BUỔI 1. BIỂU DIỄN ĐỒ THỊ

Mục đích:

- Biểu diễn đồ thị trên máy tính
- Cài đặt 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 của chúng ta và hỗ trợ một số phép toán trên đồ thị:

- adjacent (G, x, y): kiểm tra xem y có phải là đỉnh kề của x không.
- neighbors(G, x): trả về danh sách các đỉnh kề của x.
- add(G, x, y): thêm cung (x, y) vào đồ thị nếu có chưa tồn tại.
- delete(G, x, y): xoá cung (x, y) ra khỏi đồ thị.
- degree(G, x): trả về bậc của đỉnh x.

Giả sử ta có đồ thị G = <V, E> có n đỉnh, m cạnh. Xét một số cách biểu diễn đồ thị trên máy tính sau.

1.2 Ma trận đỉnh – cung (ma trận liên thuộc):

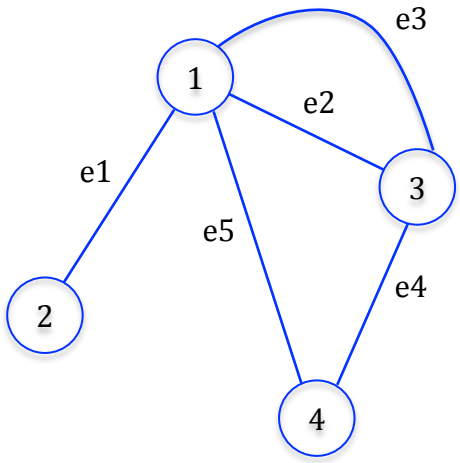
- Đánh số đỉnh từ 1 đến n.
- Đánh số cung từ 1 đến m.
- Thường dùng để biểu diễn **đồ thị vô hướng** tổng quát (chấp nhận đa cung, tuy nhiên lại không lưu được khuyên).

Dùng 1 ma trận n hàng, m cột: A = $\{a_{ij}\}$ với i = 1..n, j = 1..m.

Phần tử hàng i, cột j có giá trị

- a_{ij} = 1 nếu đỉnh i liên thuộc với cung j.
- a_{ij} = 0, các trường hợp khác.

Ví dụ:



Ma trận liên thuộc

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

1.2.1 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

```
#define MAX_VERTICES 100
#define MAX_EDGES 500

typedef struct {
  int n, m; /* n: so dinh, m: so cung */
  /* ma tran dinh - cung */
  int A[MAX_VERTICES][MAX_EDGES];
} Graph;
```

Sử dụng phương pháp biểu diễn này ta có thể cài đặt các phép toán trên ma trận.

1.2.2 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

```
/* Khoi tao do thi G co n dinh, m cung */
void init_graph(Graph* G, int n, int m) {
  int i, j;
  G->n = n;
  G->m = m;
  for (i = 1; i <= n; i++)
    for (j = 1; j <= m; j++)
      G->A[i][j] = 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(G, e, x, y).

1.2.3 Thêm cung e = (x, y) vào đồ thị

Phép toán này gồm hai phần:

- Cho đỉnh x và cung e liên thuộc với nhau: A[x][e] = 1
- Cho đỉnh y và cung e liên thuộc với nhau: A[y][e] = 1

```
/* Them cung e = (x, y) vao do thi G */
void add_edge(Graph* G, int e, int x, int y) {
  G->A[x][e] = 1; //x lien thuoc voi e
  G->A[y][e] = 1; //y lien thuoc voi e
}
```

1.2.4 Kiểm tra đỉnh y có kề với đỉnh x không

Theo định nghĩa, y kề với x ⇔ có cung nối đỉnh x với y. Đối với đồ thị vô hướng, cung (x, y) và (y, x) là như nhau.

Giải thuật:

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

```
/* Kiem tra y co ke voi x khong */
int adjacent(Graph* G, int x, int y) {
  int e;
  for (e = 1; e <= G->m; e++)
    if (G->A[x][e] == 1 && G->A[y][e] == 1)
      return 1;
  return 0;
}
```

1.2.5 Tính bậc của một đỉnh

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

Giải thuật:

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

```
/* Tinh bac cua dinh x: deg(x) */
int degree(Graph* G, int x) {
  int e, deg = 0;
  for (e = 1; e <= G->m; e++)
    if (G->A[x][e] == 1)
      deg++;
  return deg;
}
```

1.2.6 Bài tập 1

Biểu diễn đồ thị trong ví dụ trên và in ra bậc của các đỉnh ra màn hình (tên file: **buoi1-bai1.c**)

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 = 4, số cung = 5
  - Gọi hàm add_edge 5 lần để thêm 5 cung vào đồ thị

- Cho một vòng lặp với biến v chạy từ đỉnh 1 đến đỉnh 4, gọi hàm degree(v) để tính bậc của v.

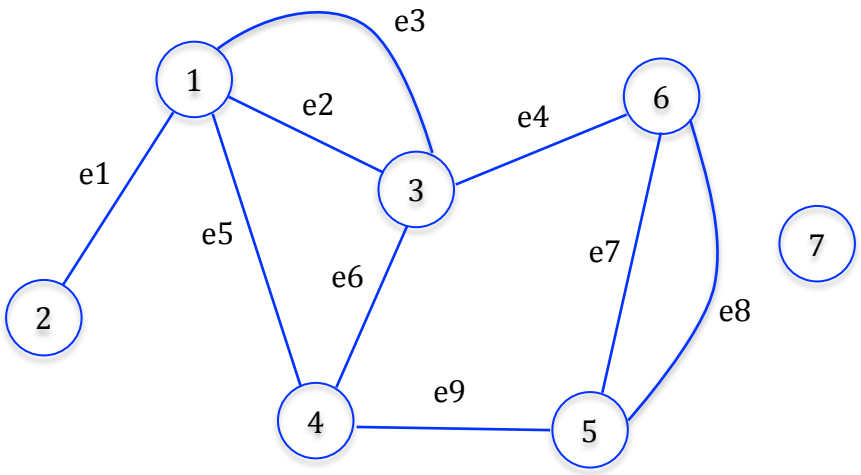
```
#include <stdio.h>

Hãy bổ sung các khai báo và cài đặt hàm cần thiết để tạo thành chương trình hoàn chỉnh

/* Ham main() */
int main() {
  Graph G;
  int n = 4, m = 5, v;
  init_graph(&G, n, m);
  add_edge(&G, 1, 1, 2);
  add_edge(&G, 2, 1, 3);
  add_edge(&G, 3, 1, 3);
  add_edge(&G, 4, 3, 4);
  add_edge(&G, 5, 1, 4);

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

Nếu bạn cài đặt đúng, kết quả sẽ là:

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

1.2.7 Bài tập 2

Dựa vào bài tập 1, biểu diễn và in ra bậc của các đỉnh của đồ thị sau (tên file: **buoi1-bai2.c**):

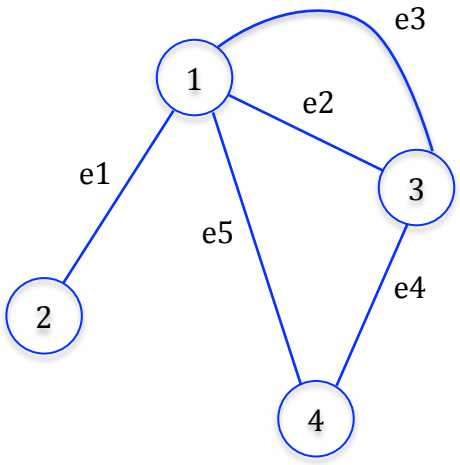


1.2.8 Nhập dữ liệu từ tập tin

Trong hai 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 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. Chúng 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: file dothi.txt đặt cùng thư mục với file chương trình.

```
#include <stdio.h>

Hãy bổ sung các khai báo và cài đặt hàm cần thiết để tạo
thành chương trình hoàn chỉnh

/* Ham main() */
int main() {
  Graph G;
  int n, m, e, u, v;

  FILE* file = fopen("dothi.txt", "r");
  fscanf(file, "%d%d", &n, &m);
  init_graph(&G, n, m);
  for (e = 1; e <= m; e++) {
    fscanf(file, "%d%d", &u, &v);
    add_edge(&G, e, u, v);
  }

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

1.2.9 Bài tập 3

Sử dụng mẫu chương trình trên biểu diễn và in ra bậc của các đỉnh của đồ thị trong bài 2 (tên file: **buoi1-bai3.c**). So sánh kết quả với bài 2.

1.2.10 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.

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 dẽo 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 lệnh 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 lòng lệnh* (command line).

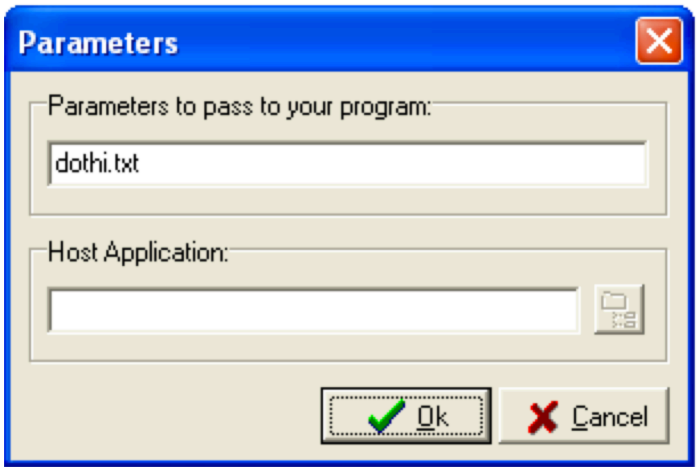
Ta viết lại chương trình như sau (tên file: **buoi1-bai4.c**):

```
#include <stdio.h>

Hãy bổ sung các khai báo và cài đặt hàm cần thiết để tạo
thành chương trình hoàn chỉnh

/* Ham main() */

int main(int argc, char* argv[]) {
  Graph G;
  int n, m, e, u, v;
  if (argc < 2) {
    printf("Hay go: %s <ten-file>\n", argv[0]);
    return 1;
  }

  FILE* file = fopen(argv[1], "r");
  fscanf(file, "%d%d", &n, &m);
  init_graph(&G, n, m);
  for (e = 1; e <= m; e++) {
    fscanf(file, "%d%d", &u, &v);
    add_edge(&G, e, u, v);
  }

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

Để chạy chương trình, từ dấu nhắc ta gõ: **buoi1-bai4.exe dothi.txt**

```
D:\>
D:\>
D:\>buoi1-bai4.exe dothi.txt_
```

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



1.3 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). Ta cần phải:

- Đánh số đỉnh từ 1 đến n.
- Thường dùng để biểu diễn **ĐƠN** đồ thị có hướng hoặc **ĐƠN** đồ thị vô hướng (không chấp nhận đa cung, nhưng có thể lưu được khuyên).

Dùng 1 ma trận vuông n hàng, n cột: A = {a_{ij}} với i = 1..n, j = 1..n.

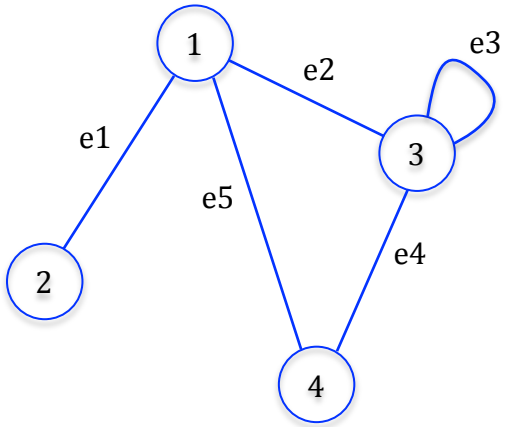
Phần tử hàng i, cột j có giá trị

- a_{ij} = 1 nếu đỉnh i kề với đỉnh j.
- a_{ij} = 0 ngược lại.

Ma trận kề mô tả mối quan hệ kề nhau giữa hai đỉnh.

Trường hợp đồ thị có chứa khuyên, thì phần tử a_{ii} tương ứng với đỉnh i sẽ có giá trị 1.

Ví dụ:

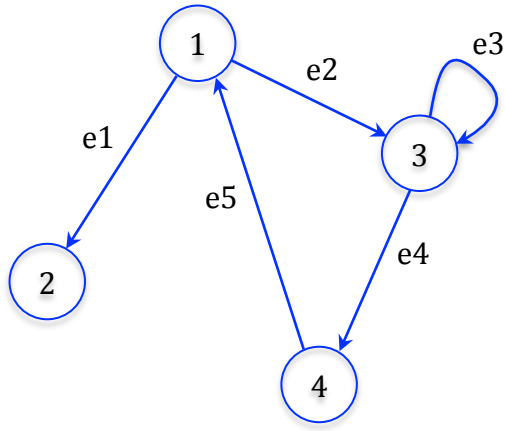


Ma trận kề:

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

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

Ta cũng có thể dùng ma trận kề để biểu diễn đồ thị có hướng, ví dụ:



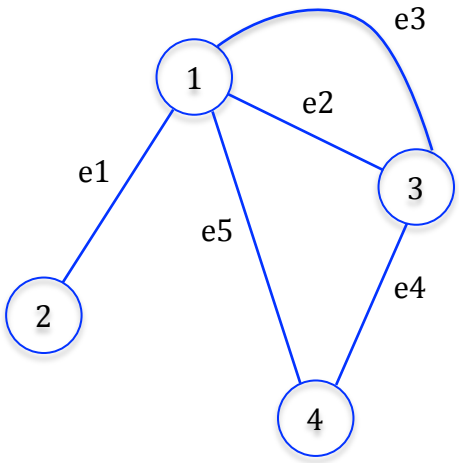
Ma trận kề:

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

Chú ý: Do ta sử dụng đỉnh đầu và đỉnh cuối của cung để mô tả cung nên chỉ có thể biểu diễn được đơn cung. Tuy nhiên, nếu nhu cầu chỉ là biểu diễn số lượng cung giữa hai đỉnh, ta cũng có thể biến đổi một chút phương pháp này để biểu diễn đa cung:

- a_{ij} = số lượng cung nối i với đỉnh j.

Ví dụ:



	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

```
#define MAX_VERTICES 100

typedef struct {
  int n; /* n: so dinh */
  /* ma tran dinh – dinh */
  int A[MAX_VERTICES][MAX_VERTICES];
} Graph;
```

Sử dụng phương pháp biểu diễn này ta có thể cài đặt các phép toán trên ma trận.

1.3.2 Khởi tạo đồ thị n đỉnh

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

```
/* Khoi tao do thi G co n dinh */
void init_graph(Graph* G, int n) {
  int i, j;
  G->n = n;
  for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
      G->A[i][j] = 0;
}
```

Chú ý: phép toán này chỉ khởi tạo đồ thị để DỰ TRÙ chứa các 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(G, x, y).

1.3.3 Thêm cung e = (x, y) vào đồ thị

Đối với đồ thị vô hướng:

- Cho đỉnh y kề với đỉnh x: A[x][y] = 1
- Cho đỉnh x kề với đinht y: A[y][x] = 1

```
/* Them cung e = (x, y) vao do thi G */
void add_edge(Graph* G, int x, int y) {
  G->A[x][y] = 1; //y ke với x
  G->A[y][x] = 1; //x ke voi y
}
```

Chú ý:

- **Nếu đồ thị chứa các đa cung**, ta phải cộng dồn số cung vào ô G->A[x][y] (G->A[x][y] += 1) chứ không phải gán = 1.
- **Nếu G là đồ thị có hướng**, ta không thay đổi giá trị của G->A[y][x]. Ma trận kề sẽ không đối xứng.

1.3.4 Kiểm tra đỉnh y có kề với đỉnh x 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[x][y] có khác 0 hay không.

```
/* Kiem tra y co ke voi x khong */
int adjacent(Graph* G, int x, int y) {
  return G->A[x][y] != 0;
}
```

Giải thuật này cũng chạy được với đồ thị có đa cung (phần tử A[x][y] > 1).

1.3.5 Tính bậc của một đỉnh

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

Giải thuật:

- Để tìm bậc của đỉnh x, ta chỉ cần đếm trên hàng x xem có bao nhiêu số > 0 (tương ứng với số đỉnh kề với x = số cung liên thuộc với x)

```
/* Tinh bac cua dinh x: deg(x) */
int degree(Graph* G, int x) {
  int y, deg = 0;
  for (y = 1; y <= G->n; y++)
    if (G->A[x][y] > 0)
      deg++;
  return deg;
}
```

Để giải thuật chạy được với đa cung, ta sửa một chút: thay vì đếm số đỉnh kề, ta đếm số cung bằng cách cộng dồn các giá trị G->A[x][y].

```
/* Tinh bac cua dinh x: deg(x), truong hop da cung */
int degree(Graph* G, int x) {
  int y, deg = 0;
  for (y = 1; y <= G->n; y++)
    deg += G->A[x][y];
  return deg;
}
```

Ta không cần kiểm tra G->A[x][y] có lớn hơn 0 hay không mà chỉ cần cộng dồn vào là được.

1.3.6 Bài tập 4

Làm lại bài 1 với phương pháp biểu diễn bằng ma trận kề (tên file: **buoi1-bai4.c**)

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 = 4
  - Gọi hàm add_edge 5 lần để thêm 5 cung vào đồ thị
  - Cho một vòng lặp với biến v chạy từ đỉnh 1 đến đỉnh 4, gọi hàm degree(v) để tính bậc của v.

```
#include <stdio.h>

Hãy bổ sung các khai báo và cài đặt hàm cần thiết để tạo
thành chương trình hoàn chỉnh

/* Ham main() */
int main() {
  Graph G;
  int n = 4, v;
  init_graph(&G, n);
  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);

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

Nếu bạn cài đặt đúng, kết quả sẽ giống như bài 1.

1.3.7 Bài tập 5

Làm lại bài 3 (đọc đồ thị từ tập tin dothi.txt) với phương pháp biểu diễn bằng ma trận kề (tên file: **buoi1-bai5.c**). Chú ý với các đa cung, ta phải cộng dồn số cung vào ô G->A[x][y] chứ không phải gán = 1.

1.3.8 Tìm các đỉnh kề với x

Ta đã có biết cách kiểm tra hai đỉnh có kề nhau không nên việc lấy danh sách các đỉnh kề của đỉnh x sẽ khá dễ dàng.

Giải thuật:

- Duyệt qua các đỉnh y từ 1 đến n
- Kiểm tra xem y có kề với x không, nếu có thêm vào danh sách kết quả.

Giả sử bạn đã biết cách cài đặt một cấu trúc dữ liệu danh sách (xem môn học cấu trúc dữ liệu). Ta sẽ dùng cấu trúc dữ liệu danh sách để lưu các đỉnh kề của một đỉnh.

Đoạn chương trình bên dưới định nghĩa một cấu trúc dữ liệu danh sách List dùng để lưu trữ các số nguyên. Bạn có thể dùng nó để lưu các đỉnh kề của x. Bạn cũng có thể tự mình định nghĩa một cấu trúc dữ liệu khác để lưu trữ theo ý bạn.

Hãy thêm đoạn chương trình này chương trình của bạn để có thể sử dụng cấu trúc dữ liệu danh sách

```
/* KHAI BAO VA DINH NGHIA CTDL DANH SACH */
#define MAX_ELEMENTS 100
typedef int ElementType;

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

/* Tao danh sach rong */
void make_null(List* L) {
  L->size = 0;
}

/* Them mot phan tu vao cuoi danh sach */
void push_back(List* L, ElementType x) {
  L->data[L->size] = x;
  L->size++;
}

/* Lay phan tu tai vi tri i, phan tu bat dau o vi tri 1
*/
ElementType element_at(List* L, int i) {
  return L->data[i-1];
}

/* Tra ve so phan tu cua danh sach */
int count_list(List* L) {
  return L->size;
}
/*****************************************/
```

Với CTDL List đã có giải thuật lấy các đỉnh kề của một định sẽ được cài đặt như sau:

```
/* Tim cac dinh ke cua dinh x */
List neighbors(Graph* G, int x) {
  Graph G;
  int y;
  List list;
  make_null(&list);
  for (y = 1; i <= G->n; y++)
    if (adjacent(G, x, y))
      push_back(&list, y);
  return list;
}
```

1.3.9 Bài tập 6

Viết chương trình đọc đồ thị từ tập tin như bài 5 (hoặc cũng có thể tạo đồ thị trong chương trình như bài 4). Liệt kê danh sách các đỉnh kề của tất cả các đỉnh.

```
#include <stdio.h>

Hãy bổ sung các khai báo và cài đặt hàm cần thiết để tạo
thành chương trình hoàn chỉnh

/* Ham main() */
int main() {
  Graph G;
  int y;
  /* Biểu diễn đồ thị */

  ...

  /* Liệt kê danh sách kề của các đỉnh */
  for (v = 1; v <= n; v++) {
    List list = neighbors(&G, v);
    printf("Cac dinh ke cua %d: [", v);
    for (y = 1; y <= list.size; y++)
      printf("%d ", element_at(&list, y));
    printf("]\n");
  }
  return 0;
}
```

1.3.10 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.

Trong phần cài đặt hàm neighbors ta đã sử dụng một CTDL List do ta 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.

```
#include <stdio.h>
/* Hãy thêm 2 dòng bên dưới để sử dụng CTDL vector */
#include <vector>
using namespace std;

/* Dinh nghia CTDL do thi Graph */
...

/* Tim cac dinh ke cua dinh x */
vector<int> neighbors(Graph* G, int x) {
  Graph G;
  vector<int> list;

  for (int y = 1; i <= G->n; y++)
    if (adjacent(G, x, y))
      list.push_back(y);
  return list;
}

/* Ham main() */
int main() {
  Graph G;
  /* Biểu diễn đồ thị */

  ...

  /* Liệt kê danh sách kề của các đỉnh */
  for (v = 1; v <= n; v++) {
    vector<int> list = neighbors(&G, v);
    printf("Cac dinh ke cua %d: [", v);
    for (int y = 0; y < list.size(); y++)
      printf("%d ", list[y]);
    printf("]\n");
  }
  return 0;
}
```

Một số lưu ý trong chương trình:

- vector<int> là một CTDL danh sách dùng để lưu các số nguyên.
- Hàm push_back của vector<int> (ví dụ: list.push_back(y)) cho phép thêm 1 số nguyên vào cuối danh sách.
- Hàm size() của vector<int> (ví du: list.size()) trả về số phần tử trong danh sách.
- Phép toán [y] (ví dụ: list[y]) dùng để lấy phần tử tại vị trí thứ i trong danh sách. Chú ý: phần tử đầu tiên trong danh sách có vị trí 0. Vì thế ta cho y chạy từ 0 đến < list.size().
- Để có thể sử dụng được CTDL vector, bạn cần phải thêm 2 dòng: #include <vector> và using namespace std; vào đầu chương trình.

- C++ cho phép bạn khai báo biến bên trong vòng for (ví dụ: for (int y = 0; ...)), rất tiện lợi.

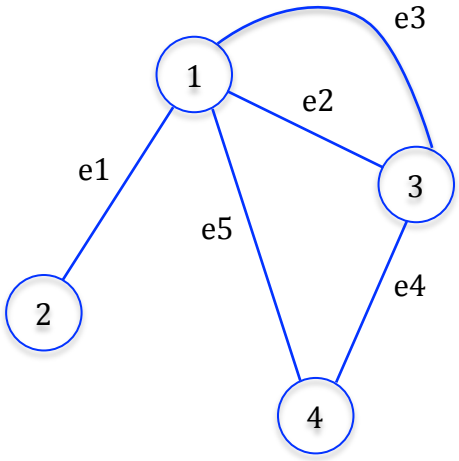
Chương trình đơn giản và dễ viết hơn phải không ?

1.4 Sử dụng 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 danh sách đỉnh kề của tất cả các đỉnh trong đồ thị hay 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[]

```
#define MAX_VERTICES 100

typedef struct {
  int n; /* n: so dinh */
  /* mang danh sach cac dinh ke */
  List adj[MAX_ VERTICES];
} Graph;
```

1.4.2 Khởi tạo đồ thị n đỉnh

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

```
/* Khoi tao do thi G co n dinh */
void init_graph(Graph* G, int n) {
  int i, j;
  G->n = n;
  for (i = 1; i <= n; i++)
    make_null(&G->adj[i]);
}
```

Chú ý: phép toán này chỉ khởi tạo đồ thị để DỰ TRÙ chứa các 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(G, x, y).

1.4.3 Thêm cung e = (x, y) vào đồ thị

Đối với đồ thị vô hướng:

- Cho đỉnh y kề với đỉnh x: thêm y vào danh sách kề của x (adj[x]).
- Cho đỉnh x kề với đinht y: thêm x vào danh sách kề của y (adj[y]).

Đối với đồ thị có hướng:

- Cho đỉnh y kề với đỉnh x: thêm y vào danh sách kề của x (adj[x]).

```
/* Them cung e = (x, y) vao do thi G */
void add_edge(Graph* G, int x, int y) {
  push_back(&G->adj[x], y); //y ke với x
  push_back(&G->adj[y], x); //x ke voi y
}
```

Chú ý:

- **Nếu G là đồ thị có hướng**, ta không thêm x vào danh sách kề của y.

1.4.4 Kiểm tra đỉnh y có kề với đỉnh x không

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

```
/* Kiem tra y co ke voi x khong */
int adjacent(Graph* G, int x, int y) {
  int j;
  for (j = 1; j <= G->adj[x].size; j++)
    if (element_at(&G->adj[x], j) == y)
      return 1;
  return 0;
}
```

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 x.

```
/* Tinh bac cua dinh x: deg(x) */
int degree(Graph* G, int x) {
  return G->adj[x].size;
}
```

1.4.6 Bài tập 7

Viết chương trình đọc đồ thị từ tập tin như bài 5 (hoặc cũng có thể tạo đồ thị trong chương trình như bài 4). Liệt kê bậc của các đỉnh

1.4.7 Bài tập 8

Viết chương trình đọc đồ thị từ tập tin như bài 5 (hoặc cũng có thể tạo đồ thị trong chương trình như bài 4). Liệt kê danh sách các đỉnh kề của tất cả các đỉnh.

1.4.8 Bài tập 9 (nâng cao)

Làm lại bài tập 7 và 8 sử dụng cấu trúc dữ liệu vector<int> của STL.

1.5 Duyệt đồ thị

Viếng thăm 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.

Duyệt đồ thị cũng có thể dùng để đếm số thành phần liên thông của một **đồ thị vô hướng**.

Phương pháp duyệt đồ thị tổng quát:

- Sử dụng một danh sách frontier lưu các đỉnh chuẩn bị duyệt.
- Đưa một đỉnh bất kỳ vào frontier, đánh dấu nó đã xét.
- while (frontier chưa rỗng)
    - lấy một đỉnh trong frontier ra, làm gì đó với nó (vd: in nó ra màn hình)
    - for (các đỉnh kề của nó) nếu nó chưa xét thì đưa vào frontier và đánh dấu đã xét nó.

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

- frontier là Ngăn xếp (stack): duyệt theo chiều sâu
- frontier là Hàng đợi (queue): duyệt theo chiều rộng.

1.5.1 Duyệt theo chiều sâu

Ta sử dụng một ngăn xếp để lưu trữ frontier và một mảng mark[] để kiểm tra xem một đỉnh đã được xét chưa, khởi tạo tất cả các đỉnh đều chưa xét. Bạn cần phải định nghĩa một cấu trúc dữ liệu Ngăn xếp (Stack) với các phép toán:

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

```
/* Khai bao Stack*/

...

/* Duyet do thi theo chieu sau */
void depth_first_search(Graph* G) {
  Stack frontier;
  int mark[MAX_VERTEXES];
  make_null_stack(&frontier);

  /* Khởi tạo mark, chưa đỉnh nào được xét */
  int j;
  for (j = 1; j <= G->n; j++)
    mark[j] = 0;

  /* Đưa 1 vào frontier */
  push(&frontier, 1);
  mark[1] = 1;

  /* Vòng lặp chính dùng để duyệt */
  while (!empty(&frontier)) {
    /* Lấy phần tử đầu tiên trong frontier ra */
    int x = top(&frontier); pop(&frontier);
    printf("Duyet %d\n", x);
    /* Lấy các đỉnh kề của nó */
    List list = neighbors(G, x);
    /* Xét các đỉnh kề của nó */
    for (j = 1; j <= list.size; j++) {
      int y = element_at(&list, j);
      if (mark[y] == 0) {
        mark[y] = 1;
        push(&frontier, y);
      }
    }
  }
}
```

Bạn có thể sử dụng khai báo ngăn xếp này:

```
/* Khai bao Stack*/
#define MAX_ELEMENTS 100
typedef struct {
  int data[MAX_ELEMENTS];
  int size;
} Stack;

void make_null_stack(Stack* S) {
  S->size = 0;
}

void push(Stack* S, int x) {
  S->data[S->size] = x;
  S->size++;
}

int top(Stack* S) {
  return S->data[S->size - 1];
}

void pop(Stack* S) {
  S->size--;
}

int empty(Stack* S) {
  return S->size == 0;
}
```

1.5.2 Bài tập 10

Viết chương trình đọc đồ thị từ tập tin như bài 5 (hoặc cũng có thể tạo đồ thị trong chương trình như bài 4). Duyệt đồ thị theo chiều sâu.

1.5.3 Duyệt theo chiều rộng

Tương tự như duyệt theo chiều sâu, ta cần một hàng đợi để lưu trữ frontier.

```
/* Khai bao Queue */

...

/* Duyệt đồ thị theo chiều rộng */
void breath_first_search(Graph* G) {
  Queue frontier;
  int mark[MAX_VERTEXES];
  make_null_queue(&frontier);

  /* Khởi tạo mark, chưa đỉnh nào được xét */
  int j;
  for (j = 1; j <= G->n; j++)
    mark[j] = 0;

  /* Đưa 1 vào frontier */
  push(&frontier, 1);
  mark[1] = 1;

  /* Vòng lặp chính dùng để duyệt */
  while (!empty(&frontier)) {
    /* Lấy phần tử đầu tiên trong frontier ra */
    int x = top(&frontier); pop(&frontier);
    printf("Duyet %d\n", x);
    /* Lấy các đỉnh kề của nó */
    List list = neighbors(G, x);
    /* Xét các đỉnh kề của nó */
    for (j = 1; j <= list.size; j++) {
      int y = element_at(&list, j);
      if (mark[y] == 0) {
        mark[y] = 1;
        push(&frontier, y);
      }
    }
  }
}
```

Bạn có thấy phần duyệt đồ thị gần như không thay đổi gì cả ngoại trừ kiểu của frontier.

Cài đặt Queue có thể như thế này.

```
/* Khai bao Queue */
#define MAX_ELEMENTS 100
typedef struct {
  int data[MAX_ELEMENTS];
  int front, rear;
} Queue;

void make_null_queue(Queue* Q) {
  Q->front = 0;
  Q->rear = -1;
}

void push(Queue* Q, int x) {
  Q->rear++;
  Q->data[Q->rear] = x;
}

int top(Queue* Q) {
  return Q->data[Q->front];
}

void pop(Queue* Q) {
  Q->front++;
}

int empty(Queue* Q) {
  return Q->front > Q->rear;
}
```

1.5.4 Bài tập 11

Viết chương trình đọc đồ thị từ tập tin như bài 5 (hoặc cũng có thể tạo đồ thị trong chương trình như bài 4). Duyệt đồ thị theo chiều rộng.

1.5.5 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 frontier. Cách này cài đặt nhanh hơn phương pháp sử dụng stack và thường được sử dụng.

```
/* Biến hỗ trợ */
  int mark[MAX_VERTEXES];

/* Duyệt đệ quy đỉnh x */
void traversal(Graph* G, int x) {
  /* Nếu đỉnh x đã duyệt, không làm gì cả */
  if (mark[x] == 1)
    return;
  /* Ngược lại, duyệt nó */
  printf("Duyet %d\n", x);

  /* Lấy các đỉnh kề của nó và duyệt các đỉnh kề */
  List list = neighbors(G, x);
  int j;
  for (j = 1; j <= list.size; j++) {
    int y = element_at(&list, j);
    traversal(G, y);
  }
}

void depth_first_search(Graph* G) {
  /* Khởi tạo mark, chưa đỉnh nào được xét */
  int j;
  for (j = 1; j <= G->n; j++)
    mark[j] = 0;
  traversal(G, 1);
}
```

Bạn có thấy, phương pháp này đơn giản hơn không ?

1.5.6 Bài tập 12

Viết chương trình đọc đồ thị từ tập tin như bài 5 (hoặc cũng có thể tạo đồ thị trong chương trình như bài 4). Duyệt đồ thị theo chiều sâu bằng phương pháp đệ quy.

1.5.7 Bài tập 13

Áp dụng phương pháp duyệt đồ thị để kiểm tra tính liên thông của đồ thị vô hướng. 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.

Viết chương trình đọc đồ thị từ tập tin như bài 5 (hoặc cũng có thể tạo đồ thị trong chương trình như bài 4). Kiểm tra tính liên thông của đồ thị và in kết quả ra màn hình. Nếu liên thông in ra: “Yes”, ngược lại in ra “No”.

1.5.8 Bài tập 14

Áp dụng phương pháp duyệt đồ thị, ta cũng có thể đếm được số thành phần liên thông của đồ thị vô hướng. Ta bắt đầu từ một đỉnh, gọi hàm để duyệt nó. Tăng số thành phần liên thông lên 1. Nếu tất cả các đỉnh đều được duyệt => kết thúc, ngược lại tìm đỉnh chưa được duyệt và duyệt đó, tăng số thành phần liên thông lên 1, và cứ như thế.

Khung chương trình có dạng:

```
/* Biến hỗ trợ */
  int mark[MAX_VERTEXES];

/* Duyệt đệ quy đỉnh x */
void traversal(Graph* G, int x) {
  /* Nếu đỉnh x đã duyệt, không làm gì cả */
  if (mark[x] == 1)
    return;
  /* Ngược lại, duyệt nó */
  /* Lấy các đỉnh kề của nó và duyệt các đỉnh kề */
  List list = neighbors(G, x);
  int j;
  for (j = 1; j <= list.size; j++) {
    int y = element_at(&list, j);
    traversal(G, y);
  }
}

/* Đếm số thành phần liên thông của đồ thị */
int count_connected_components(Graph* G) {
  /* Khởi tạo mark, chưa đỉnh nào được duyệt */
  int j;
  for (j = 1; j <= G->n; j++)
    mark[j] = 0;
  int cnt = 0;
  for (j = 1; j <= G->n; j++)
    /* Nếu đỉnh j chưa được duyệt, duyệt nó */
    if (mark[j] == 0) {
      traversal(G, j);
      cnt++;
    }
  return cnt;
}
```

Viết chương trình đọc đồ thị từ tập tin như bài 5 (hoặc cũng có thể tạo đồ thị trong chương trình như bài 4). In ra số thành phần liên thông của đồ thị.

1.5.9 Bài tập 15 (nâng cao)

Dựa trên bài tập 14, viết chương trình in ra số đỉnh của mỗi thành phần liên thông.

1.5.10 Bài tập 16 (nâng cao)

Làm lại các bài tập duyệt đồ thị theo chiều rộng và chiều sâu sử dụng stack<int> và queue<int> của STL.

1.6 Bài tập 17 – Kiểm tra đồ thị phân đôi (nâng cao)

Ta có thể sử dụng phương pháp biểu diễn đồ thị nào cũng được. Để đơn giản có thể sử dụng phương pháp biểu diễn bằng ma trận kề.

Để kiểm tra tính phân đôi của đồ thị ta sẽ tô màu các đỉnh của đồ thị bằng hai màu: đen (0) và trắng (1) sao cho hai đỉnh kề nhau sẽ có màu khác nhau. Nếu tô màu được, thì đồ thị là đồ thị phân đôi, ngược lại đồ thị không thể phân đôi.

Ta cũng sẽ dựa trên phương pháp duyệt đệ quy để tô màu.

```
/* Một số biến hỗ trợ */
int color[MAX_VERTEXES];
int fail;

/* Tô màu đỉnh bằng phương pháp đệ quy */
void colorize(Graph* G, int x, int c) {
  /* Nếu đỉnh x đã chưa có màu => tô nó */
  if (color[x] == -1) {
    color[x] = c;
    /* Lấy các đỉnh kề và tô màu các đỉnh kề bằng màu
       ngược với c */
    List list = neighbors(G, x);
    int j;
    for (j = 1; j <= list.size; j++) {
      int y = element_at(&list, j);
      colorize(G, y, !c);
  } else /* x đa có màu */
    if (color[x] != c) /* 1 đỉnh bị tô 2 màu khác nhau */
      fail = 1; /*thất bại*/
}

/* Kiểm tra đồ thị có là đồ thị phân đôi */
int is_bigraph(Graph* G) {
  /* Khởi tạo color, chưa đỉnh nào có màu */
  int j;
  for (j = 1; j <= G->n; j++)
    color[j] = -1;
  fail = 0;
  colorize(G, 1, 0); /* Tô màu đỉnh 1 bằng màu đen */
  /* Nếu không thất bại, G là bigraph */
  return !fail;
}
```

Viết chương trình đọc đồ thị từ tập tin như bài 5 (hoặc cũng có thể tạo đồ thị trong chương trình như bài 4). Kiểm tra xem nó có phải là đồ thị phân đôi hay không và in kết quả ra màn hình. Nếu phải in ra: “Yes”, ngược lại in ra “No”.

1.7 Bài tập 18 – Phân chia đội bóng

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:

Ví dụ 1:

```
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.

Chú ý: thành viên a đã từng thi đấu chung với thành viên b và thành viên b đã từng thi đấu chung với thành viên c **KHÔNG CÓ NGHĨA LÀ** thành viên a đã từng thi đấu với thành viên c.

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

Nếu phân chi không được, in ra IMPOSSIBLE.

Gợi ý:

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

Ví dụ 2:

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

Ví dụ 3:

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