BUỔI 2. TÍNH LIÊN THÔNG CỦA ĐỒ THỊ

Mục đích

- *Thực hành cài đặt các thuật toán duyệt đồ thị bằng ngôn ngữ C*
- *Ứng dụng các thuật toán duyệt đồ thị vào các bài toán thực tế*
- *Củng cố lý thuyết, rèn luyện kỹ năng lập trình*

Nội dung

- Duyệt đồ thị
- Kiểm tra tính liên thông của đồ thị vô hướng
- Đếm số bộ phận/thành phần liên thông của đồ thị vô hướng
- Kiểm tra đồ thị có chứa chu trình (có hướng)
- Kiểm tra đồ thị có chứa chu trình (vô hướng)
- Kiểm tra đồ thị phân đôi
- Kiểm tra tính liên thông của đồ thị có hướng
- Cài đặt giải thuật Tarjan để tìm các bộ phận liên thông mạnh của đồ thị có hướng

Yêu cầu

- Biết biểu diễn đồ thị trên máy tính

2.1 Biểu diễn đồ thị

Chọn một phương pháp biểu diễn bất kỳ. Với phương pháp đã chọn, cài đặt ít nhất các hàm sau:

- `init_gaph(n)`: Khởi tạo đồ thị có n đỉnh, 0 cung
- `add_edge(u, v)`: Thêm cung (u, v) vào đồ thị
- `adjacent(u, v)`: Kiểm tra u kề với v
- Liệt kê các đỉnh kề của đỉnh u

```
for (int v = 1;  v <= pG->n; v++)
    if (adjacent(pG, u, v)) {
    }
```

2.2 Duyệt đồ thị

Duyệt đồ thị (Graph traversal) là *đi qua các đỉnh của đồ thị theo một thứ tự nào đó* và để *làm điều gì đó với từng đỉnh đã đi qua*. Có thể chỉ đơn giản là in ra nhãn của các đỉnh theo thứ tự duyệt.

Duyệt đồ thị có thể dùng để:

- Kiểm tra xem một đồ thị vô hướng có liên thông hay không
- Đếm số thành phần liên thông của một đồ thị vô hướng
- Tìm đường đi từ đỉnh s đến đỉnh t (trên đồ thị vô hướng hoặc có hướng)

Phương pháp duyệt đồ thị tổng quát từ 1 đỉnh u bất kỳ gồm các bước chính sau:

- Sử dụng một danh sách L lưu các *đỉnh sắp sửa duyệt*.
- Đưa đỉnh bắt đầu (đỉnh u) vào L
- `while` (L chưa rỗng)
    - lấy một đỉnh trong L ra, làm gì đó với nó (ví dụ: in nó ra màn hình), và *đánh dấu nó đã được duyệt*.
    - xem xét từng đỉnh kề của nó và đưa vào L

Tuỳ theo cấu trúc dữ liệu (CTDL) của L là gì mà ta có phương pháp duyệt khác nhau, ví dụ:

- L là ngăn xếp (stack): duyệt theo chiều sâu
- L là hàng đợi (queue): duyệt theo chiều rộng

Với cách duyệt này ta có thể tìm được *tập các đỉnh có thể đi đến được từ đỉnh u (reachable from u)* hay *một bộ phận liên thông chứa u (của một đồ thị vô hướng)*.

2.2.1 Duyệt theo chiều rộng

Áp dụng phương pháp duyệt đồ thị tổng quát với danh sách L là một hàng đợi (Queue), ta sẽ có phương pháp duyệt theo chiều rộng. Để nhất quán với kiểu dữ liệu, ta đặt tên cho hàng đợi là Q thay vì L. Ta sử dụng thêm một mảng `mark[]` để lưu trạng thái của các đỉnh: đã duyệt (1)/chưa duyệt (0). Mảng `mark[]` nên khai báo toàn cục[1] để có thể duyệt toàn bộ đồ thị.

CTDL hàng đợi (Queue) phải hỗ trợ ít nhất các phép toán sau:

- `make_null_queue(Q)`: tạo hàng đợi rỗng
- `enqueue(Q, u)`: đưa phần tử u vào cuối hàng đợi
- `front(Q)`: trả về phần tử ở đầu hàng đợi
- `dequeue(Q)`: loại bỏ phần tử đầu hàng đợi
- `empty(Q)`: kiểm tra hàng đợi có rỗng hay không

Thuật toán *duyệt đồ thị theo chiều rộng sử dụng hàng đợi* để duyệt đồ thị `pG` bắt đầu từ đỉnh `s` có thể cài đặt như bên dưới.

```
/* Khai báo CTDL Graph*/
...
/* Khai báo CTDL Queue*/
...
/* BFS: Duyệt đồ thị theo chiều rộng */
//Biến hỗ trợ dùng để lưu trạng thái của đỉnh: đã duyệt/chưa duyệt
int mark[MAX_N];

void BFS(Graph *pG, int s) {
      //1. Khai báo hàng đợi Q, khởi tạo rỗng
      Queue Q;
      make_null_queue(&Q);

      //2. Đưa s vào Q, bắt đầu duyệt từ đỉnh s */
      enqueue(&Q, s);

      //3. Vòng lặp chính dùng để duyệt
      while (!empty(&Q)) {

            //3a. Lấy phần tử ở đầu hàng đợi
            int u = front(&Q); dequeue(&Q);
            if (mark[u] != 0)             //u đã duyệt rồi, bỏ qua
                  continue;

            printf("Duyet %d\n", u);      //Làm gì đó trên u
            mark[u] = 1;                  //Đánh dấu nó đã duyệt

            //3b. Xét các đỉnh kề của u, đưa vào hàng đợi Q
            for (int v = 1; v <= pG->n; v++)
                  if (adjacent(pG, u, v))
                        enqueue(&Q, v);
      }
}
```

Trong bước 3b, ta liệt kê các đỉnh kề (không lặp lại) của `pG` bằng cách cho `v` chạy từ đỉnh `1` đến đỉnh `n` và kiểm tra `u` có kề với `v` không bằng hàm `adjacent(pG, u, v)`. Như thế, các đỉnh kề của `u` sẽ được liệt kê theo thứ tự tăng dần. Ta không cần các đỉnh kề của u phải lặp lại. Như thế đồ thị có đa cung hay chứa khuyên cũng được duyệt giống như đơn đồ thị.

[1] Nếu không khai báo toàn cục, có thể khai báo mảng mark[] như là một tham số của của hàm BFS(Graph *pG, int u, int mark[]).

Hãy xem lại các phương pháp cài đặt CTDL Queue (trong học phần Cấu trúc dữ liệu), chọn 1 phương pháp bất kỳ có hỗ trợ các phép toán trên.

Đoạn mã bên dưới là *bản cài đặt (rất) đơn giản* cho CTDL Queue và các phép toán yêu cầu. Bản cài đặt này không tối ưu, không khuyến khích sử dụng bản cài đặt này. Người học hãy tự cài đặt CTDL Queue và các phép toán cần thiết để sử dụng trong hàm duyệt đồ thị.

```
/* Khai báo CTDL Queue*/

#define MAX_SIZE 100
typedef int ElementType;

typedef struct {
      ElementType data[MAX_SIZE];
      int front, rear;
} Queue;


/* Khởi tạo hàng đợi rỗng */
void make_null_queue(Queue *pQ) {
      pQ->front = 0;
      pQ->rear = -1;
}
/* Đưa phần tử u vào cuối hàng đợi */
void enqueue(Queue *pQ, ElementType u) {
      pQ->rear++;
      pQ->data[pQ->rear] = u;
}


/* Xem phần tử đầu hàng đợi */
ElementType front(Queue *pQ) {
      return pQ->data[pQ->front];
}


/* Xoá bỏ phần bỏ đầu hàng đợi */
void dequeue(Queue *pQ) {
      pQ->front++;
}


/* Kiểm tra hàng đợi rỗng */
int empty(Queue *pQ) {
      return pQ->front > pQ->rear;
}
```

Để tạo thành chương trình hoàn chỉnh, ta viết thêm hàm `main()` thực hiện các bước sau:

- Khai báo biến đồ thị `G`
- Đọc dữ liệu đồ thị và dựng đồ thị
- Khởi tạo mảng `mark[u] = 0` với u = 1, 2, …, n
- Gọi `BFS(&G, 1)` để duyệt đồ thị từ đỉnh 1.

```
//Duyệt đồ thị theo chiều rộng từ đỉnh 1
#include <stdio.h>

//Các khai báo và cài đặt
...

int main() {
	//1. Khai báo đồ thị G
	Graph G;
	//2. Đọc dữ liệu và dựng đồ thị
	...
	//3. Khởi tạo mảng mark[u] = 0, với mọi u = 1, 2, ..., n
	for (int u = 1; u <= G.n; u++)
		mark[u] = 0;
	//4. Gọi hàm BFS duyệt theo chiều rộng từ đỉnh 1
	BFS(&G, 1);

	return 0;
}
```

2.2.1.1 Bài tập 1a – BFS từ đỉnh 1 (vô hướng)

Viết chương trình đọc một đồ thị vô hướng từ bàn phím. Duyệt đồ thị theo chiều rộng từ đỉnh 1. In các đỉnh theo thứ tự duyệt ra màn hình.

2.2.1.2 Bài tập 1b – BFS từ đỉnh 1 (có hướng)

Viết chương trình đọc một đồ thị có hướng từ bàn phím. Duyệt đồ thị theo chiều rộng từ đỉnh 1. In các đỉnh theo thứ tự duyệt ra màn hình.

2.2.1.3 Bài tập 1c – BFS từ đỉnh s (vô hướng)

Viết chương trình đọc một đồ thị vô hướng từ bàn phím. Duyệt đồ thị theo chiều rộng từ đỉnh s. đỉnh s cũng được đọc từ bàn phím. In các đỉnh theo thứ tự duyệt ra màn hình.

2.2.1.4 Bài tập 1d – BFS từ đỉnh s (có hướng)

Viết chương trình đọc một đồ thị vô hướng từ bàn phím. Duyệt đồ thị theo chiều rộng từ đỉnh s. đỉnh s cũng được đọc từ bàn phím. In các đỉnh theo thứ tự duyệt ra màn hình.

2.2.2 Duyệt toàn bộ đồ thị

Gọi hàm `BFS(&G, u)` ở trên chỉ cho phép duyệt đỉnh `u` và các đỉnh mà `u` có thể đi đến. Để có thể duyệt toàn bộ các đỉnh của đồ thị, ta sẽ tìm đỉnh chưa được duyệt và gọi `BFS()` để duyệt nó. Quá trình **tìm đỉnh chưa duyệt và gọi BFS()** sẽ lặp lại cho đến khi tất các các đỉnh đều đã được duyệt.

```
//Duyệt toàn bộ đồ thị theo chiều rộng
#include <stdio.h>

//Các khai báo và cài đặt
...

int main() {
	//1. Khai báo đồ thị G
	Graph G;
	//2. Đọc dữ liệu và dựng đồ thị
	...
	//3. Khởi tạo mảng mark[u] = 0, với mọi u = 1, 2, ..., n
	for (int u = 1; u <= G.n; u++)
		mark[u] = 0;
	//4. Duyệt toàn bộ đồ thị: tìm đỉnh chưa duyệt và gọi BFS()
	for (int u = 1; u <= G.n; u++)
		if (mark[u] == 0)	//u chưa duyệt
			BFS(&G, u);	//gọi BFS(&G, u) để duyệt từ u

	return 0;
}
```

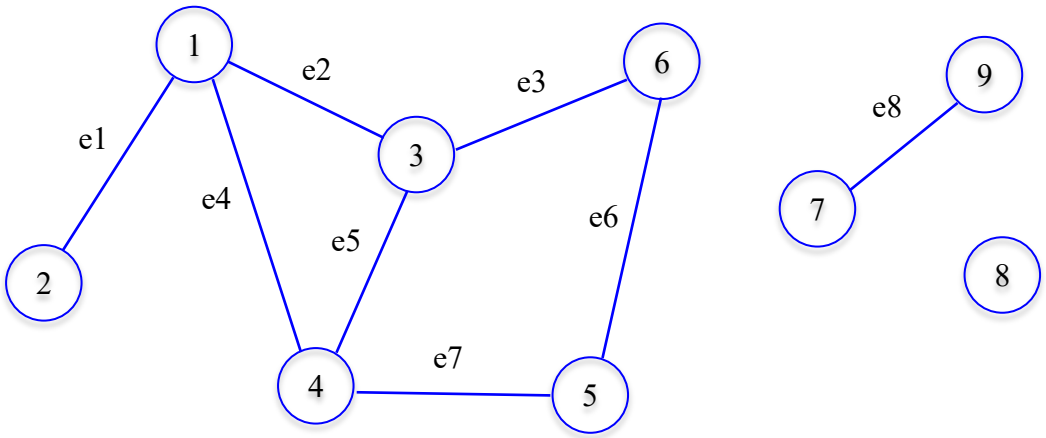
2.2.2.1 Bài tập 2a – BFS toàn bộ đồ thị (vô hướng)

Viết chương trình đọc một đồ thị vô hướng từ bàn phím. Duyệt toàn bộ đồ thị theo chiều rộng với quy tắc chọn đỉnh bắt đầu: *chọn đỉnh chưa duyệt có số thứ tự nhỏ nhất*. In các đỉnh theo thứ tự duyệt ra màn hình.

2.2.2.2 Bài tập 2b – BFS toàn bộ đồ thị (có hướng)

Tương tự bài 2b nhưng với đồ thị có hướng.

Chạy thử các bài tập này với đồ thị bên dưới.



2.2.3 Duyệt theo chiều sâu

Tương tự như duyệt theo chiều rộng, ta sẽ áp dụng phương pháp duyệt đồ thị tổng quát với danh sách L là một ngăn xếp (Stack). Ta sẽ gọi tên ngăn xếp này là S thay vì L.

CTDL ngăn xếp (Stack) phải hỗ trợ ít nhất các phép toán sau:

- `make_null_stack(S)`: tạo ngăn xếp rỗng
- `push(S, u)`: đưa phần tử u vào ngăn xếp
- `top(S)`: trả về phần tử đầu trên ngăn xếp
- `pop(S)`: loại bỏ phần tử đầu ngăn xếp
- `empty(S)`: kiểm tra ngăn xếp có rỗng hay không

Thuật toán *duyệt đồ thị theo chiều sâu sử dụng ngăn xếp* để duyệt đồ thị `pG` bắt đầu từ đỉnh `s` có thể cài đặt như bên dưới.

```
/* Khai báo CTDL Graph*/
...
/* Khai báo CTDL Stack*/
...
/* Duyet do thi theo chieu sau */

//Biến hỗ trợ dùng để lưu trạng thái của đỉnh: đã duyệt/chưa duyệt
int mark[MAX_N];

void DFS(Graph *pG, int s) {
      //1. Khai báo ngăn xếp S, khởi tạo rỗng
      Stack S;
      make_null_stack(&S);

      //2. Đưa s vào S, bắt đầu duyệt từ đỉnh s */
      push(&S, s);

      //3. Vòng lặp chính dùng để duyệt
      while (!empty(&S)) {

            //3a. Lấy phần tử trên đỉnh S ra
            int u = top(&S); pop(&S);
            if (mark[u] != 0)             //u đã duyệt rồi, bỏ qua
                  continue;

            printf("Duyet %d\n", u);      //Làm gì đó trên u
            mark[u] = 1;                  //Đánh dấu nó đã duyệt

            //3b. Xét các đỉnh kề của u, đưa vào ngăn xếp S
            for (int v = 1; v <= pG->n; v++)
                  if (adjacent(pG, u, v))
                        push(&S, v);
      }
}
```

Trong bước 3b, ta liệt kê các đỉnh kề (không lặp lại) của `pG` bằng cách cho `v` chạy từ đỉnh `1` đến đỉnh `n` và kiểm tra `u` có kề với `v` không bằng hàm `adjacent(pG, u, v)`. Như thế, các đỉnh kề của `u` sẽ được liệt kê theo thứ tự tăng dần. Theo nguyên tắc của ngăn xếp, đỉnh vào sau sẽ được lấy ra trước nên *đỉnh kề có thứ tự lớn sẽ được duyệt trước*. Nếu muốn đỉnh có thứ tự nhỏ được duyệt trước hãy cho `v` chạy từ `n` lùi về `1`.

Chú ý: phương pháp này xem các đa cung như đơn cung.

Hãy xem lại các phương pháp cài đặt CTDL Stack (trong học phần Cấu trúc dữ liệu), chọn 1 phương pháp bất kỳ có hỗ trợ các phép toán trên.

Đoạn mã bên dưới là *bản cài đặt (rất) đơn giản* cho CTDL Stack và các phép toán yêu cầu. Không khuyến khích sử dụng bản cài đặt này. Người học nên tự cài đặt phiên bản của riêng mình và sử dụng.

```
/* Khai báo CTDL Stack*/

#define MAX_SIZE 100
typedef int ElementType;
typedef struct {
	ElementType data[MAX_SIZE];
	int top_idx;
} Stack;

/* Hàm khởi tạo ngăn xếp rỗng */
void make_null_stack(Stack *pS) {
	pS->top_idx = -1;
}

/* Hàm thêm phần tử u vào đỉnh ngăn xếp */
void push(Stack *pS, ElementType u) {
	pS->top_idx++;
	pS->data[pS->top_idx] = u;
}

/* Hàm xem phần tử trên đỉnh ngăn xếp */
ElementType top(Stack *pS) {
	return pS->data[pS->top_idx];
}

/* Hàm xoá bỏ phần tử trên đỉnh ngăn xếp */
void pop(Stack *pS) {
	pS->top_idx--;
}

/* Hàm kiểm tra ngăn xếp rỗng */
int empty(Stack *pS) {
	return pS->top_idx == -1;
}
```

Bản cài đặt này lưu các phần tử của ngăn xếp vào mảng `data` từ chỉ số 0 đến MAX_SIZE - 1. Phần tử trên đỉnh ngăn xếp có chỉ số mảng `top_idx`.

Sử dụng hàm `DFS()` để duyệt từ 1 đỉnh hay duyệt cả đồ thị tương tự như `BFS()`.

2.2.3.1 Bài tập 3a – DFS từ đỉnh 1 (vô hướng)

Viết chương trình đọc một đồ thị vô hướng từ bàn phím. Duyệt đồ thị theo chiều sâu từ đỉnh 1. In các đỉnh theo thứ tự duyệt ra màn hình.

2.2.3.2 Bài tập 3b – DFS từ đỉnh 1 (có hướng)

Viết chương trình đọc một đồ thị có hướng từ bàn phím. Duyệt đồ thị theo chiều sâu từ đỉnh 1. In các đỉnh theo thứ tự duyệt ra màn hình.

2.2.3.3 Bài tập 3c – DFS từ đỉnh s (vô hướng)

Viết chương trình đọc một đồ thị vô hướng từ bàn phím. Duyệt đồ thị theo chiều sâu từ đỉnh s. đỉnh s cũng được đọc từ bàn phím. In các đỉnh theo thứ tự duyệt ra màn hình.

2.2.3.4 Bài tập 3d – DFS từ đỉnh s (có hướng)

Viết chương trình đọc một đồ thị vô hướng từ bàn phím. Duyệt đồ thị theo chiều sâu từ đỉnh s. đỉnh s cũng được đọc từ bàn phím. In các đỉnh theo thứ tự duyệt ra màn hình.

2.2.3.5 Bài tập 4a – DFS toàn bộ đồ thị (vô hướng)

Viết chương trình đọc một đơn đồ thị vô hướng từ bàn phím. Duyệt toàn bộ đồ thị theo chiều sâu với quy tắc chọn đỉnh bắt đầu: *chọn đỉnh chưa duyệt có số thứ tự nhỏ nhất*. In các đỉnh theo thứ tự duyệt ra màn hình.

2.2.3.6 Bài tập 4b – DFS toàn bộ đồ thị (có hướng)

Tương tự bài 2b nhưng với đơn đồ thị có hướng.

2.2.4 Duyệt theo chiều sâu bằng phương pháp đệ quy

Ta có thể áp dụng kỹ thuật đệ quy để duyệt đồ thị theo chiều sâu mà không cần đến Stack S. Phương pháp này ngắn hơn và cài đặt nhanh hơn phương pháp sử dụng ngăn xếp và thường được sử dụng.

```
/* Khai báo CTDL Graph*/
...
/* Duyệt đồ thị theo chiều sâu dùng đệ quy */
//Biến hỗ trợ dùng để lưu trạng thái của đỉnh: đã duyệt/chưa duyệt
int mark[MAX_N];

void DFS(Graph *pG, int u) {
      //1. Đánh dấu u đã duyệt
      printf("Duyet %d\n", u);      //Làm gì đó trên u
      mark[u] = 1;                  //Đánh dấu nó đã duyệt
      //2. Xét các đỉnh kề của u
      for (int v = 1; v <= pG->n; v++)
            if (adjacent(pG, u, v) && mark[u] == 0) //Nếu v chưa duyệt
                  DFS(&S, v);                       //gọi đệ quy duyệt nó
}
```

Đơn giản hơn bản cài đặt dùng ngăn xếp phải không? Sử dụng nó giống hệt phiên bản dùng ngăn xếp.

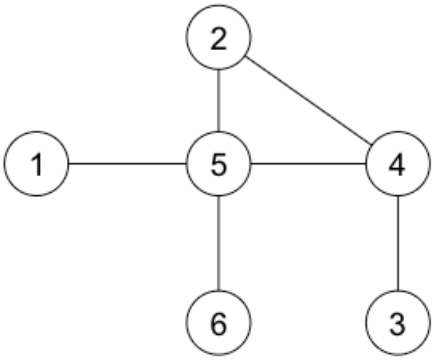
2.2.4.1 Bài tập 5 – DFS đệ quy

Làm lại các bài tập 3 & 4 bằng phương pháp DFS đệ quy. So sánh thứ tự các đỉnh được in ra màn hình so với phương pháp DFS dùng ngăn xếp. Muốn có kết quả giống nhau thì phương pháp DFS dùng ngăn xếp sẽ phải điều chỉnh như thế nào?

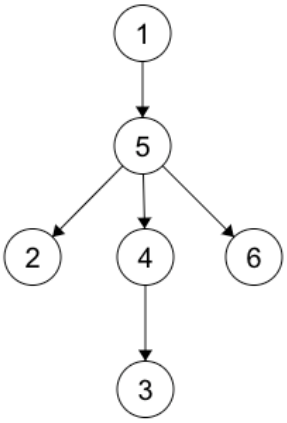
2.2.5 Dựng cây duyệt đồ thị

Quá trình duyệt đồ thị sẽ sinh ra cây duyệt đồ thị.

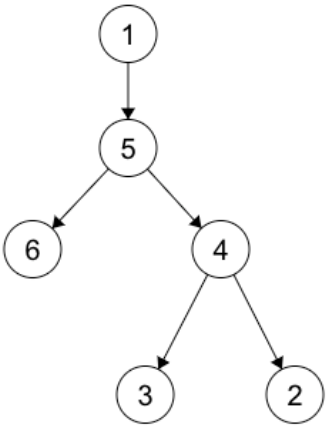
Ví dụ: cho đồ thị như bên dưới.



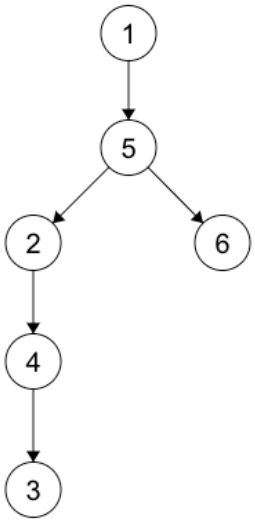
Vẽ cây duyệt đồ thị theo chiều rộng và theo chiều sâu.



Cây BFS



Cây DFS (duyệt lớn trước, nhỏ sau)



Cây DFS (nhỏ trước, lớn sau)

Để có được cây này, ta cần thêm một mảng `parent[]` để lưu đỉnh cha của các đỉnh.

Với phương pháp duyệt đồ thị dùng hàng đợi hoặc ngăn xếp, mỗi phần tử của hàng đợi/ngăn xếp là một cấu trúc gồm 2 trường: u và p.

```
/* Khai báo CTDL Queue*/
#define MAX_SIZE 100
typedef struct {
	int u, p;
} ElementType;

typedef struct {
	ElementType data[MAX_SIZE];
	int front, rear;
} Queue;
```

Trong hàm BFS/DFS

- Mỗi khi lấy phần tử ra, ta lấy ra 1 cặp (u, p)
- Mỗi khi đánh dấu đỉnh thì ta cho `parent[u] = p`
- Mỗi khi đưa đỉnh v vào hàng đợi/ngăn xếp ta thay bằng đưa cặp (v, u) vào

```
/* BFS: Duyệt đồ thị theo chiều rộng */
int mark[MAX_N];        //Lưu trạng thái của các đỉnh
int parent[MAX_N];      //Lưu cha của các đỉnh

void BFS(Graph *pG, int s) {
      //1. Khai báo hàng đợi Q, khởi tạo rỗng
      Queue Q;
      make_null_queue(&Q);

      //2. Đưa s vào Q, bắt đầu duyệt từ đỉnh s */
      ElementType pair;
      pair.u = s; pair.p = -1;
      enqueue(&Q, pair);

      //3. Vòng lặp chính dùng để duyệt
      while (!empty(&Q)) {

            //3a. Lấy phần tử ở đầu hàng đợi
            ElementType pair = front(&Q); dequeue(&Q);
            int u = pair.u, p = pair.p;
            if (mark[u] != 0)             //u đã duyệt rồi, bỏ qua
                  continue;

            printf("Duyet %d\n", u);      //Làm gì đó trên u
            mark[u] = 1;                  //Đánh dấu nó đã duyệt
            parent[u] = p;                //Đánh dấu nó đã duyệt


            //3b. Xét các đỉnh kề của u, đưa vào hàng đợi Q
            for (int v = 1; v <= pG->n; v++)
                  if (adjacent(pG, u, v)) {
                        ElementType pair;
                        pair.u = v; pair.p = u;
                        enqueue(&Q, pair);
                  }
      }
}
```

Với phương pháp duyệt đồ thị đệ quy, ta bổ sung thêm 1 tham số p.

- Mỗi khi đánh dấu đỉnh thì ta cho `parent[u] = p`
- Khi gọi đệ quy để duyệt v, ta truyền u là cha của v: `DFS(pG, v, u)`.

```
/* Dựng cây duyệt đồ thị theo chiều sâu dùng đệ quy */
int mark[MAX_N];        //Lưu trạng thái của các đỉnh
int parent[MAX_N];      //Lưu cha của các đỉnh

void DFS(Graph *pG, int u, int p) {
      //1. Đánh dấu u đã duyệt
      mark[u] = 1;                  //Đánh dấu nó đã duyệt
      parent[u] = p;                //Cho cha của u là p

      //2. Xét các đỉnh kề của u
      for (int v = 1; v <= pG->n; v++)
            if (adjacent(pG, u, v) && mark[v] == 0) //Nếu v chưa duyệt
                  DFS(pG, v, u);                    //gọi đệ quy duyệt nó
}
```

Trước khi duyệt đồ thị, ta khởi tạo `parent[u] = -1`. Sau khi duyệt đồ thị xong, cây duyệt đồ thị được lưu trong mảng `parent[u]`.

2.2.5.1 Bài tập 6a – Dựng cây BFS duyệt đồ thị (vô hướng)

Viết chương trình đọc vào một đồ thị vô hướng từ bàn phím. Duyệt đồ thị theo chiều rộng từ đỉnh 1. Dựng cây duyệt đồ thị bằng cách in ra các đỉnh cha của các đỉnh.

2.2.5.2 Bài tập 6b – Dựng cây BFS duyệt đồ thị (có hướng)

Viết chương trình đọc vào một đồ thị có hướng từ bàn phím. Duyệt đồ thị theo chiều rộng từ đỉnh 1. Dựng cây duyệt đồ thị bằng cách in ra các đỉnh cha của các đỉnh.

2.2.5.3 Bài tập 6c – Dựng cây DFS duyệt đồ thị (vô hướng)

Viết chương trình đọc vào một đồ thị vô hướng từ bàn phím. Duyệt đồ thị theo chiều sâu từ đỉnh 1. Quy ước: khi xét các đỉnh kề, đỉnh có số thứ tự nhỏ sẽ duyệt trước. Dựng cây duyệt đồ thị bằng cách in ra các đỉnh cha của các đỉnh.

2.2.5.4 Bài tập 6d – Dựng cây DFS duyệt đồ thị (có hướng)

Viết chương trình đọc vào một đồ thị có hướng từ bàn phím. Duyệt đồ thị theo chiều sâu từ đỉnh 1. Quy ước: khi xét các đỉnh kề, đỉnh có số thứ tự nhỏ sẽ duyệt trước. Dựng cây duyệt đồ thị bằng cách in ra các đỉnh cha của các đỉnh.

2.3 Tính liên thông của đồ thị vô hướng

2.3.1 Kiểm tra đồ thị liên thông

Duyệt đồ thị (theo BFS hay DFS) có thể dùng để kiểm tra đồ thị vô hướng có liên thông không. Ta bắt đầu duyệt đồ thị từ một đỉnh bất kỳ. Sau khi duyệt xong đồ thị nếu tất cả các đỉnh đều được duyệt thì đồ thị sẽ liên thông, ngược lại sẽ không liên thông.

Quy trình kiểm tra đồ thị vô hướng liên thông gồm các bước sau:

- Khởi tạo tất cả các đỉnh đều chưa duyệt mark[u] = 0
- Gọi `BFS()` hoặc `DFS()` để duyệt đồ thị từ đỉnh 1
- Kiểm tra xem có còn đỉnh nào chưa duyệt không (`mark[u] == 0`). Nếu còn thì đồ thị không liên thông, ngược lại thì liên thông.

```
// Kiểm tra pG có liên thông không
int connected(Graph *pG) {
	//1. Khởi tạo tất cả đỉnh đều chưa duyệt
	for (int u = 1; u <= pG->n; u++)
		mark[u] = 0;
	//2. Duyệt đồ thị từ đỉnh bất kỳ, ví dụ: 1
	DFS(pG, 1);
	//3. Kiểm tra xem có đỉnh nào chưa duyệt không
	for (int u = 1; u <= pG->n; u++)
		if (mark[u] == 0) //Vẫn còn đỉnh chưa duyệt
			return 0;   //Đồ thị không liên thông, thoát luôn

	return 1;               //Tất cả các đỉnh đều đã duyệt => liên thông
}
```

2.3.2 Bài tập 7 – Kiểm tra liên thông (vô hướng)

Viết chương trình đọc một đồ thị vô hướng từ bàn phím. Nếu đồ thị liên thông in ra: "CONNECTED", ngược lại in ra "DISCONNECTED".

2.3.3 Đếm số bộ phận liên thông

Áp dụng phương pháp duyệt toàn bộ đồ thị, ta cũng có thể *đếm được số bộ phận liên thông của đồ thị vô hướng*. Cứ mỗi lần tìm được 1 đỉnh chưa duyệt, gọi DFS/BFS để duyệt nó, tăng số bộ phận liên thông lên 1.

Quy trình duyệt như sau:

- Khởi tạo tất cả các đỉnh đều chưa duyệt mark[u] = 0 với mọi u = 1, 2, ..., n
- Khởi tạo biến đếm `cnt = 0`
- Cho u chạy từ 1 đến n
    - Nếu u chưa duyệt, gọi `BFS()` hoặc `DFS()` để duyệt từ đỉnh u. Duyệt xong tăng biến đếm cnt lên 1.

Kết thúc quá trình duyệt, biến `cnt` chứa số BPLT của đồ thị.

```
//Đếm số BPLT của đồ thị vô hướng
#include <stdio.h>

//Các khai báo và cài đặt
...

int main() {
      ...
      //1. Khởi tạo mảng mark[u] = 0, với mọi u = 1, 2, ..., n
      for (int u = 1; u <= G.n; u++)
            mark[u] = 0;
      //2. Khởi tạo biến đếm
      int cnt = 0;
      //3. Duyệt toàn bộ đồ thị để đếm số BPLT
      for (int u = 1; u <= G.n; u++)
            if (mark[u] == 0) {     //u chưa duyệt
                  DFS(&G, u);       //gọi BFS(&G, u) để duyệt từ u
                  cnt++;
            }

      ...

      return 0;
}
```

2.3.4 Bài tập 8a – Số BPLT

Viết chương trình đọc một đồ thị vô hướng từ bàn phím và đếm số thành phần liên thông của đồ thị.

2.3.5 Bài tập 8b – Số đỉnh của 1 BPLT

Viết chương trình đọc một đồ thị vô hướng từ bàn phím. In ra số đỉnh của bộ phận liên thông chứa 1.

Gợi ý:

- Khai báo thêm một biến đếm (toàn cục) `nb_u`.
- Trong hàm DFS/BFS, mỗi lần đánh dấu đỉnh u, tăng biến đếm `nb_u` lên 1.
- Trong hàm `main()`, trước khi gọi hàm `DFS(&G, 1)`, khởi tạo 1 biến đếm `nb_u = 0`
- Sau khi gọi hàm xong, `nb_u` sẽ chứa số đỉnh của BPLT chứa 1.

2.3.6 Bài tập 8c – Số đỉnh của 1 BPLT chứa s

Viết chương trình đọc một đồ thị vô hướng từ bàn phím. In ra số đỉnh của bộ phận liên thông chứa s. Đỉnh s được đọc từ bàn phím.

Gợi ý:

- Làm tương tự bài 8b, thay 1 bằng s.

2.3.7 Bài tập 8d – Tìm BPLT có nhiều đỉnh nhất

Viết chương trình đọc một đồ thị vô hướng từ bàn phím. In ra số đỉnh của BPLT có nhiều đỉnh nhất.

Gợi ý: Sử dụng biến `max_cnt` dùng để chứa số đỉnh nhiều nhất.

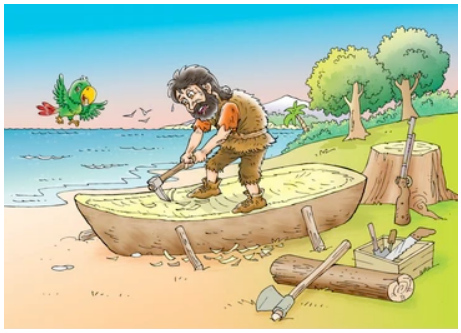
- Khai báo thêm một biến đếm (toàn cục) `nb_u`.
- Trong hàm DFS/BFS, mỗi lần đánh dấu đỉnh u, tăng biến đếm `nb_u` lên 1.
- Trong hàm `main()`, sử dụng đoạn chương trình duyệt toàn bộ đồ thị, trước khi gọi hàm `DFS(&G, u)`, khởi tạo 1 biến đếm `nb_u = 0`
- Sau khi gọi hàm xong, `nb_u` sẽ chứa số đỉnh của BPLT chứa u.
- So sánh `nb_u` với `kq`. Nếu `nb_u > max_cnt` thì cho `max_cnt = nb_u`

2.3.8 Ứng dụng tính liên thông của duyệt đồ thị

Phương pháp chung để giải các bài toán ứng dụng LTĐT là:

- Mô hình hoá bài toán về đồ thị
  - Đỉnh là gì?
  - Cung là gì?
  - Vô hướng hay có hướng?
- Chuyển yêu cầu của bài toán thành yêu cầu trên đồ thị
- Chọn thuật toán
- Lấy kết quả thuật toán, biến đổi, trả về lời giải cho bài toán gốc

2.3.8.1 Bài tập 9a – Qua đảo



Có n hòn đảo và m cây cầu. Mỗi cây cầu bắt qua 2 hòn đảo. Một hôm chúa đảo tự hỏi là với các cây cầu hiện tại thì đứng ở một hòn đảo bất kỳ có thể nào đi đến được tất cả các hòn đảo khác hay không mà không cần phải dùng thuyền.

Là người rất giỏi về lý thuyết đồ thị, hãy giúp chúa đảo viết chương trình kiểm tra.

Đầu vào (Input)

Dữ liệu đầu vào được nhập từ bàn phím với định dạng:

- Dòng đầu tiên chứa 2 số nguyên n và m, tương ứng là số đảo và số cây cầu.
- m dòng tiếp theo mỗi dòng chứa 2 số nguyên u v nói rằng có 1 cây cầu bắt qua hai hòn đảo u và v.

Đầu ra (Output)

- Nếu có thể đi được in ra màn hình YES, ngược lại in ra NO.

Ví dụ:

Với đầu vào

```
4 3
1 2
2 3
3 4
```

Ta có kết quả:

```
YES
```

Gợi ý:

- Nếu đồ thị liên thông in ra YES, ngược lại in ra NO.
- Đưa về đồ thị:
  - Đỉnh: hòn đảo
  - Cung: cây cầu nối hai hòn đảo u, v
  - Đồ thị vô hướng
- Yêu cầu bài toán: *từ 1 hòn đảo, có thể đi đến được tất cả các hòn đảo khác không (dĩ nhiên là đi qua các cây cầu)?* chuyển thành yêu cầu trên đồ thị: *từ 1 đỉnh, có thể đi đến tất cả các đỉnh khác trong đồ thị không? = đồ thị có liên thông không?*
- Thuật toán trên đồ thị: *kiểm tra đồ thị vô hướng liên thông*
- Nếu đồ thị liên thông in ra YES, ngược lại in ra NO

2.3.8.2 Bài tập 9b – Tôn Ngộ Không

Tôn Ngộ Không là một trong các nhân vật chính của truyện Tây du ký. Khi còn ở Hoa Quả Sơn, Tôn Ngộ Không là vua của loài khỉ (Mỹ Hầu Vương). Hoa Quả Sơn có rất nhiều cây ăn trái, nên loài khỉ rất thích. Do đặc tính của mình, khỉ không thích đi mà chỉ thích chuyền từ cây này sang cây khác. Tuy nhiên, nếu khoảng cách giữa hai cây quá xa thì chúng không thể chuyền qua lại được.

Đường đường là vua của loài khỉ, Tôn Ngộ Không muốn vạch ra một kế hoạch hái trái cây trên tất cả các cây có trên Hoa Quả Sơn mà không cần phải nhảy xuống đất. Tôn Ngộ Không dự định sẽ bắt đầu leo lên một cây, hái trái của cây này, sau đó chuyền qua một cây kế tiếp hái trái của này và tiếp tục như thế cho đến khi tất cả các cây đều được hái trái. Một cây có thể được chuyền qua chuyền lại nhiều lần.

Hãy giúp Tôn Ngộ Không kiểm tra xem kế hoạch này có thể thực hiện được không.

Đầu vào

Giả sử số lượng cây ăn trái ở Hoa Quả Sơn là n cây và được đánh số từ 1 đến n. Dữ liệu đầu vào được nhập từ bàn phím với định dạng:

- Dòng đầu tiên chứa 2 số nguyên n và m, là số cây và số cặp cây có thể chuyền qua lại.
- m dòng tiếp theo mỗi dòng chứa 2 số nguyên u v, cách nhau 1 khoảng trắng, nói rằng có thể chuyền từ cây u sang cây v hoặc chuyền từ cây v sang cây u.

Đầu ra

- Nếu kế hoạch của Tôn Ngộ Không có thể thực hiện được DUOC, ngược lại in ra KHONG.

Ví dụ:

```
4 3
2 1
1 3
3 4
```

Tôn Ngộ Không bắt đầu từ cây 1, chuyền qua cây 2, sau đó chuyền ngược về 1, chuyền tiếp sang 3 và sau cùng là sang 4.

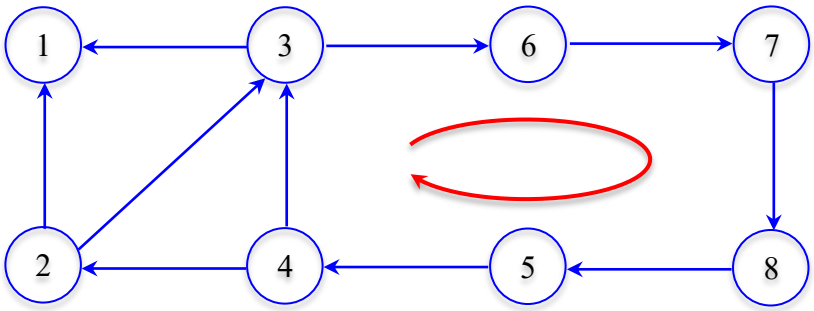
Gợi ý:

- Đưa về đồ thị:
  - Đỉnh: cây
  - Cung: hai cây u, v có thể chuyền qua lại với nhau được
  - Đồ thị vô hướng
- Yêu cầu bài toán: *từ 1 cây, có thể chuyền qua chuyền lại để đi đến tất cả các cây không?* chuyển thành yêu cầu trên đồ thị: *từ 1 đỉnh, có thể đi đến tất cả các đỉnh khác trong đồ thị không? = đồ thị có liên thông không?*
- Thuật toán trên đồ thị: *kiểm tra đồ thị vô hướng liên thông*

2.4 Kiểm tra đồ thị chứa chu trình

Nhắc lại: *Chu trình (cycle) là một đường đi (path) có đỉnh đầu trùng đỉnh cuối.*

Cho đồ thị G = <V, E>, làm thế nào để kiểm tra G có chứa chu trình hay không?



Ý tưởng: sử dụng phương pháp duyệt đồ thị theo chiều sâu (cài đặt bằng đệ quy) kết hợp với tô màu các đỉnh.

- Ta sẽ tô màu các đỉnh theo trạng thái của nó
  - Chưa được duyệt: trắng (WHITE)
  - Đang duyệt (chưa duyệt xong các đỉnh kề): xám (GRAY)
  - Đã duyệt xong: đen (BLACK)
- Khởi tạo tất cả các đỉnh đều có màu trắng

Hàm đệ quy DFS(u) gồm các bước:

- Tô màu màu xám (đang duyệt) cho u và
- Xét các đỉnh kề v của u:
  - Nếu v có màu trắng => gọi đệ quy duyệt v
  - Nếu đỉnh kề v nào đó có màu xám, có chu trình: v → … → u → v.
  - Nếu v có màu đen => duyệt xong rồi, bỏ qua

2.4.1 Kiểm tra đồ thị có hướng chứa chu trình

Ta cài đặt thuật toán DFS kết hợp với tô màu để kiểm đồ thị có hướng chứa chu trình như bên dưới.

```
/* Kiểm tra đồ thị chứa chu trình */
#define WHITE 0
#define GRAY  1
#define BLACK 2

int color[MAX_N];       //Lưu trạng thái của các đỉnh
int has_circle;         //Đồ thị chứa trình hay không

void DFS(Graph *pG, int u) {
      //1. Tô màu dang duyệt cho u
      color[u] = GRAY;


      //2. Xét các đỉnh kề của u
      for (int v = 1; v <= pG->n; v++)
            if (adjacent(pG, u, v)) {
                  if (color[v] == WHITE)        //2a. Nếu v chưa duyệt
                        DFS(pG, v, u);          //gọi đệ quy duyệt nó
                  else if (color[v] == GRAY)    //2b. v đang duyệt
                        has_circle = 1;         //chứa chu trình
            }
      //3. Tô màu duyệt xong cho u
      color[u] = BLACK;
}
```

Áp dụng phương pháp duyệt toàn bộ đồ thị để kiểm tra chu trình.

```
//Kiểm tra đồ thị có hướng chứa chu trình
#include <stdio.h>

//Các khai báo và cài đặt
...

int main() {
	...
	//1. Khởi tạo mảng color[u] = WHITE, với mọi u = 1, 2, ..., n
	for (int u = 1; u <= G.n; u++)
		color[u] = WHITE;
	//2. Khởi tạo biến has_circle
	has_circle = 0;
	//3. Duyệt toàn bộ đồ thị để kiểm tra chu trình
	for (int u = 1; u <= G.n; u++)
		if (color[u] == WHITE)  //u chưa duyệt
			DFS(&G, u);       //gọi DFS(&G, u) để duyệt từ u
	//4. Kiểm tra has_circle
	...

	return 0;
}
```

2.4.1.1 Bài tập 10a – Kiểm tra đồ thị chứa chu trình (có hướng)

Viết chương trình đọc một đồ thị có hướng từ bàn phím và kiểm tra xem đồ thị đó có chứa chu trình không. Nếu có, ghi ra "CIRCLED", ngược lại ghi ra "NO CIRCLE".

2.4.1.2 Bài tập 10b* – Các đỉnh của chu trình (có hướng)

Viết chương trình đọc một đồ thị có hướng từ bàn phím và kiểm tra xem đồ thị đó có chứa chu trình không. Nếu đồ thị có chu trình, in ra các đỉnh có trong chu trình trên cùng một dòng, các đỉnh cách nhau một khoảng trắng. Nếu đồ thị có nhiều chu trình, chỉ cần in ra 1 chu trình bất kỳ. Nếu đồ thị không chứa chu trình, in ra -1.

Gợi ý: sử dụng thêm mảng `parent[]` để lưu cha của các đỉnh trong quá trình duyệt, giống như phần dựng cây. Khi phát hiện có chu trình, hãy lưu lại đỉnh bắt đầu chu trình `start = u` và `end = v`. Sau khi duyệt kết thúc, lần ngược từ `start`, `parent[start], parent[parent[start]], ..., end` để lưu các đỉnh có trong chu trình vào một mảng. Sau cùng in các phần tử này ra màn hình theo hướng ngược lại.

2.4.2 Kiểm tra đồ thị vô hướng chứa chu trình

Để kiểm tra đồ thị VÔ HƯỚNG, ta điều chỉnh một chút ở hàm duyệt: *kiểm tra xem đỉnh kề của một đỉnh có phải là đỉnh cha của đỉnh đang xét không*. Khi xét v, ta kiểm tra v xem nó có phải là đỉnh cha của u (đỉnh vừa mới duyệt trước khi duyệt u) không, nếu phải thì bỏ qua.

```
/* Kiểm tra đồ thị chứa chu trình */
#define WHITE 0
#define GRAY  1
#define BLACK 2

int color[MAX_N];       //Lưu trạng thái của các đỉnh
int has_circle;         //Đồ thị chứa trình hay không

void DFS(Graph *pG, int u, int p) {
      //1. Tô màu đang duyệt cho u
      color[u] = GRAY;


      //2. Xét các đỉnh kề của u
      for (int v = 1; v <= pG->n; v++)
            if (adjacent(pG, u, v)) {
                  if (v == p)                   //2a. Nếu v là cha của p
                        continue;               //bỏ qua
                  if (color[v] == WHITE)        //2b. Nếu v chưa duyệt
                        DFS(pG, v, u);          //gọi đệ quy duyệt nó
                  else if (color[v] == GRAY)    //2c. v đang duyệt
                        has_circle = 1;         //chứa chu trình
            }
      //3. Tô màu duyệt xong cho u
      color[u] = BLACK;
}
```

Áp dụng phương pháp duyệt toàn bộ đồ thị để kiểm tra chu trình.

```
//Các khai báo và cài đặt
...

int main() {
      ...
      //3. Duyệt toàn bộ đồ thị để kiểm tra chu trình
      for (int u = 1; u <= G.n; u++)
            if (color[u] == WHITE)  //u chưa duyệt
                  DFS(&G, u, -1);   //u là đỉnh bắt đầu, nó không có cha
      //4. Kiểm tra has_circle
      ...

      return 0;
}
```

2.4.2.1 Bài tập 10c – Kiểm tra đồ thị chứa chu trình (vô hướng)

Viết chương trình đọc một đồ thị có hướng từ bàn phím và kiểm tra xem đồ thị đó có chứa chu trình không. Nếu có, ghi ra "CIRCLED", ngược lại ghi ra "NO CIRCLE".

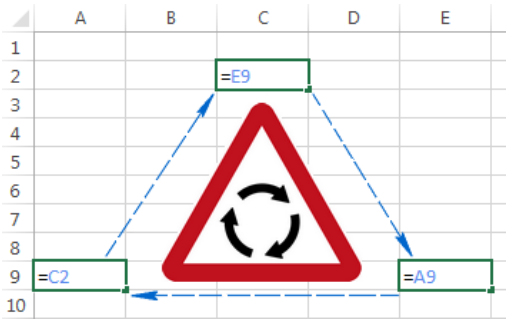
2.4.2.2 Bài tập 10d* – Kiểm tra đồ thị chứa chu trình (vô hướng)

Tương tự bài 10b nhưng với đồ thị vô hướng.

2.4.3 Ứng dụng kiểm tra chu trình

Các bài tập tiếp theo ứng dụng thuật toán kiểm tra chu trình để giải các bài toán thực tế.

2.4.3.1 Bài tập 11a – Phát hiện tham chiếu vòng trong Excel



Microsoft Excel là chương trình xử lý bảng tính nằm trong bộ Microsoft Office của hãng phần mềm Microsoft. Excel được thiết kế để giúp ghi lại, trình bày các thông tin xử lý dưới dạng bảng, thực hiện tính toán và xây dựng các số liệu thống kê trực quan.

Bảng tính Excel được chia thành nhiều hàng và cột. Giao điểm của hàng và cột gọi là một ô. Trong một ô của Excel, ta có thể nhập một giá trị hoặc một công thức có sử dụng giá trị của 1 hay nhiều ô khác. Khi ô A1 sử dụng giá trị của ô B2 ta nói A1 tham chiếu đến B2 hay A1 phụ thuộc vào B2.

Một lỗi thường gặp trong Excel là *tham chiếu vòng* (circular reference). Tham chiếu vòng là trường hợp một ô tham chiếu đến chính nó hoặc như ví dụ bên dưới: A9 tham chiếu đến C2, C2 tham chiếu đến E9 và E9 tham chiếu đến A9 (xem hình).

Vấn đề đặt ra là làm thế nào để phát hiện có tham chiếu vòng trong bảng tính. Giả sử bảng tính có n ô, để đơn giản, ta đánh số các ô đang xét là 1, 2, 3, ..., n. Ta biết được ô nào tham chiếu đến (các) ô nào. Hãy viết chương trình kiểm tra xem có xuất hiện tham chiếu vòng hay không.

Đầu vào (Input)

Dữ liệu đầu vào được nhập từ bàn phím với định dạng:

- Dòng đầu tiên chứa 2 số nguyên n và m, tương ứng là số ô và số tham chiếu.
- m dòng tiếp theo mỗi dòng chứa 2 số nguyên u, v nói rằng ô u tham chiếu đến ô v.

Đầu ra (Output)

- In ra màn hình CIRCULAR REFERENCE nếu bảng tính có tham chiếu vòng, ngược lại in ra OK.

Ví dụ:

```
5 6
1 2
2 5
5 3
3 2
1 4
4 3
```

Kết quả:

```
CIRCULAR REFERENCE
```

Gợi ý:

- Mô hình hoá về đồ thị
  - Đỉnh: ô
  - Cung (u, v): ô u tham chiếu v
  - Đồ thị có hướng
- Yêu cầu bài toán: phát hiện *tham chiếu vòng* chuyển thành phát hiện *đồ thị có chứa chu trình*
- Thuật toán: kiểm tra đồ thị có hướng chứa chu trình

2.4.3.2 Bài tập 11b – Thuyền trưởng Haddock

Thuyền trưởng Haddock (một trong những nhân vật chính trong truyện Những cuộc phiêu lưu của Tintin) là một người luôn say xỉn. Vì thế đôi khi Tintin không biết ông ta đang say hay tỉnh. Một ngày nọ, Tintin hỏi ông ta về cách uống. Haddock nói như thế này: Có nhiều loại thức uống (soda, wine, water, …), tuy nhiên Haddock lại tuân theo quy tắc "*để uống một loại thức uống nào đó cần phải uống tất cả các loại thức uống tiên quyết của nó*". Ví dụ: để uống rượu (wine), Haddock cần phải uống soda và nước (water) trước. Vì thế muốn say cũng không phải dễ!

Cho danh sách các thức uống và các thức uống tiên quyết của nó. Hãy xét xem thuyền trưởng Haddock có thể nào say không? Để làm cho Haddock say, ông ta phải uống hết tất cả các thức uống.

Ví dụ 1:

```
soda wine
water wine
```

Thức uống tiên quyết được cho dưới dạng a b, có nghĩa là để uống b bạn phải uống a trước. Trong ví dụ trên để uống wine, Haddock phải uống soda và water trước. Do soda và water không có thức uống tiên quyết nên Haddock sẽ SAY.

Ví dụ 2:

```
soda wine
water wine
wine water
```

Để uống wine, cần uống water. Tuy nhiên để uống water lại cần wine. Vì thế Haddock không thể uống hết được các thức uống nên ông ta KHÔNG SAY.

Để đơn giản ta có thể giả sử các thức uống được mã hoá thành các số nguyên từ 1, 2, … và dữ liệu đầu vào được cho trong tập tin có dạng như sau, ví dụ:

```
3 2
1 2
3 2
```

Có loại thức uống (soda: 1, wine: 2 và water: 3) và có 2 điều kiện tiên quyết

1 → 2 và 3 → 2.

Viết chương trình đọc một tập tin mô tả thức uống và các điều kiện tiên quyết như trên, và in ra màn hình "YES" nếu Haddock có thể say, ngược lại in ra "NO".

Có thể kiểm thử với danh sách các thức uống sau:

```
5 3
1 2
2 3
3 4
4 2
5 4
```

Mô hình bài toán về đồ thị

Rõ ràng bài toán này có thể mô hình về dạng đồ thị có hướng:

- Đỉnh tương ứng với các thức uống và
- Mỗi điều kiện tiên quyết ứng với 1 cung.

Để có thể uống hết tất cả các thức uống thì **đồ thị này phải không chứa chu trình**.

Áp dụng thuật toán LTĐT

- Áp dụng thuật toán kiểm tra đồ thị có hướng chứa chu trình
- Nếu đồ thị chứa chu trình in ra: NO, ngược lại in ra YES.

2.5 Kiểm tra đồ thị phân đôi

Để kiểm tra tính phân đôi của đồ thị ta tô màu các đỉnh của đồ thị bằng hai màu: xanh (BLUE) và đỏ (RED) sao cho *hai đỉnh kề nhau có màu khác nhau*. Nếu tô màu được, thì đồ thị đang xét là đồ thị phân đôi, ngược lại đồ thị không thể phân đôi. Ta cũng sẽ áp dụng duyệt đồ thị kết hợp với tô màu như trường hợp kiểm tra đồ thị chứa chu trình.

Mỗi đỉnh sẽ có 1 trong 3 trạng thái ứng với 3 màu:

- Chưa có màu: NO_COLOR
- Được tô màu xanh: BLUE
- Được tô màu đỏ: RED

Thuật toán duyệt đồ thị đệ quy kết hợp tô màu `colorize(u, c)` gồm các bước sau

- Tô màu c cho u
- Xét các đỉnh kề v của u, có 3 trường hợp:
  - Nếu v chưa có màu => Gọi đệ quy tô màu ngược lại với c cho nó
  - Nếu v đã có màu và giống màu với u => đụng độ, không tô được
  - Nếu v đã có màu và khác màu với u => bỏ qua

Tìm màu ngược lại của màu c.

Gọi S = BLUE + RED

Để tìm màu ngược lại của c, ta lấy S trừ cho c, cụ thể:

- Ngược lại của BLUE là S – BLUE = RED
- Ngược lại của RED là S – RED = BLUE

Thuật toán tô màu đệ quy được cài đặt như sau:

```
//Thuật toán kiểm tra đồ thị chứa chu trình
#define NO_COLOR 0
#define BLUE     1
#define RED      2

//Các biến hỗ trợ
int color[MAX_N];       //Lưu trạng thái của các đỉnh
int conflict;           //Có đụng độ trong khi tô màu không

//Tô màu các đỉnh của đồ thị bắt đầu từ đỉnh u với màu c
void colorize(Graph *pG, int u, int c) {
      //1. Tô màu c cho u
      color[u] = c;
      //2. Xét các đỉnh kề của u
      for (int v = 1; v <= pG->n; v++)
            if (adjacent(pG, u, v)) {
                  if (color[v] == NO_COLOR)      //2a. Nếu v chưa có màu
                        colorize(pG, v, 3 - c);  //tô màu nó ngược với c
                  else if (color[v] == color[u]) //2b. u và v  cùng màu
                        conflict = 1;            //đụng độ, không tô được
            }
}
```

Khởi tạo tất cả các đỉnh đều không có màu (`NO_COLOR`). Để kiểm tra trên BPLT chứa u, ta gọi `colorize(&G, u, BLUE)`. Để kiểm tra trên toàn bộ đồ thị, hãy áp dụng phương pháp duyệt toàn bộ đồ thị.

2.5.1 Bài tập 12a - Kiểm tra đồ thị phân đôi

Viết chương trình đọc vào một đồ thị vô hướng từ bàn phím. Kiểm tra xem đồ thị có phải là đồ thị phân đôi không. Nếu có in ra **YES**, ngược lại in ra **NO**.

2.5.2 Bài tập 12b – Phân chia đội bóng (ứng dụng)

David là huấn luyện viên của một đội bóng gồm N thành viên. David muốn chia đội bóng thành hai nhóm. Để tăng tính đa dạng của các thành viên trong nhóm, David quyết định không xếp hai thành viên đã từng thi đấu với nhau vào chung một nhóm. Bạn hãy lập trình giúp David phân chia đội bóng.

Dữ liệu đầu vào có dạng:

```
3 2
1 2
2 3
```

Dòng đầu tiên (3 2) mô tả *số thành viên trong đội* (N = 3) và *số cặp các thành viên đã từng thi đấu chung với nhau* (M = 2). M dòng tiếp theo mô tả các cặp cầu thủ đã từng thi đấu chung với nhau. Ví dụ: thành viên 1 đã từng thi đấu chung với thành viên 2; thành viên 2 đã từng thi đấu chung với thành viên 3.

Nếu có thể phân chia được, in các thành viên của từng nhóm ra màn hình theo mẫu.

```
Nhom 1: 1 2 5
Nhom 2: 3 4
```

- In các thành viên theo thứ tự tăng dần, cách nhau 1 khoảng trắng.
- Quy ước: thành viên 1 nằm trong nhóm 1.

Nếu phân chia không được, in ra IMPOSSIBLE.

Gợi ý:

- Mô hình hoá bài toán về dạng đồ thị
- Áp dụng phương pháp kiểm tra đồ thị phân đôi: *hai cầu thủ đã từng thi đấu chung sẽ không nằm trong một nhóm*.

Sử dụng các mẫu dữ liệu bên dưới để kiểm tra chương trình của mình.

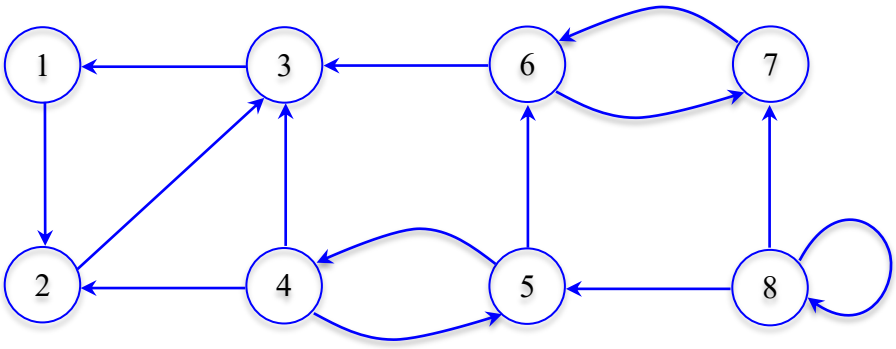
```
3 3
1 2
2 3
3 1
```

```
9 8
1 2
1 3
1 4
1 5
1 6
1 7
1 8
1 9
```

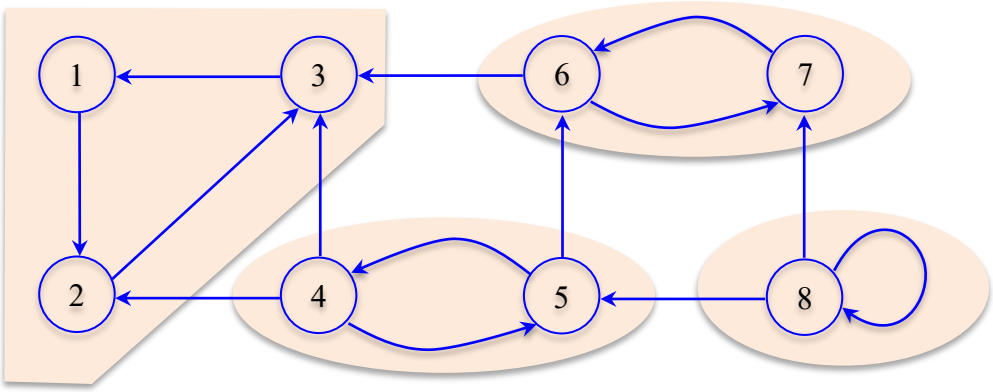
2.6 Tính liên thông của đồ thị có hướng

Đồ thị có hướng G = <V, E> được gọi là liên thông nếu với mọi cặp đỉnh (u, v) luôn tồn tại đường đi từ u đến v. Đồ thị G như thế được gọi là đồ thị *liên thông mạnh* (strong connected). Nếu đồ thị G không liên thông mạnh, ta có thể chia đồ thị G thành các bộ phận mà mỗi bộ phận đều liên thông mạnh.

Ví dụ, xét đồ thị G như hình vẽ bên dưới.



Rõ ràng G không liên thông mạnh. G có 4 thành phần liên thông mạnh:



2.6.1 Thuật toán Tarjan tìm các BPLT mạnh

Thuật toán Tarjan (Robert Tarjan, 1972) sử dụng chiến lược *duyệt đồ thị theo chiều sâu kết hợp với đánh số các đỉnh* để xác định các bộ phận thông mạnh của đồ thị có hướng.

Trong quá trình duyệt và đánh số, ta cần một số biến (toàn cục) hỗ trợ:

- `num[v]`: lưu chỉ số của đỉnh v trong quá trình duyệt
- `min_num[v]`: lưu chỉ số nhỏ nhất của nút có thể đi đến được từ v (trong quá trình duyệt v theo chiều sâu).
- `S`: ngăn xếp, lưu các theo thứ tự được duyệt. Các đỉnh còn trong S là các đỉnh chưa thuộc về BPLT mạng nào.
- `on_stack[v]`: v có đang ở trong stack S hay không
- `k`: chỉ số đỉnh (tăng dần trong quá trình duyệt)

Thuật toán bắt đầu duyệt từ đỉnh u, gán chỉ số cho nó và đưa nó vào S. Sau đó, xét các đỉnh kề của nó. Có ba trường hợp xảy ra đối với đỉnh kề v của u:

- Nếu v chưa được duyệt => gọi đệ quy để duyệt nó. Sau khi duyệt v xong, xem xét cập nhật lại `min_num[u] = min(min_num[u], min_num[v])`.
- Ngược lại, có nghĩa là v đã được duyệt rồi. Nếu nó vẫn còn đang nằm trong ngăn xếp, ta tìm được chu trình v → … u → v. Như thế, u và v sẽ thuộc về một BPLTM mạnh. Ta cập nhật lại `min_num[u] = min(min_num[u], num[v])`.
- Ngược lại, v này đã thuộc về một BPLT mạnh khác => không cần làm gì cả.

Sau khi xét xong tất cả các đỉnh kề của u, nếu num[u] = min_num[u], có nghĩa là ta đã tìm được 1 vòng lặp bắt đầu từ u và đi lòng vòng sau đó trở về u. Các đỉnh trong vòng này chính là một bộ phận liên thông mạnh. Ta sẽ loại bỏ các đỉnh này ra khỏi S. Đỉnh u được gọi là *đỉnh khớp* hay *đỉnh cắt* (articulation/cut vertex).

```
//Thuật toán Tarjan tìm các BPLT mạnh từ đỉnh u

// Các biến hỗ trợ
int num[MAX_N], min_num[MAX_N];
int k;
Stack S;
int on_stack[MAX_N];


//Duyệt đồ thị bắt đầu từ đỉnh u
void SCC(Graph *pG, int u) {

	//1. Đánh số u, đưa u vào ngăn xếp S
	num[u] = min_num[u] = k; k++;
	push(&S, u);
	on_stack[u] = 1;

	//2. Xét các đỉnh kề của u
	for (int v = 1; v <= pG->n; v++) {
		if (adjacent(pG, u, v)) {
			if (num[v] < 0) {
				SCC(pG, v);
				min_num[u] = min(min_num[u], min_num[v]);
			} else if (on_stack[v])
				min_num[u] = min(min_num[u], num[v]);
		}
	}

	//3. Kiểm tra u có phải là đỉnh khớp
	if (num[u] == min_num[u]) {
		printf("Tim duoc BPLT manh, %d la dinh khop.\n", u);
		int w;
		do {  //Lấy các đỉnh trong S ra cho đến khi gặp u
			w = top(&S);
			pop(&S);
			on_stack[w] = 0;
		} while (w != u);
	}
}
```

Mỗi lần gọi hàm SCC(u) ta sẽ tìm được các BPLT mạnh từ các đỉnh đến được từ u. Để tìm tất cả các BPLT mạnh trên đồ thị, hãy sử dụng phương pháp duyệt toàn bộ đồ thị.

```
//Sử dụng thuật toán Tarjan để duyệt toàn bộ đồ thị
int main() {
	...
	//1. Khởi tạo
	for (int u = 1; u <= G.n; u++)//1a. Tất cả đều chưa duyệt
		num[u] == -1;
	k = 1;                        //1b. Tất cả đều chưa duyệt
	make_null_stack(&S);          //1c. Làm rỗng ngăn xếp
	//2. Duyệt toàn bộ đồ thị để tìm BPLT mạnh
	for (int u = 1; u <= G.n; u++)
		if (num[u] == -1)         //u chưa duyệt
			SCC(&G, u);           //duyệt nó
	...

	return 0;
}
```

2.6.2 Bài tập

2.6.2.1 Bài tập 13a – Áp dụng thuật toán Tarjan

Viết chương trình đọc đồ thị có hướng, áp dụng thuật toán thuật Tarjan và in ra các giá trị num[] và min_num[] của từng đỉnh của toàn bộ đồ thị.

2.6.2.2 Bài tập 13b – Kiểm tra đồ thị liên thông mạnh

Viết chương trình đọc đồ thị và kiểm tra xem nó có liên thông mạnh hay không. Nếu có in ra "STRONG CONNECTED", ngược lại in ra "DISCONNECTED".

Gợi ý: *trong quá trình duyệt đồ thị, nếu số đỉnh lấy ra từ stack bằng với n thì đồ thị liên thông mạnh. Ta cũng có thể đếm số BPLT mạnh. Nếu số BPLT mạnh = 1 thì đồ thị liên thông mạnh.*

2.6.2.3 Bài tập 13c – Số BPLT mạnh

Viết chương trình đọc đồ thị có hướng, in ra số lượng BPLT mạnh của đồ thị.

Gợi ý: *mỗi khi tìm được một BPLT mạnh* (`if (num[u] == min_num[u])`) *tăng biến đếm lên 1.*

2.6.2.4 Bài tập 13d – BPLT mạnh lớn nhất

Viết chương trình đọc đồ thị có hướng, tìm BPLT mạnh có nhiều đỉnh nhất.

Gợi ý: *mỗi khi tìm được một BPLT mạnh* (`if (num[u] == min_num[u])`), *đếm số đỉnh có trong BPLT mạnh này (trong lúc lấy các đỉnh trong ngăn xếp ra) và so sánh với* `max_cnt`.

2.6.2.5 Bài tập 14a – Come and Go (ứng dụng)

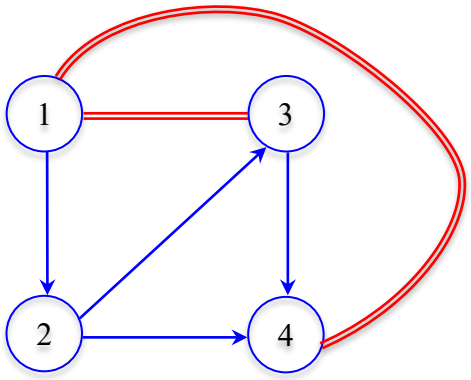
Come and Go (nguồn: UVA Online Judge, Problem 11838)

Trong một thành phố có n địa điểm được nối với nhau bằng các con đường 1 chiều lẫn 2 chiều. Yêu cầu tối thiểu của một thành phố là từ địa điểm này bạn phải có thể đi đến một địa điểm khác bất kỳ.

Hãy viết chương trình kiểm tra xem các con đường của thành phố có thoả mãn điều kiện này không. Nếu có in ra **YES**, ngược lại in ra **NO**.

Dữ liệu đầu vào được cho trong tập tin có dạng như sau:
Ví dụ 1:

```
4 5
1 2 1
1 3 2
2 4 1
3 4 1
4 1 2
```



Trong ví dụ này, có n = 4 địa điểm và 5 con đường, mỗi con đường có dạng a b p, trong đó a, b là các địa điểm; và nếu p = 1, con đường đang xét là đường 1 chiều, ngược lại nó là đường 2 chiều.

Gợi ý:

- Xây dựng đồ thị có hướng từ dữ liệu các con đường và các địa điểm
  - Địa điểm ~ đỉnh
  - Đường 1 chiều ~ cung
  - Đường 2 chiều ~ 2 cung
- Áp dụng giải thuật kiểm tra đồ thị có liên thông mạnh hay không.

2.6.2.6 Bài tập 14b – Trust groups

(nguồn: UVA Online Judge, Problem 11709)

Phòng nhân sự của tổ chức *Association of Cookie Monsters* (ACM) nhận thấy rằng gần đây hiệu quả làm việc của các nhân viên có chiều hướng giảm sút. Vì thế họ đã lấy ý kiến các nhân viên và phát hiện ra nguyên nhân của vấn đề này, đó là: sự tin cậy. Một số nhân viên không tin cậy vào các nhân viên khác trong nhóm làm việc của mình. Điều này làm giảm động lực và niềm vui trong công việc của các nhân viên.

Phòng nhân sự muốn giải quyết triệt để vấn đề này nên họ quyết định tổ chức lại các nhóm làm việc sao cho ổn định. Một nhóm làm việc sẽ ổn định khi mà những người trong nhóm tin cậy lẫn nhau. Họ đã hỏi các nhân viên và biết được những người mà một nhân viên tin cậy trực tiếp. Ngoài ra, sự tin cậy có tính bắt cầu: *nếu A tin cậy B và B tin cậy C thì A cũng sẽ tin cậy C*. Lẽ dĩ nhiên, một nhân viên sẽ tự tin cậy chính bản thân mình. Tuy nhiên, cần chú ý là sự tin cậy lại không có tính đối xứng: *A tin cậy B thì không nhất thiết B phải tin cậy A*.

Phòng nhân sự muốn tổ chức thành ít nhóm nhất có thể. Hãy lập trình để giúp họ.

Giả sử các nhân viên được đánh số là 1, 2, …, n.

Đầu vào (Input)

Dữ liệu đầu vào được nhập từ bàn phím theo định dạng:

- Dòng đầu tiên chứa 2 số nguyên n và m, tương ứng là số nhân viên và số cặp nhân viên có mối quan hệ tin cậy trực tiếp.
- m dòng tiếp theo, mỗi dòng chứa 2 số nguyên a b, nói rằng người a tin cậy vào người b.

Đầu ra (Output)

- In ra màn hình số lượng nhóm ít nhất mà những người trong nhóm đều tin cậy lẫn nhau.