

|  |  |
| --- | --- |
| **Input** | **Output** |
| **6 7 1 5**  **1 2**  **1 3**  **3 5**  **2 4**  **4 6**  **2 3**  **1 4** | **2**  **1 3 5** |

**Bài tập 1: Tìm đường đi ngắn nhất  
Cho đồ thị vô hướng, không có trọng số gồm đỉnh, cạnh, 2 đỉnh S và T lần lượt là điểm bắt đầu và kết thúc của một đường đi.  
Yêu cầu: Tìm đường đi ngắn nhất từ S tới T  
Dữ liệu vào:   
+ Dòng đầu tiên chứa 2 số nguyên dương 3 ≤ ≤ 200)  
+ dòng tiếp theo, mỗi dòng chứa hai số nguyên dương mô tả một cạnh của đồ thị.  
Kết quả ra:  
- Nếu không tồn tại đường đi từ S đến T thì ghi ra -1;  
- Nếu tồn tại đường đi thì ghi ra dòng đầu tiên là số cạnh đã đi qua, dòng tiếp theo là danh sách các đỉnh đi từ đỉnh S tới T**

**BƯỚC 1. KHỞI TẠO U,V**

cin >> n >> m >> S >> T;  
for (int i = 1; i <= m; i++)

{  
 int u, v; cin >> u >> v;  
 adj[u].push\_back(v);  
 // tức v kề u  
 adj[v].push\_back(u);  
 // tức u kề v **}**

|  |  |  |  |
| --- | --- | --- | --- |
| **u** | | **v** | |
| **1** | **2;3;4** | **1** |  |
| **2** | **4;3** | **2** | **1** |
| **3** | **5** | **3** | **1;2** |
| **4** | **6** | **4** | **2;1** |
| **5** |  | **5** | **3** |
| **6** |  | **6** | **4** |

**Vised**



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **f** | **f** | **f** | **f** | **f** | **f** |
| **1** | **2** | **3** | **4** | **5** | **6** |

**L:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **-1** | **-1** | **-1** | **-1** | **-1** | **-1** |
| **1** | **2** | **3** | **4** | **5** | **6** |

**Dfs(1)**

**Queue**

|  |
| --- |
| **1** |

**U=1**

**V=2,3,4: Chưa thăm**

**Vised**



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **T** | **T** | **T** | **T** | **f** | **f** |
| **1** | **2** | **3** | **4** | **5** | **6** |

**L:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **0** | **1** | **1** | **1** | **-1** | **-1** |
| **1** | **2** | **3** | **4** | **5** | **6** |

**path**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **1** | **1** | 1 | **-1** | **-1** |
| **1** | **2** | **3** | 4 | **5** | **6** |

**QUEUE**

|  |  |  |  |
| --- | --- | --- | --- |
| **2** | **3** | **4** |  |

**U=2**

**QUEUE**

|  |  |  |
| --- | --- | --- |
| **3** | **4** | **5** |

**V=4;V=3: ĐÃ THĂM**

**U=3**

**QUEUE**

|  |  |
| --- | --- |
| **4** | **5** |

**V=5;chưa thăm**

**Vised**



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **T** | **T** | **T** | **T** | **T** | **f** |
| **1** | **2** | **3** | **4** | **5** | **6** |

**L:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **0** | **1** | **1** | **1** | **2** | **-1** |
| **1** | **2** | **3** | **4** | **5** | **6** |

**path**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **1** | **1** | 1 | **3** | **-1** |
| **1** | **2** | **3** | 4 | **5** | **6** |

**U=4**

**QUEUE**

|  |  |
| --- | --- |
| **5** | **6** |

**V=6;chưa thăm**

**Vised**



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **T** | **T** | **T** | **T** | **T** | T |
| **1** | **2** | **3** | **4** | **5** | **6** |

**L:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **0** | **1** | **1** | **1** | **2** | **2** |
| **1** | **2** | **3** | **4** | **5** | **6** |

**path**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **1** | **1** | 1 | **3** | **4** |
| **1** | **2** | **3** | 4 | **5** | **6** |

**U=5: RỒNG**

**U=6: RỖNG**

**KẾT THÚC BFS**

**PATH:**

|  |  |  |
| --- | --- | --- |
| **5** | **3** | **1** |

**Gọi Hàm:** memset(L, -1, sizeof (L));  
 // gán mọi phần tử ở mảng L là -1 tức hiện tại chưa tồn tại đường đi  
 bfs(S);  
 cout << L[T] << '\n';  
 if (L[T] == -1) {  
 return 0;  
 }  
 find\_path(T);

**BFS:**

void bfs(int s) {  
 memset(visited, false, sizeof (visited));  
 // gán tất cả phần tử ở mảng visited = false   
// tức là chưa có đỉnh nào được thăm  
 queue<int> q;  
 q.push(s);  
 L[s] = 0;  
 // đường đi từ s tới chính nó được cho là 0  
 visited[s] = true;  
 // đã thăm đỉnh s  
 while (q.size()) {  
 // trong khi q chưa rỗng  
 int u = q.front();  
 // u là phần tử ở đầu hàng đợi q  
 q.pop();  
 // xóa u  
 for (auto v : adj[u]) {  
 // for v kề u  
 if (visited[v] == false) {  
 // nếu v chưa thăm  
 L[v] = L[u] + 1;  
 trace[v] = u;  
 // gán trace[v] = u do u là đỉnh được xét liền trước v  
 q.push(v);  
 // push v vào cuối hàng đợi q  
 visited[v] = true;  
 // đánh dấu v đã được thăm  
 }  
 }   
 }  
}

**Cây:**

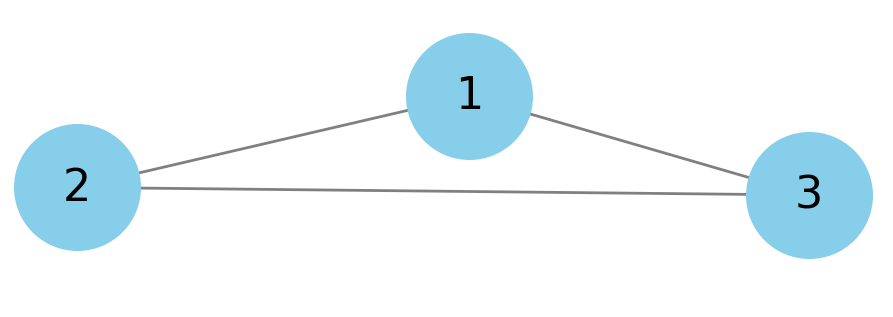
void find\_path(int t) {  
 int x = t;  
 while (true) {  
 Path.push\_back(x);  
 if (x == S) {  
 break;  
 }  
 x = trace[x];  
 }  
 for (int i = Path.size() - 1; i >= 0; i--) {  
 cout << Path[i] << " ";  
 }  
}

**Tổng quan chương trình:**

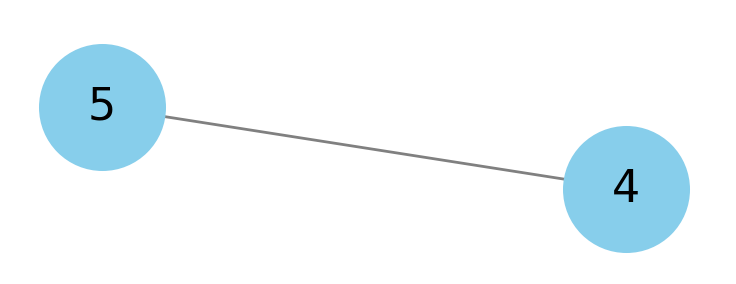
#include<bits/stdc++.h>  
using namespace std;  
const int N = 200;  
vector<int> adj[N + 1], Path;  
  
bool visited[N + 1];  
  
int n, m, S, T, L[N + 1], trace[N + 1];  
  
void bfs(int s) {  
 memset(visited, false, sizeof (visited));  
 // gán tất cả phần tử ở mảng visited = false   
// tức là chưa có đỉnh nào được thăm  
 queue<int> q;  
 q.push(s);  
 L[s] = 0;  
 // đường đi từ s tới chính nó được cho là 0  
 visited[s] = true;  
 // đã thăm đỉnh s  
 while (q.size()) {  
 // trong khi q chưa rỗng  
 int u = q.front();  
 // u là phần tử ở đầu hàng đợi q  
 q.pop();  
 // xóa u  
 for (auto v : adj[u]) {  
 // for v kề u  
 if (visited[v] == false) {  
 // nếu v chưa thăm  
 L[v] = L[u] + 1;  
 trace[v] = u;  
 // gán trace[v] = u do u là đỉnh được xét liền trước v  
 q.push(v);  
 // push v vào cuối hàng đợi q  
 visited[v] = true;  
 // đánh dấu v đã được thăm  
 }  
 }   
 }  
}  
  
void find\_path(int t) {  
 int x = t;  
 while (true) {  
 Path.push\_back(x);  
 if (x == S) {  
 break;  
 }  
 x = trace[x];  
 }  
 for (int i = Path.size() - 1; i >= 0; i--) {  
 cout << Path[i] << " ";  
 }  
}  
  
signed main() {  
 ios\_base::sync\_with\_stdio(false); cin.tie(nullptr);  
 cin >> n >> m >> S >> T;  
 for (int i = 1; i <= m; i++) {  
 int u, v; cin >> u >> v;  
 adj[u].push\_back(v);  
 // tức v kề u  
 adj[v].push\_back(u);  
 // tức u kề v  
 }  
 memset(L, -1, sizeof (L));  
 // gán mọi phần tử ở mảng L là -1 tức hiện tại chưa tồn tại đường đi  
 bfs(S);  
 cout << L[T] << '\n';  
 if (L[T] == -1) {  
 return 0;  
 }  
 find\_path(T);  
 return 0;  
}

**TÍNH THÀNH PHẦN LIÊN THÔNG:**

Bài tập 2: Đếm thành phần liên thông  
Cho đồ thị vô hướng gồm đỉnh, cạnh .  
Yêu cầu: Hãy xác định xem đồ thị có bao nhiêu thành phần liên thông?  
Dữ liệu vào:   
+ Dòng đầu tiên chứa 2 số nguyên dương   
+ dòng tiếp theo, mỗi dòng chứa hai số nguyên dương mô tả một cạnh của đồ thị.  
Kết quả ra:  
Số nguyên duy nhất là số thành phần liên thông của đồ thị

****

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Giải thích** |
| **5 4**  **1 2**  **2 3**  **3 1**  **4 5** | **2** | **Hai thành phần liên thông là (1, 2, 3) và (4, 5)** |

****

#include <bits/stdc++.h>

using namespace std;

int n, m;

bool visited[100001];

vector < int > adj[100001];

void dfs(int u)

{

visited[u] = true;

for (int v: adj[u])

if (!visited[v])

dfs(v); }

int dem()

{

int ccp\_amount = 0; // Số thành phần liên thông.

// Duyệt từng đỉnh, đỉnh nào chưa thăm thì thăm DFS từ đỉnh đó.

for (int u = 1; u <= n; ++u)

if (!visited[u])

{

dfs(u);

++ccp\_amount;

}

return ccp\_amount;

}

int main()

{

cin >> n >> m;

// Xây dựng danh sách kề của đồ thị.

for (int i = 1; i <= m; ++i)

{

int u, v;

cin >> u >> v;

// Do đồ thị là vô hướng nên u thuộc danh sách kề của v và ngược lại.

adj[u].push\_back(v);

adj[v].push\_back(u);

}

cout << dem();

}

**Bài toán:**

Nam đang chơi trò chơi tìm đường trong mê cung, và mục tiêu là kiểm tra xem Nam có thể di chuyển từ ô trên cùng bên trái (ô số 1) tới ô dưới cùng bên phải của mê cung (ô số 50) hay không. Mê cung được mô tả dưới dạng ma trận 2D, trong đó:

* **1** là ô có thể đi qua.
* **0** là tường hoặc ô không thể đi qua.

Chúng ta sẽ sử dụng BFS để tìm đường ngắn nhất từ vị trí bắt đầu đến vị trí kết thúc trong mê cung.

**Giải thích thuật toán:**

1. **Cấu trúc mê cung**:
   * Mê cung được biểu diễn dưới dạng ma trận 2D với các ô có giá trị 1 là đường đi và 0 là tường.
2. **Hàm isValid**:
   * Hàm kiểm tra xem ô tiếp theo có hợp lệ không, tức là không vượt quá kích thước của mê cung, không phải là tường, và chưa được thăm trước đó.
3. **BFS (Breadth-First Search)**:
   * BFS sử dụng hàng đợi để duyệt qua tất cả các ô có thể đi được, bắt đầu từ ô xuất phát. Mỗi lần duyệt, BFS kiểm tra tất cả các ô liền kề theo các hướng đi (xuống, phải, lên, trái) và thêm chúng vào hàng đợi nếu hợp lệ.
4. **Kết quả**:
   * Nếu tìm thấy đường đi đến ô kết thúc (4, 9), chương trình sẽ in ra "Nam có thể thoát khỏi mê cung!".
   * Nếu không tìm thấy đường, in ra "Nam không thể thoát khỏi mê cung!".

**Đầu vào và Đầu ra:**

* **Đầu vào**: Mê cung được mô tả như trong ví dụ với 5 hàng và 10 cột.
* **Đầu ra**: Xác định xem Nam có thể thoát khỏi mê cung từ ô bắt đầu đến ô kết thúc hay không.

**Ưu điểm của BFS:**

* BFS đảm bảo tìm được đường đi ngắn nhất nếu có.
* BFS duyệt theo tầng, nên sẽ kiểm tra tất cả các ô xung quanh trước khi chuyển sang tầng tiếp theo.

**Test 1: Có đường đi (đơn giản)**

1 1 1 1 1 1 1 1 1 1

0 0 0 0 0 0 0 0 0 1

1 1 1 1 1 1 1 1 0 1

1 0 0 0 0 0 0 1 0 1

1 1 1 1 1 1 1 1 1 1

👉 **Kết quả mong đợi:** YES

**🚫 Test 2: Bị chặn giữa đường**

1 1 0 1 1 1 1 1 1 1

1 0 0 1 0 0 0 0 0 1

1 0 1 1 1 1 1 1 0 1

1 0 0 0 0 0 0 1 0 1

1 1 1 1 1 1 1 0 0 0

👉 **Kết quả mong đợi:** NO

**⚡ Test 3: Đường quanh co phức tạp**

1 0 1 1 1 0 1 1 1 1

1 0 1 0 1 0 1 0 0 1

1 1 1 0 1 1 1 0 1 1

0 0 1 0 0 0 1 1 1 0

1 1 1 1 1 1 0 0 1 1

👉 **Kết quả mong đợi:** YES

**Bài toán:**

Cho một lưới hình chữ nhật **m x n** (m hàng và n cột), trong đó:

* **1** đại diện cho đất liền.
* **0** đại diện cho nước.

Nhiệm vụ của bạn là đếm số lượng **đảo** trên lưới. Một đảo được tạo thành từ các ô đất liền liền kề theo các hướng lên, xuống, trái, phải. Hãy viết chương trình sử dụng BFS để giải bài toán này.

**Giải thích:**

1. **Định nghĩa bài toán**:
   * Lưới 2D được biểu diễn bởi một ma trận grid trong đó 1 là đất liền và 0 là nước.
   * Mỗi ô đất liền được kết nối với ô liền kề theo các hướng lên, xuống, trái, phải để tạo thành một đảo.
2. **Hàm isValid**:
   * Kiểm tra xem một ô có hợp lệ không, tức là ô không vượt quá giới hạn của lưới và là đất liền (giá trị là 1) và chưa được thăm.
3. **BFS để tìm một đảo**:
   * Khi tìm thấy một ô đất liền chưa được thăm, ta bắt đầu duyệt BFS từ ô đó và đánh dấu tất cả các ô đất liền thuộc cùng một đảo. Điều này đảm bảo rằng mỗi đảo chỉ được đếm một lần.
4. **Hàm countIslands**:
   * Hàm này duyệt qua tất cả các ô trong lưới và mỗi khi tìm thấy một ô đất liền chưa được thăm, nó sẽ gọi hàm BFS để khám phá toàn bộ đảo đó. Đồng thời, biến islandCount sẽ được tăng lên mỗi khi phát hiện một đảo mới.
5. **Đầu ra**:
   * Kết quả của chương trình là số lượng đảo trên lưới.

**Ví dụ:**

* Với lưới:

1 1 0 0 0

1 1 0 0 1

0 0 0 1 1

0 0 0 0 0

1 1 0 1 1

Kết quả sẽ là:

Số lượng đảo là: 4

**Giải thuật:**

* **BFS** giúp duyệt toàn bộ các ô của một đảo bằng cách bắt đầu từ một ô đất liền, sau đó mở rộng dần sang các ô lân cận để đánh dấu các ô cùng thuộc đảo đó.
* Sau khi hoàn thành việc duyệt một đảo, chúng ta tăng số lượng đảo lên 1.

**Độ phức tạp:**

* **Độ phức tạp thời gian**: O(m \* n), với m là số hàng và n là số cột. Chúng ta duyệt qua toàn bộ lưới chỉ một lần.
* **Độ phức tạp không gian**: O(m \* n), dùng để lưu trạng thái đã thăm của các ô.

Chương trình này giải quyết bài toán đếm số lượng đảo một cách hiệu quả bằng cách sử dụng BFS.

Dưới đây là một bài toán khác vận dụng thuật toán **BFS (Breadth-First Search)** để giải bài toán "lũ lụt" trong C++.

**Bài toán:**

Bạn được cho một bản đồ thành phố được biểu diễn dưới dạng một lưới 2D có kích thước **m x n**, trong đó:

* **0** đại diện cho các ngôi nhà.
* **1** đại diện cho các bức tường hoặc chướng ngại vật.
* **2** đại diện cho các trạm cứu hỏa.

Mục tiêu của bạn là tìm khoảng cách ngắn nhất từ mỗi ngôi nhà đến trạm cứu hỏa gần nhất. Nếu không có cách nào để đến được trạm cứu hỏa, hãy trả về -1 cho ngôi nhà đó.

**Giải thích:**

1. **Định nghĩa bài toán**:
   * Bạn cần tính khoảng cách ngắn nhất từ mỗi ngôi nhà (ô có giá trị **0**) đến trạm cứu hỏa (ô có giá trị **2**). Các ô có giá trị **1** là chướng ngại vật và không thể đi qua.
   * Nếu một ngôi nhà không thể đến được trạm cứu hỏa, trả về giá trị **-1** cho ngôi nhà đó.
2. **Cấu trúc thuật toán**:
   * **BFS** là lựa chọn tốt cho bài toán này vì bạn cần tính khoảng cách ngắn nhất trong một lưới.
   * **Hàng đợi BFS** chứa tất cả các trạm cứu hỏa ban đầu, và từ đó mở rộng ra để tính khoảng cách đến các ô ngôi nhà.
3. **BFS**:
   * Bắt đầu từ tất cả các trạm cứu hỏa.
   * Duyệt qua 4 hướng di chuyển (lên, xuống, trái, phải) và kiểm tra xem các ô tiếp theo có hợp lệ không.
   * Cập nhật khoảng cách của các ngôi nhà bằng cách cộng thêm **1** vào khoảng cách của ô trước đó.
4. **Hàm shortestDistanceToFireStation**:
   * Khởi tạo lưới để lưu khoảng cách, tất cả giá trị ban đầu là **-1** (chưa thăm).
   * Sau khi BFS hoàn thành, trả về kết quả là lưới khoảng cách.

**Đầu ra:**

Với lưới đầu vào:

0 1 2 0 0

0 0 1 1 0

2 0 0 1 0

0 1 0 0 0

0 1 0 2 0

Kết quả khoảng cách ngắn nhất từ các ngôi nhà đến trạm cứu hỏa:

2 -1 0 1 2

1 2 -1 -1 3

0 1 2 -1 3

1 -1 2 1 2

2 -1 1 0 1

**Giải thuật:**

* Chúng ta dùng BFS để đảm bảo rằng các ô được duyệt theo từng lớp khoảng cách, điều này giúp tính toán khoảng cách ngắn nhất từ các trạm cứu hỏa đến các ngôi nhà một cách chính xác.

**Độ phức tạp:**

* **Độ phức tạp thời gian**: O(m \* n) với m là số hàng và n là số cột, vì ta duyệt qua toàn bộ lưới một lần.
* **Độ phức tạp không gian**: O(m \* n) cho hàng đợi và lưới lưu khoảng cách.

Chương trình này là một ứng dụng điển hình của BFS trong việc tìm khoảng cách ngắn nhất trong một lưới, với các chướng ngại vật và điểm xuất phát từ nhiều vị trí khác nhau (các trạm cứu hỏa).