**ĐẠI HỌC QUỐC GIA HÀ NỘI**

**Trường Đại học Khoa học Tự nhiên**

**Khoa Toán – Cơ – Tin học**

****

**BÁO CÁO SƠ BỘ**

***Đề tài:***

THUẬT TOÁN TÌM KIẾM CHIỀU SÂU DFS

**Môn:** Cấu trúc dữ liệu và thuật toán

**Giảng viên:** Nguyễn Thị Hồng Minh

**Thành viên:** Mai Việt Hưng

Phùng Thị Mai Anh

Lê Minh Hiếu

Đỗ Minh Hiếu

*Hà Nội, 2022*

**MỤC LỤC**

[MỞ ĐẦU 2](#_Toc104230197)

[NỘI DUNG CHÍNH 3](#_Toc104230198)

[1. Definition : 3](#_Toc104230199)

[2. Implementation: 3](#_Toc104230200)

[3. Complexity (Độ phức tạp) 5](#_Toc104230201)

[4. Application: 6](#_Toc104230202)

[TỔNG KẾT 8](#_Toc104230203)

MỞ ĐẦU

Một vấn đề rất quan trọng trong Lý thuyết đồ thị là bài toán duyệt tất cả các đỉnh có thể đến được từ một đỉnh xuất phát nào đó, không duyệt lặp lại cũng như không bỏ sót đỉnh nào cả. Vì vậy, ta phải xây dựng được những phép duyệt các đỉnh của đồ thị theo một hệ thống nhất định, những phép duyệt đó gọi là các thuật toán tìm kiếm trên đồ thị.

Có hai giải thuật tìm kiếm trên đồ thị cơ bản: Tìm kiếm theo chiều sâu (Depth First Search - DFS) và Tìm kiếm theo chiều rộng (Breadth First Search - BFS). Hai giải thuật này có độ phức tạp thuật toán như nhau, nhưng sẽ có những ứng dụng khác nhau và cách cài đặt cũng khác nhau.

Sau đây là phần trình bày về đề tài giải thuật Tìm kiếm theo chiều sâu (Depth First Search) của nhóm đã thực hiện.

NỘI DUNG CHÍNH

1. Definition :

**Định nghĩa:** Giải thuật tìm kiếm theo chiều sâu (***Depth First Search*** – viết tắt là DFS), còn được gọi là giải thuật tìm kiếm ưu tiên chiều sâu, là giải thuật duyệt hoặc tìm kiếm trên một cây hoặc một đồ thị. Thuật toán khởi đầu tại gốc (hoặc chọn một đỉnh nào đó coi như gốc hay còn gọi là gốc của cây con) và duyệt đến lá của mỗi nhánh và lùi lại rồi duyệt các đỉnh tiếp theo.

*DFS trong tree:*

* có 3 thứ tự duyệt khác nhau sử dụng DFS trong tree: pre, post, inin

*DFS cho đồ thị*giống với DFS cho tree, điểm khác biệt là trong đồ thị thì có chu trình (sẽ có những node được ‘visit’ 2 lần). để tránh điều này thì ta đánh dấu mỗi node khi mà visit node đó.

1. Implementation:

* Từ một đỉnh đi sâu vào từng nhánh. Khi đã duyệt hết nhánh của đỉnh, ta lùi lại để duyệt đỉnh tiếp theo.
* Thuật toán dừng khi đi qua hết các đỉnh.

**Ý tưởng theo code:**

1. Đệ quy:

dfs(u)

for each neighbor v of u

if v unvisited

mark v visited

recursively call dfs(u)

1. Khử đệ quy(Dùng stack)

dsf(v)

    push(root)

    while stack not empty

vertex = pop()

if v visited => continue

    mark v visited

for each neighbor i of v

if i unvisited

push(i)

* Có 2 implement là có đệ quy và không có đề quy:

**DFS dùng đệ quy:**

* Chúng ta sẽ bắt đầu từ một node nhất định.
* Đánh dấu node đã được duyệt.
* Thăm node hiện tại.
* Đi các đỉnh liên kề mà chưa được đánh đánh dấu đã thăm.

public void **dfs**(**int** start) {

**boolean**[] isVisited = new **boolean**[adjVertices.size()];

    dfsRecursive(start, isVisited);

}

private void **dfsRecursive**(**int** current, **boolean**[] isVisited) {

    isVisited[current] = true;

    visit(current);

    for (**int** dest : adjVertices.get(current)) {

        if (!isVisited[dest])

            dfsRecursive(dest, isVisited);

    }

}

**DFS không dùng đệ quy (dùng Stack)**

* Chúng ta sẽ bắt đầu từ một node nhất định
* Xếp node bắt đầu vào ngăn xếp
* Nếu ngăn xếp trống:
  + Đánh dấu node vừa rồi đã được duyệt
  + Thăm các node hiện tại
  + Đưa các đỉnh liền kề chưa thăm vào stack

public void **dfsWithoutRecursion**(**int** start) {

    Stack<Integer> stack = new **Stack**<Integer>();

**boolean**[] isVisited = new **boolean**[adjVertices.size()];

    stack.push(start);

    while (!stack.isEmpty()) {

**int** current = stack.pop();

        if(!isVisited[current]){

            isVisited[current] = true;

            visit(current);

            for (**int** dest : adjVertices.get(current)) {

                if (!isVisited[dest])

                    stack.push(dest);

            }

    }

}

1. Complexity (Độ phức tạp)

***Thời gian:***

* Độ phức tạp về thời gian của DFS nếu toàn bộ tree được duyệt qua là O (V) trong đó V là số nút (đỉnh).
* Nếu đồ thị được biểu diễn dưới dạng danh sách kề:
* Ở đây, mỗi nút duy trì một danh sách tất cả các cạnh liền kề của nó. Giả sử rằng có V số nút(đỉnh) và E số cạnh trong đồ thị.
* Đối với mỗi nút, ta duyệt tất cả các nút lân cận của nó bằng cách duyệt qua danh sách kề của nó chỉ một lần trong thời gian tuyến tính.
* Đối với đồ thị có hướng, tổng kích thước của danh sách kề của tất cả các nút là E. Vì vậy, độ phức tạp về thời gian trong trường hợp này là O (V) + O (E) = O (V + E).
* Đối với đồ thị vô hướng, mỗi cạnh xuất hiện hai lần. Mỗi lần nằm trong danh sách kề của một trong hai đầu của cạnh. Độ phức tạp thời gian cho trường hợp này sẽ là O (V) + O (2E) ~ O (V + E).
* Nếu đồ thị được biểu diễn dưới dạng ma trận kề (mảng V x V):

Đối với mỗi nút, chúng ta sẽ phải duyệt qua toàn bộ một hàng có độ dài V trong ma trận để thăm tất cả các cạnh đi ra của nó. Chú ý rằng mỗi hàng trong ma trận kề tương ứng với một nút trong đồ thị và hàng đó lưu trữ thông tin về các cạnh xuất hiện từ nút. Do đó, độ phức tạp thời gian của DFS trong trường hợp này là O (V \* V) = O (V2).

***Nhận xét:*** Độ phức tạp về thời gian của DFS thực sự phụ thuộc vào cấu trúc dữ liệu được sử dụng để biểu diễn đồ thị.

***Không gian***

Vì DFS sử dụng một stack để theo dõi tới nút cuối cùng được duyệt nên trong trường hợp xấu nhất, stack có thể chiếm tối đa kích thước của các nút(đỉnh) trong đồ thị. Do đó, độ phức tạp không gian là O(V).

1. Application:

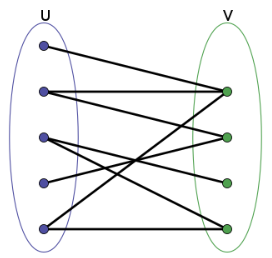
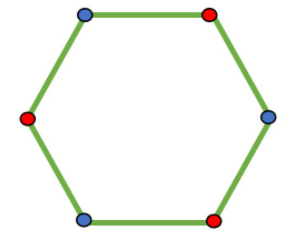
**Check bipartite graph:**

Kiểm tra xem đồ thị đã cho có phải là đồ thị 2 phía không?

Cho 1 đồ thị liên thông và kiểm tra xem có phải đồ thị 2 phía không?

**Định nghĩa:**

Đồ thị hai phía (đồ thị lưỡng phân hay đồ thị hai phần) ([bipartite graph](https://vi.wikipedia.org/w/index.php?title=Bipartite_graph&action=edit&redlink=1)) là một [đồ thị](https://vi.wikipedia.org/wiki/%C4%90%E1%BB%93_th%E1%BB%8B_(to%C3%A1n_h%E1%BB%8Dc)) đặc biệt, trong đó [tập](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_h%E1%BB%A3p) các [đỉnh](https://vi.wikipedia.org/wiki/%C4%90%E1%BB%89nh_(l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B)) có thể được chia thành hai [tập không giao nhau](https://vi.wikipedia.org/wiki/Ph%C3%A9p_giao#T%E1%BA%ADp_h%E1%BB%A3p_kh%C3%B4ng_giao_nhau) thỏa mãn điều kiện không có [cạnh](https://vi.wikipedia.org/wiki/C%E1%BA%A1nh_(l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B)) nối hai đỉnh bất kỳ thuộc cùng một tập.



**Ý tưởng:**

Thực hiện tô màu đồ thị bằng 2 màu khác nhau sao cho các đỉnh trong một tập hợp được tô cùng màu. Tức là 2 đỉnh kề nhau bất kỳ sẽ được tô 2 màu khác nhau.

**Thuật toán:**

Sử dụng mảng color[] để lưu trữ 2 loại màu khác nhau kí hiệu 0 và 1 cho các đỉnh kề nhau với các vị trí mảng là kí hiệu của các đỉnh (0, 1, 2, 3, …)

* Sử dụng hàm DFS để duyệt từ bất kì đỉnh nào được chọn, kí hiệu đỉnh đó là v.
* DFS sẽ tìm kiếm các đỉnh kề với v. Gọi u là 1 trong các đỉnh kề đó, nếu u chưa được duyệt thì gán màu khác với v: color[u] = 1 - color[v]

Nếu tại bất kỳ điểm nào mà color[v] != color[u] thì đồ thị đã cho không phải đồ thị hai phía.

* Gọi lại DFS để duyệt các đỉnh được nối với u và lặp lại bước 1.

TỔNG KẾT

Thông qua quá trình học tập và thực hành cũng như sự giúp đỡ của thầy cô, nhóm em đã hoàn thành được báo cáo cuối kỳ với đề tài giải thuật tìm kiếm theo chiều sâu (DFS). Trong đó, nhóm đã trình bày về khái niệm kỹ thuật tìm kiếm theo chiều sâu, ví dụ, độ phức tạp và cách triển khai của nó trong ngôn ngữ lập trình java. Cùng với đó, ta cũng đã thấy các ứng dụng của thuật toán tìm kiếm theo chiều sâu, trong số đó ứng dụng tiêu biểu nhất là kiểm tra đồ thị hai phía.