BUỔI 2. TÍNH LIÊN THÔNG CỦA ĐỒ THỊ

Mục đích:

- Duyệt đồ thị
- Kiểm tra tính liên thông của đồ thị vô hướng
- Đếm số thành phần liên thông của đồ thị vô hướng
- Kiểm tra đồ thị phân đôi
- Kiểm tra đồ thị có chứa chu trình hay không
- Kiểm tra tính liên thông của đồ thị có hướng
- Cài đặt giải thuật Tarjan để tìm các thành phần liên thông của đồ thị có hướng

Yêu cầu:

- Biết biểu diễn đồ thị trên máy tính

2.1 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 từ 1 đỉnh x bất kỳ:

- Sử dụng một danh sách L lưu các *đỉnh chuẩn bị duyệt*.
- Đưa một đỉnh bất kỳ (đỉnh x) vào L.
- while (L chưa rỗng)
    - lấy một đỉnh trong L ra, làm gì đó với nó (ví dụ: in nó ra màn hình), và *đánh dấu nó đã được duyệt*.
    - duyệt qua từng đỉnh kề của nó và đưa vào L.

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

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

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

2.1.1 Duyệt theo chiều sâu

Ta sử dụng một ngăn xếp để lưu trữ L và một mảng mark[] để kiểm tra xem một đỉnh đã được duyệt chưa, khởi tạo tất cả các đỉnh đều chưa duyệt. 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 L;
  int mark[MAX_VERTEXES];
  make_null_stack(&L);

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

  /* Đưa 1 vào L, bắt đầu duyệt từ đỉnh 1 */
  push(&L, 1);

  /* Vòng lặp chính dùng để duyệt */
  while (!empty(&L)) {
    /* Lấy phần tử đầu tiên trong L ra */
    int x = top(&L); pop(&L);
    if (marl[x] != 0) // Đã duyệt rồi, bỏ qua
      continue;
    printf("Duyet %d\n", x);
    mark[x] = 1; //Đánh dấu nó đã duyệt
    /* 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);
      push(&L, y);
    }
  }
}
```

Bạn có thể sử dụng khai báo ngăn xếp như thế 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;
}
```

2.1.2 Bài tập 2.1

Viết chương trình đọc đồ thị từ tập tin và duyệt đồ thị theo chiều sâu.

2.1.3 Duyệt theo chiều rộng

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

```
/* Khai bao Queue */

...

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

  /* 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);

  /* Vòng lặp chính dùng để duyệt */
  while (!empty(&L)) {
    /* Lấy phần tử đầu tiên trong L ra */
    int x = top(&L); pop(&L);
    if (marl[x] != 0) // Đã duyệt rồi, bỏ qua
      continue;
    printf("Duyet %d\n", x);
    mark[x] = 1; //Đánh dấu nó đã duyệt
    /* 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);
      push(&L, y);
    }
  }
}
```

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

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;
}
```

2.1.4 Bài tập 2.2

Viết chương trình đọc đồ thị từ tập tin và duyệt đồ thị theo chiều rộng. Quan sát thứ tự các đỉnh được in ra màn hình, so sánh với bài tập 2.1 (thứ tự có giống nhau không, giải thích tại sao ?).

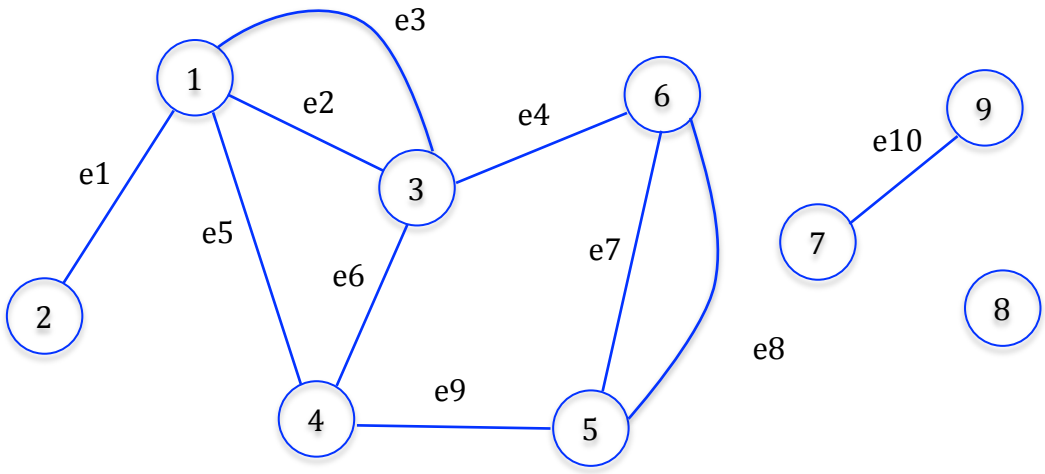
2.1.5 Bài tập 2.3

Cần phải chú ý rằng cả 2 phép duyệt này (rộng và sâu) chỉ có thể duyệt được một bộ phận liên thông (chứa đỉnh 1) của đồ thị. Để duyệt toàn bộ các đỉnh của đồ thị ta cần phải:

- Bổ sung thêm một tham số x vào hàm duyệt:
  - `depth_first_search(Graph* G, int x)` hoặc
  - `breath_first_search(Graph* G, int x)`
- Gọi lại hàm duyệt này nhiều lần, mỗi lần duyệt một phận liên thông

```
for (j = 1; j <= G->n; j++)
  /* Nếu đỉnh j chưa được duyệt, duyệt nó */
  if (mark[j] == 0)
    depth_first_search(G, j);
```

Làm lại bài tập 2.1 và 2.2 theo cách này. Sử dụng đồ thị bên dưới để kiểm tra.



2.1.6 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 L. Cách này cài đặt nhanh hơn phương pháp sử dụng stack và thường được sử dụng. Biến hỗ trợ mark phải được khai báo bên ngoài (toàn cục hoặc là tham số của hàm traversal)

```
/* 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);
  mark[x] = 1; // Đánh dấu đã 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);
  }
}

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 ?

2.1.7 Bài tập 2.4

Làm lại bài tập 2.1, 2.2 và 2.3 bằng phương pháp duyệt đệ quy. So sánh kết quả của phương pháp duyệt theo chiều sâu và duyệt đệ quy. Kết quả có giống nhau không ? Nếu khác nhau, làm thế nào để cả hai cho kết quả giống nhau ?

2.2 Tính liên thông của đồ thị vô hướng

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.

2.2.1 Bài tập 2.5

Viết chương trình đọc đồ thị từ tập tin, 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”.

2.2.2 Bài tập 2.6

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 và in ra số thành phần liên thông của đồ thị.

2.2.3 Bài tập 2.7 (nâng cao)

Dựa trên bài tập 2.6, viết chương trình in ra số đỉnh của mỗi thành phần liên thông. Gợi ý: trước khi duyệt 1 bộ phận liên thông ta khởi động biến đếm = 0, mỗi lần duyệt 1 đỉnh ta tăng biến đếm lên 1. Sau khi duyệt xong 1 bộ phận liên thông biến đếm này cho biết số đỉnh của thành phần liên thông vừa duyệt.

2.2.4 Bài tập 2.8 (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.

2.3 Bài tập 2.9 – 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, 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”.

2.4 Bài tập 2.10 – Phân chia đội bóng (ứng dụng)

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

Dữ liệu đầu vào có dạng:

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
```

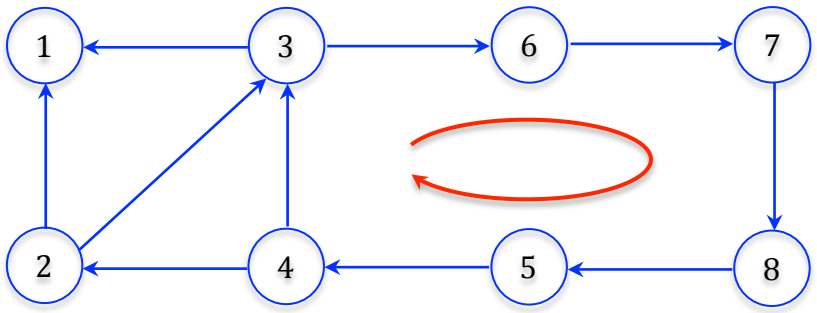
2.5 Kiểm tra đồ thị có chứa chu trình

Chu trình là một đường đi đơn đỉnh (path) có đỉnh đầu trùng đỉnh cuối.
Cho đồ thị G = <V, E>, làm thế nào để kiểm tra G có chứa chu trình hay không ?
Ý tưởng: sử dụng phương pháp duyệt đồ thị theo chiều sâu (cài đặt bằng đệ quy)

- Ta sẽ tô màu các đỉnh theo trạng thái của nó
  - Chưa được duyệt (trắng)
  - Đang duyệt, duyệt chưa xong (xám): đỉnh chưa duyệt xong sẽ nằm trên stack.
  - Đã duyệt xong (đen)
- Đầu tiên các đỉnh đều có màu trắng
- Để duyệt đỉnh đỉnh x, ta tô nó bằng màu xám (đang duyệt) và
- Xét các đỉnh kề với nó
  - Nếu đỉnh kề y nào đó có màu xám, có nghĩa là ta đã tìm được 1 chu trình: vì từ y -> … -> x -> y.
  - Nếu y có màu trắng => duyệt y
  - Nếu y có màu đen => duyệt xong rồi, bỏ qua

Cài đặt kiểm tra chu trình gồm hai phần: duyệt theo chiều sâu bắt đầu từ 1 đỉnh bằng cách tô màu các đỉnh đã đi qua. Nếu một đỉnh màu xám được xét lại => có chu trình.

Để kiểm tra toàn bộ đồ thị, trước hết ta khởi tạo các đỉnh đều có màu trắng. Lần lượt xét từng đỉnh nếu nó có màu trắng => gọi hàm duyệt để duyệt nó.



```
/* Biến hỗ trợ */
#define white 0
#define black 1
#define gray 2
#define MAX_VERTICES 1000

int color[MAX_VERTICES];
int cycle;

/* Duyệt đồ thị bắt đầu từ đỉnh x */
void dfs(Graph* G, int x) {
  color[x] = gray;
  int j;
  List list = neighbors(G, x);
  int j;
  for (j = 1; j <= list.size; j++) {
    int y = element_at(&list, j);
    if (color[y] == gray) {
      cycle = 1; /* Tồn tại chu trình */
      return;
    }
    if (color[y] == white)
      dfs(v);
  }
  color[x] = black;
}

/* Kiểm tra toàn bộ đồ thị*/
int contains_cycle(Graph* G) {
  int j;
  for (j = 1; j <= G->n; j++) {
    color[i] = white;
  }
  cycle = 0;
  for (j = 1; j <= G->n; j++) {
    if (color[j] == white)
      dfs(G, j);
  return cycle;
}
```

2.5.1 Bài tập 2.11 (đồ thị có hướng)

Viết chương trình đọc một đồ thị có hướng từ file và kiểm tra xem đồ thị đó có chứa chu trình không. Nếu có, ghi ra "Yes", ngược lại ghi ra "No".

Chú ý: bản cài đặt trên chỉ dùng để kiểm tra đồ thị CÓ HƯỚNG. Để kiểm tra đồ thị VÔ HƯỚNG, ta điều chỉnh 1 chút ở hàm duyệt: kiểm tra xem đỉnh kề của 1 đỉnh có phải là đỉnh cha của đỉnh đang xét không.

```
/* Duyệt đồ thị bắt đầu từ đỉnh x */
void dfs(Graph* G, int x, int parent) {
  color[x] = gray;
  int j;
  List list = neighbors(G, x);
  int j;
  for (j = 1; j <= list.size; j++) {
    int y = element_at(&list, j);
    if (y == parent) continue;
    if (color[y] == gray) {
      cycle = 1; /* Tồn tại chu trình */
      return;
    }
    if (color[y] == white)
      dfs(v);
  }
  color[x] = black;
}

/* Kiểm tra toàn bộ đồ thị*/
int contains_cycle(Graph* G) {
  int j;
  for (j = 1; j <= G->n; j++) {
    color[i] = white;
  }
  cycle = 0;
  for (j = 1; j <= G->n; j++) {
    if (color[j] == white)
      dfs(G, j, 0);
  return cycle;
}
```

Ta kiểm tra y xem nó có phải là đỉnh cha của x (đỉnh vừa mới duyệt trước khi duyệt x) không, nếu phải thì bỏ qua. Để gọi hàm dfs ta truyền thêm cho nó 1 đối số 0, có nghĩa là đỉnh j là đỉnh bắt đầu, không có cha.

Nếu không có đoạn kiểm tra này giải thuật sẽ nói rằng tất cả các đồ thị vô hướng đều có CHU TRÌNH.

2.5.2 Bài tập 2.12 (đồ thị vô hướng)

Viết chương trình đọc một đồ thị vô hướng từ file và kiểm tra xem đồ thị đó có chứa chu trình không. Nếu có, ghi ra "Yes", ngược lại ghi ra "No".

2.5.3 Bài tập 2.13 (ứng dụng)

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

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

Ví dụ 1:
soda wine
water wine

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

Ví dụ 2:
soda wine
water wine
wine water

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

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

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

Có loại thức uống (soda: 1, wine: 2 và water: 3) và có 2 điều kiện tiên quyết
1 -> 2 và 3 -> 2.
Với ví dụ 2, ta có tập tin:

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

Rõ ràng bài toán này có thể mô hình về dạng *đồ thị có hướng*: *đỉnh tương ứng với các thức uống* và mỗi *điều kiện tiên quyết ứng với 1 cung*.
Để có thể uống hết tất cả các thức uống thì **đồ thị này phải không chứa chu trình**.

Viết chương trình đọc một tập tin mô tả thức uống và các đồ thị tiên quyết như trên, và in ra màn hình "YES" nếu Haddock có thể say, ngược lại in ra "NO".
Có thể kiểm thử với danh sách các thức uống sau:

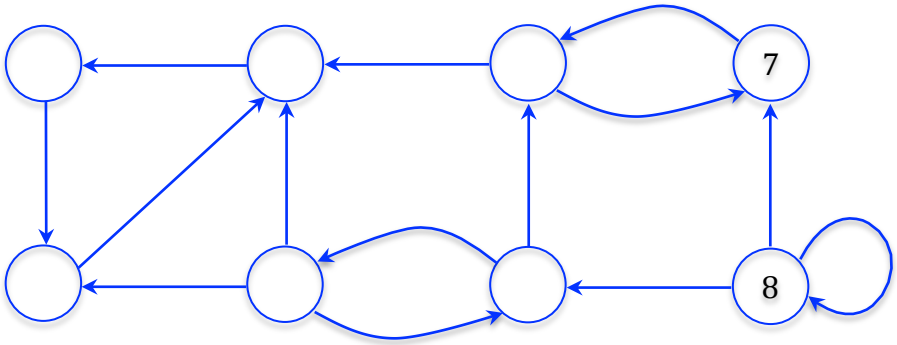
```
5 3
1 2
2 3
3 4
4 2
5 4
```

2.6 Tính liên thông của đồ thị có hướng

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

Đồ thị có hướng G là đồ thị liên thông yếu nếu như đồ thị vô hướng nền của nó liên thông. Xem bài thực hành 1 về cách kiểm tra đồ thị vô hướng liên thông.

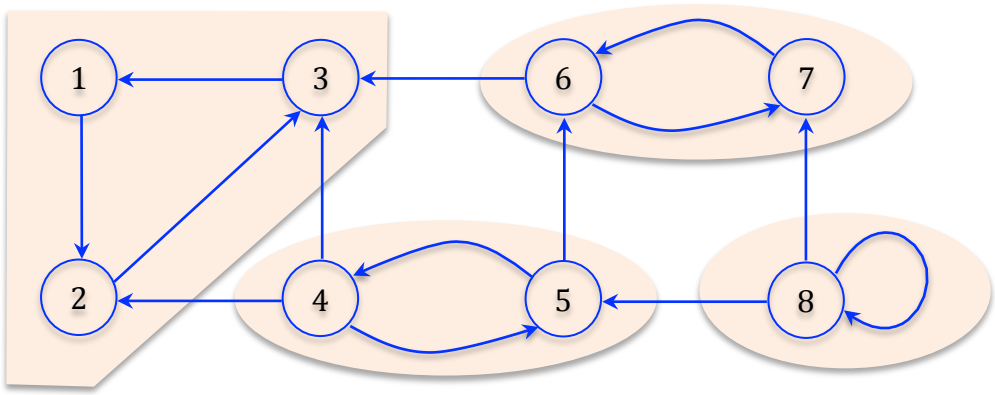
Ví dụ:



Tập tin biểu diễn đồ thị trên: 8 đỉnh, 14 (1 khuyên).

```
8 14
1 2
2 3
3 1
4 2
4 3
4 5
5 4
5 6
6 3
6 7
7 6
8 5
8 7
8 8
```

Rõ ràng đồ thị này không liên thông mạnh. Nó gồm 4 thành phần liên thông mạnh:



2.7 Giải thuật Tarjan

Giải thuật Tarjan (Robert Tarjan, 1972) sử dụng chiến lược duyệt đồ thị theo chiều sâu để xác định các thành phần liên thông mạnh của đồ thị có hướng. Ta cần một số biến hỗ trợ cho giải thuật:

- num[v]: lưu chỉ số của đỉnh v trong quá trình duyệt
- min_num[v]: lưu chỉ số nhỏ nhất của nút có thể đi đến được từ v (trong quá trình duyệt theo chiều sâu).
- S: ngăn xếp, lưu các đỉnh theo thứ tự được duyệt
- on_stack[v]: v có đang ở trong stack S hay không

Giải thuật bắt đầu từ 1 đỉnh x, gán chỉ số cho nó và đưa nó vào stack. Sau đó xét các đỉnh kề của nó. Nếu đỉnh kề chưa được duyệt => gọi đệ quy để duyệt nó và cập nhật lại min_num[x] cho x. Nếu đỉnh kề đang nằm trên stack (ta tìm được 1 vòng lặp), cập nhật lại min_num[x] cho x. Ngược lại không cần làm gì cả.

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

```
/* Duyệt đồ thị bắt đầu từ đỉnh x */
void strong_connect(Graph* G, int x) {
  num[x] = min_num[x] = k; k++;
  push(&S, x);
  on_stack[x] = 1;

  List list = neighbors(G, x);
  int j;
  /* Xét các đỉnh kề của nó */
  for (j = 1; j <= list.size; j++) {
    int y = element_at(&list, j);
    if (num[y] < 0) {
      strong_connect(G, y);
      min_num[x] = min(min_num[x], min_num[y]);
    } else if (on_stack[y])
      min_num[x] = min(min_num[x], num[y]);
  }
  printf("min_num[%d] = %d\n", x, min_num[x]);

  if (num[x] == min_num[x]) {
    printf("%d la dinh khop.\n", x);
    int w;
    do {
      w = top(&S);
      pop(&S);
      on_stack[w] = 0;
    } while (w != x);
  }
}
```

Hàm **strong_connect** cho phép tìm 1 hoặc một số thành phần liên thông mạnh bắt đầu từ đỉnh x. Ta có thể gọi hàm này nhiều lần để tất cả các thành phần liên thông mạnh của đồ thị.

```
for (v = 1; v <= n; v++) {
    num[v] = -1;
    on_stack[v] = 0;
}
k = 1;
make_null_stack(&S);

for (v = 1; v <= n; v++) {
    if (num[v] == -1)
      strong_connect(&G, v);
```

2.8 Bài tập 2.14

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

2.9 Bài tập 2.15

Viết chương trình đọc đồ thị và kiểm tra xem nó có liên thông mạnh hay không. Nếu có in ra "Yes", ngược lại in ra "No".

Gợi ý: *trong quá trình duyệt đồ thị, nếu số đỉnh lấy ra từ stack bằng với n thì đồ thị liên thông mạnh.*

2.10 Bài tập 2.16

Viết chương trình đọc đồ thị, in ra số lượng thành phần liên thông mạnh của đồ thị.

Gợi ý: *mỗi khi tìm được một thành phần liên thông (if (num[x] == min_num[x])) tăng biến đếm lên 1.*

2.11 Bài tập 2.17 (nâng cao)

Viết chương trình đọc đồ thị, liệt kê các đỉnh trong từng thành phần liên thông mạnh.

Gợi ý: *mỗi khi tìm được một thành phần liên thông (if (num[x] == min_num[x])), in các đỉnh đang nằm trên stack ra màn hình (biến w).*

2.12 Bài tập 2.18 – Come and Go (ứng dụng)

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

Hãy viết chương trình kiểm tra xem các con đường của thành phố có thoả mãn điều kiện này không. Nếu có in ra 1, ngược lại in ra 0.

(nguồn: UVA Online Judge, Problem 11838)

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

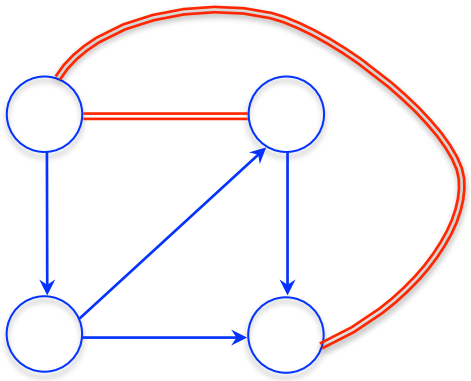
Ví dụ 1:

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

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

Gợi ý:

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



Ví dụ 2:

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

kết quả: 1

Ví dụ 3:

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

kết quả: 0

Ví dụ 4:

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

kết quả: 0