BUỔI 1. BIỂU DIỄN ĐỒ THỊ

Mục đích:

- Biểu diễn đồ thị trên máy tính
- Cài đặt cấu trúc dữ liệu đồ thị (Graph) và một số phép toán cơ bản trên đồ thị

Yêu cầu:

- Biết sử dụng ngôn ngữ lập trình C. Ngôn ngữ thực hành chính thức là ngôn ngữ C
- Biết cài đặt các cấu trúc dữ liệu cơ bản

1.1 Cấu trúc dữ liệu đồ thị và các phép toán

Để có thể lưu trữ đồ thị vào máy tính, ta phải xác định những thông tin cần thiết để biểu diễn đồ thị và các số phép toán trên đồ thị cần phải hỗ trợ.

Cho đồ thị G = <V, E> có n đỉnh, m cung. Các thông tin cần lưu trữ của đồ thị bao gồm:

- Đỉnh: *tên/nhãn đỉnh* và các thông tin khác liên quan đến đỉnh (vị dụ vị trí đỉnh)
- Cung: *hai đỉnh đầu mút (endpoints) của cung, nhãn của cung* (tên cung/trọng số cung/chiều dài cung) và các thông tin khác liên quan tới cung (ví dụ: hình dạng cung)

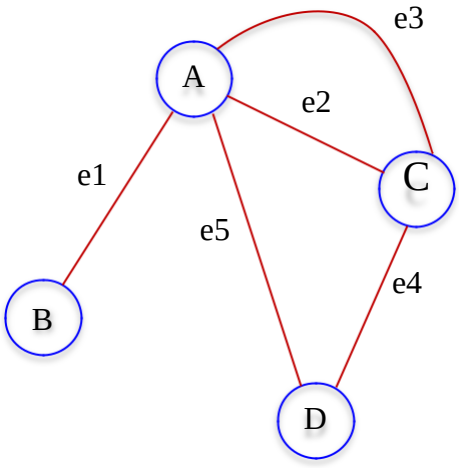
Các phép toán cơ bản trên đồ thị bao gồm:

- init_graph(G, n, m): khởi tạo đồ thị có n đỉnh (và m cung)
- adjacent (G, u, v): kiểm tra xem v có phải là đỉnh kề của u không (u kề với v)
- neighbors(G, u): trả về danh sách các đỉnh kề của u, hoặc liệt kê các đỉnh của u
- add_edge(G, u, v): thêm cung (u, v) vào đồ thị nếu có chưa tồn tại
- remove_edge(G, u, v): xoá cung (u, v) ra khỏi đồ thị
- degree(G, u): trả về bậc của đỉnh u

Xét một số cách biểu diễn đồ thị trên máy tính sau.

1.2 Danh sách cung (edge list)

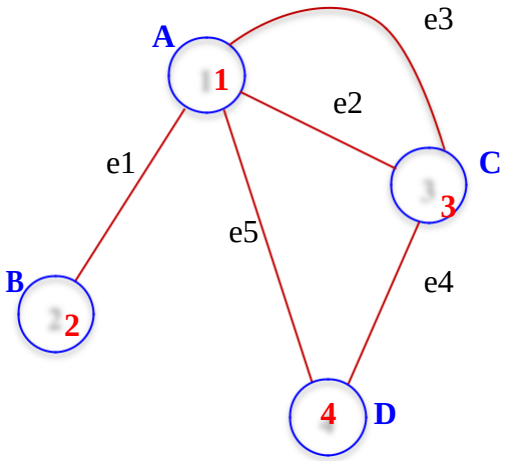
Ví dụ: Xét đồ thị như bên dưới.



Ta có:

- Các đỉnh bao gồm: A, B, C, D
- Các cung bao gồm: e1: (A, B), e2: (A, C), e3: (A, C), e4: (C, D) và e5: (A, D)

Để dễ dàng hơn cho việc lưu trữ các cung trên máy tính, ta đánh số các đỉnh: A:1, B: 2, C: 3, B: 4



Như vậy để lưu trữ các đỉnh ta chỉ cần **một biến n** để lưu số **4** là đủ (có nghĩa là có 4 đỉnh, được đánh số là 1, 2, 3, 4). Nếu muốn lưu trữ thêm nhãn của đỉnh, ta chỉ cần một mảng 1 chiều.

Chỉ số mảng	1	2	3	4
Nhãn của đỉnh	A	B	C	D

Trong tài liệu thực hành này, để đơn giản hóa việc lưu trữ, ta bỏ qua nhãn của các đỉnh.

Khi đó, mỗi cung sẽ được biểu bằng 2 số nguyên (u, v) tương ứng với chỉ số của hai đỉnh đầu mút của cung. Nếu cung là khuyên, thì u = v.
Các cung của đồ thị trong ví dụ trên sẽ trở thành:

STT	**Tên/nhãn cung**	**Các cung trên đồ thị**		**Các cung trên máy tính**	
		Đầu mút 1	**Đầu mút 2**	**u**	**v**
1	e1	A	B	1	2
2	e2	A	C	1	3
3	e3	A	C	1	3
4	e4	C	D	3	4
4	e5	A	D	1	4

Tóm lại, các bước để biểu diễn đồ thị bằng danh sách cung như sau:

- Đánh số các đỉnh 1, 2, …, 3
- Mỗi cung lưu 2 đỉnh đầu mút (endpoints) của nó
- Lưu tất cả các cung của đồ thị vào một danh sách (danh sách đặc hoặc danh sách liên kết)

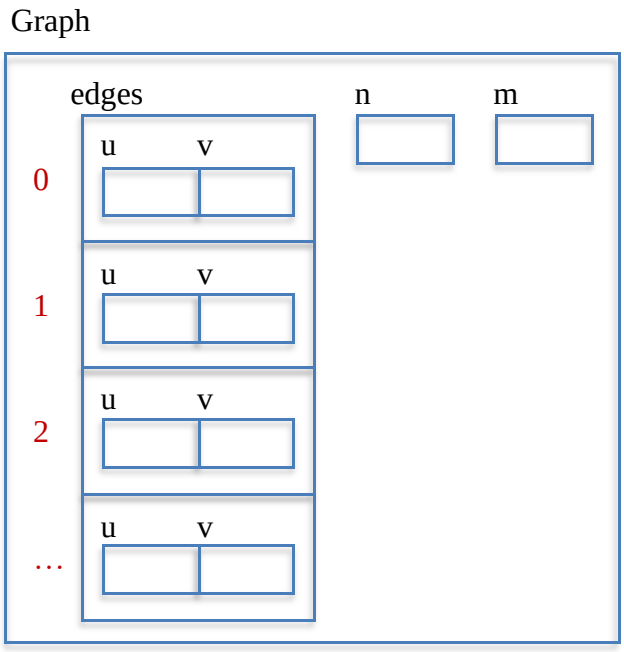
Phương pháp này lưu được tất cả các loại đồ thị.

1.2.1 Cài đặt

Từ các phân tích trên, ta đề xuất một cấu trúc dữ liệu **Graph** để lưu trữ đồ thị bằng phương pháp danh sách cung như sau:

```
//Định nghĩa hằng MAX_M: số cung tối đa đồ thị có thể chứa
#define MAX_M 500


//Định nghĩa cấu trúc dữ liệu Edge biểu diễn 1 cung (u, v)
typedef struct {
	//Mỗi cung lưu đỉnh đầu u, đỉnh cuối v
	int u, v;
} Edge;

//Định nghĩa cấu trúc dữ liệu Graph biểu diễn 1 đồ thị
typedef struct {
	//n: đỉnh, m: cung
	int n, m;

	//Mảng edges lưu các cung của đồ thị
	Edge edges[MAX_M];
} Graph;
```

Sơ đồ tổ chức dữ liệu của cấu trúc dữ liệu **Graph** như sau:

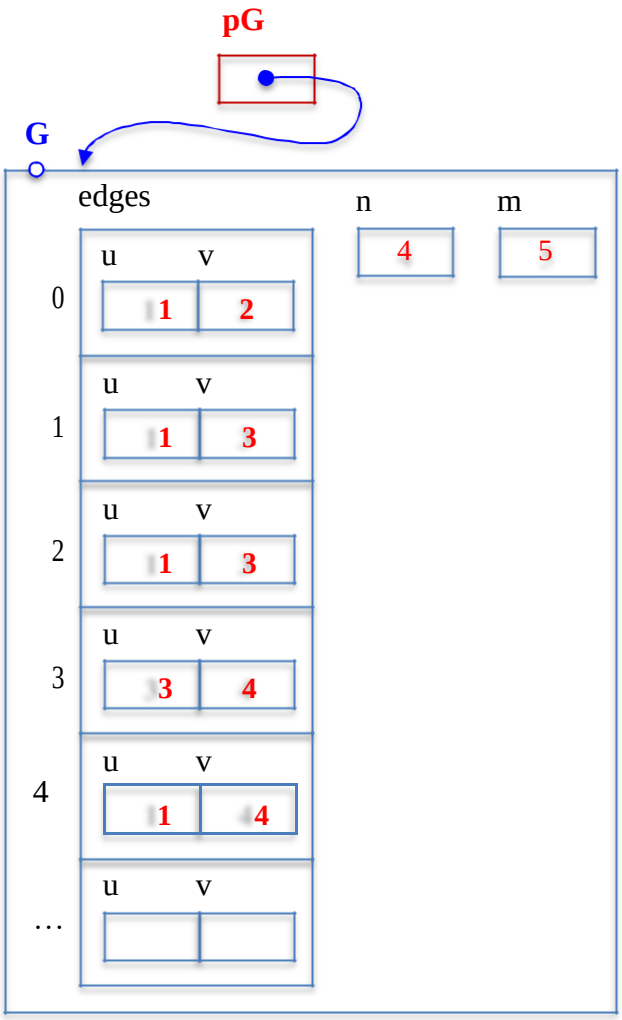


Chú ý: Chỉ số mảng đánh số từ 0.

Giả sử **G** là biến có kiểu Graph và **pG** là biến con trỏ Graph để lưu đồ thị trong ví dụ trên:

```
Graph G, *pG;
```

Nội dung của G và pG sẽ như hình bên dưới.



Ta có thể dùng G hoặc pG để truy xuất cấu trúc dữ liệu này. Hãy chú ý dấu chấm (.) đối với G và mũi tên (->) đối với pG.

```
G.n = 4;
pG->m = 5;
printf("So dinh cua do thi: %d\n", pG->n);
printf("So cung cua do thi: %d\n", G.m);

//In cung thứ 2 của đồ thị
printf("Cung e1: (%d, %d)\n", G.edges[2].u, G.edges[2].v);

//Thay đổi hai đầu mút của cung thứ 3 của đồ thị thành (2, 4)
pG->edges[3].u = 2;
pG->edges[3].v = 4;
```

1.2.2 Khởi tạo đồ thị

Quy trình thông thường để đưa dữ liệu của đồ thị trên máy tính gồm 2 bước:

- Khởi tạo đồ thị
- Lần lượt thêm từng cung vào đồ thị

Với phương pháp biểu diễn bằng danh sách cung, hàm khởi tạo đồ thị gồm 2 việc:

- Gán số đỉnh đồ thị = n.
- Khởi tạo số cung m = 0.

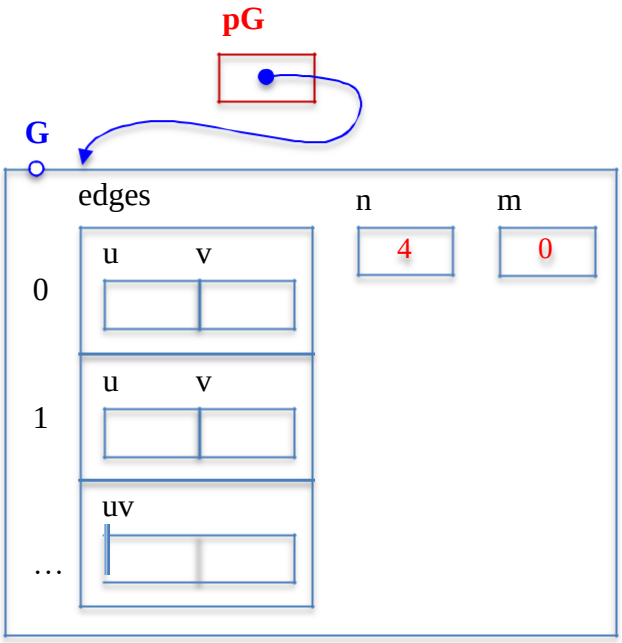
```
//Thêm cung u và vào đồ thị do pG trỏ đến
void init_graph(Graph *pG, int n) {
        pG->n = n;              //Gán số cung của (*pG) = n
        pG->m = 0;              //Gán số cung của (*pG) = 0
}
```

Trong khai báo hàm init_graph, tham số đầu tiên là pG có kiểu là **con trỏ Graph**. Đây là cách thường dùng để truyền dữ liệu kiểu cấu trúc cho hàm.

Ví dụ:

```
//Khai báo đồ thị G
Graph G;

Khởi tạo đồ thị G có 4 đỉnh
init_graph(&G, 4);
```

Để gọi hàm này, ta phải truyền địa chỉ của G (hay &G) cho pG. Sau lệnh này G có nội dung như sau:



1.2.3 Bài tập 1 – DSC: hàm init_graph()

Cho một chương trình cài đặt cấu trúc dữ liệu đồ thị bằng phương pháp "Danh sách cung". Hãy hoàn thành chương trình bằng cách viết thêm hàm init_graph(Graph *pG, int n) vào chỗ ba chấm (…) để khởi tạo đồ thị pG có n đỉnh và 0 cung.

```
//Khai báo thư viện
#include <stdio.h>

#define MAX_M 500          //M: số cung tối đa đồ thị có thể chứa

//Định nghĩa cấu trúc dữ liệu Edge biểu diễn 1 cung (u, v)
typedef struct {
        //Mỗi cung lưu đỉnh đầu u, đỉnh cuối v
        int u, v;
} Edge;

//Định nghĩa cấu trúc dữ liệu Graph biểu diễn 1 đồ thị
typedef struct {
        //n: đỉnh, m: cung
        int n, m;
        //Mảng edges lưu các cung của đồ thị
        Edge edges[MAX_M];
} Graph;

/* Viết mã lệnh của bạn ở đây */
//Định nghĩa hàm void init_graph(Graph *pG, int n)
...
/* Hết phần mã lệnh của bạn */
//Chương trình chính
int main() {
        Graph G;
        init_graph(&G, 5);
        printf("Do thi co %d dinh va %d cung.", G.n, G.m); return 0;

}
```

Quy trình làm bài tập:

- Sử dụng IDE bất kỳ (Dev-C++, Code::Blocks, Visual Studio Code, …) viết mã lệnh, biên dịch và chạy thử chương trình.
- Sau khi chạy thử thành công, bạn có thể nộp bài tập lên hệ thống ELSE để tự đánh giá bài làm của mình.

Chú ý: Khi mới bắt đầu, không nên làm bài trực tiếp trên hệ thống ELSE. Hãy viết chương trình hoàn chỉnh trên IDE và chạy thử. Sau khi kiểm tra kết quả chạy thử đúng với ý muốn của bạn, thì mới nên nộp bài lên hệ thống.

Nếu viết chương trình không có lỗi, khi chạy chương trình sẽ in ra:

```
Do thi co 5 dinh va 0 cung.
```

Nộp bài tập trên hệ thống ELSE:

- Đọc kỹ yêu cầu của đề bài nhất là phần **Chú ý**.

Cho một chương trình cài đặt cấu trúc dữ liệu đồ thị theo phương pháp "Danh sách cung" như bên dưới.

Hãy viết hàm void init_graph(Graph *pG, int n) vào chỗ trống để khởi tạo đồ thị (*pG) có số đỉnh bằng n và số cung bằng 0.

Chú ý

- Không nộp toàn bộ chương trình
- Chỉ nộp phần định nghĩa hàm init_graph()

Answer: (penalty regime: 10, 20, ... %)

```
#include <stdio.h>

#define MAX_M 500
typedef struct {
    int u, v;
} Edge;
typedef struct {
    int n, m;
    Edge edges[MAX_M];
} Graph;

/* Viết mã lệnh của bạn ở đây */
//Định nghĩa hàm void init_graph(Graph *pG, int n)
```

```
1
```

```
/* Hết phần mã lệnh của bạn */

//Chương trình chính
int main() {
    Graph G;
    init_graph(&G, 5);
    printf("Do thi co %d dinh va %d cung.", G.n, G.m);
    return 0;
}
```

- Sao chép (copy) và dán (dán) phần định nghĩa hàm init_graph() vào ô trống. Ấn nút "**Check**" để kiểm tra.

Answer: (penalty regime: 10, 20, ... %)

```
#include <stdio.h>

#define MAX_M 500
typedef struct {
    int u, v;
} Edge;
typedef struct {
    int n, m;
    Edge edges[MAX_M];
} Graph;

/* Viết mã lệnh của bạn ở đây */
//Định nghĩa hàm void init_graph(Graph *pG, int n)
```

```
1 void init_graph(Graph *pG, int n) {
2     pG->n = n;
3     pG->m = 0;
4 }
5
```

```
/* Hết phần mã lệnh của bạn */

//Chương trình chính
int main() {
    Graph G;
    init_graph(&G, 5);
    printf("Do thi co %d dinh va %d cung.", G.n, G.m);
    return 0;
}
```

Check

- Kết quả:

	Test	Expected	Got	
✔	init_graph(&G, 5);	Do thi co 5 dinh va 0 cung.	Do thi co 5 dinh va 0 cung.	✔
✔	init_graph(&G, 10);	Do thi co 10 dinh va 0 cung.	Do thi co 10 dinh va 0 cung.	✔
✔	init_graph(&G, 3);	Do thi co 3 dinh va 0 cung.	Do thi co 3 dinh va 0 cung.	✔
✔	init_graph(&G, 100);	Do thi co 100 dinh va 0 cung.	Do thi co 100 dinh va 0 cung.	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

1.2.4 Thêm cung vào đồ thị

Thêm một cung vào đồ thị gồm 2 bước:

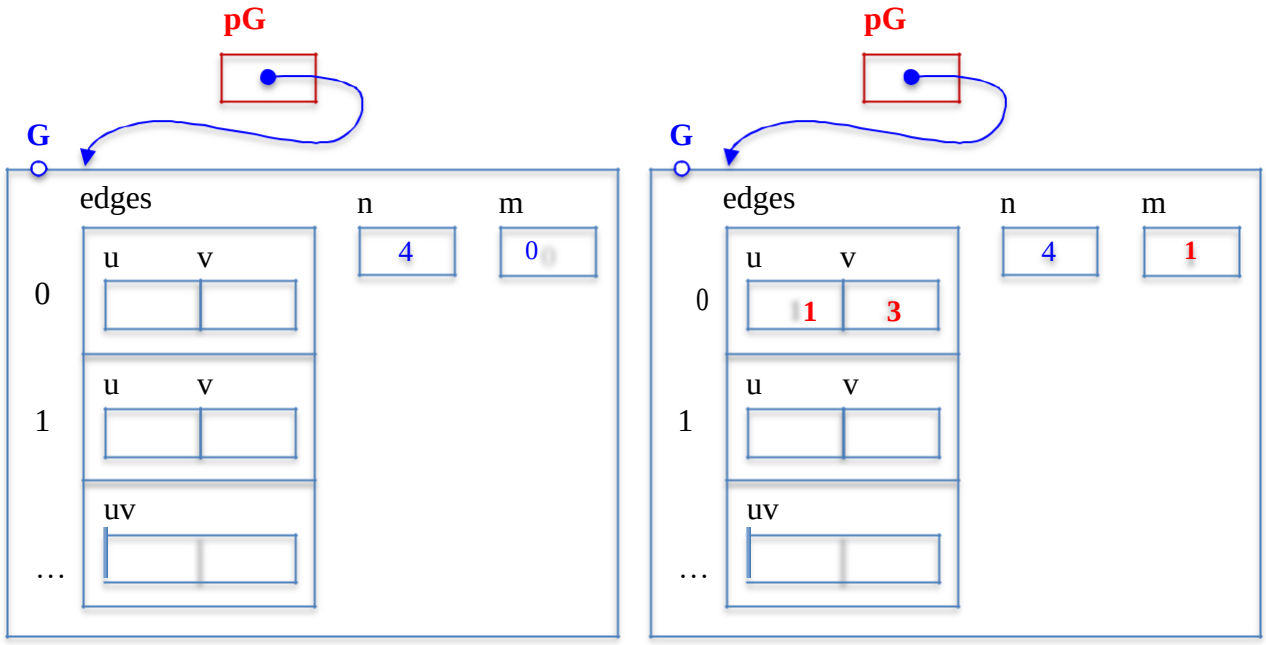
- Tạo một cung mới với 2 đầu mút (u, v) và thêm nó vào vào danh sách cung.
- Tăng số cung lên 1.

```
//Thêm cung u và vào đồ thị do pG trỏ đến
void add_edge(Graph *pG, int u, int v) {
        //Đưa cung (u, v) vào edges
        pG->edges[pG->m].u = u;
        pG->edges[pG->m].v = v;
        //Tăng số cung lên 1
        pG->m++;
}
```

Ví dụ: sau khi khởi tạo, ta thêm cung (1, 3) vào đồ thị.

```
Graph G;                    //Khai báo biến G dùng để chứa đồ thị
init_graph(&G, 4);          //Khởi tạo đồ thị
add_edge(&G, 1, 3);         //Thêm cung (1, 3)
```

Hình bên dưới cho thấy dữ liệu của G trước và sau khi thêm cung (1, 3).



(a) Trước khi thêm cung (1, 3)

(b) Sau khi thêm cung (1, 3)

1.2.5 Bài tập 2 – DSC: hàm add_edge() cơ bản

Cho cấu trúc dữ liệu đồ thị Graph được cài đặt bằng phương pháp "Danh sách cung" như sau:

```
//Cấu trúc Edge lưu dữ liệu của 1 cung
typedef struct		{
	int u, v;
} Edge;

//Khai báo cấu trúc dữ liệu Graph
typedef struct {
	int n, m;
	Edge edges[MAX_M];
} Graph;
```

Các cung được lưu trong danh sách edges với chỉ số từ 0, 1, 2, ..., m-1.

Hàm khởi tạo đồ thị:

```
//Khởi tạo đồ thị có n đỉnh và 0 cung
void init_graph(Graph *pG, int n) {
	pG->n = n;
	pG->m = 0;
}
```

Yêu cầu: viết hàm add_edge() để thêm cung (u, v) vào đồ thị G theo mẫu (prototype):

```
void add_edge(Graph *pG, int u, int v) {
}
```

Khác với bài tập 1, bài tập này không có sẵn khung chương trình hoàn chỉnh để chạy thử. Vì vậy, ta cần phải tự mình viết ra chương trình để kiểm tra hàm add_edge().

Khung chương trình dùng để kiểm tra một hàm bao gồm các phần chính sau:

```
//1. Khai báo thư viện, hằng
#include <stdio.h>

#define MAX_M 500

//2. Khai báo cấu trúc dữ liệu & các hàm cho sẵn
	typedef struct {
		int u, v;
} Edge;

//Định nghĩa hàm init_graph
void init_graph(Graph *pG, int n) {
}

//3. Định nghĩa hàm cần kiểm tra: add_edge()
/* Viết mã lệnh của bạn ở đây */
void add_edge(Graph *pG, int u, int v) {
}
/* Hết phần mã lệnh của bạn */

//4. Hàm main()
int main() {
	//Gọi hàm bạn đã định nghĩa
	add_edge(&G, 1, 2);
	return 0;
}
```

Bên dưới là một mẫu chương trình dùng để kiểm tra hàm add_edge().

```
//Khai báo hằng và thêm thư viện
#include <stdio.h>

#define MAX_M 500
typedef struct {
        int u, v;
} Edge;

typedef struct {
        int n, m;
        Edge edges[MAX_M];
} Graph;

//Định nghĩa hàm init_graph
void init_graph(Graph *pG, int n) {
        pG->n = n;
        pG->m = 0;
}

//Định nghĩa hàm add_edge
/* Viết mã lệnh của bạn ở đây */
void add_edge(Graph* pG, int u, int v) {

}
/ *Hết phần mã lệnh của bạn */

//Hàm main()
int main() {
        Graph G;
        init_graph(&G, 4);

        //Gọi hàm add_edge()
        add_edge(&G, 1, 2);
        add_edge(&G, 3, 4);

        //Kiểm tra hàm add_edge bằng cách in dữ liệu của đồ thị ra màn hình

        //1. In số đỉnh, số cung của đồ thị ra màn hình
        printf("n = %d, m = %d\n", G.n, G.m);

        //2. In các cung của đồ thị ra màn hình
        int e;
        for (e = 0; e < G.m; e++)
                printf("%d %d\n", G.edges[e].u, G.edges[e].v); return 0;

}
```

Mở IDE, lập trình và chạy thử. Nếu viết đúng, kết quả khi chạy chương trình sẽ là:

```
n = 4, m = 2
1  2
3  4
```

Nộp bài lên hệ thống ELSE:

- Đọc đề bài cẩn thận. Đề bài chỉ yêu cầu nộp phần định nghĩa hàm add_edge().

Cho cấu trúc dữ liệu đồ thị được cài đặt bằng phương pháp "Danh sách cung" như sau:

```
typedef struct  {
         int u, v;
} Edge;
typedef struct {
        int n, m;
        Edge edges[MAX_M];
} Graph;
```

Các cung được lưu trong danh sách **edges** với chỉ số từ 0, 1, 2, ..., m-1

Hàm khởi tạo đồ thị:

```
void init_graph(Graph *pG, int n){
        pG->n = n;
        pG->m = 0;
}
```

Viết hàm **add_edge(Graph *pG, int u, int v)** để thêm cung (u, v) vào đồ thị G theo mẫu:

```
void add_edge(Graph *pG, int u, int v) {
}
```

Chú ý

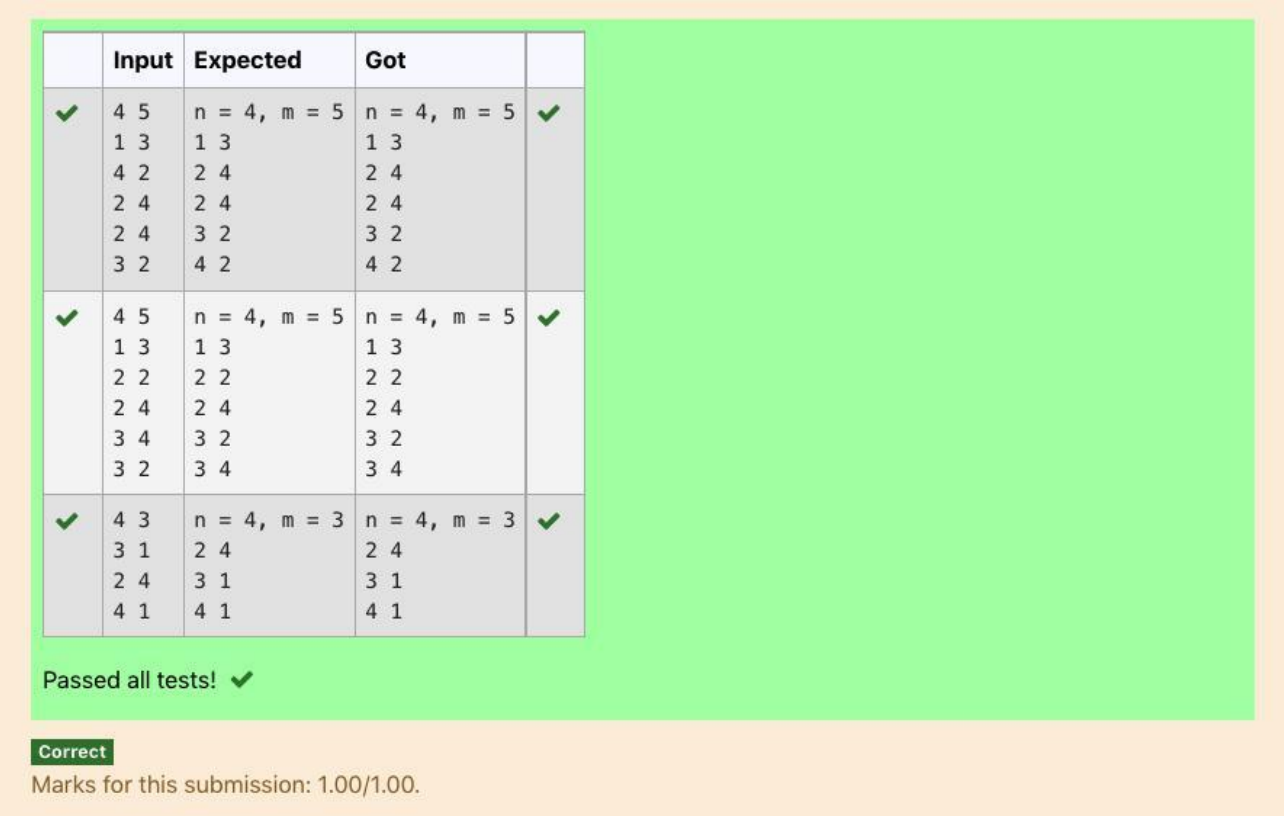
- Các tham số của hàm **add_edge()** luôn hợp lệ ($1 \leq u, v \leq n$), bạn không cần kiểm tra.
- Không nộp toàn bộ chương trình, chỉ nộp phần định nghĩa hàm **add_edge()**.

- Sao chép và dán hàm add_edge() vào ô trả lời:

Answer: (penalty regime: 10, 20, ... %)

```
1 void add_edge(Graph *pG, int u, int v) {
2     pG->edges[pG->m].u = u;
3     pG->edges[pG->m].v = v;
4
5     pG->m++;
6 }
7
```

- Kết quả:



	Input	Expected	Got	
✔	4 5 1 3 4 2 2 4 2 4 3 2	n = 4, m = 5 1 3 2 4 2 4 3 2 4 2	n = 4, m = 5 1 3 2 4 2 4 3 2 4 2	✔
✔	4 5 1 3 2 2 2 4 3 4 3 2	n = 4, m = 5 1 3 2 2 2 4 3 2 3 4	n = 4, m = 5 1 3 2 2 2 4 3 2 3 4	✔
✔	4 3 3 1 2 4 4 1	n = 4, m = 3 2 4 3 1 4 1	n = 4, m = 3 2 4 3 1 4 1	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

1.2.6 Bài tập 3a – DSC: hàm add_edge() nâng cao

Tương tự bài tập 2 với điều kiện bổ sung: nếu cung (u, v) không hợp lệ (ví dụ: u < 1 hoặc v > n, …) thì bỏ qua không làm gì cả.

Viết hàm **add_edge(Graph *pG, int u, int v)** để thêm cung (u, v) vào đồ thị G theo mẫu:

```
void add_edge(Graph *pG, int u, int v) {
}
```

Chú ý

- Nếu cung (u, v) không hợp lệ (ví dụ: u < 1 hoặc v > n, ...) thì bỏ qua không làm gì cả.
- Không nộp toàn bộ chương trình, chỉ nộp phần định nghĩa hàm **add_edge()**.

1.2.7 Bài tập 3b (*) – DSC: hàm add_edge() nâng cao

Tương tự bài tập 2 với điều kiện bổ sung: đồ thị pG là **đơn đồ thị có hướng**, nếu cung (u, v) đã có trong pG->edges rồi thì bỏ qua, không làm gì cả. Giả sử u, v luôn hợp lệ (1 £ u, v £ u), không cần phải kiểm tra.

Gợi ý: Trước khi thêm, kiểm tra xem cung (u, v) đã có trong đồ thị chưa.

1.2.8 Bài tập 3c (*) – DSC: hàm add_edge() nâng cao

Tương tự bài tập 2 với điều kiện bổ sung: đồ thị pG là **đơn đồ thị vô hướng**, nếu cung (u, v) hoặc hoặc cung (v, u) đã có trong pG->edges rồi thì bỏ qua, không làm gì cả. Giả sử u, v luôn hợp lệ (1 £ u, v £ u), không cần phải kiểm tra.

Gợi ý: Trước khi thêm, kiểm tra xem cung (u, v) và cung (v, u) đã có trong đồ thị chưa.

1.2.9 Kiểm tra u kề với v (v là đỉnh kề của u)

Nhắc lại: *Trong đồ thị vô hướng G, đỉnh u được gọi là kề với v nếu như G có cung (u, v) hoặc cung (v, u).*

Để kiểm tra đỉnh u có kề với đỉnh v không, ta:

- Lần lượt duyệt qua từng cung trong danh sách cung
  - Tìm xem có cung nào có dạng (u, v) hoặc (v, u) không, nếu có trả về 1
- Nếu không có cung nào có dạng (u, v) hoặc (v, u) trả về 0

```
//Kiểm tra đỉnh u có kề với đỉnh v trong đồ thị vô hướng
int adjacent(Graph *pG, int u, int v) {
        int e;
        //Duyệt qua từng cung 0, 1, 2, …, m - 1
        for (e = 0; e < pG->m; e++)
                if ((pG->edges[e].u == u && pG->edges[e].v == v) || (pG->edges[e].u
                        == v && pG->edges[e].v == u)) return 1;

        //Không có cung nào có dạng (u, v) hoặc (v, u)
        return 0;
}
```

Thuật toán này có một vòng lặp i chạy từ 0 đến m vì thế nó có độ phức tạp O(m).

Đối với đồ thị có hướng, đỉnh u được gọi là kề với v nếu có cung đi từ u đến v. Thuật toán kiểm tra u kề với v được cài đặt tương tự.

1.2.10 Bài tập 4a – DSC: hàm adjacent(), vô hướng

Cho cấu trúc dữ liệu đồ thị Graph được cài đặt bằng phương pháp "Danh sách cung" dùng để biểu diễn các **đồ thị vô hướng**.

```
#define MAX_M 500

//Cấu trúc Edge lưu dữ liệu của 1 cung
typedef struct          {
        int u, v;
} Edge;

//Khai báo cấu trúc dữ liệu Graph
typedef struct {
        int n, m;
        Edge edges[MAX_M];
} Graph;
```

Các cung được lưu trong danh sách edges với chỉ số từ 0, 1, 2, ..., m-1.

Yêu cầu: viết hàm adjacent() để kiểm tra đỉnh u có kề với đỉnh v không, theo mẫu (prototype):

```
int adjacent(Graph *pG, int u, int v) {
}
```

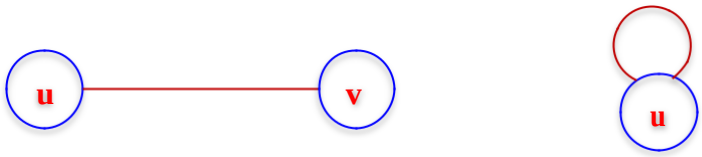
Trả về 1, nếu đỉnh u kề với v, ngược lại trả về 0.

1.2.11 Bài tập 4b – DSC: hàm adjacent(), có hướng

Tương tự bài tập 4a nhưng pG là **đồ thị có hướng**.

1.2.12 Tính bậc của đỉnh

Nhắc lại: *bậc của đỉnh u, ký hiệu deg(u), là số cung liên thuộc với đỉnh u, khuyên được tính 2 lần.*



Dễ dàng nhìn thấy rằng:

- Cung (u, v) góp:
  - 1 bậc cho deg(u) và
  - 1 bậc cho deg(v)
- Khuyên (u, u) góp:
  - 2 bậc cho deg(u). Điều này cũng tương đương với:
    - góp 1 bậc cho deg(u), rồi góp 1 bậc cho deg(u) thêm 1 lần nữa.

Từ nhận xét trên ta có thuật toán tính bậc cho đỉnh u như sau:

- Gán deg_u = 0
- Duyệt qua từng cung trong danh sách cung
  - Nếu cung đang xét có dạng (u, -) tăng deg_u thêm 1
  - Nếu cung đang xét có dạng (-, u) tăng deg_u thêm 1

Đối với đồ thị có hướng,

- Bậc vào của đỉnh u, deg_in(u) = số cung có dạng (-, u)
- Bậc ra của đỉnh u, deg_out(u) = số cung có dạng (u, -)

Thuật toán tính bậc của đỉnh u trong đồ thị bất kỳ (có hướng/vô hướng)

```
//Đếm bậc của đỉnh u của đồ thị bất kỳ
int degree(Graph *pG, int u) {
	int e, deg_u = 0;
	//Duyệt qua từng cung 0, 1, 2, …, m - 1
	for (e = 0; e < pG->m; e++) {
		//Nếu cung có dạng (u, -)
		if (pG->edges[e].u == u)
			deg_u++;
		//Nếu cung có dạng (-, u)
		if (pG->edges[e].v == u)
			deg_u++;
	}
	return deg_u;
}
```

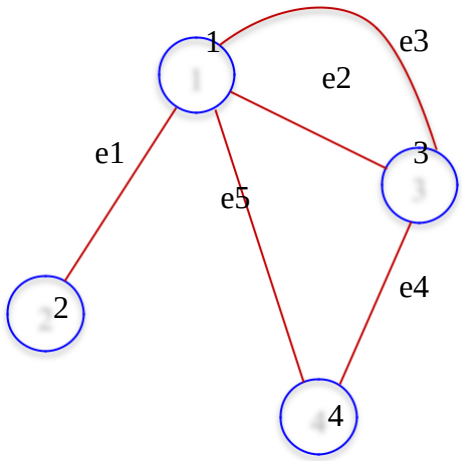
1.2.13 Bài tập 5a – DSC: tổng hợp

Biểu diễn đồ thị và in ra bậc của các đỉnh ra màn hình.

Các bước thực hiện:

Khai báo cấu trúc dữ liệu đồ thị: Graph

- Cài đặt hàm khởi tạo: init_graph()
- Cài đặt hàm thêm cung: add_edge()
- Cài đặt hàm tính bậc: degree()
- Viết hàm main(), trong đó:
  - Khai báo một biến đồ thị G
  - Gọi hàm khởi tạo đồ thị G với số đỉnh n = 4, số cung m = 5
  - Gọi hàm add_edge() 5 lần để thêm 5 cung vào đồ thị
  - Cho một vòng lặp với biến u chạy từ đỉnh 1 đến đỉnh n, gọi hàm degree(u) để tính bậc của u



Yêu cầu: Cho chương trình có hàm main() như bên dưới. Hãy viết thêm khai báo CTDL Graph (biểu diễn đồ thị bằng phương pháp danh sách cung) và cài đặt các hàm cần thiết vào chỗ ba chấm (…) để có được chương trình hoàn chỉnh, chạy được.

```
//Khai báo thư viện xuất nhập
#include <stdio.h>

/* Bổ sung khai báo CTDL Graph và cài đặt các hàm cần thiết */
...
/* Hết phần mã lệnh của bạn */

//Hàm main()
int main() {
	Graph G;
	int n = 4, u;

	//Khởi tạo đồ thị
	init_graph(&G, n);

	//Thêm cung vào đồ thị
	add_edge(&G, 1, 2);
	add_edge(&G, 1, 3);
	add_edge(&G, 1, 3);
	add_edge(&G, 3, 4);
	add_edge(&G, 1, 4);

	//In bậc của các đỉnh
	for (u = 1; u <= n; u++)
		printf("deg(%d) = %d\n", u, degree(&G, u)); return 0;

}
```

Mở IDE lên và cài đặt. Nếu bạn cài đặt đúng, kết quả sẽ là:

```
deg(1) = 4
deg(2) = 1
deg(3) = 3
deg(4) = 2
```

Nộp bài lên hệ thống ELSE:

- Đọc kỹ đề bài, nhất là phần **Chú ý** (nếu có).

Cho chương trình có hàm **main()** như bên dưới. Hãy viết thêm khai báo CTDL **Graph** (biểu diễn đồ thị bằng phương pháp **danh sách cung**) và cài đặt các hàm cần thiết vào chỗ ba chấm (...) để có được chương trình hoàn chỉnh, chạy được.

```
#include <stdio.h>

/* Bổ sung khai báo CTDL Graph và cài đặt các hàm cần thiết */
...
/* Hết phần mã lệnh của bạn */

//Hàm main()
int main() {
    Graph G;
    int n = 4, u;

    //Khởi tạo đồ thị
    init_graph(&G, n);
    //Thêm cung vào đồ thị
    add_edge(&G, 1, 2);
    add_edge(&G, 1, 3);
    add_edge(&G, 1, 3);
    add_edge(&G, 3, 4);
    add_edge(&G, 1, 4);

    //In bậc của các đỉnh
    for (u = 1; u <= n; u++)
        printf("deg(%d) = %d\n", u, degree(&G, u));
    return 0;
}
```

Chú ý

- Không nộp toàn bộ chương trình, chỉ nộp phần bạn viết.

- Copy và paste bài làm vào ô "Answer":

Answer: (penalty regime: 10, 20, ... %)

```
1  #define MAX_M 500
2
3  typedef struct {
4      
```

- Ấn vào nút "Check" để kiểm tra.

Check

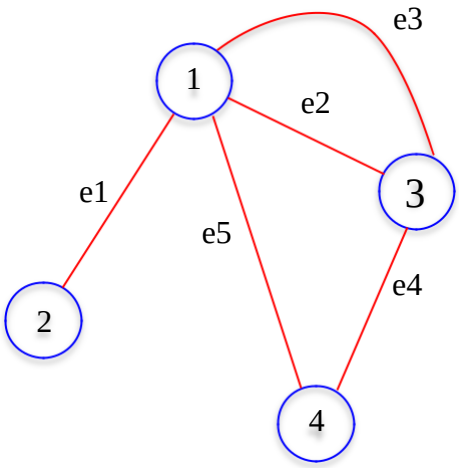
1.2.14 Nhập dữ liệu cho đồ thị từ bàn phím

Trong các bài tập trên, ta thấy rằng để biểu diễn đồ thị ta phải sử dụng các lệnh add_edge() trực tiếp trong chương trình và mỗi lần thay đổi đồ thị ta phải **sửa lại các lệnh này** hoặc **viết lại chương trình khác** và phải biên dịch lại chương trình.

Để tránh vấn đề này, ta sẽ cho phép người dùng nhập dữ liệu cho đồ thị từ bàn phím hoặc tập tin. Ta sẽ mô phỏng quá trình đọc dữ liệu (từ bàn phím hay tập tin) cũng tương tự như việc nhập dữ liệu trực tiếp bằng cách gọi hàm add_edge(). Quá trình đọc dữ liệu gồm các bước sau:

- Đọc số đỉnh n và số cung m.
- Gọi hàm init_graph() để khởi tạo đồ thị.
- Lặp m lần, mỗi lần đọc 1 cung
  - Đọc 2 đỉnh u, v.
  - Gọi hàm add_edge() để thêm cung (u, v) vào đồ thị.

```
//Hàm main()
int main() {
        Graph G;
        int n, m, e, u, v;

        //Đọc số đỉnh và số cung & khởi tạo đồ thị
        scanf("%d%d", &n, &m);
        init_graph(&G, n);

        //Đọc m cung và thêm vào đồ thị
        for (e = 0; e < m; e++) {
                scanf("%d%d", &u, &v);
                add_edge(&G, u, v);
        }
        ...
}
```



Với đồ thị này, khi chạy chương trình ta nhập dữ liệu như sau:

```
4  5<ENTER>
1  2<ENTER>
1  3<ENTER>
1  3<ENTER>
3  4<ENTER>
1  4<ENTER>
```

1.2.15 Bài tập 5b – DSC: tổng hợp, đọc dữ liệu từ bàn phím

Viết lại toàn bộ chương trình của bài tập 4, cho phép người dùng nhập dữ liệu cho đồ thị từ bàn phím. In bậc của các đỉnh của đồ thị theo mẫu:

```
deg(1) = 4
deg(2) = 1
deg(3) = 3
deg(4) = 2
```

Mở IDE lên, lập trình và chạy thử.

Nộp bài lên hệ thống ELSE: đọc kỹ đề bài, nhất là các phần **Đầu vào**, **Đầu ra**, **Chú ý**, **For example**.

Viết chương trình bằng ngôn ngữ C cho phép người dùng nhập dữ liệu của một đồ thị và in bậc của các đỉnh ra màn hình.

Đầu vào

Dữ liệu đầu vào được đọc từ dòng nhập chuẩn (stdin, bàn phím) theo định dạng:

- Dòng đầu tiên chứa 2 số nguyên n và m cách nhau một khoảng trắng, n: số đỉnh, m: số cung
- m dòng tiếp theo, mỗi dòng chứa 2 số nguyên u v cách nhau 1 khoảng trắng mô tả cung (u, v).

Đầu ra

- In ra n dòng, dòng thứ i in bậc của đỉnh i, theo mẫu: deg(2) = 3

Chú ý

- Giả sử dữ liệu đầu vào luôn hợp lệ, không cần phải kiểm tra
- Nộp toàn bộ chương trình
- Xem thêm định dạng đầu vào và đầu ra trong phần **For example**
- Ấn "Precheck" (nếu có) để kiểm tra chương trình trên các ví dụ (sai KHÔNG bị trừ điểm)
- Ấn "Check" (nếu có) để kiểm tra chương trình trên toàn bộ dữ liệu kiểm tra (sai bị TRỪ ĐIỂM)

For example:

Input	Result
4 5 1 3 4 2 2 4 2 4 3 2	deg(1) = 1 deg(2) = 4 deg(3) = 2 deg(4) = 3

Bài tập này yêu cầu nộp toàn bộ chương trình. Hãy copy toàn bộ chương trình và dán vào ô **Answer**. Sau đó ấn "**Precheck**" hoặc "**Check**".

Answer: (penalty regime: 10, 20, ... %)

```c
//Hàm main()
int main() {
    Graph G;
    int n, m, e, u, v;

    //Đọc số đỉnh và số cung & khởi tạo đồ thị
    scanf("%d%d", &n, &m);
    init_graph(&G, n);

    //Đọc m cung và thêm vào đồ thị
    for (e = 0; e < m; e++) {
        scanf("%d%d", &u, &v);
        add_edge(&G, u, v);
    }

    for (int u = 1; u <= n; u++)
        printf("deg(%d) = %d\n", u, degree(&G, u));


    return 0;
}
```

Precheck Check

Ấn “**Precheck**”: chỉ kiểm tra trên các ví dụ trong phần For example. Sai KHÔNG bị trừ điểm.

Precheck only

	Input	Expected	Got	
✔	4 5 1 3 4 2 2 4 2 4 3 2	deg(1) = 1 deg(2) = 4 deg(3) = 2 deg(4) = 3	deg(1) = 1 deg(2) = 4 deg(3) = 2 deg(4) = 3	✔

Ấn “Check”: kiểm tra trên toàn bộ dữ liệu. Sai sẽ bị TRỪ ĐIỂM. Hãy cẩn thận khi làm bài thi hoặc kiểm tra.

	Input	Expected	Got	
✔	4 5 1 3 4 2 2 4 2 4 3 2	deg(1) = 1 deg(2) = 4 deg(3) = 2 deg(4) = 3	deg(1) = 1 deg(2) = 4 deg(3) = 2 deg(4) = 3	✔
✔	4 5 1 1 2 2 2 4 3 4 3 2	deg(1) = 2 deg(2) = 4 deg(3) = 2 deg(4) = 2	deg(1) = 2 deg(2) = 4 deg(3) = 2 deg(4) = 2	✔
✔	4 3 3 1 2 4 4 2	deg(1) = 1 deg(2) = 2 deg(3) = 1 deg(4) = 2	deg(1) = 1 deg(2) = 2 deg(3) = 1 deg(4) = 2	✔
✔	6 8 3 1 2 4 4 2 2 3 4 6 5 5 5 5 6 4	deg(1) = 1 deg(2) = 3 deg(3) = 2 deg(4) = 4 deg(5) = 4 deg(6) = 2	deg(1) = 1 deg(2) = 3 deg(3) = 2 deg(4) = 4 deg(5) = 4 deg(6) = 2	✔

Passed all tests! ✔

Correct

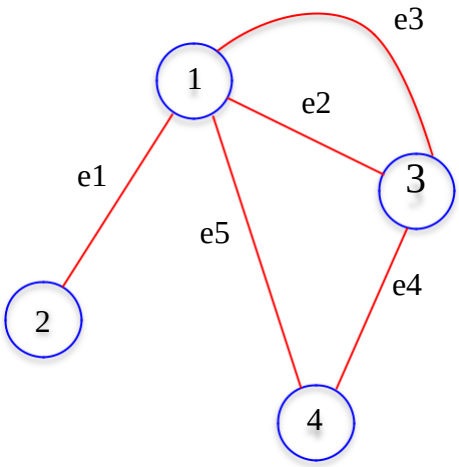
Marks for this submission: 1.00/1.00.

1.2.16 Nhập dữ liệu cho đồ thị từ tập tin

Nhập dữ liệu cho đồ thị từ bàn phím có ưu điểm là không cần phải sửa lại chương trình, không cần biên dịch lại. Tuy nhiên, mỗi lần chạy chương trình lại phải nhập dữ liệu lại. Nếu đồ thị có nhiều cung, việc nhập lại này cũng mất không ít thời gian. Hơn nữa nếu trong quá trình người dùng nhập liệu có sai sót mà đã bấm phím ENTER thì không thể quay lên để sửa.

Để giải quyết các vấn đề trên, ta sẽ mô tả đồ thị trong một *tập tin văn bản* và chương trình của chúng ta sẽ đọc tập tin này để xây dựng đồ thị.

Ta lấy lại ví dụ đồ thị trong phần trên:



Với đồ thị này, ta có thể mô tả thông tin đỉnh và cung trong một tập tin văn bản như sau (lưu nội dung này vào một tập tin ví dụ: **dothi.txt**):

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

- Dòng đầu tiên của tập tin (4 5) nói rằng đồ thị ta có 4 đỉnh và 5 cung.
- 5 dòng tiếp theo, mỗi dòng mô tả một cung, ví dụ: (1 2) nói rằng cung 1 có hai đầu mút là đỉnh 1 và đỉnh 2.

Để đọc tập tin này và tạo đồ thị ta có thể sử dụng mẫu chương trình bên dưới. Giống như đọc dữ liệu từ bàn phím, ta không cần gán trực tiếp n = 4, m = 5. Các giá trị này được đọc từ nội dung tập tin "dothi.txt". Ta cũng không sử dụng lệnh add_edge() với các đỉnh tường minh trong chương trình. Các đầu mút của cung e (đỉnh u và đỉnh v) cũng được đọc từ file.

Chú ý: Tập tin **dothi.txt** đặt cùng thư mục với file chương trình. Nếu file này đặt nơi khác cần chỉ rõ đường dẫn đi đến nó, ví dụ: "D:\\LTDT\\dothi.txt"

```
//Hàm main()
int main() {
	Graph G;
	int n, m, e, u, v;

	//Mở file dothi.txt để đọc dữ liệu
	FILE *file = fopen("dothi.txt", "r");


	//Đọc số đỉnh và số cung & khởi tạo đồ thị
	fscanf(file, "%d%d", &n, &m);
	init_graph(&G, n, m);

	//Đọc m cung
	for (e = 0; e < m; e++) {
		fscanf(file, "%d%d", &u, &v);
		add_edge(&G, u, v);
	}

	//Đóng file
	fclose(file);
	...
}
```

Ngoài cách sử dụng các lệnh fopen(), fscanf() và fclose() để đọc dữ liệu từ tập tin, ta còn một cách khác tiện hơn rất nhiều: đọc dữ liệu từ tập tin bằng hàm scanf() chứ không phải fscanf(). Phương pháp này rất hữu dụng khi bạn làm bài trên hệ thống hỗ trợ thực hành ELSE.

Trong ngôn ngữ C, hàm scanf() sẽ đọc dữ liệu từ dòng nhập chuẩn (stdin: mặc định là bàn phím). Nếu ta có cách chuyển dữ liệu từ tập tin vào stdin thì ta có thể sử dụng scanf() để đọc dữ liệu từ tập tin. May thay, ngôn ngữ C hỗ trợ chúng ta lệnh freopen() cho phép chuyển dữ liệu từ tập tin sang stdin (bàn phím), ví dụ:

```
//Chuyển dữ liệu file.txt sang dòng nhập chuẩn (stdin)
freopen("file.txt","r", stdin);


//Sau đó, ta có thể đọc dữ liệu từ file bằng lệnh scanf
scanf("%d", &n);
```

Để phục hồi lại việc đọc từ bàn phím sau khi dùng freopen(), ta sử dụng lệnh:

```
//Phục hồi lại việc đọc dữ liệu từ bàn phím
stdin = fdopen(1, "r");
```

Lệnh này sẽ trả bàn phím về cho stdin và do đó lệnh scanf() sẽ tiếp tục đọc dữ liệu từ bàn phím.

Ví dụ sau đây đọc đồ thị từ tập tin "dothi.txt" dùng freopen() và scanf().

```
//Hàm main()
int main() {
	Graph G;
	int n, m, e, u, v;

	//Chuyển dữ liệu từ file sang dòng nhập chuẩn
	freopen("dothi.txt", "r", stdin);


	//Đọc số đỉnh và số cung & khởi tạo đồ thị
	scanf("%d%d", &n, &m);
	init_graph(&G, n, m);

	//Đọc m cung
	for (e = 0; e < m; e++) {
		scanf("%d%d", &u, &v);
		add_edge(&G, u, v);
	}

	...
}
```

Mẹo: Các bài tập có đọc dữ liệu trên hệ thống ELSE thường yêu cầu đọc dữ liệu bàn phím (stdin). Khi lập trình và chạy thử chương trình trên IDE thì ta lại muốn nhập dữ liệu từ tập tin cho tiện. Để giải quyết mâu thuẫn này, khi lập trình trên IDE hãy sử dụng freopen(), và khi nộp bài lên hệ thống, hãy xoá dòng freopen() này đi hoặc biến nó thành chú thích như bên dưới.

```
//Hàm main()
int main() {
	Graph G;
	int n, m, e, u, v;

	//Chuyển dữ liệu từ file sang dòng nhập chuẩn
	//Khi nộp bài trên hệ thống ELSE, chú thích/xoá dòng bên dưới
	//freopen("dothi.txt", "r", stdin);


	//Đọc số đỉnh và số cung & khởi tạo đồ thị
	scanf("%d%d", &n, &m);
	init_graph(&G, n, m);

	//Đọc m cung
	for (e = 0; e < m; e++) {
		scanf("%d%d", &u, &v);
		add_edge(&g, u, v);
	}

	...
}
```

1.2.17 Bài tập 5c – DSC: tổng hợp, đọc dữ liệu từ tập tin

Làm lại bài tập 5b bằng cách sử dụng fopen() và fscanf() hoặc freopen() và scanf() để đọc dữ liệu từ tập tin. So sánh giải pháp này với giải pháp đọc trực tiếp dữ liệu từ bàn phím.

1.2.18 Liệt kê các đỉnh kề của một đỉnh

Liệt kê các đỉnh kề của một đỉnh là một phép toán thường được dùng trong nhiều thuật toán trên đồ thị như: duyệt đồ thị, tìm đường đi ngắn nhất, xếp hạng đồ thị, …

Thuật toán tổng quát dùng liệt kê các đỉnh kề của đỉnh u là lần lượt xét các đỉnh v từ 1 đến n, nếu u kề với v thì v là đỉnh kề của u.

```
//Liệt kê các đỉnh kề của đỉnh u bất kỳ
void neighbours(Graph* pG, int u) {
        int v;
        for (v = 1; v <= pG->n; v++)
                if (adjacent(pG, u, v) != 0)              //nếu u kề với v
                        printf("%d ", v);  //Liệt kê v printf("\n");

}
```

Chú ý:

- Thuật toán tổng quát chỉ phụ thuộc vào hàm adjacent() nên có thể dùng cho mọi phương pháp biểu diễn đồ thị.
- Thuật toán này liệt kê mỗi đỉnh nhiều nhất 1 lần. Trường hợp đồ thị có chứa đa cung (và đa khuyên), cần phải tìm một thuật toán khác để liệt kê đầy đủ các đỉnh kề được lặp lại.
- Với phương pháp biểu diễn bằng danh sách cung, do hàm adjacent() có độ phức tạp O(m) nên độ phức tạp của thuật toán này là O(n.m).

Một cách khác để tìm các đỉnh kề của đỉnh u là duyệt qua tất cả các cung, nếu cung đang xét có dạng (u, v) thì v là đỉnh kề của u. Đối với đồ thị vô hướng, ta xét thêm cung có dạng (v, u).

```
//Liệt kê các đỉnh kề của đỉnh u trong đồ thị có hướng
void neighbours(Graph* pG, int u) {
        int e;
        for (e = 0; e < pG->m; e++)
                if (pG->edges[e].u == u) //Nếu cung e có dạng (u, v) printf("%d ", pG-
                        >edges[e].v); //Liệt kê v
        printf("\n");
}
```

Thuật toán này hiệu quả hơn thuật toán cơ bản và có khả năng liệt kê được các đỉnh kề lặp lại. Độ phức tạp thuật toán là O(m). Tuy nhiên, nhược điểm của nó là các đỉnh kề không được liệt kê theo theo thứ tự tăng dần.

1.2.19 Bài tập 6a – DSC: liệt kê đỉnh kề của đồ thị vô hướng

Viết chương trình đọc vào một **đồ thị vô hướng** và in ra các đỉnh kề của các đỉnh.

Viết chương trình bằng ngôn ngữ C cho phép người dùng nhập dữ liệu của một **đồ thị vô hướng** và in các đỉnh kề của các đỉnh ra màn hình.

Đầu vào

Dữ liệu đầu vào được đọc từ dòng nhập chuẩn (stdin, bàn phím) theo định dạng:

- Dòng đầu tiên chứa 2 số nguyên n và m cách nhau một khoảng trắng, n: số đỉnh, m: số cung
- m dòng tiếp theo, mỗi dòng chứa 2 số nguyên u v cách nhau 1 khoảng trắng mô tả cung (u, v).

Đầu ra

- In ra n dòng, dòng thứ i in các đỉnh kề của đỉnh i, cách nhau 1 khoảng trắng, theo thứ tự tăng dần (không lặp lại), ví dụ: **neighbours(2) = 1 2 4**

Chú ý

- Giả sử dữ liệu đầu vào luôn hợp lệ, không cần phải kiểm tra
- Nộp toàn bộ chương trình
- Xem thêm định dạng đầu vào và đầu ra trong phần **For example**
- Ấn "Precheck" (nếu có) để kiểm tra chương trình trên các ví dụ (sai KHÔNG bị trừ điểm)
- Ấn "Check" (nếu có) để kiểm tra chương trình trên toàn bộ dữ liệu kiểm tra (sai bị TRỪ ĐIỂM)

For example:

Input	Result
4 3 1 3 4 1 3 3	neighbours(1) = 3 4 neighbours(2) = neighbours(3) = 1 3 neighbours(4) = 1

1.2.20 Bài tập 6b – DSC: liệt kê đỉnh kề của đồ thị có hướng

Tương tự bài 6a nhưng cho **đồ thị có hướng**.

1.3 Phương pháp ma trận đỉnh – đỉnh (ma trận kề)

Đây là một trong hai phương pháp thường dùng để biểu diễn đồ thị (vô hướng và có hướng). Tương tự như phương pháp danh sách cung, ta giả sử các đỉnh đã được đánh số từ 1 đến n.

Ta không lưu trữ trực tiếp các cung mà thay vào đó ta sẽ lưu trữ **sự kề nhau của hai đỉnh**. Vì thế phương pháp này còn có tên là **ma trận kề**. Ma trận kề mô tả mối quan hệ kề nhau giữa hai đỉnh.

Dùng 1 ma trận vuông n hàng, n cột: A = {a_{uv}} với u = 1, 2, …, n, v = 1, 2, …, n

Với từng loại đồ thị, có cách biểu diễn khác nhau

Đối với đơn đồ thị (vô hướng/có hướng): phần tử hàng u, cột v có giá trị

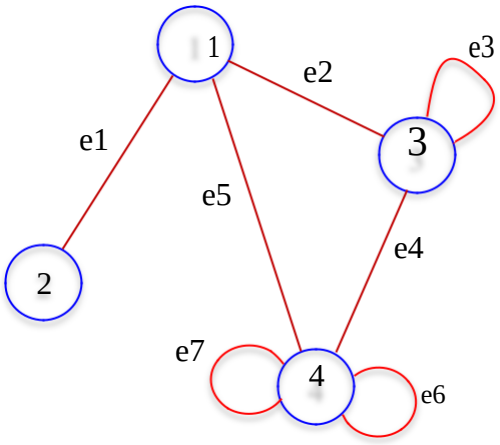
- a_{uv} = 1 nếu đỉnh u kề với đỉnh v.
- a_{uv} = 0 ngược lại.

Đối với đa đồ thị (vô hướng/có hướng): phần tử hàng u, cột v có giá trị

- a_{uv} = số cung đi từ u đến v.
- a_{uv} = 0 ngược lại.

Trường hợp đồ thị có chứa khuyên, phần tử a_{uu} tương ứng với đỉnh u sẽ có giá trị bằng số khuyên tại đỉnh u.

Ví dụ:
Ma trận kề đồ thị vô hướng.

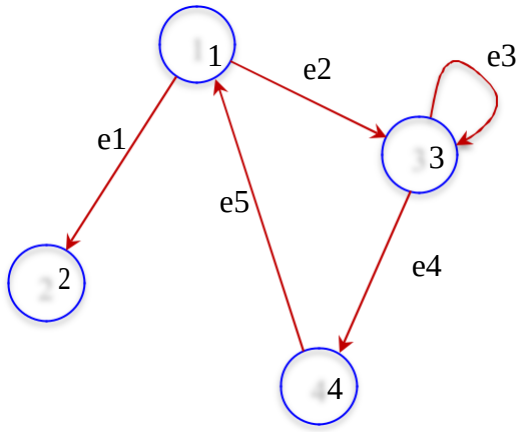


Ma trận kề:

	1	**2**	**3**	**4**
1	0	1	1	1
2	1	0	0	0
3	1	0	**1**	1
4	1	0	1	**2**

Nhận xét: Ma trận kề của đồ thị vô hướng là ma trận đối xứng.

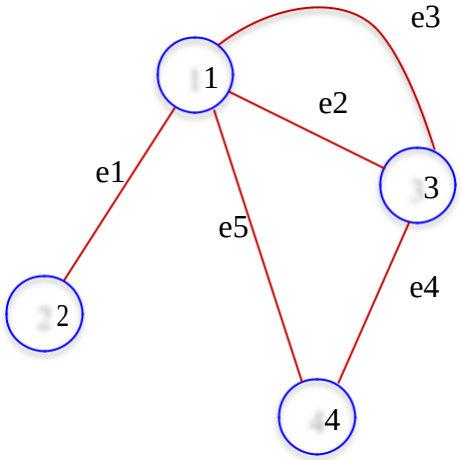
Ma trận kề của đồ thị có hướng:



Ma trận kề:

	1	**2**	**3**	**4**
1	0	1	1	0
2	0	0	0	0
3	0	0	1	1
4	1	0	0	0

Đồ thị có đa cung:



	1	**2**	**3**	**4**
1	0	1	**2**	1
2	1	0	0	0
3	**2**	0	0	1
4	1	0	1	0

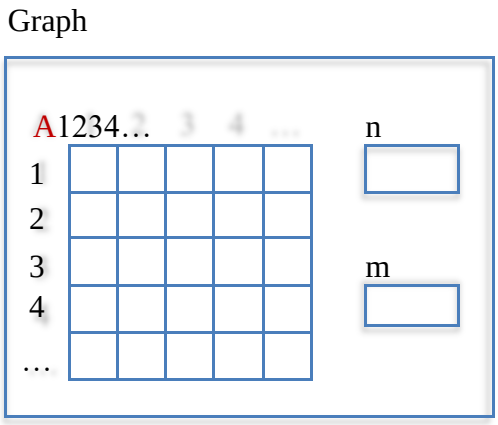
1.3.1 Cài đặt

Sử dụng một cấu trúc gồm các trường sau:

- A[][]: mảng hai chiều lưu ma trận kề (đỉnh – đỉnh)
- n: số đỉnh
- Có thể lưu thêm số cung m nếu muốn

```
//Định nghĩa hằng MAX_N: số đỉnh tối đa đồ thị có thể chứa
#define MAX_N 100

//Định nghĩa cấu trúc dữ liệu Graph biểu diễn 1 đồ thị
typedef struct {
        //n: đỉnh, m: cung
        int n, m;

        //Mảng A lưu ma trận kề
        int A[MAX_N][MAX_N];
} Graph;
```

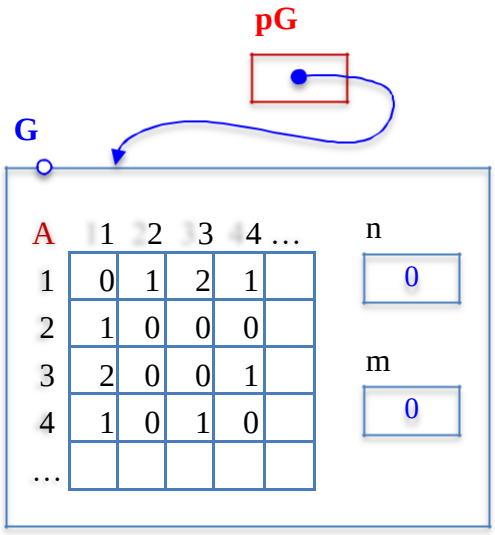
Sơ đồ tổ chức dữ liệu của cấu trúc dữ liệu Graph:



Giả sử **G** là biến có kiểu Graph và **pG** là biến con trỏ Graph để lưu đồ thị trong ví dụ trên:

```
Graph G, *pG;
```

Nội dung của G và pG sẽ như hình bên dưới.



1.3.2 Khởi tạo đồ thị

- Gán số đỉnh cho n, (nếu có lưu số lượng cung, gán số cung bằng 0)
- Khởi tạo ma trận A chứa toàn số 0

```
//Khởi tạo đồ thị có n đỉnh và 0 cung
void init_graph(Graph *pG, int n) {
        //n: đỉnh, 0: cung
        pG->n = n;
        pG->m = 0;
        //Khởi tạo ma trận A chứa toàn số 0
        for (int u = 1; u <= n; u++)
                for (int v = 1; v <= n; v++)
                        pG->A[u][v] = 0;
}
```

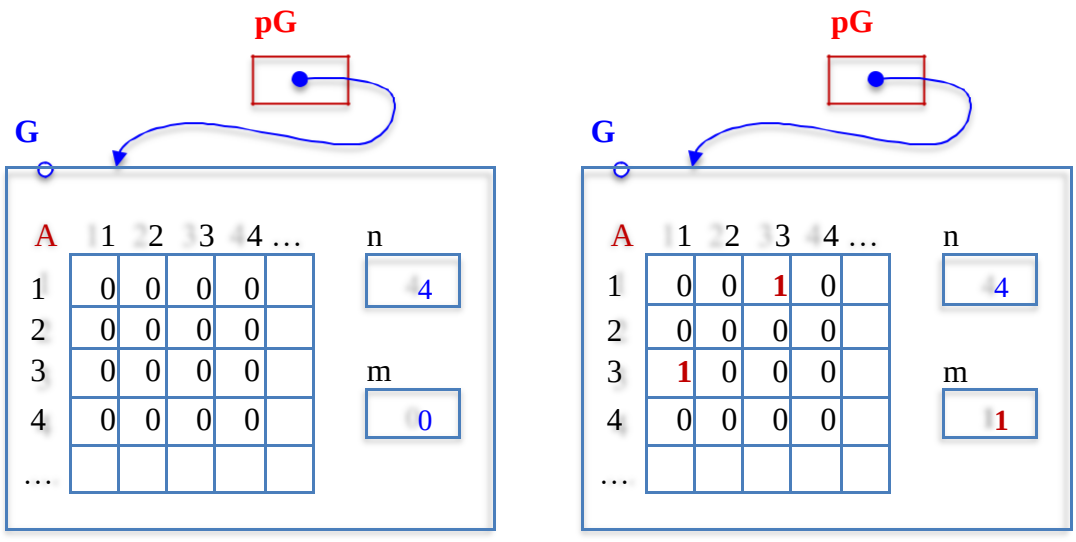
1.3.3 Thêm cung vào đồ thị

Đối với đơn đồ thị vô hướng:

- Cho đỉnh u kề với đỉnh v: A[u][v] = 1
- Cho đỉnh v kề với đỉnh u: A[v][u] = 1

```
//Khởi tạo đồ thị có n đỉnh và 0 cung
void add_edge(Graph *pG, int u, int v) { pG->A[u][v] =
        1;  //cho u kề với v pG->A[v][u] = 1;
        //cho v kề với u

        //Tăng số cung lên 1
        pG->m++;
}
```

Hình bên dưới minh hoạ nội dung của cấu trúc dữ liệu Graph G (hay *pG) sau khi thêm cung (1, 3).



(a) Trước khi thêm cung (1, 3)

(b) Sau khi thêm cung (1, 3)

Đối với trường hợp khác: đơn đồ thị có hướng, đa đồ thị, đồ thị có khuyên cần phải xem lại cách biểu diễn bên trên để có cách cài đặt hàm này thích hợp, cụ thể:

- **Nếu đồ thị chứa các đa cung**, ta phải cộng dồn số cung vào ô pG->A[u][v] (pG->A[u][v] += 1) chứ không phải gán = 1.
- **Nếu G là đồ thị có hướng**, ta không tăng giá trị của pG->A[v][u]. Ma trận kề sẽ không đối xứng.
- **Nếu G là đồ thị vô hướng có chứa khuyên (giả đồ thị),** chỉ tăng G->[u][u] lên 1 chứ không phải 2.

1.3.4 Bài tập 7a – MTK: hàm init_graph() & add_edge(), đơn đồ thị vô hướng

Cho cấu trúc dữ liệu đồ thị Graph được cài đặt bằng phương pháp "Ma trận kề" dùng để lưu trữ các **đơn đồ thị vô hướng**. Hãy hoàn chỉnh chương trình bên dưới để đọc đồ thị từ bàn phím và in ra ma trận kề của đồ thị ra màn hình. Viết mã lệnh của bạn vào chỗ ba chấm (…).

```
//Khai báo hằng và thư viện
#include <stdio.h>
#define MAX_N 100

typedef struct {
        int n, m;
        int A[MAX_N][MAX_N];
} Graph;


//Viết mã lệnh của bạn ở đây
...
//Hết phần mã lệnh của bạn


//Hàm main()
int main() {
        Graph G;
        int n, m, u, v;
        //Đọc số đỉnh và số cung & khởi tạo đồ thị n đỉnh 0 cung
        scanf("%d%d", &n, &m);
        init_graph(&G, n);
        //Đọc m cung và thêm vào đồ thị
        for (int e = 0; e < m; e++) {
                scanf("%d%d", &u, &v);
                add_edge(&G, u, v);
        }
        //In ma trận kề của đồ thị
        for (int u = 1; u <= G.n; u++) {
                for (int v = 1; v <= G.n; v++)
                        printf("%d \n", G.A[u][v]);
                printf("\n");
        }
        return 0;
}
```

Mở IDE, lập trình và chạy thử.

Copy và paste hai hàm: init_graph() và add_edge() lên hệ thống ELSE.

1.3.5 Bài tập 7b

Tương tự bài 7b nhưng cho **đa đồ thị vô hướng, có thể có khuyên**.

1.3.6 Bài tập 7c

Tương tự bài 7a nhưng cho **đa đồ thị có hướng, có thể có khuyên**.

1.3.7 Bài tập 8a – MTK: hàm add_edge(), đơn đồ thị vô hướng

Cho cấu trúc dữ liệu Graph được cài đặt bằng phương pháp "Ma trận kề" dùng để lưu trữ các **đơn đồ thị vô hướng.**

```
#define MAX_N 100
typedef struct {
        int n, m;
        int A[MAX_N][MAX_N];
} Graph;
```

Hãy viết hàm void add_edge(Graph *pG, int u, int v) để thêm cung (u, v) vào đồ thị.

1.3.8 Bài tập 8b – MTK: hàm add_edge(), đơn đồ thị có hướng

Tương tự bài tập 8a nhưng cho **đơn đồ thị có hướng**.

1.3.9 Bài tập 8c – MTK: hàm add_edge(), đa đồ thị vô hướng

Tương tự bài tập 8a nhưng cho **đa đồ thị vô hướng**.

1.3.10 Bài tập 8d – MTK: hàm add_edge(), đa đồ thị có hướng

Tương tự bài tập 8a nhưng cho **đa đồ thị có hướng**.

1.3.11 Bài tập 9a – MTK: In ma trận kề của đơn đồ thị vô hướng

Viết chương trình cho phép người dùng nhập vào một **đơn đồ thị vô hướng**, in ma trận kề của nó ra màn hình.

1.3.12 Bài tập 9b – MTK: In ma trận kề của đơn đồ thị có hướng

Tương tự bài tập 9a nhưng cho **đơn đồ thị có hướng**.

1.3.13 Bài tập 9c – MTK: In ma trận kề của đa đồ thị vô hướng

Tương tự bài tập 9a nhưng cho **đa đồ thị vô hướng và có thể chứa khuyên (giả đồ thị)**.

1.3.14 Bài tập 9d – MTK: In ma trận kề của đa đồ thị có hướng

Tương tự bài tập 9a nhưng cho **đa đồ thị có hướng và có thể chứa khuyên (quiver).**

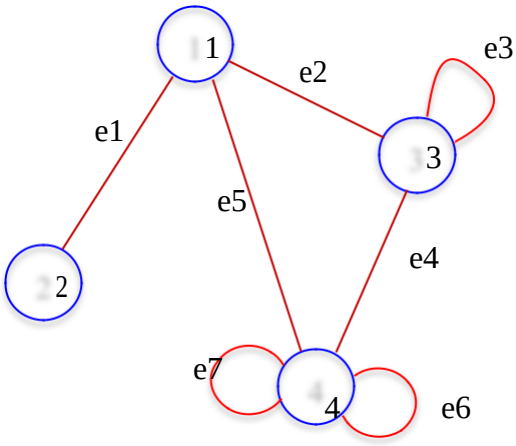
1.3.15 Kiểm tra đỉnh u có kề với đỉnh v không

Với cách biểu diễn này, cách kiểm tra hai đỉnh kề nhau khá đơn giản và trực tiếp. Ta chỉ cần kiểm tra phần tử A[u][v] có khác 0 hay không.

```
//Kiểm tra u có kề với v không
int adjacent(Graph *pG, int u, int v) {
        return pG->A[u][v] > 0;
}
```

1.3.16 Tính bậc của đỉnh u

Xét đồ thị vô hướng bên dưới và ma trận kề của nó.



Ma trận kề:

	1	**2**	**3**	**4**
1	0	1	1	1
2	1	0	0	0
3	1	0	**1**	1
4	1	0	1	**2**

Đối với *đỉnh không có khuyên, ví dụ:* deg(1) = 0 + 1 + 1 + 1 = 3

- Bậc của đỉnh u, deg(u) = tổng các phần tử trên hàng u

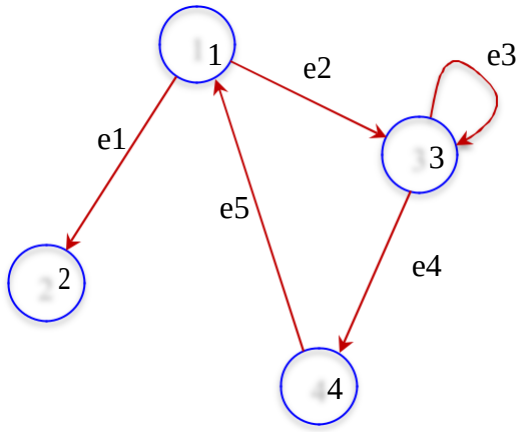
```
//Tính bậc của đỉnh u
int degree(Graph *pG, int u) {
	int deg_u = 0;
	//Tính tổng các phần tử trên hàng u
	for (int v = 1; v <= pG->n; v++)
		deg_u += pG->A[u][v];

	return deg_u;
}
```

Đối với *đỉnh có khuyên, ví dụ:* deg(3) = 1 + 0 + (1 + 1) + 1 = 4

- Phần tử trên đường chéo phải được cộng 2 lần (khuyên được tính 2 lần).

```
//Tính bậc của đỉnh u
int degree(Graph *pG, int u) {
	int deg_u = 0;
	//Tính tổng các phần tử trên hàng u
	for (int v = 1; v <= pG->n; v++)
		deg_u += pG->A[u][v];
	//Phần tử trên đường chéo phải được cộng 2 lần
	return deg_u + pG->A[u][u];
}
```

Xét đồ thị có hướng và ma trận kề của nó:



Ma trận kề:

	1	**2**	**3**	**4**
1	0	1	1	0
2	0	0	0	0
3	0	0	1	1
4	1	0	0	0

Đối với đồ thị có hướng (bao gồm cả trường hợp đa cung và khuyên),

- Bậc vào của u, deg_in(u) = tổng các phần tử của cột u.
- Bậc ra của u, deg_out(u) = tổng các phần tử của hàng u.
- Bậc của u, deg(u) = deg_in(u) + deg_out(u)

Vì thế bậc của đỉnh u = tổng các phần tử trên hàng u + tổng các phần tử trên cột u, ví dụ:

- deg(1) = (**0** + 1 + 1 + 0) + (**0** + 0 + 0 + 1) = 3
- deg(3) = (0 + 0 + **1** + 0) + (1 + 0 + **1** + 0) = 4

Theo cách tính này, phần tử trên đường chéo (ứng với khuyên) cũng được cộng 2 lần.

```
//Tính bậc của đỉnh u
int degree(Graph *pG, int u) {
        int deg_u = 0, v;
        //Tính tổng các phần tử trên hàng u
        for (v = 1; v <= pG->n; v++)
                deg_u += pG->A[u][v] + pG->A[v][u];
        //Khi cộng hàng u và cột u, phần tử A[u][u] đã được cộng 2 lần
        return deg_u;
}
```

1.3.17 Bài tập 10a – MTK: hàm degree(), vô hướng

Cho cấu trúc dữ liệu Graph được cài đặt bằng phương pháp "Ma trận kề" dùng để lưu trữ các **đa đồ thị vô hướng (có thể có khuyên)**, hãy viết hàm int degree(Graph *pG, int u) để tính bậc của đỉnh u.

1.3.18 Bài tập 10b – MTK: hàm degree(), có hướng

Tương tự bài tập 10a nhưng cho **đa đồ thị có hướng (có thể có khuyên)**.

1.3.19 Bài tập 10c – MTK: hàm indegree(), có hướng

Cho cấu trúc dữ liệu Graph được cài đặt bằng phương pháp "Ma trận kề" dùng để lưu trữ các **đa đồ thị có hướng (có thể có khuyên)**, hãy viết hàm int indegree(Graph *pG, int u) để tính bậc vào của đỉnh u.

1.3.20 Bài tập 10d – MTK: hàm outdegree(), có hướng

Cho cấu trúc dữ liệu Graph được cài đặt bằng phương pháp "Ma trận kề" dùng để lưu trữ các **đa đồ thị có hướng (có thể có khuyên)**, hãy viết hàm int outdegree(Graph *pG, int u) để tính bậc ra của đỉnh u.

1.3.21 Liệt kê các đỉnh kề của một đỉnh

Nếu đồ thị không chứa đa cung, áp dụng thuật toán tổng quát như trường hợp "Danh sách cung" hoặc đơn giản hơn như bên dưới:

```
//Liệt kê các đỉnh kề của đỉnh u bất kỳ
void neighbours(Graph* pG, int u) {
	int v;
	for (v = 1; v <= pG->n; v++)
		if (pG->A[u][v] != 0)
			printf("%d ", v);
	printf("\n");
}
```

Nếu đồ thị có chứa đa cung, ta liệt kê đỉnh v A[u][v] lần (do A[u][v] chứa số cung đi từ u đến v).

```
//Liệt kê các đỉnh kề của đỉnh u
void neighbours(Graph* pG, int u) {
	int v, j;
	for (v = 1; v <= pG->n; v++)
		for (j = 1; j <= pG->A[u][v]; j++)
			printf("%d ", v);
	printf("\n");
}
```

1.3.22 Bài tập 11a – MTK: liệt kê đỉnh kề của đồ thị vô hướng

Viết chương trình đọc vào một **đồ thị vô hướng** và in ra các đỉnh kề của các đỉnh.

1.3.23 Bài tập 11b – MTK: liệt kê đỉnh kề của đồ thị có hướng

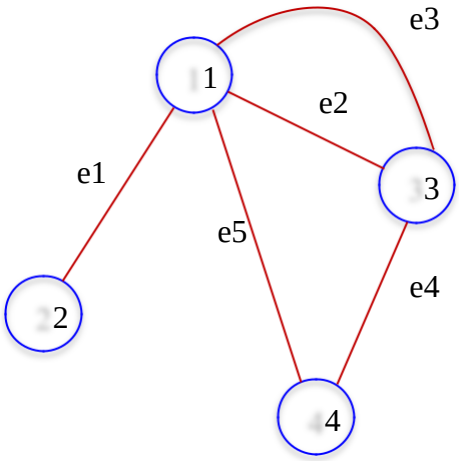
Viết chương trình đọc vào một **đồ thị có hướng** và in ra các đỉnh kề của các đỉnh.

1.4 Phương pháp danh sách đỉnh kề

Trong trường hợp đồ thị thưa (số lượng cung của đồ thị ít, mỗi đỉnh chỉ có một ít đỉnh kề với nó), ta có thể sử dụng phương pháp danh sách đỉnh kề để tiết kiệm không gian lưu trữ.

- Với mỗi đỉnh ta lưu các đỉnh kề với nó vào trong một danh sách.
- Nếu đồ thị không chứa đa cung: danh sách đỉnh kề không chứa các phần tử trùng nhau. Ngược lại nếu đồ thị có chứa đa cung, danh sách đỉnh kề sẽ có thể chứa nhiều đỉnh giống nhau.
- Đồ thị sẽ bao gồm các danh sách đỉnh kề của tất cả các đỉnh trong đồ thị (một mảng các danh sách).

Ví dụ:



Các danh sách đỉnh kề:

adj[1] = [2, 3, 3, 4]

adj[2] = [1]

adj[3] = [1, 1, 4]

adj[4] = [1, 3]

1.4.1 Cài đặt

Sử dụng một cấu trúc dữ liệu gồm:

- Số đỉnh của đồ thị: n
- Một mảng các danh sách: adj[]

Giả sử ta đã có CTDL List (xem lại học phần CTDL) dùng để lưu trữ các số nguyên, CTDL Graph sẽ được cài đặt như sau:

```
//Khai báo hằng và thêm thư viện
#include <stdio.h>
#define MAX_N 100

typedef struct {
        int n;                  //số đỉnh
        List adj[MAX_N];        //mảng các danh sách các đỉnh kề
} Graph;
```

Bên dưới là một bản cài đặt đơn giản của CTDL List và các phép toán cơ bản trên nó.

```
//Khai báo CTDL List và các phép toán cơ bản
#define MAX_ELEMENTS 100
typedef int ElementType;

typedef struct {
        ElementType data[MAX_ELEMENTS];
        int size;
} List;

//Tạo danh sách rỗng
void make_null(List *pL) {
        pL->size = 0;
}

//Thêm một phần tử vào cuối danh sách
void push_back(List *pL, ElementType x) { pL-
        >data[pL->size] = x; pL->size++;

}

//Lấy phần tử thứ i, phần tử bắt đầu có vị trí 1
ElementType element_at(List *pL, int i) { return pL-
        >data[i-1];
}

//Trả về số phần tử của danh sách
int count_list(List *pL) {
        return pL->size;
}
```

1.4.2 Khởi tạo đồ thị

- Gán số đỉnh cho n
- Khởi tạo các danh sách kề rỗng

```
//Khởi tạo đồ thị có n đỉnh và 0 cung
void init_graph(Graph *pG, int n) {
        int u;
        pG->n = n;


        //Khởi tạo các danh sách kề rỗng
        for (u = 1; u <= n; u++)
                make_null(&pG->adj[u]);
}
```

1.4.3 Thêm cung vào đồ thị

Đối với đồ thị vô hướng:

- v là đỉnh kề của u: thêm v vào danh sách kề của u (adj[u]).
- u là đỉnh kề của v: thêm u vào danh sách kề của v (adj[v]).

Đối với đồ thị có hướng:

- v là đỉnh kề của u: thêm v vào danh sách kề của u (adj[u]).

```
//Thêm cung (u, v) vào đồ thị vô hướng *pG
void add_edge(Graph *pG, int u, int v) {
        push_back(&pG->adj[u],          v);//v là kề của u
        push_back(&pG->adj[v], u);          //u là kề của v
}
```

Chú ý:

- **Đối với đồ thị có hướng**, ta không thêm u vào danh sách kề của v.

1.4.4 Kiểm tra u có kề với v không

Với cách biểu diễn này, để kiểm tra u có kề với v không, ta kiểm tra xem v có nằm trong danh sách kề của u không.

```
//Thêm cung e = (u, v) vào đồ thị vô hướng *pG
int adjacent(Graph *pG, int u, int v) {
        int j;
        for (j = 1; j <= pG->adj[u].size; j++)
                if (element_at(&pG->adj[u], j) == v)
                        return 1; //v nằm trong dánh sách adj[u] return
        0; //v không có trong danh sách adj[u]
}
```

1.4.5 Tính bậc của một đỉnh

Theo định nghĩa, bậc của 1 đỉnh = số cung liên thuộc với nó. Với cách biểu diễn này, số cung liên thuộc với một đỉnh chính là số đỉnh kề của đỉnh. Ta trả về số phần tử trong danh sách đỉnh kề của đỉnh u hay adj[u].size.

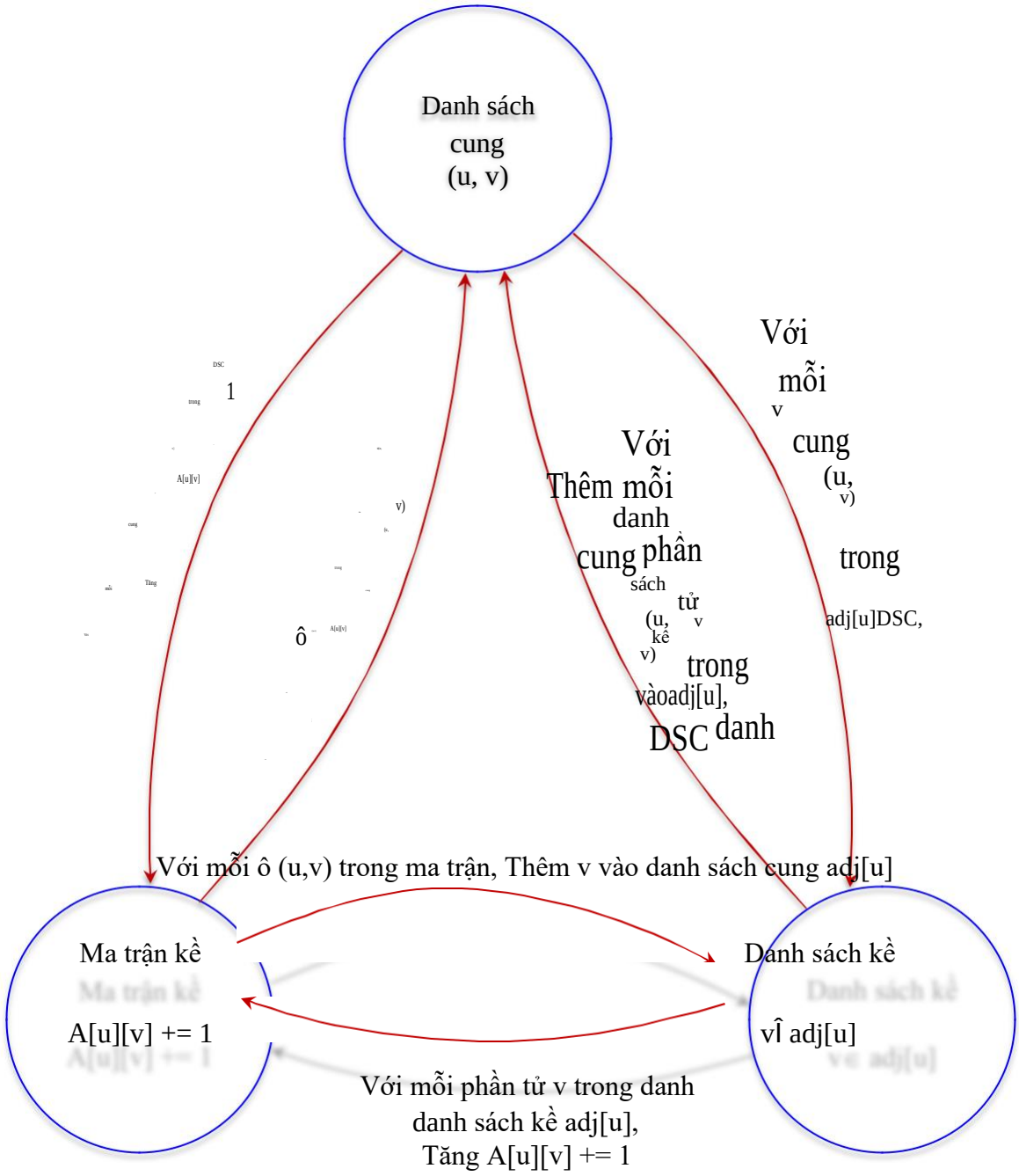
1.4.6 Liệt kê các đỉnh kề của một đỉnh

Chỉ cần liệt kê các đỉnh trong danh sách adj[u] hoặc sử dụng thuật toán tổng quát như hai phương pháp trên.

1.4.7 Bài tập 12

Làm lại các bài tập trong phương pháp "ma trận kề" với cấu trúc dữ liệu Graph được cài đặt theo phương pháp "Danh sách kề".

1.5 Chuyển đổi qua lại giữa các phương pháp biểu diễn



Chuyển đổi từ danh sách cung sang các dạng khác:

- Lần lượt đọc từng cung (u, v), ta gọi hàm add_edge(u, v) để thêm cung (u, v) vào đồ thị. Xem lại các phần nhập dữ liệu bên trên.

Chuyển từ ma trận kề sang các dạng khác:

- Đọc từng phần tử của ma trận kề. Giả sử phần tử A[u][v] = k (có nghĩa là có k cung đi từ u đến v), gọi hàm add_edge() k lần để k cung (u, v).
- Chú ý: đối với đồ thị vô hướng, cung (u, v) được lưu ở 2 chỗ: A[u][v] và A[v][u].

Chuyển từ danh sách kề sang các dạng khác:

- Đọc từng danh sách kề.
- Với mỗi danh sách kề adj[u], đọc từng phần tử v của danh sách, gọi hàm add_edge(u, v) để thêm cung (u, v).
- Chú ý: đối với đồ thị vô hướng, cung (u, v) được lưu ở 2 chỗ: adj[u] chứa v và adj[v] chứa u.

1.5.1 Bài tập 13a. MT kề => DS cung (vô hướng)

Viết chương trình nhập vào ma trận kề của một đồ thị vô hướng, in ra danh sách cung ra màn hình.

1.5.2 Bài tập 13a. MT kề => DS cung (có hướng)

Viết chương trình nhập vào ma trận kề của một đồ thị có hướng, in ra danh sách cung ra màn hình.

1.5.3 Bài tập 14a. MT kề => DS kề (vô hướng)

Viết chương trình nhập vào ma trận kề của một đồ thị vô hướng, in ra danh sách kề của các đỉnh

1.5.4 Bài tập 14b. MT kề => DS kề (có hướng)

Viết chương trình nhập vào ma trận kề của một đồ thị có hướng, in ra danh sách kề của các đỉnh.

1.5.5 Bài tập 15a. DS kề => MT kề (vô hướng)

Viết chương trình nhập vào danh sách kề của các đỉnh của một đồ thị vô hướng, in ra ma trận kề của nó.

1.5.6 Bài tập 15b. DS kề => MT kề (có hướng)

Viết chương trình nhập vào danh sách kề của các đỉnh của một đồ thị có hướng, in ra ma trận kề của nó.

1.6 Phương pháp Ma trận đỉnh – cung (ma trận liên thuộc):

Phương pháp này tuy không phổ biến lắm nhưng cũng là một phương pháp để biểu diễn đồ thị

- Ngoài việc đánh số đỉnh từ 1 đến n, ta phải đánh số cung từ 1 đến m.
- Thường dùng để biểu diễn **đa đồ thị vô hướng** (chấp nhận đa cung, tuy nhiên lại không lưu được khuyên).

Dùng 1 ma trận n hàng, m cột: A = {a_{ue}} với u = 1, 2, …, n, e = 1, 2, …, m.

Phần tử hàng u, cột e có giá trị

- a_{ue} = 1 nếu đỉnh u liên thuộc với cung e.
- a_{ue} = 0, các trường hợp khác.

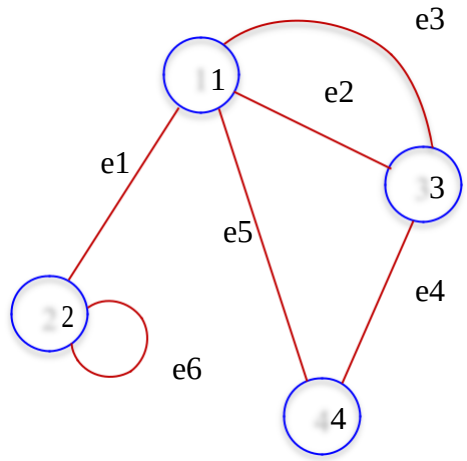
Ta có thể mở rộng phương pháp này để lưu trữ khuyên e = (u, u):

- a_{ue} = 2

Để lưu trữ đồ thị có hướng ta có thể mở rộng như sau:

- a_{ue} = +1 nếu u là đỉnh gốc của cung e.
- a_{ue} = -1, nếu u là đỉnh ngọn của cung e.

Ví dụ:



Ma trận liên thuộc

	e1	**e2**	**e3**	**e4**	**e5**	**e6**
1	1	1	1	0	1	0
2	1	0	0	0	0	**2**
3	0	1	1	1	0	0
4	0	0	0	1	1	0

1.6.1 Cài đặt

Sử dụng một cấu trúc gồm các trường sau:

- A[][]: mảng hai chiều lưu ma trận đỉnh – cung
- n: số đỉnh
- m: số cung

```
//Khai báo cấu trúc dữ liệu Graph
#define MAX_N 100
#define MAX_M 500

typdef struct {
        int n, m;                       //n: số đỉnh, m: số cung
        int A[MAX_N][MAX_M];            //Ma trận liên thuộc
} Graph;
```

1.6.2 Khởi tạo đồ thị n đỉnh, m cung

- Gán số đỉnh cho n
- Gán số cung cho m
- Khởi tạo ma trận A chứa toàn số 0

```
//Khởi tạo đồ thị gồm n đỉnh, m cung
void init_graph(Graph *pG, int n, int m) { int u, e;

        pG->n = n;
        pG->m = m;
        for (u = 1; u <= n; u++)
                for (e = 1; e <= m; e++)
                        pG->A[u][e] = 0;
}
```

Chú ý: phép toán này chỉ khởi tạo đồ thị để DỰ TRÙ chứa m cung thôi chứ chưa thêm các cung vào đồ thị. Để thêm các cung vào đồ thị ta phải sử dụng phép toán thêm cung: add_edge(G, e, u, v).

1.6.3 Thêm cung e = (u, v) vào đồ thị

Phép toán này gồm hai phần:

- Cho đỉnh u liên thuộc với cung e: A[u][e] = 1
- Cho đỉnh v liên thuộc với cung e: A[v][e] = 1

```
//Thêm cung e = (u, v) vào đồ thị vô hướng *pG
void init_graph(Graph *pG, int n, int m) { pG->A[u][e] = 1;
       //u liên thuộc với e pG->A[v][e] = 1; //v
       liên thuộc với e
}
```

Để lưu trữ khuyên ta thay = 1 bằng += 1.

1.6.4 Kiểm tra đỉnh u có kề với đỉnh v không

Đối với đồ thị vô hướng không chứa khuyên:

- Ta duyệt qua từng cung e (từ 1 đến m) và kiểm tra xem u và v có cùng liên thuộc với e không: A[u][e] == 1 **VÀ** A[u][e] == 1.
- Nếu có một cung nào đó mà nó liên thuộc với u và v thì trả về 1 (TRUE).
- Nếu duyệt qua hết các cung mà vẫn không có cung nào liên thuộc đồng thời với u và v thì trả về 0 (FALSE).

```
//Kiểm tra u kề với v
int adjacent(Graph *pG, int u, int v) {
        int e;
        for (e = 1; e <= pG->m; e++)
                if (pG->A[u][e] == 1 && pG->A[v][e] = 1)
                        return 1;       //có cung e liên thuộc với cả u và v
        return 0;                       //không có cung nào liên thuộc với cả u và v
}
```

1.6.5 Tính bậc của một đỉnh

Theo định nghĩa, bậc của 1 đỉnh = số cung liên thuộc với nó.

Thuật toán tính bậc của đỉnh u:

- Đếm trên hàng u (tương ứng với đỉnh u) xem có bao nhiêu số 1 (tương ứng với số cung liên thuộc với u).

```
//Kiểm tra u kề với v
int degree(Graph *pG, int u) {
        int e, deg_u = 0;
        for (e = 1; e <= pG->m; e++)
                if (pG->A[u][e] == 1)
                        deg_u++;
        return deg_u;
}
```

1.6.6 Bài tập 16.

Làm lại các bài tập trong phương pháp "ma trận kề" với cấu trúc dữ liệu Graph được cài đặt theo phương pháp "Ma trận đỉnh – cung".

1.7 Nâng cao

Bạn có thể bỏ qua phần này trong các buổi thực hành. Tuy nhiên, sẽ tốt hơn cho bạn nếu bạn tự tìm hiểu thêm để nâng cao kỹ năng lập trình cho mình.

1.7.1 Truyền tên file bằng đối số dòng lệnh

Ta thấy rằng mặc dù sử dụng file để biểu diễn đồ thị và chương trình tự đọc và xây dựng đồ thị nhưng tên file "dothi.txt" vẫn còn nằm trong chương trình. Điều này sẽ làm cho chương trình gắn cứng với tên "dothi.txt" này. Ta không thể sử dụng đồ thị với tên file khác. Để mềm dẻo hơn, tên file chứa đồ thị sẽ truyền cho chương trình khi chạy (thực thi). Có nhiều cách để làm điều này ví dụ bạn có thể yêu cầu người dùng nhập tên file chứa đồ thị bằng hàm scanf() hay gets(). Tuy nhiên, các ngôn ngữ như C/C++ cho phép ta truyền tham số từ bên ngoài khi chạy chương trình thông qua hàm main(). Để thực hiện điều này, bạn cần có kiến thức về *chạy chương trình bằng dòng lệnh* (command line).

Ta viết lại chương trình như sau (tên file: **nang-cao.c**):

```
//Khai báo thư viện
#include <stdio.h>

Hãy bổ sung các khai báo và cài đặt hàm cần thiết để tạo thành chương trình hoàn
chỉnh. Bạn có thể sử dụng 1 trong 4 phương pháp biểu diễn đồ thị.


//Hàm main()
int main(int argc, char *argv[]) {
        Graph G;
        int n, m, e, u, v;

        //Kiểm tra đối số để lấy tên file
        if (argc < 2) {
                printf("Hay go: %s <ten-file>\n", argv[0]); return 1;

        }

        //Chuyển dữ liệu từ file argv[1] sang dòng nhập chuẩn
        freopen(argv[1], "r", stdin);

        //Đọc số đỉnh và số cung & khởi tạo đồ thị
        scanf("%d%d", &n, &m);
        init_graph(&G, n);
        //Đọc m cung và thêm vào đồ thị
        for (e = 0; e < m; e++) {
                scanf("%d%d", &u, &v);
                add_edge(&G, u, v);
        }

        for (u = 1; u <= G.n; u++)
                printf("deg(%d) = %d\n", u, degree(&G, u)); return 0;

}
```

Trong chương trình trên, ta thay đổi danh sách các đối số của hàm main() thành:

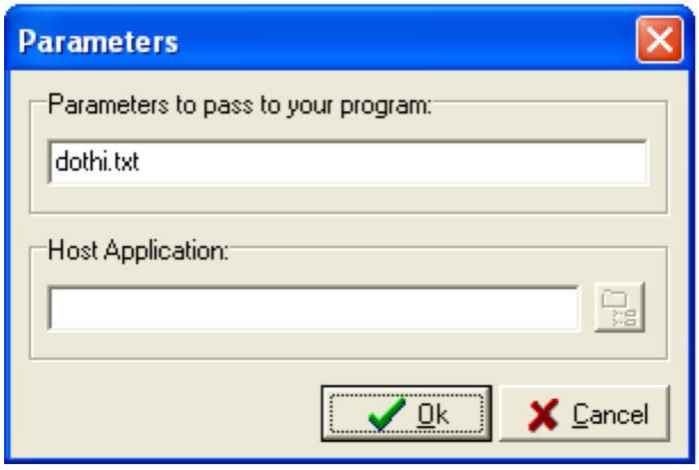
```
int main(int argc, char *argv[])
```

Tham số argc là số lượng tham số ta truyền vào khi chạy chương trình (kể cả tên chương trình thực thi) và các tham số được truyền vào được lưu ở argv (mảng các chuỗi). Tham số đầu tiên argv[0] là tên của tập tin thực thi, argv[1] là tham số thứ 2, argv[2] là tham số thứ 3, …

Để chạy chương trình, mở cửa sổ để gõ lệnh (Run > cmd).

Từ dấu nhắc ta gõ: **nang-cao.exe dothi.txt**

Hoặc từ Dev-C++, chọn menu **Execute/Parameters**: điền tên file chứa đồ thị vào.



Khi gọi chương trình như thế, hàm main() sẽ nhận vào argc = 2 và argv là một mảng có hai phần tử: argv[0] = "nang-cao.exe", argv[1] = "dothi.txt".

1.7.2 Sử dụng các CTDL nâng cao

Trong phần cài đặt cấu trúc dữ liệu Graph sử dụng danh sách các đỉnh kề, ta đã sử dụng một CTDL List tự định nghĩa để lưu trữ các số nguyên. Nếu bạn đã quen thuộc hoặc biết đôi chút về ngôn ngữ C++, bạn có thể sử dụng các cấu trúc dữ liệu sẵn có của thư viện STL để tiết kiệm thời gian cài đặt các chương trình của mình. Cấu trúc dữ liệu vector của STL cho phép bạn lưu trữ một danh sách các đối tượng bất kỳ ví dụ như số nguyên, số thực, ký tự, …

Để sử dụng tính năng này bạn phải viết chương trình dùng ngôn ngữ C++ (mở rộng hơn và dễ hơn so với C) và phải đặt tên chương trình có phần mở rộng là *.cpp hoặc *.cc.

```
//Khai báo hằng và thêm thư viện
#include <stdio.h>

//Khai báo thư viện vector, vector có thể xem như một danh sách
#include <vector>
using namespace std;

#define MAX_N 100

typedef struct {
        int n; //số đỉnh n vector<int> adj[MAX_N]; //mảng các
        vector<int>
} Graph;
```

Ta có thể sử dụng vector<int> như một danh sách chứa các số nguyên.

Hàm init_graph():

```
//Khởi tạo đồ thị có n đỉnh và 0 cung
void init_graph(Graph *pG, int n) {
        int u;
        pG->n = n;

        //Khởi tạo các danh sách kề rỗng
        for (u = 1; u <= n; u++)
                pG->adj[u].clear();             //làm rỗng danh sách adj[u]
}
```

Hàm add_edge() có thể viết lại như sau:

```
//Thêm cung (u, v) vào đồ thị vô hướng *pG
void add_edge(Graph *pG, int u, int v) {
        pG->adj[u].push_back(v);                //thêm v vào danh sách adj[u]
        pG->adj[v].push_back(u);                //thêm u vào danh sách adj[v]
  }
```

Một số lưu ý trong chương trình:

- Để có thể sử dụng được CTDL vector, bạn cần phải thêm 2 dòng: #include <vector> và using namespace std; vào đầu chương trình.
- vector<int> là một CTDL danh sách dùng để lưu các số nguyên.
- Hàm push_back(v) cho phép thêm 1 số nguyên v vào cuối danh sách, ví dụ: ○ pG->adj[u].push_back(v);
- Hàm size() trả về số phần tử trong danh sách, ví dụ: L.size() trả về só phần tử của danh sách L.
- Phép toán [i] dùng để lấy phần tử tại vị trí thứ i trong danh sách (thứ tự tính từ 0), ví dụ: L[0] trả về phần tử đầu tiên trong danh sách L, L[2] trả về phần tử thứ 3 trong danh sách L.

1.7.3 Bài tập 17 (*)

Làm lại các bài tập trong phương pháp "Ma trận kề" với cấu trúc dữ liệu Graph được cài đặt theo phương pháp "Danh sách kề" sử dụng cấu trúc dữ liệu vector<int> của STL thay vì kiểu List tự định nghĩa.