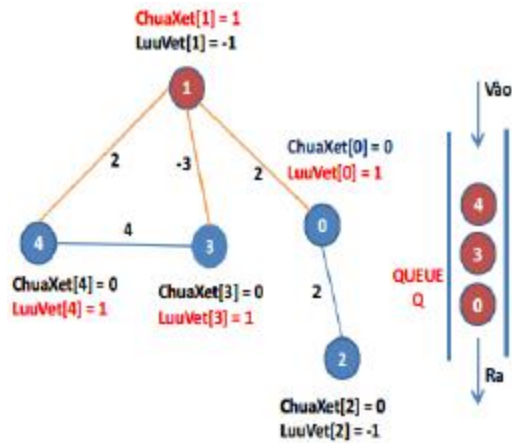


**Hình 3.3:** Duyệt đồ thị BFS

Thi hành thuật toán duyệt theo BFS tìm đường đi từ đỉnh 1 đến các đỉnh còn lại của đồ thị như sau

Bước	Trạng thái đồ thị	Xử lý
1		<p>Bước 1 và 2: Tạo ra 2 mảng 1 chiều <b>LuuVet</b> và <b>ChuaXet</b> với có kích thước là n.</p> <p>Và Gán:</p> <ul style="list-style-type: none"> <li>- <b>LuuVet[i] = -1;</b> // Vì ban đầu chưa chạy thuật toán nên các đỉnh i đều chưa có vết đi đến đó nên đặt giá trị là -1</li> <li>- <b>ChuaXet[i] = 0;</b> // Vì ban đầu chưa chạy thuật toán nên các đỉnh i trong đồ thị g đều chưa được xét đến nên gán giá trị 0</li> </ul>
2		<p>Xây dựng với <b>QUEUE</b> các hàm cần thiết như</p> <ul style="list-style-type: none"> <li>- <b>KhởiTaoQueue.</b></li> <li>- <b>DayGiaTriVaoQueue</b></li> <li>- <b>LayGiaTriRaKhoiQueue</b></li> <li>- <b>KiemTraQueueRong.</b></li> </ul> <p>Đẩy đỉnh 1 (tức là đỉnh S) vào hàng đợi Q</p>

3



Kiểm tra hàng đợi Queue Q có rỗng ko.

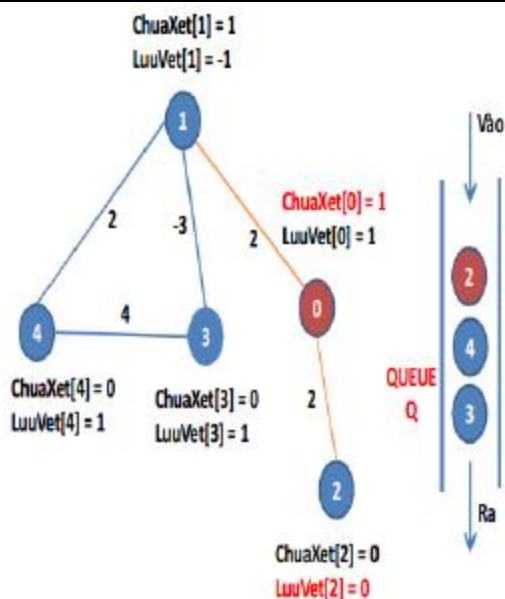
Hàng đợi Queue Q ko rỗng.

Lấy điểm dưới cùng ra. Là đỉnh 1 và tiến hành xét đỉnh 1.

- `ChuaXet[1] = 1;` // gán lại giá trị đỉnh 1 trong mảng ChuaXet là 1, điều này có nghĩa là đỉnh 1 đã và đang được xét hay duyệt đến theo thuật toán.
- Từ đỉnh 1 ta xem xét có đường đi đến đỉnh 0, 3, 4. Và 3 đỉnh này đều có giá trị ChuaXet là 0.
  - `Đẩy 3 đỉnh 0, 3, 4 vào Queue Q.`
  - `LuuVet[0] = 1; LuuVet[3] = 1; LuuVet[4] = 1;` // đánh dấu lại đỉnh 0, 3, 4 được đi đến từ đỉnh 1 trong quá trình duyệt theo thuật toán BFS

Sau đó ta quay lại đầu bước này

4



Kiểm tra hàng đợi Queue Q có rỗng ko.

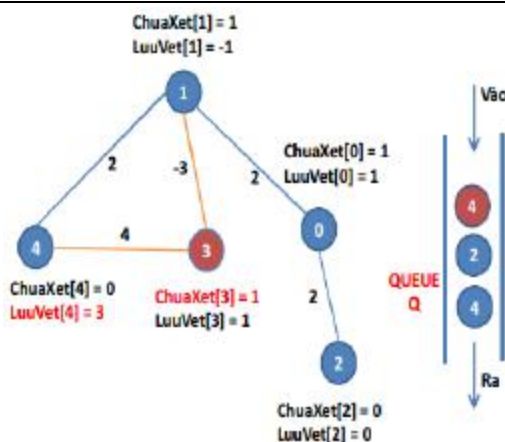
Hàng đợi Queue Q ko rỗng.

Lấy điểm dưới cùng ra. Là đỉnh 0 và tiến hành xét đỉnh 0.

- ChuaXet[0] = 1; // gán lại giá trị đỉnh 0 trong mảng ChuaXet là 1, điều này có nghĩa là đỉnh 0 đã và đang được xét hay duyệt đến theo thuật toán.
- Từ đỉnh 0 ta xem xét có đường đi đến đỉnh 1, 2. Và đỉnh 1 thì xét rồi (ChuaXet[1] = 1) nên bỏ qua, đỉnh 2 thì chưa xét nên:
  - Đẩy đỉnh 2 vào Queue Q.
  - LuuVet[2] = 0; // đánh dấu lại đỉnh 2 được đi đến từ đỉnh 0 trong quá trình duyệt theo thuật toán BFS

Sau đó ta quay lại đầu bước này

5



Kiểm tra hàng đợi Queue Q có rỗng ko.

Hàng đợi Queue Q ko rỗng.

Lấy điểm dưới cùng ra. Là đỉnh 3 và tiến hành xét đỉnh 3.

- ChuaXet[3] = 1;
- Từ đỉnh 3 ta xem xét có đường đi đến đỉnh 1, 4. Và đỉnh 1 thì xét rồi (ChuaXet[1] = 1) nên bỏ qua, đỉnh 4 thì chưa xét nên:
  - Đẩy đỉnh 4 vào Queue Q.
  - LuuVet[4] = 3; // đánh dấu lại đỉnh 4 được đi đến từ đỉnh 3 trong quá trình duyệt theo thuật toán BFS

Sau đó ta quay lại đầu bước này



6	<p>ChuaXet[1] = 0 LuuVet[1] = -1</p> <p>ChuaXet[4] = 1 LuuVet[4] = 3</p> <p>ChuaXet[3] = 1 LuuVet[3] = 1</p> <p>ChuaXet[0] = 1 LuuVet[0] = 1</p> <p>ChuaXet[2] = 0 LuuVet[2] = 0</p> <p>QUEUE Q</p> <p>Vào</p> <p>Ra</p>	<p>Kiểm tra hàng đợi Queue Q có rỗng ko. Hàng đợi Queue Q ko rỗng. Lấy điểm dưới cùng ra. Là đỉnh 4 và tiến hành xét đỉnh 4.</p> <ul style="list-style-type: none"> <li>- ChuaXet[4] = 1;</li> <li>- Từ đỉnh 4 ta xem xét có đường đi đến đỉnh 1, 3. Và đỉnh 1, 3 thì xét rồi (ChuaXet[1] = 1, ChuaXet[3] = 1) nên bỏ qua. Vậy ko có đỉnh nào mới đưa vào QUEUE Q.</li> </ul> <p>Sau đó ta quay lại đầu bước này.</p>
7	<p>ChuaXet[1] = 0 LuuVet[1] = -1</p> <p>ChuaXet[4] = 1 LuuVet[4] = 3</p> <p>ChuaXet[3] = 1 LuuVet[3] = 1</p> <p>ChuaXet[0] = 1 LuuVet[0] = 1</p> <p>ChuaXet[2] = 1 LuuVet[2] = 0</p> <p>QUEUE Q</p> <p>Vào</p> <p>Ra</p>	<p>Kiểm tra hàng đợi Queue Q có rỗng ko. Hàng đợi Queue Q ko rỗng. Lấy điểm dưới cùng ra. Là đỉnh 2 và tiến hành xét đỉnh 2.</p> <ul style="list-style-type: none"> <li>- ChuaXet[2] = 1;</li> <li>- Từ đỉnh 2 ta xem xét có đường đi đến đỉnh 0. Và đỉnh 0 thì xét rồi (ChuaXet[0] = 1) nên bỏ qua. Vậy ko có đỉnh nào mới đưa vào QUEUE Q.</li> </ul> <p>Sau đó ta quay lại đầu bước này.</p>
8	<p>ChuaXet[1] = 0 LuuVet[1] = -1</p> <p>ChuaXet[4] = 1 LuuVet[4] = 3</p> <p>ChuaXet[3] = 1 LuuVet[3] = 1</p> <p>ChuaXet[0] = 1 LuuVet[0] = 1</p> <p>ChuaXet[2] = 1 LuuVet[2] = 0</p> <p>QUEUE Q</p> <p>Vào</p> <p>Ra</p>	<p>Kiểm tra hàng đợi Queue Q có rỗng ko. Hàng đợi Queue Q ko rỗng. Lấy điểm dưới cùng ra. Là đỉnh 4 và tiến hành xét đỉnh 4. Mà đỉnh 4 có giá trị ChuaXet[4] = 1; nên không cần phải xét nữa. Tới đây, ta thấy Queue Q rỗng nên thuật toán dừng.</p>

Sau khi duyệt BFS xong, bạn muốn biết đường đi từ đỉnh S (là đỉnh 1 trong trường hợp này) thì bạn căn cứ vào mảng LuuVet và ChuaXet.

LuuVet[0]	LuuVet[1]	LuuVet[2]	LuuVet[3]	LuuVet[4]
1	-1	0	1	3
ChuaXet[0]	ChuaXet[0]	ChuaXet[0]	ChuaXet[0]	ChuaXet[0]
1	1	1	1	1

Giả sử, bạn muốn tìm đường đi từ đỉnh 1 đến đỉnh 4. Thì đầu tiên căn cứ vào giá trị của mảng ChuaXet.

- Ta thấy  $\text{ChuaXet}[4] = 1 \rightarrow$  Có đường đi từ 1 đến 4.
- Ta sẽ truy vết đường đi dựa vào mảng LuuVet.
  - o Ta thấy  $\text{LuuVet}[4] = 3 \rightarrow$  đỉnh 4 được đi đến từ đỉnh 3 trong đường đi duyệt theo BFS.
  - o Rồi qua xét  $\text{LuuVet}[3] = 1 \rightarrow$  đỉnh 3 được đi đến từ đỉnh 1 trong đường đi duyệt theo BFS.
  - o Ta thấy đỉnh 1 là đỉnh xuất phát đường đi nên dừng.
- Vậy đường đi từ 1  $\rightarrow$  4 là  $4 \leftarrow 3 \leftarrow 1$ .
- Tương tự ta có đường đi từ 1  $\rightarrow$  2 và 1  $\rightarrow$  3.

### 3.5 HƯỚNG DẪN TÌM KIẾM ĐƯỜNG ĐI BẰNG PHƯƠNG PHÁP DUYỆT THEO CHIỀU RỘNG (BFS)

Bạn có một đồ thị  $g$  (gồm đỉnh  $n$  và ma trận kề  $a$ ), bạn muốn tìm đường đi từ một đỉnh  $S$  (Start) đến một đỉnh  $F$  (Finish) trong đồ thị đó. Hãy viết chương trình tìm đường đi có thể theo thuật toán duyệt theo chiều rộng (BFS). Nếu tìm có đường đi từ đỉnh  $S$  (Start) đến đỉnh  $F$  (Finish) trong đồ thị  $g$  này thì bạn xuất ra đường đi, còn nếu không có đường đi thì bạn thông báo không có đường đi từ đỉnh  $S$  (Start) đến đỉnh  $F$  (Finish).

**Bước 1:** Tạo 1 mảng 1 chiều LuuVet dùng để lưu vết đường đi từ  $S \rightarrow F$ , và 1 mảng 1 chiều khác với tên là ChuaXet dùng để đánh dấu đỉnh nào trong đồ thị đã xét rồi, đỉnh nào chưa xét trong quá trình tìm đường đi từ  $S \rightarrow F$ .

```
int LuuVet[MAX]; // LuuVet[i] = đỉnh liền trước i trên đường đi từ S → i

int ChuaXet[MAX]; // ChuaXet[i] = 0 là đỉnh i chưa được xét đến trong quá trình
tìm đường đi, còn ChuaXet[i] = 1 là đỉnh i được xét đến rồi trong quá trình tìm đường
đi.
```

**Bước 2: Code phân duyệt theo chiều rộng (BFS) không dùng đệ qui.**

Đầu tiên tạo một cấu trúc hàng đợi như sau dùng để lưu thứ tự danh sách các đỉnh cần được duyệt.

```
struct QUEUE
{
    int size; // kích thước hiện tại hay số phần tử có trong hàng đợi
    int array[MAX]; // mảng lưu các giá trị trong hàng đợi
};
```

Để khởi tạo một hàng đợi thì dùng hàm KhoiTaoQueue như sau:

```
void KhoiTaoQueue(QUEUE &Q)
{
    Q.size = 0; // vì ban đầu hàng đợi rỗng nên size của nó là 0
}
```

Để đẩy một giá trị vào hàng đợi, ta dùng hàm DayGiaTriVaoQueue như sau:

```
int DayGiaTriVaoQueue(QUEUE &Q,int value)
{
    if(Q.size + 1 >= 100) // nếu hàng đợi đã đầy rồi thì không thể đẩy thêm giá trị
vào hàng đợi được nữa
        return 0; // trả về kết quả 0 báo cho người lập trình biết là không thể đẩy
thêm giá trị vào hàng đợi được nữa.
    Q.array[Q.size] = value; // đẩy giá trị vào hàng đợi.
    Q.size++; // tăng số lượng phần tử trong hàng đợi
    return 1; // trả về kết quả 1 báo cho người lập trình biết là đã đẩy thêm giá trị
vào hàng đợi thành công.
}
```

Để lấy một giá trị ra từ hàng đợi ta dùng hàm LayGiaTriRaKhoiQueue như sau:

```
int LayGiaTriRaKhoiQueue(QUEUE &Q,int &value)
{
    if(Q.size <= 0) // nếu hàng đợi rỗng thì không thể lấy ra giá trị nào được
        return 0; // trả về kết quả 0 báo cho người lập trình biết là hàng đợi rỗng,
        không thể lấy giá trị nào
    value = Q.array[0]; // lấy giá trị đầu tiên trong hàng đợi.
    for(int i = 0; i < Q.size - 1 ; i++) // tiến hành loại bỏ giá trị đầu tiên ra khỏi hàng
    đợi
        Q.array[i] = Q.array[i+1];
    Q.size--; // giảm kích thước hàng đợi vì đã lấy ra 1 giá trị hay phần tử
    return 1; // trả về kết quả 1 báo cho người lập trình biết là đã lấy giá trị ra khỏi
    hàng đợi thành công.
}
```

Để kiểm tra hàng đợi có rỗng hay không, ta dùng hàm KiemTraQueueRong như sau:

```
int KiemTraQueueRong(QUEUE Q)
{
    if(Q.size <= 0) // nếu hàng đợi rỗng thì trả về kết quả 1, ngược lại không rỗng là
    0
        return 1;
    return 0;
}
```

**Bước 3:** Để tiến hành duyệt BFS tại một đỉnh  $v$  nào đó của đồ thị  $g$ , ta viết hàm BFS sau:

```
void BFS(int v, GRAPH g)
{
    QUEUE Q;
    KhoiTaoQueue(Q); // khởi tạo hàng đợi vì ban đầu thuật toán hàng đợi ta chưa
    có gì
```

```

DayGiaTriVaoQueue(Q,v); // đẩy đỉnh v vào hàng đợi theo như thuật toán đã
trình bày

while(!KiemTraQueueRong(Q)) // trong khi hàng đợi chưa rỗng thì tiến hành các
bước sau
{
    LayGiaTriRaKhoiQueue(Q,v); //lấy phần tử đầu tiên trong hàng đợi ra và
gán giá trị đó vào v

    ChuaXet[v] = 1; //bắt đầu xét đỉnh v nên nhãn ChuaXet của đỉnh v được
gán là 1, tức đang xét hoặc đã xét trong thuật toán BFS

    for(int u = 0; u < g.n ; u++)
    {
        if(g.a[v][u] != 0 && ChuaXet[u] == 0) // Xem xét có cạnh nào nối từ
đỉnh v đến đỉnh u trong đồ thị g không (điều này tương ứng với g.a[v][u] != 0) và
đỉnh u đã được xét hay duyệt đến hay chưa? Nếu có cạnh nối từ đỉnh v đến đỉnh u và
đỉnh u chưa được xét hay duyệt đến thì tiến hành như sau

        {
            DayGiaTriVaoQueue(Q,u); // đẩy đỉnh u vào hàng đợi theo như
thuật toán đã trình bày

            if (LuuVet[u] == -1)

                LuuVet[u] = v; // đánh dấu lại đỉnh u được đi đến từ đỉnh v trong quá trình duyệt
theo thuật toán BFS

        }
    }
}

```

**Bước 4:** Trước khi tiến hành duyệt BFS để tìm đường đi  $S \rightarrow F$  thì cần phải khởi tạo các giá trị thích hợp cho các mảng LuuVet và ChuaXet. Quá trình đó như sau

```

void duyetttheoBFS (int S, int F, GRAPH g) // hàm này dùng để tìm đường đi từ
S  $\rightarrow$  F trong đồ thị g theo thuật toán BFS
{
    // khởi tạo lại các giá trị thích hợp cho mảng LuuVet và ChuaXet

    /* các bạn tự viết code khởi tạo các giá trị cho mảng LuuVet và ChuaXet. Gợi ý vì
ban đầu chưa chạy thuật toán nên các đỉnh i đều chưa có vết đi đến đó nên đặt giá trị

```

-1. Và ban đầu chưa chạy thuật toán nên đỉnh  $i$  trong đồ thị  $g$  đều chưa được xét nên giá trị là 0 \*/

```
BFS(S,g); // tiến hành thuật toán BFS
```

```
// xuất đường đi
```

if (ChuaXet[F] == 1) // sau khi thuật toán duyệt xong mà đỉnh  $F$  được xét hay duyệt đến thì nhãn của nó = 1 điều này có nghĩa là có đường đi từ  $S \rightarrow F$ . Tiến hành xuất đường đi.

```
{
```

```
    printf("Duong di tu dinh %d den dinh %d la: \n\t",S,F);
```

```
    i = F;
```

```
    printf("%d ", F);
```

/\* các bạn tự viết code xuất ra đường đi từ  $F \leftarrow S$  hén. Gợi ý dựa vào mảng LuuVet để tiến hành truy vết ra đường đi từ  $S \rightarrow F$ \*/

```
}
```

Else // sau khi thuật toán duyệt xong mà đỉnh  $F$  chưa được xét hay duyệt đến thì nhãn của nó = 0 điều này có nghĩa là không có đường đi từ  $S \rightarrow F$ . Thông báo không có đường đi từ  $S \rightarrow F$

```
{
```

```
    printf("Khong co duong di tu dinh %d den dinh %d \n",S,F);
```

```
}
```

```
}
```

**Bước 5:** Code trong hàm main để gọi hàm các hàm tương ứng và chạy.

# BÀI 4: TÌM CÂY KHUNG NHỎ NHẤT

Học xong bài này người học sẽ:

- Hiểu được khái niệm cây khung, cây bao trùm.
- Nắm được và thi hành thuật toán Prim.
- Nắm được và thi hành thuật toán Kruskal.

## 4.1 ĐỊNH NGHĨA CÂY KHUNG

Cho  $G=(X, E)$  là một đồ thị liên thông và  $T=(X, F)$  là một đồ thị bộ phận của  $G$ . Nếu  $T$  là cây thì  $T$  được gọi là một cây khung của  $G$ .

Cây khung còn có thể được gọi bằng các tên khác như cây bao trùm, cây phủ hoặc là cây tối đại.

Mọi đồ thị liên thông đều có chứa ít nhất một cây khung.

## 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 ngắn nhất của  $G$  như sau:

**Bước 1:** Chọn tùy ý  $v \in X$  và khởi tạo  $Y := \{v\}$ ;  $T := \emptyset$ . Trong đó  $X$  là tập các đỉnh chưa xét của đồ thị,  $Y$  là tập các đỉnh được chọn vào cây khung ngắn nhất, tức các đỉnh đã xét của đồ thị và  $T$  là tập các cạnh của cây khung này.

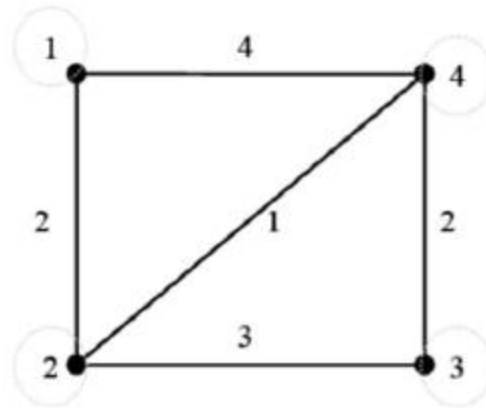
**Bước 2:** Trong số những cạnh  $e$  nối đỉnh  $w$  với đỉnh  $v$  trong  $Y$  với  $w \in X \setminus Y$  và  $v \in Y$ , ta chọn cạnh có trọng lượng nhỏ nhất.

**Bước 3:** Gán  $Y := Y \cup \{w\}$  và  $T := T \cup \{e\}$ . Ghi chú phép  $\cup$  được gọi là phép hợp trong phép toán tập hợp.

**Bước 4:** Nếu  $Y$  có đủ  $n$  phân tử hoặc  $T$  có đủ  $n-1$  cạnh thì dừng, ngược lại làm tiếp tục bước 2.

**Chú ý:** trong các thuật toán tìm cây khung ngắn nhất, chúng ta có thể bỏ đi hướng các cạnh và khuyên, đối với cạnh song song thì có thể bỏ đi hoặc chỉ để lại một cạnh có trọng lượng nhỏ nhất trong chúng.

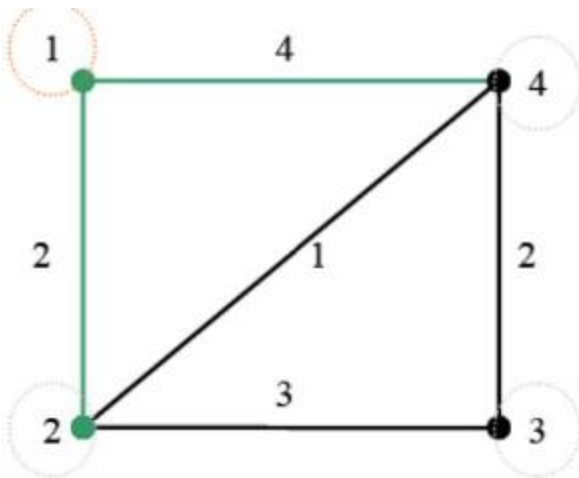
Ví dụ thi hành thuật toán Prim cho đồ thị sau:



**Hình 4.1:** Tìm cây khung với thuật toán Prim

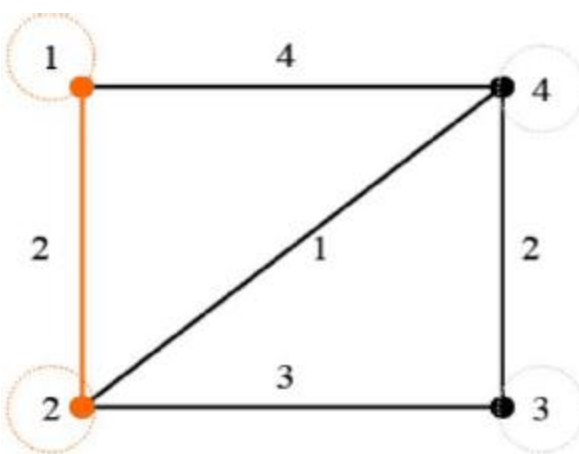
Trạng thái đồ thị	Xử lý
	<p>Bước 1: Chọn tùy ý <math>v \in X</math> và khởi tạo <math>Y := \{v\}; T := \emptyset</math>.</p> <p>Tập <math>X</math> là tập các đỉnh chưa xét của đồ thị nên <math>X = \{1, 2, 3, 4, 5\}</math>.</p> <p>Ta chọn đỉnh <b>1</b> làm đỉnh xét đến đầu tiên. Khi đó, <math>Y = \{1\}; T = \emptyset</math>. Như vậy <math>X \setminus Y = \{2, 3, 4, 5\}</math>.</p>





Bước 2: Trong số những cạnh  $e$  nối đỉnh  $w$  với đỉnh  $v$  trong  $Y$  với  $w \in X \setminus Y$  và  $v \in Y$ , ta chọn cạnh có trọng lượng nhỏ nhất.

Cạnh  $(4,1)$  và  $(2,1)$  nối đỉnh 2 và 4 đến đỉnh 1 trong  $Y$ , ta chọn cạnh  $(2,1)$  vì nó có trọng nhỏ nhất trong hai cạnh (giá trị là 2).

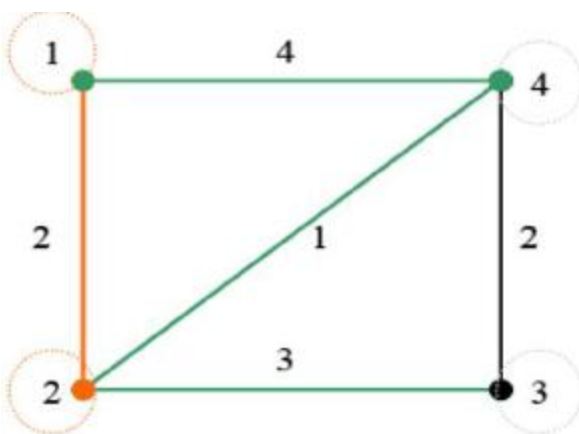


Bước 3: Gán  $Y := Y \cup \{w\}$  và  $T := T \cup \{e\}$ .

→  $Y = Y \cup \{2\}$ ,  $T = T \cup \{(2,1)\}$ .

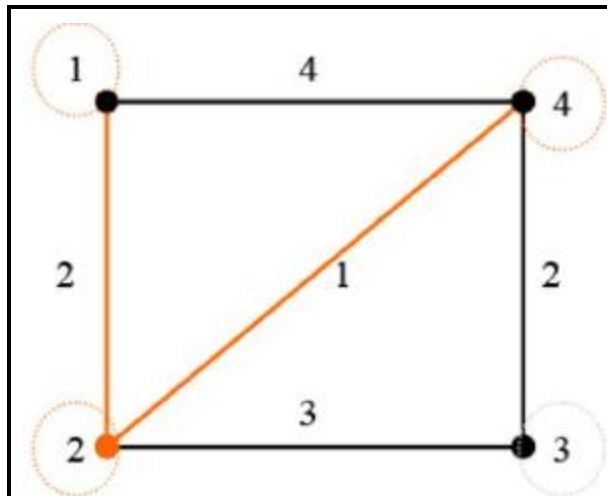
Bước 4: Nếu  $Y$  có đủ  $n$  phần tử hoặc  $T$  có đủ  $n-1$  cạnh thì dừng, ngược lại làm tiếp tục bước 2.

$Y$  chỉ có 2 phần tử  $< n = 4$  ( $T$  có 1 cạnh  $< n-1 = 4-1 = 3$ ) nên thuật toán chưa dừng.



Bước 2 (lần 2): Cạnh  $(4,1)$ ;  $(4,2)$ ;  $(3,2)$  là các cạnh nối đỉnh 4; 3 (tập những đỉnh chưa có trong cây) đến 1; 2 (tập các đỉnh đã có trong cây).

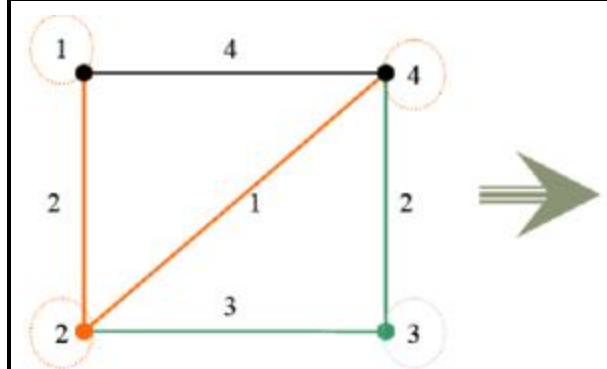
Tương tự, ta chọn cạnh có trọng lượng nhỏ nhất:  $(4,2)$ .



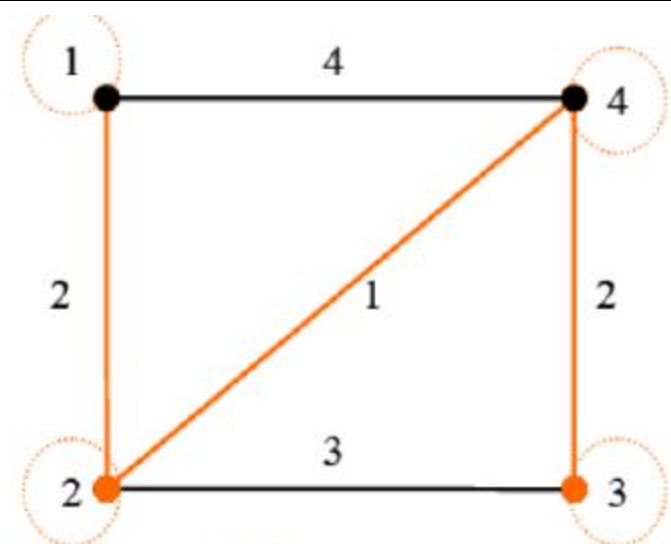
Bước 3 (lần 2):  $Y = Y \cup \{4\}$ ;

$T = T \cup \{(4,2)\}$ .

$Y$  chỉ có 3 phần tử  $< n = 4$  ( $T$  có 2 cạnh  $< n - 1 = 4 - 1 = 3$ ) nên thuật toán tiếp tục chạy.



Tương tự cho các bước lặp kế tiếp. Chọn cạnh (3,4) với giá trị 2.



Lúc này  $T$  đã chứa đủ 3 cạnh vì vậy thuật toán dừng. Tập đỉnh:  $Y = 1, 2, 4, 3$   
Tập cạnh:  $T = \{(1,2); (4,2); (3,4)\}$

**Chú ý:** ta cũng cần xác định đồ thị đưa vào có liên thông hay không trước khi thi hành thuật toán Prim đồng thời chú ý đồ thị có hướng (xét cả 2 chiều và lấy cạnh có trọng số nhỏ nhất).

## 4.3 HƯỚNG DẪN THI HÀNH THUẬT TOÁN PRIM

Với một đồ thị cho trước gồm số đỉnh  $n$  và ma trận kề. Viết chương trình thi hành thuật toán Prim trên đồ thị đó để tìm cây khung nhỏ nhất. Nếu tìm được cây khung nhỏ nhất, thì hãy xuất ra  $k$  là tổng các giá trị của cạnh trong cây khung nhỏ nhất

(trọng lượng của cây khung nhỏ nhất) và dòng tiếp theo là các cạnh (u,v) thuộc cây khung này. Còn nếu không tìm được, xuất ra thông báo không tìm được cây khung nhỏ nhất hay đồ thị không liên thông.

**Bước 1:** Trước khi thi hành thuật toán Prim cần phải kiểm tra đồ thị có liên thông hay không? Nếu đồ thị không liên thông thì không có cây khung nhỏ nhất. Còn ngược lại thì có.

Lật lại bài "**xét tính liên thông**", xem và code phần xét liên thông trên đồ thị. Tuy nhiên, bạn cần có một sự thay đổi nhỏ là hàm liên thông phải trả về kết quả để mình biết đường (đồ thị liên thông hay không) mà xử lý.

```
int XetLienThong(DOTHI g)
{
    /*code như phần xét liên thông chỉ thêm trả kết quả về */
    return SoThanhPhanLT;
}
```

**Bước 2:** Tiến hành code thuật toán **Prim** như sau:

```
int ChuaXet[MAX]; // giá trị 0 là chưa xét, giá trị 1 là xét rồi.
typedef struct EDGE // khai báo 1 cấu trúc CANH cho cạnh của đồ thị
{
    int u;
    int v;
    int value;
}CANH;
CANH T[MAX]; // mảng lưu các cạnh trong thuật toán Prim
void Prim (DOTHI g)
{
```

```
if (XetLienThong(g) != 1) // đi kiểm tra đồ thị có liên thông không, nếu kết quả trả về là 1 thì đồ thị liên thông (tức chỉ có 1 thành phần liên thông), còn khác 1 thì đồ thị không liên thông
```

```
{
```

```
    printf ("Do thi khong lien thong, do do khong thuc hien duoc thuat toan Prim tim cay khung nho nhat\n");
```

```
    return;
```

```
}
```

```
int nT = 0; // dùng để lưu số cạnh trong thuật toán Prim
```

```
for (int i = 0; i < g.n; i++) // ban đầu thuật toán Prim, chưa làm gì nên gán giá trị chưa xét cho tất cả các đỉnh bằng 0
```

```
    ChuaXet[i] = 0;
```

```
ChuaXet[0] = 1; // bắt đầu thuật toán Prim, xuất phát từ đỉnh 1
```

```
while (nT < g.n - 1) // nếu thuật toán đã thu thập đủ g.n-1 cạnh thì thuật toán dừng
```

```
{
```

```
    CANH CanhNhoNhat; // dùng để lưu cạnh nhỏ nhất nối từ tập những đỉnh đã xét (giá trị chưa xét == 1) đến 1 đỉnh chưa xét nào đó trong đồ thị.
```

```
    int GiaTriNhoNhat = 100; // khởi tạo giá trị nhỏ nhất của cạnh nhỏ nhất là 100, bạn có thể thay đổi số này sao cho phù hợp với bài toán của bạn.
```

```
    for (int i = 0; i < g.n; i++) // duyệt các đỉnh trong đồ thị
```

```
{
```

```
        if (ChuaXet[i] == 1) // xem đỉnh nào đã xét
```

```
{
```

```
            for (int j = 0; j < g.n; j++)
```

```

        if (ChuaXet[j] == 0 && g.a[i][j] != 0 && GiaTriNhoNhat >
g.a[i][j]) // tìm đỉnh chưa xét j, có cạnh nối từ đỉnh i đến đỉnh j và giá trị của cạnh đó
nhỏ hơn giá trị nhỏ nhất ở trên

        {

            CanhNhoNhat.u = i; // cập nhập lại giá trị trong
CanhNhoNhat

            CanhNhoNhat.v = j;

            CanhNhoNhat.value = g.a[i][j];

            GiaTriNhoNhat = g.a[i][j]; // cập nhập lại GiaTriNhoNhat

        }

    }

    T[nT] = CanhNhoNhat; //Thêm cạnh nhỏ nhất vào tập cạnh T của mình
    nT++; // tăng số cạnh lên 1

    ChuaXet[CanhNhoNhat.v] = 1; // gán lại giá trị chưa xét của đỉnh v là 1,
tức đã xét rồi

}

int TongTrongSoCuaCayKhung = 0; // trọng số của cây khung nhỏ nhất
// xuất cây khung nhỏ nhất bởi thuật toán Prim và giá trị của cây khung
printf ("Cay khung nho nhat cua do thi la \n");
for (i = 0; i < nT - 1; i++)
{

    printf("(%d,%d), ", T[i].u, T[i].v);

    TongTrongSoCuaCayKhung += T[i].value;

}

```

```
printf("(%d,%d).\n", T[nT - 1].u, T[nT - 1].v);  
  
TongTrongSoCuaCayKhung += T[nT - 1].value;  
  
printf("Tong gia tri cua cay khung la %d\n", TongTrongSoCuaCayKhung);  
  
}
```

**Bước 3:** Code trong hàm main để gọi hàm các hàm tương ứng và chạy.

## 4.4 THUẬT TOÁN KRUSKAL

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

**Bước 1:** Duyệt các cạnh của đồ thị  $G$  và tạo danh sách các cạnh listEdge.

**Bước 2:** Sắp xếp lại danh sách các cạnh listEdge của đồ thị  $G$  theo trọng số tăng dần, rồi đánh dấu tất cả các cạnh là chưa xét và khởi tạo  $T := \emptyset$  ( $T$  là tập cạnh của cây khung của đồ thị  $G$ ).

**Bước 3:** Lấy cạnh  $e$  chưa xét có trọng số bé nhất trong danh sách listEdge đã sắp xếp, nếu  $T \cup \{e\}$  không chứa chu trình thì gán  $T := T \cup \{e\}$ .

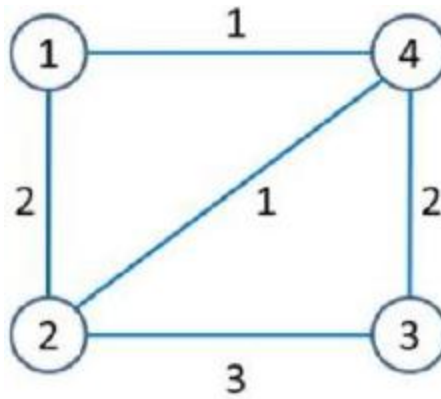
Đánh dấu cạnh  $e$  đã xét.

**Bước 4:** Nếu hết cạnh chưa xét (tức các cạnh đã xét hết) hoặc  $T$  có đủ  $n-1$  cạnh thì dừng, ngược lại làm tiếp tục bước 3.

Thuật toán dừng, nếu  $T$  không đủ  $n-1$  cạnh thì đồ thị không liên thông và không có cây khung nhỏ nhất. Ngược lại thì có cây khung/bao trùm.

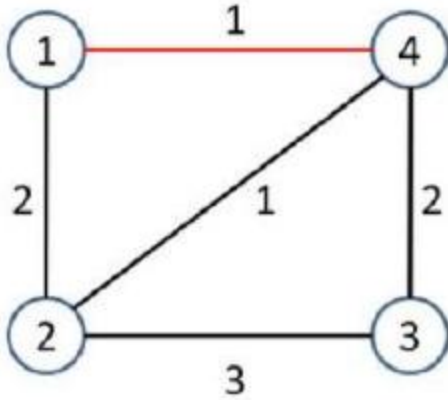
Chú ý: Trong các thuật toán tìm cây khung nhỏ nhất, chúng ta có thể bỏ đi hướng các cạnh và các khuyên, đối với cạnh song song thì có thể bỏ đi hoặc chỉ để lại một cạnh có trọng lượng nhỏ nhất trong chúng.

Ví dụ thi hành thuật toán Kruskal cho đồ thị sau:



Hình 4.2: Đồ thị thi hành thuật toán Kruskal

Trạng thái đồ thị	Xử lý
	<p>Bước 1: Duyệt các cạnh của đồ thị <math>G</math> và tạo danh sách các cạnh <math>listEdge</math>.</p> <p>Đồ thị có <math>G</math> có 5 cạnh <math>\rightarrow listEdge</math> có 5 cạnh.</p> <p><math>listEdge = \{(1,2); (1,4); (2,3); (2,4); (3,4)\}</math></p>
	<p>Bước 2: Sắp xếp lại danh sách các cạnh <math>listEdge</math> của đồ thị <math>G</math> theo trọng số tăng dần, rồi đánh dấu tất cả các cạnh là chưa xét và khởi tạo <math>T := \emptyset</math>.</p> <p>Sắp xếp lại <math>listEdge</math> theo trọng số tăng dần <math>\rightarrow listEdge = \{(1,4); (2,4); (1,2); (3,4); (2,3)\}</math></p> <p>Đánh dấu các cạnh chưa xét là màu đen.</p> <p><math>T := \emptyset</math>.</p>



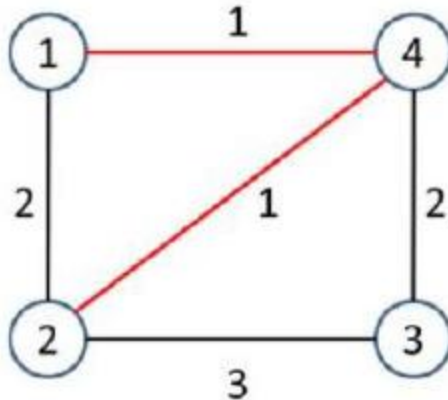
Bước 3: Lấy cạnh  $e$  chưa xét có trọng số bé nhất trong danh sách  $listEdge$  đã sắp xếp, nếu  $T \cup \{e\}$  không chứa chu trình thì gán  $T := T \cup \{e\}$ . Đánh dấu cạnh  $e$  đã xét.

Lấy cạnh  $e$  chưa xét có trọng số bé nhất trong danh sách  $listEdge \rightarrow$  lấy cạnh  $(1,4)$ .

$T \cup \{(1,4)\}$  không tạo chu trình  $\rightarrow T :=$

$T \cup \{(1,4)\} = \{(1,4)\}$ .

Đánh dấu cạnh  $(1,4)$  đã xét bằng màu đỏ.



Bước 4: Nếu hết cạnh chưa xét (tức các cạnh đã xét hết) hoặc  $T$  có đủ  $n-1$  cạnh thì dừng, ngược lại làm tiếp tục bước 3.

$T$  chỉ mới có 1 cạnh  $< n-1 = 4-1 = 3 \rightarrow$  quay lại bước 3.

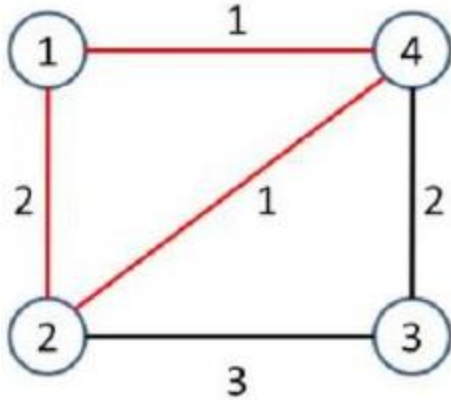
Bước 3 (lần 2): Lấy cạnh  $e$  chưa xét có trọng số bé nhất trong danh sách  $listEdge \rightarrow$  lấy cạnh  $(2,4)$ .

$T \cup \{(2,4)\}$  không tạo chu trình  $\rightarrow T :=$

$T \cup \{(2,4)\} = \{(1,2), (2,4)\}$ .

Đánh dấu cạnh  $(2,4)$  đã xét bằng màu đỏ.





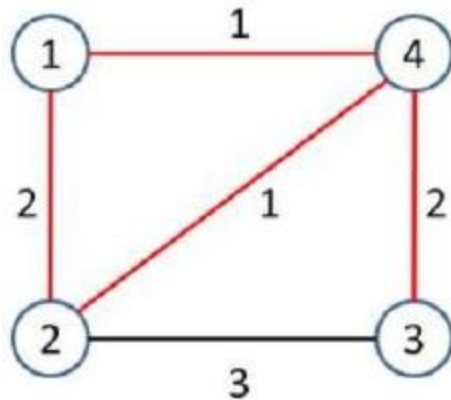
Bước 4 (lần 2): T chỉ mới có 2 cạnh  $< n-1 = 4 - 1 = 3 \rightarrow$  quay lại bước 3.

Bước 3 (lần 3): Lấy cạnh e chưa xét có trọng số bé nhất trong danh sách listEdge  $\rightarrow$  lấy cạnh (1,2).

$T \cup \{(1,2)\}$  tạo chu trình  $1 \rightarrow 2 \rightarrow 4 \rightarrow 1 \rightarrow$

Không thể lấy cạnh (1,2).  $\rightarrow T = \{(1,2); (2,4)\}$ .

Đánh dấu cạnh (1,2) đã xét bằng màu đỏ.



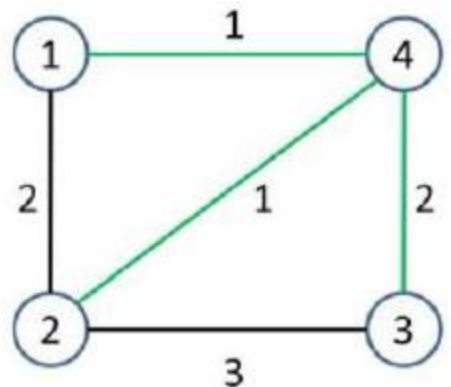
Bước 4 (lần 3): T chỉ mới có 2 cạnh  $< n-1 = 4 - 1 = 3 \rightarrow$  quay lại bước 3.

Bước 3 (lần 4): Lấy cạnh e chưa xét có trọng số bé nhất trong danh sách listEdge  $\rightarrow$  lấy cạnh (3,4).

$T \cup \{(3,4)\}$  không tạo chu trình  $\rightarrow T =$

$T \cup \{(3,4)\} = \{(1,2); (2,4); (3,4)\}$ .

Đánh dấu cạnh (3,4) đã xét bằng màu đỏ



Bước 4 (lần 4): T chỉ mới có 3 cạnh  $= n-1 = 4 - 1 = 3 \rightarrow$  dừng.

Vậy cây khung/bao trùm là màu xanh lá cây tương ứng với những cặp cạnh trong  $T = \{(1,2); (2,4); (3,4)\}$

Khi đó tổng trọng số của cây khung/cây bao trùm là  $1+1+2 = 4$ .

## 4.5 SO SÁNH SỰ KHÁC BIỆT GIỮA KRUSKAL VỚI PRIM

Khác với Prim, 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.

Khác với Prim là mở rộng tập đỉnh đã xét, 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 như thế này sẽ tốn chi phí. Ta có thể áp dụng cách sau để làm giảm chi phí kiểm tra chu trình.

**Bước 1:** Dùng một mảng Nhãn có kích thước bằng số đỉnh của đồ thị. Giá trị  $Nhan[i]$  tại đỉnh  $i$  được khởi tạo bằng với chính chỉ số  $i$  của đỉnh.

Ví dụ với đồ thị  $G$  trên ta có

Đỉnh	1	2	3	4
Nhãn	1	2	3	4

**Bước 2:** Khi chọn một cạnh  $(u,v)$  được thêm vào cây khung/cây bao trùm thì 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  $(1,4)$  vào cây khung/bao trùm hay tập  $T$ , ta sửa nhãn của tất cả các đỉnh có cùng giá trị với nhãn của đỉnh 4 (là 4) thành nhãn của đỉnh 1 (là 1).

Đỉnh	1	2	3	4
Nhãn	1	2	3	<u>1</u>

Sau đó, nếu như ta chọn tiếp cạnh  $(2,4)$ , ta sửa nhãn của tất cả các đỉnh có cùng giá trị với nhãn của đỉnh 4 (là 1) thành nhãn của đỉnh 2 (là 2).

Đỉnh	1	2	3	4
Nhãn	<u>2</u>	2	3	<u>2</u>

**Bước 3:** Sau đó, khi chọn cạnh để thêm vào cây khung thì chỉ chọn cạnh có nhãn hai đỉnh là khác nhau → không tạo thành chu trình.

Ví dụ khi ta xét tới cạnh (1,2) thì ta không chọn vì đỉnh 1 và đỉnh 2 có cùng nhãn là 2, do đó nếu ta chọn cạnh (1,2) sẽ tạo ra chu trình → không chọn cạnh (1,2) đưa vào cây khung.

Lưu ý: Sau khi ta đã chọn đủ  $n-1$  cạnh, ta sẽ có mảng nhãn chỉ chứa một giá trị duy nhất.

## 4.6 HƯỚNG DẪN THI HÀNH THUẬT TOÁN KRUSKAL

Với một đồ thị cho trước gồm số đỉnh  $n$  và ma trận kề. Hãy viết chương trình thi hành thuật toán Kruskal trên đồ thị đó để tìm cây khung nhỏ nhất. Nếu tìm được cây khung nhỏ nhất, thì hãy xuất ra  $k$  là tổng các giá trị/trọng số của các cạnh trong cây khung nhỏ nhất và dòng tiếp theo là các cạnh  $(u,v)$  thuộc cây khung này. Còn nếu không tìm được, xuất ra thông báo không tìm được cây khung nhỏ nhất hay đồ thị không liên thông.

**Bước 1:** Định nghĩa cấu trúc cạnh như sau:

```
typedef struct EDGE // khai báo 1 cấu trúc CANH cho cạnh của đồ thị
{
    int u;
    int v;
    int value;
}CANH;
```

**Bước 2:** viết hàm sắp xếp các cạnh trong danh sách các cạnh theo trọng số tăng dần.

```
void SapXepTang(CANH E[100],int tongsocanh)
{
    CANH canhtam;
    for(int i = 0 ; i < tongsocanh - 1 ; i++)
    {
        for(int j = i + 1 ; j < tongsocanh ; j++)
            if(E[i].value > E[j].value)
            {
```

```

        canhtram = E[i];
        E[i] = E[j];
        E[j] = canhtram;
    }
}
}

```

**Bước 3:** viết hàm Kruskal để thi hành thuật toán Kruskal như sau

```

void Kruskal (DOTHI g)
{
    CANH listEdge[MAX]; // chứa danh sách tất cả các cạnh của đồ thị
    int tongsocanh = 0; // chứa tổng số cạnh trong đồ thị
    int i,j;
    for(i = 0 ; i < g.n ; i++) // tiến hành thêm các cạnh trong đồ thị vào listEdge
    {
        for( j = i+1 ; j< g.n ; j++)
            if(g.a[i][j] > 0)
            {
                listEdge[tongsocanh].u = i;
                listEdge[tongsocanh].v = j;
                listEdge[tongsocanh].value = g.a[i][j];
                tongsocanh++;
            }
    }
    // tiến hành sắp xếp các cạnh trong listEdge theo trọng số tăng dần
    SapXepTang(listEdge, tongsocanh);
    int nT = 0; // số lượng các cạnh trong cây khung
    CANH T[MAX]; // chứa các cạnh của cây khung
    int nhan[MAX]; // chứa nhãn của các đỉnh trong đồ thị theo thuật toán
    for (i = 0; i < g.n ; i++)
        nhan[i] = i;
    int canh dangxet = 0; // lưu lại thuật toán đang xét cạnh thứ mấy trong danh
    sách listEdge
    while(nT < g.n && canh dangxet < tongsocanh)
    {

```

```

// tiến hành thêm một cạnh vào cây khung mà không tạo chu trình bằng
cách chọn cạnh mà đỉnh tại nhãn khác nhau
if (nhân[listEdge[canhdangxet].u] != nhân[listEdge[canhdangxet].v])
{
    T[nT] = listEdge[canhdangxet];
    nT++;
    // tiến hành sửa nhãn của tất cả các đỉnh có cùng giá trị với nhãn của
    đỉnh v thành nhãn của đỉnh u
    int giatri = nhân[listEdge[canhdangxet].v];
    for (j = 0; j < g.n; j++)
        if (nhân[j] == giatri)
            nhân[j] = nhân[listEdge[canhdangxet].u];
}
canhdangxet++; // xét cạnh kế tiếp trong danh sách cạnh listEdge
}
if(nT != g.n - 1) // nếu số cạnh trong cây khung không đủ n-1 cạnh thì suy ra đồ
thị không liên thông
    printf("\nDo thi khong lien thong \n");
else // có cây khung, tiến hành xuất
{
    int TongTrongSoCuaCayKhung = 0;
    printf("\nDo thi lien thong \n");
    printf ("Cay khung nho nhat cua do thi la \n");
    for (i = 0; i < nT; i++)
    {
        printf("(%d,%d), ", T[i].u, T[i].v);
        TongTrongSoCuaCayKhung += T[i].value;
    }
    printf ("\nTong gia tri cua cay khung la %d\n",TongTrongSoCuaCayKhung);
}
}

```

**Bước 4:** Code trong hàm main để gọi hàm các hàm tương ứng và chạy.

# BÀI 5: CÁC THUẬT TOÁN TÌM ĐƯỜNG ĐI NGẮN NHẤT

Học xong bài này người học sẽ nắm được các nội dung sau:

- Tìm được lời giải phù hợp cho từng bài toán tìm đường đi tối ưu với các điều kiện cho trước.
- Hiểu được ý tưởng và các bước thực hiện các thuật toán tìm đường đi như Dijkstra, Floyd.

## 5.1 GIỚI THIỆU

---

Trong đời sống, chúng ta thường gặp những tình huống như sau: để đi từ địa điểm A đến địa điểm B trong thành phố, có nhiều đường đi, nhiều cách đi; có lúc ta chọn đường đi ngắn nhất (theo nghĩa cự ly), có lúc lại cần chọn đường đi nhanh nhất (theo nghĩa thời gian) và có lúc phải cân nhắc để chọn đường đi rẻ tiền nhất (theo nghĩa chi phí), v.v...

Có thể coi sơ đồ của đường đi từ A đến B trong thành phố là một đồ thị, với đỉnh là các giao lộ (A và B coi như giao lộ), cạnh là đoạn đường nối hai giao lộ. Trên mỗi cạnh của đồ thị này, ta gán một số dương, ứng với chiều dài của đoạn đường, thời gian đi đoạn đường hoặc cước phí vận chuyển trên đoạn đường đó, ...

Đồ thị có trọng số là đồ thị  $G=(V,E)$  mà mỗi cạnh (hoặc cung)  $e \in E$  được gán bởi một số thực  $m(e)$ , gọi là trọng số của cạnh (hoặc cung)  $e$ .

Trong phần này, trọng số của mỗi cạnh được xét là một số dương và còn gọi là chiều dài của cạnh đó. Mỗi đường đi từ đỉnh  $u$  đến đỉnh  $v$ , có chiều dài là  $m(u,v)$ , bằng tổng chiều dài các cạnh mà nó đi qua. Khoảng cách  $d(u,v)$  giữa hai đỉnh  $u$  và  $v$  là chiều dài đường đi ngắn nhất (theo nghĩa  $m(u,v)$  nhỏ nhất) trong các đường đi từ  $u$  đến  $v$ .

Có thể xem một đồ thị  $G$  bất kỳ là một đồ thị có trọng số mà mọi cạnh đều có chiều dài 1. Khi đó, khoảng cách  $d(u,v)$  giữa hai đỉnh  $u$  và  $v$  là chiều dài của đường đi từ  $u$  đến  $v$  ngắn nhất, tức là đường đi qua ít cạnh nhất.

## 5.2 THUẬT TOÁN DIJKSTRA

Cho  $G=(X,E)$  là một đồ thị có trọng số không âm gồm  $n$  đỉnh. Thuật toán Dijkstra dùng để tìm đường đi ngắn nhất giữa 2 đỉnh  $S$ (Start) và  $F$ (Finish) trong đồ thị  $G$  như sau:

Ta sử dụng 1 mảng 1 chiều  $LuuVet$  dùng để lưu vết đường đi từ  $S \rightarrow F$ , 1 mảng 1 chiều khác với tên là  $ChuaXet$  dùng để đánh dấu đỉnh nào trong đồ thị đã xét rồi, đỉnh nào chưa xét trong quá trình tìm đường đi từ  $S \rightarrow F$ , 1 mảng  $DoDaiDuongDiToi$  để lưu lại độ dài nhỏ nhất trong quá trình tìm đường đi từ  $S \rightarrow F$ .

- $LuuVet[MAX]$ : lưu đỉnh liền trước nó trên đường đi.
- $ChuaXet[MAX]$ : đánh dấu đỉnh nào trong đồ thị đã xét rồi, đỉnh nào chưa xét.
- $DoDaiDuongDiToi[MAX]$ : Lưu độ dài từ đỉnh đầu  $i$  đến các đỉnh trong đồ thị.

Các bước thi hành thuật toán Dijkstra như sau

**Bước 1:** Khởi tạo

$$ChuaXet[i] = 0; \forall i \in X$$

$$LuuVet[i] = -1; \forall i \in X$$

$$DoDaiDuongDiToi[S] = 0$$

$$DoDaiDuongDiToi[i] = +\infty, \forall i \in X \setminus \{S\}$$

**Bước 2:** Nếu  $ChuaXet[F] == 1$  (tức đã xét tới  $F$ ) thì dừng thuật toán và giá trị  $DoDaiDuongDiToi[F]$  là độ dài đường đi ngắn nhất từ  $S \rightarrow F$  và  $LuuVet[F]$  lưu đỉnh nằm ngay trước  $F$  trên đường đi đến  $F$ .

**Bước 3:** Chọn đỉnh  $v \in X$  sao cho  $ChuaXet[v]=0$  và  $DoDaiDuongDiToi[v]$  nhỏ nhất. Khi đó gán  $ChuaXet[v] = 1$ .

**Bước 4:**  $\forall k \in X$  mà  $ChuaXet[k] == 0$  và có cạnh nối từ  $v$  đến  $k$ :

$$\text{Nếu } DoDaiDuongDiToi[k] > DoDaiDuongDiToi[v] + e(v,k) \text{ thì}$$

$$\text{DoDaiDuongDiToi}[k] = \text{DoDaiDuongDiToi}[v] + e(v,k);$$

$$\text{LuuVet}[k] = v;$$

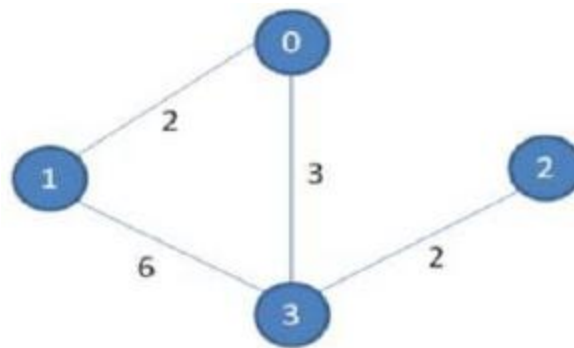
Cuối nếu

Cuối với mọi

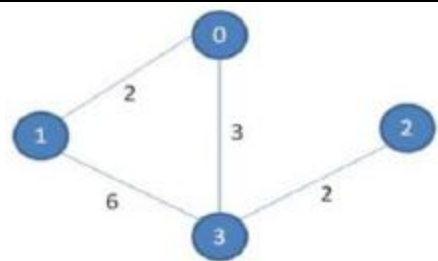
Trở về bước 2.

**Chú ý:** Khi thuật toán dừng, nếu  $\text{DoDaiDuongDiToi}[F] = +\infty$  thì không tồn tại đường đi từ S đến F, nếu ngược lại thì  $\text{DoDaiDuongDiToi}[F]$  là độ dài đường đi ngắn nhất.

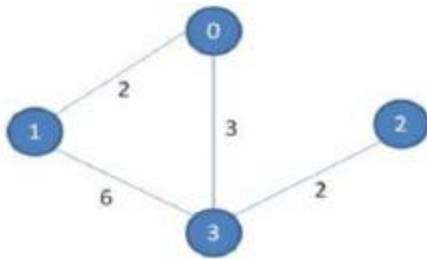
Vì dụ thi hành thuật toán Dijkstra trên đồ thị sau, tìm đường đi ngắn nhất từ đỉnh 1 đến đỉnh 2.



**Hình 5.1:** Đồ thị minh họa thuật toán Dijkstra

Trạng thái đồ thị	Xử lý																				
	<p>Bước 1: Khởi tạo</p> <p>Khởi tạo đỉnh 1 với độ dài min hiện tại là 0, và nhãn đỉnh trước là -1. Các đỉnh còn lại đều được gán độ dài min là <math>+\infty</math>. Dưới đây là bảng mô tả:</p> <table><tr><th>Mảng\Chỉ số</th><th>0</th><th>1</th><th>2</th><th>3</th></tr><tr><td>ChưaXét</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>LưuVết</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td></tr><tr><td>Độ dài</td><td><math>+\infty</math></td><td>0</td><td><math>+\infty</math></td><td><math>+\infty</math></td></tr></table>	Mảng\Chỉ số	0	1	2	3	ChưaXét	0	0	0	0	LưuVết	-1	-1	-1	-1	Độ dài	$+\infty$	0	$+\infty$	$+\infty$
Mảng\Chỉ số	0	1	2	3																	
ChưaXét	0	0	0	0																	
LưuVết	-1	-1	-1	-1																	
Độ dài	$+\infty$	0	$+\infty$	$+\infty$																	





Bước 2: Đỉnh 2 vẫn có giá trị ChưaXét[2] = 0 nên ta sang bước kế tiếp.

Bước 3: Chọn đỉnh có độ dài nó nhỏ nhất (trong mảng Độ dài) mà đỉnh giá trị chưa xét của nó là 0. Ở đây trong các đỉnh chưa xét đến là 0, 1, 2, 3. Đỉnh 1 có độ dài nhỏ nhất là 0.

Ta xét giá trị ChưaXét[1] = 1.

Mảng\Chỉ số	0	1	2	3
ChưaXét	0	1	0	0
LưuVết	-1	-1	-1	-1
Độ dài	$+\infty$	0	$+\infty$	$+\infty$

Bước 4: Từ đỉnh 1 ta có đường đi đến đỉnh 0 và đỉnh 3.

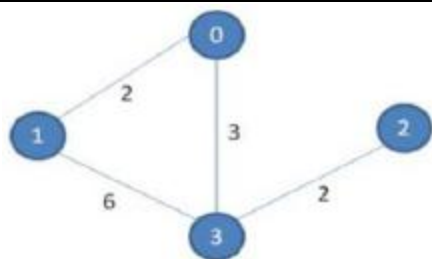
Chi phí từ đỉnh Các đỉnh còn lại đều có

DoDaiDuongDiToi cực đại nên ta cập nhật lại như sau:

Mảng\Chỉ số	0	1	2	3
ChưaXét	0	1	0	0
LưuVết	-1	-1	-1	-1
Độ dài	$+\infty$	0	$+\infty$	$+\infty$

DoDaiDuongDiToi\Bước	0	1
0	$+\infty$	2
1	0	0
2	$+\infty$	$+\infty$
3	$+\infty$	6

LưuVết\Bước	0	1
0	-1	1
1	-1	-1
2	-1	-1
3	-1	1



Bước 2 (lần 2): Đỉnh 2 vẫn có giá trị ChưaXet[2] = 0 nên ta sang bước kế tiếp.

Bước 3 (lần 2): Chọn đỉnh có độ dài nó nhỏ nhất. Ở đây là đỉnh 0, ta xét giá trị ChưaXet[0] = 1.

ChưaXet\Bước	0	1	2
0	0	0	1
1	0	1	1
2	0	0	0
3	0	0	0

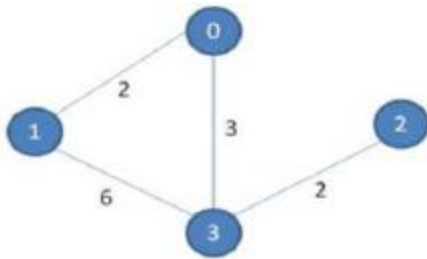
Bước 4: Tính độ dài từ đỉnh 0 vừa xét ở trên đến những còn lại có giá trị ChưaXet[i] = 0.

Đỉnh số 3 có chi phí mới là  $2+3 = 5$  nhỏ hơn chi phí cũ là 6. Vì vậy ta cập nhập lại đỉnh này.

DoDaiDuongDiToi\Bước	0	1	2
0	$+\infty$	2	2

1	0	0	0
2	$+\infty$	$+\infty$	$+\infty$
3	$+\infty$	6	5

LuuVet\Bước	0	1	2
0	-1	1	1
1	-1	-1	-1
2	-1	-1	-1
3	-1	1	0



Bước 2 (lần 3): Đỉnh 2 vẫn có giá trị  $ChuaXet[2] = 0$  nên ta sang bước kế tiếp.

Bước 3 (lần 3): Chọn đỉnh có độ dài nhỏ nhất. Ở đây là đỉnh 3, ta xét giá trị  $ChuaXet[3] = 1$ .

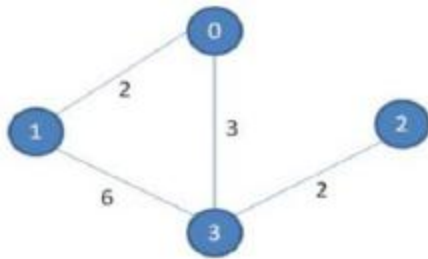
ChuaXet\Bước	0	1	2	3
0	0	0	1	1
1	0	1	1	1
2	0	0	0	0
3	0	0	0	1

Bước 4: Tính độ dài từ đỉnh 3 vừa xét ở trên đến những còn lại có giá trị  $ChuaXet[i] = 0$ .

Đỉnh số 2 có chi phí mới là  $5+2 = 7$  nhỏ hơn chi phí cũ là  $+\infty$ . Vì vậy ta cập nhập lại đỉnh này.

DoDaiDuongDiToi\Bước	0	1	2	3
0	$+\infty$	2	2	2
1	0	0	0	0
2	$+\infty$	$+\infty$	$+\infty$	7
3	$+\infty$	6	5	5

LuuVet\Bước	0	1	2	3
0	-1	1	1	1
1	-1	-1	-1	-1
2	-1	-1	-1	3
3	-1	1	0	0



Bước 2 (lần 4): Đỉnh 2 vẫn có giá trị ChuaXet[2] = 0 nên ta sang bước kế tiếp.

Bước 3 (lần 4): Chọn đỉnh có độ dài nó nhỏ nhất. Ở đây là đỉnh 2, ta xét giá trị ChuaXet[2] = 1.

ChuaXet\Bước	0	1	2	3	4
0	0	0	1	1	1
1	0	1	1	1	1
2	0	0	0	0	1
3	0	0	0	1	1

Lúc này, đỉnh 2 đã được xét đến và chúng ta kết thúc thuật toán.

Khi đó đường đi là  $1 \rightarrow 0 \rightarrow 3 \rightarrow 2$  với độ dài là 7.

## 5.3 HƯỚNG DẪN THI HÀNH TÌM ĐƯỜNG ĐI NGẮN NHẤT VỚI THUẬT TOÁN DIJKSTRA

Bạn có một đồ thị  $g$  (gồm đỉnh  $n$  và ma trận kề  $a$ ), bạn muốn tìm đường đi ngắn nhất từ một đỉnh  $S$  (Start) đến một đỉnh  $F$  (Finish) trong đồ thị đó. Hãy viết chương trình tìm đường đi ngắn nhất có thể theo thuật toán Dijkstra. Nếu tìm có đường đi từ đỉnh  $S$  (Start) đến đỉnh  $F$  (Finish) trong đồ thị  $g$  này thì bạn xuất ra đường đi, còn nếu không có đường đi thì bạn thông báo không có đường đi từ đỉnh  $S$  (Start) đến đỉnh  $F$  (Finish).

**Bước 1:** Định nghĩa VOCUC, tạo 1 mảng 1 chiều LuuVet dùng để lưu vết đường đi từ  $S \rightarrow F$ , 1 mảng 1 chiều khác với tên là ChuaXet dùng để đánh dấu đỉnh nào trong đồ thị đã xét rồi, đỉnh nào chưa xét trong quá trình tìm đường đi từ  $S \rightarrow F$ , 1 mảng DoDaiDuongDiToi để lưu lại độ dài nhỏ nhất trong quá trình tìm đường đi từ  $S \rightarrow F$ .

```
#define VOCUC 1000
```

```
int LuuVet[MAX]; // LuuVet[i] = đỉnh liền trước i trên đường đi từ S → i
```

int ChuaXet[MAX]; // ChuaXet[i] = 0 là đỉnh i chưa được xét đến trong quá trình tìm đường đi, còn ChuaXet[i] = 1 là đỉnh i được xét đến rồi trong quá trình tìm đường đi.

int DoDaiDuongDiToi[MAX]; // Lưu lại độ dài nhỏ nhất tới đỉnh i

**Bước 2:** viết hàm TimDuongDiNhoNhat để tìm đỉnh chưa xét có giá trị đường đi nhỏ nhất, rồi tiếp tục thi hành thuật toán, căn cứ vào mảng DoDaiDuongDiToi.

```
int TimDuongDiNhoNhat(DOTHI g)
{
    int li = -1; // nếu không tìm thấy đỉnh nào thỏa điều kiện thì trả về -1
    float p = VOCUC;
    for(int i = 0 ; i < g.n ; i++)
    {
        if(ChuaXet[i] == 0 && DoDaiDuongDiToi[i] < p)
        {
            p = DoDaiDuongDiToi[i];
            li = i;
        }
    }
    return li; // trả về đỉnh chưa xét có giá trị đường đi nhỏ nhất
}
```

**Bước 3:** viết hàm CapNhatDuongDi để tiến hành cập nhập lại giá trị đường đi trong quá trình thi hành thuật toán tại đỉnh u.

```
void CapNhatDuongDi(int u, DOTHI g)
{
    ChuaXet[u] = 1; // đỉnh u đã được chọn nên phải gán giá trị ChuaXet của nó = 1
    for(int v = 0; v < g.n ; v++)
    {
        if(ChuaXet[v]==0 && g.a[u][v] > 0 && g.a[u][v] < VOCUC) // tìm một đỉnh v chưa xét và có cạnh nối từ u → v
        {
            if(DoDaiDuongDiToi[v] > DoDaiDuongDiToi[u] + g.a[u][v]) //nếu như độ dài đường đi tới đỉnh v từ đỉnh khác mà lớn hơn độ dài đường đi tới đỉnh u + cạnh (u,v) thì tiến hành cập nhập lại
            {
                DoDaiDuongDiToi[v] = DoDaiDuongDiToi[u] + g.a[u][v];
            }
        }
    }
}
```

```

        LuuVet[v] = u;
    }
}
}

```

**Bước 4:** viết hàm Dijkstra để tiến hành thuật toán Dijkstra tìm đường đi ngắn nhất từ đỉnh S  $\rightarrow$  đỉnh F.

```

void Dijkstra (int S, int F, DOTHI g)
{
    //Khởi tạo các giá trị cần thiết cho thuật toán
    int i;
    for (i = 0; i < g.n ; i++)
    {
        ChuaXet[i] = 0;
        DoDaiDuongDiToi[i] = VOCUC;
        LuuVet[i] = -1;
    }
    DoDaiDuongDiToi[S] = 0;

    //Thi hành thuật toán Dijkstra
    while(ChuaXet[F] == 0) // trong khi thuật toán tìm đường đi vẫn chưa xét đến
đỉnh F thì tiếp tục
    {
        int u = TimDuongDiNhoNhat(g); // tìm đỉnh mà có độ dài đường đi nhỏ nhất
ở bước hiện tại
        if(u == -1) break; // nếu như không tìm được đỉnh nào thì dừng thuật toán
và kết quả không tìm thấy đường đi. Ngược lại tiến hành cập nhật lại độ dài đường đi.
        CapNhatDuongDi(u,g); // cập nhật lại độ dài đường đi
    }

    if (ChuaXet[F] == 1) //Xuất kết quả đường đi nếu quá trình thi hành thuật toán
đã tới điểm F
    {
        printf("Duong di ngan nhat tu dinh %d den dinh %d la: \n\t",S,F);
        i = F;
        printf("%d ", F);
        while (LuuVet[i] != S)

```

```

    {
        printf("<-%d", LuuVet[i]);
        i = LuuVet[i];
    }
    printf("<-%d\n", LuuVet[i]);
    printf("\tVoi do dai la %d\n", DoDaiDuongDiToi[F]);
}
else // ngược lại thì không có đường đi từ S → F
{
    printf("Khong co duong di tu dinh %d den dinh %d \n", S, F);
}
}

```

**Bước 5:** Code trong hàm main để gọi hàm các hàm tương ứng và chạy.

## 5.4 THUẬT TOÁN FLOYD

Cho  $G=(X,E)$  là một đồ thị có trọng số không âm gồm  $n$  đỉnh. Thuật toán Floyd dùng để tìm đường đi ngắn nhất giữa 2 đỉnh  $S$ (Start) và  $F$ (Finish) trong đồ thị  $G$  như sau:

Gọi  $L$  là ma trận lưu lại độ dài đường đi ngắn nhất từ đỉnh  $i$  đến đỉnh  $j$  trong đồ thị (với qui ước  $L[h,k] = \infty$  nếu không có cạnh nối từ đỉnh  $h$  đến đỉnh  $k$ ).

Ta sử dụng thêm một ma trận  $n \times n$  để lưu vết của quá trình tìm đường đi

$Sau\_Nut[i,j]$ : lưu chỉ số của đỉnh ngay sau  $i$  trên đường đi từ  $i$  đến  $j$ .

Các bước thi hành thuật toán Floyd:

**Bước 1:** (khởi tạo)  $\forall u,v \in X$

Nếu  $e(u,v) > 0$  thì

$$L[u,v] = e(u,v).$$

$$Sau\_Nut[u,v] = v;$$

Ngược lại

$$L[u,v] = +\infty$$

$$Sau\_Nut[u,v] = -1;$$

Cuối nếu

**Bước 2:** Với mỗi đỉnh trung gian  $k$ , tìm cặp  $i, j$  nào đó thỏa mãn  $L[i, j] = 0$  hoặc  $L[i, j] > L[i, k] + L[k, j]$

Nếu thỏa mãn thì

$$L[i, j] = L[i, k] + L[k, j]$$

$$\text{Sau\_Nut}[i, j] = \text{Sau\_Nut}[i, k]$$

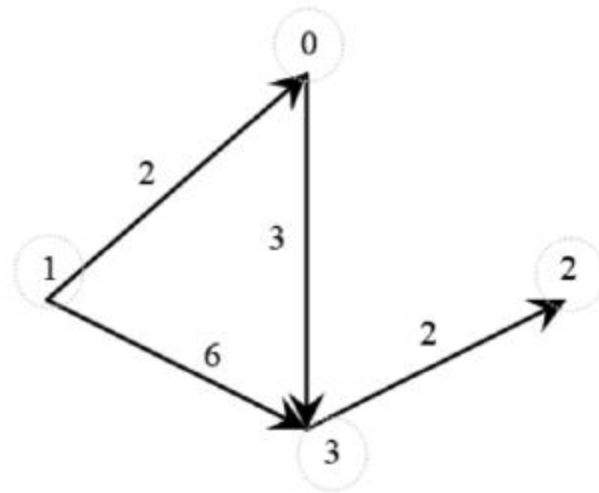
Cuối nếu

Cuối với mọi  $k, i, j$

**Chú ý:**

- Đỉnh  $k$  được gọi là trung gian của  $i, j$  nếu nó có đường đi trực tiếp từ  $i \rightarrow k$  và  $k \rightarrow j$ .
- Khi thuật toán kết thúc, nếu  $L[i, j] = +\infty$  hoặc  $\text{Sau\_Nut}[i, j] = -1$  thì không tồn tại đường đi từ  $i$  đến  $j$ , ngược lại thì  $L[i, j]$  là độ dài đường đi ngắn nhất.

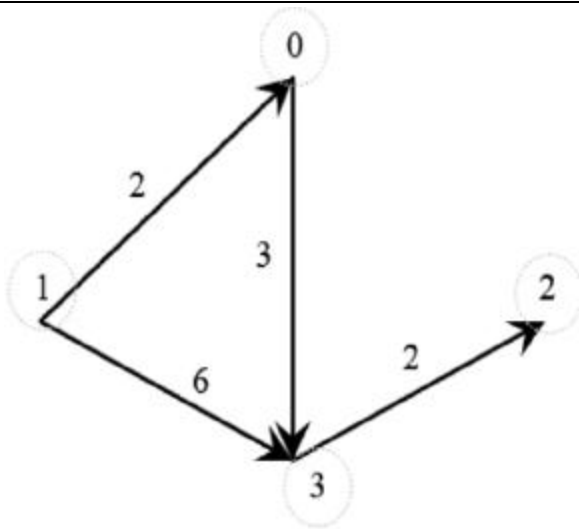
Ví dụ thi hành thuật toán Floyd cho đồ thị sau tìm đường đi ngắn nhất của mọi cặp đỉnh trong đồ thị



**Hình 5.2:** Đồ thị thi hành thuật toán Floyd



## Trạng thái đồ thị



## Xử lý

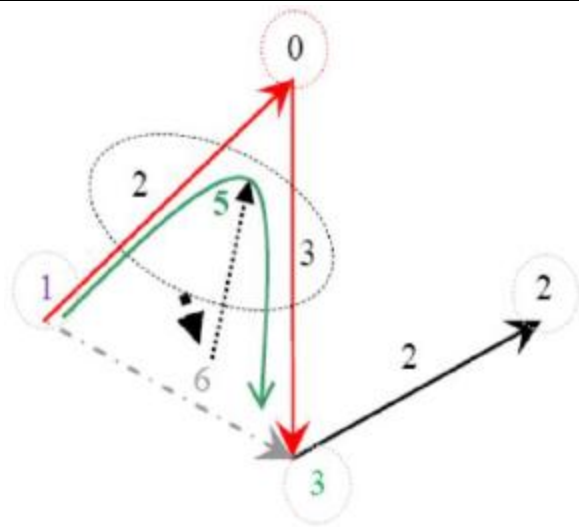
**Bước 1:** khởi tạo

Ma trận trọng số:

L	0	1	2	3
0	$+\infty$	$+\infty$	$+\infty$	3
1	2	$+\infty$	$+\infty$	6
2	$+\infty$	$+\infty$	$+\infty$	$+\infty$
3	$+\infty$	$+\infty$	2	$+\infty$

Sau\_Nut sẽ được khởi tạo giá trị là j nếu có cạnh nối i đến j và được khởi tạo giá trị -1 nếu ngược lại

Sau_Nut	0	1	2	3
0	-1	-1	-1	3
1	0	-1	-1	3
2	-1	-1	-1	-1
3	-1	-1	2	-1

**Bước 2:**Xét đỉnh:  $k = 0$ \*  $[i = 0, j = ?]$ : k không là trung gian.

(?: đại diện cho bất kỳ đỉnh nào trong đồ thị)

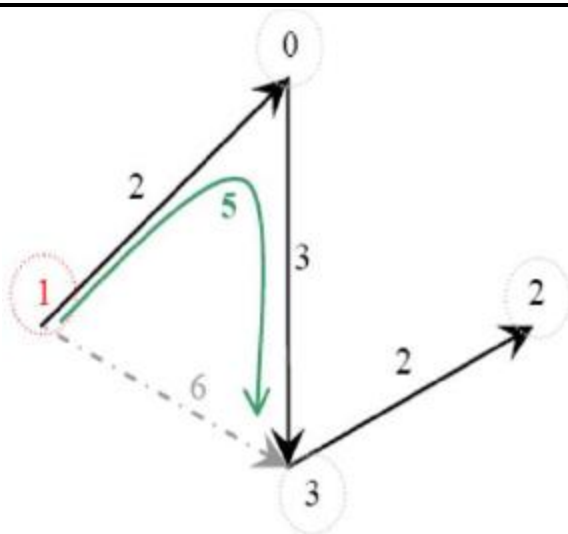
\*  $[i = 1, j = 0]$ : k không là trung gian.\*  $[i = 1, j = 1]$ : k không là trung gian.\*  $[i = 1, j = 2]$ : k không là trung gian.\*  $[i = 1, j = 3]$ :N/x:  $L[i,j] > L[i,k] + L[k,j]$  hay $L[1,3] = 6 > L[1,0] + L[0,3] = 5$  nên: $L[1,3] = L[1,0] + L[0,3] = 5;$  $Sau\_Nut[1,3] = Sau\_Nut[1,0] = 0;$

L	0	1	2	3
0	$+\infty$	$+\infty$	$+\infty$	3
<b>i --&gt; 1</b>	2	$+\infty$	$+\infty$	<b>6 --&gt; 5</b>
2	$+\infty$	$+\infty$	$+\infty$	$+\infty$
3	$+\infty$	$+\infty$	2	$+\infty$

Sau_Nut	0	1	2	3
0	-1	-1	-1	3
<b>i --&gt; 1</b>	0	-1	-1	<b>3 --&gt; 0</b>
2	-1	-1	-1	-1
3	-1	-1	2	-1

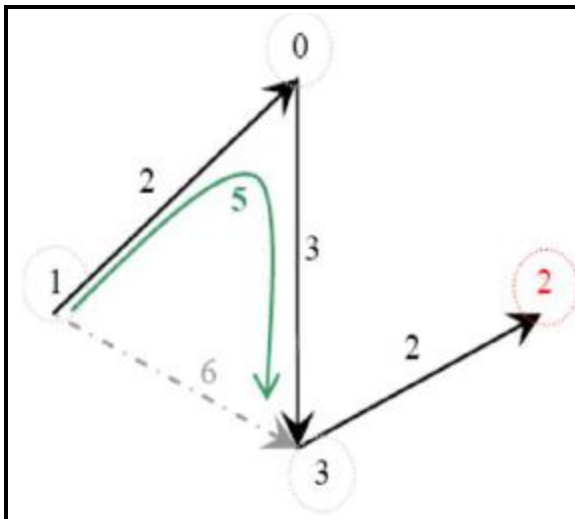
\* [i = 2, j = ?]: k không là trung gian.

\* [i = 3, j = ?]: k không là trung gian.



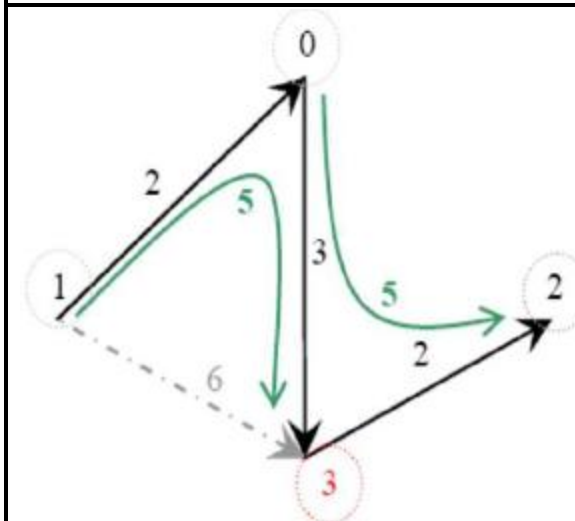
Xét đỉnh: k = 1

\* [i = ?, j = ?]: k không là trung gian.



Xét đỉnh:  $k = 2$

\*  $[i = ?, j = ?]$ :  $k$  không là trung gian.



Xét đỉnh:  $k = 3$

\*  $[i = 0, j = 0]$ :  $k$  không là trung gian.

\*  $[i = 0, j = 1]$ :  $k$  không là trung gian

\*  $[i = 0, j = 2]$ :

Vì  $L[i, j] = 0$  hay  $L[0, 2] = 0$  nên:

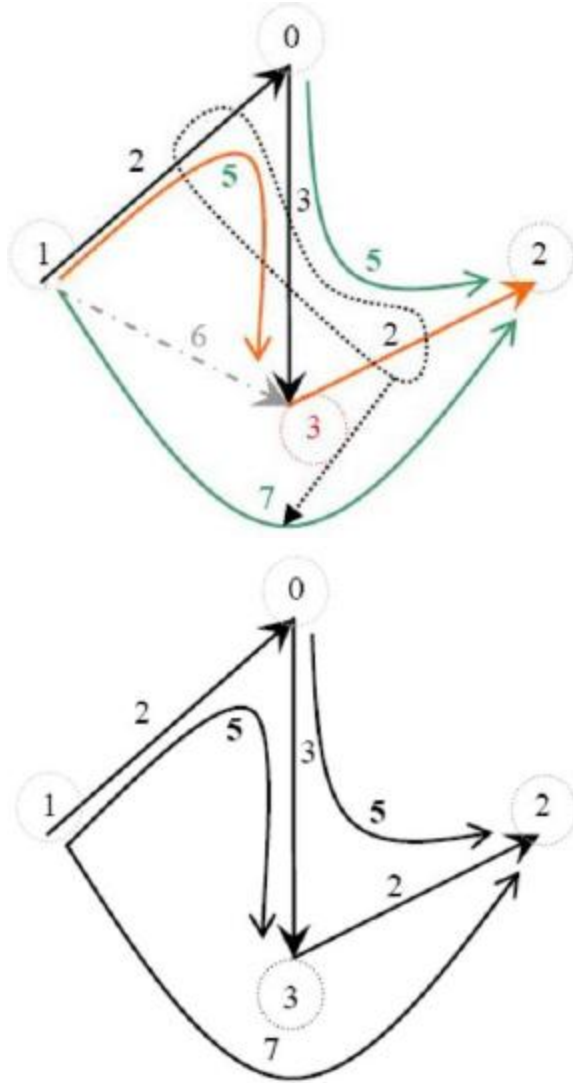
$$L[0, 2] = L[0, 3] + L[3, 2] = 5;$$

$$\text{Sau\_Nut}[0, 2] = \text{Sau\_Nut}[0, 3] = 3$$

L	0	1	2	3
$i \rightarrow 0$	$+\infty$	$+\infty$	$+\infty \rightarrow 5$	3
1	2	$+\infty$	$+\infty$	5
2	$+\infty$	$+\infty$	$+\infty$	$+\infty$
3	$+\infty$	$+\infty$	2	$+\infty$

Sau_Nut	0	1	2	3
$i \rightarrow 0$	-1	-1	$-1 \rightarrow 3$	3
1	0	-1	-1	0
2	-1	-1	-1	-1
3	-1	-1	2	-1

\*  $[i = 0, j = 3]$ :  $k$  không là trung gian



(K = 3 tiếp tục)

\*  $[i=1, j=0]$ : k không là trung gian.

\*  $[i=1, j=1]$ : k không là trung gian.

\*  $[i=1, j=2]$ :

N/x:  $L[1,2] = 0$  nên

$$L[1,2] = L[1,3] + L[3,2] = 5 + 2 = 7;$$

$$\text{Sau\_Nut}[1,2] = \text{Sau\_Nut}[1,3] = 0;$$

L	0	1	2	3
0	$+\infty$	$+\infty$	5	3
$i \rightarrow 1$	2	$+\infty$	$+\infty \rightarrow 7$	5
2	$+\infty$	$+\infty$	$+\infty$	$+\infty$
3	$+\infty$	$+\infty$	2	$+\infty$

Sau_Nut	0	1	2	3
0	-1	-1	3	3
$i \rightarrow 1$	0	-1	$-1 \rightarrow 0$	0
2	-1	-1	-1	-1
3	-1	-1	2	-1

\*  $[i=1, j=3]$ : k không là trung gian.

\*  $[i=2, j=?]$ : k không là trung gian.

\*  $[i=3, j=?]$ :

Kết thúc thuật toán.

## 5.5 HƯỚNG DẪN THI HÀNH TÌM KIẾM ĐƯỜNG ĐI NGẮN NHẤT VỚI THUẬT TOÁN FLOYD

Bạn có một đồ thị  $g$  (gồm đỉnh  $n$  và ma trận kề  $a$ ), bạn muốn tìm đường đi ngắn nhất từ một đỉnh  $S$  (Start) đến một đỉnh  $F$  (Finish) trong đồ thị đó. Hãy viết chương trình tìm đường đi ngắn nhất có thể theo thuật toán Floyd. Nếu tìm có đường đi từ đỉnh  $S$  (Start) đến đỉnh  $F$  (Finish) trong đồ thị  $g$  này thì bạn xuất ra đường đi, còn nếu không có đường đi thì bạn thông báo không có đường đi từ đỉnh  $S$  (Start) đến đỉnh  $F$  (Finish).

**Bước 1:** Định nghĩa VOCUC, tạo 1 mảng 2 chiều Sau\_Nut dùng để lưu vết đường đi từ bất kỳ đỉnh  $i$  đến đỉnh  $j$  nào, 1 mảng 2 chiều khác với tên là  $L$  dùng để lưu lại độ dài đường đi ngắn nhất từ đỉnh  $i$  tới đỉnh  $j$  trong đồ thị.

```
#define VOCUC 1000

int Sau_Nut[MAX][MAX]; // Sau_Nut[i][j] = đỉnh liền sau i trên đường đi từ i → j

int L[MAX][MAX]; // L[i][j] = lưu lại độ dài đường đi ngắn nhất từ đỉnh i tới đỉnh j
trong đồ thị
```

**Bước 2:** viết hàm Floyd để tiến hành thuật toán Floyd tìm đường đi ngắn nhất từ đỉnh giữa hai đỉnh bất kỳ  $i$  và  $j$ .

```
void Floyd(DOTHI g)
{
    int i,j;
    // khởi tạo giá trị cho 2 mảng 2 chiều L và Sau_Nut như trong thuật toán
    for(i = 0; i < g.n ; i++)
    {
        for( j = 0; j < g.n ; j++)
        {
            if(g.a[i][j] > 0) // nếu có cạnh nối đỉnh i với đỉnh j
            {
                Sau_Nut[i][j] = j; // khởi tạo đỉnh liền sau i trên đường tìm đường
                đi từ i tới j là j.
            }
        }
    }
}
```

```

        L[i][j] = g.a[i][j]; // lưu lại độ dài ngắn nhất tại thời điểm hiện tại
        từ điểm i đến đỉnh j là cạnh nối đỉnh i với đỉnh j.
    }
    else // không có cạnh nối từ đỉnh i đến đỉnh j
    {
        Sau_Nut[i][j] = -1; // khởi tạo đỉnh liền sau i trên đường tìm
        đường đi từ i tới j là -1.
        L[i][j] = VOCUC; // lưu lại độ dài ngắn nhất tại thời điểm hiện tại
        từ điểm i đến đỉnh j là VOCUC (không có đường đi).
    }
}

// Thi hành thuật toán Floyd
for(int k = 0; k < g.n ; k++)
{
    for(i = 0 ; i < g.n ; i++)
    {
        for(j = 0; j < g.n ; j++)
        {
            if(L[i][j] > L[i][k] + L[k][j]) // nếu tồn tại một đỉnh trung gian k
            sao cho đường đi từ đỉnh i qua đỉnh k tới đỉnh j ngắn hơn đường đi từ đỉnh i đến đỉnh j
            thì tiến hành chọn đường đi đó.
            {
                L[i][j] = L[i][k] + L[k][j]; // cập nhập lại độ dài đường đi từ i
                tới j.
                Sau_Nut[i][j] = Sau_Nut[i][k]; // lưu lại đỉnh liền sau i trên
                đường tìm đường đi từ i tới j là k.
            }
        }
    }
}

```

```

// xuất kết quả tìm đường đi ngắn nhất từ S --> F
int S,F;
printf ("nhap vao dinh bat dau: ");
scanf("%d",&S);
printf("Nhap vao dinh ket thuc: ");
scanf("%d",&F);

if (Sau_Nut[S][F] == -1) // nếu giá trị Sau_Nut[S][F] là -1 thì điều này có nghĩa
là đỉnh liền sau S là -1 trên đường tìm đường đi từ S tới F. Tương đương với việc
không có đường đi từ đỉnh S đến F.
{
    printf ("Khong co duong di tu dinh %d den dinh %d la :\n",S,F);
}
else // ngược lại có đường đi từ S tới F.

{
    printf ("Duong di ngan nhat tu dinh %d den dinh %d la :\n",S,F);
    printf ("\t%d",S);
    int u = S;
    while(Sau_Nut[u][F] != F) // trong khi giá trị đỉnh liền sau u trên đường tìm
đường đi ngắn nhất từ S tới F không phải là F thì tiếp tục theo vết đi tới đến khi nào
gặp F thì dừng.
    {
        u = Sau_Nut[u][F];
        printf (" --> %d",u);
    }
    printf (" --> %d",F);
    printf ("\n\tVoi tong trong so la %d",L[S][F]);
}
}

```

**Bước 3:** Code trong hàm main để gọi hàm các hàm tương ứng và chạy.

## TÀI LIỆU THAM KHẢO

1. Lê Mậu Gia Bảo (2015). Lý thuyết đồ thị. HUTECH.
2. Reinhard Diestel (2000). *Graph Theory*. Second Edition. Springer-Verlag New York.
3. Robert Sedgewick (1995). *Cẩm nang thuật toán – tập 2 (bản dịch Việt ngữ)*.
4. L. Lovász and K. Vesztergombi (1999). *Discrete Mathematics*. Lecture notes. Yale University. Spring.
5. Nguyễn Trung Trực (1997), *Cấu trúc Dữ liệu*, Khoa CNTT – Đại học Bách Khoa TP.HCM.