HƯỚNG DẪN CÁC THUẬT TOÁN TÌM KIẾM TRÊN ĐỒ THỊ

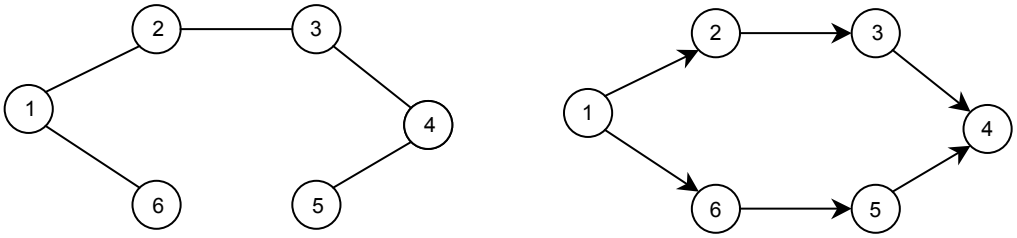
1. Bài toán

Cho đồ thị G = (V, E). u và v là hai đỉnh của G. Một **đường đi** (path) độ dài l từ đỉnh u đến đỉnh v là dãy $(u = x_0, x_1, ..., x_l = v)$ thoả mãn $(x_i, x_{i+1}) \in E$ với $\forall i: (0 \leq i < l)$.

Đường đi nói trên còn có thể biểu diễn bởi dãy các cạnh: $(u = x_0, x_1), (x_1, x_2), ..., (x_{l-1}, x_l = v)$

Đỉnh u được gọi là đỉnh đầu, đỉnh v được gọi là đỉnh cuối của đường đi. Đường đi có đỉnh đầu trùng với đỉnh cuối gọi là **chu trình** (Circuit), đường đi không có cạnh nào đi qua hơn 1 lần gọi là **đường đi đơn,** tương tự ta có khái niệm **chu trình đơn.**

Ví dụ: Xét một đồ thị vô hướng và một đồ thị có hướng dưới đây:



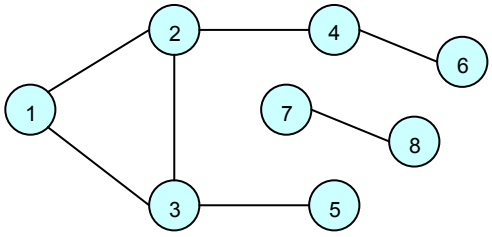
Trên cả hai đồ thị, (1, 2, 3, 4) là đường đi đơn độ dài 3 từ đỉnh 1 tới đỉnh 4. Bởi (1, 2) (2, 3) và (3, 4) đều là các cạnh (hay cung). (1, 6, 5, 4) không phải đường đi bởi (6, 5) không phải là cạnh (hay cung).

Một bài toán 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. Vấn đề này đưa về một bài toán liệt kê mà yêu cầu của nó là không được bỏ sót hay lặp lại bất kỳ đỉnh nào. Chính vì vậy mà ta phải xây dựng những thuật toán cho phép **duyệt một cách hệ thống** các đỉnh, những thuật toán như vậy gọi là những thuật toán **tìm kiếm trên đồ thị** và ở đây ta quan tâm đến hai thuật toán cơ bản nhất: **thuật toán tìm kiếm theo chiều**

sâu và **thuật toán tìm kiếm theo chiều rộng** cùng với một số ứng dụng của chúng.

Lưu ý:

1. Những cài đặt dưới đây là cho đơn đồ thị vô hướng, muốn làm với đồ thị có hướng hay đa đồ thị cũng không phải sửa đổi gì nhiều.
2. Dữ liệu về đồ thị sẽ được nhập từ file văn bản GRAPH.INP. Trong đó:
   - Dòng 1 chứa số đỉnh n (≤ 100), số cạnh m của đồ thị, đỉnh xuất phát S, đỉnh kết thúc F cách nhau một dấu cách.
   - m dòng tiếp theo, mỗi dòng có dạng hai số nguyên dương u, v cách nhau một dấu cách, thể hiện có cạnh nối đỉnh u và đỉnh v trong đồ thị.
3. Kết quả ghi ra file văn bản PATH.OUT
   - Danh sách các đỉnh có thể đến được từ S
   - Đường đi từ S tới F



GRAPH.INP	PATH.OUT
8 7 1 5	From 1 you can visit:
1 2	1, 2, 3, 5, 4, 6,
1 3	Path from 1 to 5:
2 3	5<-3<-2<-1
2 4	
3 5	
4 6	
7 8	

2. Thuật toán tìm kiếm theo chiều sâu

i. Cài đặt đệ quy

Tư tưởng của thuật toán có thể trình bày như sau: Trước hết, mọi đỉnh x kề với S tất nhiên sẽ đến được từ S. Với mỗi đỉnh x kề với S đó thì tất nhiên những đỉnh y kề với x cũng đến được từ S... Điều đó gợi ý cho ta viết một thủ tục đệ quy

DFS(u) mô tả việc duyệt từ đỉnh u bằng cách thông báo thăm đỉnh u và tiếp tục quá trình duyệt DFS(v) với v là một đỉnh chưa thăm kề với u.

- Để không một đỉnh nào bị liệt kê tới hai lần, ta sử dụng kỹ thuật đánh dấu, mỗi lần thăm một đỉnh, ta đánh dấu đỉnh đó lại để các bước duyệt đệ quy kế tiếp không duyệt lại đỉnh đó nữa
- Để lưu lại đường đi từ đỉnh xuất phát S, trong thủ tục DFS(u), trước khi gọi đệ quy DFS(v) với v là một đỉnh kề với u mà chưa đánh dấu, ta lưu lại vết đường đi từ u tới v bằng cách đặt TRACE[v] = u, tức là TRACE[v] lưu lại đỉnh liền trước v trong đường đi từ S tới v. Khi quá trình tìm kiếm theo chiều sâu kết thúc, đường đi từ S tới F sẽ là:

$$F \leftarrow p_1 = Trace[F] \leftarrow p_2 = Trace[p_1] \leftarrow ... \leftarrow S.$$

```
void DFS(u∈V)
{
  < 1. Thông báo tới được u >;
  < 2. Đánh dấu u là đã thăm (có thể tới được từ S)>;
  < 3. Xét mọi đỉnh v kề với u mà chưa thăm, với mỗi đỉnh v đó >;
    {
      Trace[v] = u;              // Lưu vết đường đi, đỉnh mà từ đó tới v là u
      DFS(v);                    // Gọi đệ quy duyệt tương tự đối với v
    }
}

void main()  // Chương trình chính
{
  < Nhập dữ liệu: đồ thị, đỉnh xuất phát S, đỉnh đích F >;
  < Khởi tạo: Tất cả các đỉnh đều chưa bị đánh dấu >;
  DFS(S);
  < Nếu F chưa bị đánh dấu thì không thể có đường đi từ S tới F >;
  < Nếu F đã bị đánh dấu thì truy theo vết để tìm đường đi từ S tới F
>;
}
```

* Thuật toán tìm kiếm theo chiều sâu đệ quy

```
  int a[max][max];              // Ma trận kề của đồ thị
  int Free[max];                // Free[v] = 0 ⇔ v chưa được thăm đến
  int Trace[max];               // Trace[v] = đỉnh liền trước v trên đường đi từ S tới v
  int n, S, F;                  // n: số đỉnh _ S: điểm xuất phát _ F: điểm kết thúc
void DFS(int u)                 // Thuật toán tìm kiếm theo chiều sâu bắt đầu từ đỉnh u
{
  printf("%d ",u);              // Thông báo tới được u
  Free[u] = 1;                  // Đánh dấu u đã thăm
  for (int v = 1; v<=n; v++)
    if (Free[v]==0 && a[u, v]>0)      // Với mỗi đỉnh v chưa thăm kề với u
      {
        Trace[v] = u;     // Lưu vết đường đi: Đỉnh liền trước v trong đường đi từ S tới v là u
        DFS(v);           // Tiếp tục tìm kiếm theo chiều sâu bắt đầu từ v
      }
}
```

Chú ý:

a) Vì có kỹ thuật đánh dấu, nên thủ tục DFS sẽ được gọi ≤ n lần (n là số đỉnh)

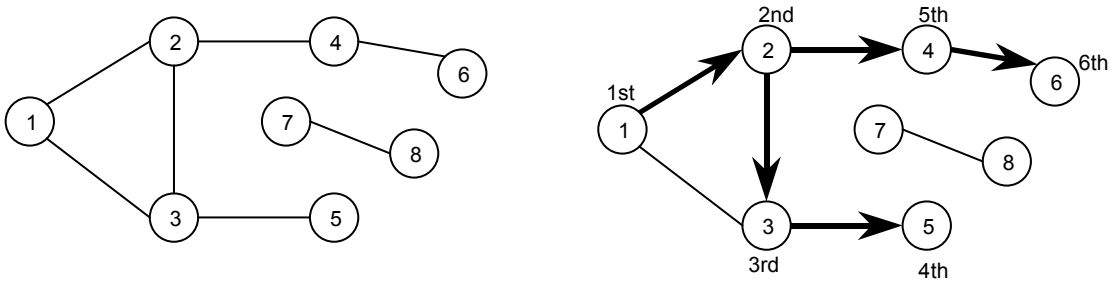
b) Đường đi từ S tới F có thể có nhiều, ở trên chỉ là một trong số các đường đi. Cụ thể là đường đi có thứ tự từ điển nhỏ nhất.

c) Có thể chẳng cần dùng mảng đánh dấu Free, ta khởi tạo mảng lưu vết Trace ban đầu toàn 0, mỗi lần từ đỉnh u thăm đỉnh v, ta có thao tác gán vết Trace[v] = u, khi đó Trace[v] sẽ khác 0. Vậy việc kiểm tra một đỉnh v là chưa được thăm ta có thể kiểm tra Trace[v] = 0. Chú ý: ban đầu khởi tạo Trace[S] = -1 (Chỉ là để cho khác 0 thôi).

```
void DFS(int u)                 // Cải tiến
{
  printf("%d ",u);
  for (int v = 1; v<=n; v++)
    if (Trace[v]==0 && a[u, v]>0)     // Trace[v] = 0 thay vì Free[v] = True

      {
```

```
            Trace[v] = u;      // Lưu vết cũng là đánh dấu luôn
            DFS(v);
        }
}
```

Ví dụ: Với đồ thị sau đây, đỉnh xuất phát S = 1: quá trình duyệt đệ quy có thể vẽ trên cây tìm kiếm DFS sau (Mũi tên u→v chỉ thao tác đệ quy: DFS(u) gọi DFS(v)).



Hình 1: Cây DFS

Hỏi: Đỉnh 2 và 3 đều kề với đỉnh 1, nhưng tại sao DFS(1) chỉ gọi đệ quy tới DFS(2) mà không gọi DFS(3) ?.

Trả lời: Đúng là cả 2 và 3 đều kề với 1, nhưng DFS(1) sẽ tìm thấy 2 trước và gọi DFS(2). Trong DFS(2) sẽ xét tất cả các đỉnh kề với 2 mà chưa đánh dấu thì dĩ nhiên trước hết nó tìm thấy 3 và gọi DFS(3), khi đó 3 đã bị đánh dấu nên khi kết thúc quá trình đệ quy gọi DFS(2), lùi về DFS(1) thì đỉnh 3 đã được thăm (đã bị đánh dấu) nên DFS(1) sẽ không gọi DFS(3) nữa.

Hỏi: Nếu F = 5 thì đường đi từ 1 tới 5 trong chương trình trên sẽ in ra thế nào ?.

Trả lời: DFS(5) do DFS(3) gọi nên Trace[5] = 3. DFS(3) do DFS(2) gọi nên Trace[3] = 2. DFS(2) do DFS(1) gọi nên Trace[2] = 1. Vậy đường đi là: 5 ← 3 ← 2 ←1.

Với cây thể hiện quá trình đệ quy DFS ở trên, ta thấy nếu dây chuyền đệ quy là: DFS(S) → DFS (u_1) → DFS(u_2) ... Thì thủ tục DFS nào gọi cuối dây chuyền sẽ được thoát ra đầu tiên, thủ tục DFS(S) gọi đầu dây chuyền sẽ được thoát cuối cùng. Vậy nên chăng, ta có thể mô tả dây chuyền đệ quy bằng một ngăn xếp (Stack).

ii. Cài đặt không đệ quy

Khi mô tả quá trình đệ quy bằng một ngăn xếp, ta luôn luôn để cho ngăn xếp lưu lại dây chuyền duyệt sâu từ nút gốc (đỉnh xuất phát S).

```
<Thăm S, đánh dấu S đã thăm>;
```

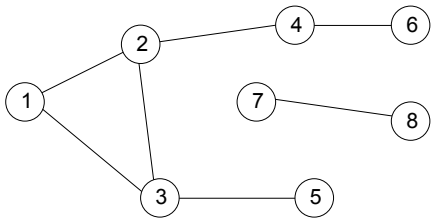
```
<Đẩy S vào ngăn xếp>;            // Dây chuyền đệ quy ban đầu chỉ có một đỉnh S
do {
  <Lấy u khỏi ngăn xếp>;         // Đang đứng ở đỉnh u
  if <u có đỉnh kề chưa thăm>
    {
      <Chỉ chọn lấy 1 đỉnh v, là đỉnh đầu tiên kề u mà chưa được
thăm>;
      <Thông báo thăm v>;
      <Đẩy u trở lại ngăn xếp>;        // Giữ lại địa chỉ quay lui
      <Đẩy tiếp v vào ngăn xếp>;   // Dây chuyền duyệt sâu được "nối" thêm v nữa
    }
  // Còn nếu u không có đỉnh kề chưa thăm thì ngăn xếp sẽ ngắn lại, tương ứng với quá trình lùi về
của dây chuyền DFS
} while <Ngăn xếp khác rỗng>;
```

*** Thuật toán tìm kiếm theo chiều sâu không đệ quy**

```
int a[max][max];              // Ma trận kề của đồ thị
int Free[max];                // Free[v] = 0 ⇔ v chưa được thăm đến
int Trace[max];               // Trace[v] = đỉnh liền trước v trên đường đi từ S tới v
int Stack[max];
int n, S, F, Last;                    // n: số đỉnh _ S: điểm xuất phát _ F: điểm kết thúc

void Push(int V)   // Đẩy một đỉnh V vào ngăn xếp
{
  Last++;
  Stack[Last] = V;
}

int Pop()                     // Lấy một đỉnh khỏi ngăn xếp, trả về trong kết quả hàm
{
  int x = Stack[Last];
  Last--;
  return x;
}
```

```
void DFS()
{
  int u, v;
  printf("%d ",S);
  Free[S] = 1;              // Thăm S, đánh dấu S đã thăm
  Push(S);
                    // Khởi động dây chuyền duyệt sâu
  do {
    // Dây chuyền duyệt sâu đang là S→ ...→ u
    u = Pop;
      // u là điểm cuối của dây chuyền duyệt sâu hiện tại
    for (v = 1; v<=n; v++)
      if (Free[v]==0 && a[u, v]>0)  // Chọn v là đỉnh đầu tiên chưa thăm kề với u, nếu có:
        {
          printf("%d ",v);
          Free[v] = 1;   // Thăm v, đánh dấu v đã thăm
          Trace[v] = u;
      // Lưu vết đường đi
          Push(u); Push(v);          // Dây chuyền duyệt sâu bây giờ là S→ ...→ u→ v
          break;
        }
  } while (Last != 0);   // Ngăn xếp không rỗng
}
```

Ví dụ: Với đồ thị dưới đây (S = 1), Ta thử theo dõi quá trình thực hiện thủ tục tìm kiếm theo chiều sâu dùng ngăn xếp và đối sánh thứ tự các đỉnh được thăm với thứ tự từ 1st đến 6th trong cây tìm kiếm của thủ tục DFS dùng đệ quy.



Trước hết ta thăm đỉnh 1 và đẩy nó vào ngăn xếp.

Bước lặp	Ngăn xếp	u	**v**	Ngăn xếp sau mỗi bước	Giải thích
1	(1)	1	**2**	(1, 2)	Tiến sâu xuống thăm 2

Bước lặp	Ngăn xếp	u	**v**	Ngăn xếp sau mỗi bước	Giải thích
2	(1, 2)	2	**3**	(1, 2, 3)	Tiến sâu xuống thăm 3
3	(1, 2, 3)	3	**5**	(1, 2, 3, 5)	Tiến sâu xuống thăm 5
4	(1, 2, 3, 5)	5	**Không có**	(1, 2, 3)	Lùi lại
5	(1, 2, 3)	3	**Không có**	(1, 2)	Lùi lại
6	(1, 2)	2	**4**	(1, 2, 4)	Tiến sâu xuống thăm 4
7	(1, 2, 4)	4	**6**	(1, 2, 4, 6)	Tiến sâu xuống thăm 6
8	(1, 2, 4, 6)	6	**Không có**	(1, 2, 4)	Lùi lại
9	(1, 2, 4)	4	**Không có**	(1, 2)	Lùi lại
10	(1, 2)	2	**Không có**	(1)	Lùi lại
11	(1)	1	**Không có**	∅	Lùi hết dây chuyền, Xong

Trên đây là phương pháp dựa vào tính chất của thủ tục đệ quy để tìm ra phương pháp mô phỏng nó. Tuy nhiên, trên mô hình đồ thị thì ta có thể có một cách viết khác tốt hơn cũng không đệ quy: Thử nhìn lại cách thăm đỉnh của DFS: Từ một đỉnh u, chọn lấy một đỉnh v kề nó mà chưa thăm rồi tiến sâu xuống thăm v. Còn nếu mọi đỉnh kề u đều đã thăm thì lùi lại một bước và lặp lại quá trình tương tự, việc lùi lại này có thể thực hiện dễ dàng mà không cần dùng Stack nào cả, bởi với mỗi đỉnh u đã có một nhãn Trace[u] (là đỉnh mà đã từ đó mà ta tới thăm u), khi quay lui từ u sẽ lùi về đó.

Vậy nếu ta đang đứng ở đỉnh u, thì đỉnh kế tiếp phải thăm tới sẽ được tìm như trong hàm FindNext dưới đây:

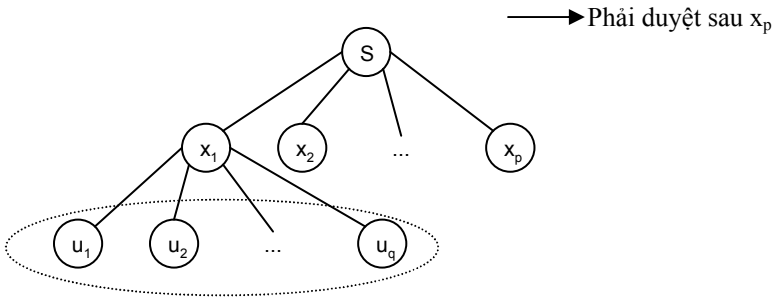
```
int FindNext(u∈V) // Tìm đỉnh sẽ thăm sau đỉnh u, trả về 0 nếu mọi đỉnh tới được từ S đều đã thăm
{
  do {
    for (∀v ∈ Kề(u))
      if <v chưa thăm>   // Nếu u có đỉnh kề chưa thăm thì chọn đỉnh kề đầu tiên chưa thăm để thăm tiếp}
        {
          Trace[v] = u; // Lưu vết
          return v;
        }
    u = Trace[u]; // Nếu không, lùi về một bước. Lưu ý là Trace[S] được gán bằng n + 1
```

```
  while (u < n + 1);
  return 0; // ở trên không Exit được tức là mọi đỉnh tới được từ S đã duyệt xong
}

void DFS()   // Thuật toán duyệt theo chiều sâu không đệ quy cải tiến
{
  Trace[S] = n + 1;
  <Khởi tạo các đỉnh đều là chưa thăm>
  u = S;
  do
    <Thông báo thăm u, đánh dấu u đã thăm>;
    u = FindNext(u);
  while (u > 0);
}
```

3. Thuật toán tìm kiếm theo chiều rộng

i. Cài đặt bằng hàng đợi

Cơ sở của phương pháp cài đặt này là "lập lịch" duyệt các đỉnh. Việc thăm một đỉnh sẽ lên lịch duyệt các đỉnh kề nó sao cho thứ tự duyệt là ưu tiên chiều rộng (đỉnh nào gần S hơn sẽ được duyệt trước). Ví dụ: Bắt đầu ta thăm đỉnh S. Việc thăm đỉnh S sẽ phát sinh thứ tự duyệt những đỉnh (x_1, x_2, ..., x_p) kề với S (những đỉnh gần S nhất). Khi thăm đỉnh x_1 sẽ lại phát sinh yêu cầu duyệt những đỉnh (u_1, u_2 ..., u_q) kề với x_1. Nhưng rõ ràng các đỉnh u này "xa" S hơn những đỉnh x nên chúng chỉ được duyệt khi tất cả những đỉnh x đã duyệt xong. Tức là thứ tự duyệt đỉnh sau khi đã thăm x_1 sẽ là: (x_2, x_3..., x_p, u_1, u_2, ..., u_q).



Hình 2: Cây BFS

Giả sử ta có một danh sách chứa những đỉnh đang "chờ" thăm. Tại mỗi bước, ta thăm một đỉnh đầu danh sách và cho những đỉnh chưa "xếp hàng" kề với nó xếp hàng thêm vào cuối danh sách. Chính vì nguyên tắc đó nên danh sách chứa những đỉnh đang chờ sẽ được tổ chức dưới dạng hàng đợi (Queue)

Ta sẽ dựng giải thuật như sau:

Bước 1: Khởi tạo:

- Các đỉnh đều ở trạng thái chưa đánh dấu, ngoại trừ đỉnh xuất phát S là đã đánh dấu
- Một hàng đợi (Queue), ban đầu chỉ có một phần tử là S. Hàng đợi dùng để chứa các đỉnh sẽ được duyệt theo thứ tự ưu tiên chiều rộng

Bước 2: Lặp các bước sau đến khi hàng đợi rỗng:

- Lấy u khỏi hàng đợi, thông báo thăm u (Bắt đầu việc duyệt đỉnh u)
- Xét tất cả những đỉnh v kề với u mà chưa được đánh dấu, với mỗi đỉnh v đó:
    1. Đánh dấu v.
    2. Ghi nhận vết đường đi từ u tới v (Có thể làm chung với việc đánh dấu)
    3. Đẩy v vào hàng đợi (v sẽ chờ được duyệt tại những bước sau)

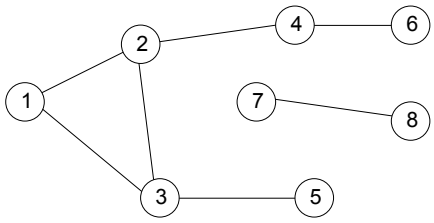
Bước 3: Truy vết tìm đường đi.

* Thuật toán tìm kiếm theo chiều rộng dùng hàng đợi

```
  int a[max][max];              // Ma trận kề của đồ thị
  int Free[max];                // Free[v] = 0 ⇔ v chưa được thăm đến
  int Trace[max];               // Trace[v] = đỉnh liền trước v trên đường đi từ S tới v
  int Queue[max];
  int n, S, F, First, Last;

void Push(int V)   // Đẩy một đỉnh V vào hàng đợi
{
  Last++;
  Queue[Last] = V;
}

int Pop()                   // Lấy một đỉnh khỏi hàng đợi, trả về trong kết quả hàm
{
  int x = Queue[First];
  First++;
  return x;
}

void BFS()          // Thuật toán tìm kiếm theo chiều rộng
{
  int u, v;

  Queue[1] = S;                                          // Hàng đợi chỉ gồm có một đỉnh S
  Last = 1;
  First = 1;

  do {
      u = Pop;      // Lấy một đỉnh u khỏi hàng đợi
      printf("%d ",u);              // Thông báo thăm u
      for (v = 1; v<=n; v++)
      if (Free[v]==0 && a[u, v]>0)
        {
          Push(v);          // Đưa v vào hàng đợi để chờ thăm
```

```
            Free[v] = 1;    // Thăm v, đánh dấu v đã thăm
            Trace[v] = u;  // Lưu vết đường đi: đỉnh liền trước v trong đường đi từ S là u
        }
    } while (First<=Last);          // Còn thực hiện  khi hàng đợi khác rỗng
```

Ví dụ: Xét đồ thị dưới đây, Đỉnh xuất phát S = 1.

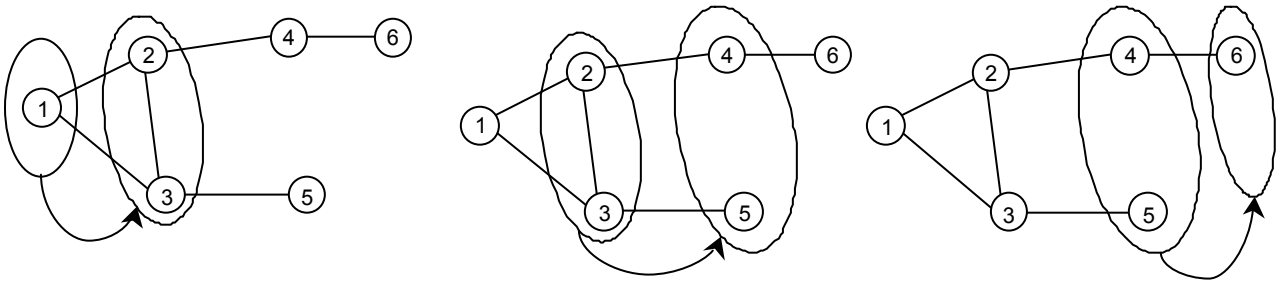


Hàng đợi	**Đỉnh u (lấy ra từ hàng đợi)**	Hàng đợi (sau khi lấy u ra)	Các đỉnh v kề u mà chưa lên lịch	Hàng đợi sau khi đẩy những đỉnh v vào
(1)	**1**	∅	2, 3	(2, 3)
(2, 3)	**2**	(3)	4	(3, 4)
(3, 4)	**3**	(4)	5	(4, 5)
(4, 5)	**4**	(5)	6	(5, 6)
(5, 6)	**5**	(6)	Không có	(6)
(6)	**6**	∅	Không có	∅

Để ý thứ tự các phần tử lấy ra khỏi hàng đợi, ta thấy trước hết là 1; sau đó đến 2, 3; rồi mới tới 4, 5; cuối cùng là 6. Rõ ràng là đỉnh gần S hơn sẽ được duyệt trước. Và như vậy, ta có nhận xét: nếu kết hợp lưu vết tìm đường đi thì **đường đi từ S tới F sẽ là đường đi ngắn nhất** (theo nghĩa qua ít cạnh nhất)

ii. Cài đặt bằng thuật toán loang

Cách cài đặt này sử dụng hai tập hợp, một tập "cũ" chứa những đỉnh "đang xét", một tập "mới" chứa những đỉnh "sẽ xét". Ban đầu tập "cũ" chỉ gồm mỗi đỉnh xuất phát, tại mỗi bước ta sẽ dùng tập "cũ" tính tập "mới", tập "mới" sẽ gồm những đỉnh chưa được thăm mà kề với một đỉnh nào đó của tập "cũ". Lặp lại công việc trên (sau khi đã gán tập "cũ" bằng tập "mới") cho tới khi tập cũ là rỗng:



Hình 3: Thuật toán loang

Giải thuật loang có thể dựng như sau:

Bước 1: Khởi tạo

Các đỉnh khác S đều chưa bị đánh dấu, đỉnh S bị đánh dấu, tập "cũ" Old := {S}

Bước 2: Lặp các bước sau đến khi Old = ∅

- Đặt tập "mới" New = ∅, sau đó dùng tập "cũ" tính tập "mới" như sau:
- Xét các đỉnh u ∈ Old, với mỗi đỉnh u đó:
  - Thông báo thăm u
  - Xét tất cả những đỉnh v kề với u mà chưa bị đánh dấu, với mỗi đỉnh v đó:
    - Đánh dấu v
    - Lưu vết đường đi, đỉnh liền trước v trong đường đi S→v là u
    - Đưa v vào tập New
- Gán tập "cũ" Old := tập "mới" New và lặp lại (có thể luân phiên vai trò hai tập này)

Bước 3: Truy vết tìm đường đi.

*** Thuật toán tìm kiếm theo chiều rộng dùng phương pháp loang**

```
void BFS()          // Thuật toán loang
{
   int u, v;
   int d[2][max];  // Tập lưu giữ các điểm duyệt cả Old và New}
   int Old, New;
   int sl[2];      // Số lượng các đỉnh trong tập Old và New  tương ứng trong d[0] & d[1]

   Old = 0;
```

```
    New = 1;
    sl[Old] = 1;
    d[Old][0] = S;
    Free[S] = 1;

    do {
       sl[New] = 0;
       for (int i=0; i<sl[Old]; i++)
       {
          u = d[Old][i];
          for (v=1; v<=n; v++)   // Quét tất cả những đỉnh v chưa bị đánh dấu mà kề với u}
          if (Free[v]==0 && a[u, v]>0)
            {
               Free[v] = 1;      // Thăm v, đánh dấu v đã thăm
               Trace[v] = u;     // Lưu vết đường đi: đỉnh liền trước v trong đường đi từ S là u
               d[New][sl[New]] = v;   // Đưa v vào tập New
               sl[New]++;
          }
       }
       Old = 1 - Old;            // Luân phiên vai trò hai tập Old và New
       New = 1 - New;
    } while (sl[Old]>0);         // Cho tới khi không loang được nữa
}
```