Chương 3

Chiến lược giảm-để-trị (Decrease-and-conquer)

Nội dung

1. **Chiến lược giảm-để-trị**
2. **Sắp thứ tự bằng phương pháp chèn**
3. **Các giải thuật duyệt đồ thị**
4. **Sắp xếp tôpô**
5. **Giải thuật sinh các hoán vị từ một tập**

1. Chiến lược thiết kế giải thuật giảm-để-trị (Decrease-and-conquer)

- Kỹ thuật thiết kế giải thuật giảm-để-trị lợi dụng mối liên hệ giữa lời giải cho một thể hiện của một bài toán và lời giải cho một thể hiện nhỏ hơn của cùng một bài toán.
- Có ba biến thể của chiến lược này.
  - Giảm bởi một hằng số (decrease by a constant)
  - Giảm bởi một hệ số (decrease by a factor)
  - Giảm kích thước của biến (variable size decrease)
- Sắp thứ tự bằng phương pháp chèn (insertion sort) là một thí dụ điển hình của chiến lược giảm-để-trị.

Chiến lược thiết kế giải thuật giảm-để-trị (tt.)

- **Giải thuật tìm ước số chung lớn nhất của 2 số theo công thức gcd(m,n) = gcd(n, m mod n) cũng là thí dụ của chiến lược giảm-để-trị theo lối *giảm kích thước của biến*.**

```
Algorithm Euclid(m,n)
/* m,n : two nonnegative
integers m and n  */
while n<>0 do
   r := m mod n;
   m:= n;
   n:= r
endwhile
return m;
```

Thí dụ: m = 60 và n = 24

2) m = 60 và n = 24

3) m = 24 và n = 12

4) m = 12 và n = 0

Vậy 12 là ước số chung lớn nhất

Chiến lược thiết kế giải thuật giảm-để-trị (tt.)

- Tại mỗi bước của giải thuật duyệt đồ thị theo chiều sâu trước (DFS) hay duyệt theo bề rộng trước (BFS), giải thuật đánh dấu đỉnh đã được viếng và tiến sang xét các đỉnh kế cận của đỉnh đó.
- Hai giải thuật duyệt đồ thị này đã áp dụng kỹ thuật giảm-bớt-một (decrease-by-one), một trong 3 dạng chính của chiến lược Giảm-để-trị.

2. Sắp thứ tự bằng phương pháp chèn

Ý tưởng :

- **Xét một ứng dụng của kỹ thuật "giảm để trị" vào việc sắp thứ tự một mảng a[0..n-1]. Theo tinh thần của kỹ thuật, ta giả sử rằng bài toán nhỏ hơn: sắp thứ tự một mảng a[0..n-2] đã được thực hiện. Vấn đề là phải chèn phần tử a[n-1] vào mảng con đã có thứ tự a[0..n-2].**
- **Có hai cách để thực hiện điều này.**
  - **Một là ta duyệt mảng con đã có thứ tự từ trái sang phải cho đến khi tìm thấy phần tử đầu tiên lớn hơn hay bằng với phần tử a[n-1] và chèn phần tử a[n-1] vào bên trái phần tử này.**
  - **Hai là ta duyệt mảng con đã có thứ tự từ phải sang trái cho đến khi tìm thấy phần tử đầu tiên nhỏ hơn hay bằng với phần tử a[n-1] và chèn phần tử a[n-1] vào bên phải phần tử này.**

2. Sắp thứ tự bằng phương pháp chèn (tt.)

Cách thứ hai thường được chọn:

a[0] ≤ … ≤ a[j] < a[j+1] ≤ … ≤ a[i-1] | a[i] … a[n-1]

390	→ **205**	→ **182**	→ **45**	
45				
205	**390**	**205**	**182**	**182**
182	182	**390**	**205**	→ **205**
45	45	45	**390**	**235**
235	235	235	235	**390**

Giải thuật sắp thứ tự bằng phương pháp chèn

```
procedure insertion;
var i; j; v:integer;
begin
        for i:=2 to N do
        begin
                v:=a[i]; j:= i;
                while a[j-1]> v do
                begin
                    a[j] := a[j-1]; // pull down
                     j:= j-1 end;
                a[j]:=v;
        end;
end;
```

Những lưu ý về giải thuật insertion sort

- **Chúng ta dùng một trị khóa "cầm canh" (*sentinel*) tại *a[0]*, làm cho nó nhỏ hơn phần tử nhỏ nhất trong mảng.**

2. Vòng lặp ngoài của giải thuật được thực thi N-1 lần. Trường hợp xấu nhất xảy ra khi mảng đã có thứ tự đảo ngược. Khi đó, vòng lặp trong được thực thi với tổng số lần sau đây:

$$(N-1) + (N-2) + \ldots + 1 = N(N-1)/2 = O(N^2)$$

Số bước chuyển = N(N-1)/2 Số so sánh = N(N-1)/2

3. Trung bình có khoảng chừng (i-1)/2 so sánh được thực thi trong vòng lặp trong. Do đó, trong trường hợp trung bình, tổng số lần so sánh là:

$$(N-1)/2 + (N-2)/2 + \ldots + 1/2 = N(N-1)/4 = O(N^2)$$

Độ phức tạp của sắp thứ tự bằng phương pháp chèn

Tính chất 1.2: ***Sắp thứ tự bằng phương pháp chèn thực thi khoảng $N^2/2$ so sánh và $N^2/4$ hoán vị trong trường hợp xấu nhất*.**

Tính chất 1.3: ***Sắp thứ tự bằng phương pháp chèn thực thi khoảng $N^2/4$ so sánh và $N^2/8$ hoán vị trong trường hợp trung bình*.**

Tính chất 1.4: ***Sắp thứ tự bằng phương pháp chèn có độ phức tạp tuyến tính đối với một mảng đã gần có thứ tự.***

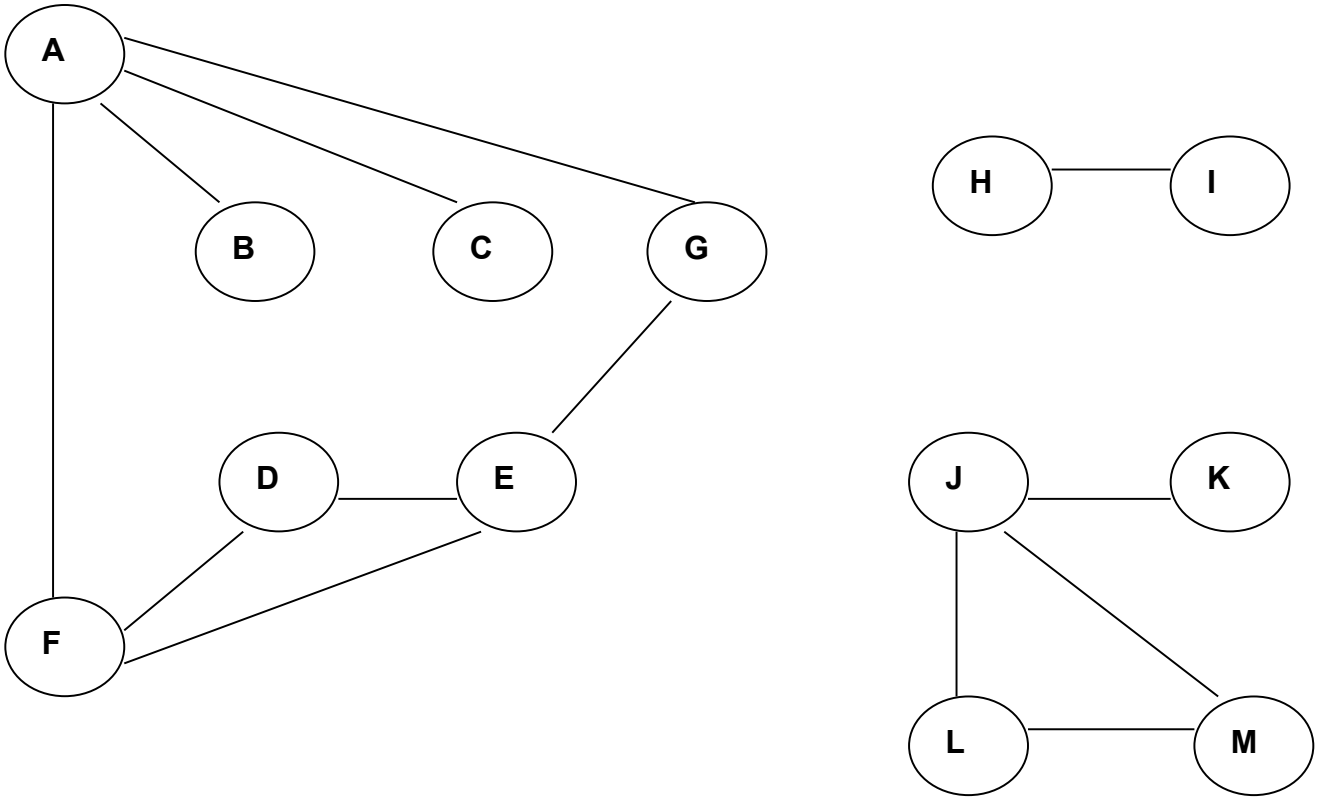
3. Các giải thuật duyệt đồ thị

Có nhiều bài toán được định nghĩa theo đối tượng và các kết nối giữa các đối tượng ấy.

Một đồ thị là một đối tượng toán học mà mô tả những bài toán như vậy.
Các ứng dụng trong các lãnh vực:

- **Giao thông**
- **Viễn thông**
- **Điện lực**
- **Mạng máy tính**
- **Cơ sở dữ liệu**
- **Trình biên dịch**
- **Các hệ điều hành**
- **Lý thuyết đồ thị**

Một thí dụ



Hình 3.1a Một đồ thị thí dụ

Cách biểu diễn đồ thị

Ta phải ánh xạ các tên đỉnh thành những số nguyên trong tầm trị giữa 1 và V.

Giả sử có tồn tại hai hàm:

- **hàm *index*: chuyển đổi từ tên đỉnh thành số nguyên**
- **hàm *name*: chuyển đổi số nguyên thành tên đỉnh.**

Có hai cách biểu diễn đồ thị:

- **dùng ma trận kế cận**
- **dùng tập danh sách kế cận**

Cách biểu diễn ma trận kế cận

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	1	1	0	0	1	1	0	0	0	0	0	0
B	1	1	0	0	0	0	0	0	0	0	0	0	0
C	1	0	1	0	0	0	0	0	0	0	0	0	0
D	0	0	0	1	1	1	0	0	0	0	0	0	0
E	0	0	0	1	1	1	1	0	0	0	0	0	0
F	1	0	0	1	1	1	0	0	0	0	0	0	0
G	1	0	0	0	1	0	1	0	0	0	0	0	0
H	0	0	0	0	0	0	0	1	1	0	0	0	0
I	0	0	0	0	0	0	0	1	1	0	0	0	0
J	0	0	0	0	0	0	0	0	0	1	1	1	1
K	0	0	0	0	0	0	0	0	0	1	1	0	0
L	0	0	0	0	0	0	0	0	0	1	0	1	1
M	0	0	0	0	0	0	0	0	0	1	0	1	1

Một ma trận V hàng V cột chứa các giá trị Boolean mà *a[x, y]* là *true* if nếu tồn tại một cạnh từ đỉnh *x* đến đỉnh *y* và *false* nếu ngược lại.

Hình 3.1b: Ma trận kế cận của đồ thị ở hình 3.1a

Giải thuật

```
program adjmatrix (input, output);
const maxV = 50;
var j, x, y, V, E: integer;
    a: array[1..maxV, 1..maxV] of boolean;
begin
    readln (V, E);
    for x: = 1 to V do /*initialize the matrix */
        for y: = 1 to V do a[x, y]: = false;
    for x: = 1 to V do a[x, x]: = true;
    for j: = 1 to E do
    begin
        readln (v1, v2);
        x := index(v1); y := index(v2);
        a[x, y] := true; a[y, x] := true
     end;
end.
```

Lưu ý: Mỗi cạnh tương ứng với 2 bit trong ma trận: mỗi cạnh nối giữa *x* và *y* được biểu diễn bằng giá trị *true* tại cả *a[x, y]* và *a[y, x].*

Để tiện lợi giả định rằng có tồn tại một cạnh nối mỗi đỉnh về chính nó.

Cách biểu diễn bằng tập danh sách kế cận

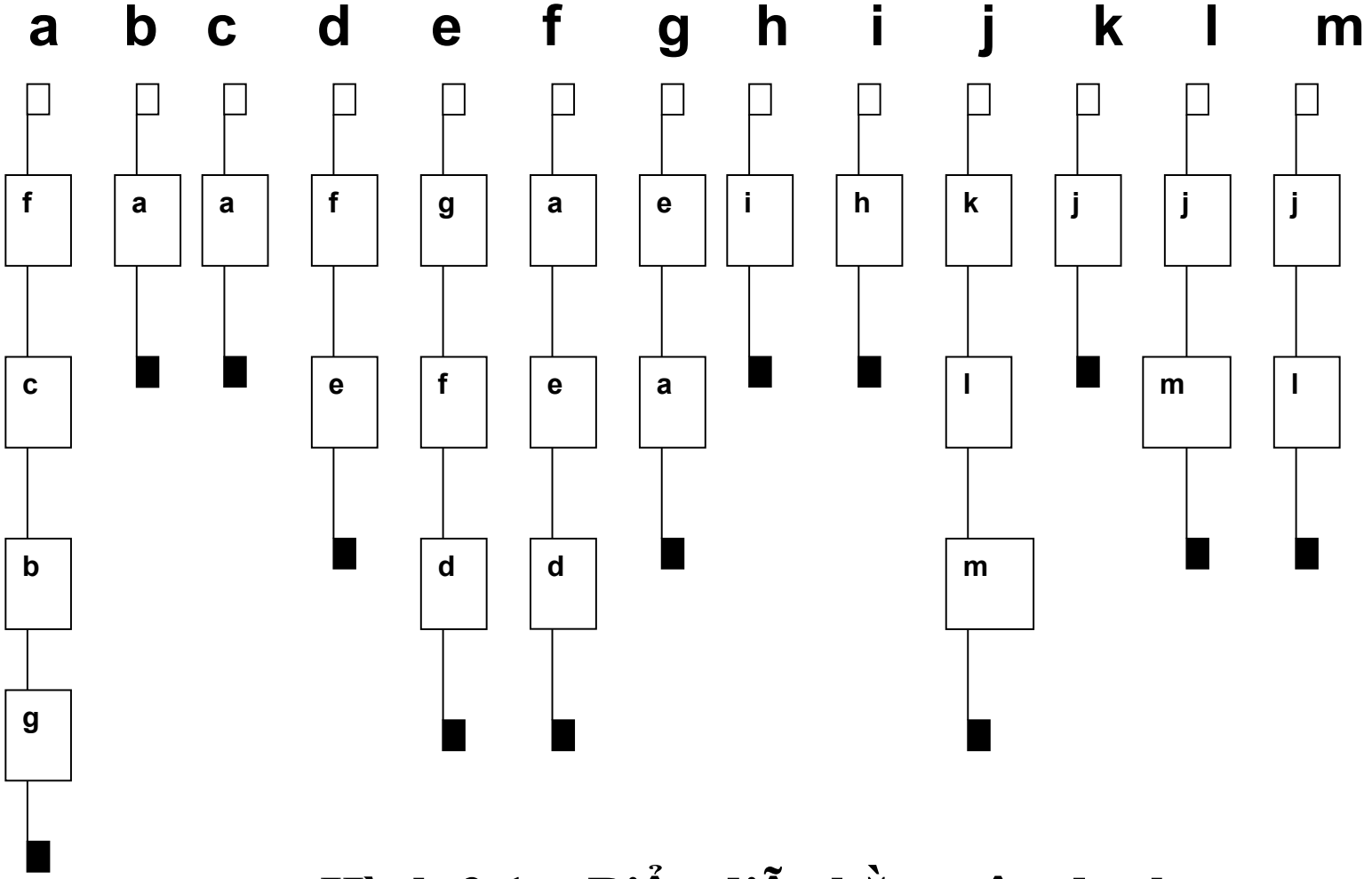
Trong cách biểu diễn này, mọi đỉnh mà nối tới một đỉnh được kết thành một *danh sách kế cận* (*adjacency-list*) cho đỉnh đó.

```
program adjlist (input, output);
const maxV = 100;
type link = ↑node
       node = record v: integer; next: link end;
var j, x, y, V, E: integer;
      t, x: link;
      adj: array[1..maxV] of link;
```

```
begin
    readln(V, E);
    new(z); z↑.next: = z;
    for j: = 1 to V do adj[j]: = z;
    for j: 1 to E do
    begin
        readln(v1, v2);
        x: = index(v1); y: = index(v2);
        new(t); t↑.v: = x; t↑.next: = adj[y];
        adj[y]: = t;     /* insert x to the first element of
                                        y's adjacency list */
        new(t); t↑.v = y; t↑.next: = adj[x];
        adj[x]:= t;      /* insert y to the first element of
                                        x's adjacency list */
    end;
end.
```

Lưu ý: Mỗi cạnh trong đồ thị tương ứng với hai nút trong tập danh sách kế cận.

Số nút trong tập danh sách kế cận bằng 2|E|.



Hình 3.1c: Biểu diễn bằng tập danh sách kế cận của đồ thị ở hình 3.1

So sánh hai cách biểu diễn đồ thị

- Nếu biểu diễn đồ thị bằng tập danh sách kế cận, việc kiểm tra xem *có tồn tại một cạnh giữa hai đỉnh u và v* sẽ có độ phức tạp thời gian O(V) vì có thể có O(V) đỉnh tại danh sách kế cận của đỉnh *u*.
- Nếu biểu diễn đồ thị bằng ma trận kế cận, việc kiểm tra xem *có tồn tại một cạnh giữa hai đỉnh u và v* sẽ có độ phức tạp thời gian O(1) vì chỉ cần xem xét phần tử tại vị trí (u,v) của ma trận.
- Biểu diễn đồ thị bằng ma trận kế cận gây lãng phí chỗ bộ nhớ khi đồ thị là một đồ thị thưa (không có nhiều cạnh trong đồ thị, do đó số vị trí mang giá trị 1 là rất ít)

Các phương pháp duyệt đồ thị

Duyệt hay tìm kiếm trên đồ thị: viếng mỗi đỉnh/nút trong đồ thị một cách có hệ thống.

Có hai cách chính để duyệt đồ thị:

- **duyệt theo chiều sâu trước (*depth-first-search*)**
- **duyệt theo chiều rộng trước (*breadth-first-search*).**

Duyệt theo chiều sâu trước

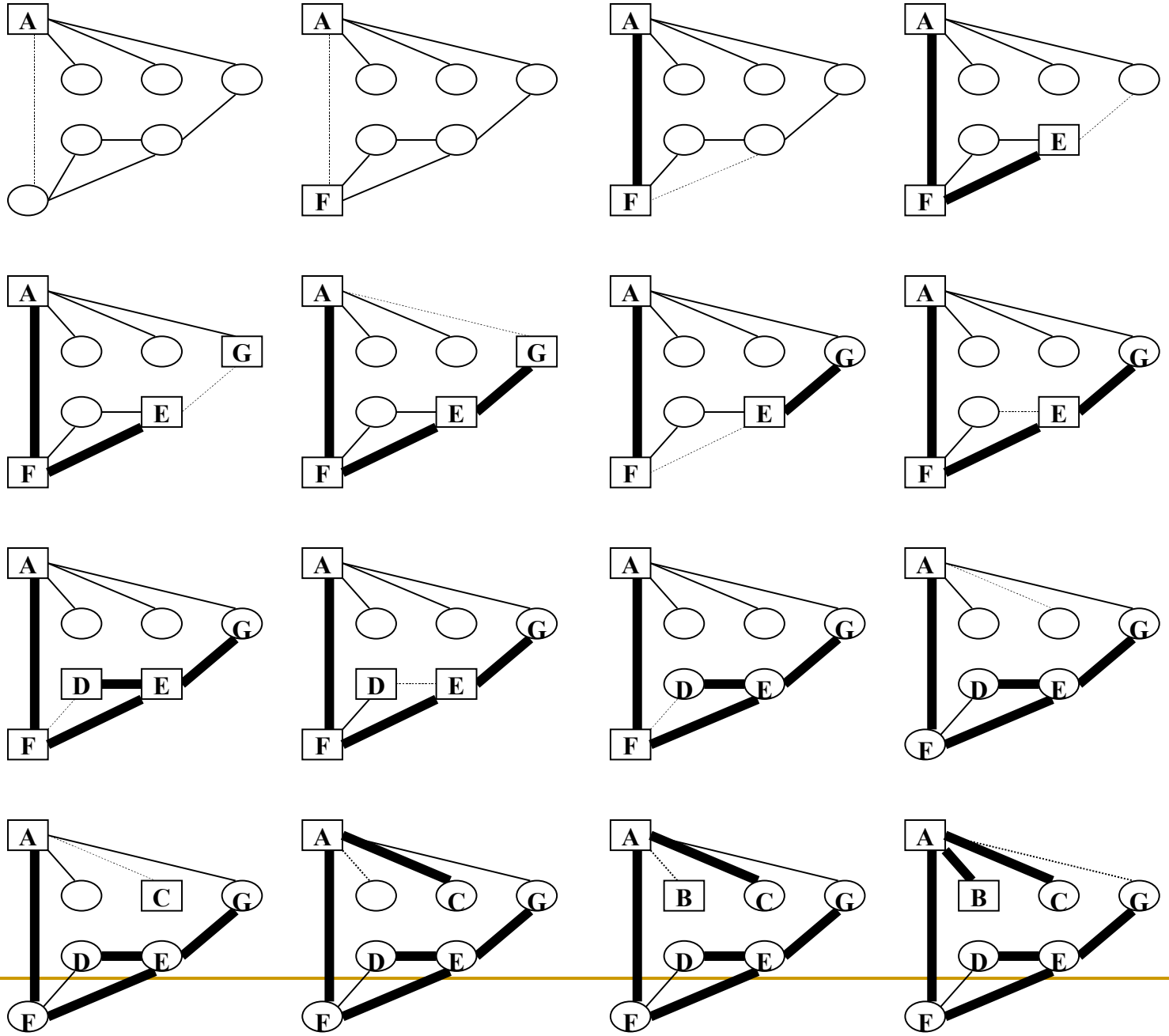
```
procedure dfs;
  procedure visit(n:vertex);
  begin
      add n to the ready stack;
      while the ready stack is not empty do
          get a vertex from the stack, process it,
          and add any neighbor vertex that has not been processed
to the stack.
          if a vertex has already appeared in the stack,  there is no
need to push it to the stack.
      end;
begin
    Initialize status;
    for each vertex, say n, in the graph do
       if the status of n is "not yet visited"  then visit(n)
end;
```

Tìm kiếm theo chiều sâu trước – biểu diễn danh sách kế cận (giải thuật đệ quy)

```
procedure list-dfs;
var id, k: integer;
      val: array[1..maxV] of integer;
  procedure visit (k: integer);
  var t: link;
  begin
    id: = id + 1; val[k]: = id; /* change the status of k
                                    to "visited" */
    t: = adj[k];              / * find the neighbors of the vertex k */
    while t <> z do
    begin
       if val[t ↑.v] = 0 then visit(t↑.v);
       t: = t↑.next
    end
  end;
```

```
begin
    id: = 0;
    for k: = 1 to V do val[k]: = 0;  /initialize
                                    the status of all vetices */
    for k: = 1 to V do
        if val[k] = 0 then visit(k)
end;
```

Ghi chú: **Mảng *val[1..V]* chứa trạng thái của cácđỉnh.**
val[k] = 0 nếu đỉnh *k* chưa hề được viếng ("not yet visited"),
val[k] ≠ 0 nếu đỉnh *k* đã được viếng.
val[k]: = j nghĩa là đỉnh *j*th mà được viếng trong quá trình duyệt là đỉnh *k*.



Hình 3.2 Duyệt theo chiều sâu trước

Như vậy kết quả của DFS trên đồ thị cho ở hình 3.1a với tập danh sách kế cận cho ở hình 3.1c là

A F E G D C B

Lưu ý: thứ tự của các đỉnh trong các danh sách kế cận có ảnh hưởng đến thứ tự duyệt của các đỉnh khi áp dụng DFS.

Độ phức tạp của DFS

Tính chất 3.1.1 *Duyệt theo chiều sâu trước một đồ thị biểu diễn bằng các danh sách kế cận đòi hỏi thời gian tỉ lệ V+ E.*

Chứng minh: Chúng ta phải gán trị cho mỗi phần tử của mảng *val* (do đó tỉ lệ với O(V)), và xét mỗi nút trong các danh sách kết cận biểu diễn đồ thị (do đó tỉ lệ với O(E)).

DFS – biểu diễn bằng ma trận kế cận

Cùng một phương pháp có thể được áp dụng cho đồ thị được biểu diễn bằng ma trận kế cận bằng cách dùng thủ tục *visit* sau đây:

```
procedure visit(k: integer);
var t: integer;
begin
   id: = id + 1; val[k]: = id;
   for t: = 1 to V do
      if a[k, t] then
         if val[t] = 0 then visit(t)
end;
```

Tính chất 3.1.2 ***Duyệt theo chiều sâu trước một đồ thị biểu diễn bằng ma trận kế cận tỉ lệ với V^2.***

Chứng minh: Bởi vì mỗi bit trong ma trận kế cận của đồ thị đều phải kiểm tra.

Duyệt theo chiều rộng trước

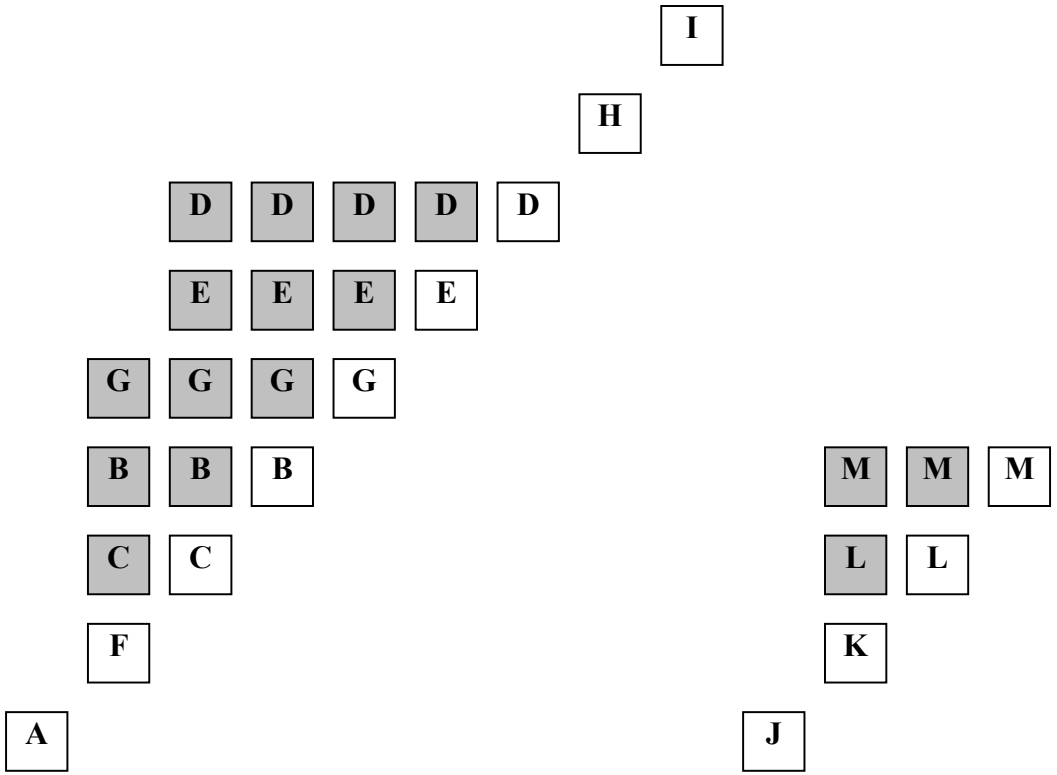
Khi duyệt đồ thị nếu ta dùng một queue thay vì một stack, ta sẽ đi đến một giải thuật *duyệt theo chiều rộng trước* (*breadth-first-search)*.

```
procedure bfs;
    procedure visit(n: vertex);
    begin
        add n to the ready queue;
        while the ready queue is not empty do
        get a vertex from the queue, process it, and add any neighbor
        vertex that has not been processed to the queue and
        change their  status to ready.
    end;
    begin
        Initialize status;
        for each vertex, say n, in the graph
            if the status of n is “not yet visited”  then visit(n)
    end;
```

```
procedure list-bfs;
var id, k: integer; val: array[1..max V] of integer;
  procedure visit(k: integer);
  var t: link;
  begin
    put(k); /* put a vertex to the queue */
    repeat
      k: = get; /* get a vertex from the queue */
      id: = id + 1; val[k]: = id;     /* change the status  of k to "visited" */
      t: = adj[k];              /* find the neighbors of the vertex k */
      while t <> z do
      begin
        if val[t ↑.v] = 0 then
        begin
          put(t↑.v); val [t↑.v]: = -1      /* change  the vertex t↑.v to "ready" */
        end;
        t: = t↑.next
      end
    until queueempty
  end;
```

```
begin
    id: = 0; queue-initialze;
    for k: = 1 to V do val[k]: = 0; /initialize the
                                    status of all vertices */
    for k: = 1 to V do
        if val[k] = 0 then visit(k)
end;
```

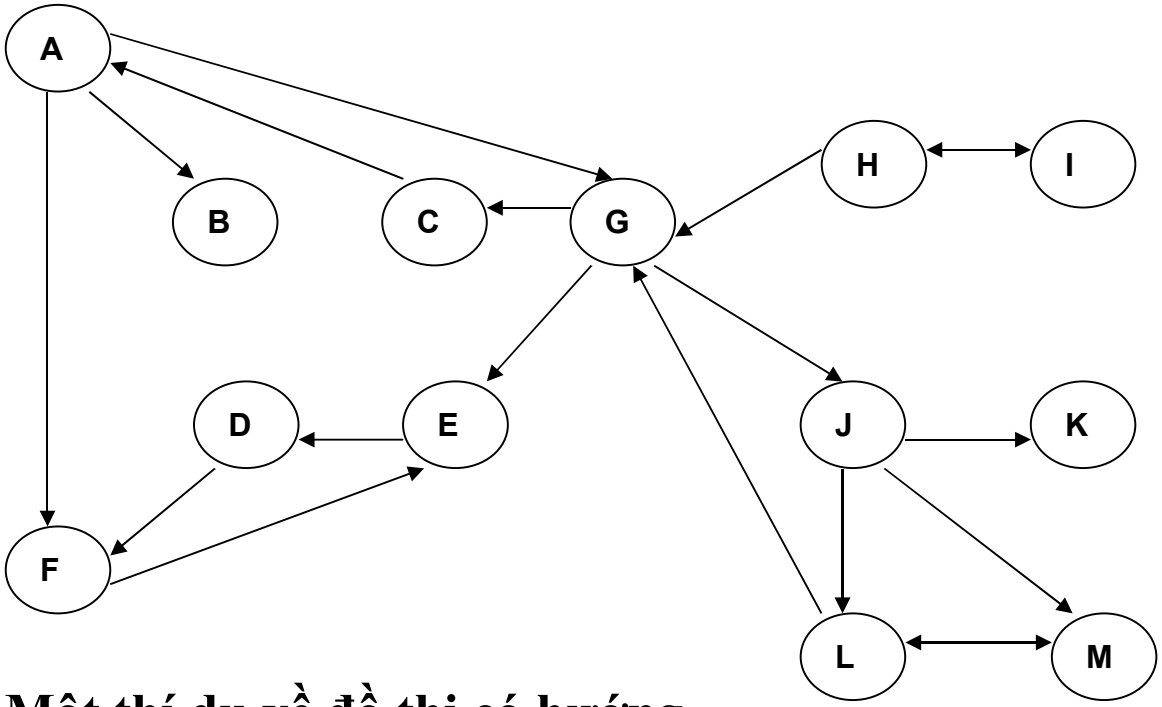
Duyệt theo chiều sâu trước và duyệt theo chiều rộng trước chỉ khác nhau ở chỗ giải thuật đầu dùng stack và giải thuật sau dùng hàng đợi. Do đó, độ phức tạp tính toán của DFS và BFS là như nhau.



Hình 3.3 Nội dung của hàng đợi khi thực hiện BFS

4. Xếp thứ tự tôpô

Các đồ thị có hướng là các đồ thị trong đó các cạnh nối với các nút có hướng.



Hình 3.4. Một thí dụ về đồ thị có hướng

Thường thì hướng của các cạnh biểu thị mối liên hệ trước sau (precedence relationship) trong ứng dụng được mô hình hóa.

Thí dụ, đồ thị có hướng có thể được dùng để mô hình hóa một đường dây sản xuất (assembly line).

Trong phần này, chúng ta xem xét giải thuật sắp thứ tự topo (topological sorting)

Lưu ý về cách biểu diễn đồ thị có hướng

- **Nếu ta biểu diễn đồ thị có hướng bằng tập danh sách kế cận, mỗi cạnh trong đồ thị tương ứng với một nút trong tập danh sách kế cận. (mỗi cạnh nối từ *x* đến *y* được biểu diễn bằng một nút có nhãn *y* được đưa vào danh sách kế cận của đỉnh *x*)**
  - Số nút trong tập danh sách kế cận bằng với số cạnh |E|

- **Nếu ta biểu diễn đồ thị có hướng bằng ma trận kế cận, mỗi cạnh trong đồ thị tương ứng với một bit 1 trong ma trận kế cận. (mỗi cạnh nối từ *x* đến *y* được biểu diễn bằng giá trị *true* tại *a[x, y]).***

Xếp thứ tự tôpô

Đồ thị có hướng không chu trình (Directed Acyclic Graph)

Đồ thị có hướng mà không có chu trình được gọi là các *đồ thị có hướng không chu trình* (*dag*s).

Tập thứ tự riêng phần và xếp thứ tự tôpô

Cho G là một đồ thị có hướng không chu trình. Xét quan hệ thứ tự < được định nghĩa như sau:

u < v nếu có một lối đi từ *u* đến *v* trong G.

Quan hệ này có 3 tính chất:

(1) Với mỗi đỉnh trong V[G], not (u < u). (*không phản xạ*)

(2) nếu u < v, thì not(v < u) . (*không đối xứng*)

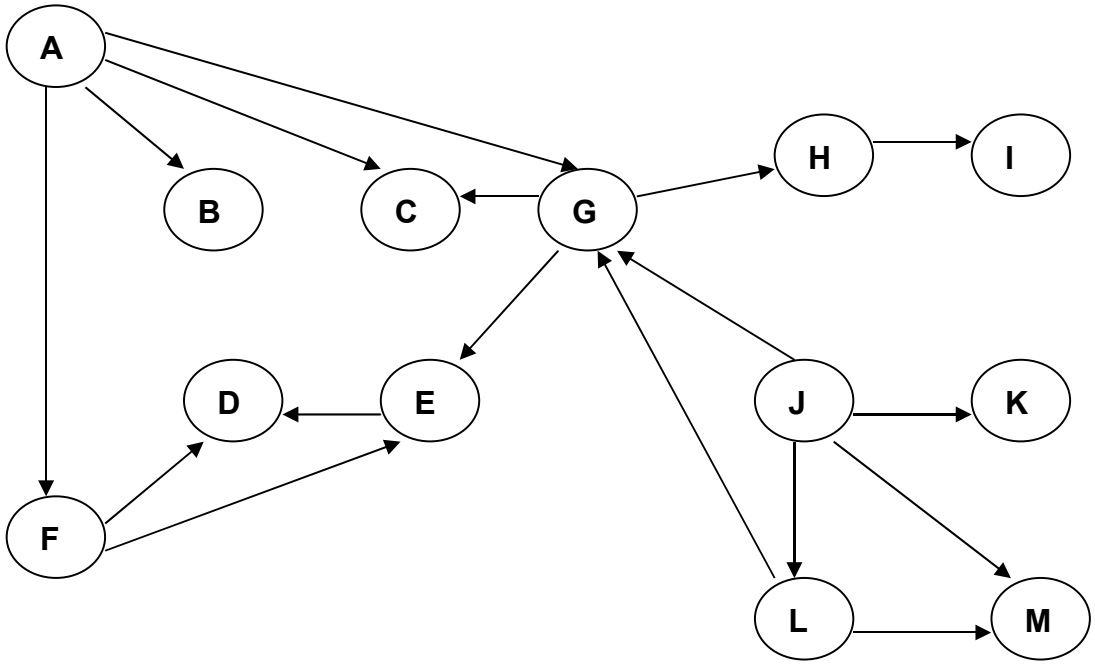
(3) nếu u < v và v < w, thì u < w. (*Truyền*)

Quan hệ < là một *quan hệ thứ tự riêng phần*.

Xếp thứ tự tôpô

Cho G là một đồ thị có hướng không chu trình. Một thứ tự tôpô (*topological sort*)T của G là một thứ tự tuyến tính mà bảo toàn thứ tự riêng phần ban đầu trong tập đỉnh V[G].

Nghĩa là: nếu *u < v trong V (tức là nếu có một lối đi từ u đến v trong G), thì u xuất hiện trước v trong thứ tự tuyến tính T*.



Các nút trong đồ thị ở hình trên có thể được sắp thứ tự tôpô theo thứ tự sau:

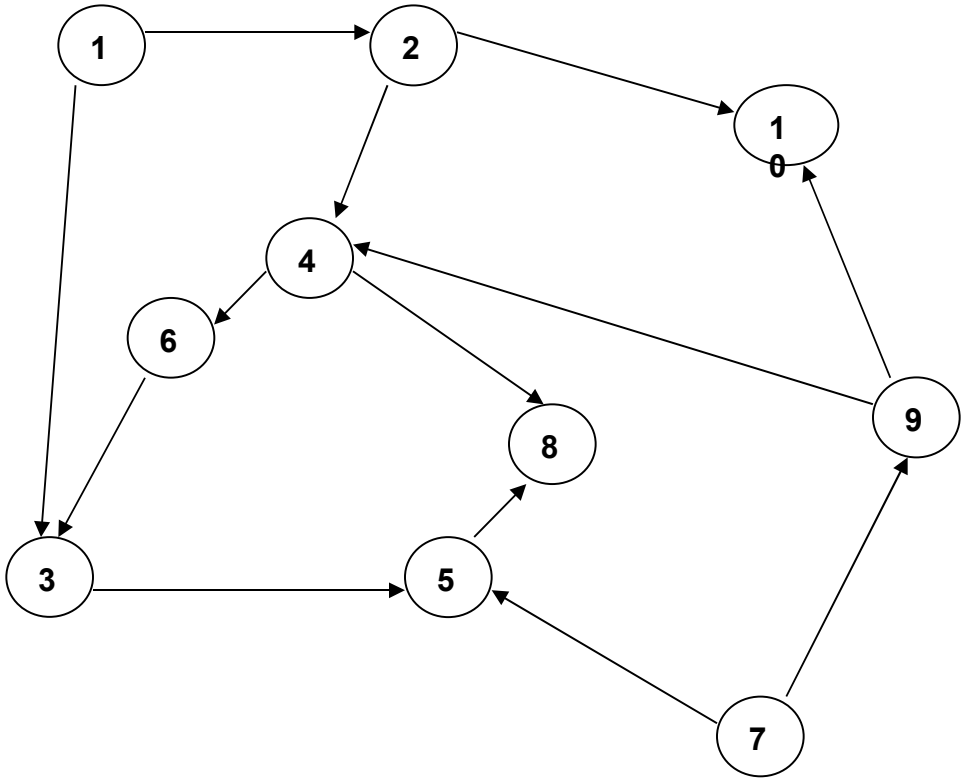
J K L M A G H I F E D B C

Phương pháp 1 sắp xếp tôpô

Phương pháp này thực hiện theo kiểu *tìm kiếm theo chiều sâu trước* và thêm một nút vào danh sách mỗi khi cần thiết lấy một nút ra khỏi stack để tiếp tục. Khi gặp một nút không có nút đi sau thì ta sẽ lấy ra (pop) một phần tử từ đỉnh stack. Lặp lại quá trình này cho đến khi stack rỗng. Đảo ngược danh sách này ta sẽ được thứ tự tôpô.

Algorithm:

Start with nodes with no predecessor, put them in the stack.
while the stack is not empty **do**
  if the node at top of the stack has some successors
  then
    push all its successors onto the stack
  else pop it from the stack and add it to the list.



			8					6							
		5	5	5			4	4	4						
	3	3	3	3	3		10	10	10	10					
	2	2	2	2	2	2	2	2	2	2	2				
1	1	1	1	1	1	1	1	1	1	1	1	1		9	
7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7

Hình 3.5 Sắp thứ tự tôpô

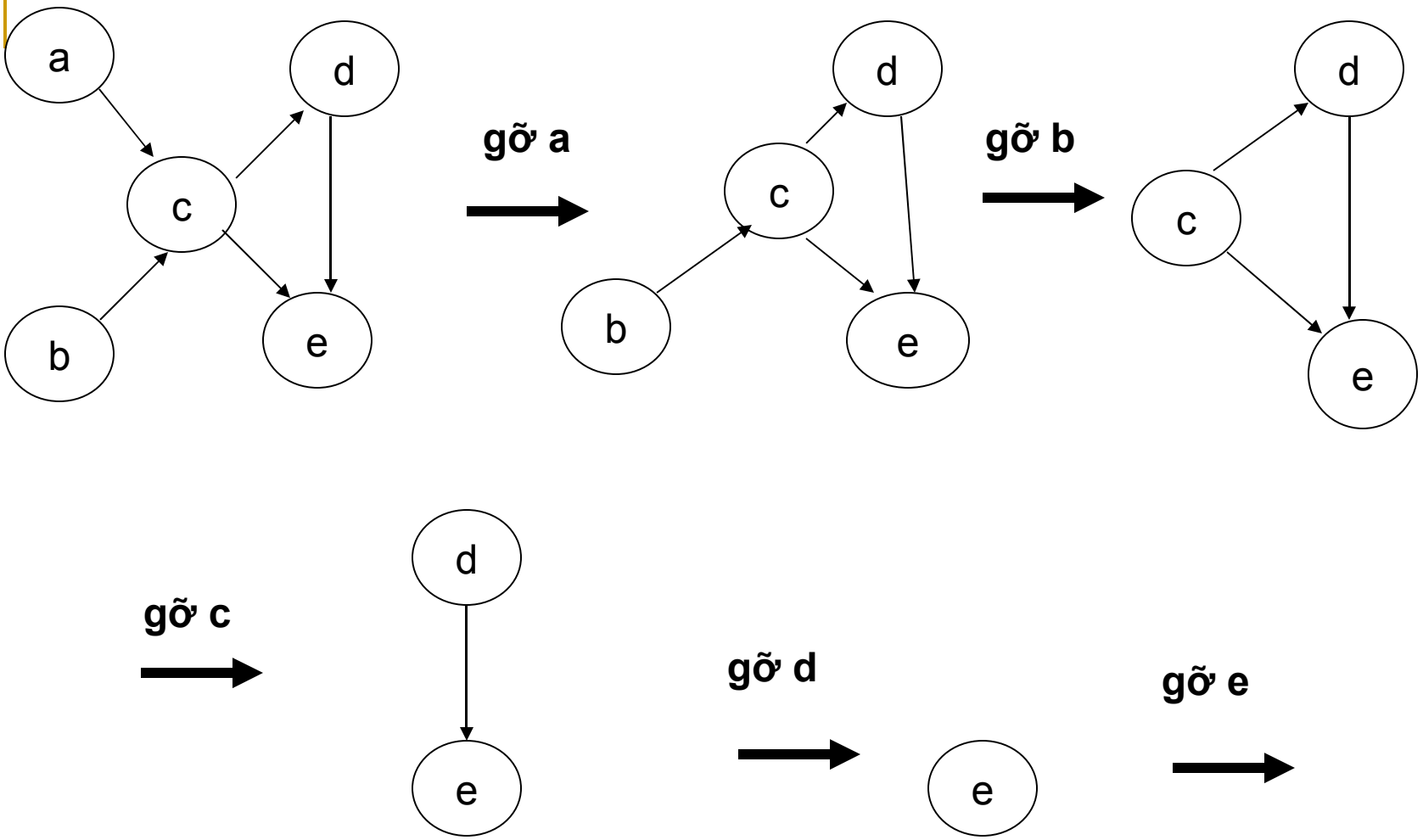
Độ phức tạp của giải thuật sắp xếp tô pô phương pháp 1

- **Tính chất: Độ phức tạp tính toán của giải thuật sắp xếp tô pô là O(|E|+|V|) nếu đồ thị được diễn tả bằng tập danh sách kế cận.**

- **Chứng minh: Điều này hiển nhiên vì thân của vòng lặp *while* được thực hiện tối đa 1 lần cho mỗi cạnh. Và tác vụ khởi tạo stack thì tỉ lệ với số đỉnh của đồ thị**

Phương pháp 2 sắp thứ tự tô pô

- **Ý tưởng: Liên tiếp nhận diện một nút là nút nguồn (nút không có nút đi sau) và tháo gỡ nó ra khỏi đồ thị cùng với các cạnh đi ra từ nó. Quá trình lặp sẽ dừng lại khi không còn nút trong đồ thị. Thứ tự của các nút bị xóa bỏ sẽ tạo thành một lời giải của bài toán sắp thứ tự tô pô.**
- **Giải thuật này thể hiện rất rõ chiến lược *giảm (một)-để-trị*.**



Thứ tự tô pô là a, b, c, d, e

Giải thuật của phương pháp 2

Algorithm:

Start with nodes with no predecessor, put them in the queue.

while the queue is not empty **do**

    remove the front node N of the queue

    for each neighbor node M of node N **do**

        delete the edge from N to M

        if the node M has no successor **then**

            add M to the rear of the queue

    endfor

endwhile

Độ phức tạp của giải thuật này là bao nhiêu?

5. Giải thuật sinh các hoán vị

- Cho một tập *n* phần tử A= {a_1,a_2,…,a_n}. Ta muốn sinh ra tất cả n! hoán vị của tập ấy.
- Chiến lược *Giảm-để-trị* có thể có gợi ý gì về giải thuật sinh tất cả các hoán vị của một tập *n* phần tử?
- *Bài toán nhỏ hơn một bậc* là sinh ra tất cả (n-1)! hoán vị cho một tập con *n* -1 phần tử của tập *A* nói trên. Giả sử bài toán này đã được giải xong, ta có thể giải bài toán nguyên thủy bằng cách *chèn* phần tử còn lại vào tại mỗi vị trí trong *n* vị trí khả hữu của các phần tử trong từng hoán vị của tập *n* -1 phần tử đã sinh.
- Tất cả các hoán vị đạt được bằng cách này sẽ khác biệt nhau.

Thí dụ

khởi đầu	1					
thêm 2	12	21				
	right to left					
thêm 3	123	132	312	213	231	321
	right to left			right to left		

Để đơn giản, giả sử tập *A* là một tập hợp các số nguyên từ 1 đến *n*. Chúng có thể được hiểu như là tập các chỉ số của tập n phần tử $\{a_1,a_2,\ldots,a_n\}$.

Giải thuật PERM

1. Set j:= 1 and write down the permutation <1>
2. set j:= j+1
3. **for** each permutation <a_1 a_2…a_{j-1}> on j-1 elements **do**
4. create and list P:=< a_1 a_2…a_{j-1}*j*>
5. **for** i:= j-1 **downto** 1 **do**
6. set P:= P with the values assigned to positions i and i+1 switched and list P

// end for loop at step 3

7. **if** j < n, **then** go to step 2 **else** stop.

Độ phức tạp của giải thuật PERM

Tính chất: Độ phức tạp của giải thuật PERM sinh ra tất cả các hoán vị của tập n phần tử là n!

Chứng minh:

- Thao tác căn bản: thao tác chèn phần tử còn lại vào một hoán vị đã có.
- Với mỗi hoán vị từ tập con n-1 phần tử (gồm tất cả (n-1)! các hoán vị này), ta đưa phần tử còn lại vào n vị trí khả hữu. Như vậy tổng cọng có n.(n-1)! thao tác chèn phần tử còn lại vào một hoán vị đã có.
- Do đó: C(n) = O(n!)
- Nhận xét: Vì n! tăng rất nhanh nên với n chỉ hơi lớn (10 trở lên), giải thuật cho ra kết quả cực kỳ chậm.

Công thức Stirling

n! xấp xỉ bằng với hàm

$$\sqrt{2\prod n}\,(n/e)^n$$

với e là cơ số logarit tự nhiên (e = 2.71828)