ĐỒ THỊ

**1. Đồ thị (Graph)**

* **Định nghĩa**: Một đồ thị là một tập hợp các đỉnh (nodes hoặc vertices) và các cạnh (edges) kết nối các đỉnh. Đồ thị có thể được dùng để biểu diễn các mối quan hệ hoặc kết nối giữa các đối tượng.

**2. Các khái niệm cơ bản**

* **Đỉnh (Vertex)**: Một điểm trong đồ thị, biểu thị một thực thể.
* **Cạnh (Edge)**: Một liên kết giữa hai đỉnh, biểu thị một mối quan hệ hoặc kết nối. Có thể là cạnh có hướng (directed) hoặc không có hướng (undirected).

**Loại đồ thị**

* **Đồ thị không hướng (Undirected Graph)**: Cạnh không có hướng, nghĩa là nếu có cạnh nối từ đỉnh A đến đỉnh B, thì cũng có cạnh nối từ B đến A.
* **Đồ thị có hướng (Directed Graph)**: Cạnh có hướng, nghĩa là mỗi cạnh có một hướng từ một đỉnh đến đỉnh khác. Cạnh từ đỉnh A đến đỉnh B không đồng nghĩa với cạnh từ B đến A.
* **Đồ thị có trọng số (Weighted Graph)**: Mỗi cạnh có một trọng số (hoặc giá trị), thường dùng để biểu thị chi phí, khoảng cách, hoặc bất kỳ giá trị nào khác liên quan đến cạnh.
* **Đồ thị vô hướng có trọng số (Undirected Weighted Graph)**: Đồ thị vô hướng với mỗi cạnh có một trọng số.
* **Đồ thị có hướng có trọng số (Directed Weighted Graph)**: Đồ thị có hướng với mỗi cạnh có một trọng số.

**3. Đặc điểm và thuộc tính**

* **Bậc (Degree)**: Số lượng cạnh kề với một đỉnh. Trong đồ thị có hướng, có thể phân biệt giữa bậc vào (in-degree) và bậc ra (out-degree).
* **Đường đi (Path)**: Một chuỗi các đỉnh liên tiếp nhau, trong đó mỗi cặp đỉnh kề nhau bằng một cạnh.
* **Chu trình (Cycle)**: Một đường đi mà điểm đầu và điểm cuối là cùng một đỉnh.
* **Kết nối (Connectivity)**: Đồ thị là liên thông nếu có thể đi từ bất kỳ đỉnh nào đến bất kỳ đỉnh nào khác thông qua các cạnh.
* **Thành phần liên thông (Connected Component)**: Một tập hợp con của các đỉnh mà giữa bất kỳ hai đỉnh nào trong tập hợp có thể đi đến nhau qua các cạnh trong đồ thị.
* **Đường đi ngắn nhất (Shortest Path)**: Đường đi từ đỉnh nguồn đến đỉnh đích với tổng trọng số nhỏ nhất. Thuật toán Dijkstra và Bellman-Ford thường được sử dụng để tìm đường đi ngắn nhất.

**4. Các thuật toán cơ bản**

* **BFS (Breadth-First Search)**: Duyệt qua các đỉnh theo chiều rộng, thích hợp cho tìm kiếm ngắn nhất trong đồ thị không trọng số.
* **DFS (Depth-First Search)**: Duyệt qua các đỉnh theo chiều sâu, hữu ích cho tìm kiếm, phân loại và xác định các thành phần liên thông.
* **Thuật toán Dijkstra**: Tìm đường đi ngắn nhất từ một đỉnh nguồn đến tất cả các đỉnh khác trong đồ thị có trọng số dương.
* **Thuật toán Bellman-Ford**: Tìm đường đi ngắn nhất từ một đỉnh nguồn đến tất cả các đỉnh khác trong đồ thị có trọng số, kể cả trọng số âm.
* **Thuật toán Floyd-Warshall**: Tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh trong đồ thị.

**5. Các cấu trúc dữ liệu**

* **Danh sách kề (Adjacency List)**: Lưu trữ danh sách các đỉnh kề với mỗi đỉnh. Hiệu quả về không gian cho đồ thị thưa.
* **Ma trận kề (Adjacency Matrix)**: Ma trận vuông lưu trữ thông tin về các cạnh giữa các đỉnh. Thích hợp cho đồ thị dày đặc (dense graph).

**6. Đồ thị cây (Tree)**

* **Định nghĩa**: Cây là một đồ thị liên thông không có chu trình. Mỗi cặp đỉnh trong cây đều được nối với nhau bằng một đường đi duy nhất.
* **Tính chất**: Có n - 1 cạnh với n đỉnh.

Các khái niệm và thuật toán cơ bản này là nền tảng cho lý thuyết đồ thị và rất quan trọng trong việc giải quyết các bài toán liên quan đến đồ thị trong khoa học máy tính và toán học.

**Thuật toán BFS (Breadth-First Search)**

**BFS** (Breadth-First Search) là một thuật toán tìm kiếm và duyệt qua các đỉnh trong một đồ thị theo chiều rộng, tức là từ một đỉnh gốc, nó sẽ tìm tất cả các đỉnh kề trước khi di chuyển đến các đỉnh sâu hơn. BFS được sử dụng để tìm đường đi ngắn nhất trong đồ thị không trọng số và có nhiều ứng dụng khác trong lý thuyết đồ thị.

**Nguyên lý hoạt động của BFS**

1. **Khởi tạo**:
   * Bắt đầu từ đỉnh gốc (hoặc đỉnh bắt đầu).
   * Đưa đỉnh gốc vào hàng đợi (queue) và đánh dấu nó là đã thăm.
2. **Duyệt các đỉnh**:
   * Lấy đỉnh từ hàng đợi.
   * Duyệt qua tất cả các đỉnh kề của đỉnh đó.
   * Nếu một đỉnh kề chưa được thăm, đánh dấu nó là đã thăm và thêm nó vào hàng đợi.
3. **Tiếp tục**:
   * Lặp lại bước 2 cho đến khi hàng đợi rỗng.

**Đặc điểm của BFS**

* **Chiều rộng trước**: BFS duyệt qua các đỉnh theo chiều rộng của đồ thị.
* **Khoảng cách ngắn nhất**: Trong đồ thị không trọng số, BFS đảm bảo tìm đường đi ngắn nhất từ đỉnh gốc đến các đỉnh khác.
* **Hàng đợi**: BFS sử dụng cấu trúc dữ liệu hàng đợi (queue) để theo dõi các đỉnh cần duyệt.

**Thuật toán BFS trong C++**

Dưới đây là một ví dụ về cách triển khai thuật toán BFS trong C++:

cpp

Copy code

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

const int MAXN = 1000; // Kích thước tối đa của đồ thị

vector<int> graph[MAXN];

bool visited[MAXN];

void bfs(int start) {

queue<int> q; // Khởi tạo hàng đợi

q.push(start); // Đưa đỉnh gốc vào hàng đợi

visited[start] = true; // Đánh dấu đỉnh gốc là đã thăm

while (!q.empty()) {

int u = q.front(); // Lấy đỉnh đầu hàng đợi

q.pop(); // Loại bỏ đỉnh khỏi hàng đợi

cout << u << " "; // Xử lý đỉnh u (ví dụ: in ra đỉnh u)

for (int v : graph[u]) { // Duyệt qua các đỉnh kề của đỉnh u

if (!visited[v]) { // Nếu đỉnh kề chưa được thăm

visited[v] = true; // Đánh dấu đỉnh kề là đã thăm

q.push(v); // Thêm đỉnh kề vào hàng đợi

}

}

}

}

int main() {

int n, m;

cin >> n >> m;

// Đọc đồ thị

for (int i = 0; i < m; ++i) {

int a, b;

cin >> a >> b;

graph[a].push\_back(b);

graph[b].push\_back(a); // Nếu đồ thị có hướng, chỉ thêm một hướng

}

// Khởi tạo mảng visited

fill(visited, visited + n + 1, false);

// Bắt đầu BFS từ đỉnh 1

bfs(1);

return 0;

}

**Giải thích mã**

1. **Khởi tạo**:
   * queue<int> q;: Khởi tạo hàng đợi.
   * q.push(start);: Đưa đỉnh gốc vào hàng đợi.
   * visited[start] = true;: Đánh dấu đỉnh gốc là đã thăm.
2. **Duyệt qua các đỉnh**:
   * int u = q.front();: Lấy đỉnh đầu hàng đợi.
   * q.pop();: Loại bỏ đỉnh khỏi hàng đợi.
   * for (int v : graph[u]): Duyệt qua các đỉnh kề của đỉnh u.
   * if (!visited[v]): Nếu đỉnh kề chưa được thăm.
   * visited[v] = true;: Đánh dấu đỉnh kề là đã thăm.
   * q.push(v);: Thêm đỉnh kề vào hàng đợi.
3. **Kết thúc**:
   * Khi hàng đợi rỗng, thuật toán hoàn tất.

**Ứng dụng của BFS**

* **Tìm đường đi ngắn nhất**: Trong đồ thị không trọng số.
* **Kiểm tra tính liên thông**: Để xác định nếu đồ thị là liên thông.
* **Tìm thành phần liên thông**: Trong đồ thị không liên thông.
* **Thuật toán tìm kiếm trong trò chơi**: Như tìm kiếm các nước đi trong trò chơi cờ vua.

**Ví dụ Đồ thị**

Giả sử chúng ta có đồ thị không hướng sau đây:

markdown

Copy code

1

/ \

2 3

| |

4 - 5

* Đỉnh 1 kết nối với đỉnh 2 và 3.
* Đỉnh 2 kết nối với đỉnh 4.
* Đỉnh 3 kết nối với đỉnh 5.
* Đỉnh 4 kết nối với đỉnh 5.

**Thuật toán BFS từ Đỉnh 1**

**Bước 1: Khởi tạo**

* **Danh sách kề (adjacency list)**:
  + 1: [2, 3]
  + 2: [1, 4]
  + 3: [1, 5]
  + 4: [2, 5]
  + 5: [3, 4]
* **Hàng đợi (queue)**: [1]
* **Danh sách đã thăm (visited)**: [false, true, false, false, false, false] (đỉnh 1 đã thăm)

**Bước 2: Duyệt đỉnh 1**

* Lấy đỉnh 1 từ hàng đợi.
* **Danh sách đã thăm**: [true, true, true, false, false, false] (đỉnh 1 đã thăm)
* **Danh sách kề của đỉnh 1**: [2, 3]
  + Đỉnh 2 chưa thăm, thêm vào hàng đợi.
  + Đỉnh 3 chưa thăm, thêm vào hàng đợi.
* **Hàng đợi**: [2, 3]

**Bước 3: Duyệt đỉnh 2**

* Lấy đỉnh 2 từ hàng đợi.
* **Danh sách đã thăm**: [true, true, true, true, false, false] (đỉnh 2 đã thăm)
* **Danh sách kề của đỉnh 2**: [1, 4]
  + Đỉnh 1 đã thăm.
  + Đỉnh 4 chưa thăm, thêm vào hàng đợi.
* **Hàng đợi**: [3, 4]

**Bước 4: Duyệt đỉnh 3**

* Lấy đỉnh 3 từ hàng đợi.
* **Danh sách đã thăm**: [true, true, true, true, true, false] (đỉnh 3 đã thăm)
* **Danh sách kề của đỉnh 3**: [1, 5]
  + Đỉnh 1 đã thăm.
  + Đỉnh 5 chưa thăm, thêm vào hàng đợi.
* **Hàng đợi**: [4, 5]

**Bước 5: Duyệt đỉnh 4**

* Lấy đỉnh 4 từ hàng đợi.
* **Danh sách đã thăm**: [true, true, true, true, true, true] (đỉnh 4 đã thăm)
* **Danh sách kề của đỉnh 4**: [2, 5]
  + Đỉnh 2 đã thăm.
  + Đỉnh 5 đã thăm.
* **Hàng đợi**: [5]

**Bước 6: Duyệt đỉnh 5**

* Lấy đỉnh 5 từ hàng đợi.
* **Danh sách đã thăm**: [true, true, true, true, true, true] (đỉnh 5 đã thăm)
* **Danh sách kề của đỉnh 5**: [3, 4]
  + Đỉnh 3 đã thăm.
  + Đỉnh 4 đã thăm.
* **Hàng đợi**: [] (rỗng)

**Kết quả**

* **Danh sách đã thăm** (theo thứ tự BFS): 1, 2, 3, 4, 5
* **Hàng đợi**: [] (kết thúc)

**Hình ảnh Minh Họa**

less

Copy code

Bắt đầu từ đỉnh 1:

Hàng đợi: [1]

Danh sách đã thăm: [true, false, false, false, false, false]

Duyệt đỉnh 1:

Hàng đợi: [2, 3]

Danh sách đã thăm: [true, true, true, false, false, false]

Duyệt đỉnh 2:

Hàng đợi: [3, 4]

Danh sách đã thăm: [true, true, true, true, false, false]

Duyệt đỉnh 3:

Hàng đợi: [4, 5]

Danh sách đã thăm: [true, true, true, true, true, false]

Duyệt đỉnh 4:

Hàng đợi: [5]

Danh sách đã thăm: [true, true, true, true, true, true]

Duyệt đỉnh 5:

Hàng đợi: []

Danh sách đã thăm: [true, true, true, true, true, true]

**Thuật toán DFS (Depth-First Search)**

**DFS** (Depth-First Search) là một thuật toán duyệt hoặc tìm kiếm trong một đồ thị hoặc cây. Thuật toán đi sâu vào mỗi nhánh của đồ thị càng xa càng tốt trước khi quay lại và thử các nhánh khác. DFS được sử dụng rộng rãi trong các ứng dụng như tìm thành phần liên thông, phát hiện chu trình, và tìm kiếm đường đi trong mê cung.

**Nguyên lý hoạt động của DFS**

1. **Khởi tạo**:
   * Bắt đầu từ một đỉnh gốc (hoặc đỉnh bắt đầu).
   * Đánh dấu đỉnh đó là đã thăm và xử lý đỉnh (ví dụ, in ra đỉnh đó).
2. **Duyệt đệ quy**:
   * Duyệt qua tất cả các đỉnh kề của đỉnh hiện tại.
   * Nếu đỉnh kề chưa được thăm, thực hiện DFS từ đỉnh kề đó.
3. **Quay lại**:
   * Khi không còn đỉnh kề nào để đi, thuật toán quay lại đỉnh trước đó và tiếp tục quá trình.

DFS có thể được triển khai bằng cách sử dụng đệ quy hoặc ngăn xếp (stack).

**Thuật toán DFS đệ quy trong C++**

Dưới đây là một ví dụ về cách triển khai DFS bằng đệ quy:

cpp

Copy code

#include <iostream>

#include <vector>

#include <cstring>

using namespace std;

const int MAXN = 1000; // Kích thước tối đa của đồ thị

vector<int> graph[MAXN];

bool visited[MAXN];

void dfs(int u) {

visited[u] = true; // Đánh dấu đỉnh u là đã thăm

cout << u << " "; // Xử lý đỉnh u (ví dụ: in ra đỉnh u)

for (int v : graph[u]) { // Duyệt qua tất cả các đỉnh kề của u

if (!visited[v]) { // Nếu đỉnh kề chưa được thăm

dfs(v); // Thực hiện DFS trên đỉnh kề

}

}

}

int main() {

int n, m;

cin >> n >> m;

// Đọc đồ thị

for (int i = 0; i < m; ++i) {

int a, b;

cin >> a >> b;

graph[a].push\_back(b);

graph[b].push\_back(a); // Nếu đồ thị có hướng, chỉ thêm một hướng

}

// Khởi tạo mảng visited

memset(visited, false, sizeof(visited));

// Bắt đầu DFS từ đỉnh 1

dfs(1);

return 0;

}

**Giải thích mã**

1. **Khởi tạo**:
   * vector<int> graph[MAXN];: Danh sách kề để lưu trữ đồ thị.
   * bool visited[MAXN];: Mảng visited để đánh dấu các đỉnh đã thăm.
2. **Hàm dfs(int u)**:
   * visited[u] = true;: Đánh dấu đỉnh u là đã thăm.
   * cout << u << " ";: Xử lý đỉnh u (ở đây là in ra đỉnh u).
   * for (int v : graph[u]): Duyệt qua tất cả các đỉnh kề của u.
   * if (!visited[v]): Nếu đỉnh kề v chưa được thăm, gọi đệ quy hàm dfs(v).
3. **Hàm main()**:
   * Đọc số đỉnh n và số cạnh m.
   * Đọc các cạnh của đồ thị và xây dựng danh sách kề.
   * Khởi tạo mảng visited với giá trị false.
   * Bắt đầu DFS từ đỉnh 1 bằng cách gọi dfs(1).

**Đặc điểm của DFS**

* **Chiều sâu trước**: DFS đi sâu vào các nhánh của đồ thị trước khi quay lại.
* **Đệ quy**: DFS thường được triển khai bằng cách sử dụng đệ quy, nhưng cũng có thể được triển khai bằng cách sử dụng ngăn xếp.
* **Ứng dụng**: DFS được sử dụng để tìm tất cả các đường đi, phát hiện chu trình, tìm thành phần liên thông và nhiều bài toán đồ thị khác.

**Ứng dụng của DFS**

* **Tìm thành phần liên thông**: Xác định các nhóm đỉnh trong đồ thị mà có đường đi giữa mọi cặp đỉnh.
* **Phát hiện chu trình**: Kiểm tra xem có chu trình trong đồ thị hay không.
* **Duyệt qua tất cả các đường đi**: Tìm tất cả các đường đi từ một đỉnh này đến một đỉnh khác.
* **Sắp xếp tô pô (Topological Sort)**: Trong đồ thị có hướng và không chu trình (DAG).

**Ví dụ Đồ thị**

Giả sử chúng ta có một đồ thị không hướng như sau:

markdown

Copy code

1

/ \

2 3

| |

4 - 5

* Đỉnh 1 kết nối với đỉnh 2 và 3.
* Đỉnh 2 kết nối với đỉnh 4.
* Đỉnh 3 kết nối với đỉnh 5.
* Đỉnh 4 kết nối với đỉnh 5.

**Thuật toán DFS từ Đỉnh 1**

**Bước 1: Bắt đầu từ Đỉnh 1**

* Đánh dấu đỉnh 1 là đã thăm.
* Danh sách đã thăm: [true, false, false, false, false, false]

**Bước 2: Đi sâu từ Đỉnh 1 đến Đỉnh 2**

* Đánh dấu đỉnh 2 là đã thăm.
* Danh sách đã thăm: [true, true, false, false, false, false]

**Bước 3: Đi sâu từ Đỉnh 2 đến Đỉnh 4**

* Đánh dấu đỉnh 4 là đã thăm.
* Danh sách đã thăm: [true, true, false, true, false, false]

**Bước 4: Đi sâu từ Đỉnh 4 đến Đỉnh 5**

* Đánh dấu đỉnh 5 là đã thăm.
* Danh sách đã thăm: [true, true, false, true, true, false]

**Bước 5: Quay lại và đi sâu từ Đỉnh 1 đến Đỉnh 3**

* Đánh dấu đỉnh 3 là đã thăm.
* Danh sách đã thăm: [true, true, true, true, true, false]

**Kết quả cuối cùng**

* Danh sách đã thăm theo thứ tự DFS: 1, 2, 4, 5, 3

**Hình ảnh Minh Họa**

less

Copy code

Bước 1: Bắt đầu từ đỉnh 1

1\* // 1 đã thăm

Bước 2: Đi sâu từ 1 đến 2

1\*

/ \

2\* 3

Bước 3: Đi sâu từ 2 đến 4

1\*

/ \

2\* 3

|

4\*

Bước 4: Đi sâu từ 4 đến 5

1\*

/ \

2\* 3

| |

4\* 5\*

Bước 5: Quay lại và đi từ 1 đến 3

1\*

/ \

2\* 3\*

| |

4\* 5\*

Các đỉnh có dấu \* là các đỉnh đã được thăm.

**Giải thích**

* **DFS đi sâu vào một nhánh của đồ thị trước**: Bắt đầu từ đỉnh 1, DFS đi sâu vào nhánh từ 1 -> 2 -> 4 -> 5.
* **Quay lại khi không còn đỉnh kề nào chưa thăm**: Sau khi thăm hết các đỉnh liên kết với 4, DFS quay lại 2 và tiếp tục từ 2.
* **Tiếp tục duyệt các nhánh khác**: Sau khi quay lại từ 2, DFS tiếp tục đến đỉnh 3.

DFS giúp thăm tất cả các đỉnh có thể tiếp cận được từ đỉnh ban đầu, bằng cách đi sâu vào mỗi nhánh trước khi quay lại và tiếp tục các nhánh khác. Trong ví dụ trên, thứ tự thăm các đỉnh là 1, 2, 4, 5, 3.

\*. Tìm thành phân liên thông

**Thành phần liên thông** trong đồ thị không hướng là một tập hợp các đỉnh sao cho có một đường đi giữa bất kỳ cặp đỉnh nào trong tập hợp đó và không có đỉnh nào trong tập hợp đó có đường đi tới đỉnh nào khác ngoài tập hợp.

Thuật toán để tìm tất cả các thành phần liên thông trong đồ thị có thể được thực hiện bằng cách sử dụng DFS (Depth-First Search) hoặc BFS (Breadth-First Search). Dưới đây là cách thực hiện bằng cả hai phương pháp.

**1. Thuật toán DFS để tìm thành phần liên thông**

**Ý tưởng**

* Duyệt qua tất cả các đỉnh của đồ thị.
* Nếu một đỉnh chưa được thăm, sử dụng DFS để tìm tất cả các đỉnh có thể đạt được từ đỉnh này, tất cả các đỉnh được thăm sẽ thuộc về cùng một thành phần liên thông.

**Cài đặt**

cpp

Copy code

#include <iostream>

#include <vector>

#include <cstring>

using namespace std;

const int MAXN = 100000;

vector<int> graph[MAXN];

bool visited[MAXN];

void dfs(int u) {

visited[u] = true;

for (int v : graph[u]) {

if (!visited[v]) {

dfs(v);

}

}

}

int main() {

int n, m;

cin >> n >> m;

// Đọc đồ thị

for (int i = 0; i < m; ++i) {

int a, b;

cin >> a >> b;

graph[a].push\_back(b);

graph[b].push\_back(a);

}

// Khởi tạo mảng visited

memset(visited, false, sizeof(visited));

int components = 0;

// Duyệt qua tất cả các đỉnh

for (int i = 1; i <= n; ++i) {

if (!visited[i]) {

// Nếu đỉnh chưa được thăm, đó là một thành phần liên thông mới

dfs(i);

components++;

}

}

cout << "Number of connected components: " << components << endl;

return 0;

}

**2. Thuật toán BFS để tìm thành phần liên thông**

**Ý tưởng**

* Tương tự như DFS, nhưng sử dụng BFS để duyệt qua các đỉnh trong cùng một thành phần liên thông.

**Cài đặt**

cpp

Copy code

#include <iostream>

#include <vector>

#include <queue>

#include <cstring>

using namespace std;

const int MAXN = 100000;

vector<int> graph[MAXN];

bool visited[MAXN];

void bfs(int start) {

queue<int> q;

q.push(start);

visited[start] = true;

while (!q.empty()) {

int u = q.front();

q.pop();

for (int v : graph[u]) {

if (!visited[v]) {

visited[v] = true;

q.push(v);

}

}

}

}

int main() {

int n, m;

cin >> n >> m;

// Đọc đồ thị

for (int i = 0; i < m; ++i) {

int a, b;

cin >> a >> b;

graph[a].push\_back(b);

graph[b].push\_back(a);

}

// Khởi tạo mảng visited

memset(visited, false, sizeof(visited));

int components = 0;

// Duyệt qua tất cả các đỉnh

for (int i = 1; i <= n; ++i) {

if (!visited[i]) {

// Nếu đỉnh chưa được thăm, đó là một thành phần liên thông mới

bfs(i);

components++;

}

}

cout << "Number of connected components: " << components << endl;

return 0;

}

**Giải thích mã nguồn**

1. **Cấu trúc dữ liệu**:
   * graph[MAXN]: Mảng vector để lưu danh sách kề của đồ thị.
   * visited[MAXN]: Mảng bool để lưu trạng thái đã thăm của các đỉnh.
2. **Hàm dfs(int u) (hoặc bfs(int start))**:
   * Duyệt đồ thị từ đỉnh u (hoặc start) và đánh dấu tất cả các đỉnh thuộc thành phần liên thông với đỉnh này là đã thăm.
3. **Vòng lặp chính trong main()**:
   * Duyệt qua tất cả các đỉnh từ 1 đến n.
   * Nếu đỉnh chưa được thăm, nó là một phần của thành phần liên thông mới, gọi hàm dfs hoặc bfs để duyệt qua toàn bộ thành phần liên thông và tăng bộ đếm components.

**Kết quả**

* Biến components lưu trữ số lượng thành phần liên thông trong đồ thị.

**Độ phức tạp thời gian**

Cả hai phương pháp DFS và BFS đều có độ phức tạp thời gian là **O(V + E)**, với V là số đỉnh và E là số cạnh trong đồ thị. Điều này là do chúng ta duyệt qua tất cả các đỉnh và tất cả các cạnh một lần.

**Bài 3: THÔNG BÁO (5.0 điểm)D:\BTC\THONGBAO.CPP**

Cơn bão Noru đã đi qua và để lại hậu quả nặng nề cho người dân vùng XX. Một số mạnh thường quân trên khắp cả nước đã hỗ trợ tiền, lương thực, thực phẩm cho người dân ở đây nhằm khắc phục phần nào những mất mát, đau thương mà người dân đã gánh chịu.

Ban cứu trợ của địa phương đã thông báo vấn đề trên đến người đứng đầu vùng XX (tổ trưởng) biết các thông tin cần thiết và yêu cầu tổ trưởng lập danh sách các hộ gia đình trong vùng đảm bảo điều kiện theo quy định của ban cứu trợ để nhận quà.

Nhằm thông tin kịp thời, bác tổ trưởng đã dùng ứng dụng tin nhắn của Zalo để liên lạc cho một số gia đình mà bác có sẵn số điện thoại, đồng thời nhờ các gia đình này tiếp tục chuyền thông báo cho các hàng xóm của họ biết để đối chiếu với thực tế thiệt hại của gia đình mình rồi đăng ký về tổ trưởng qua đường link được thực tế thiệt hại của gia đình mình rồi đăng ký về tổ trưởng qua đường link được tạo sẵn (khi nhận tin, có gia đình trả lời tin nhắn, có gia đình đăng ký trực tiếp luôn vào đường link mà không cần phản hồi). Quá trình cứ diễn ra như vậy cho đến khi thông tin được truyền đến tất cả các gia đình trong vùng.

**Yêu cầu**: Hãy cho biết bác tổ trưởng cần liên lạc trực tiếp với ít nhất bao nhiêu gia đình.

**Dữ liệu vào:** Cho bởi file MESS.INP gồm:

* Dòng thứ nhất chứa hai số n,mn, mn,m lần lượt là số gia đình trong vùng và số liên lạc (n,m≤105).
* M dòng tiếp theo, mỗi dòng chứa 2 số a,b cho biết gia đình aaa có thể liên lạc với gia đình b.

**Kết quả:** Ghi ra file MESS.OUT số gia đình ít nhất mà bác tổ trưởng cần liên lạc trực tiếp.

**Ví dụ:**

| **MESS.INP** | **MESS.OUT** |
| --- | --- |
| 13 9 | 4 |
| 1 6 |  |
| 6 3 |  |
| 6 8 |  |
| 12 9 |  |
| 2 4 |  |
| 4 5 |  |
| 7 10 |  |
| 10 11 |  |
| 12 13 |  |

**Ý tưởng bài toán:**

1. **Mô hình hóa bài toán**:
   * Bài toán có thể được mô hình hóa dưới dạng đồ thị, trong đó các gia đình là các đỉnh và các liên lạc giữa các gia đình là các cạnh không hướng.
2. **Xác định số thành phần liên thông**:
   * Một thành phần liên thông là một tập hợp các đỉnh trong đồ thị mà giữa các đỉnh trong cùng một tập hợp có thể đi tới nhau qua các cạnh của đồ thị.
   * Mục tiêu là tìm số thành phần liên thông trong đồ thị. Mỗi thành phần liên thông là một nhóm các gia đình mà bác tổ trưởng cần liên lạc trực tiếp ít nhất một gia đình để thông báo đến toàn bộ nhóm đó.
3. **Giải pháp**:
   * Sử dụng các thuật toán đồ thị như DFS (Depth-First Search) hoặc BFS (Breadth-First Search) để duyệt qua đồ thị và đếm số thành phần liên thông.
   * Mỗi lần bắt đầu một duyệt đồ thị từ một đỉnh chưa được thăm, bạn phát hiện ra một thành phần liên thông mới và tăng số lượng thành phần liên thông.

**Minh họa test trên thành đồ thị:**

**Dữ liệu mẫu:**

MESS.INP

13 9

1 6

6 3

6 8

12 9

2 4

4 5

7 10

10 11

12 13

**Đồ thị biểu diễn:**

* Số đỉnh (n = 13): 1, 2, 3, ..., 13 tương ứng với 13 hộ gia đình.
* Các cạnh biểu diễn mối liên lạc:

1 -- 6 -- 3

|

8

2 -- 4 -- 5

7 -- 10 -- 11

9 -- 12 -- 13

**Phân tích thành phần liên thông:**

* Nhóm 1: {1, 6, 3, 8}
* Nhóm 2: {2, 4, 5}
* Nhóm 3: {7, 10, 11}
* Nhóm 4: {9, 12, 13}

Như vậy có 4 thành phần liên thông. Bác tổ trưởng cần liên lạc trực tiếp với ít nhất 4 gia đình, mỗi gia đình đại diện cho một thành phần liên thông.

**Kết luận:**

* Đáp án là số lượng thành phần liên thông, trong ví dụ này là 4.

**Cách cài đặt:**

Bạn có thể dùng thuật toán DFS hoặc BFS để duyệt qua các đỉnh của đồ thị và tìm số lượng thành phần liên thông.

**Các bước giải quyết:**

1. **Xây dựng đồ thị**:
   * Đọc số lượng gia đình (n) và số lượng liên lạc (m).
   * Xây dựng đồ thị bằng cách thêm các cạnh giữa các đỉnh tương ứng với các liên lạc được cung cấp.
2. **Duyệt đồ thị để tìm thành phần liên thông**:
   * Sử dụng DFS hoặc BFS để duyệt qua tất cả các đỉnh của đồ thị.
   * Đếm số lần duyệt bắt đầu từ các đỉnh chưa được thăm (mỗi lần là một thành phần liên thông).
3. **Kết quả**:
   * Số lượng thành phần liên thông chính là số gia đình mà bác tổ trưởng cần liên lạc trực tiếp.

**Ví dụ:**

* **Input:**

Copy code

13 9

1 6

6 3

6 8

12 9

2 4

4 5

7 10

10 11

12 13

* **Output:**

Copy code

4

**Giải thích:** Đồ thị có 4 thành phần liên thông:

* + {1, 6, 3, 8}
  + {2, 4, 5}
  + {7, 10, 11}
  + {12, 9, 13}

Code 1

#include <bits/stdc++.h>

using namespace std;

const int MAXN = 100000;

vector<int> graph[MAXN];

bool visited[MAXN];

void bfs(int start) {

queue<int> q;

q.push(start);

visited[start] = true;

while (!q.empty()) {

int u = q.front();

q.pop();

for (int v : graph[u]) {

if (!visited[v]) {

visited[v] = true;

q.push(v);

}

}

}

}

int main() {

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

int n, m;

cin >> n >> m;

// Đọc đồ thị

for (int i = 0; i < m; ++i) {

int a, b;

cin >> a >> b;

graph[a].push\_back(b);

graph[b].push\_back(a);

}

// Đếm số thành phần liên thông

int components = 0;

memset(visited, false, sizeof(visited));

for (int i = 1; i <= n; ++i) {

if (!visited[i]) {

bfs(i);

components++;

}

}

// In kết quả ra file

cout << components << endl;

return 0;

}

**Giải thích từng bước:**

1. **Thư viện và khai báo:**

cpp

Copy code

#include <iostream>

#include <vector>

#include <queue>

#include <cstring>

using namespace std;

const int MAXN = 100000;

vector<int> graph[MAXN];

bool visited[MAXN];

* + #include <iostream>: Để sử dụng các thao tác nhập/xuất (input/output).
  + #include <vector>: Để sử dụng cấu trúc dữ liệu vector cho danh sách kề của đồ thị.
  + #include <queue>: Để sử dụng hàng đợi (queue) trong thuật toán BFS.
  + #include <cstring>: Để sử dụng hàm memset để khởi tạo mảng.
  + const int MAXN = 100000;: Định nghĩa hằng số MAXN, kích thước tối đa của đồ thị.
  + vector<int> graph[MAXN];: Mảng các vector lưu trữ đồ thị.
  + bool visited[MAXN];: Mảng đánh dấu các đỉnh đã được thăm.

1. **Hàm BFS (Breadth-First Search):**

cpp

Copy code

void bfs(int start) {

queue<int> q;

q.push(start);

visited[start] = true;

while (!q.empty()) {

int u = q.front();

q.pop();

for (int v : graph[u]) {

if (!visited[v]) {

visited[v] = true;

q.push(v);

}

}

}

}

* + Hàm BFS duyệt qua các đỉnh của đồ thị bắt đầu từ đỉnh start.
  + Dùng hàng đợi để thực hiện BFS.
  + Đánh dấu đỉnh start là đã thăm.
  + Duyệt các đỉnh kề của đỉnh u, nếu đỉnh v chưa được thăm thì đánh dấu và thêm vào hàng đợi.

1. **Hàm main:**

cpp

Copy code

int main() {

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

int n, m;

cin >> n >> m;

* + ios\_base::sync\_with\_stdio(false); và cin.tie(NULL); là các tối ưu hóa cho việc nhập/xuất để chương trình chạy nhanh hơn.

cpp

Copy code

// Đọc đồ thị

for (int i = 0; i < m; ++i) {

int a, b;

cin >> a >> b;

graph[a].push\_back(b);

graph[b].push\_back(a);

}

* + Đọc số lượng đỉnh n và số lượng cạnh m.
  + Đọc các cạnh và xây dựng đồ thị bằng cách thêm các đỉnh vào danh sách kề của nhau.

cpp

Copy code

// Đếm số thành phần liên thông

int components = 0;

memset(visited, false, sizeof(visited));

for (int i = 1; i <= n; ++i) {

if (!visited[i]) {

bfs(i);

components++;

}

}

* + Khởi tạo mảng visited để đánh dấu các đỉnh đã thăm.
  + Duyệt qua tất cả các đỉnh. Nếu một đỉnh chưa được thăm, gọi hàm BFS từ đỉnh đó để đánh dấu tất cả các đỉnh trong thành phần liên thông và tăng số lượng thành phần liên thông.

cpp

Copy code

// In kết quả ra file

cout << components << endl;

return 0;

}

* + In số lượng thành phần liên thông ra màn hình (hoặc file đầu ra).

BFS (Breadth-First Search) là một thuật toán tìm kiếm trong đồ thị hoặc cây, được sử dụng để duyệt qua các đỉnh của đồ thị theo chiều rộng (tức là từ một đỉnh gốc, nó sẽ tìm tất cả các đỉnh kề của nó trước khi di chuyển đến các đỉnh sâu hơn).

**Nguyên tắc hoạt động của BFS:**

1. **Khởi tạo**:
   * Bắt đầu từ một đỉnh gốc (hoặc đỉnh bắt đầu).
   * Đặt đỉnh gốc vào hàng đợi (queue) và đánh dấu nó là đã thăm.
2. **Duyệt các đỉnh**:
   * Lấy đỉnh từ hàng đợi.
   * Duyệt qua tất cả các đỉnh kề của đỉnh đó.
   * Nếu một đỉnh kề chưa được thăm, đánh dấu nó là đã thăm và thêm nó vào hàng đợi.
3. **Tiếp tục**:
   * Lặp lại bước 2 cho đến khi hàng đợi rỗng.

**Đặc điểm của BFS:**

* **Chiều rộng trước**: BFS duyệt qua các đỉnh theo chiều rộng của đồ thị, nghĩa là nó sẽ kiểm tra tất cả các đỉnh ở một cấp độ (hoặc khoảng cách) trước khi di chuyển xuống cấp độ tiếp theo.
* **Khoảng cách ngắn nhất**: Trong một đồ thị không trọng số, BFS đảm bảo tìm được đường đi ngắn nhất từ đỉnh gốc đến các đỉnh khác.
* **Hàng đợi**: BFS sử dụng cấu trúc dữ liệu hàng đợi (queue) để theo dõi các đỉnh cần duyệt.

**Ứng dụng của BFS:**

* **Tìm đường đi ngắn nhất** trong đồ thị không trọng số.
* **Tìm thành phần liên thông** trong đồ thị.
* **Thuật toán tìm kiếm tối ưu** cho nhiều bài toán trong đồ thị, như tìm mức độ kết nối.

CÁch 2

#include <iostream>

#include <vector>

#include <cstring>

using namespace std;

const int MAXN = 100000;

vector<int> graph[MAXN];

bool visited[MAXN];

void dfs(int u) {

visited[u] = true;

for (int v : graph[u]) {

if (!visited[v]) {

dfs(v);

}

}

}

int main() {

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

int n, m;

cin >> n >> m;

// Đọc đồ thị

for (int i = 0; i < m; ++i) {

int a, b;

cin >> a >> b;

graph[a].push\_back(b);

graph[b].push\_back(a);

}

// Đếm số thành phần liên thông

int components = 0;

memset(visited, false, sizeof(visited));

for (int i = 1; i <= n; ++i) {

if (!visited[i]) {

dfs(i);

components++;

}

}

// In kết quả

cout << components << endl;

return 0;

}

const int MAXN = 100000;

vector<int> graph[MAXN];

bool visited[MAXN];

* **const int MAXN = 100000;**: Định nghĩa hằng số MAXN với giá trị là 100000, đây là kích thước tối đa của đồ thị (số lượng đỉnh).
* **vector<int> graph[MAXN];**: Mảng graph chứa các vector. Mỗi graph[i] là danh sách các đỉnh kề của đỉnh i. Điều này biểu diễn đồ thị dưới dạng danh sách kề.
* **bool visited[MAXN];**: Mảng visited dùng để đánh dấu các đỉnh đã được thăm trong quá trình duyệt đồ thị.

cpp

Copy code

void dfs(int u) {

visited[u] = true;

for (int v : graph[u]) {

if (!visited[v]) {

dfs(v);

}

}

}

* **void dfs(int u)**: Hàm thực hiện tìm kiếm theo chiều sâu (Depth-First Search) bắt đầu từ đỉnh u.
* **visited[u] = true;**: Đánh dấu đỉnh u là đã thăm.
* **for (int v : graph[u])**: Duyệt qua tất cả các đỉnh kề v của đỉnh u.
* **if (!visited[v])**: Nếu đỉnh v chưa được thăm, gọi đệ quy hàm dfs(v) để tiếp tục duyệt từ đỉnh v.

cpp

Copy code

int main() {

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

* **ios\_base::sync\_with\_stdio(false);** và **cin.tie(NULL);**: Các dòng này dùng để tối ưu hóa hiệu suất của thao tác nhập/xuất bằng cách tắt đồng bộ hóa giữa C++ streams và C streams.

cpp

Copy code

int n, m;

cin >> n >> m;

* **int n, m;**: Khai báo biến n (số lượng đỉnh) và m (số lượng cạnh).
* **cin >> n >> m;**: Đọc giá trị n và m từ đầu vào.

cpp

Copy code

// Đọc đồ thị

for (int i = 0; i < m; ++i) {

int a, b;

cin >> a >> b;

graph[a].push\_back(b);

graph[b].push\_back(a);

}

* **for (int i = 0; i < m; ++i)**: Vòng lặp để đọc từng cạnh trong đồ thị.
* **int a, b;**: Khai báo hai biến a và b để lưu trữ các đỉnh của mỗi cạnh.
* **cin >> a >> b;**: Đọc các đỉnh a và b của một cạnh.
* **graph[a].push\_back(b);** và **graph[b].push\_back(a);**: Thêm đỉnh b vào danh sách kề của đỉnh a, và đỉnh a vào danh sách kề của đỉnh b, vì đồ thị là không hướng.

cpp

Copy code

// Đếm số thành phần liên thông

int components = 0;

memset(visited, false, sizeof(visited));

for (int i = 1; i <= n; ++i) {

if (!visited[i]) {

dfs(i);

components++;

}

}

* **int components = 0;**: Khai báo biến components để đếm số thành phần liên thông.
* **memset(visited, false, sizeof(visited));**: Đặt tất cả các giá trị trong mảng visited thành false, nghĩa là chưa đỉnh nào được thăm.
* **for (int i = 1; i <= n; ++i)**: Vòng lặp qua tất cả các đỉnh từ 1 đến n.
* **if (!visited[i])**: Nếu đỉnh i chưa được thăm, gọi hàm dfs(i) để duyệt từ đỉnh i, và tăng số lượng thành phần liên thông.

cpp

Copy code

// In kết quả

cout << components << endl;

return 0;

}

* **cout << components << endl;**: In số lượng thành phần liên thông ra màn hình.
* **return 0;**: Kết thúc chương trình và trả về mã trạng thái 0, biểu thị chương trình đã chạy thành công.