CHUYÊN ĐỀ: LÝ THUYẾT ĐỒ THỊ

Đặng Phúc An Khang*

Ngày 9 tháng 7 năm 2025

Tóm tắt nội dung

Code:

- C/C++: https://github.com/GrootTheDeveloper/OLP-ICPC/tree/master/2025/C%2B%2B.
- Python:

Tài khoản trên các Online Judge:

- Codeforces: https://codeforces.com/profile/vuivethoima.
- VNOI: oj.vnoi.info/user/Groot.
- IUHCoder: oj.iuhcoder.com/user/ankhang2111.
- MarisaOJ: https://marisaoj.com/user/grootsiuvip/submissions.
- CSES: https://cses.fi/user/212174.
- UMTOJ: sot.umtoj.edu.vn/user/grootsiuvip.
- SPOJ: www.spoj.com/users/grootsiuvip/.
- POJ: http://poj.org/userstatus?user_id=vuivethoima.
- ATCoder: https://atcoder.jp/users/grootsiuvip
- OnlineJudge.org: vuivethoima
- updating...

Mục lục

*E-mail: ankhangluonvuituoi@gmail.com. Tây Ninh, Việt Nam.

1 Preliminaries – Kiến thức chuẩn bị

Resources – Tài nguyên.

1. [**CP10**]. *CP10. Competitive Programming* `https://drive.google.com/drive/folders/1MTEVHT-7nBnMJ7C9LgyAR_pEVSE3FlKz?fbclid=IwAR3TovIj2rKCRe1a4oZxW-LQCoEoVkipVAvCzwrr0nJ6GzcAd47P6LO1Rwc`
2. [**cp-algorithms**]. *Algorithms for Competitive Programming* `https://cp-algorithms.com`
3. [**VNOI-WIKI**]. *Thư viện VNOI* `https://wiki.vnoi.info`

2 Kiến thức

2.1 Giới thiệu về đồ thị và thuật toán DFS

2.1.1 Lý thuyết đồ thị là gì?

Lý thuyết đồ thị là một nhánh của toán học, cụ thể thuộc toán rời rạc. Lý thuyết đồ thị chuyên nghiên cứu các bài toán liên quan đến việc biểu diễn và phân tích các sự vật, hiện tượng hoặc trạng thái có mối quan hệ lẫn nhau thông qua mô hình đồ thị.

Một số minh họa điển hình: Mạng lưới giao thông, cây phả hệ (cây gia phả), mạng máy tính, sơ đồ tổ chức, v.v.

2.1.2 Một số khái niệm căn bản trong lý thuyết đồ thị

1. **Đỉnh:** Được biểu diễn nhằm mục đích thể hiện sự vật, sự việc hay một trạng thái.
2. **Cạnh:** Biểu diễn cho mối quan hệ giữa 2 đỉnh với nhau. **Lưu ý:** Giữa 2 đỉnh trong đồ thị có thể có cạnh, không có, hoặc có thể có nhiều cạnh với nhau. Cạnh được chia thành 2 dạng:
   (a) **Cạnh vô hướng:** Nếu một cạnh vô hướng nối 2 đỉnh u và v, thì u có thể đến v trực tiếp và ngược lại.
   (b) **Cạnh có hướng:** Nếu một cạnh có hướng nối từ đỉnh u đến đỉnh v, thì ta có thể đi trực tiếp từ u đến v, nhưng không thể đi ngược lại từ v đến u trừ khi có một cạnh khác từ v đến u.
3. **Đường đi:** Một đường đi là một danh sách các đỉnh $x_1, x_2, x_3, x_4, \ldots, x_k$. Trong đó 2 đỉnh x_i và x_{i+1} thì có một đường nối trực tiếp để đi từ $x_i \to x_{i+1}$.
4. **Trọng số:** Là một giá trị trên cạnh (hoặc trên đỉnh) nhằm thể hiện một thông số nào đó với bài toán ta đang xét.

2.1.3 Danh sách kề

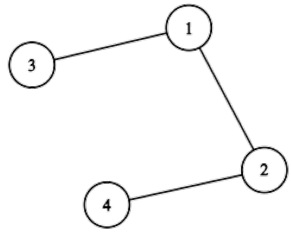
Một trong những cách phổ biến để biểu diễn đồ thị là sử dụng **danh sách kề (adjacency list)**. Cách biểu diễn này đặc biệt hiệu quả đối với đồ thị thưa (sparse graph).
Cụ thể, ta sử dụng cấu trúc dữ liệu `vector<int> adj[u]` trong C++, trong đó:

- Mỗi phần tử `adj[u]` là một vector chứa các đỉnh kề với đỉnh u.
- Nghĩa là nếu có cạnh nối từ đỉnh u đến đỉnh v thì v sẽ xuất hiện trong `adj[u]`.
- Đối với đồ thị vô hướng, nếu có cạnh giữa u và v thì cả $v \in \texttt{adj}[u]$ và $u \in \texttt{adj}[v]$.

Ví dụ:
Giả sử đồ thị vô hướng có các cạnh: $(1, 2)$, $(1, 3)$, $(2, 4)$ thì danh sách kề sẽ là:

$$\texttt{adj[1]} = \{2, 3\}$$

$$\texttt{adj[2]} = \{1, 4\}$$
$$\texttt{adj[3]} = \{1\}$$
$$\texttt{adj[4]} = \{2\}$$



Hình 1: Minh họa danh sách kề của đồ thị vô hướng vừa mô tả

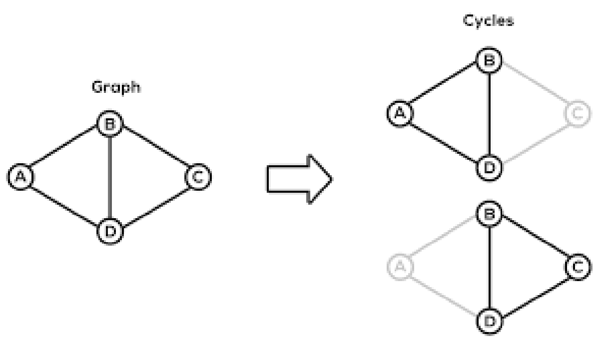
Cách biểu diễn này có độ phức tạp về bộ nhớ là $\mathcal{O}(n + m)$, với n là số đỉnh và m là số cạnh.

2.1.4 Chu trình (Cycle)

Một **chu trình** là một đường đi bắt đầu và kết thúc tại cùng một đỉnh, trong đó không có đỉnh nào khác (ngoại trừ đỉnh đầu/cuối) được lặp lại.

- Trong đồ thị vô hướng: chu trình là dãy đỉnh $v_1 \to v_2 \to \cdots \to v_k \to v_1$ với $k \geq 3$.
- Trong đồ thị có hướng: các cung phải có hướng phù hợp với trình tự chu trình.

Một chu trình được gọi là *chu trình đơn giản* nếu không có cạnh hoặc đỉnh nào bị lặp lại (trừ đỉnh đầu/cuối).

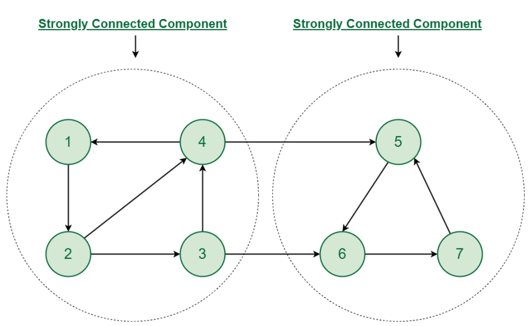


Hình 2: Minh họa chu trình của đồ thị vô hướng

2.1.5 Thành phần liên thông (Connected Component)

Một **thành phần liên thông** là một tập con các đỉnh sao cho giữa mọi cặp đỉnh trong đó đều tồn tại một đường đi.

- Với đồ thị vô hướng: liên thông nếu có đường đi giữa mọi cặp đỉnh.
- Với đồ thị có hướng:
  - *Liên thông mạnh* nếu tồn tại đường đi theo chiều từ mọi đỉnh đến mọi đỉnh khác.
  - *Liên thông yếu* nếu bỏ hướng trên các cạnh thì đồ thị trở nên liên thông.



Hình 3: Minh họa thành phần liên thông mạnh của đồ thị có hướng

2.1.6 Bậc của đỉnh (Degree)

- Trong đồ thị vô hướng, bậc của một đỉnh là số cạnh nối với nó.
- Trong đồ thị có hướng:
  - *Bậc vào* (in-degree): số cung đi vào đỉnh.
  - *Bậc ra* (out-degree): số cung đi ra từ đỉnh.

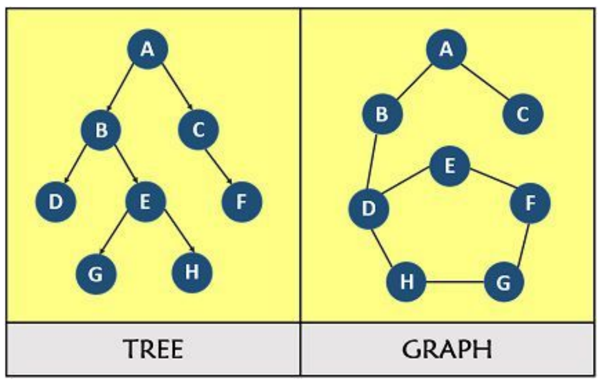
2.1.7 Đường đi đơn giản (Simple Path)

Một **đường đi đơn giản** là đường đi không đi qua một đỉnh nào hai lần (trừ khi là chu trình).

2.1.8 Cây (Tree)

Một **cây** là một đồ thị vô hướng liên thông và không có chu trình.
Tính chất quan trọng của cây:

- Với n đỉnh, cây có đúng $n - 1$ cạnh.
- Có duy nhất một đường đi giữa hai đỉnh bất kỳ.

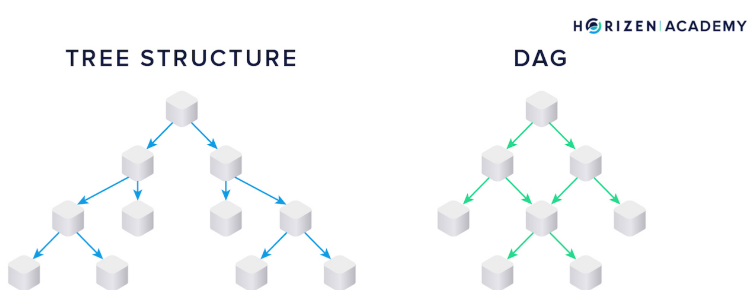


Hình 4: Phân biệt Cây và Đồ thị (không phải cây)

2.1.9 Đồ thị vô chu trình (Acyclic Graph).

Đồ thị gọi là **vô chu trình** nếu không tồn tại chu trình nào trong nó.

- Với đồ thị **vô hướng**, một đồ thị được xem là **Acyclic Graph** nếu không tồn tại dãy các đỉnh $v_1 \to v_2 \to \cdots \to v_k \to v_1$ với $k \geq 3$ và các cạnh liên tiếp nối các đỉnh đó.
- Một đồ thị vô hướng đơn giản gồm hai đỉnh được nối với nhau bằng một cạnh cũng là một **Acyclic Graph**, vì không tồn tại chu trình nào (phải có ít nhất 3 đỉnh để hình thành chu trình trong đồ thị vô hướng).
- Đồ thị có hướng vô chu trình gọi là **DAG** (Directed Acyclic Graph). Nghĩa là một đồ thị có hướng không chứa bất kỳ chu trình nào tuân theo chiều các cung.
- Cây là một DAG, nhưng không phải tất cả DAG đều là cây.



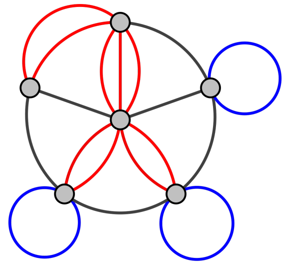
Hình 5: Phân biệt Cây và DAG

2.1.10 Đồ thị đơn (Simple Graph).

Đồ thị đơn là đồ thị không có *cạnh lặp* giữa cùng một cặp đỉnh và không có *khuyên* (loop – cạnh nối đỉnh với chính nó).

2.1.11 Đa đồ thị (Multigraph).

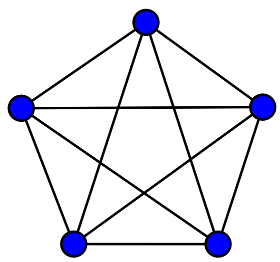
Đa đồ thị cho phép tồn tại *nhiều cạnh song song* giữa hai đỉnh và/hoặc khuyên. Thường dùng để mô hình hoá các mạng có nhiều kênh kết nối.



Hình 6: Minh họa Đa đồ thị

2.1.12 Đồ thị đầy đủ (Complete Graph).

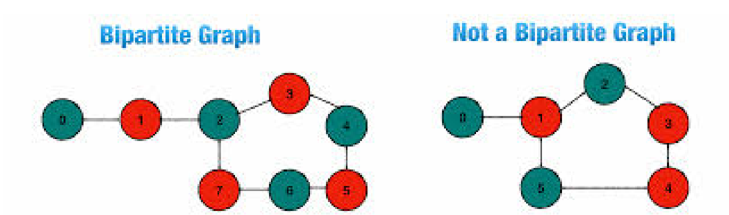
Đồ thị vô hướng K_n có n đỉnh, trong đó mọi cặp đỉnh phân biệt đều được nối bởi một cạnh.
Số cạnh là $\frac{n(n-1)}{2}$.
Đồ thị có hướng đầy đủ có $n(n-1)$ cung.



Hình 7: Minh họa đồ thị đầy đủ

2.1.13 Đồ thị hai phía (Bipartite Graph).

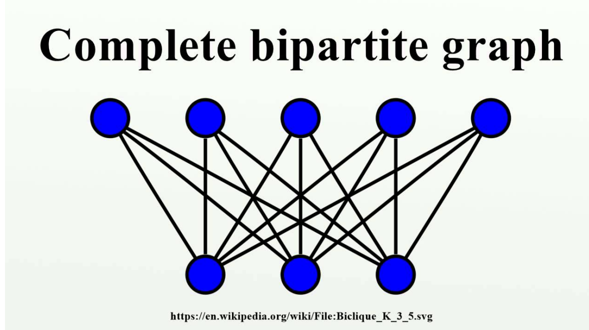
Một **đồ thị hai phía** là đồ thị mà tập đỉnh có thể chia thành hai tập rời U và V sao cho mọi cạnh đều nối một đỉnh từ U đến một đỉnh từ V.



Hình 8: Phân biệt đồ thị hai phía

2.1.14 Đồ thị hai phía đầy đủ (Complete Bipartite Graph).

Cho hai tập đỉnh rời U và V với $|U| = m$, $|V| = n$. Đồ thị $K_{m,n}$ chứa mọi cạnh nối một đỉnh của U với một đỉnh của V và *không có cạnh nội bộ* trong U hay V. Tổng số cạnh: $m\,n$.



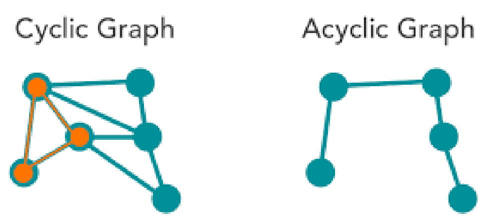
Hình 9: Minh họa đồ thị hai phía đầy đủ

2.1.15 Đồ thị vòng (Cycle Graph)

Đồ thị vòng ký hiệu C_n ($n \geq 3$) là đồ thị vô hướng gồm n đỉnh $\{v_1, v_2, \ldots, v_n\}$ và n cạnh

$$E = \{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}, \{v_n, v_1\}\}.$$

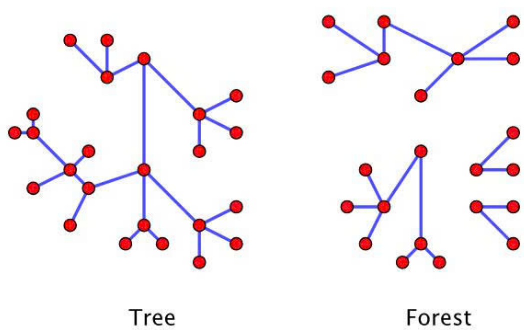
- Mỗi đỉnh có bậc 2 – C_n là đồ thị 2-chính quy.
- C_n chứa đúng một chu trình đơn giản độ dài n.
- C_n là **bipartite** khi và chỉ khi n chẵn.
- Số cạnh $m = n$; số đỉnh n; đường kính (diameter) $\text{diam}(C_n) = \lfloor n/2 \rfloor$.
- Tồn tại cây khung nhỏ nhất với trọng số tổng bằng $n - 1$ nếu mọi cạnh có trọng số 1.



Hình 10: Minh họa CG và AG

2.1.16 Rừng (Forest).

Một **rừng** là đồ thị vô hướng *không chứa chu trình* nhưng *không nhất thiết liên thông*. Mỗi thành phần liên thông của rừng là một **cây**. Nếu rừng có c cây và n đỉnh, nó có đúng $n - c$ cạnh.



Hình 11: Tree vs Forest

2.2 Thuật toán DFS

Thuật toán DFS (Depth-First Search – Duyệt theo chiều sâu) là một trong những thuật toán cơ bản để duyệt hoặc tìm kiếm trên đồ thị. Ý tưởng chính là xuất phát từ một đỉnh ban đầu, đi sâu theo từng nhánh con của đồ thị cho đến khi không còn đỉnh nào có thể đi tiếp, sau đó quay lui để khám phá các nhánh khác.
DFS có thể được cài đặt đệ quy hoặc sử dụng ngăn xếp. Nó thường được dùng để:

- Kiểm tra tính liên thông của đồ thị
- Tìm thành phần liên thông
- Phát hiện chu trình
- Tìm đường đi trong mê cung hoặc đồ thị

2.2.1 Ý tưởng cài đặt thuật toán DFS

DFS thường được cài đặt bằng đệ quy hoặc sử dụng ngăn xếp. Trong cài đặt đệ quy, ta cần một mảng đánh dấu để theo dõi các đỉnh đã được thăm nhằm tránh lặp vô hạn trong trường hợp đồ thị có chu trình.
Các bước cơ bản trong cài đặt DFS đệ quy:

1. Khởi tạo một mảng `visited[]` để đánh dấu các đỉnh đã được duyệt, với ý nghĩa: `visited[u] = true / false` nếu đỉnh u đã thăm / chưa thăm.
2. Gọi hàm `DFS(u)` tại đỉnh bắt đầu u.
3. Trong mỗi lần gọi:
   - Đánh dấu `visited[u] = true`.
   - Duyệt qua tất cả các đỉnh kề v của u:
     - Nếu v chưa được thăm (`visited[v] == false`), đệ quy gọi `DFS(v)`.

2.2.2 Cài đặt DFS sử dụng đệ quy trong C++

Listing 1: Thuật toán DFS sử dụng đệ quy

```cpp
#include <iostream>
#include <vector>
using namespace std;

const int MAXN = 100005; // So dinh toi da
vector<int> adj[MAXN];   // Danh sach ke
bool visited[MAXN];      // Mang danh dau

void DFS(int u) {
    visited[u] = true;
    cout << "Tham dinh: " << u << endl;
    for (int v : adj[u]) {
        if (!visited[v]) {
            DFS(v);
        }
    }
}
int main() {
    int n, m; // so dinh va so canh
    cin >> n >> m;
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u); // Neu la do thi vo huong
    }
    for (int i = 1; i <= n; i++) {
        visited[i] = false;
    }
    // Goi DFS tu dinh 1 (hoac 1 dinh bat ky)
    DFS(1);

    return 0;
}
```

Độ phức tạp: $\mathcal{O}(V + E)$ với V là số đỉnh, E là số cạnh.

2.2.3 Bài tập

2.2.4.1. MAKEMAZE Đề bài
Một mê cung hợp lệ là một mê cung có chính xác 1 lối vào và 1 lối ra và phải tồn tại ít nhất một đường đi thỏa mãn từ lối vào đến lối ra. Cho một mê cung, hãy chỉ ra rằng mê cung có hợp lệ hay không. Nếu có, in "valid", ngược lại in "invalid"
Input

Dòng đầu chứa một số nguyên t $(1 \leq t \leq 10^4)$ là số lượng test cases. Sau đó với mỗi test case, dòng đầu chứa 2 số nguyên m $(1 \leq m \leq 20)$ và n $(1 \leq n \leq 20)$, lần lượt là số lượng hàng và cột trong mê cung. Sau đó, là mê cung M với kích thước m x n. $M[i][j] =' \#'$ đại diện cho bức tường, $M[i][j] =' .'$ đại diện cho ô trống có thể đi vào được.

Output

Với mỗi test case, tìm xem mê cung tương ứng là "invalid" hay "valid"

Example

1
4 4
####
#...
#.##
#.##

→ "valid"

Phân tích bài toán

Vì mê cung chỉ có chính xác 1 lối vào và 1 lối ra. Trước hết ta cần kiểm tra biên ngoài của mê cung, nếu có chính xác 2 ô '.' thì có thể đó là một mê cung hợp lệ. Ngược lại (có ít hơn hoặc nhiều hơn 1 ô '.' ở biên) ta có thể khẳng định rằng đó không phải là một mê cung hợp lệ.

Khi ta đã có giả thuyết rằng mê cung là hợp lệ, ta cần 2 biến *start* và *target* lần lượt lưu lại tọa độ (x, y) của 2 ô ngoài biên. Áp dụng thuật toán DFS tại ô *start*, sau khi DFS nếu *visited[target.first][target.second]* $=$ *true* thì khẳng định rằng mê chung hợp lệ. Ngược lại, mê cung khônxg hợp lệ.

Listing 2: Cài đặt C++ bài MAKEMAZE

```cpp
#include <bits/stdc++.h>
#define int long long
#define endl "\n"
using namespace std;

const int MAXN = 21;
bool visited[MAXN][MAXN];

int dx[4] = {1, -1, 0, 0};
int dy[4] = {0, 0, 1, -1};

void dfs(pair<int, int> start, const vector<vector<char>> &a, int m, int n) {
    auto [x, y] = start;
    visited[x][y] = true;
    for (int i = 0; i < 4; i++) {
        int new_x = dx[i] + x;
        int new_y = dy[i] + y;
        if (new_x >= 1 && new_x <= m && new_y >= 1 && new_y <= n &&
            !visited[new_x][new_y] && a[new_x][new_y] == '.') {
            dfs({new_x, new_y}, a, m, n);
        }
    }
}

signed main() {
    int t; cin >> t;
    while (t--) {
        int m, n; cin >> m >> n;
        vector<vector<char>> a(m + 1, vector<char>(n + 1));
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                cin >> a[i][j];
                visited[i][j] = false;
            }
        }

        pair<int, int> start = {-1, -1}, target = {-1, -1};
        int cnt = 0;

        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if ((i == 1 || i == m || j == 1 || j == n) && a[i][j] == '.') {
                    if (start == make_pair(-1LL, -1LL)) start = {i, j};
                    else target = {i, j};
                    cnt++;
                }
            }
        }

        if (cnt != 2) {
```

```
            cout << "invalid" << endl;
            continue;
        } else {
            dfs(start, a, m, n);
            if (!visited[target.first][target.second]) cout << "invalid";
            else cout << "valid";
            cout << endl;
        }
    }
    return 0;
}
```

2.3 Topological Sorting

2.4 Khớp và Cầu (Joints and Brides)

2.5 Thành phần liên thông mạnh (Strongly Connected Components)

2.6 Thuật toán BFS

2.7 Thuật toán Dijkstra + Heap

2.8 Disjoint Set Union (DSU)

2.9 Maximum Flow and Maximum Matching

2.10 Minimum Cut

2.11 Euler Tour

2.12 Lowest Common Ancestor

2.13 Heavy Light Decomposition

2.14 Centroid Decomposition

2.15 2-SAT

3 Miscellaneous

3.1 Contributors