Chương 3

Kiểm thử hộp trắng

(Buổi 4, 5, 6)

Nội dung

- Kiểm thử hộp trắng là gì?
- Kiểm thử dòng điều khiển (*control flow*)
  - Đường thi hành
  - Đồ thị dòng điều khiển
  - Đường độc lập
  - Độ phức tạp *cyclomatic*
  - Qui trình kiểm thử
  - Mức phủ mã lệnh
- Kiểm thử dòng dữ liệu (*data flow*)

Kiểm thử hộp trắng

- Trong **kiểm thử hộp trắng** (*white-box tetsting*), còn được gọi là kiểm thử cấu trúc (*structural testing*), phần mềm được xem là một hộp trắng và các *test-case* được xác định từ sự thực hiện của phần mềm.
- Các kỹ thuật kiểm thử hộp trắng
  - **Kiểm thử dòng điều khiển** (*control flow testing*): sử dụng các cấu trúc điều khiển của chương trình để xây dựng các *test-case*.
  - **Kiểm thử dòng dữ liệu** (*data flow testing*): tập trung vào các nơi mà các biến được gán giá trị và các nơi mà chúng được sử dụng trong chương trình.

Kiểm thử hộp trắng

- **Mục tiêu:** Kiểm thử để xác định một thành phần phần mềm đó có thực hiện đúng với thiết kế.
- Kiểm thử một thành phần phần mềm:
  - Đơn vị chương trình (*program unit*): chương trình con (*subprogram*) ở dạng:
    - Hàm (*function*)
    - Thủ tục (*procedure*)
    - Phương thức của một lớp (*method*)
  - Một phân hệ chức năng: nhiều đơn vị chương trình chạy chung với nhau.

Kiểm thử hộp trắng

- Kiểm thử hộp trắng dựa vào mã nguồn (*source code*):
  - Giải thuật cụ thể (*algorithm*)
  - Cấu trúc dữ liệu cụ thể (*data structure*)
- Người kiểm thử hộp trắng phải nắm vững chi tiết về đoạn mã cần kiểm thử:
  - Kiến thức và kỹ năng lập trình về ngôn ngữ lập trình
  - Giải thuật
  - Cấu trúc dữ liệu

Kiểm thử hộp trắng

- Thường tốn rất nhiều thời gian và công sức nếu thành phần phần mềm quá lớn:
  - Kiểm thử tích hợp
  - Kiểm thử chức năng
  - Chủ yếu kiểm thử đơn vị (*unit testing*)

Đường thi hành

- **Đường thi hành** (***execution path***) **là một kịch bản thi hành của một đơn vị chương trình.**
  - **Danh sách có thứ tự các lệnh được thực hiện tương ứng với một lần chạy cụ thể của đơn vị phần mềm, bắt đầu từ điểm nhập cho đến điểm kết thúc của đơn vị chương trình.**
- **Mỗi đơn vị chương trình có thể có nhiều đường thi hành khác nhau (có thể rất nhiều).**

Đường thi hành

- **Ví dụ 1**

```
for (i = 1; i <= 1000; i++)
  for (j = 1; j <= 1000; j++)
    for (k = 1; k <= 1000; k++)
      Func(i,j,k);
```

chỉ có 1 đường thi hành nhưng rất dài, bao gồm 1000 * 1000 * 1000 = 1 tỉ lệnh gọi hàm `Func(i,j,k)` khác nhau.

Đường thi hành

- **Ví dụ 2**

```
if (c1) s11 else s12;
if (c2) s21 else s22;
if (c3) s31 else s32;
...
if (c32) s321 else s322;
```

có 2 ^ 32 = 4 tỉ đường thi hành khác nhau.

Mục tiêu của kiểm thử dòng điều khiển

- Đảm bảo mọi đường thi hành của đơn vị chương trình đều chạy đúng.
- Trong thực tế, công sức và thời gian để đạt mục tiêu trên là rất lớn, ngay cả đối với các đơn vị chương trình nhỏ.

Thiếu sót về dòng điều khiển

- Không thể phát hiện một số đường thi hành cần có trong thực tế vì đoạn mã bị thiếu một số trường hợp.

```
if (a > 0) Func1();
if (a == 0) Func2();
```

thiếu xử lý trường hợp `a < 0`.

Thiếu sót về dòng điều khiển

- **Kiểm tra thiếu một số trường hợp đặc biệt của một đường thi hành.**

```
int Func(int a, int b)
{ return a / b; }
```

thông thường chỉ kiểm tra `b <> 0`, thiếu kiểm tra trường hợp `b = 0`, lỗi sai chia cho số 0 (*divide by zero*) trong thời gian thực hiện chương trình (*execution time*).

Đồ thị dòng điều khiển

- Cấu trúc điều khiển của chương trình có thể được biểu diễn bởi đồ thị dòng điều khiển của chương trình.
- **Đồ thị dòng điều khiển** (*control flow graph*), hoặc **đồ thị chương trình** (*program graph*), của chương trình là một đồ thị biểu diễn dòng điều khiển (*control flow*) của chương trình này.
  - Mô tả các dòng điều khiển luận lý (*logical control flow*) của các cấu trúc điều khiển (*control structure*) được sử dụng trong chương trình.

Đồ thị dòng điều khiển

- **Đồ thị dòng điều khiển *G* = (*N*, *E*) của chương trình bao gồm tập nút *N* và tập cạnh *E*.**
  - **Có 5 loại nút:**
    - **Nút bắt đầu** (***entry node***)
    - **Nút kết thúc** (***exit node***)
    - **Nút quyết định** (***decision node***) **chứa một phát biểu điều kiện** (***conditional statement***) **có hai hoặc nhiều nhánh (ví dụ *if* và *switch*).**
    - **Nút kết nối** (***connection node***) **thường không chứa phát biểu nào và biểu diễn điểm nối của nhiều nhánh điều khiển.**
    - **Nút phát biểu** (***statement node***) **chứa một chuỗi các phát biểu, điều khiển đi từ phát biểu đầu tiên đến phàt biểu cuối cùng.**

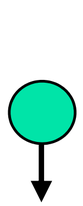
Đồ thị dòng điều khiển

- **Một cạnh (*edge*) đi từ nút *x* đến nút *y* nếu điều khiển đi từ phát biểu cuối cùng của nút *x* đến phát biểu đầu tiên của nút *y*.**

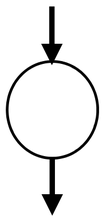
Đồ thị dòng điều khiển

- **Ba cấu trúc điều khiển cơ bản**
  - **Cấu trúc tuần tự** (*sequence structure*): thực hiện một chuỗi các phát biểu một cách tuần tự theo thứ tự xuất hiện của chúng trong chương trình.
  - **Cấu trúc điều kiện** (*conditional structure*): thực hiện việc rẽ nhánh dựa vào một điều kiện.
  - **Cấu trúc lặp** (*iterational structure*): thực hiện nhiều lần một hoặc nhiều phát biểu dựa vào một điều kiện.

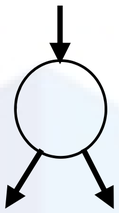
Đồ thị dòng điều khiển



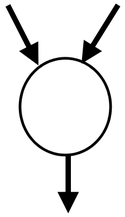
nút bắt đầu



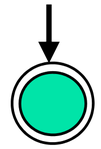
nút phát biểu



nút quyết định

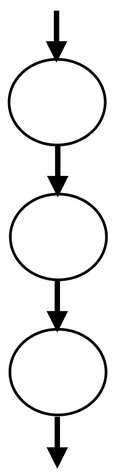


nút kết nối

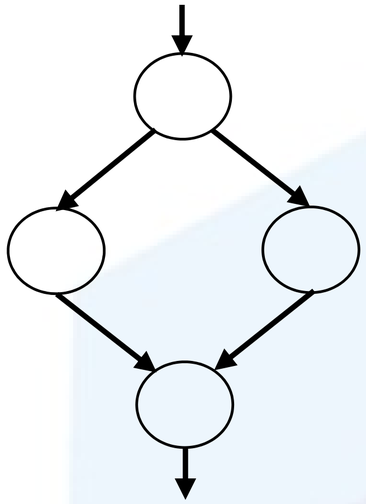


nút kết thúc

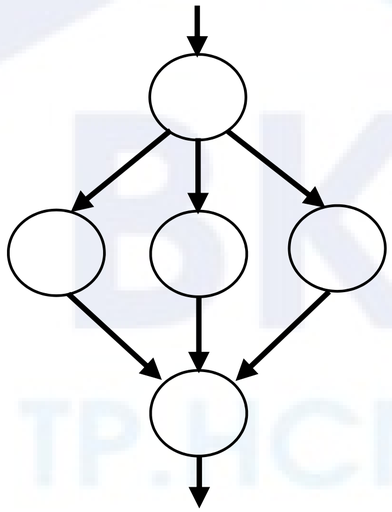
Các loại nút trong đồ thị dòng điều khiển



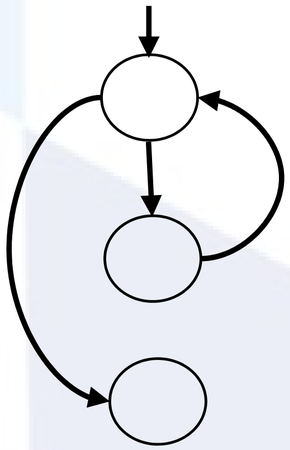
tuần tự



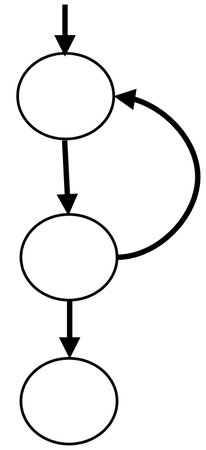
If



switch



while c ...

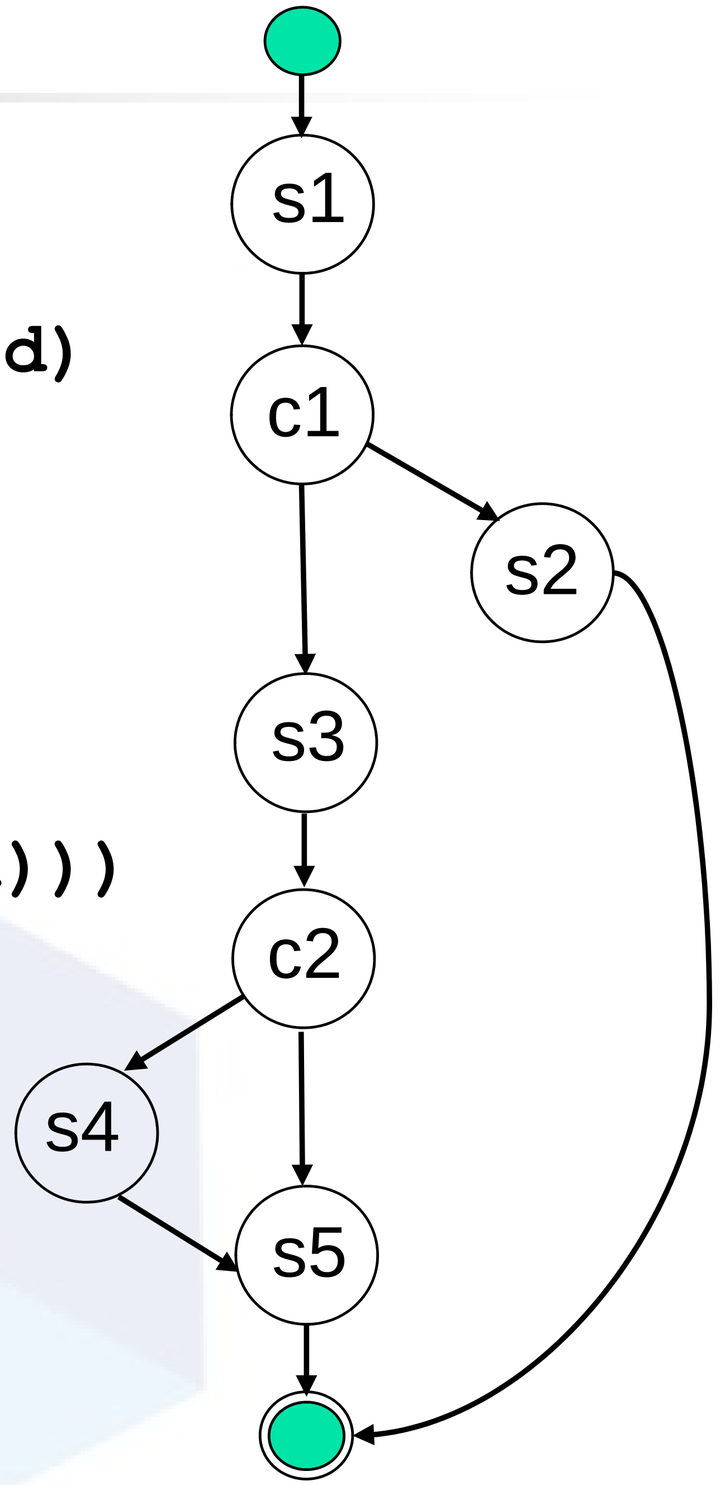


do ... while c

Đồ thị của các cấu trúc điều khiển cơ bản

Đồ thị dòng điều khiển

```
    float Func(int a, int b, int c, int d)
    {
s1    float e;
c1    if (a == 0)
s2      return 0;
s3    int x = 0;
c2    if ((a == b) || ((c == d) && bug(a)))
s4      x = 1;
s5    e = 1 / x;
s5    return e;
    }
```

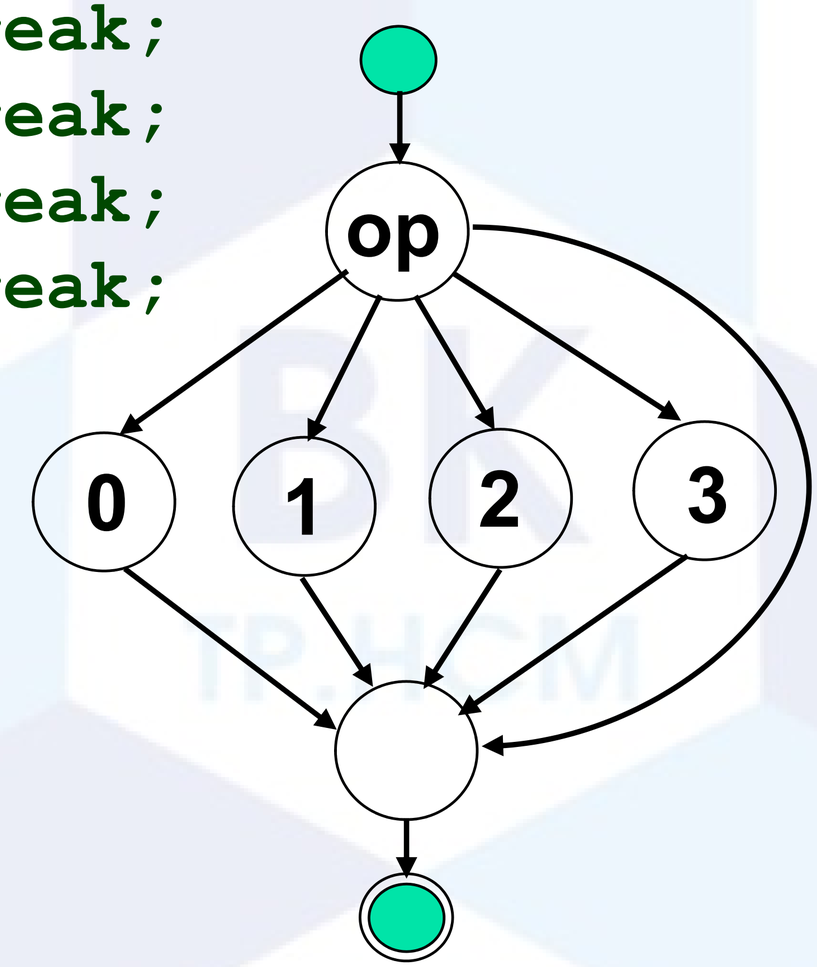


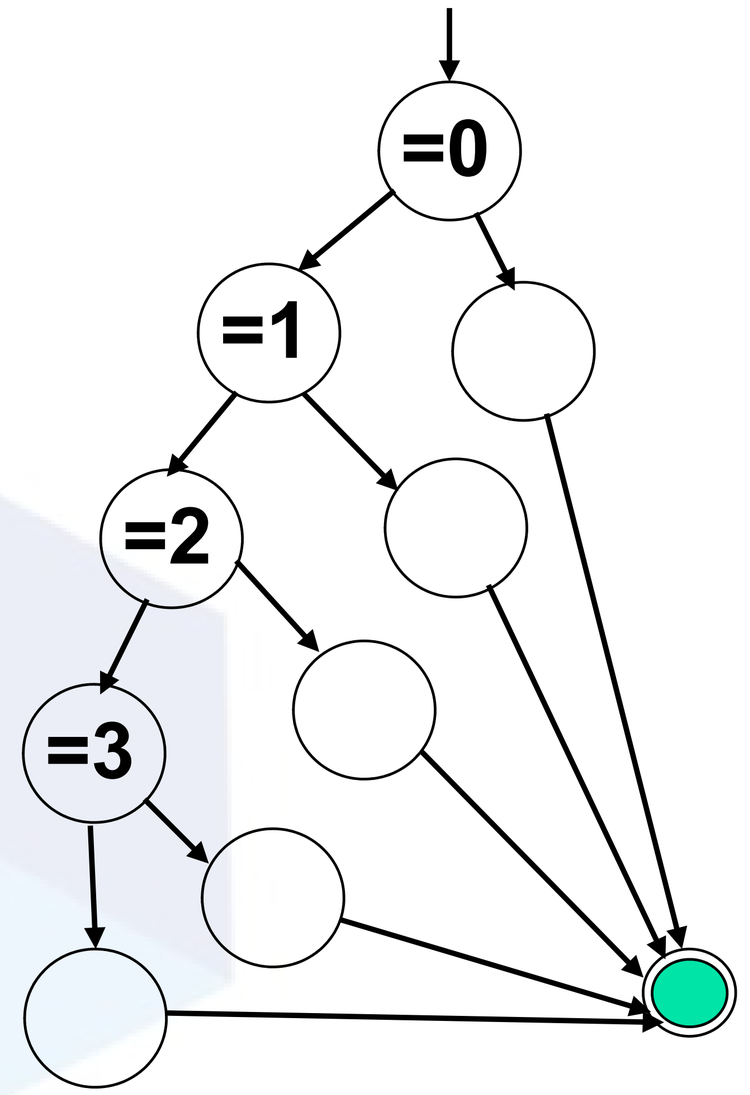
Đồ thị dòng điều khiển

- Đồ thị dòng điều khiển chỉ chứa các nút quyết định có hai nhánh thì được gọi là **đồ thị dòng điều khiển nhị phân**.
- Một đồ thị dòng điều khiển bất kỳ có thể được biến đổi thành đồ thị dòng điều khiển nhị phân.
- Một điều kiện phức hợp (bao gồm các toán tử luận lý *AND*, *OR*, *NAND*, *NOR*) có thể bao gồm các điều kiện đơn chỉ có một toán tử. Điều kiện đơn được gọi là **vị từ** (*predicate*).
- Nếu đồ thị dòng điều khiển nhị phân chỉ chứa các nút quyết định là vị từ thì được gọi là **đồ thị dòng điều khiển cơ bản**.

Đồ thị dòng điều khiển

```
int ProcessOp (int op)
{
   switch (op)
   {
   case 0 : ...; break;
   case 1 : ...; break;
   case 2 : ...; break;
   case 3 : ...; break;
   }
}
```





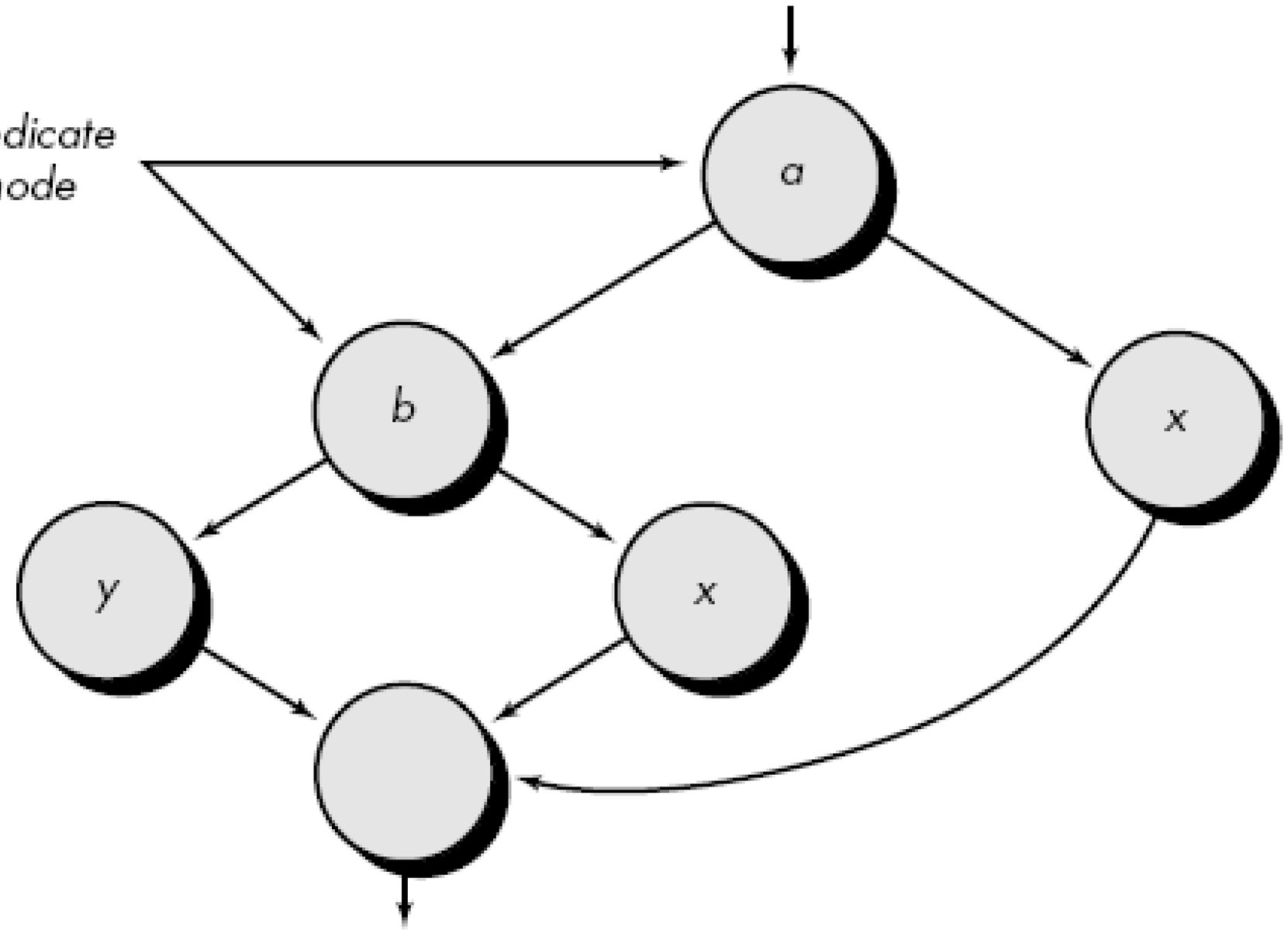
Đồ thị dòng điều khiển

- **Xây dựng đồ thị dòng điều khiển từ mã nguồn**
  - **Phân rã điều kiện phức hợp thành các điều kiện đơn là vị từ.**
  - **Mỗi vị từ được biểu diễn bởi một nút quyết định của đồ thị.**
  - **Một chuỗi các phát biểu thực thi được biểu diễn bởi một nút phát biểu của đồ thị.**
  - **Các cấu trúc điều khiển được biểu diễn bởi đồ thị của các cấu trúc điều khiển cơ bản.**

Đồ thị dòng điều khiển

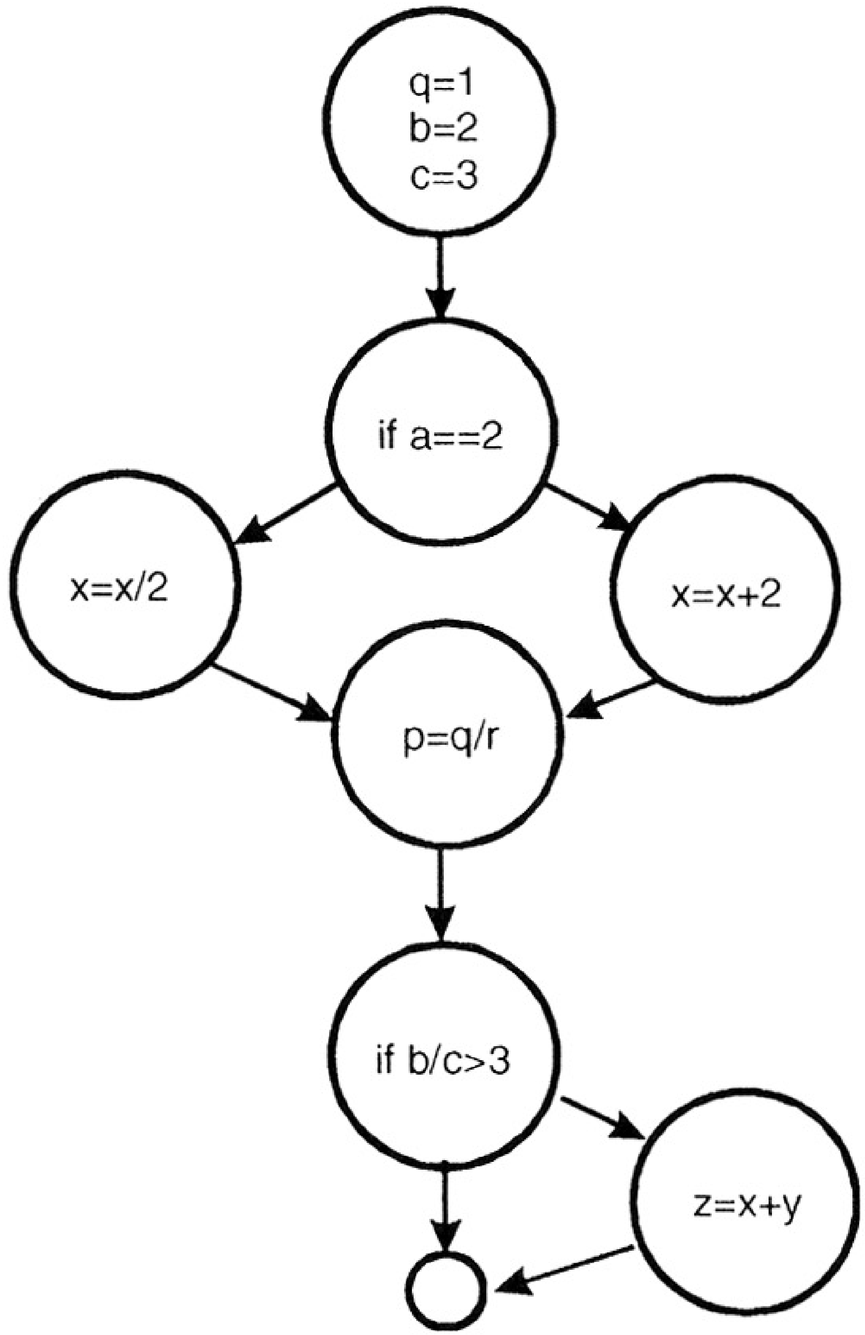
.
.
.

IF a OR b
then procedure x
else procedure y
ENDIF

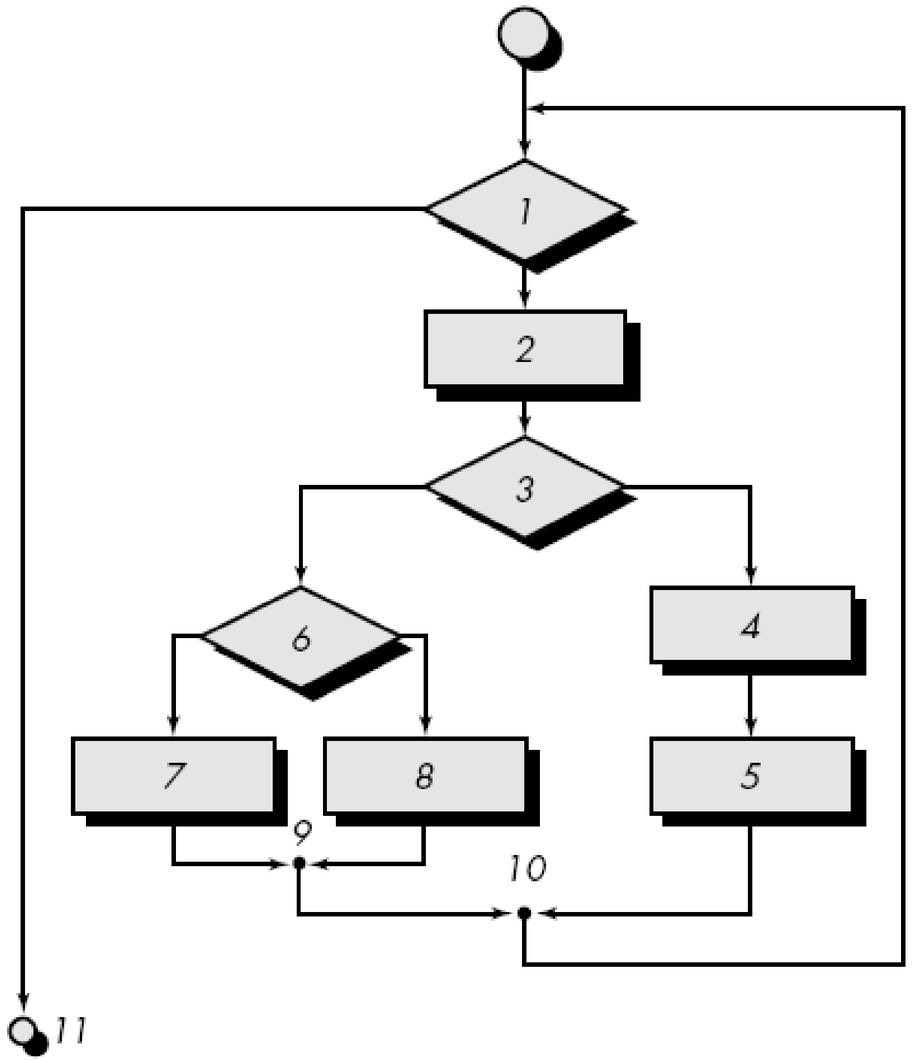


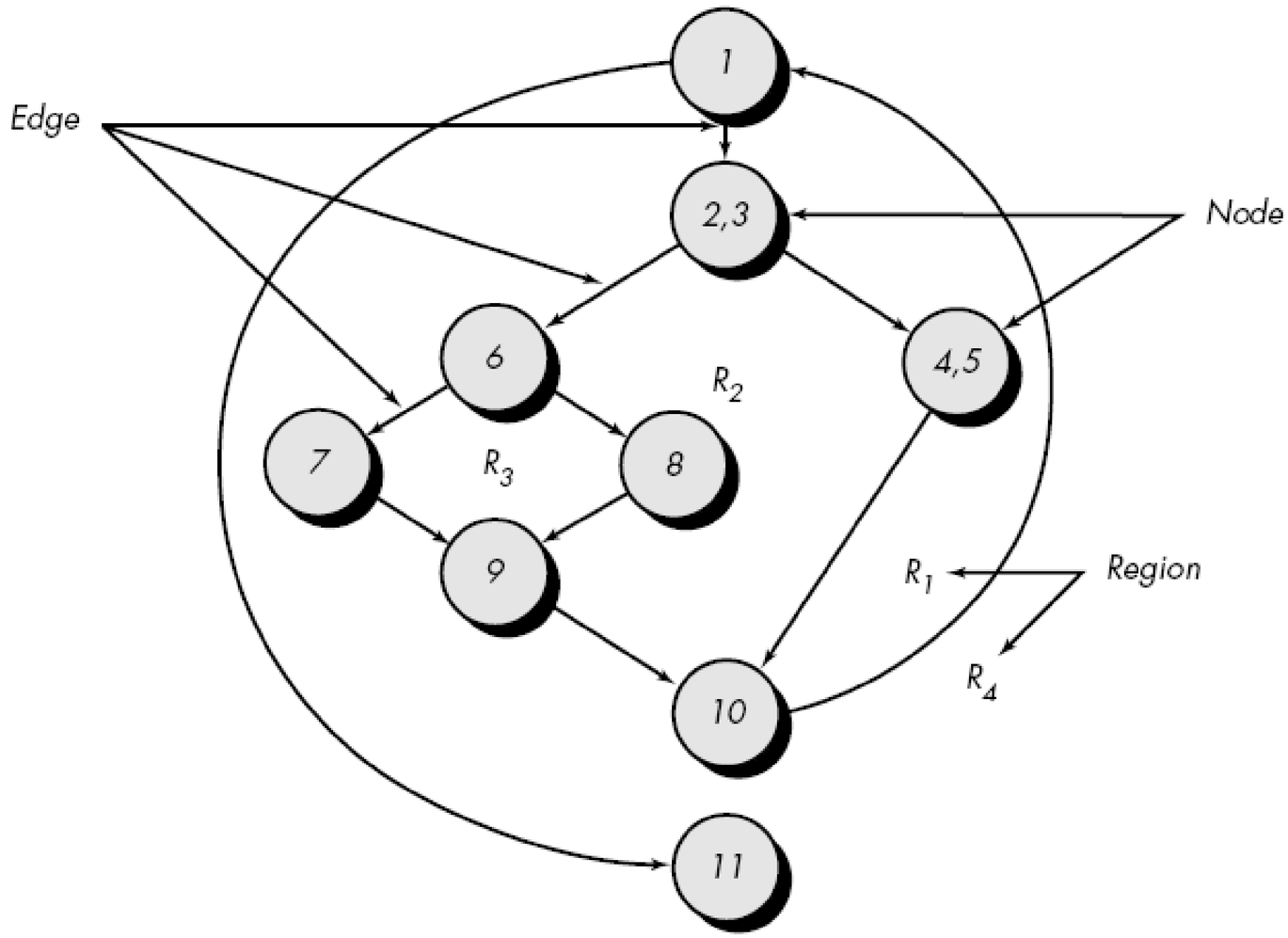
Đồ thị dòng điều khiển

```
q=1;
b=2;
c=3;
if (a==2) {x=x+2;}
else {x=x/2;}
p=q/r;
if (b/c>3) {z=x+y;}
```



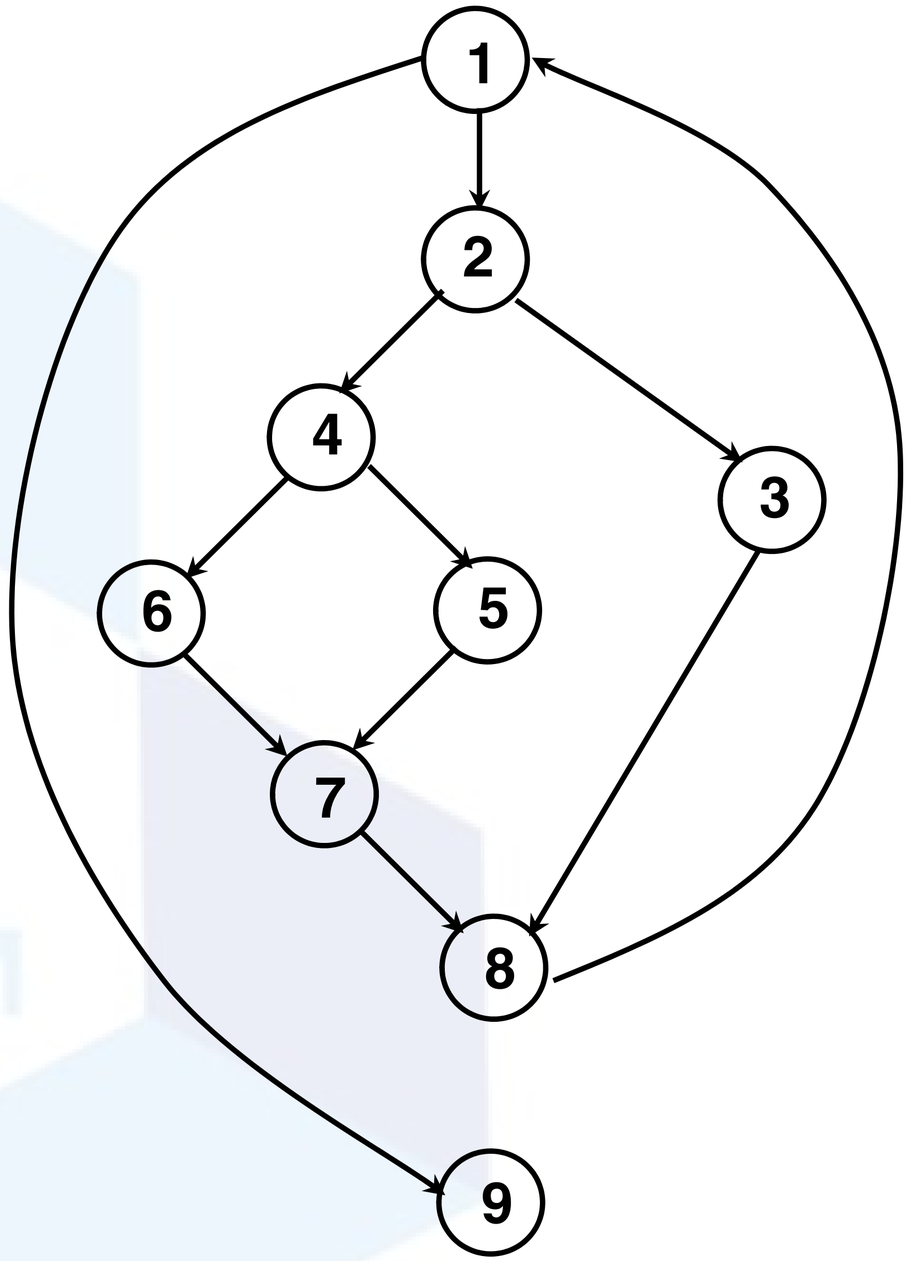
Đồ thị dòng điều khiển





Đồ thị dòng điều khiển

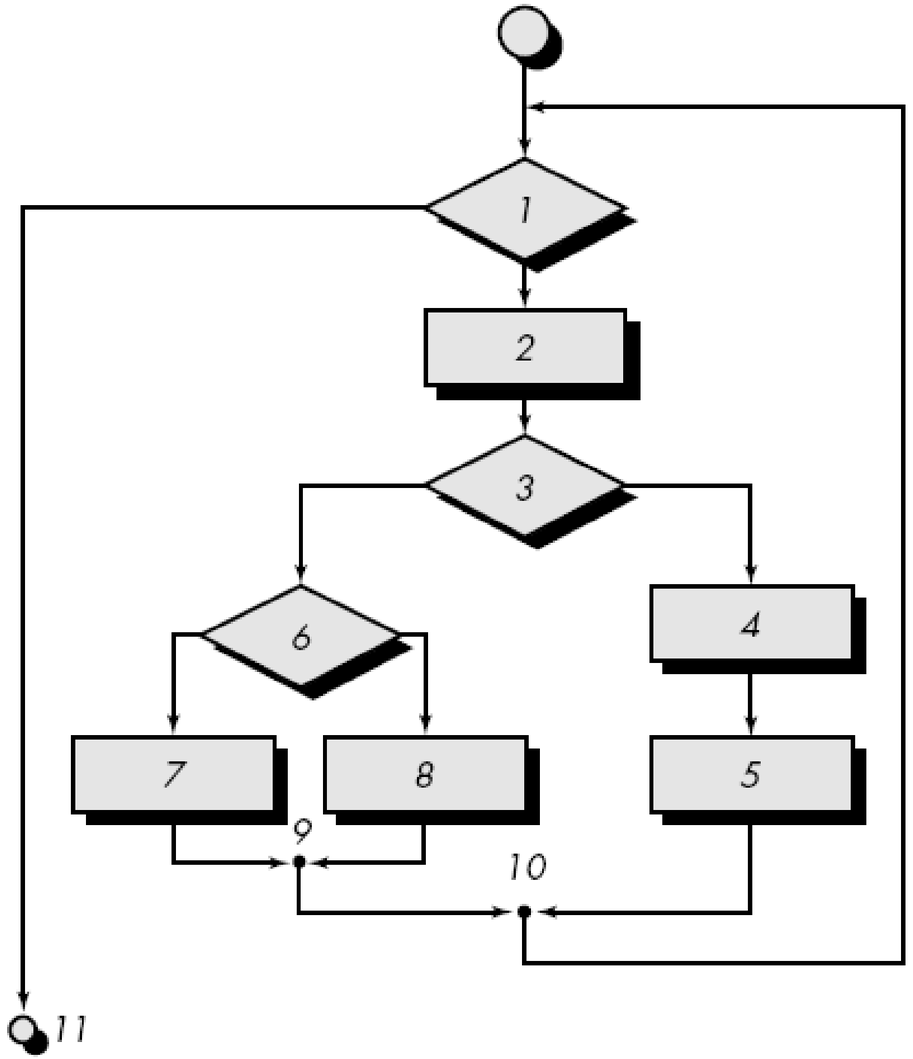
```
    {
1:    while (x == 0)
      {
2:      if (y == 0)
3:        z = 0;
        else
        {
4:        if (k == 0)
5:          z = 1;
          else
6:          x = 1;
7:      }
8:    }
9: }
```

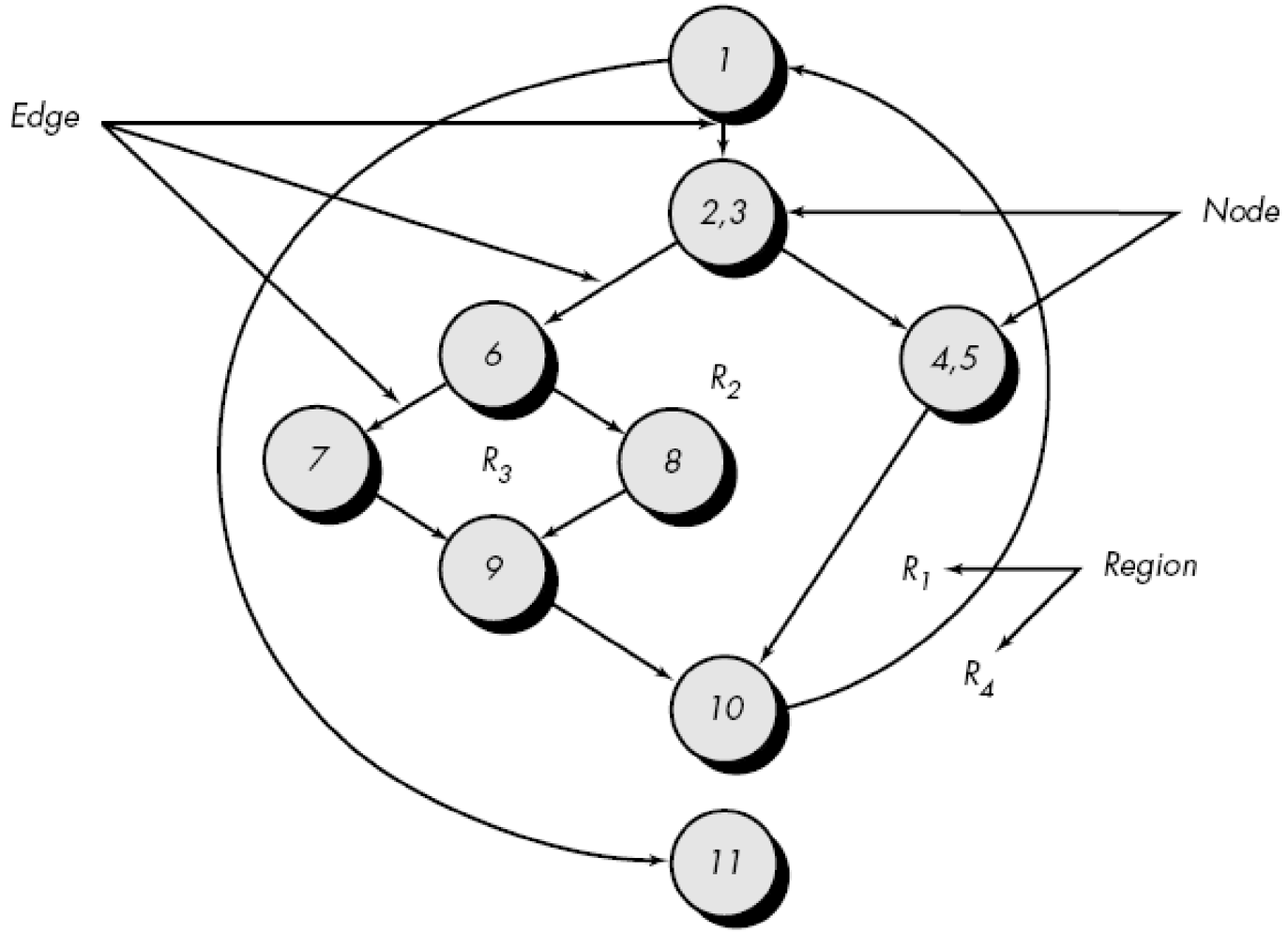


Đường độc lập

- **Đường độc lập** (***independent path***) **là một đường thi hành có đi qua ít nhất một cạnh mà chúng chưa được đi qua trong các đường thi hành độc lập trước đó.**

Đường độc lập





Đường độc lập 1: 1 – 11
Đường độc lập 2: 1 – 2 – 3 – 4 – 5 – 10 – 1 – 11
Đường độc lập 3: 1 – 2 – 3 – 6 – 8 – 9 – 10 – 1 – 11
Đường độc lập 4: 1 – 2 – 3 – 6 – 7 – 9 – 10 – 1 – 11

Đường độc lập

- **Tập cơ sở** (***basic set***) **của đồ thị dòng điều khiển bao gồm tất cả các đường độc lập.**
- **Thiết kế các *test-case* phải dựa vào tập cơ sở của đồ thị dòng điều khiển để kiểm thử tất cả các đường độc lập.**
  - **Mọi phát biểu thực thi trong chương trình phải được thực hiện ít nhất một lần.**
  - **Mọi điều kiện phải được thực hiện cả hai phía *true* và *false*.**

Độ phức tạp *cyclomatic*

- **Độ phức tạp *cyclomatic M* của một đoạn mã lệnh là số lượng các đường độc lập của đồ thị dòng điều khiển của đoạn mã này.**

Độ phức tạp *cyclomatic*

- Độ phức tạp *cyclomatic M* (số đường độc lập)

  M = E – N + 2P

  E = số cạnh của đồ thị

  N = số nút của đồ thị

  P = số thành phần liên thông.

  Đối với một đơn vị chương trình thì P = 1.

- Đối với đồ thị dòng điều khiển nhị phân

  M = N + 1

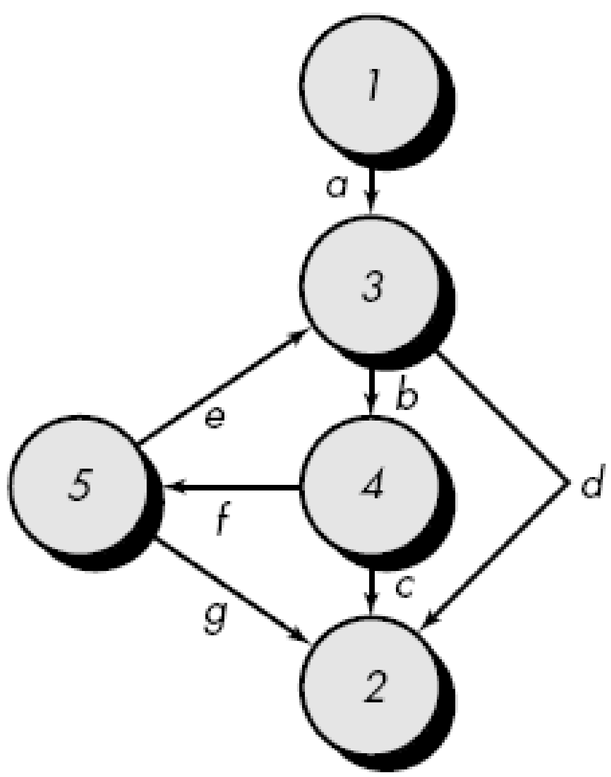
  N = số nút điều kiện rẽ nhánh nhị phân

Độ phức tạp *cyclomatic*

E = số cạnh = 9
N = số nút = 8
P = số thành phần liên thông = 1

M = E – N + 2P = 9 – 8 + 2 = 3

Độ phức tạp *cyclomatic*



Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

Node \ Connected to node	1	2	3	4	5	Connections
1			1			1 – 1 = 0
2						
3		1		1		2 – 1 = 1
4		1			1	2 – 1 = 1
5		1	1			2 – 1 = 1

Graph matrix

3 + 1 = 4 ← Cyclomatic complexity

Qui trình kiểm thử của *Tom McCabe*

- **Bước 1:** Xây dựng đồ thị dòng điều khiển của mã nguồn.
  - Biến đổi đồ thị dòng điều khiển thành đồ thị dòng điều khiển nhị phân.
  - Biến đổi đồ thị dòng điều khiển nhị phân thành đồ thị dòng điều khiển cơ bản.
- **Bước 2:** Xác định tập cơ sở của đồ thị dòng điều khiển cơ bản.
  - Tính độ phức tạp *cyclomatic M*.
  - Xác định tất cả các đường độc lập tuyến tính cơ bản, được gọi là **đường cơ bản** (*basis path*).

Qui trình kiểm thử của *Tom McCabe*

- **Bước 3:** Kiểm thử tất cả các đường độc lập tuyến tính cơ bản.
  - Tạo *test-case* cho mỗi đường độc lập tuyến tính cơ bản.
  - Thực hiện kiểm thử với mỗi *test-case*.
  - So sánh kết quả kiểm thử với kết quả mong muốn.
- **Bước 4:** Lập báo cáo kết quả kiểm thử để phản hồi cho những người liên quan.

Tìm các đường độc lập tuyến tính cơ bản

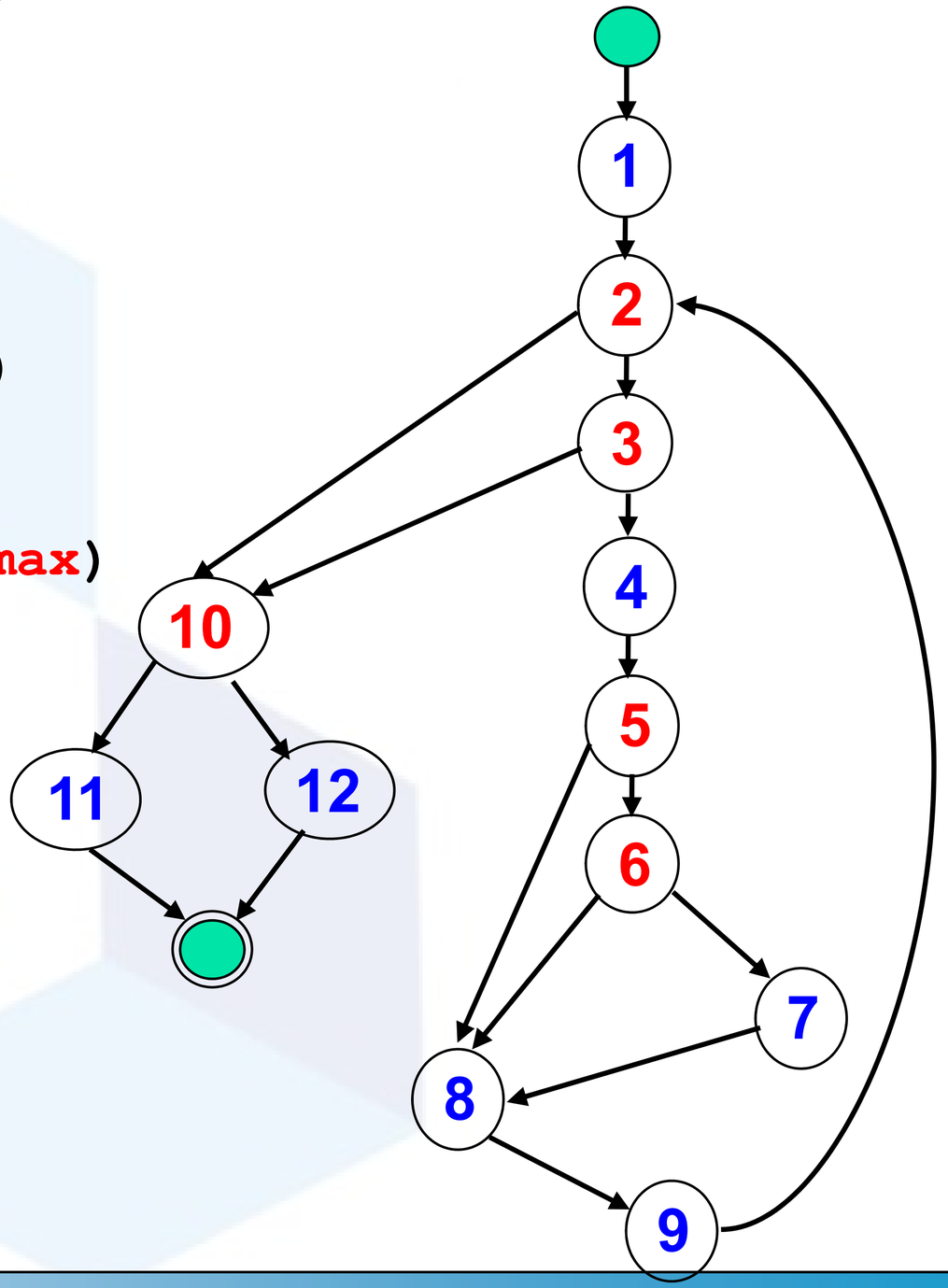
- **Sử dụng phép duyệt *DFS* (*Depth-First Search*) trên đồ thị dòng điều khiển cơ bản.**
  - **Bước 1: Bắt đầu từ nút hiện tại là nút bắt đầu.**
  - **Bước 2: Xét nút hiện tại:**
    - **Đánh dấu duyệt nút hiện tại.**
    - **Xét các cạnh kề của nút hiện tại: Nếu một cạnh kề chưa được duyệt thì duyệt cạnh kề này và đến nút kề này.**

- Nếu nút kề này chưa được duyệt thì nút này là nút hiện tại và đến bước 2.

- Nếu nút kề đã được duyệt hoặc là nút kết thúc thì ta đã tìm được một đường độc lập tuyến tính cơ bản.

Tìm các đường độc lập tuyến tính cơ bản

```
double average(double value[], double min,
double max, int& tcnt, int& vcnt)
{
1    double sum = 0;
1    int i = 1;
1    tcnt = vcnt = 0;
2,3  while (value[i] <> -999 && tcnt < 100)
     {
4      tcnt++;
5,6    if (min <= value[i] && value[i] <= max)
       {
7        sum += value[i];
7        vcnt++;
       }
8      i++;
9    }
10   if (vcnt > 0)
11       return sum / vcnt;
12   return -999;
}
```



Tìm các đường độc lập tuyến tính cơ bản

M = 5 + 1 đường thi hành độc lập tuyến tính cơ bản

P1: 1 → 2 → 10 → 11

p(11) = 11 p(10) = 10,11

P2: 1 → 2 → 10 → 12

p(12) = 12 p(2) = 2,p(10) p(1) = 1,p(2)

P3: 1 → 2 → 3 → 10 → 11

p(3) = 3,p(10)

P4: 1 → 2 → 3 → 4 → 5 → 8 → 9 → 2 → 10 → 11

p(9) = 9,p(2) p(8) = 8,p(9)

p(5) = 5,p(8)

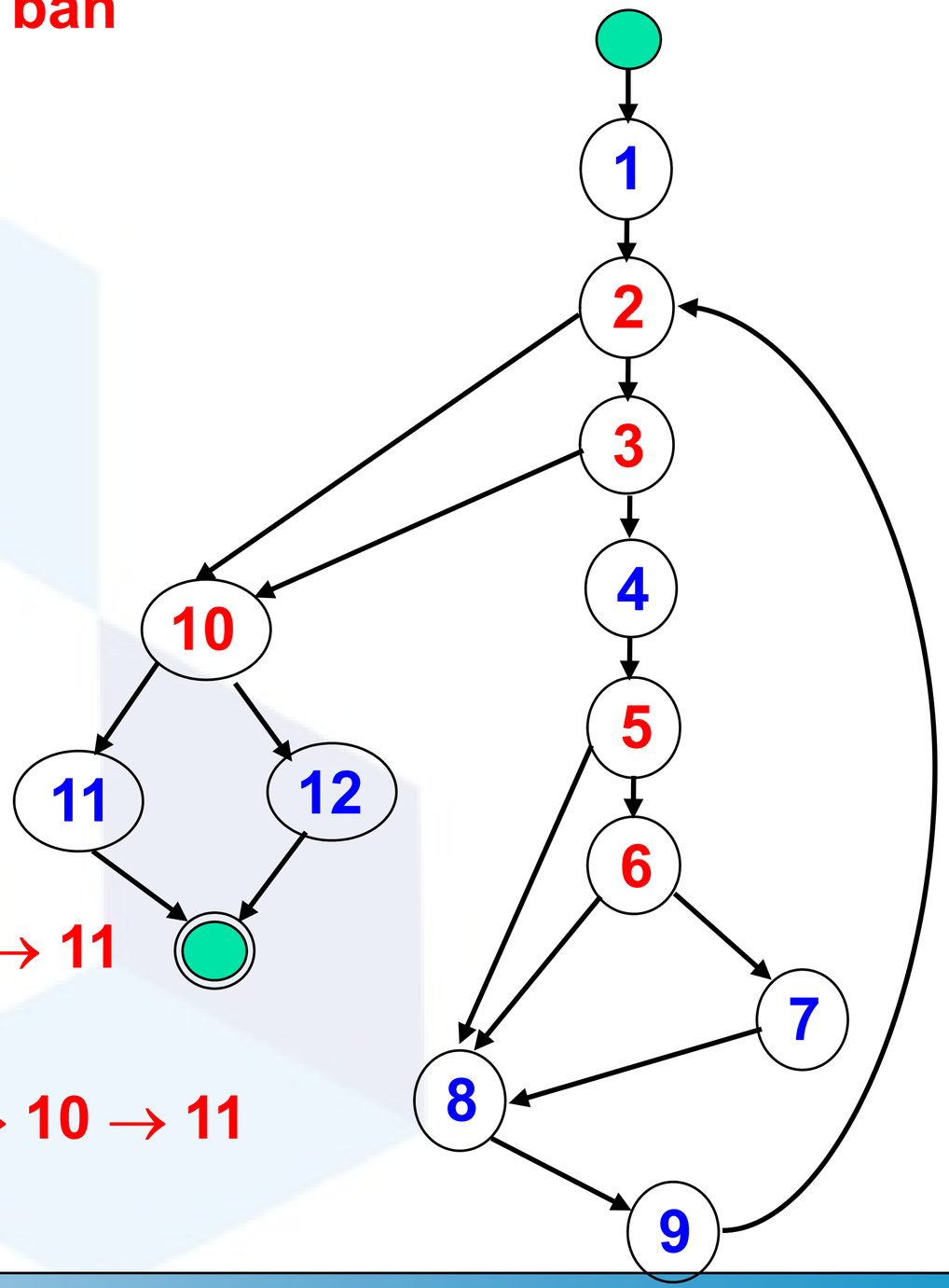
p(4) = 4,p(5)

P5: 1 → 2 → 3 → 4 → 5 → 6 → 8 → 9 → 2 → 10 → 11

p(6) = 6,p(8)

P6: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 2 → 10 → 11

p(7) = 7,p(8)



Tìm các đường độc lập tuyến tính cơ bản

Các test-case:

P1: 1 → 2 → 10 → 11 được kiểm như là một phần của P4, P5, P6

value[ ] có n giá trị với 2 ≤ n ≤ 100

value[n] = -999 value[i] <> -999 với 1 ≤ i < n

ví dụ: value[ ] = {1, 2, 3, 4, -999} với n = 5

P2: 1 → 2 → 10 → 12

value[ ] chỉ có 1 giá trị với n = 1

value[ ] = {-999}

P3: 1 → 2 → 3 → 10 → 11

value[ ] có nhiều hơn 100 giá trị với 100 giá trị đầu tiên khác -999

ví dụ: value[ ] = {1, 2, …, 100, 101, -999} với n = 102

Tìm các đường độc lập tuyến tính cơ bản

Các test-case:

P4: 1 → 2 → 3 → 4 → 5 → 8 → 9 → 2 → 10 → 11

value[ ] có n giá trị với 1 ≤ n < 100

value[n] = -999 value[i] < min và khác -999 với 1 ≤ i < n

ví dụ: value[ ] = {1, 2, 3, 4, -999} với n = 5 và min = 10

P5: 1 → 2 → 3 → 4 → 5 → 6 → 8 → 9 → 2 → 10 → 11

value[ ] có n giá trị với 1 ≤ n < 100

value[n] = -999 value[i] > max và khác -999 với 1 ≤ i < n

ví dụ: value[ ] = {21, 22, 23, 24, -999} với n = 5 và max = 20

P6: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 2 → 10 → 11

value[ ] có n giá trị với 1 ≤ n < 100

value[n] = -999 min ≤ value[i] ≤ max với 1 ≤ i < n

ví dụ: value[ ] = {11, 12, 13, 14, -999} với n = 5, min = 10 và max = 20

Mức phủ mã lệnh

- **Mức phủ mã lệnh** (*code coverage*) là độ đo được dùng để mô tả mã nguồn của một chương trình được kiểm thử bởi một bộ kiểm thử cụ thể.
- Một chương trình có mức phủ mã lệnh cao nếu toàn bộ chương trình đều được kiểm thử và có thể có ít lỗi sai.
- Có nhiều độ đo khác nhau được dùng để tính toán mức phủ mã lệnh, thông thường là phần trăm các chương trình con và phần trăm các phát biểu của chương trình được kiểm thử với bộ kiểm thử (*test suite*).

Mức phủ mã lệnh

- Các loại mức phủ
  - Các mức phủ cơ bản
    - Mức phủ hàm
    - Mức phủ phát biểu
    - Mức phủ rẽ nhánh
    - Mức phủ điều kiện
  - Mức phủ điều kiện / quyết định thay đổi
  - Mức phủ đa điều kiện
  - Mức phủ giá trị tham số

Mức phủ mã lệnh

- **Các loại mức phủ**
  - **Các mức phủ khác**
    - **Mức phủ đường thi hành**
    - **Mức phủ vào / ra**
    - **Mức phủ vòng lặp**
    - **Mức phủ trạng thái**

Các mức phủ cơ bản

- **Mức phủ hàm** (*function coverage*): Mỗi hàm (hoặc chương trình con) trong chương trình đều được thực hiện?
- **Mức phủ phát biểu** (*statement coverage*, *node coverage*): Mỗi phát biểu trong chương trình đều được thực hiện?
- **Mức phủ rẽ nhánh** (*branch coverage*): Mỗi nhánh của mỗi cấu trúc điều kiện (mỗi cạnh của nút điều kiện của đồ thị dòng điều khiển) đều được thực hiện?
- **Mức phủ điều kiện** (*condition coverage*, *predicate coverage*): mỗi biểu thức con luận lý (*boolean subexpression*) đều được định trị *true* và *false*.

Các mức phủ cơ bản

❖ Ví dụ

```
int Func (int x, int y)
{
   int z = 0;
   if ((x > 0) && (y > 0))
   {
      z = x;
   }
   return z;
}
```

- **Mức phủ hàm được thỏa mãn khi hàm *Func*() phải được thực hiện ít nhất một lần.**

Các mức phủ cơ bản

- Mức phủ phát biểu được thỏa mãn khi gọi hàm *Func*(1,1) thì mọi phát biểu của hàm đều được thực hiện, kể cả phát biểu *z* = *x*.
- Mức phủ rẽ nhánh được thỏa mãn khi gọi hàm *Func*(1,1) thì điều kiện *if* là *true* và *z* = *x* được thực hiện và khi gọi hàm *Func*(1,0) thì điều kiện *if* là *false* và *z* = *x* không được thực hiện.
- Mức phủ điều kiện được thỏa mãn khi gọi hàm *Func*(1,1), *Func*(1,0) và *Func*(0,0). (*x* > 0) là *true* trong trường hợp 1, 2 và là *false* trong trường hợp 3. (y > 0) là *true* trong trường hợp 1 và là *false* trong trường hợp 2, 3.

Các mức phủ cơ bản

- **Mức phủ điều kiện có thể không bao hàm mức phủ rẽ nhánh.**
  - **Ví dụ xét mã lệnh sau:**

```
if a and b then
```

  - **Mức phủ điều kiện có thể được thỏa mãn với hai *test-case* sau:**

    tets1: a = *true* b = *false*

    test2: a = *false* b = *true*

  - **Tuy nhiên, hai *test-case* này không thỏa mãn mức phủ rẽ nhánh vì chúng chỉ kiểm thử một nhánh ứng với điều kiện là *false*.**

Mức phủ điều kiện / quyết định thay đổi

- **Sự kết hợp mức phủ hàm và mức phủ rẽ nhánh được gọi là mức phủ quyết định (*decision coverage*, *edge coverage*). Điều này đòi hỏi:**
  - **Mọi điểm vào và điểm ra của chương trình phải được kiểm thử ít nhất một lần.**
  - **Mọi nhánh của quyết định trong chương trình phải được kiểm thử ít nhất một lần.**
- **Mức phủ điều kiện / quyết định (*condition / decision coverage*) đòi hỏi mức phủ điều kiện và mức phủ quyết định phải được thỏa mãn.**

Mức phủ điều kiện / quyết định thay đổi

- **Tuy nhiên, các chương trình ứng dụng có độ an toàn cao (*safety-critical application*), ví dụ phần mềm về hàng không, đòi hỏi phải thỏa mãn mức phủ điều kiện / quyết định thay đổi (*MC/DC – Modified Condition / Decision Coverage*).**
- **Mức phủ này mở rộng mức phủ điều kiện / quyết định bằng cách mỗi điều kiện phải ảnh hưởng đến đầu ra của quyết định một cách độc lập.**

Mức phủ điều kiện / quyết định thay đổi

- Ví dụ xét mã lệnh sau:

  if (a or b) and c then

- Mức phủ điều kiện / quyết định được thỏa mãn với hai *test-case* sau:

  test1: a = *true* b = *true* c = *true*

  test2: a = *false* b = *false* c = *false*

- Tuy nhiên, các kiểm thử này không thỏa mãn mức phủ điều kiện / quyết định thay đổi, bởi vì *b* = *true* trong *test*1 và *c* = *false* trong *test*2 không ảnh hưởng đến kết xuất.

Mức phủ điều kiện / quyết định thay đổi

- Các test-case sau đây thỏa mãn mức phủ MC/DC:

  test1: a = *false* b = *false* c = *true*

  test2: a = *true* b = *false* c = *true*

  test3: a = *false* b = *true* c = *true*

  test4: a = *false* b = *true* c = *false*

Mức phủ đa điều kiện

- **Mức phủ đa điều kiện** (***multiple condition coverage***) **đòi hỏi tất cả các tổ hợp của các điều kiện trong một quyết định phải được kiểm thử.**

Mức phủ đa điều kiện

- Ví dụ xét mã lệnh sau:

  if (a or b) and c then

- Mức phủ đa điều kiện được thỏa mãn với 8 *test-case* sau:

  test1: a = *false* b = *false* c = *false*

  test2: a = *false* b = *false* c = *true*

  test3: a = *false* b = *true* c = *false*

  test4: a = *false* b = *true* c = *true*

  tets5: a = *true* b = *false* c = *false*

  test6: a = *true* b = *false* c = *true*

  test7: a = *true* b = *true* c = *false*

  test8: a = *true* b = *true* c = *true*

Mức phủ giá trị tham số

- **Mức phủ giá trị tham số** (***parameter value coverage – PVC***) **đòi hỏi toàn bộ miền trị của tất cả các tham số của một chương trình con phải được kiểm thử.**

Mức phủ giá trị tham số

- **Ví dụ miền trị của một chuỗi bao gồm 7 trường hợp (chuỗi có thể có rất nhiều ký tự):**
  - **rỗng (*null*)**
  - **chuỗi rỗng (*empty string*)**
  - **chuỗi khoảng trắng (*space*, *tabs*, *newline*)**
  - **chuỗi hợp lệ (*valid string*)**
  - **chuỗi không hợp lệ (*invalid string*)**
  - **chuỗi 1-*byte***
  - **chuỗi 2-*byte***
- **Nếu ta chỉ kiểm thử một trường hợp thì mức phủ *PVC* là 1 / 7 = 14.2%.**

Các mức phủ khác

- **Mức phủ đường cơ bản** (*basis path coverage*): Tất cả các đường cơ bản của một đoạn mã đều được thực hiện?
- **Mức phủ vào / ra** (*entry / exit coverage*): Tất cả gọi chương trình con (*call*) và trở về (*return*) của một chương trình con đều được thực hiện?
- **Mức phủ vòng lặp** (*loop coverage*): Tất cả các vòng lặp đều được thực hiện ít nhất 0 lần, 1 lần và nhiều lần?
- **Mức phủ trạng thái** (*state coverage*): Tất cả các trạng thái của một đối tượng đều xảy ra?

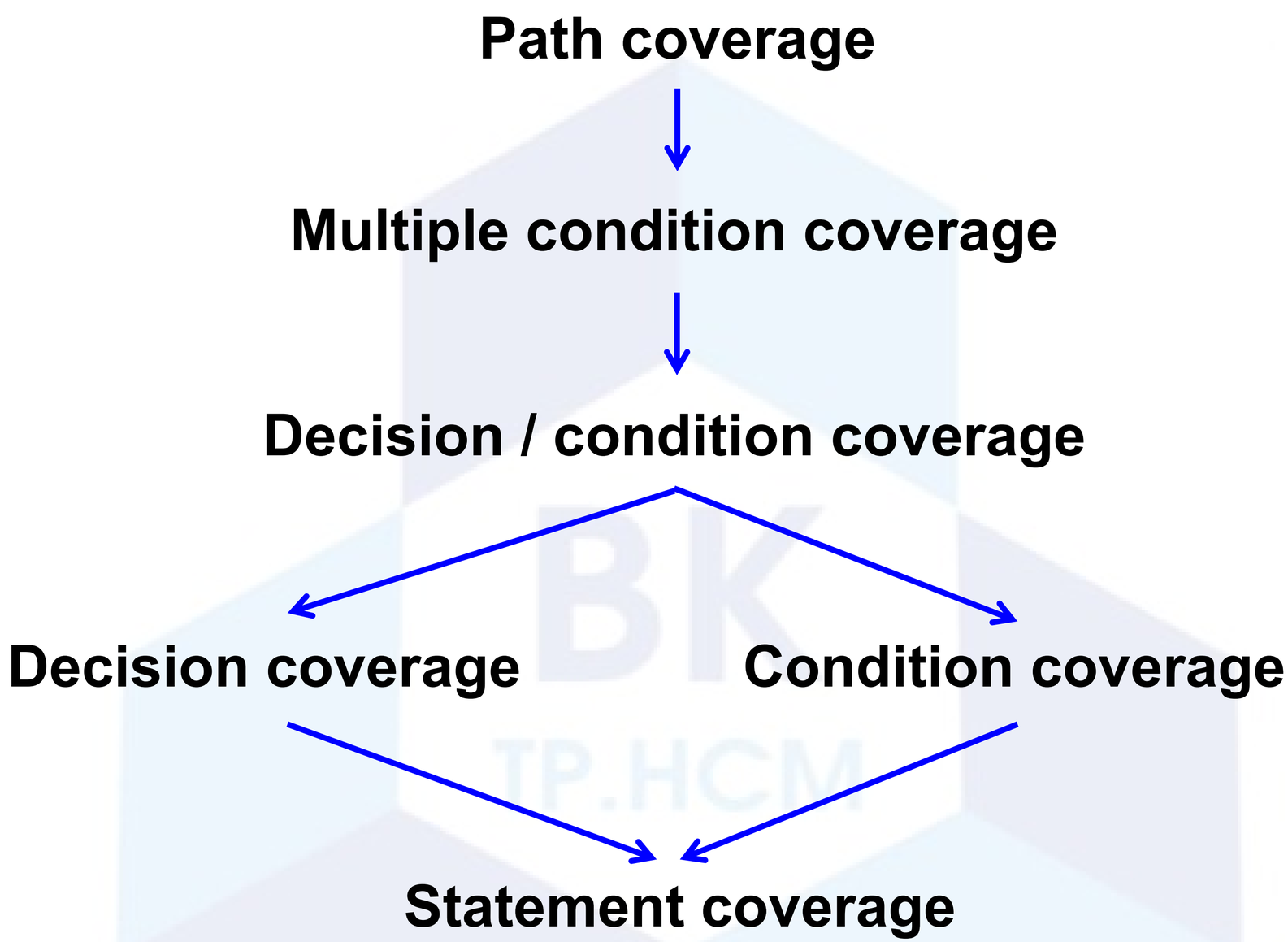
Các mức phủ khác

- Các ứng dụng có độ an toàn cao thường đòi hỏi kiểm thử phải đạt được 100% của một mức phủ nào đó.
- Một số mức phủ có liên quan với nhau.
  - Mức phủ đường cơ bản bao hàm mức phủ quyết định, điều kiện, phát biểu, vào / ra.
  - Mức phủ quyết định bao hàm mức phủ phát biểu, vì mọi phát biểu là một phần của một nhánh của một quyết định.

Các mức phủ khác

- Mức phủ đường cơ bản thường không thực tế và không khả thi.
  - Một chương trình con bao gồm một chuỗi các điều kiện sẽ tạo thành các đường cơ bản; các cấu trúc lặp (*for*, *while*, *do..while*) có thể dẫn đến một số vô hạn các đường cơ bản.
  - Nhiều đường cơ bản cũng có thể không khả thi, bởi vì chúng không có dữ liệu nhập để thực hiện một phần của đường cơ bản này (ví dụ P1 được kiểm thử như là một phần của P4, P5, P6 trong ví dụ trước).
  - Có thể đạt được mức phủ rẽ nhánh mà không cần phải đạt được mức phủ đường cơ bản.

Phân cấp các mức phủ

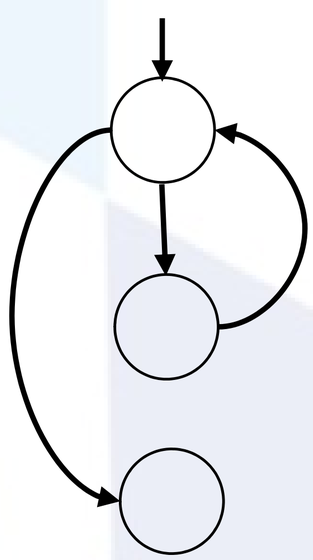


Mức phủ vòng lặp

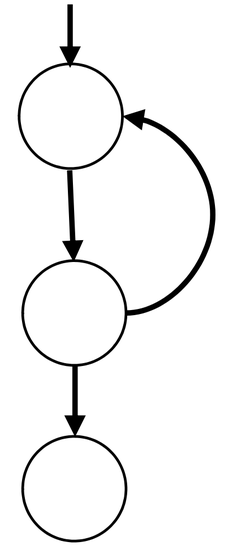
- Thân của vòng lặp (*for*, *while*, *do..while*) thường bao gồm nhiều lệnh và vòng lặp có thể được thực hiện nhiều lần (rất lớn). Chi phí của kiểm thử mức phủ vòng lặp có thể rất lớn.
- Có 4 trường hợp kiểm thử của mức phủ vòng lặp
  - Vòng lặp đơn (*simple loop*)
  - Các vòng lặp lồng nhau (*nested loops*)
  - Các vòng lặp tuần tự (*concatenated loops*)
  - Các vòng lặp không có cấu trúc (*unstructured loops*)

Mức phủ vòng lặp

- **Vòng lặp đơn** (***simple loop***)
  - Gọi *n* là số lần lặp tối đa của vòng lặp.
  - Tạo các *test-case* để thực hiện vòng lặp:
    - Test1: 0 lần
    - Test2: 1 lần
    - Test3: 2 lần
    - Test4: *k* lần với $k < n - 1$
    - Test5: $n - 1$ lần
    - Test6: *n* lần
    - Test7: $n + 1$ lần



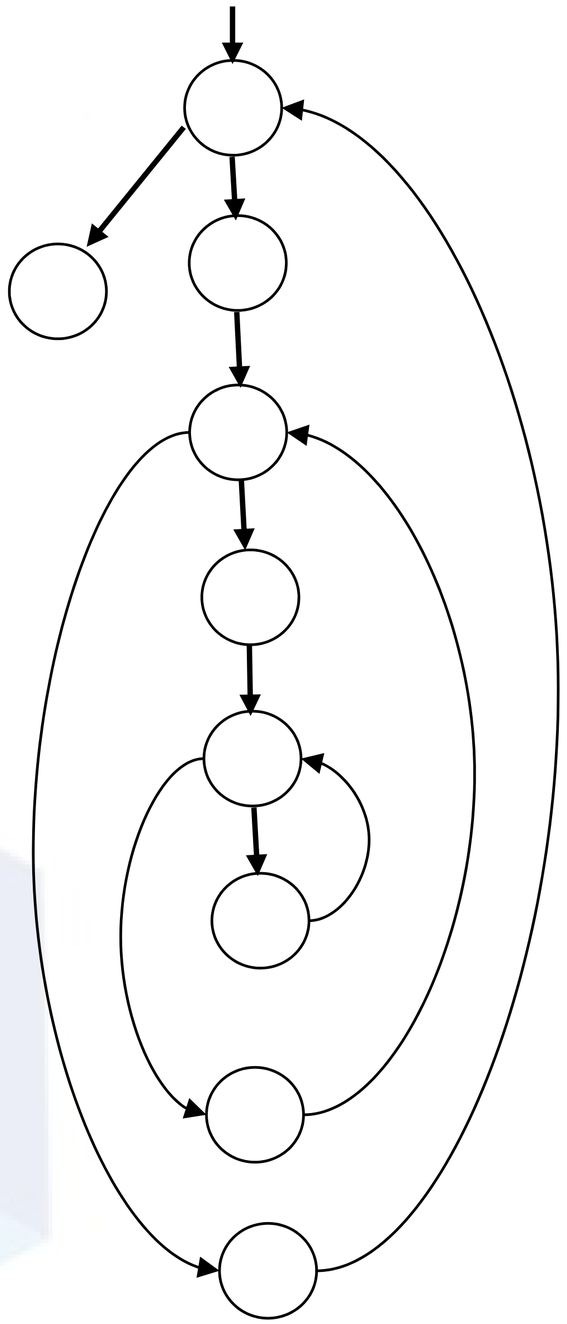
while



do..while

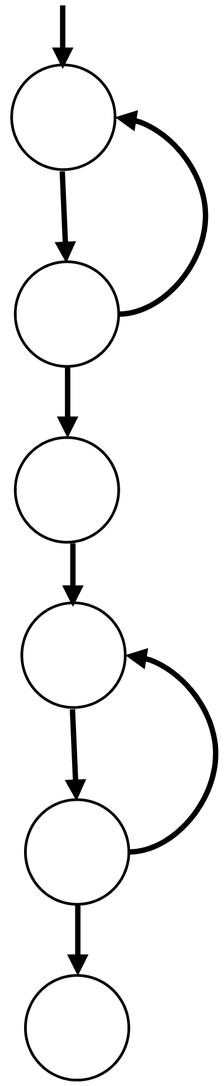
Mức phủ vòng lặp

- **Các vòng lặp lồng nhau (*nested loops*)**
  - **Kiểm thử từ vòng lặp trong cùng đến vòng lặp ngoài cùng.**
  - **Kiểm thử vòng lặp hiện tại:**
    - **Chạy các vòng lặp ngoài với số lần lặp ít nhất.**
    - **Chạy các vòng lặp nằm trong (nếu có) với số lần lặp tiêu biểu.**
    - **Chạy vòng lặp hiện tại với 7 *test-case* giống như của vòng lặp đơn.**



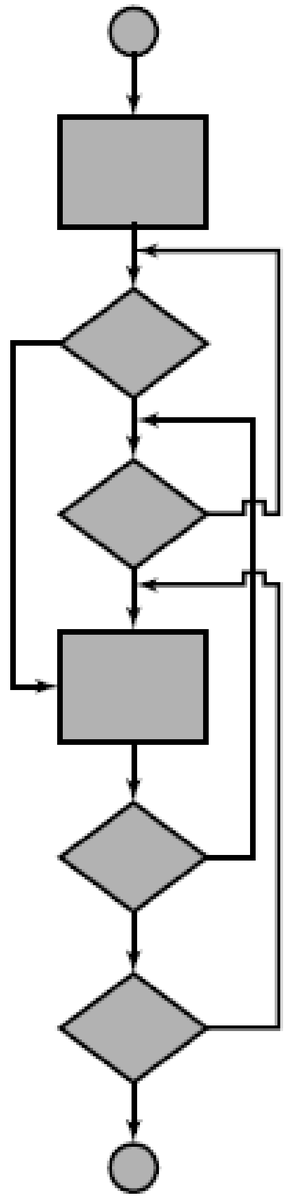
Mức phủ vòng lặp

- **Các vòng lập tuần tự (*concatenated loops*)**
  - **Kiểm thử tuần tự từ vòng lặp trên cùng đến vòng lặp dưới cùng một cách độc lập nhau.**
  - **Kiểm thử vòng lặp hiện tại với 7 *test-case* giống như của vòng lặp đơn.**



Mức phủ vòng lặp

- **Các vòng lặp không có cấu trúc (*unstructured loops*)**
  - **Đoạn mã được viết theo lập trình *if – goto*.**
  - **Viết lại đoạn mã này theo lập trình có cấu trúc hoặc lập trình hướng đối tượng. Sử dụng các cấu trúc lặp:**
    - **Vòng lặp đơn**
    - **Các vòng lặp lồng nhau**
    - **Các vòng lặp tuần tự**



Unstructured loops

Kiểm thử dòng dữ liệu

- **Kiểm thử dòng dữ liệu (*data flow testing*) là một chiến lược kiểm thử bằng cách chọn các đường thi hành thông qua dòng điều khiển của chương trình sao cho chúng bao gồm chuỗi các sự kiện có liên quan đến trạng thái của các biến hoặc các đối tượng dữ liệu.**
- **Kiểm thử dòng dữ liệu tập trung vào:**
  - **các chỗ mà các biến được gán giá trị**
  - **các chỗ mà các biến được sử dụng**

Kiểm thử dòng dữ liệu

- Kiểm thử dòng dữ liệu để xác định các vấn đề:
  - Một biến được khai báo nhưng không bao giờ được sử dụng trong chương trình.
  - Một biến được sử dụng nhưng chưa được khai báo.
  - Một biến được khai báo nhiều lần trước khi nó được sử dụng.
  - Hủy bỏ một biết trước khi nó được sử dụng.

Các mức kiểm thử dòng dữ liệu

- **Kiểm thử dòng dữ liệu tĩnh (*static data flow testing*)**
  - Xác định các sai sót (*defect*) tiềm ẩn, được gọi là sự bất thường của dòng dữ liệu (*data flow anomaly*).
  - Phân tích mã nguồn.
  - Không chạy mã nguồn.
- **Kiểm thử dòng dữ liệu động (*dynamic data flow testing*)**
  - Cần phải chạy chương trình thật.
  - Tương tự với kiểm thử dòng điều khiển.
    - Xác định và thực hiện các đường thi hành dựa vào các tiêu chí của dòng dữ liệu.

Tầm vực của biến

- **Tầm vực** (*scope*) của một biến là một đoạn chương trình mà biến này được sử dụng.
- **Tầm vực toàn cục** (*global scope*) là đoạn chương trình nằm ngoài tất cả các hàm.
  - Sau khi khai báo một biến thuộc tầm vực toàn cục, biến này được gọi là biến toàn cục (*global variable*) và nó có thể được sử dụng trong tất cả các hàm.

Tầm vực của biến

- **Tầm vực cục bộ** (***local scope***) **là một khối phát biểu bao gồm các phát biểu được ghi trong hai dấu { }.**
  - **Khối phát biểu có thể là thân hàm, hoặc là một khối nằm trong thân hàm, hoặc là một khối nằm trong một khối khác.**
  - **Nếu một khối *A* nằm trong một khối *B* thì khối *A* được gọi là khối trong (*inner block*) và khối *B* được gọi là khối ngoài (*outer block*).**
  - **Một khối phát biểu có thể chứa các phát biểu khai báo các biến và các biến này được gọi là biến cục bộ (*local variable*) trong khối này.**

Tầm vực của biến

- Một biến cục bộ được sử dụng bắt đầu từ phát biểu khai báo biến này cho đến ký hiệu kết thúc (}) của khối trong cùng chứa phát biểu khai báo biến này, kể cả các khối trong của khối này.
- Các tham số hình thức của một hàm là các biến cục bộ và chúng được sử dụng trong thân của hàm này.

Tầm vực của biến

```
int x, y;          // khai báo biến x, y toàn cục
void func()
{
   int x;          // khai báo biến x cục bộ
   ...
   ::x = 10;       // gán giá trị cho biến x toàn cục
   y = 5;          // gán giá trị cho biến y toàn cục
   {
      x = 1;       // gán giá trị cho biến x cục bộ
      int y = 6; // khai báo và gán trị cho biến y cục bộ
      ...
   }               // hủy biến y cục bộ
   ...
   cout << x;      // sử dụng biến x cục bộ
   ...
}                  // hủy biến x cục bộ
```

Định nghĩa và sử dụng biến

- **Định nghĩa một biến (*define*)**
  - **Khai báo biến: tên biến, kiểu dữ liệu, giá trị ban đầu.**
  - **Gán giá trị cho biến: phát biểu gán, phát biểu nhập.**
- **Sử dụng một biến (*use*, *reference*)**
  - **Tham chiếu biến: biểu thức, phát biểu gọi.**
  - **Xuất giá trị của biến: phát biểu xuất.**

Sơ đồ chuyển trạng thái của một biến

- **Sơ đồ chuyển trạng thái (*state-transition diagram*) của một biến được dùng để mô hình hóa một biến của chương trình để xác định sự bất thường của dòng dữ liệu.**

Sơ đồ chuyển trạng thái của một biến

❖ Các thành phần của sơ đồ chuyển trạng thái

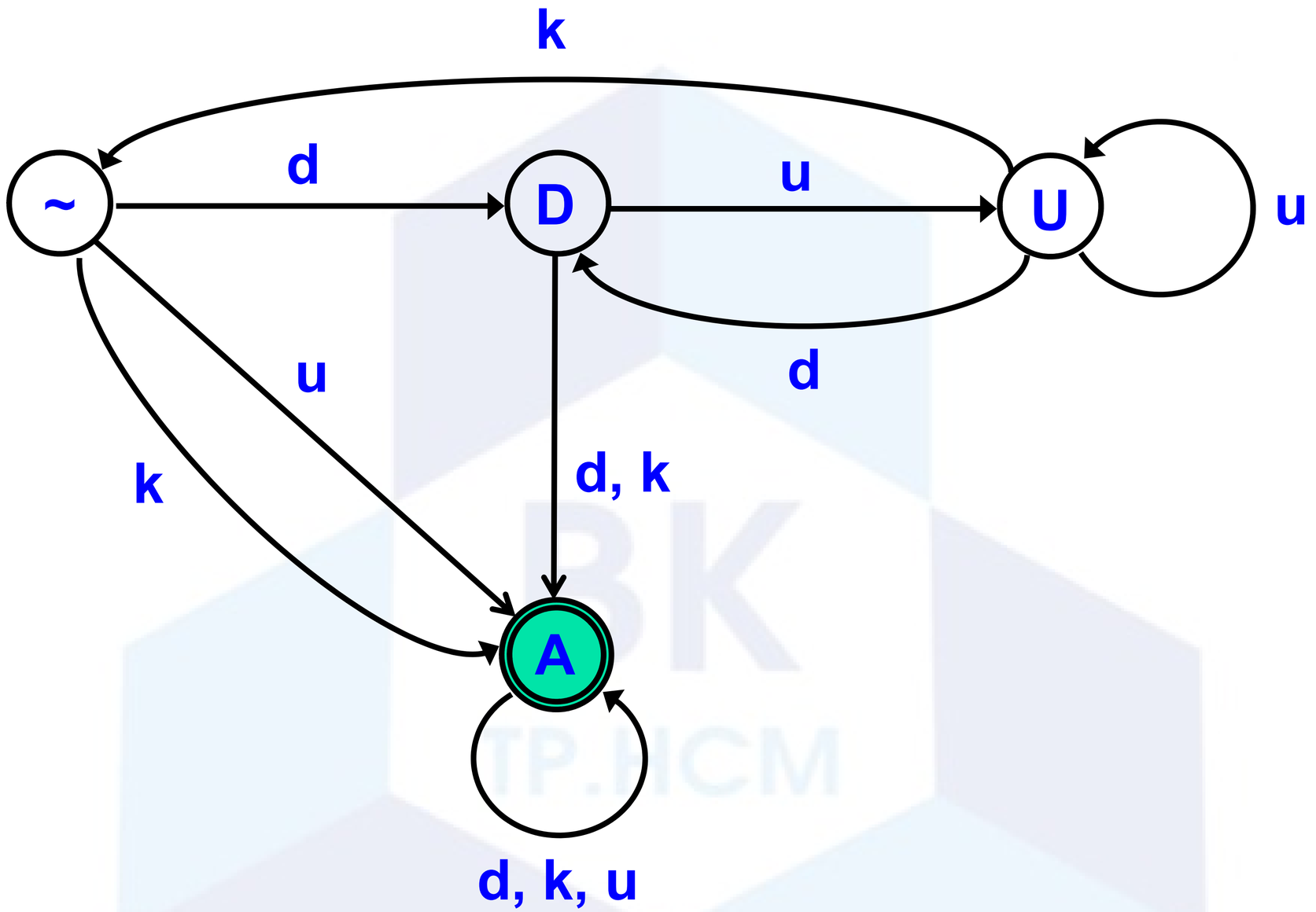
▸ Các trạng thái

- **~ (*Undefined*): chưa được định nghĩa.**
- **D (*Defined*): được định nghĩa nhưng chưa được sử dụng.**
- **U (*Used*): được định nghĩa và được sử dụng.**
- **A (*Abnormal*): bất thường.**

▸ Các hành động

- **d (*define*): định nghĩa biến.**
- **u (*use*): sử dụng biến.**
- **k (*kill*): hủy bỏ biến.**

Sơ đồ chuyển trạng thái của một biến



Các trường hợp bất thường của một biến

- Có bốn trường hợp bất thường khi sử dụng một biến.
  - TH1: Một biến đã được định nghĩa, sau đó định nghĩa lại biến này.
  - TH2: Một biến chưa được định nghĩa nhưng biến này được sử dụng.
  - TH3: Một biến đã được định nghĩa nhưng không được sử dụng.
  - TH4: Một biến chưa được định nghĩa, sau đó hủy bỏ biến này.

Các trường hợp bất thường của một biến

- **Ví dụ:**
  - **TH1: Một biến đã được định nghĩa, sau đó định nghĩa lại biến này.**

```
x = func1(y); // phát biểu 1
x = func2(z); // phát biểu 2
```

  - **Cách giải quyết:**
    - **Phát biểu 1 là dư thừa.**
    - **Phát biểu 1 bị sai: `v = func1(y)`**
    - **Phát biểu 2 bị sai: `v = func2(z)`**
    - **Xen một phát biểu vào giữa hai phát biểu này, ví dụ:**

```
v = func3(x)
```

Các trường hợp bất thường của một biến

- **Ví dụ:**
  - **TH2: Một biến chưa được định nghĩa nhưng biến này được sử dụng.**

```
x = x * y;  // biến y chưa được định nghĩa
```

  - **Cách giải quyết:**
    - **Định nghĩa biến y trước phát biểu này.**
    - **Sửa lại biến y vì viết nhầm.**
    - **Người lập trình muốn sử dụng giá trị mặc định mà trình biên dịch gán cho một biến khi khai báo biến, nhưng trình biên dịch không làm điều này.**

Các trường hợp bất thường của một biến

- **Ví dụ:**
  - **TH3: Một biến đã được định nghĩa nhưng không được sử dụng.**

```
x = func(y); // sau đó x không được sử dụng
```

  - **Cách giải quyết:**
    - **Phát biểu gán là dư thừa.**
    - **Phải thêm phát biểu sử dụng biến x.**

Sơ đồ chuyển trạng thái của một biến

- **Chuỗi trạng thái** (***state sequence***) **của một biến được biểu diễn bởi một chuỗi các hành động đối với biến này, bắt đầu từ trạng thái chưa được định nghĩa (~).**
- **Sơ đồ chuyển trạng thái cho thấy bốn trường hợp bất thường của một biến nếu chuỗi trạng thái của biến này có chứa:**
  - **TH1: dd**
  - **TH2: ~u**
  - **TH3: dk**
  - **TH4: ~k**

Sơ đồ chuyển trạng thái của một biến

- **Chuỗi trạng thái của một biến có thể chứa:**
  - **dd: biến đã được định nghĩa, sau đó định nghĩa lại, có sự bất thường.**
  - **du: biến đã được định nghĩa, sau đó sử dụng.**
  - **dk: biến đã được định nghĩa, sau đó hủy bỏ biến này, có sự bất thường.**
  - **ud: biến được sử dụng, sau đó gán giá trị mới cho biến.**
  - **uu: biến được sử dụng, sau đó sử dụng tiếp biến này.**
  - **uk: biến được sử dụng, sau đó hủy bỏ biến này.**
  - **kd: biến bị hủy bỏ, sau đó định nghĩa lại biến này.**
  - **ku: biến bị hủy bỏ, sau đó sử dụng biến này, có sự bất thường.**
  - **kk: biến bị hủy bỏ, sau đó hủy bỏ tiếp biến này, có sự bất thường.**

Sơ đồ chuyển trạng thái của một biến

- Phát hiện sự bất thường của dòng dữ liệu thông qua việc chỉnh sửa chương trình.
  - Chỉnh sửa chương trình hoặc thêm vào đoạn mã mới để giám sát các trạng thái của các biến.
  - Nếu chuỗi trạng thái của một biến chứa ***dd***, ***~u***, ***dk*** hoặc ***~k***, thì xảy ra sự bất thường của dòng dữ liệu.
  - Tìm hiểu nguyên nhân gây ra sự bất thường của dòng dữ liệu.
  - Chỉnh sửa chương trình hiện tại để không còn sự bất thường của dòng dữ liệu.

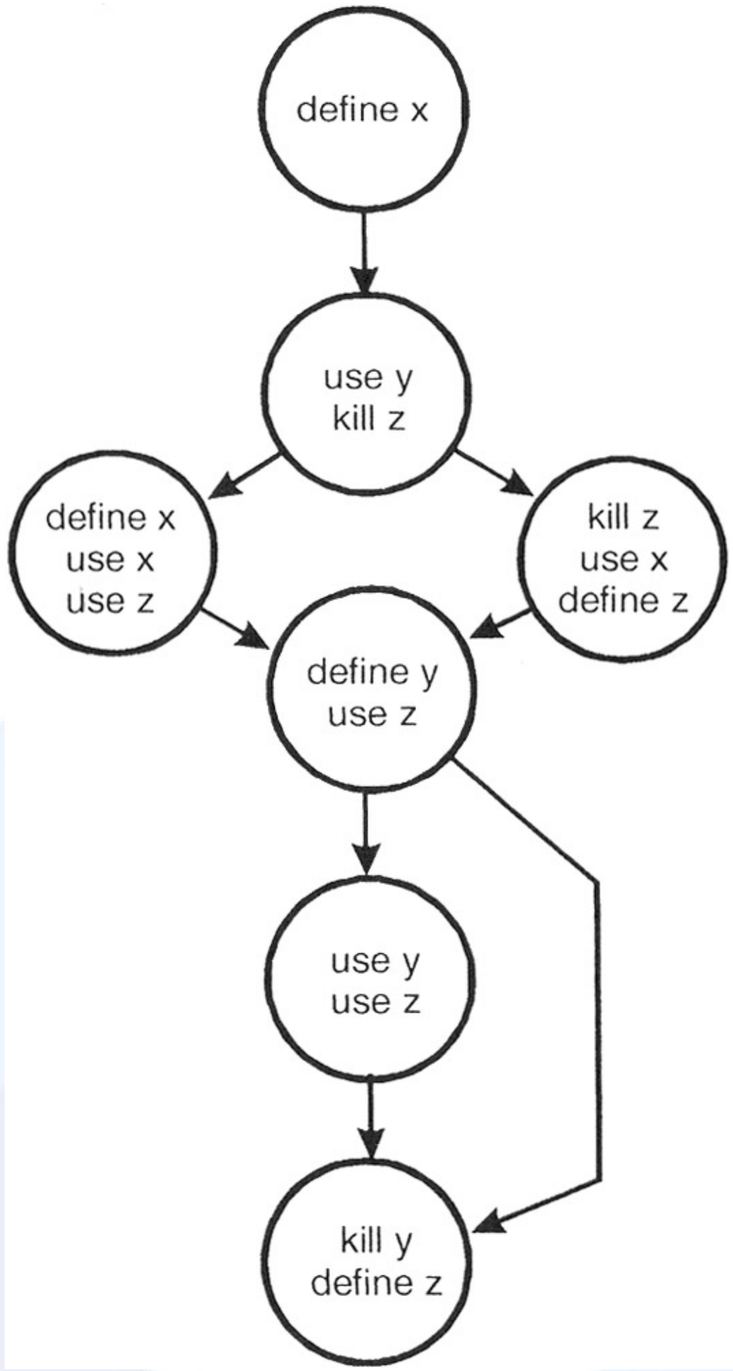
Đồ thị dòng dữ liệu

- Đồ thị dòng dữ liệu của chương trình có thể được xây dựng từ các cấu trúc điều khiển của chương trình.
- **Đồ thị dòng dữ liệu** (*data flow graph*) của chương trình là một đồ thị biểu diễn các chuỗi trạng thái của các biến trong chương trình này.

Đồ thị dòng dữ liệu

- **Đồ thị dòng dữ liệu *G* = (*N*, *E*) của chương trình bao gồm tập nút *N* và tập cạnh *E*.**
  - **Có 5 loại nút:**
    - **Nút bắt đầu** (***entry node***)
    - **Nút kết thúc** (***exit node***)
    - **Nút quyết định** (***decision node***) **chứa một chuỗi các tác vụ *u* đối với các biến trong điều kiện rẽ nhánh, nút này có hai hoặc nhiều nhánh (ví dụ *if* và *switch*).**
    - **Nút kết nối** (***connection node***) **thường không tác vụ nào và biểu diễn điểm nối của nhiều nhánh điều khiển.**
    - **Nút tác vụ** (***action node***) **chứa một chuỗi các tác vụ đối với các biến.**

Đồ thị dòng dữ liệu



Qui trình kiểm thử dòng dữ liệu

- **Bước 1: Xây dựng đồ thị dòng điều khiển của mã nguồn.**
  - **Biến đổi đồ thị dòng điều khiển thành đồ thị dòng điều khiển nhị phân.**
  - **Biến đổi đồ thị dòng điều khiển nhị phân thành đồ thị dòng điều khiển cơ bản.**

Qui trình kiểm thử dòng dữ liệu

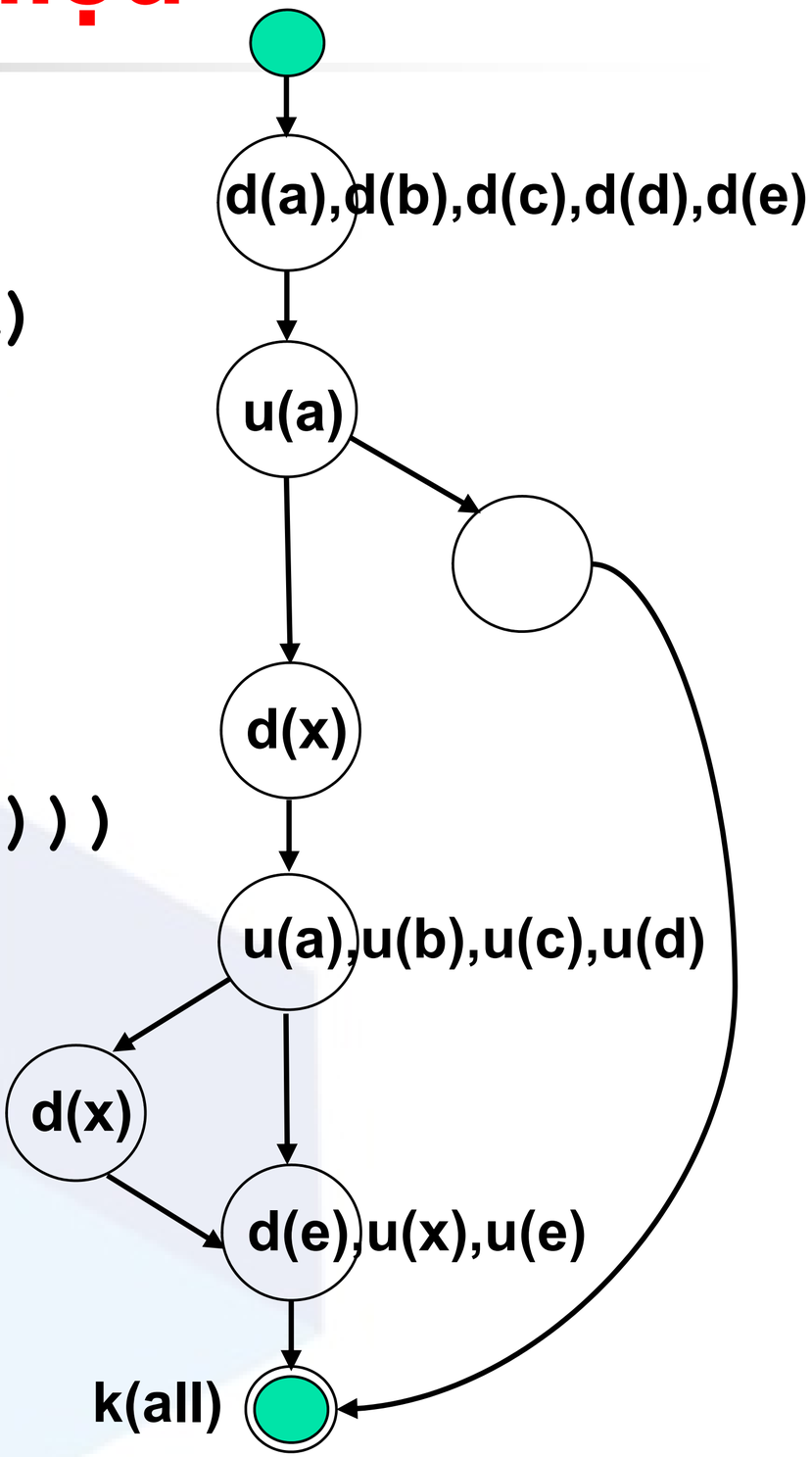
- **Bước 2:** Xây dựng đồ thị dòng dữ liệu từ đồ thị dòng điều khiển cơ bản.
  - Tính độ phức tạp *cyclomatic M*.
  - Xác định tất cả các đường độc lập tuyến tính cơ bản.
- **Bước 3:** Kiểm thử tất cả các đường độc lập tuyến tính cơ bản.
  - Xác định chuỗi trạng thái của mỗi biến trong mỗi đường độc lập tuyến tính cơ bản.
  - Nếu chuỗi trạng thái của một biến chứa *dd*, *~u*, *dk* hoặc *~k*, thì xảy ra sự bất thường của quá trình sử dụng của biến này.

Qui trình kiểm thử dòng dữ liệu

- **Bước 4:** Lập báo cáo kết quả kiểm thử để phản hồi cho những người liên quan.

Qui trình kiểm thử dòng dữ liệu

```
float foo(int a, int b, int c, int d)
{
   float e;
   if (a == 0)
      return 0;
   int x = 0;
   if ((a == b) || ((c == d) && bug(a)))
      x = 1;
   e = 1/x;
   return e;
}
```



Qui trình kiểm thử dòng dữ liệu

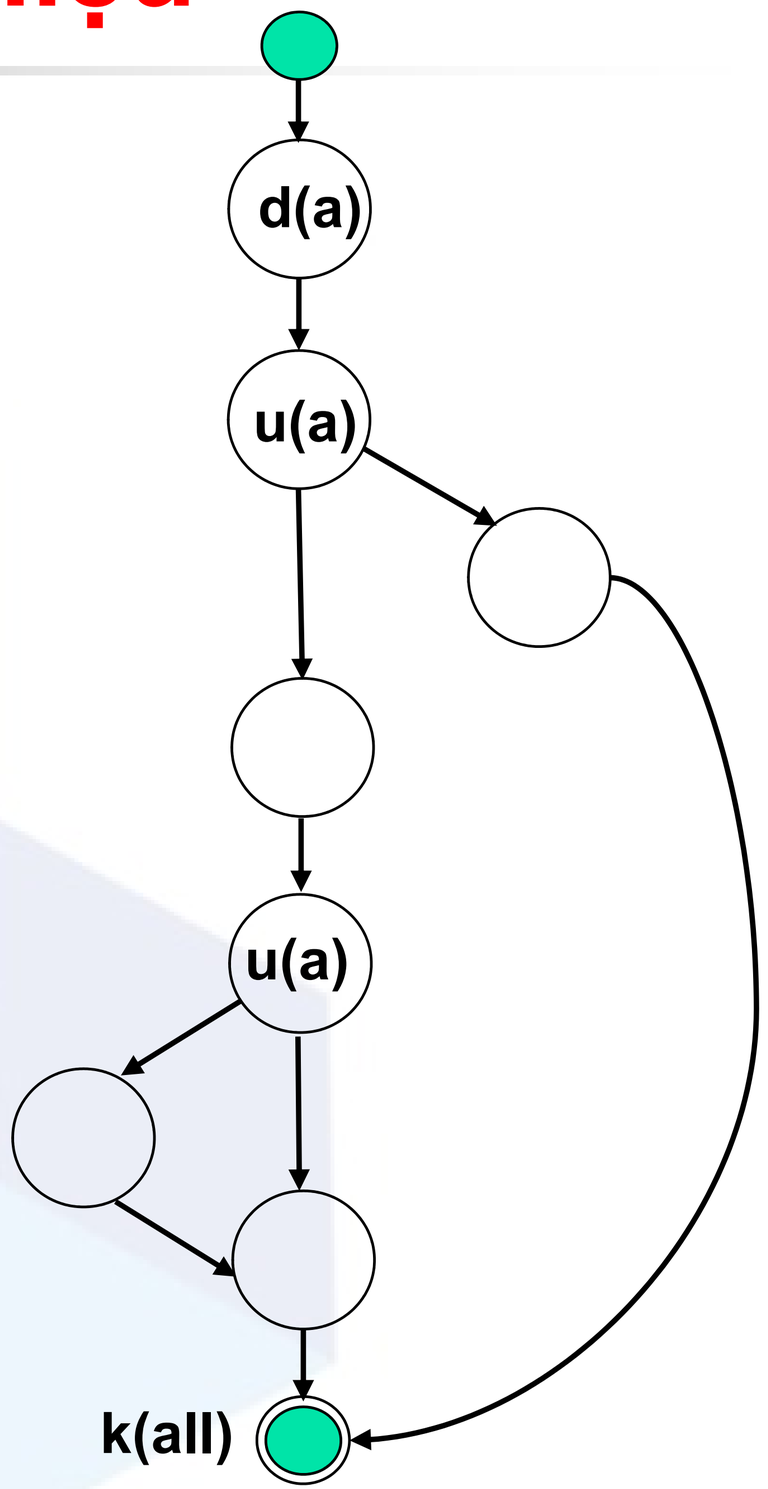
Kiểm thử biến a

P1: ~duuk

P2: ~duuk

P3: ~duk

Kết luận: Không có bất thường



Qui trình kiểm thử dòng dữ liệu

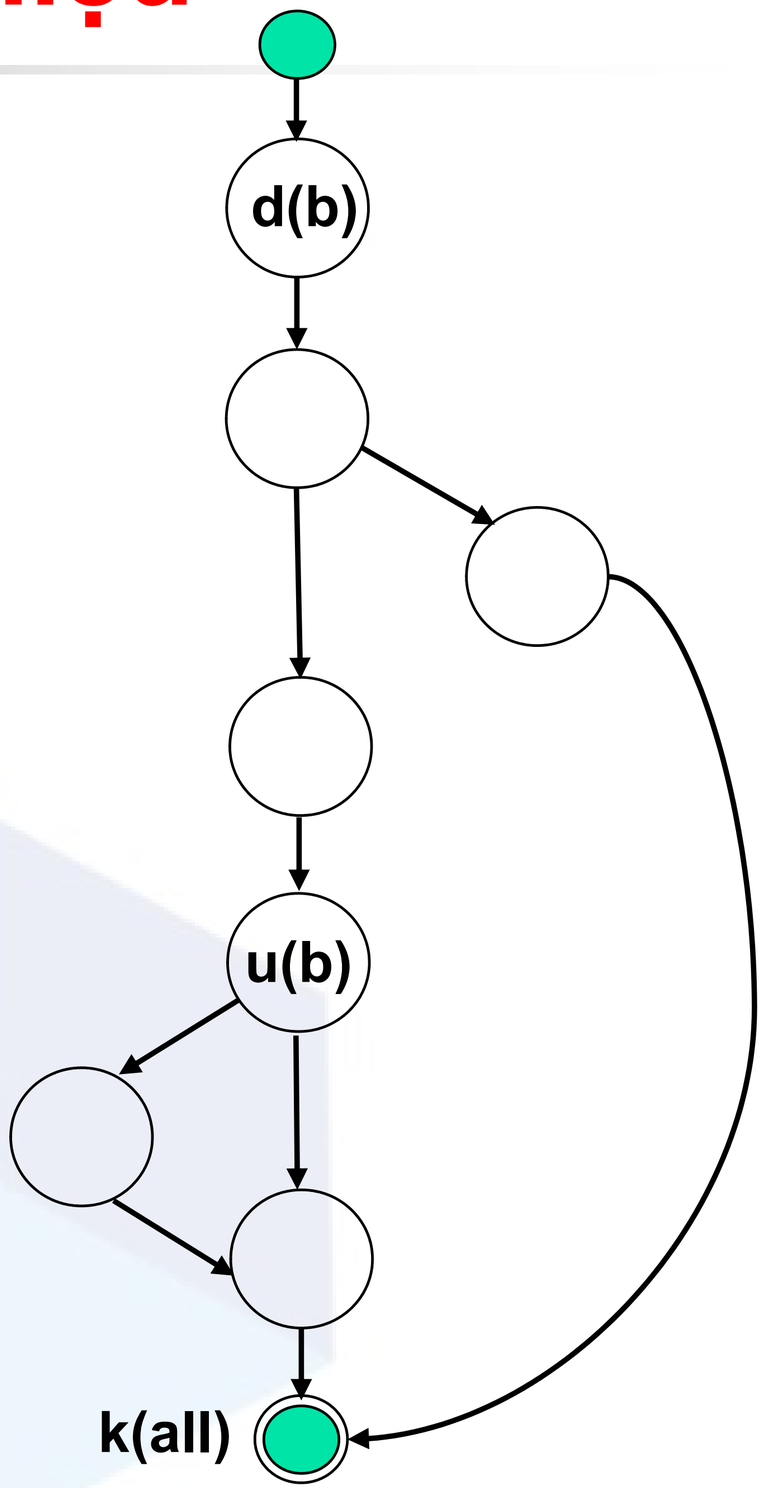
Kiểm thử biến b

P1: ~duk

P2: ~duk

P3: ~dk không có bất thường
vì b là tham số

Kết luận: Không có bất thường



Qui trình kiểm thử dòng dữ liệu

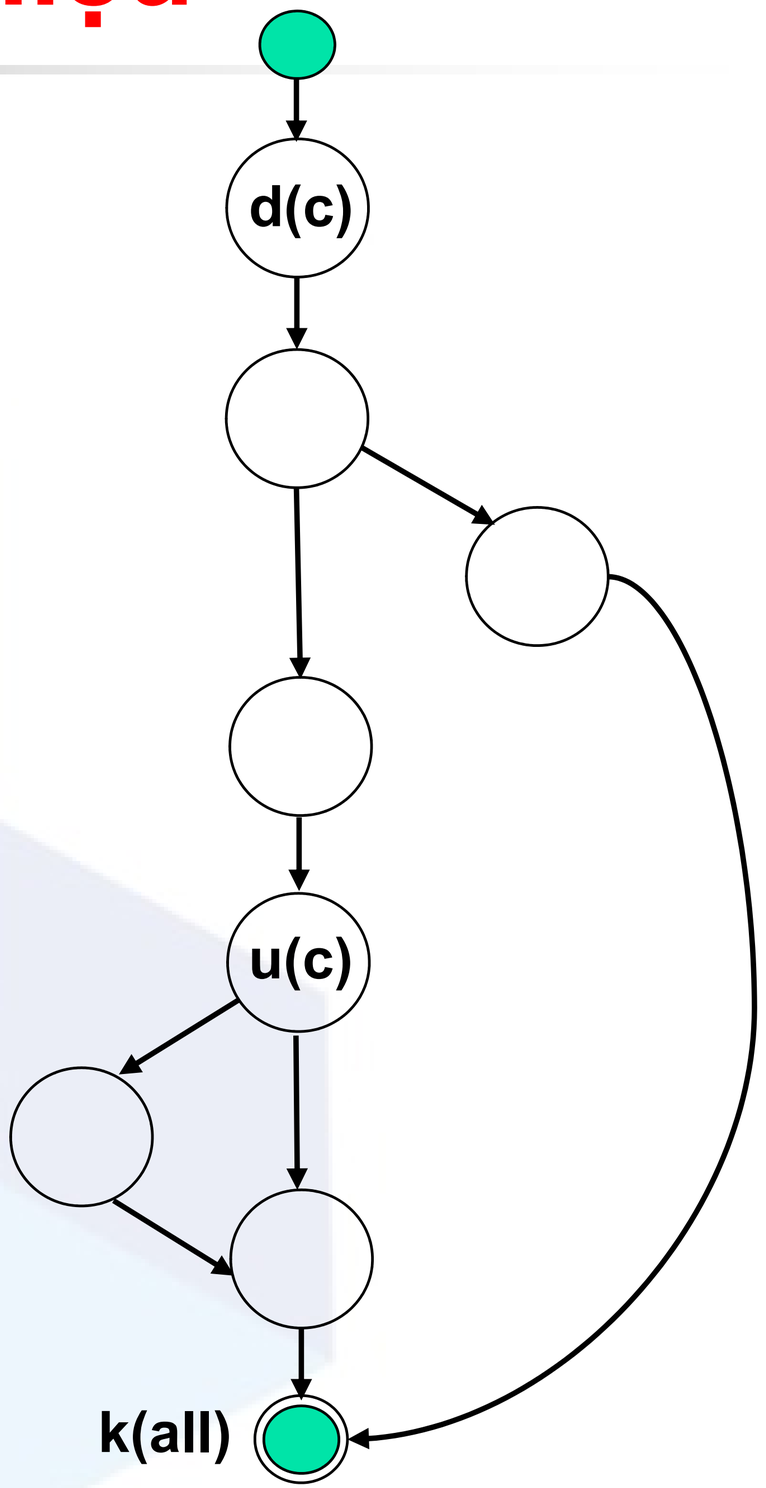
Kiểm thử biến c

P1: ~duk

P2: ~duk

P3: ~dk không có bất thường
vì c là tham số

Kết luận: Không có bất thường



Qui trình kiểm thử dòng dữ liệu

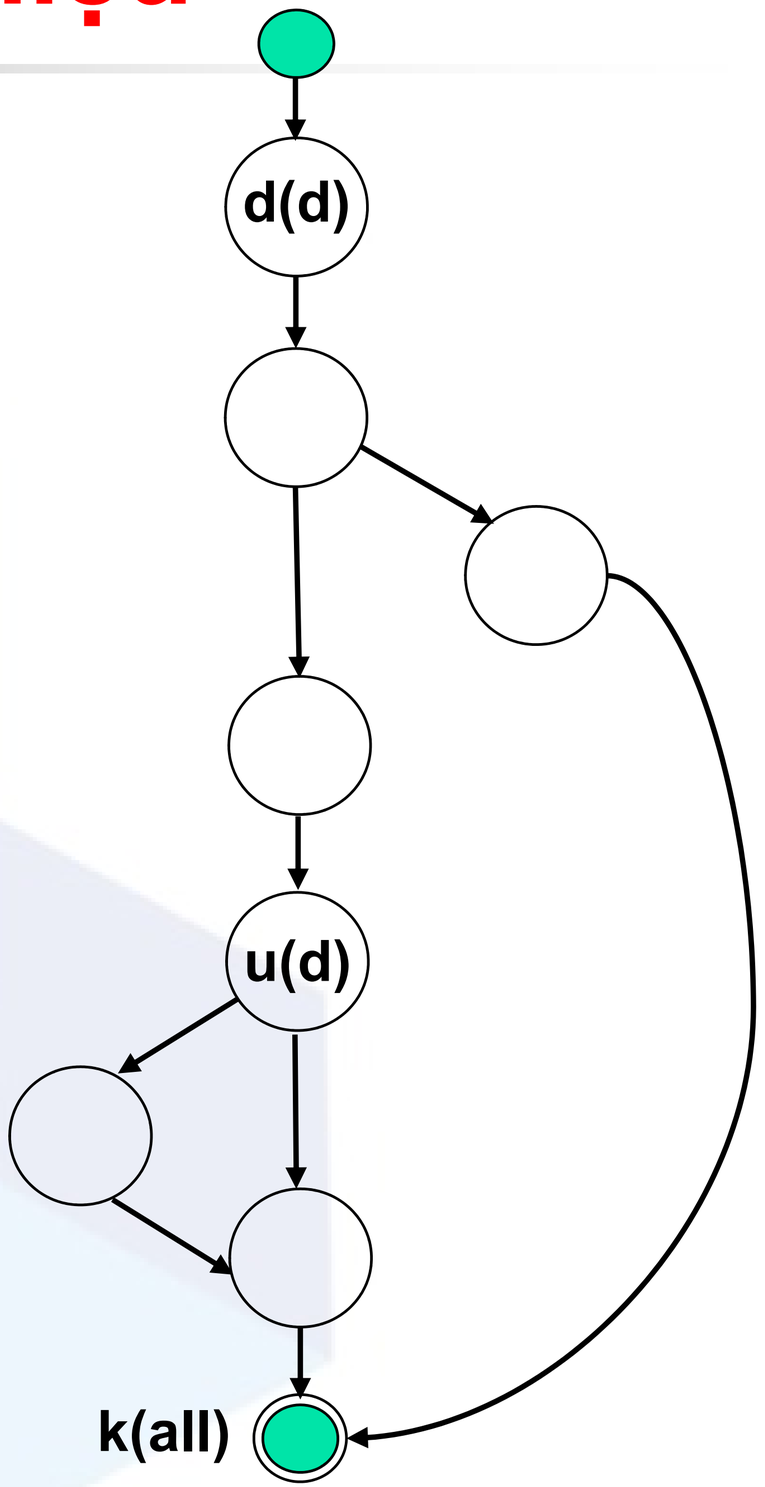
Kiểm thử biến d

P1: ~duk

P2: ~duk

P3: ~dk không có bất thường
vì d là tham số

Kết luận: Không có bất thường



Qui trình kiểm thử dòng dữ liệu

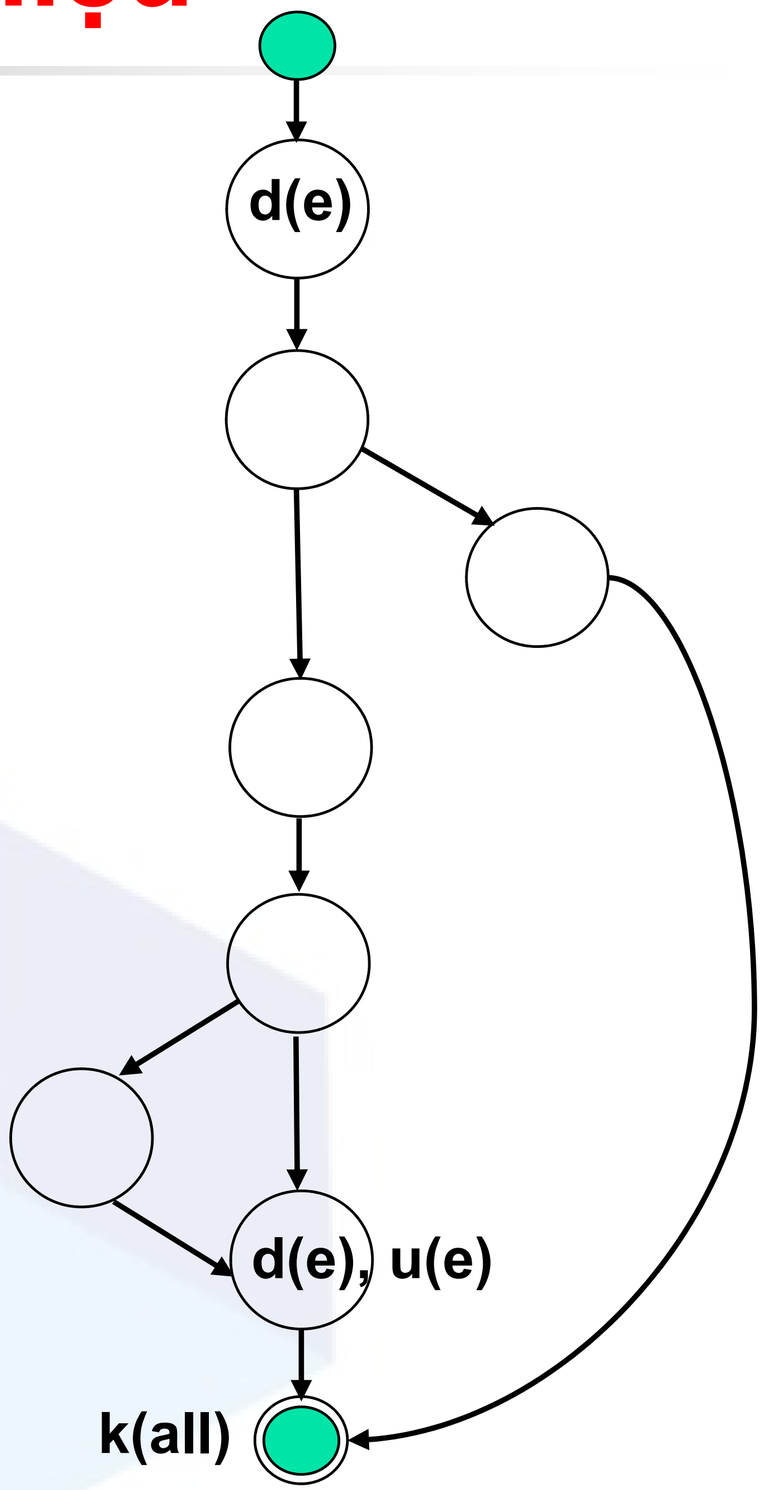
Kiểm thử biến e

P1: ~dduk bất thường

P2: ~dduk bất thường

P3: ~dk bất thường

Kết luận: Có bất thường



Qui trình kiểm thử dòng dữ liệu

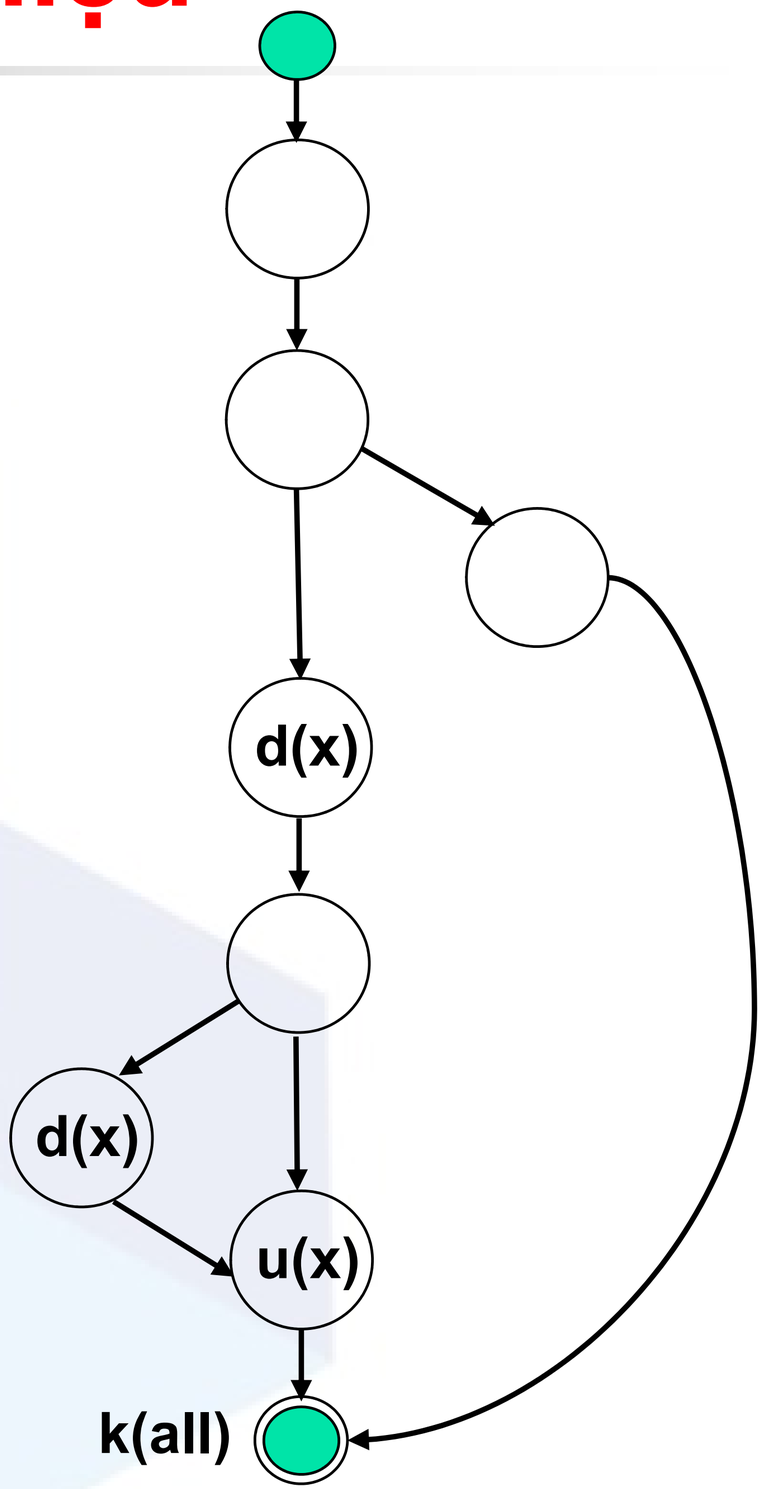
Kiểm thử biến x

P1: ~dduk bất thường

P2: ~duk

P3: ~

Kết luận: Có bất thường



Qui trình kiểm thử dòng dữ liệu

Chương trình mới không có bất thường:

```
float Func(int a, int b, int c, int d)
{
  if (a == 0)
    return 0;
  int x;
  if ((a == b) || ((c == d) && bug(a)))
    x = 1;
  else
    x = 0;
  float e = 1 / x;
  return e;
}
```