Teste Estrutural (parte 2)

Prof. Otávio Lemos (UNIFESP)
Prof. Fabiano Ferrari (UFSCar)

Geração de GFCs

- gere o GFC do seguinte programa:

```
void insercao(int a[], int size) {
  int i, j, aux;
  for (i = 1; i < size; i++) {
    aux = a[i];
    j = i - 1;
    while (j >= 0 && a[j] >= aux) {
      a[j+1] = a[j];
      j--;
    }
    a[j+1] = aux;
  }
}
```

Geração de GFCs

```
public static void heapsort(int n, double ra[]) {
  int l, j, ir, i; double rra;
  l = (n >> 1) + 1;
  ir = n;
  for(;;) {
    if (l > 1)
      rra = ra[--l];
    else {
      rra = ra[ir];
      ra[ir] = ra[1];
      if(--ir == 1) {
        ra[1] = rra;
        return;
      }
    }
    i = 1;
    j = l << 1;
    while(j <= ir) {
      if(j < ir && ra[j] < ra[j+1])
        ++j;
      if(rra < ra[j]) {
        ra[i] = ra[j];
        j += (i = j);
      } else
        j = ir + 1;
    }
    ra[i] = rra;
  }
}
```

Critérios de Fluxo de Controle

- Os critérios baseados em fluxo de controle utilizam apenas características de controle da execução do programa, como comandos ou desvios, para determinar quais estruturas são necessárias.
  - Todos-Nós: exige que a execução do programa passe, ao menos uma vez, em cada vértice do grafo de fluxo de controle; ou seja, que cada comando do programa seja executado pelo menos uma vez.
  - Todas-Arestas (ou Todos-Arcos): requer que cada arco do grafo, isto é, cada desvio de fluxo de controle do programa, seja exercitado pelo menos uma vez.
  - Todos-Caminhos: requer que todos os caminhos possíveis do programa sejam executados.

Níveis de Cobertura

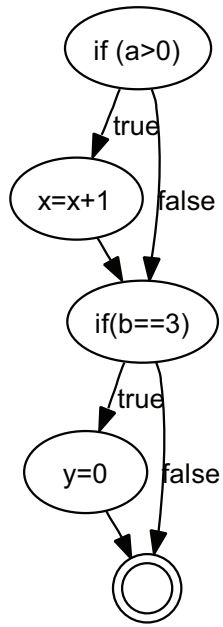
- Diferentes níveis de cobertura podem ser definidos em função dos elementos do GFC
- Cobertura: porcentagem dos **requisitos de teste** que foram testados versus o total de requisitos de teste gerados
- Oito diferentes níveis de cobertura são definidos por Copeland [1].
  - Quanto maior o nível, maior o rigor do critério de teste, ou seja, mais casos de teste ele exige para ser satisfeito
  - Nível 0 $\leftarrow$ Nível 1 $\leftarrow$ Nível 2 $\leftarrow$ Nível 3 $\leftarrow$ Nível 4 $\leftarrow$ Nível 5 $\leftarrow$ Nível 6 $\leftarrow$ Nível 7

[1] Copeland, L.: A Practitioner's Guide to Software Test Design. Artech House, 2004.

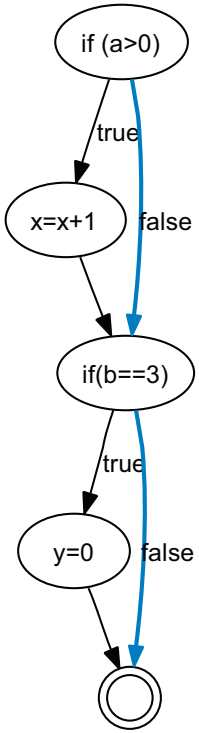
Níveis de Cobertura

- Nível 0: qualquer valor de cobertura inferior a 100% da cobertura de todos os comandos
- Nível 1: 100% de cobertura de comandos – cobertura de nós (critério todos-nós)

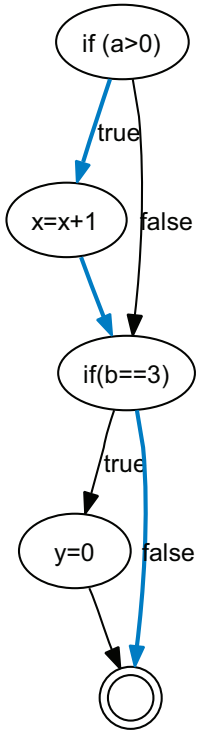
```
1  if (a>0){
2      x=x+1;
3  }
4  if (b==3){
5      y=0;
6  }
```



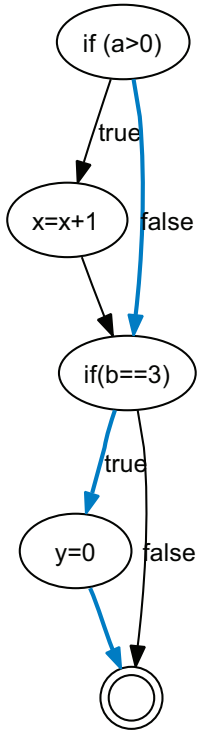
Níveis de Cobertura



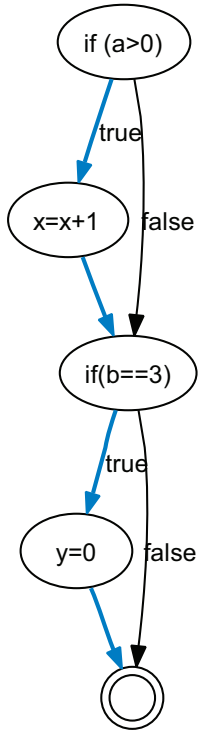
Caminho 1



Caminho 2



Caminho 3



Caminho 4

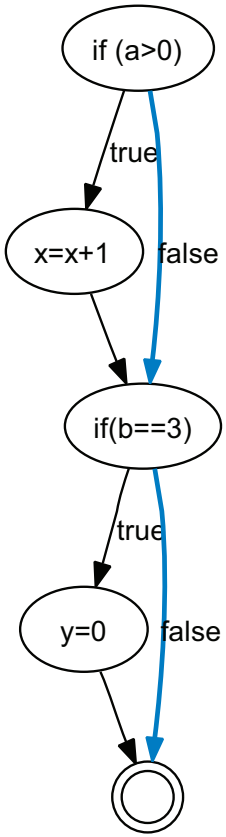
Níveis de Cobertura

- Embora o nível 1 seja o nível mais baixo de cobertura, pode ser difícil de ser atingido em alguns casos
  - Código para situações excepcionais: falta de memória, disco cheio, arquivos ilegíveis ou corrompidos, perda de conexão, dentre outras
    $\Rightarrow$ Pode ser difícil ou impossível simular tais situações excepcionais

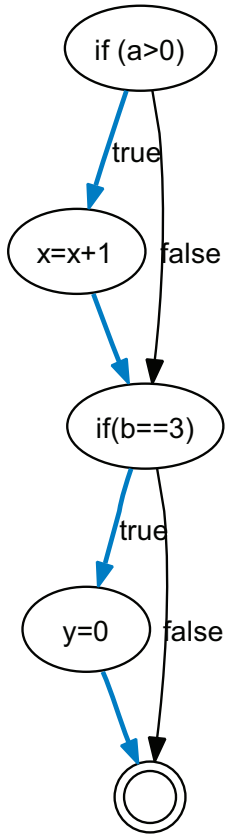
Níveis de Cobertura

- Nível 2: 100% de cobertura de decisões – cobertura de arcos/arestas (critério todas-arestas)
- Objetivo: fazer cada comando de decisão assumir os valores TRUE e FALSE

Níveis de Cobertura



Caminho 1



Caminho 4

Níveis de Cobertura

- Nem todos comandos de decisão são simples como o anterior

```
1  if (a>0 && c==1){
2     x=x+1;
3  }
4  if (b==3 || d<0){
5     y=0;
6  }
```

- linha 2: requer que a>0 e c==1 sejam ambos TRUE
  - Se a for 0, o comando c==1 pode nunca ser executado (linguagem de programação - curto-circuito)
- linha 5: requer que b==3 ou d<0 seja TRUE

Níveis de Cobertura - Exercício

- Definir casos de teste para cobrir todas as possíveis combinações de decisão do trecho de código a seguir:

```
1  if  (a>0 && c==1){
2      x=x+1;
3  }
4  if  (b==3 || d<0){
5      y=0;
6  }
```

Níveis de Cobertura

- Outros níveis da hierarquia de Copeland:
  - Exploram conceitos mais avançados relacionados aos níveis anteriores:
    - Nível 3: cobertura de condições
    - Nível 4: cobertura de decisões/condições
    - Nível 5: cobertura de condições múltiplas (emprega até conceitos de compiladores)
    - Nível 6: cobertura de laços
    - Nível 7: cobertura de caminhos (todos-caminhos)

Critério de McCabe

- Informações sobre a complexidade do programa para derivar requisitos de teste
- Bastante conhecido: critério de McCabe (ou teste de caminho básico) – pioneiro 1976
  - Informação utilizada: *complexidade ciclomática* do grafo de fluxo de controle
  - Métrica de software: medida quantitativa da complexidade lógica do programa
  - No contexto do teste de caminho básico: valor da complexidade ciclomática = número de caminhos linearmente independentes
  - Estabelece limite máximo de casos de teste para garantir execução de todas as instruções

Critério de McCabe

- Caminho linearmente independente: qualquer caminho que introduza pelo menos um novo conjunto de instruções ou uma nova condição
- No CFG: caminho que introduz novo arco
- Determinam conjunto básico de caminhos – CTs que os cobrem, cobrem todas as instruções e condições (opções V e F)
- Importante: conjunto básico não é único

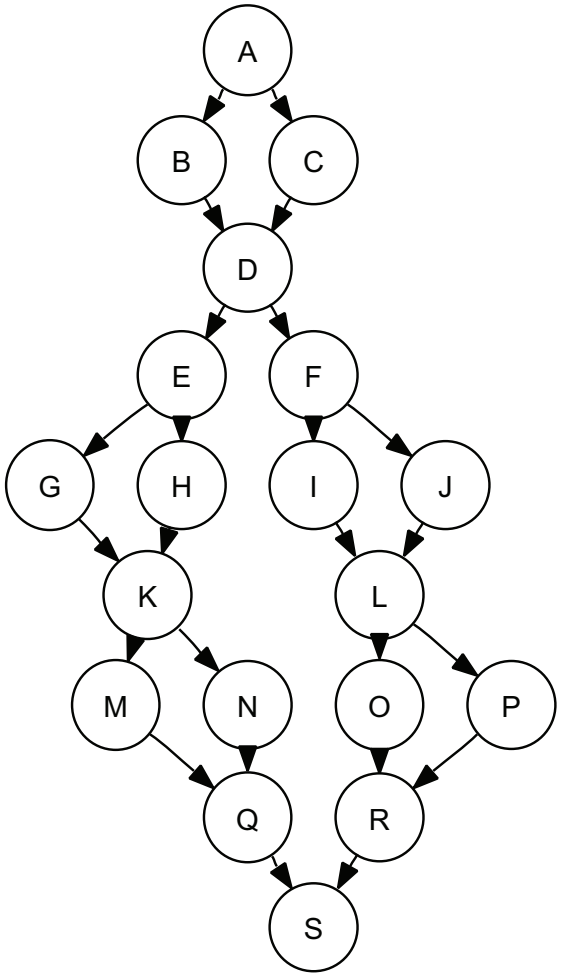
Critério de McCabe

- Complexidade ciclomática (v(G)) pode ser computada de três maneiras:
  1. O número de regiões em um grafo de fluxo de controle (todas as áreas delimitadas + área fora do grafo); ou
  2. $v(G) = E - N + 2$, onde E é o número de arcos e N é o número de nós do grafo de fluxo de controle G; ou
  3. $v(G) = P + 1$, onde P é o número de nós predicativos contido no grafo de fluxo de controle G.

Critério de McCabe

- Passos:
  1. Construir o GFC para o módulo em teste
  2. Calcular a v(G) para o módulo
     - v: número ciclomático na teoria dos grafos
     - G: grafo
     - v(G): indica que complexidade é uma função do grafo
  3. Selecionar um conjunto de caminhos básicos
  4. Criar um caso de teste para cada caminho básico
  5. Executar os casos de testes

Critério de McCabe



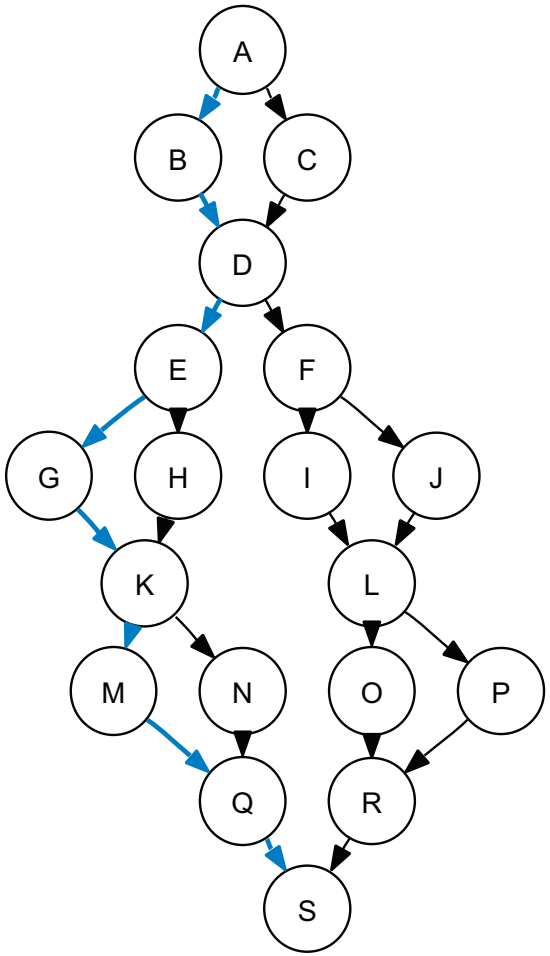
- Considere o GFC ao lado.
- McCabe (1976) define a Complexidade Ciclomática ($\mathcal{C}$) como:

  $\mathcal{C}$ = arcos - nós +2
  $\mathcal{C}$ = 24 - 19 +2
  $\mathcal{C} = 7$

  Para um GFC com p decisões binárias (dois arcos saindo):

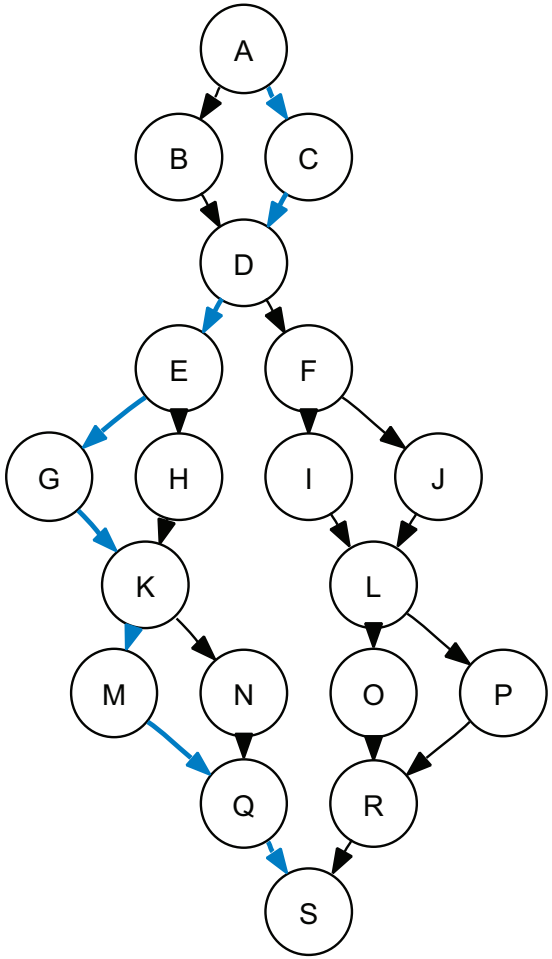
  $\mathcal{C} = p + 1$
- A complexidade ciclomática representa o número mínimo de caminhos independentes, sem loop, que gera todos os possíveis caminhos de um módulo.
- Em termos do GFC, cada caminho básico inclui pelo menos um arco que ainda não foi selecionado.
- Criar e executar $\mathcal{C}$ casos de testes (um para cada caminho básico) garante cobertura dos critérios **todos-nós** e **todos-arcos**.

Critério de McCabe



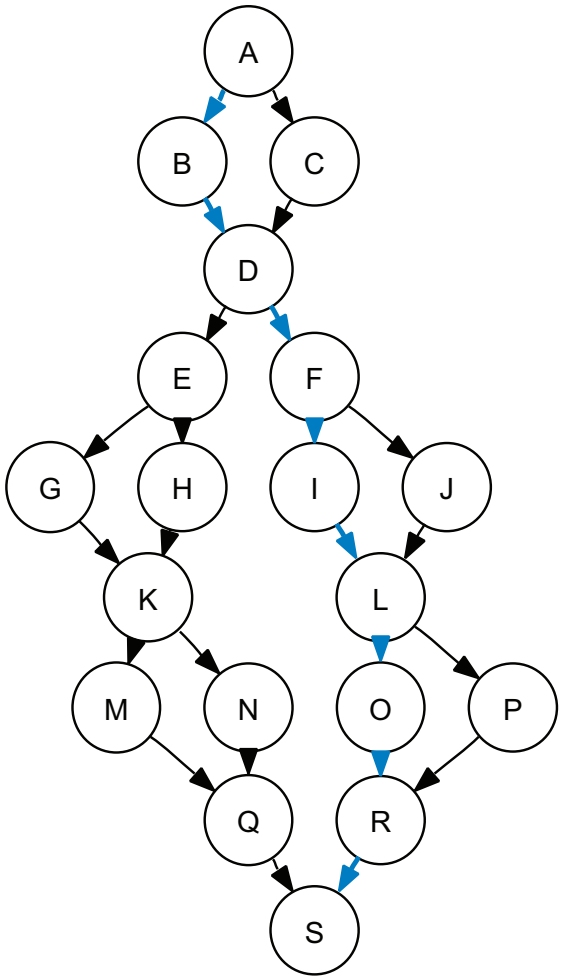
- Escolha um caminho básico. Esse caminho pode ser:
  - Caminho mais comum.
  - Caminho mais crítico.
  - Caminho mais importante do ponto de vista de teste.
- Caminho 1: ABDEGKMQS

Critério de McCabe



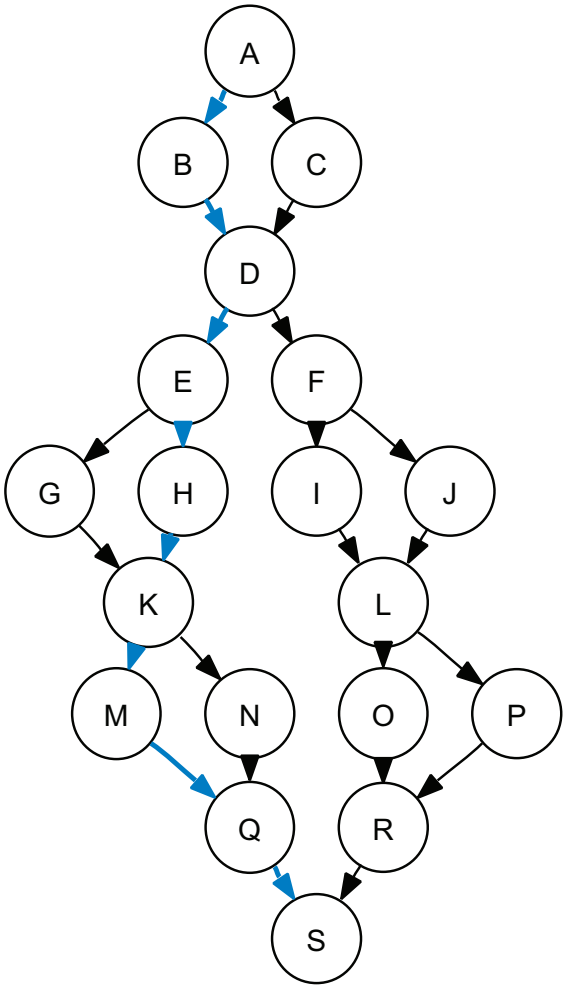
- Altere a saída do primeiro comando de decisão e mantenha o máximo possível do caminho inalterado.
- Caminho 2: ACDEGKMQS

Critério de McCabe



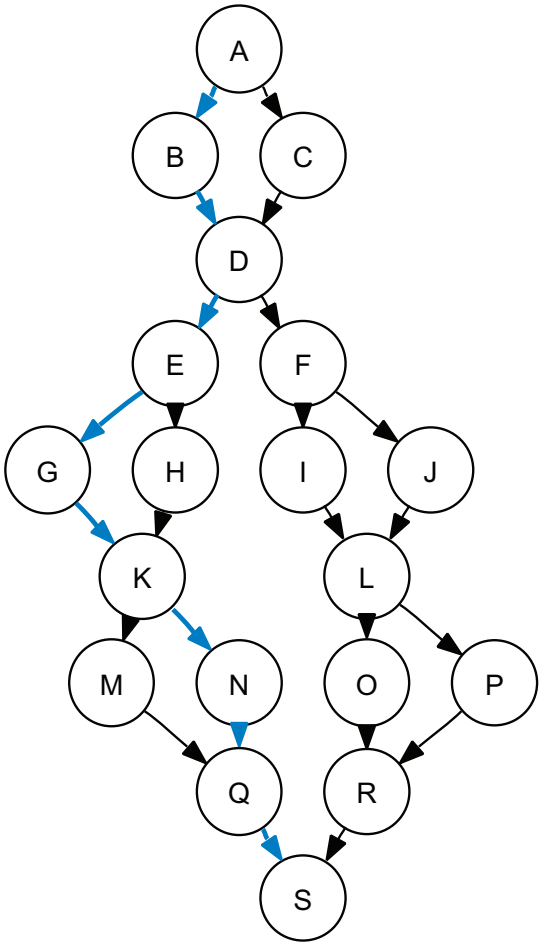
- A partir do caminho base alterar a saída do segundo comando de decisão.
- Caminho 3: ABDFILORS

Critério de McCabe



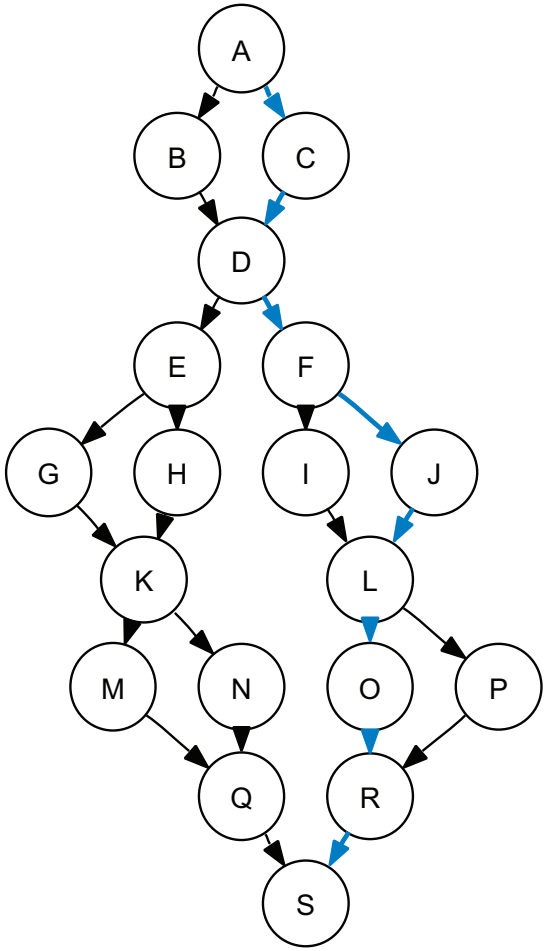
- A partir do caminho base alterar a saída do terceiro comando de decisão. Repetir esse processo até atingir o final do GFC.
- Caminho 4: ABDEHKMQS

Critério de McCabe



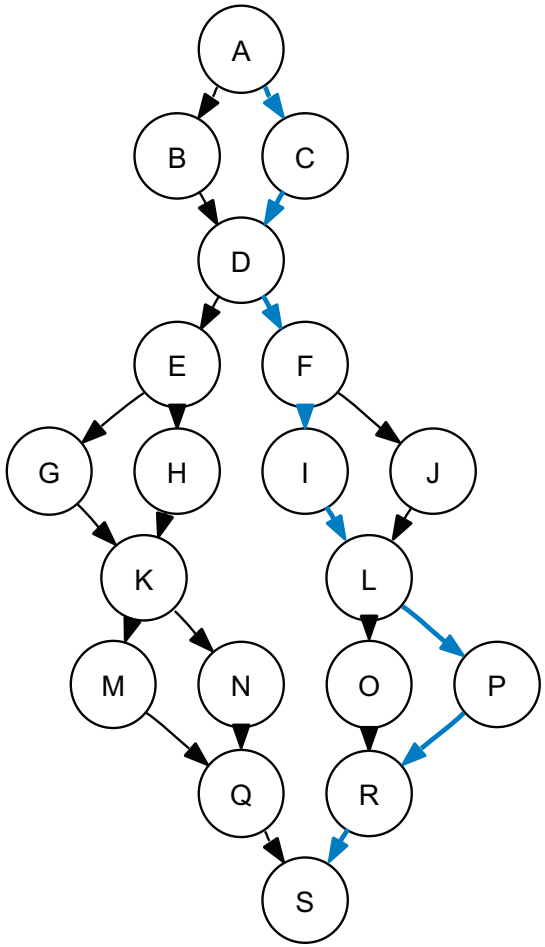
- Continuação do passo anterior.
- Caminho 5: ABDEGKNQS

Critério de McCabe



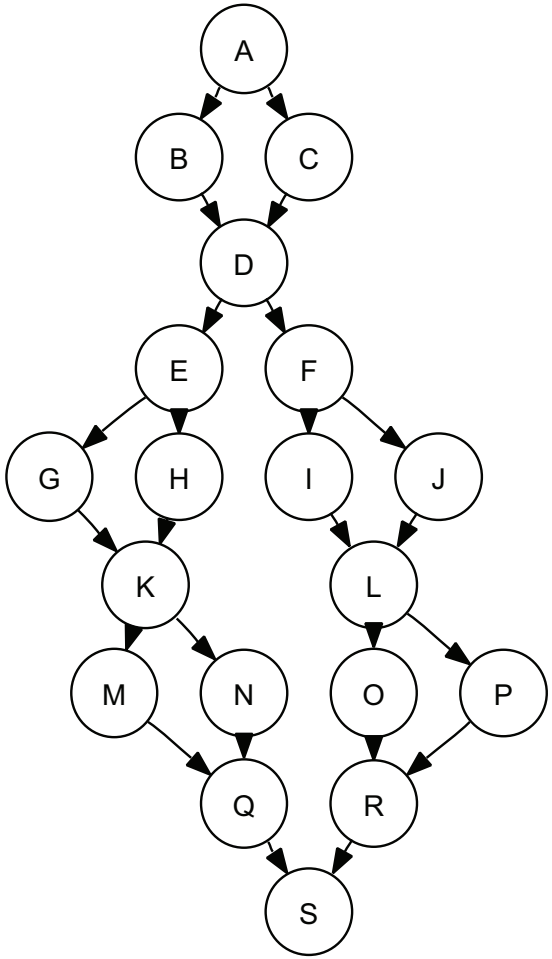
- Todas as decisões do caminho básico foram contempladas.
- A partir do segundo caminho, fazer as inversões dos comandos de decisão até o final do GFC.
- Esse padrão é seguido até que o conjunto completo de caminhos seja atingido.
- Caminho 6: ACDFJLORS

Critério de McCabe



- Continuação do passo anterior.
- Caminho 7: ACDFILPRS

Critério de McCabe

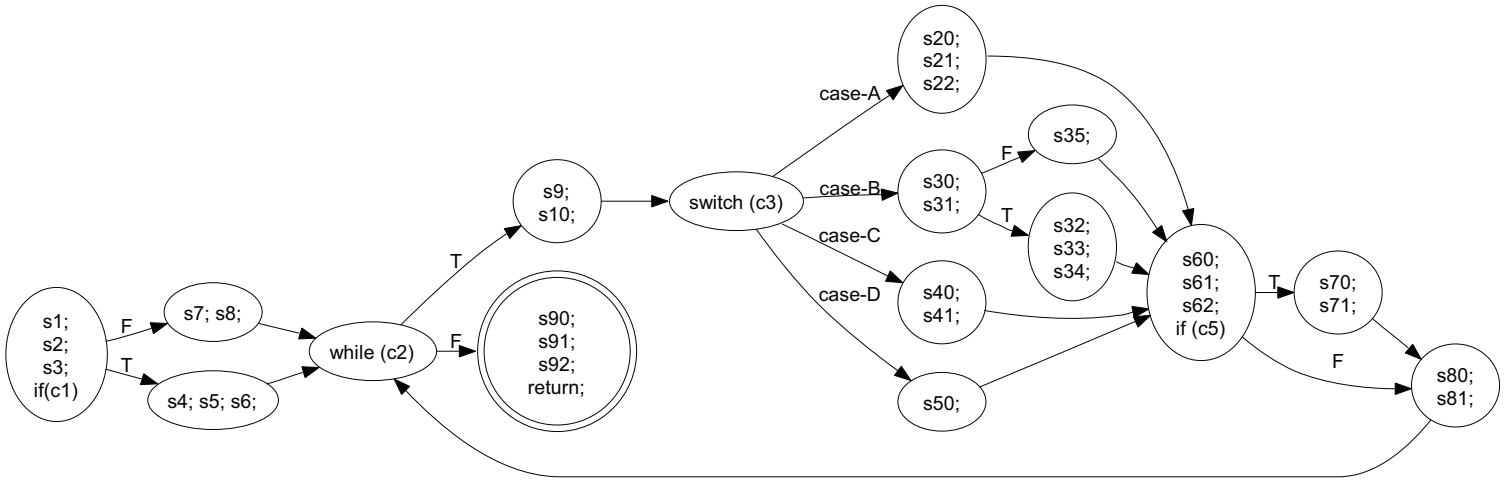


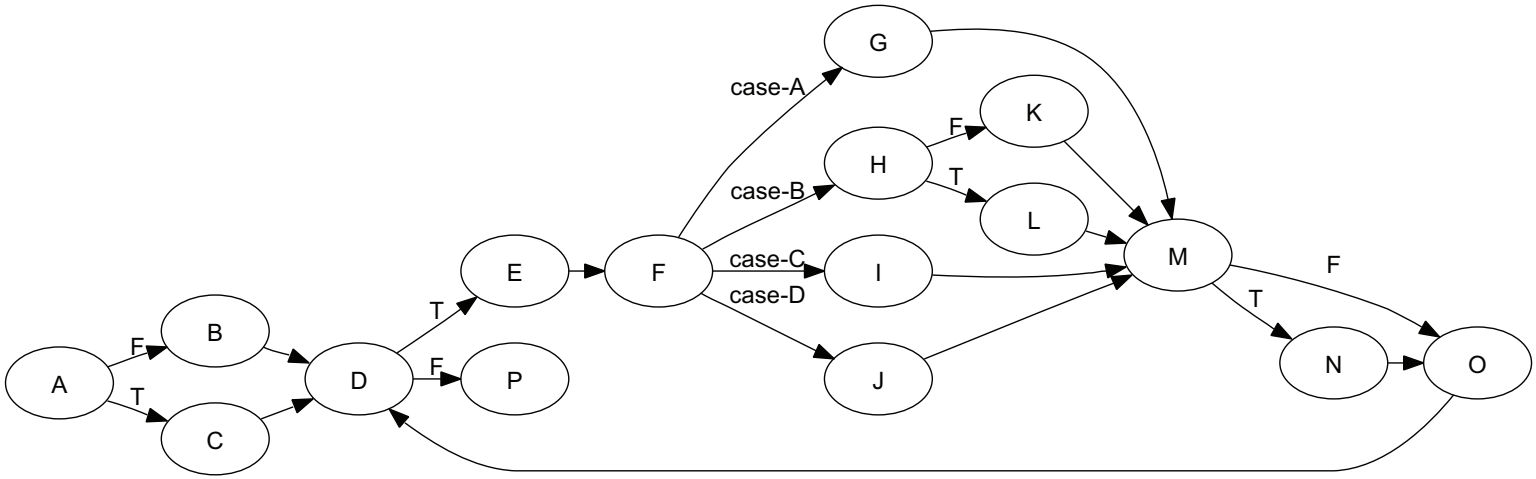
- Requisitos de testes derivado pelo critério.
  - ABDEGKMQS
  - ACDEGKMQS
  - ABDFILORS
  - ABDEHKMQS
  - ABDEGKNQS
  - ACDFJLORS
  - ACDFILPRS
- Conjunto criado não é único.
- Propriedade: o conjunto de teste que exercita os caminhos básicos também exercita todos-nós e todos-arcos do programa.

Exercício – McCabe

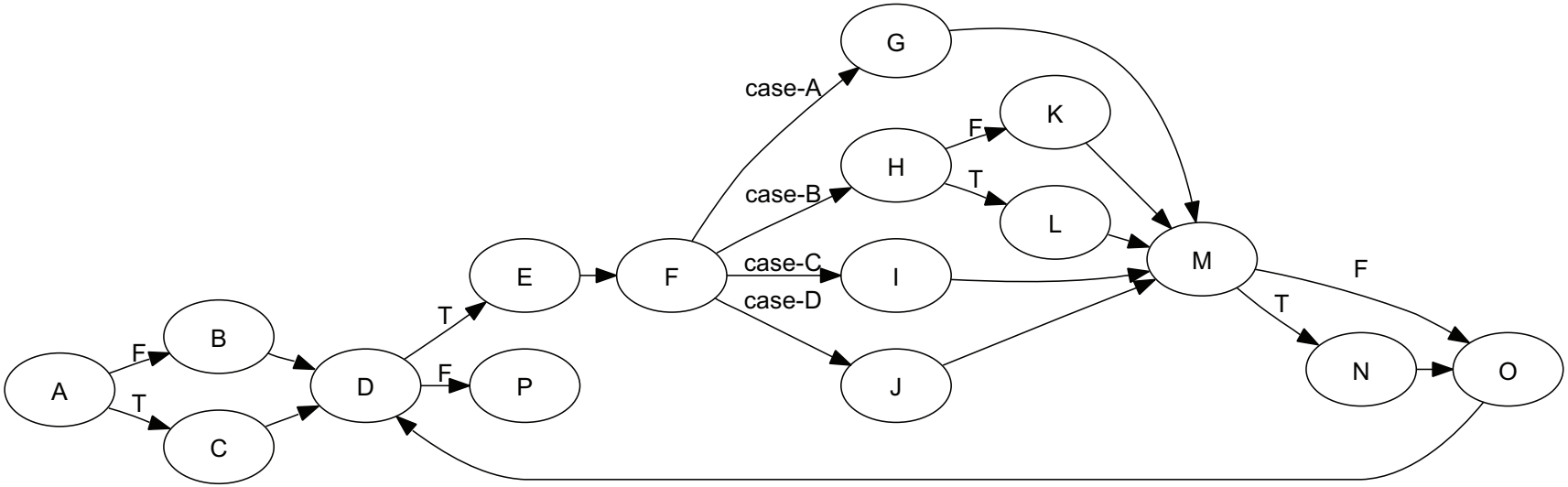
```
1   boolean evaluateBuySell (TickerSymbol ts) {
2   s1;
3   s2;
4   s3;
5   if (c1) {s4; s5; s6;}
6   else {s7; s8;}
7   while (c2) {
8      s9;
9      s10;
10     switch (c3) {
11        case-A:
12           s20;
13           s21;
14           s22;
15           break; // End of Case-A
16        case-B:
17           s30;
18           s31;
19           if (c4) {
20              s32;
21              s33;
22              s34;
23           }
24           else {
25              s35;
26           }
27           break; // End of Case-B
28        case-C:
29           s40;
30           s41;
31           break; // End of Case-C
32        case-D:
33           s50;
34           break; // End of Case-D
35        } // End Switch
36        s60;
37        s61;
38        s62;
39        if (c5) {s70; s71; }
40        s80;
41        s81;
42  } // End While
43  s90;
44  s91;
45  s92;
46  return result;
```

Exercício – McCabe





Exercício – McCabe



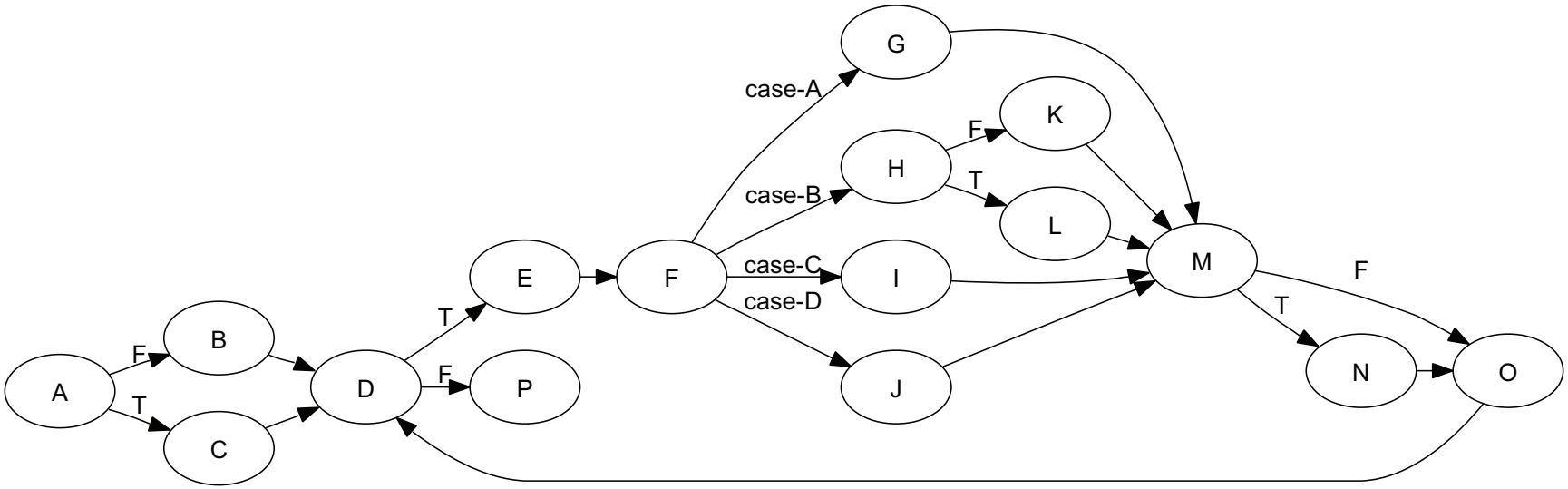
- Cálculo da complexidade ciclomática para o GFC acima:

  $\mathcal{C}$ = arcos - nós + 2

  $\mathcal{C}$ = 22-16+2

  $\mathcal{C}$ = 8

Exercício – McCabe



1. ABDP
2. ACDP
3. ABDEFGMODP
4. ABDEFHKMODP
5. ABDEFIMODP
6. ABDEFJMODP
7. ABDEFHLMODP
8. ABDEFIMNODP

Exercício – McCabe

1. ABDP
2. ACDP
3. ABDEFGMODP
4. ABDEFHKMODP
5. ABDEFIMODP
6. ABDEFJMODP
7. ABDEFHLMODP
8. ABDEFIMNODP

Caso Teste	C1	C2	C3	C4	C5
1	False	False	N/A	N/A	N/A
2	True	False	N/A	N/A	N/A
3	False	True	A	N/A	False
4	False	True	B	False	False
5	False	True	C	N/A	False
6	False	True	D	N/A	False
7	False	True	B	True	False
8	False	True	C	N/A	True

Critérios de Fluxo de Dados (1)

- Critérios pertencentes à Técnica de Teste Caixa Branca.
- Complementares aos critérios baseados em fluxo de controle.
- Busca testar o uso das variáveis em um programa, ou seja, como os dados são usados nas computações.

Critérios de Fluxo de Dados (2)

- Exemplo de defeito em fluxo de dados:

```
1 main(){
2     int x;
3     if (x==42){...}
4 }
```

- Engano: referenciar uma variável sem esta ter sido inicializada.
  - Assumir que o compilador inicializa a variável com algum valor padrão quando ele não o faz. Qual a saída do programa abaixo?

```
1 #include <stdio.h>
2 main() {
3     int x;
4     printf ("%d",x);
5 }
```

Critérios de Fluxo de Dados (3)

- Teste de fluxo de dados é uma ferramenta poderosa para revelar o uso incorreto de valores resultante de defeitos de codificação.
- Tornou-se popular com a publicação do trabalho de Rapps e Weyuker [1]:

  ""It is our belief that, just as one would not feel confident about a program without executing every statement in it as part of some test, one should not feel confident about a program without having seen the effect of using the value produced by each and every computation."

[1] Rapps, S; Weyuker, E. J.: Data Flow Analysis Techniques for Program Test Data Selection. In: ICSE'82. IEEE Computer Society, pp. 272-278.

Critérios de Fluxo de Dados (4)

Princípio da Definição dos Critérios

Programa = sequência de ações realizadas sobre variáveis

Fluxo de controle

Informações sobre onde variáveis são definidas e onde essas definições são usadas.

Critérios de Fluxo de Dados (5)

- Qual o modelo de defeitos?
  - Anomalias de fluxo de dados:
    - uso de variável não inicializada.
    - atribuição de valor a uma variável mais de uma vez sem que tenha havido uma referência a essa variável entre essas atribuições.
    - liberação ou reinicialização de uma variável antes que ela tenha sido criada ou inicializada.
    - liberação ou reinicialização de uma variável antes que ela tenha sido usada.
    - atribuir novo valor a um ponteiro sem que variável tenha sido liberada.

Ciclo de Vida de Variáveis (1)

- Variáveis são criadas, usadas e destruídas.
- Em algumas linguagens de programação (BASIC e FORTRAN, por exemplo) a criação e destruição são automáticas.
- Em outras (C, C++ e Java, por exemplo), a criação deve ser explícita.
- Exemplos de declaração:

```
1 int x;        // x é criada como um inteiro
2 String y; // y é criada como uma String
```

Ciclo de Vida de Variáveis (2)

- Declarações, em geral, ocorrem dentro de um bloco.
  - Variáveis são criadas quando a definição das mesmas são executadas.
  - Variáveis são destruídas no final do bloco (conceito de escopo da variável).

```
1 {              // início do bloco 1
2     int x;     // x é definida no bloco 1
3     ...;       // x pode ser usada aqui
4     {          // início do blodo 2
5         int y;// y é definida no bloco 2
6         ...;  // x e y podem ser usadas aqui
7     }          // y é automaticamente destruída no final do bloco 2
8     ...;       // x ainda pode ser usada daqui em diante
9 }              // x é automaticamente destruída no final do bloco 1
```

Tipos de Uso de Variáveis

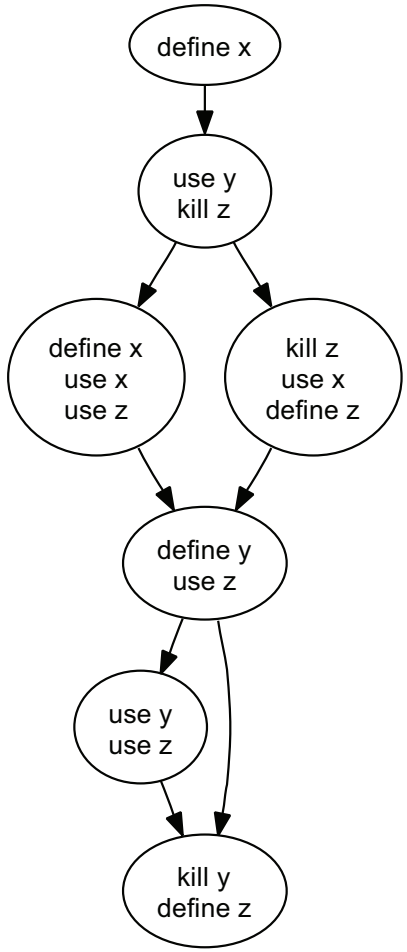
- Existem dois tipos de **uso** de variáveis:
  - Uso em computações, denominados **uso computacional**. Por exemplo: $\mathrm{a} = \mathrm{b} * 1$.
  - Uso em condições, denominado **uso predicativo**. Por exemplo: $\mathrm{if}\ (\mathrm{a} >= \mathrm{b})$.
- Independentemente do tipo de uso, é imprescindível que antes de ser usada a variável tenha sido **definida**.
  - A **definição** de uma variável ocorre quando ela recebe um valor. Por exemplo, via comando de atribuição: $\mathrm{a} = 10$ e $\mathrm{b} = 5$.

Grafo Definição-Uso (1)

- Para avaliar os diferentes estados das variável no programa é utilizado um grafo denominado **Grafo Definição-Uso** ou Grado Def-Uso.
- Semelhante ao Grafo de Fluxo de Controle.
- Inclui ainda variáveis definidas, usadas e destruídas em cada nó.
- Análise dinâmica do Grafo Def-Uso: executar o programa com casos de testes e avaliar o resultado.

Grafo Definição-Uso (2)

- Exemplo de GFC com anotações de fluxo de dados.



Grafo Definição-Uso: Análise Dinâmica

- Assumir que o fluxo de controle do módulo está correto.
- Criar casos de testes de modo que:
  - Cada definição de variável é rastreada até cada um de seus usos.
  - Cada uso é rastreado a partir de sua definição correspondente.
- Para fazer isso:
  - Enumerar os caminhos no GDU usando a mesma abordagem do teste de fluxo de controle.
  - Criar casos de testes que cubram cada par "definição-uso" (associações de fluxo de dados) entre as variáveis.

Definições (1)

- A ocorrência de variáveis em um programa pode ser classificada em:
  - **Definição** (*def* ou *d*): ocorre quando uma variável recebe um valor.

$$\mathrm{a} = 1$$

  - **Uso**: ocorre quando a variável é referenciada e tem o seu valor consultado. Um uso pode ser:
    - **Computacional** (*c-uso* ou *uc*): a variável é utilizada em uma computação.

$$\mathrm{b} = \mathrm{a} * 2$$

    - **Predicativo** (*p-uso* ou *up*): a variável é utilizada em uma condição.

$$\mathrm{if}\ (\mathrm{a} > 0)$$

Definições (2)

- **c-uso global** quando não existe *def* da variável no bloco em que ocorre o *c-uso*.
- **caminho livre de definição** em relação a uma variável x (c.d.l.(x)): caminho entre nós A e B, sendo que x é definida em A, possui um uso em B e não existe nenhuma outra definição de x entre A e B.
- **def global**: quando a *def* de uma variável x em um bloco A é usada em um bloco B (ou em um predicado) sem que haja redefinição de x entre os blocos A e B ou existe c.l.d.(x) entre A e B

Definições (3)

- $c\text{-}use(i) = \{$variáveis com c-uso global no bloco $i\}$
- $def(i) = \{$variáveis com definições globais no bloco $i\}$
- $p\text{-}use(i,j) = \{$variáveis com p-usos no arco $(i,j)\}$
- $dcu(x,i) = \{$nós j tal que $x \epsilon\ c\text{-}use(j)$ e existe um caminho livre de definição c.r.a x do nó i para o nó $j\}$
- $dpu(x,i) = \{$arcos (j,k) tal que $x \epsilon\ p\text{-}use(j,k)$ e existe um caminho livre de definição c.r.a x do nó i para o arco $(j,k)\}$

Definições (4)

- caminho $(n_1, n_2, ..., n_j, n_k)$ é um "du-caminho" c.r.a variável x se:
  - n_1 tiver uma definição global de x, e
    1. n_k tem um c-uso de x e $(n_1, n_2, ..., n_j, n_k)$ é um caminho simples livre de definição c.r.a x; ou
    2. (n_j, n_k) tem um p-uso de x e $(n_1, n_2, ..., n_j, n_k)$ é um caminho livre de definição c.r.a x e $n_1, n_2, ..., n_j$ é um caminho livre de laço;

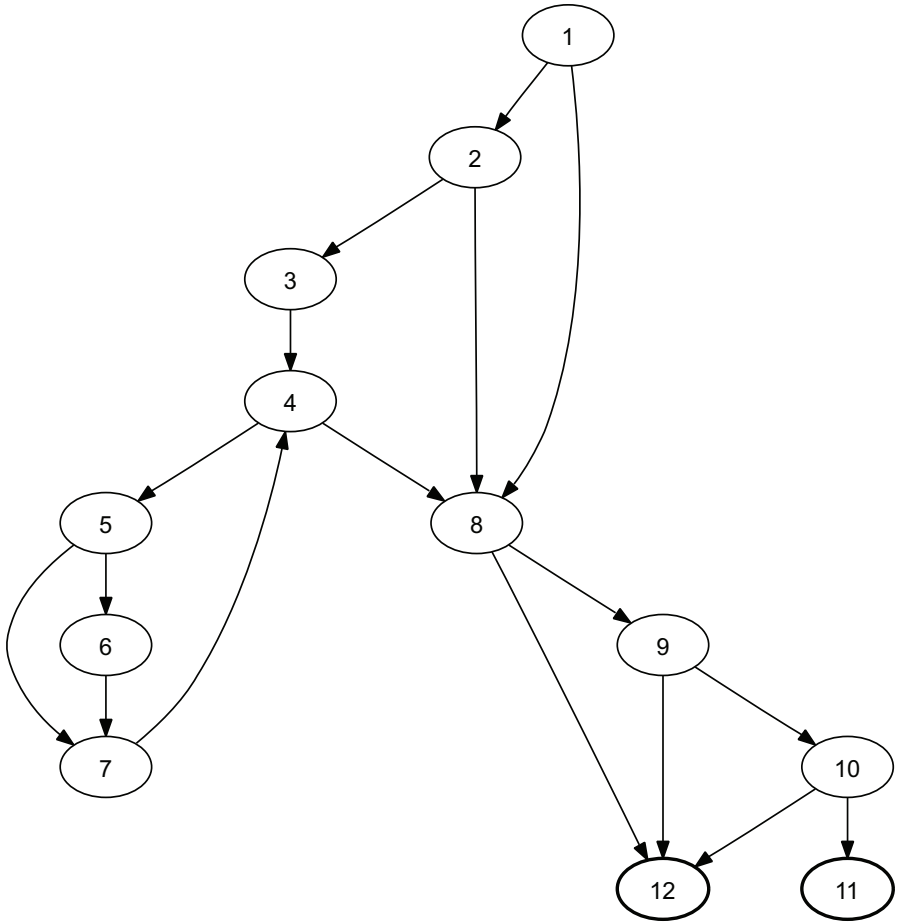
Definições (5)

- "associação definição-c-uso" é uma tripla (i, j, x) onde i é um nó que contém uma definição global de x e $j \epsilon dcu(x, i)$
- "associação definição-p-uso" é uma tripla $(i, (j, k), x)$ onde i é um nó que contém uma definição global de x e $(j, k) \epsilon dpu(x, i)$
- "associação" = associação definição-c-uso, uma associação definição-p-uso ou um du-caminho

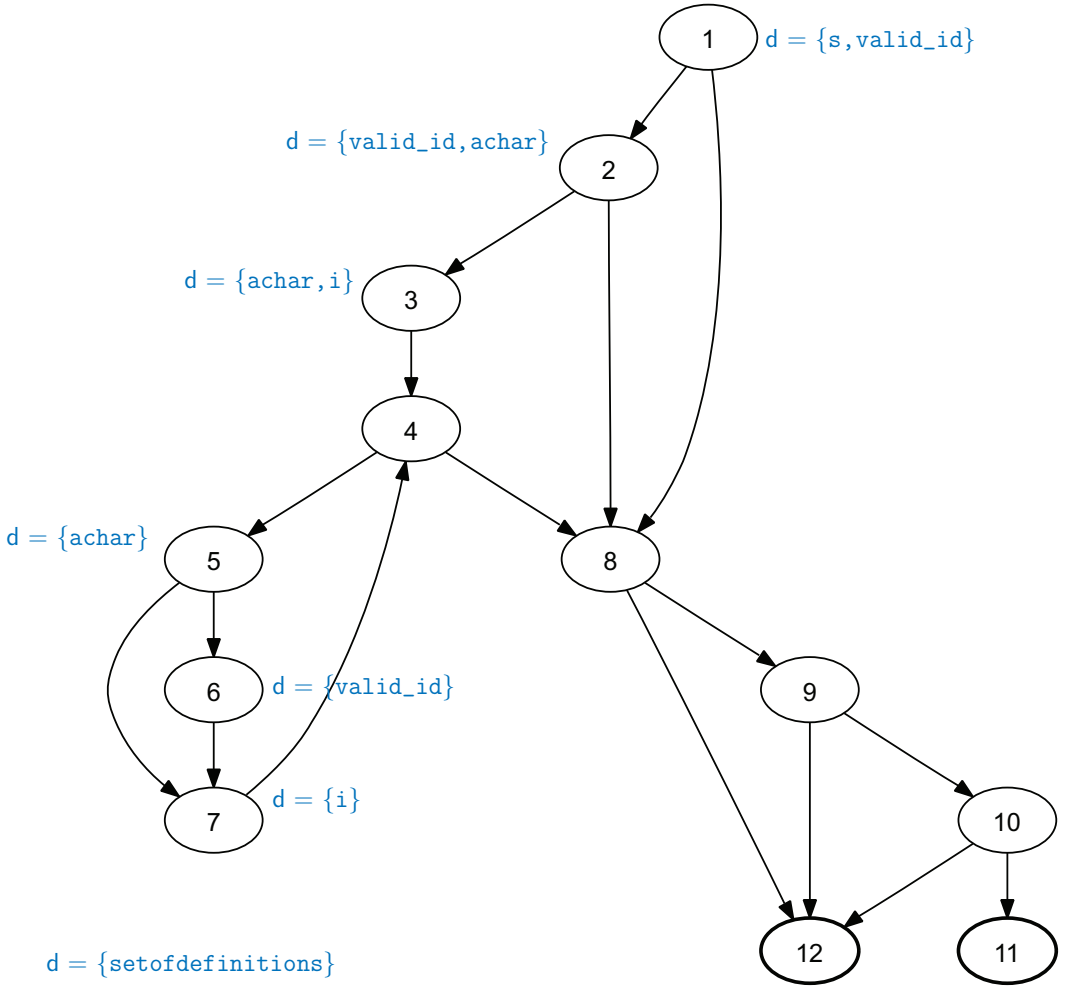
Identifier

```
4   public boolean validateIdentifier(String s) {
5           char achar;
6   /*01*/  boolean valid_id = false;
7   /*01*/  if (s.length() > 0) {
8   /*02*/      achar = s.charAt(0);
9   /*02*/      valid_id = valid_s(achar);
10  /*02*/      if (s.length() > 1) {
11  /*03*/          achar = s.charAt(1);
12  /*03*/          int i = 1;
13  /*04*/          while (i < s.length() - 1) {
14  /*05*/              achar = s.charAt(i);
15  /*05*/              if (!valid_f(achar))
16  /*06*/                  valid_id = false;
17  /*07*/              i++;
18                    }
19              }
20          }
21                  /*08*/              /*09*/                  /*10*/
22          if (valid_id && (s.length() >= 1) && (s.length() < 6))
23  /*11*/      return true;
24          else
25  /*12*/      return false;
26  }
```

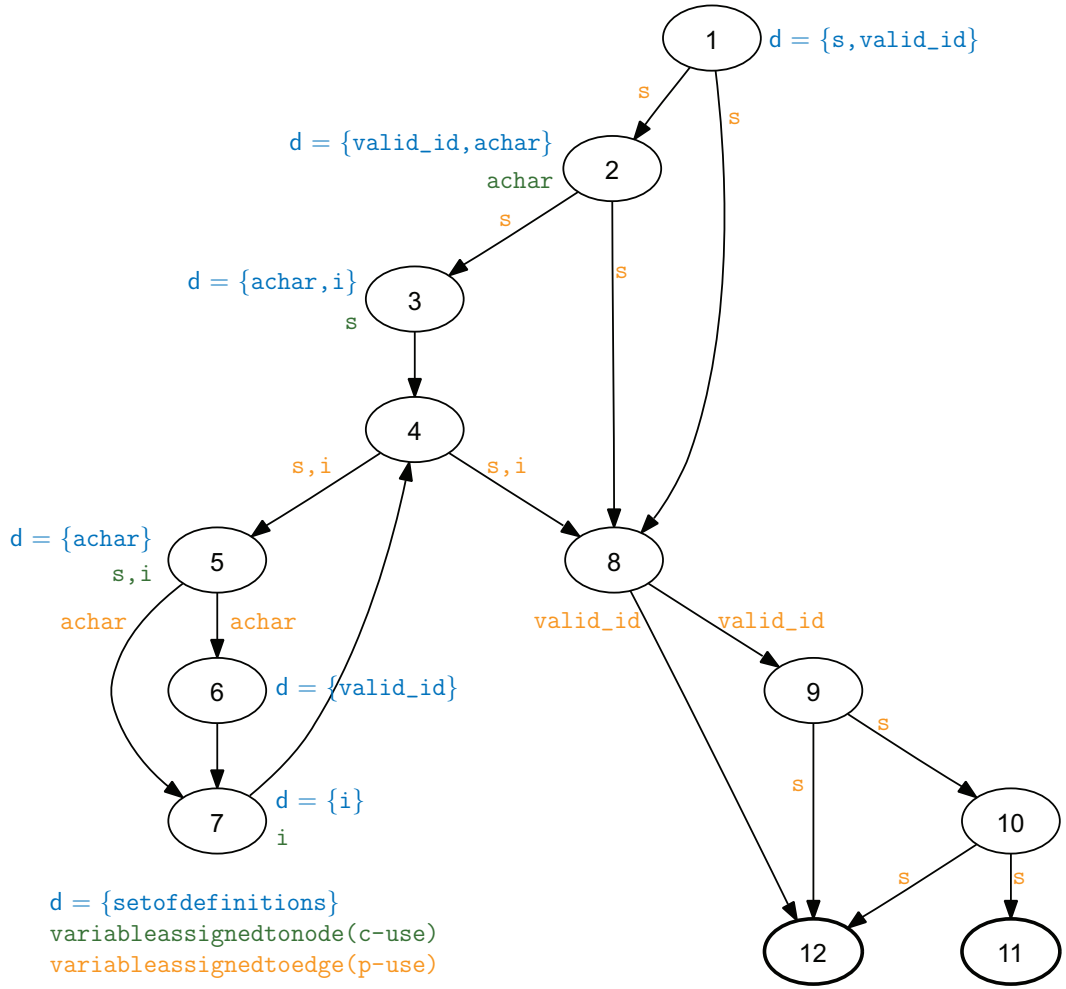
Identifier: GFC



Identifier: GFC + D



Identifier: GFC + D + U



Critérios de Rapps e Weyuker (1)

- Basseiam-se no Grafo Def-Uso para derivar os requisitos de testes.
- Objetivos: exercitar caminhos ligando definições globais a usos globais de variáveis do programa.
- Tipos:
  - todas as definições.
  - todos os p-usos.
  - todos os p-usos e alguns c-usos.
  - todos os c-usos e alguns p-usos.
  - todos os usos.

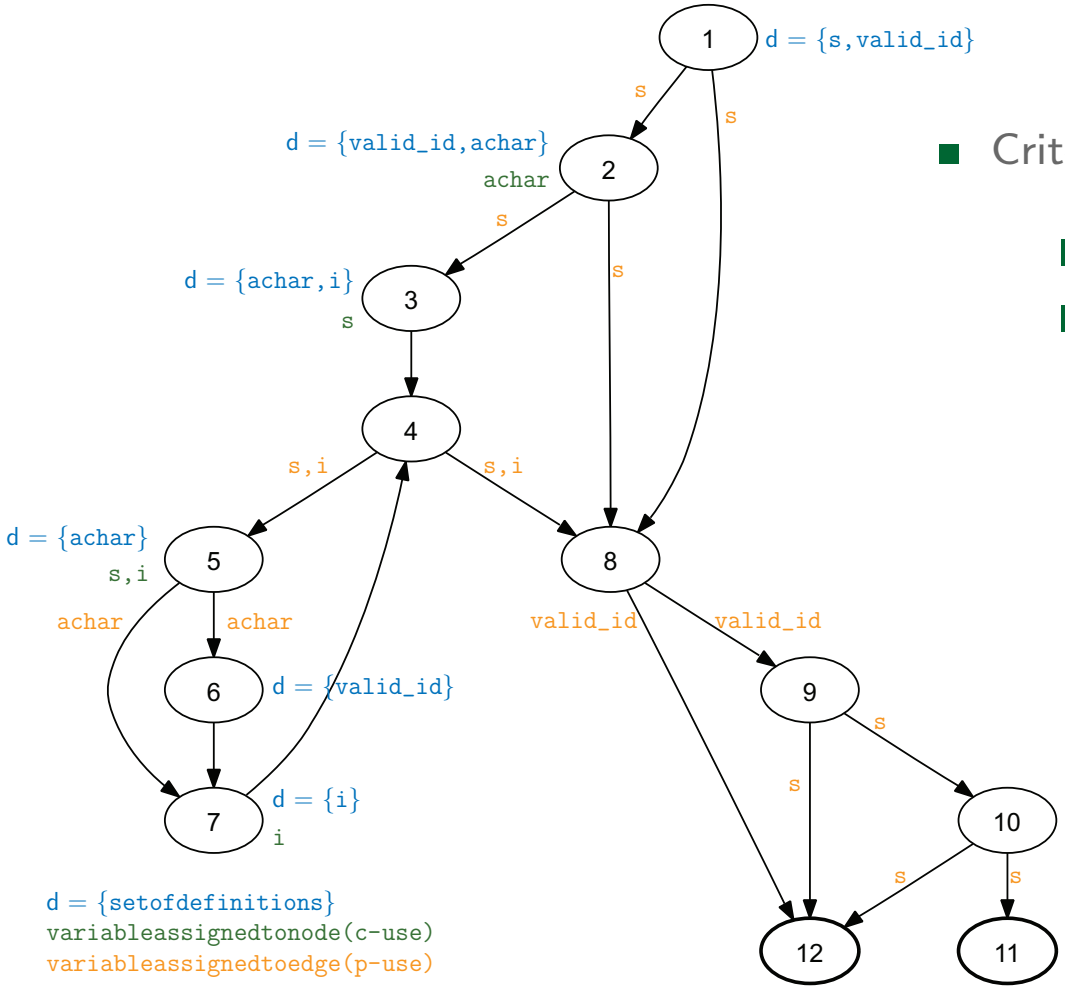
Critérios de de Rapps e Weyuker (2)

- Todas-Definições: requer que cada definição de variável seja exercitada pelo menos uma vez, não importa se por um c-uso ou por um p-uso.
- Todos-Usos: requer que todas as associações entre uma definição de variável e seus subseqüentes usos (c-usos e p-usos) sejam exercitadas pelos casos de teste, através de pelo menos um caminho livre de definição, ou seja, um caminho onde a variável não é redefinida.
  - Variações: Todos-P-Usos, Todos-P-Usos/Alguns-C-Usos e Todos-C-Usos/Alguns-P-Usos
- Todos-DU-Caminhos: requer que toda associação entre uma definição de variável e subseqüentes p-usos ou c-usos dessa variável seja exercitada por todos os caminhos livres de definição e livres de laço que cubram essa associação.

Requisitos: Teste de Fluxo de Dados

- Requisitos gerados pelos critérios de fluxo de dados:
- **Associações Definição-Uso**
  - Em geral uma tripla: $\langle var, def, uso \rangle$, sendo:
    - *var* - variável para a qual a associação definição-uso foi estabelecida.
    - *def* - nó contendo a definição da variável *var*.
    - *uso* - nó/arco com uso computacional/predicativo de *var*.

Requisitos: Teste de Fluxo de Dados



- Critérios de Fluxo de Dados:
  - Todas-Defs
  - Todos-Usos
    - $\langle i, 3, 5 \rangle$
    - $\langle i, 3, 7 \rangle$
    - $\langle valid\_id, 1, (8, 9) \rangle$
    - $\langle valid\_id, 1, (8, 12) \rangle$
    - $\vdots$

Aplicabilidade e Limitações

- Complementares aos critérios de fluxo de controle.
- Podem ser aplicados em todas as fases de testes, sendo mais comum no teste de unidade e de integração.
- Também requerem conhecimento do programa para serem aplicados.
- Exigem a análise de associações definição-uso quanto à sua executabilidade.

Exercício

1. Criar o CFG para o código a seguir.
2. Incluir as anotações de fluxo de dados (definições e usos de variáveis), considerando o programa e o GFC a seguir.
3. Considerando o critério Todos-Usos, derive 10 associações definição-uso válidas.

Programa Sort – Bolha

```
2       public void bolha(int[] a, int size) {
3           int i, j, aux;
4           for (i = 0; i < size; i++) {
5               for (j = size - 1; j > i; j--) {
6                   if (a[j - 1] > a[j]) {
7                       aux = a[j - 1];
8                       a[j - 1] = a[j];
9                       a[j] = aux;
10                  }
11              }
12          }
13      }
```