**GRAPH (ĐỒ THỊ)**

1. **CÁC KHÁI NIỆM CƠ BẢN**
2. **Định nghĩa đồ thị**

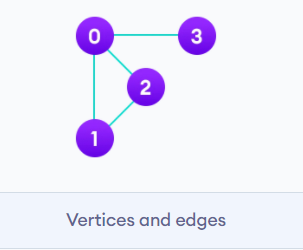
* **Đồ thị là gì?**
* Đồ thị là mô hình biểu diễn 1 tập các đối tượng và mối quan hệ 2 ngôi giữa các đối tượng

*Graph = Objects + Connections*

*G = (V, E)*

🡺 Đồ thị là một cấu trúc dữ liệu phi tuyến tính bao gồm các nút và các cạnh. Các nút đôi khi còn được gọi là các đỉnh và các cạnh là các đường hoặc cung nối hai nút bất kỳ trong đồ thị.

* Có thể định nghĩa đồ thị G là một cặp (V, E): G = (V, E). Trong đó V là tập các đỉnh (vertices) biểu diễn các đối tượng và E gọi là tập các cạnh (edges) biểu diễn mối quan hệ giữa các đối tượng. Chúng ta quan tâm tới mối quan hệ hai ngôi (pairwise relations) giữa các đối tượng nên có thể coi E là tập các cặp (u, v) với u và v là hai đỉnh của V biểu diễn hai đối tượng có quan hệ với nhau.

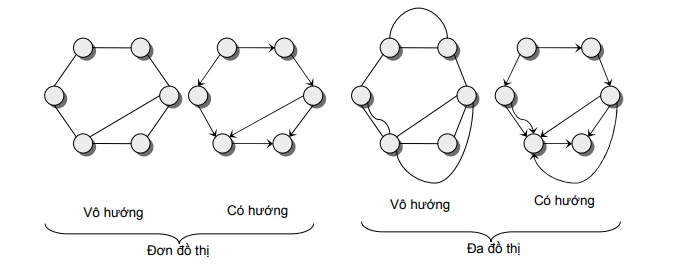


* *Trong đồ thị trên:* 
  + Tập đỉnh bao gồm 4 đỉnh: V = {0, 1, 2, 3}
* Tập cạnh bao gồm 4 cạnh, là các đường nối giữa các cặp đỉnh trong đồ thị

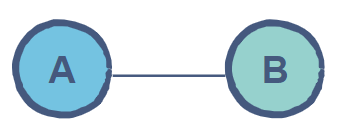
E = {(0,1), (0,2), (0,3), (1,2)}

* G = {V, E}
* **Phân loại đồ thị**

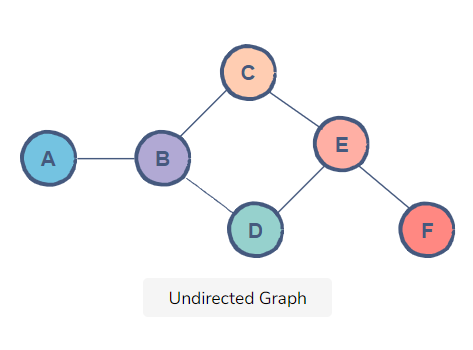
Có thể phân loại đồ thị G = (V, E) theo đặc tính và số lượng của tập các cạnh E:

****

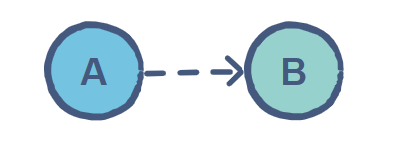
* **Đơn đồ thị:** G được gọi là đơn đồ thị (hay gọi tắt là đồ thị) nếu giữa hai đỉnh u, v € V có nhiều nhất là 1 cạnh trong E nối từ u tới v.
* **Đa đồ thị**: G được gọi là đa đồ thị (multigraph) nếu giữa hai đỉnh u, v € V có thể có nhiều hơn 1 cạnh trong E nối u và v (Hiển nhiên đơn đồ thị cũng là đa đồ thị). Nếu có nhiều cạnh nối giữa hai đỉnh u, v € V thì những cạnh đó được gọi là cạnh song song (parallel edges)
* **Vô hướng:**
* Trong ví dụ bên phải, đồ thị có thể được đi qua từ nút **A** để **B** cũng như từ nút **B** đến **A**.

****

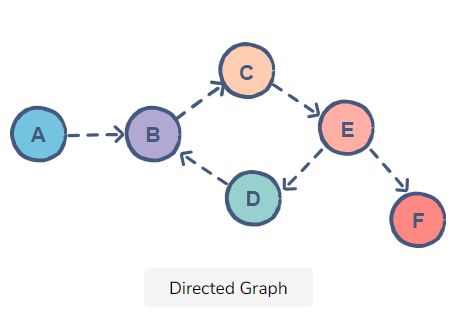
* + - G được gọi là đồ thị vô hướng (undirected graph) nếu các cạnh trong E là không định hướng, tức là cạnh nối hai đỉnh u, v € V bất kì cũng là cạnh nối hai đỉnh v, u. Hay nói cách khác, tập E gồm các cặp (u, v) không tính thứ tự: (u, v) = (v, u).

****

* **Có hướng:**
* Trong ví dụ ở bên phải, biểu đồ có thể được đi qua từ đỉnh **A** đến **B**, nhưng không phải từ đỉnh **B** đến **A**.

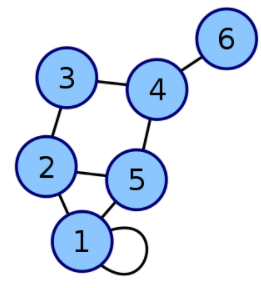


* + - G được gọi là đồ thị có hướng (directed graph) nếu các cạnh trong E là có định hướng, tức là có thể có cạnh nối từ đỉnh u tới đỉnh v nhưng chưa chắc đã có cạnh nối từ đỉnh v tới đỉnh u. Hay nói cách khác, tập E gồm các cặp (u, v) có tính thứ tự: (u, v) ≠ (v, u). Trong đồ thị có hướng, các cạnh còn được gọi là các cung (arcs). Đồ thị vô hướng cũng có thể coi là đồ thị có hướng nếu như ta coi cạnh nối hai đỉnh u, v bất kì tương đương với hai cung (u, v) và (v, u).

****

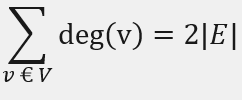
1. **Các khái niệm**

* **Cạnh liên thuộc:** Đối với đồ thị vô hướng G = (V, E). Xét một cạnh e € E, nếu e = (u, v) thì ta nói hai đỉnh u và v là kề nhau (adjacent) và cạnh e này liên thuộc (incident) với đỉnh u và đỉnh v.
* **Khuyên:** Cạnh có hai đầu trùng nhau (cùng một đỉnh).
  + - *Đồ thị có khuyên trên đỉnh 1*



* Một đồ thị được gọi là một **đơn đồ thị** (simple graph) nếu nó không có loop và cạnh song song. Nếu một đồ thị không phải là đơn đồ thị thì chúng ta sẽ goị nó là **đa đồ thị** (multigraph).
* **Đỉnh kề:** Hai đỉnh *u* và *v* được coi là *kề* nhau, ký hiệu *u* ↓ *v*, nếu có một cạnh nối chúng. Trong đồ thị ví dụ trên, các đỉnh 1 và 2 kề nhau, nhưng các đỉnh 2 và 4 không kề.
* **Bậc:**
* Với một đỉnh v trong đồ thị vô hướng, ta định nghĩa *bậc* (*degree*) của v, kí hiệu deg(v) là số cạnh liên thuộc với v. Trên đơn đồ thị thì số cạnh liên thuộc với v cũng là số đỉnh kề với v.

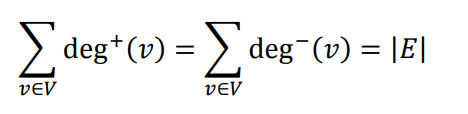
 *Giả sử* G = (V, E) *là đồ thị vô hướng, khi đó tổng tất cả các bậc đỉnh trong* V *sẽ bằng hai lần số cạnh:*



* + - **Chứng minh:** Khi lấy tổng tất cả các bậc đỉnh tức là mỗi cạnh e = (u, v) sẽ được tính một lần trong deg(u) và một lần trong deg(v). Từ đó suy ra kết quả.
    - Hệ quả: *Trong đồ thị vô hướng, số đỉnh bậc lẻ là số chẵn*
* Đối với đồ thị có hướng G = (V, E). Xét một cung e € E, nếu e = (u, v) thì ta nói u nối tới v và v nối từ u, cung e là đi ra khỏi đỉnh u và đi vào đỉnh v. Đỉnh u khi đó được gọi là đỉnh đầu, đỉnh v được gọi là đỉnh cuối của cung e.

🡺Với mỗi đỉnh v trong đồ thị có hướng, ta định nghĩa: *Bán bậc ra* (*out-degree*) của v kí hiệu deg+(v) là số cung đi ra khỏi nó; *bán bậc vào* (*in-degree*) kí hiệu deg–(v) là số cung đi vào đỉnh đó.

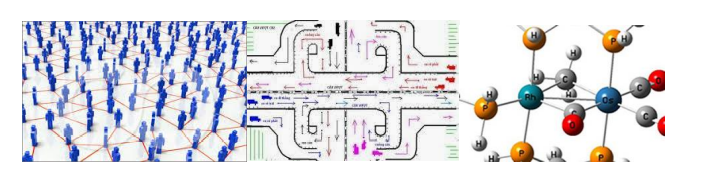
 *Giả sử* G = (V, E) *là đồ thị có hướng, khi đó tổng tất cả các bán bậc ra của các đỉnh bằng tổng tất cả các bán bậc vào và bằng số cung của đồ thị*

**

* **Chứng minh:** Khi lấy tổng tất cả các bán bậc ra hay bán bậc vào, mỗi cung (u, v) sẽ được tính đúng một lần trong deg+(u) và cũng được tính đúng một lần trong deg–(v). Từ đó suy ra kết quả.

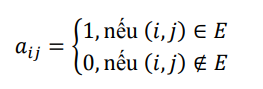
1. **Một vài ví dụ thực tế**

* Mạng xã hội Facebook: mỗi người dung (user) có thể được coi là một đỉnh trong đồ thị, mối quan hệ giữa những user có thể được coi là tập cạnh, dựa vào mối quan hệ đo, Facebook có thể sử dụng những thuật toán đồ thị để quản lý.
* Hay những ví dụ khác trực quan hơn như giao thông, các điểm nút giao thông là các đỉnh, các con đường nối các điểm nút sẽ tạo thành các cạnh, v.v, hay các phân tử, nguyên tử liên kết với nhau trong hóa học, v.v.

****

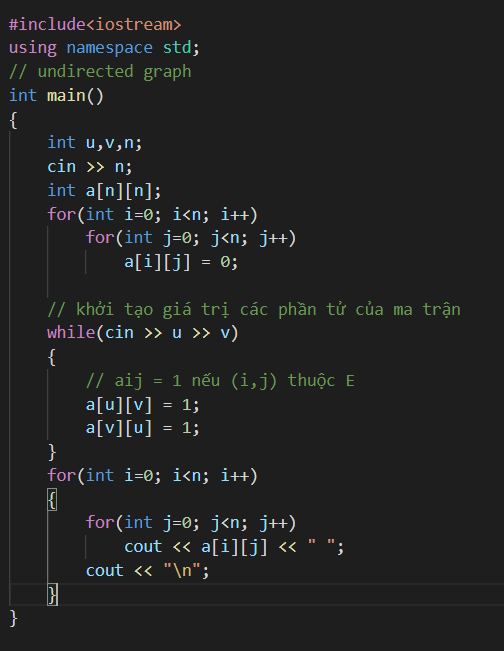
1. **BIỂU DIỄN ĐỒ THỊ TRÊN MÁY TÍNH**
2. **Ma trận kề**

* Ma trận kề là một mảng 2D gồm các đỉnh V x V. Mỗi hàng và cột đại diện cho một đỉnh, |V| = n.
* Đồ thị G có thể biểu diễn bằng ma trận vuông A = {aij}n×n. Trong đó: Nếu giá trị của bất kỳ phần tử aij nào là 1, nó thể hiện rằng có một cạnh nối đỉnh i và đỉnh j.

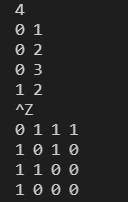
****

****

* **Xây dựng ma trận kề của đơn đồ thị vô hướng**

****

* **Ví dụ:**



**Ma trận kề có một số tính chất:**

* + Đối với đồ thị vô hướng G, thì ma trận kề tương ứng là ma trận đối xứng

aij = aji, điều này không đúng với đồ thị có hướng.

* + Nếu G là đồ thị vô hướng và A là ma trận kề tương ứng thì trên ma trận A, tổng các số trên hàng i bằng tổng các số trên cột i và bằng bậc của đỉnh i: deg (i)
  + Nếu G là đồ thị có hướng và A là ma trận kề tương ứng thì trên ma trận A, tổng các số trên hàng i bằng bán bậc ra của đỉnh i: deg+(i), tổng các số trên cột i bằng bán bậc vào của đỉnh i: deg–(i)

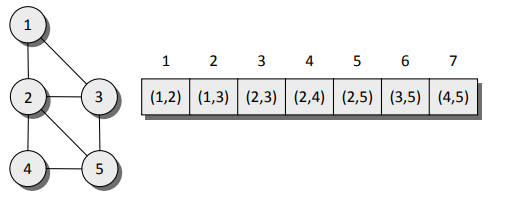
**Ưu điểm của ma trận kề:**

* + Đơn giản, trực quan, dễ cài đặt trên máy tính
  + Để kiểm tra xem hai đỉnh (u, v) của đồ thị có kề nhau hay không, ta chỉ việc kiểm tra bằng một phép so sánh: auv ≠ 0

**Nhược điểm của ma trận kề**

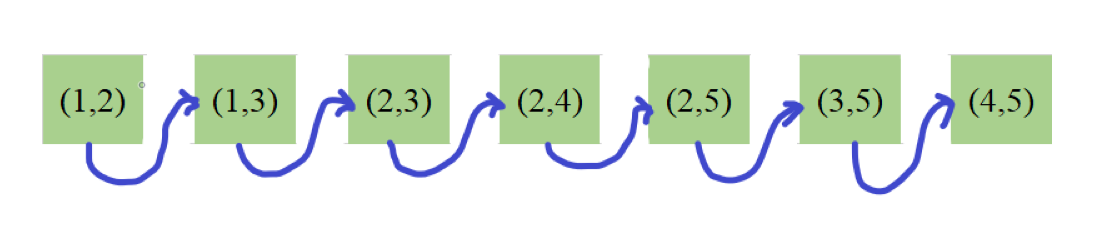
* + Bất kể số cạnh của đồ thị là nhiều hay ít, ma trận kề luôn luôn đòi hỏi n2 ô nhớ để lưu các phần tử ma trận, điều đó gây lãng phí bộ nhớ.
  + Một số bài toán yêu cầu thao tác liệt kê tất cả các đỉnh v kề với một đỉnh u cho trước. Trên ma trận kề việc này được thực hiện bằng cách xét tất cả các đỉnh v và kiểm tra điều kiện auv ≠ 0. Như vậy, ngay cả khi đỉnh u là *đỉnh cô lập* (không kề với đỉnh nào) hoặc *đỉnh treo* (chỉ kề với 1 đỉnh) ta cũng buộc phải xét tất cả các đỉnh v và kiểm tra giá trị tương ứng auv.

1. **Danh sách cạnh**

****

* + Với đồ thị G = (V, E) có n đỉnh, m cạnh, ta có thể liệt kê tất cả các cạnh của đồ thị trong một danh sách, mỗi phần tử của danh sách là một cặp (x, y) tương ứng với một cạnh của E, trong trường hợp đồ thị có hướng thì mỗi cặp (x, y) tương ứng với một cung, x là đỉnh đầu và y là đỉnh cuối của cung. Cách biểu diễn này gọi là *danh sách cạnh* (*edge list*)
  + Có nhiều cách xây dựng cấu trúc dữ liệu để biểu diễn danh sách, nhưng phổ biến nhất là dùng mảng hoặc danh sách móc nối.

*Danh sách móc nối:*

****

**Ưu điểm của danh sách cạnh:**

* + Trong trường hợp đồ thị thưa (có số cạnh tương đối nhỏ), cách biểu diễn bằng danh sách cạnh sẽ tiết kiệm được không gian lưu trữ, bởi nó chỉ cần O(m) ô nhớ để lưu danh sách cạnh.
  + Trong một số trường hợp, ta phải xét tất cả các cạnh của đồ thị thì cài đặt trên danh sách cạnh làm cho việc duyệt các cạnh dễ dàng hơn.

**Nhược điểm của danh sách cạnh:**

* + Nhược điểm cơ bản của danh sách cạnh là khi ta cần duyệt tất cả các đỉnh kề với đỉnh v nào đó của đồ thị, thì chẳng có cách nào khác là phải duyệt tất cả các cạnh, lọc ra những cạnh có chứa đỉnh v và xét đỉnh còn lại.
  + Việc kiểm tra hai đỉnh u, v có kề nhau hay không cũng bắt buộc phải duyệt danh sách cạnh, điều đó khá tốn thời gian trong trường hợp đồ thị dày (nhiều cạnh).

1. **Danh sách kề** *(adjacency list)*
   * Danh sách kề biểu thị một biểu đồ dưới dạng một mảng các danh sách được liên kết.
   * Chỉ số của mảng đại diện cho một đỉnh và mỗi phần tử trong danh sách liên kết của nó đại diện cho các đỉnh khác tạo thành một cạnh với đỉnh.



* + Bất cứ cấu trúc dữ liệu nào có khả năng biểu diễn danh sách (mảng, danh sách móc nối, cây…) đều có thể sử dụng để biểu diễn danh sách kề, nhưng mảng và danh sách móc nối được sử dụng phổ biến nhất.
  + Với đồ thị có hướng G = (V, E). V gồm n đỉnh và E gồm m cung.

Nhắc lại rằng khi sử dụng danh sách kề để biểu diễn đồ thị vô hướng, ta quy nó về đồ thị có hướng và số cung **m** được nhân đôi

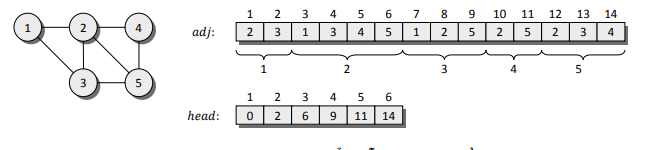
1. ***Biểu diễn danh sách kề bằng mảng***

Dùng một mảng adj[1…m] chứa các đỉnh, mảng được chia làm n đoạn, đoạn thứ u trong mảng lưu danh sách các đỉnh kề với đỉnh u. Để biết một đoạn nằm từ chỉ số nào đến chỉ số nào, ta có một mảng head[1 … n + 1] đánh dấu vị trí phân đoạn: head[u] sẽ bằng chỉ số đứng liền trước đoạn thứ u, quy ước head[n + 1] = m. Khi đó các phần tử trong đoạn:

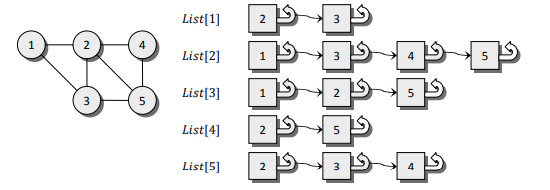
adj[head[u] + 1 … head[u + 1]]

là các đỉnh kề với đỉnh u.

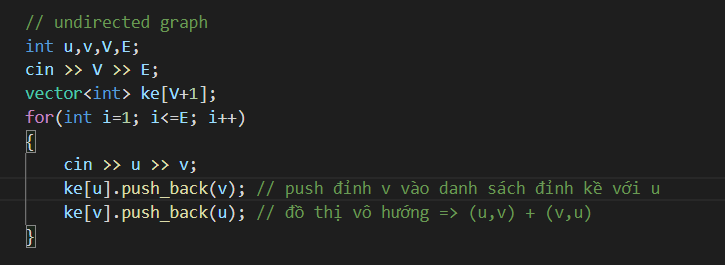
|  |
| --- |
|  |
|  |  |



1. ***Biểu diễn danh sách kề danh sách móc nối***



* + **Xây dựng danh sách kề bằng cách sử dụng mảng các vector**
* Ý tưởng của cách xây dựng này là: khởi tạo một mảng gồm n vector, mỗi vector ứng với 1 nút và chứa số hiệu của các nút liền kề với nút được xét.
* Chú ý: Trong trường hợp đồ thị có hướng thì chỉ có 1 chiều dịch chuyển trên mỗi cạnh (từ u tới v) nên sẽ không có lệnh gán u vào vector phần tử liền kề của v.
* Dưới đây là đoạn chương trình nhập vào danh sách các cạnh của đồ thị và chúng ta thực hiện xây dựng các danh sách ke[u] là danh sách các đỉnh kề với đỉnh u.



1. **CÁC THUẬT TOÁN TÌM KIẾM TRÊN ĐỒ THỊ**
2. **Khái niệm**

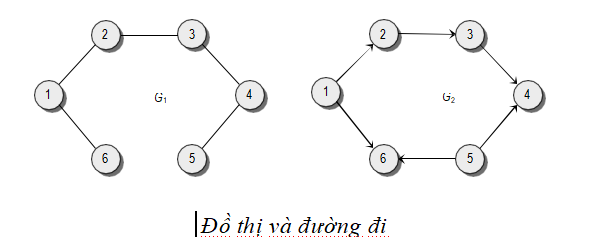
Cho đồ thì G = (, E) và 2 đỉnh s,t € V

* **Đường đi:**

Một dãy các đỉnh

P = (s = pO, p1, …, pk = t), ((pi–1, pi) € E)

được gọi là một đường đi từ s tới t, đường đi này gồm k + 1 đỉnh pO, p1, …, pk và k cạnh (pO, p1), (p1, p2), …, (pk–1, pk). Đỉnh s được gọi là đỉnh đầu và đỉnh t được gọi là đỉnh cuối của đường đi. Nếu tồn tại một đường đi từ s tới t, ta nói s đến được t và t đến được từ s: s ~ t.

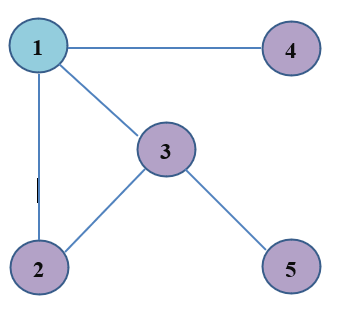


* **Chu trình:** Đường đi có đỉnh đầu trùng với đỉnh cuối (s = t) được gọi là chu trình.
* **Chu trình đơn:** Đường đi hay chu trình được gọi là đơn nếu như không có cạnh nào lặp lại.

1. **Tìm kiếm theo chiều sâu (depth first search)**
   1. **Lý thuyết:** Thuật toán Depth First Search (DFS – Tìm kiếm theo chiều sâu) là một dạng thuật toán duyệt hoặc tìm kiếm trên cây hoặc đồ thị dựa theo ưu tiên chiều sâu. Có nghĩa là, xuất phát từ một nút, ta bắt đầu duyệt đến tận cùng từng nhánh tỏa ra từ nút đó rồi mới chuyển sang nhánh tiếp theo, rồi nút tiếp theo, v.v. …
   2. **Cài đặt đệ quy – quay lui**
2. **Bài toán cụ thể:**

Cho đồ thị như hình vẽ

Yêu cầu: Duyệt tất cả các đỉnh của đồ thị sao cho mỗi đỉnh viếng thăm đúng 1 lần.

****

1. **Ý tưởng**

Tư tưởng của thuật toán tìm kiếm theo chiều sâu (Depth-First Search – DFS) có thể trình bày như sau: Trước hết, dĩ nhiên đỉnh s đến được từ s, tiếp theo, với mọi cung (s, x) của đồ thị thì x cũng sẽ đến được từ s. Với mỗi đỉnh x đó thì tất nhiên những đỉnh y nối từ x cũng đến được từ s… Điều đó gợi ý cho ta viết một thủ tục đệ quy DFSVisit(u) mô tả việc duyệt từ đỉnh u bằng cách thăm đỉnh u và tiếp tục quá trình duyệt DFSVisit(v) với v là một đỉnh chưa thăm nối từ u.

Kĩ thuật đánh dấu được sử dụng để tránh việc liệt kê lặp các đỉnh: Khởi tạo avail[v] = True, mỗi lần thăm một đỉnh, ta đánh dấu đỉnh đó lại (avail[v] = False) để các bước duyệt đệ quy kế tiếp không duyệt lại đỉnh đó nữa.

Thực hiện đi sâu vào tính từ đỉnh mới nhất cho đến khi không còn đỉnh nào để đi sâu vào nữa, chương trình sẽ quay lui lại các đỉnh (ngã rẽ) trước đó để thực hiện đi sâu theo lối rẽ khác.

1. **Áp dụng DFS**

Giả sử input các cặp đỉnh là: 1 4 1 3 1 2 2 3 3 5

Với ví dụ trên, ta có cách duyệt:

1🡺 4, 1 🡺 2, 1 🡺 3, 3 🡺 5

* Thu được thứ tự đỉnh được duyệt là: 1,4,2,3,5

1. **Biểu diễn đồ thị**

Ở đây chúng ta sẽ biểu diễn đồ thị bằng danh sách kề. Do thông thường, input đề bài luôn đọc vào được tất cả các cạnh, nên sử dụng danh sách kề cho bài toán duyệt đồ thị là hợp lý nhất, để tổng số cạnh duyệt qua đúng bằng số cạnh nhập vào

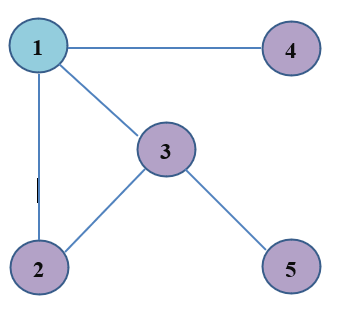
ke(1) = {4,3,2};

ke(2) = {1,3};

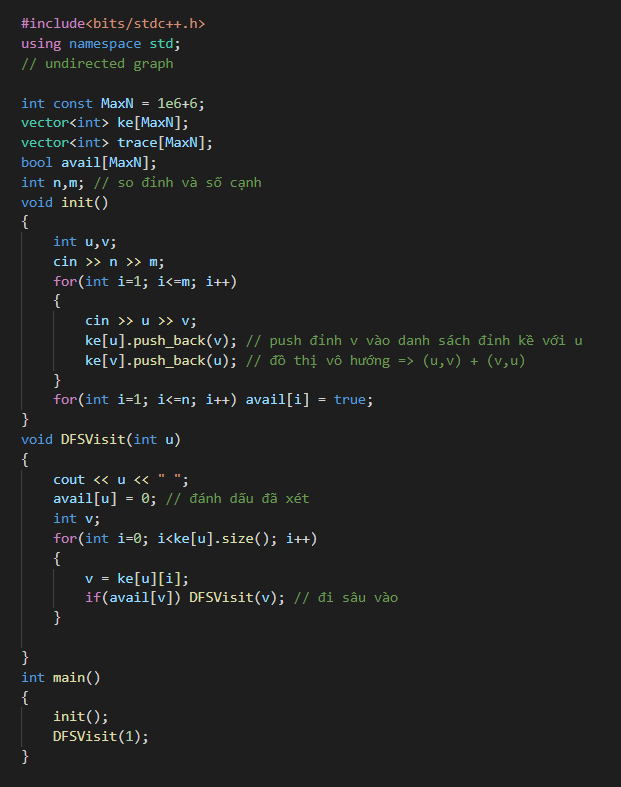
ke(3) = {1,2,5};

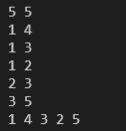
ke(4) = {1};

ke(5) = {3};

****

1. **Chương trình**

****

* **Kết quả: **

1. **Kiểm tra thuật toán**

|  |  |  |  |
| --- | --- | --- | --- |
| Các đỉnh có thể đi đến | Đỉnh duyệt | Đã duyệt avail[v] = false | Chưa duyệt avail[v] = true |
|  | () |  | 1,2,3,4,5 |
| 1,2,3,4,5 | DFSVisit (1) | 1 | 2,3,4,5 |
| 4,3,2 | DFSVisit (2) | 1,4 | 2,3,5 |
|  | DFSVisit (3) | 1,4,3 | 2,5 |
| 2 | DFSVisit (2) | 1,4,3,2 | 5 |
|  | DFSVisit (5) | 1,4,3,2,5 |  |

* 1. **Cài đặt không đệ quy (bằng stack)**

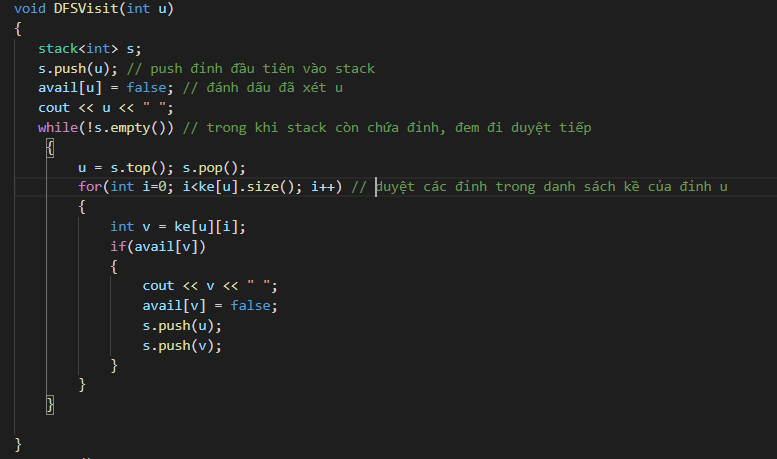
1. **Ý tưởng**

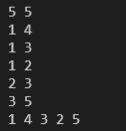
Trên tư tưởng khử đệ quy, mỗi khi pop() 1 đỉnh ra, nếu đỉnh đó còn có thể đi sâu vào ta phải thực hiện push lại đỉnh đó vào để lần pop phía sau có thể đem đỉnh đó rẽ nhánh sang nhánh khác – như tư tưởng quay lui

1. **1 đoạn chương trình (chỉ cần thay đổi hàm DFSVisit)**

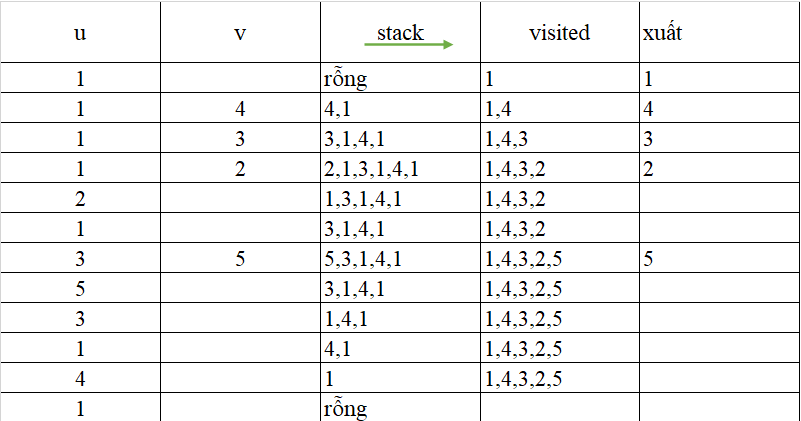
**// *sai rồi***

**// *thiếu break ở sau push(v)***

****

* **Kết quả: **

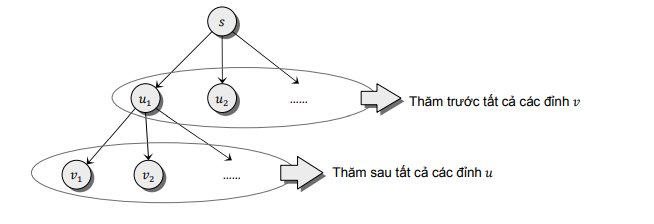
1. **Kiểm tra thuật toán**

****

* 1. **Độ phức tạp**
* Thời gian thực hiện giải thuật của DFS có thể đánh giá bằng số lần gọi thủ tục DFSVisit (|V| lần) cộng với số lần thực hiện của vòng lặp for bên trong thủ tục DFSVisit. Chính vì vậy:
* Đồ thị được biểu diễn bằng danh sách kề, vòng lặp for bên trong thủ tục DFSVisit (xét tổng thể cả chương trình) sẽ duyệt qua tất cả các cạnh của đồ thị (mỗi cạnh hai lần nếu là đồ thị vô hướng, mỗi cạnh một lần nếu là đồ thị có hướng). Trong trường hợp này, thời gian thực hiện giải thuật DFS là O (|V| + |E|)

1. **Tìm kiếm theo chiều rộng (breadth first search)**
   1. **Lý thuyết:** BFS (Breadth-First Search) là tên gọi cho thuật toán tìm kiếm trong đồ thị dựa theo ưu tiên chiều rộng. Có nghĩa là, xuất phát từ một nút, ta bắt đầu duyệt tất cả các nhánh ở tầng liền kề nó, rồi từ các nhánh đó duyệt tiếp tầng liền kề tiếp theo, cứ như vậy cho tới hết vùng liên thông.
   2. **Ý tưởng**

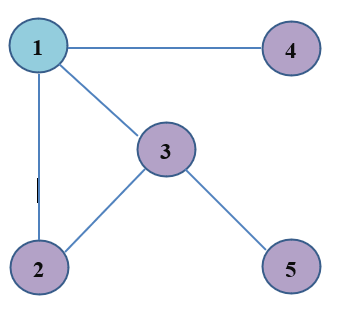
Tư tưởng của thuật toán tìm kiếm theo chiều rộng (Breadth-First Search – BFS) là “lập lịch” duyệt các đỉnh. Việc thăm một đỉnh sẽ lên lịch duyệt các đỉnh nối từ nó sao cho thứ tự duyệt là ưu tiên chiều rộng (đỉnh nào gần đỉnh xuất phát s hơn sẽ được duyệt trước). Đầu tiên ta thăm đỉnh s. Việc thăm đỉnh s sẽ phát sinh thứ tự thăm những đỉnh u1, u2, … nối từ s (những đỉnh gần s nhất). Tiếp theo ta thăm đỉnh u1, khi thăm đỉnh u1 sẽ lại phát sinh yêu cầu thăm những đỉnh v1, v2, … nối từ u1. Nhưng rõ ràng các đỉnh v này “xa” s hơn những đỉnh u nên chúng chỉ được thăm khi tất cả những đỉnh u đã thăm. Tức là thứ tự duyệt đỉnh sẽ là: s, u1, u2, …, v1, v2, …

****

Thuật toán tìm kiếm theo chiều rộng sử dụng một danh sách để chứa những đỉnh đang “chờ” thăm. Tại mỗi bước, ta thăm một đỉnh đầu danh sách, loại nó ra khỏi danh sách và cho những đỉnh chưa “xếp hàng” kề với nó xếp hàng thêm vào cuối danh sách. Thuật toán sẽ kết thúc khi danh sách rỗng.

* 1. **Cài đặt bằng queue**
  + **Cách cài đặt BFS:** Vì nguyên tắc vào trước ra trước, danh sách chứa những đỉnh đang chờ thăm được tổ chức dưới dạng hàng đợi (Queue). Ta có Queue là một hàng đợi với thủ tục Push(v) để đẩy một đỉnh v vào hàng đợi và hàm Pop trả về một đỉnh lấy ra từ hàng đợi
  + **Nguyên lý hoạt động:** Ban đầu, nút đầu tiên của đồ thị sẽ được truyền vào hàng đợi. Từ đây trở đi, mỗi khi tới một nút thì nút đó sẽ được xuất khỏi hàng đợi, đồng thời chèn vào hàng đợi những nút liền kề với nút đó mà chưa được duyệt qua. Do cấu trúc hàng đợi là LIFO – một đầu vào, 1 đầu ra, nên những đỉnh ở lớp trước sẽ được lấy ra và những đỉnh mới duyệt tới sẽ được cho vào phía sau hàng đợi, đảm bảo luôn duyệt đồ thị từ những lớp gần với đỉnh xuất phát hơn trước. Thuật toán kết thúc khi hàng đợi rỗng.
  + **Bài toán cụ thể:**

Cho đồ thị vô hướng không trọng số như hình vẽ. Với điểm xuất phát là 1, yêu cầu duyệt qua các đỉnh sao cho mỗi đỉnh đúng 1 lần bằng thuật toán BFS.

****

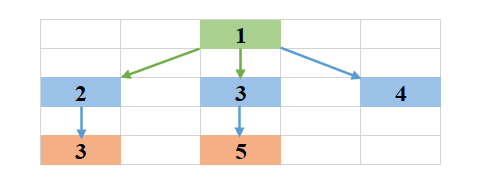
* + **Áp dụng thuật toán BFS**

1 🡺 2,3,4

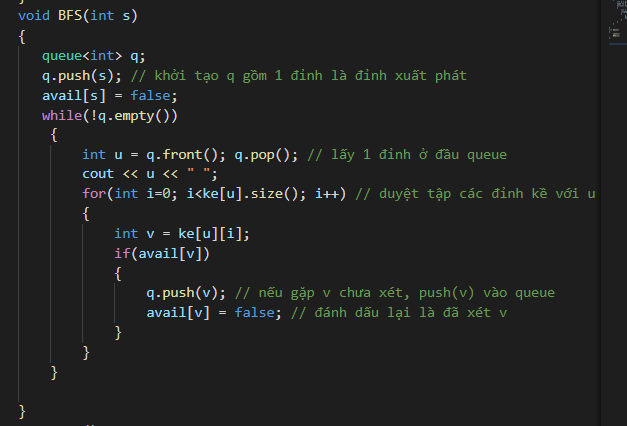
2 🡺 3

3 🡺 5

4 🡺 Ø



* + **Cài đặt thuật toán BFS:**
* Nhập dữ liệu biểu diễn đồ thị: cũng như DFS, ở đây chúng ta sẽ nhập vào các cặp cạnh (u, v) và biểu diễn đồ thị bằng danh sách kề.
* Cài đặt thuật toán BFS:
  + Đầu tiên khởi tạo hàng đợi chứa một đỉnh bắt đầu và đánh dấu đỉnh đó đã được xét, các đỉnh còn lại chưa xét
  + Mỗi bước thực hiện lấy một đỉnh ra khỏi hàng đợi và thực hiện duyệt đến các đỉnh kề với đỉnh đó mà chưa được xét, rồi thực hiện push lần lượt các đỉnh vào hàng đợi đồng thời đánh dấu đã xét.
  + Các bước lặp lại cho đến khi hàng đợi rỗng.
  + **Code**



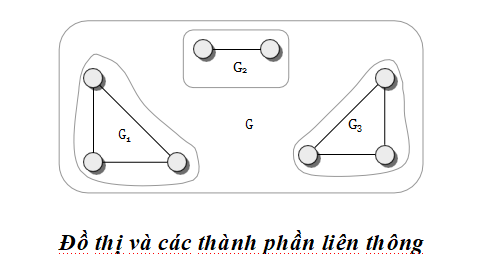
* 1. **Độ phức tạp**
  + Độ phức tạp của thuật toán: O (|V| + |E|), |V| là số đỉnh và |E| là số cạnh.
  + Độ phức tạp không gian của thuật toán là O(V)

1. **TÍNH LIÊN THÔNG CỦA ĐỒ THỊ**
2. **Định nghĩa trong đồ thị vô hướng**
   1. **Liên thông, thành phần liên thông**

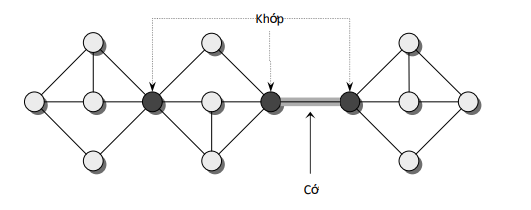
Đồ thị vô hướng G = (V, E) được gọi là *liên thông* (*connected*) nếu giữa mọi cặp đỉnh của G luôn tồn tại đường đi. Đồ thị chỉ gồm một đỉnh duy nhất cũng được coi là đồ thị liên thông.

Trong trường hợp đồ thị G không liên thông, ta có thể phân ra G thành một số đồ thị con liên thông mà chúng đôi một không có đỉnh chung. Mỗi đồ thị con như vậy là một thành phần liên thông của G. Như vậy, đồ thị liên thông khi và chỉ khi số thành phần liên thông của nó là 1.

Đối với đồ thị vô hướng, đường đi từ đỉnh u đến đỉnh v cũng giống như đỉnh đường đi từ đỉnh v đến đỉnh u. Chính vì vậy, nếu tồn tại u ∈V sao cho u có đường đi đến tất cả các đỉnh còn lại của đồ thị thì ta kết luận được đồ thị là liên thông.



* 1. **Cầu và khớp**



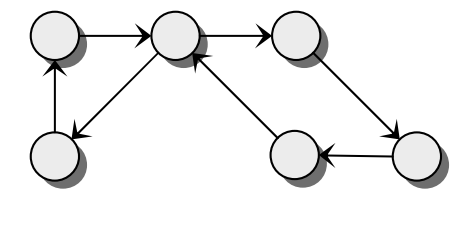
|  |
| --- |
|  |
|  |  |

|  |
| --- |
|  |
|  |  |

Đôi khi, việc xoá đi một đỉnh và tất cả các cạnh liên thuộc với nó sẽ tạo ra một đồ thị con mới có nhiều thành phần liên thông hơn đồ thị ban đầu, các đỉnh như thế gọi là *đỉnh cắt* (*cut vertices*) hay *nút khớp* (*articulation nodes*). Hoàn toàn tương tự, những cạnh mà khi ta bỏ nó đi sẽ tạo ra một đồ thị có nhiều thành phần liên thông hơn so với đồ thị ban đầu được gọi là *cạnh cắt* (*cut edges*) hay *cầu* (*bridges*).

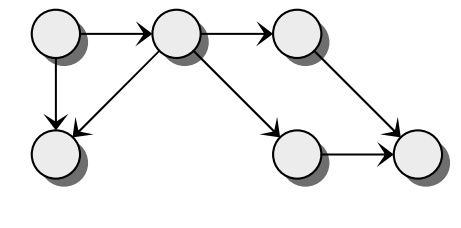
1. **Định nghĩa trong đồ thị có hướng**

* **Liên thông mạnh:** G gọi là *liên thông mạnh* (*strongly connected*) nếu luôn tồn tại đường đi (theo các cung định hướng) giữa hai đỉnh bất kì của đồ thị

****

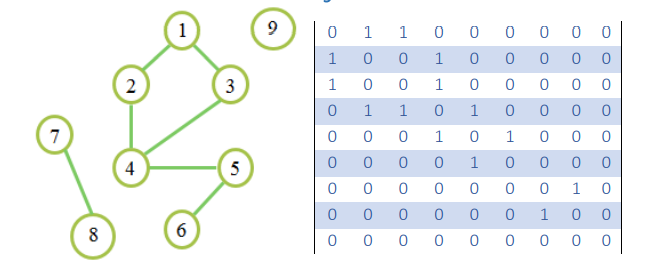
* **Liên thông yếu:**

G gọi là *liên thông yếu* (*weakly connected*) nếu phiên bản vô hướng của nó là đồ thị liên thông.

****

1. **Bài toán đếm thành phần liên thông (Dùng DFS)**
   1. **Bài toán:**

Cho đồ thị như hình vẽ xác định số thành phần liên thông

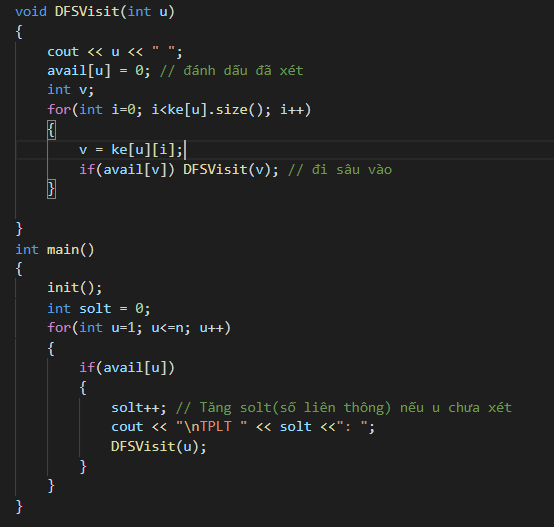
****

* 1. **Áp dụng DFS tìm thành phần liên thông**

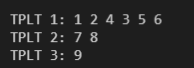
1. **Ý tưởng:**

Nếu đồ thị không liên thông (số thành phần liên thông lớn hơn 1) chúng ta có thể tách chúng thành những đồ thị con liên thông. Điều này cũng có nghĩa là trong phép duyệt đồ thị, số thành phần liên thông của nó đúng bằng số lần gọi tới thủ tục DFS (). Để xác định số các thành phần liên thông của đồ thị, chúng ta sử dụng thêm biến solt để ghi nhận các đỉnh của một thành phần thông.

1. **Code**

****

* **Kết quả**

****