Giải Thuật Lập Trình

Nơi tổng hợp và chia sẻ những kiến thức liên quan tới giải thuật nói chung và lý thuyết khoa học máy tính nói riêng.

‹ Các kĩ thuật xử lí bít I-- Bitwise tricks I • DFS, phân loại cung và sắp xếp Topo -- DFS, Arc Classification and Topological Sort ›

Đồ thị -- Introduction to Algorithmic Graph Theory

September 26, 2015 in Uncategorized | No comments

Trong bài này và loạt bài tiếp theo chúng ta sẽ làm quen với đồ thị và các thuật toán với đồ thị. Đồ thị là một đối tượng tổ hợp (combinatorial object) được nghiên cứu và ứng dụng rất nhiều trong thực tế (có lẽ hơi thừa khi viết điều này). Phần này chúng ta sẽ:

- Làm quen với các khái niệm cơ bản gắn với đồ thị
- Cách biểu diễn đồ thị trong máy tính để chúng ta có thể thao tác với nó
- Duyệt đồ thị theo chiều rộng (Breadth First Search)
- Duyệt đồ thị theo chiều sâu (Depth First Search)

Bạn đọc có thể bỏ qua các phần mà các bạn đã quen thuộc. Note [1] của Jeff Erickson vẫn là tài liệu tham khảo chính của bài này.

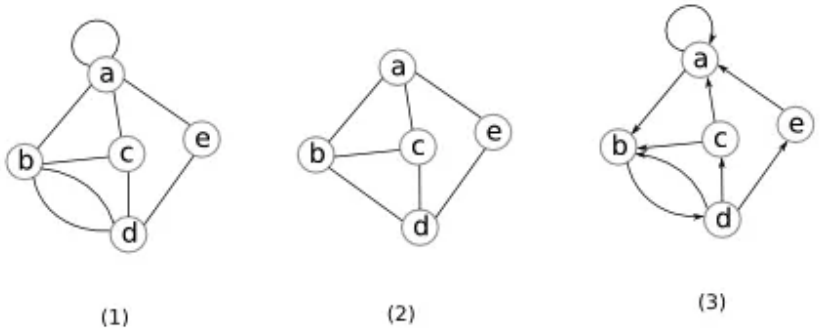
Các khái niệm

Một **đồ thị**, kí hiệu là $G(V, E)$, gồm hai thành phần:

1. Tập hợp V, bao gồm các đối tượng, được gọi là tập hợp các đỉnh (vertex) của đồ thị
2. Tập hợp $E \subseteq V^2$, bao gồm một cặp các đỉnh, được gọi là tập hợp các cạnh (vertex) của đồ thị

Ta sẽ kí hiệu n, m lần lượt là số đỉnh và số cạnh của đồ thị, i.e, $|V| = n, |E| = m$. Số đỉnh của đồ thị đôi khi ta cũng gọi là **bậc của đồ thị** (order of the graph).

Các đỉnh ta sẽ kí hiệu bằng các chữ in thường như u, v, x, y, z. Cạnh giữa hai đỉnh u, v có thể là **vô hướng** hoặc **có hướng**. Trong trường hợp đầu ta sẽ kí hiệu cạnh là uv, còn trong trường hợp sau ta sẽ kí hiệu là $u \to v$ để chỉ rõ hướng của cạnh là từ u đến v. Thông thường khi ta nói cạnh thì ta ám chỉ cạnh vô hướng còn với một cạnh có hướng ta sẽ gọi nó là một **cung** (arc). Hình $(1, 2)$ của hình dưới đây biểu diễn một đồ thị vô hướng (các cạnh là vô hướng) và hình (3) phải của hình dưới đây biểu diễn một đồ thị có hướng.



Trong hình (1), cạnh (a, a) được gọi là **cạnh lặp** (loop) và hai cạnh giữa cặp đỉnh (b, d) được gọi là hai **cạnh song song** (parallel edges). Một đồ thị được gọi là một **đơn đồ thị** (simple graph) nếu nó không có cạnh lặp và cạnh song song (hình (2)). Nếu một đồ thị không phải là đơn đồ thị thì chúng ta sẽ gọi nó là **đa đồ thị** (multigraph). Trong các loạt bài đồ thị ở đây, ta chủ yếu xét đơn đồ thị. Do đó, khi nói đồ thị ta sẽ ngầm hiểu là đơn đồ thị. Ta có:

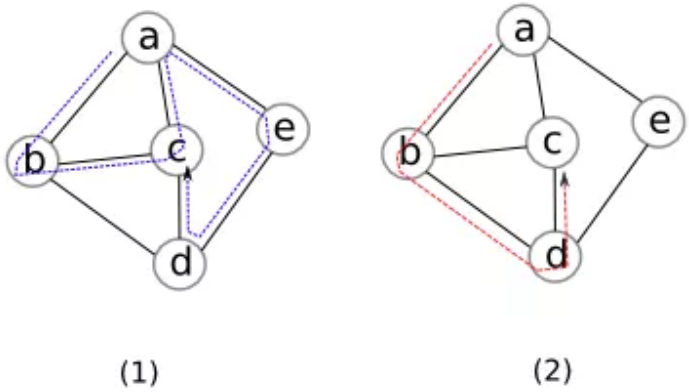
Fact 1: Nếu $G(V, E)$ là một đơn đồ thị vô hướng thì $m \leq \frac{n(n-1)}{2}$.

Nếu đồ thị $G(V, E)$ vô hướng, với mỗi cạnh uv, đỉnh v được gọi là kề (incident) với cạnh uv. Đỉnh u được gọi là hàng xóm (neighbor) của v. **Bậc** (degree) của đỉnh v, thường kí hiệu là $d(v)$, là số hàng xóm của đỉnh v. Nếu đồ thị $G(V, E)$ có hướng, với mỗi cung $u \to v$, đỉnh v được gọi là **đỉnh liền sau** (successor) của u và đỉnh u được gọi là **đỉnh liền trước** (predecessor) của v. **Bậc tới** (in-degree) của v là số đỉnh liền trước v và **bậc lui** (out-degree) của v là số đỉnh liền sau v. Ví dụ bậc tới của d trong hình (3) là 1 và bậc lui là 3.

Ta gọi $H(V_H, E_H)$ là **đồ thị con** (subgraph) của G nếu $V_H \subseteq V$ và $E_H \subseteq E$.

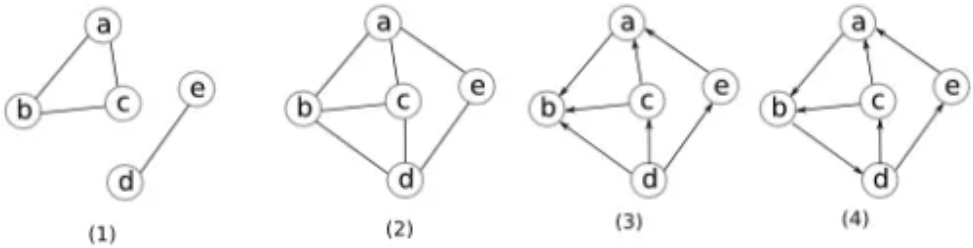
Một **đường đi** (walk) là một dãy các cạnh $\{e_1, e_2, \ldots, e_k\}$ trong đó hai cạnh liền kề bất kì e_i và e_{i+1} đều có chung một đỉnh.
Chú ý là đường đi có thể đi qua một đỉnh nhiều lần. Trong trường hợp mỗi đỉnh được thăm đúng 1 lần, ta sẽ gọi đó là **đường đi đơn** (path). Ví dụ trong đồ thì ở hình dưới đây, $\{ab, bc, ca, ae, ed, dc\}$ là một đường đi giữa a và c và $\{ab, bd, dc\}$ là một đường đi đơn giữa a và c. Một **đường đi đóng** (closed walk) là một đường đi bắt đầu và kết thục tại cùng một điểm. Một **chu trình** (cycle) là một đường đi đơn bắt đầu và kết thúc tại cùng một điểm. Có thể nói một chu trình là một đường đi đóng đi qua mỗi điểm đúng 1 lần ngoại trừ điểm đầu và điểm cuối. Các khái niệm vừa rồi nếu áp dụng

cho đồ thị có hướng thì ta sẽ thêm từ "có hướng" vào đằng trước.



(1) (2)

Một đồ thị vô hướng được gọi là **liên thông** (connected) nếu tồn tại một đường đi giữa mọi cặp điểm. Một đồ thị có hướng gọi là liên thông (yếu) nếu đồ thị vô hướng thu được từ đồ thị đó bằng cách bỏ qua hướng của cạnh là liên thông. Một đồ thị có hướng gọi là **liên thông mạnh** (strongly connected) nếu tồn tại một đường đi có hướng giữa mọi cặp điểm. Hiển nhiên nếu một đồ thị có hướng liên thông mạnh thì nó cũng liên thông yếu. Tuy nhiên điều ngược lại chưa chắc đúng (ví dụ?). Ví dụ đồ thị (1) dưới đây là không liên thông, đồ thị (2) liên thông, đồ thị (3) liên thông yếu (nhưng không mạnh) và đồ thị (4) liên thông mạnh.



Nếu một đồ thị (vô hướng) không liên thông, tập các đỉnh liên thông với nhau tạo thành một **thành phần liên thông** (connected component). Tương tự như vậy ta có thể định nghĩa thành phần liên thông (yếu hay mạnh) cho đồ thị có hướng. Một đồ thị **không có chu trình** (acyclic) thì ta gọ là một **rừng** (forest). Một rừng chỉ có một thành phần liên thông thì ta gọi nó là một **cây** (tree). Khái niệm cây và rừng có hướng tương tự như đồ thị có hướng.

> **Fact 2:** Nếu $G(V, E)$ là một cây thì $m = n - 1$. Nếu $G(V, E)$ là một rừng thì $m \leq n - 1$.

Có lẽ ta sẽ dừng định nghĩa khái niệm ở đây. Còn **rất rất nhiều** các khái niệm khác chúng ta sẽ định nghĩa khi mà chúng ta cần. Gần như tất cả các khái niệm cơ bản đã được liệt kê ở [2] mà bạn đọc có thể tham khảo thêm.

Trong phần tiếp theo, ta xét $G(V, E)$ vô hướng. Các thao tác với đồ thị có hướng có thể được mở rộng và áp dụng một cách tương tự.

Biểu diễn đồ thị

Chúng ta có thể biểu diễn đồ thị bằng một **ma trận kề** (adjacency matrix) A có kích thước $n \times n$ trong đó:

$$A[u,v] = \begin{cases} 1, & \text{if } uv \in E \\ 0, & \text{otherwise} \end{cases} \qquad (1)$$

Có thể thấy ngay là kích thước của cách biểu diễn này là $O(n^2)$ bất kể số lượng cạnh là nhiều hay ít. Theo Fact 1, số lượng cạnh m của một đồ thị có thể lên tới $O(n^2)$ cạnh (ta gọi là đồ thị dầy). Do đó, cách biểu diễn này có thể nói là phù hợp với đồ thị dầy. Tuy nhiên, nhiều đồ thị (đặc biệt các đồ thị thực tế như mạng xã hôi), số lượng cạnh $m = O(n)$ (ta gọi là đồ thị thưa). Do đó cách biểu diễn này khá tốn kém với đồ thị thưa.

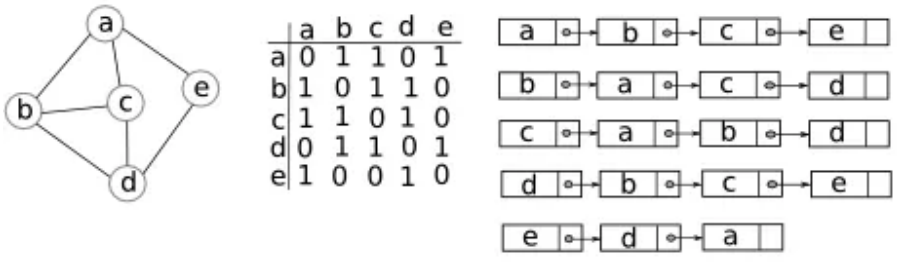
Để tiết kiệm bộ nhớ, với mỗi đỉnh $u \in V$, ta lưu trữ một danh sách các đỉnh kề với nó. Như vậy, đỉnh u cần một danh sách có $d(u)$ phần tử. Do đó tổng số phần tử của các danh sách là:

$$\sum_{u \in V} d(v) = 2m \qquad (2)$$

Tổng $\sum_{u \in V} d(v) = 2m$ là do mỗi cạnh được đếm hai lần trong tổng bậc của hai đỉnh kề với nó. Cách biểu diễn như trên gọi là biểu diễn bằng **danh sách kề** (adjacency list). Cách biểu diễn này phù hợp với cả đồ thị thưa. Mặc dù tiết kiệm bộ nhớ, cách biểu diễn này không phù hợp với một số thao tác của đồ thị. Bảng dưới đây so sánh hai cách biểu diễn vừa trình bày.

	Adjacency matrix	Adjacency list (linked list)
Space	$O(n^2)$	$O(n+m)$
Test $uv \in E$	$O(1)$	$O(1+\min(d(u),d(v)))$
List all neighbors of v	$O(n)$	$O(1+d(v))$
Add an edge uv	$O(1)$	$O(1)$
Delete an edge uv	$O(1)$	$O(d(u)+d(v)) = O(n)$

Ví dụ về hai cách biểu diễn đồ thị cho trong hình dưới đây:



Ta còn có thể kết hợp cách biểu diễn danh sách kề với một vài cấu trúc dữ liệu khác. Cụ thể, thay vì dùng danh sách liên kết để biểu diễn các đỉnh kề với một đỉnh u, ta còn có thể dùng bảng băm hoặc cấu trúc cây để biểu diễn. Trong khuôn khổ các bài viết ở đây, ta ít dùng (hoặc không dùng) các cấu trúc như vậy.

Ngoài ra ta có thể biểu diễn đồ thị bằng cách liệt kê tất cả các cặp (u,v) thỏa mãn $uv \in E$. Cách biểu diễn này có bộ nhớ là $O(m)$. Tuy nhiên, việc thực hiện các thao tác cơ bản trong cách biểu diễn này sẽ rất tốn kém. Đôi khi, ta có thể kết hợp cách biểu diễn này với cách biểu diễn danh sách

kề để tận dụng ưu thế của cả hai cách biểu diễn mà bộ nhớ vẫn là tuyến tính.

Duyệt đồ thị

Problem 1: Cho một đồ thị $G(V, E)$ và một đỉnh $s \in V$, in ra các đỉnh v thỏa mãn tồn tại một đường đi từ s tới v.

Ta gọi bài toán trên là bài toán duyệt đồ thị từ một đỉnh s. Để đơn giản, ta sẽ giả sử đồ thị là liên thông. Trường hợp đồ thị không liên thông sẽ được mở rộng ở cuối phần này. Cách thức chung để duyệt đồ thị như sau: Ta sẽ sử dụng 2 loại nhãn để gán cho các đỉnh của đồ thị: **chưa thăm** (unvisited) và **đã thăm** (visited). Ban đầu tất cả các đỉnh được đánh dấu là chưa thăm (unvisited). Ta sẽ duy trì một tập hợp C (thực thi tập C thế nào ta sẽ tìm hiểu sau), ban đầu khởi tạo rỗng. Ta sẽ thực hiện lặp 2 bước sau:

1. Lấy ra một đỉnh u trong C (thủ tục Remove(C) dưới đây).
2. Đánh dấu u là đã thăm (visited).
3. Đưa các hàng xóm của u có nhãn chưa thăm vào trong C. Thủ tục Add (C, v) dưới đây sẽ đưa đỉnh v vào trong tập C.

Thuật toán dừng khi $C = \emptyset$. Giả mã như sau:

```
GenericGraphTraverse(G(V, E), s):
    mark all vertices unvisited
    Add(C, s)
    while C ≠ ∅
        u ← Remove(C)          ≪ (*) ≫
        if u is unvisited
            mark u visited
            for all uv ∈ E and v is unvisited      ≪ (**) ≫
                Add(C, v)
```

Remark: Một đỉnh có thể được đưa nhiều lần vào tập C (do đó C không hẳn là tập hợp vì có nhiều phần tử giống nhau). Ví dụ xét 3 đỉnh u, v, w đôi một kề nhau. Đỉnh u được lấy ra từ C đầu tiên; đánh dấu u là đã thăm. Ngay sau đó, v và w sẽ được đưa vào C. Tiếp theo, lấy v ra khỏi C và đánh dấu v là đã thăm. Lúc này ta lại tiếp tục đưa w vào C một lần nữa vì theo giả mã trên, w là hàng xóm của v và có nhãn chưa thăm. Ở đây, ta sẽ **không** kiểm tra xem một đỉnh đã nằm trong C hay chưa trước khi đưa vào C.

Từ giả mã trên, ta thấy, tập C lưu các đỉnh *kề với ít nhất một đỉnh đã thăm*.

Phân tích thuật toán: Giả sử rằng ta sử dụng một cấu trúc để thực thi C sao cho việc thêm vào hoặc lấy một đỉnh bất kì (dòng $(*)$ và dòng cuối cùng) được thực hiện trong thời gian $O(1)$ (ví dụ nếu thực thi C bằng danh sách liên kết thì thêm vào hoặc lấy ra đỉnh ở đầu danh sách có thể được thực hiện trong thời gian $O(1)$). Ta có một vài nhận xét sau:

1. Các đỉnh đã được lấy ra khỏi C và bị đánh dấu là đã thăm thì nó sẽ không bao giờ được đưa trở lại tập C nữa.
2. Mỗi lần đỉnh v được đưa vào C, một hàng xóm của nó sẽ bị đánh dấu là đã thăm. Do đó, đỉnh v sẽ bị đưa vào C không quá $d(v)$ lần.
3. Mỗi khi lấy một đỉnh u ra khỏi C, ta sẽ duyệt qua tất cả các hàng xóm của u. Thao tác này mất thời gian $O(d(u))$. Theo nhận xét 1, phép duyệt này chỉ được thực hiện tối đa 1 lần.

Từ các nhận xét trên, ta suy ra tổng thời gian tính toán của thuật toán là $O(\sum_{u \in V} d(u)) = O(m)$.

Trong trường hợp đồ thị không liên thông, ta phải duyệt qua từng thành phần liên thông một. Do đồ thị có tối đa n thành phần liên thông, ta có:

> **Theorem 1:** Ta có thể duyệt qua đồ thị $G(V, E)$ trong thời gian $O(n + m)$.

Nếu ta thực thi C bằng danh sách liên kết thì có lẽ không có gì thú vị cả. Tuy nhiên, nếu ta thực thi C bằng hàng đợi (Queue) hoặc ngăn xếp (Stack) thì ta sẽ thu được một số tính chất thú vị từ đồ thị. Trường hợp ta thực thi C bằng hàng đợi, ta gọi thuật toán là **duyệt theo chiều rộng** (Breath First Search - BFS). Trường hợp ta thực thi C bằng ngăn xếp, ta gọi thuật toán là **duyệt theo chiều sâu** (Depth First Search - DFS). Sau đây ta sẽ thảo luận cả hai thuật toán.

Thuật toán duyệt theo chiều rộng BFS

Như đã nói ở trên, thuật toán BFS sẽ thực thi C bằng hàng đợi. Ta sẽ thay thủ tục ADD(C, v) bằng thủ tục ENQUEUE(C, v) và thủ tục REMOVE(C) bằng thủ tục DEQUEUE(C). Ngoài khung cơ bản như thuật toán ở trên, ta sẽ gán cho mỗi đỉnh v một nhãn $d[v]$. Giá trị của $d[v]$, như ta sẽ chỉ ra dưới đây, là khoảng cách ngắn nhất từ s tới v. Giả mã như sau:

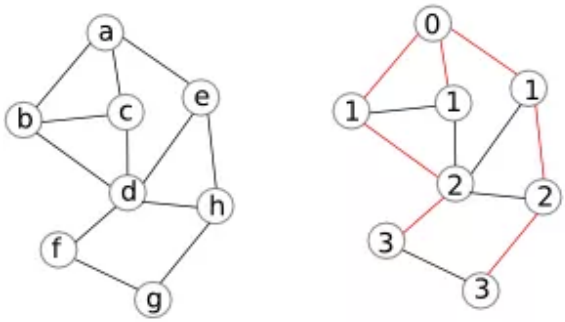
```
BFS(G(V, E), s):
    for each v ∈ V
        d[v] ← +∞
    C ← an empty Queue
    ENQUEUE(C, s)
    d[s] ← 0
    while C ≠ ∅
        u ← DEQUEUE(C)
        if u is unvisited
            mark u visited
            for all uv ∈ E and v is unvisited        ≪ (**) ≫
                ENQUEUE(C, v)
                d[v] ← d[u] + 1
```

Code của giả mã bằng C:

```
+ expand source (#)
```

Ví dụ ta thực thi thuật toán trên với đồ thị trong hình bên trái và kết qủa thu được trong hình bên phải. Các số ứng với các đỉnh tương ứng là nhãn

của các đỉnh đó. Những cạnh màu đỏ laf những cạnh mà v có nhãn unvisited ở dòng $(**)$ được thăm bởi BFS.



Theorem 2: Nhãn của mỗi đỉnh thu được sau khi duyệt BFS chính là khoảng cách ngắn nhất từ đỉnh xuất phát s tới đỉnh đó.

Chứng minh: Gọi mức (level) của một đỉnh u, kí hiệu là $level(u)$, là khoảng cách ngắn nhất từ đỉnh s tới u. Trong ví dụ trên, đỉnh b, c, e có mức 1, d, h có mức 2, ... Bằng quy nạp, ta có thể chứng minh rằng (coi như bài tập cho bạn đọc):

> **Claim:** Các đỉnh ở mức i sẽ được thăm sau các đỉnh ở các mức $1, 2, \ldots, i-1$.

Ta quy nạp trên biến $level$ để chứng minh rằng nhãn của v cũng chính là mức của v, i.e, $level(v) = d[v]$.

Xét một đỉnh v. Lần đầu tiên v được đưa vào hàng đợi C là khi ta thăm một đỉnh u nào đó kề với v và theo Claim trên, mức của u sẽ nhỏ hơn mức của v. Từ định nghĩa của mức ta suy ra $level(u) = level(v) - 1$. Theo giả thiết quy nạp $d[u] = level(u)$, ta suy ra $level(v) = d[u] + 1$. Theo giả mã, khi đưa v vào hàng đợi, ta cập nhật $d[v] = d[u + 1]$. Do đó, $d[v] = lelve(v)$, dpcm.

Kết hợp Theorm 1 và Theorem 2 ta có hệ quả sau:

Corollary 1: Trong thời gian $O(n + m)$, ta có thể tìm được khoảng cách ngắn nhất từ một đỉnh tới mọi đỉnh khác trong đồ thị vô hướng không có trọng số.

Corollary 1 có ý nghĩa rất lớn vì ta sẽ thấy (trong bài thuật toán Dijkstra (http://www.giaithuatlaptrinh.com/?p=764)), thuật toán tốt nhất tìm đường đi ngắn nhất với đồ thị có trong số có thời gian $O(n \log n)$ ngay cả trong đồ thị thưa (m = O(n)). Thuật toán BFS sẽ là thuật toán đơn giản và hiệu quả để tìm đường đi ngắn nhất trong đồ thị không có trong số.

Thuật toán duyệt theo chiều sâu DFS

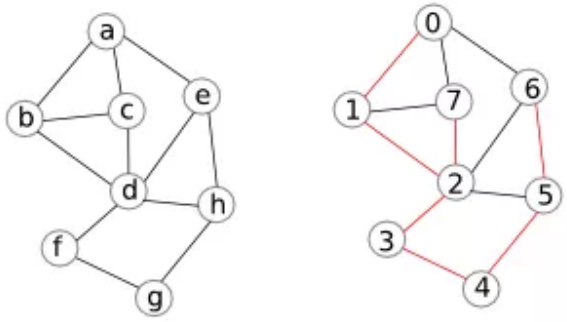
Trong duyệt theo chiều sâu DFS, ta thực thi C sử dụng ngăn xếp. Ta sẽ thay thủ tục Add(C, v) bằng thủ tục Push(C, v) và thủ tục Remove(C) bằng thủ tục Pop(C).

```
DFS(G(V, E), s):
    C ← an empty Stack
    PUSH(C, s)
    while C ≠ ∅
        u ← POP(C)
        if u is unvisited
            mark u visited
            for all uv ∈ E and v is unvisited        ≪ (**) ≫
                PUSH(C, v)
```

Code của giả mã bằng C:

```
+ expand source (#)
```

Nếu chỉ nhìn qua thi không thấy sự khác biệt qúa nhiều giữa DFS và thuật toán chung để duyệt đồ thị. Tuy nhiên bằng cách cập nhật thêm một vài thông tin trong quá trình duyệt đồ thị (giống như BFS), ta có thể phát hiện ra những tính chất rất thú vị của DFS. Ta sẽ thảo luận những tính chất đó ở bài sau. Hình dưới đây là một ví dụ thực thi DFS trên đồ thị. Số tương ứng của mỗi đỉnh ở bên phải là thứ tự của đỉnh thăm bởi DFS.



Ngoài thủ tục lặp DFS sử dụng ngăn xếp như trên, có lẽ một trong số chúng ta cũng khá quen thuộc với thủ tục thực thi DFS sử dụng đệ quy sau:

```
RECURSIVEDFS(s):
    mark s visited
    for all sv ∈ E and v is unvisited.
        RECURSIVEDFS(v)
```

Code của giả mã bằng

```
+ expand source (#)
```

Ta thấy cách thứ hai đơn giản hơn do không phải thực thi Stack. Tuy nhiên, cách này sẽ sử dụng nhiều Call Stack của máy tính và trong trường hợp độ sâu đệ quy lớn có thể gây ra Stack Overflow.

Phát hiện các thành phần liên thông

Một trong những ứng dụng đơn giản nhất để duyệt đồ thị là phát hiện ra các thành phần liên thông. Để phát hiện ra các thành phần liên thông của đồ thị (vô hướng), ta thực hiện lặp lại thao tác sau: chọn một đỉnh chưa thăm u và thực hiện thăm các đỉnh trong thành phần liên thông chứa u.

Thủ tục sau đây trả lại số thành phần liên thông của đồ thị đầu vào $G(V, E)$.

```
ConnecteComponents(G(V, E)):
    mark all vertices unvisited
    count ← 0
    for all vertices s ∈ V
        if s is unvisited
            GraphTraversal(G(V, E), s)        [[any graph traversal algorithm]]
            count ← count + 1
    return count
```

Code C:

+ expand source (#)

Code đầy đủ: list-representation (http://www.giaithuatlaptrinh.com/wp-content/uploads/2016/12/GraphBasics_List.c) , matrix-representation (http://www.giaithuatlaptrinh.com/wp-content/uploads/2016/12/GraphBasics_Matrix.c) .

Tham khảo

[1] Jeff Erickson, Graph Lecture Note (http://web.engr.illinois.edu/~jeffe/teaching/algorithms/notes/18-graphs.pdf) , UIUC.
[2] Diestel, Reinhard. *Graph theory. 2005.* Grad. Texts in Math (2005).
[3] Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford. *Introduction to Algorithms (2nd ed.)*, Chapter 23. MIT Press and McGraw-Hill (2001). ISBN 0-262-03293-7.

Thuật toán Kosaraju tìm thành phần liên thông mạnh -- Kosaraju's Algorithm (http://www.giaithua...p=1680)
December 7, 2016
In "graph algorithm"

Lý thuyết phổ đồ thị I -- Spectral Graph Theory I (http://www.giaithua...p=1358)
August 6, 2016
In "Laplacian"

DFS, phân loại cung và sắp xếp Topo -- DFS, Arc Classification and Topological Sort (http://www.giaithua...p=590)
October 5, 2015
In "dag"

Tags: bfs, connected component, dfs, graph basic, graph intro, graph representation

No comments

Comments feed for this article

Trackback link: http://www.giaithuatlaptrinh.com/wp-trackback.php?p=553

Reply

Your email address will not be published. Required fields are marked *

Your comment

Name *

Email *

Website

Post Comment

Notify me of follow-up comments by email.

Notify me of new posts by email.