Ingeniería de Software I
Más Testing

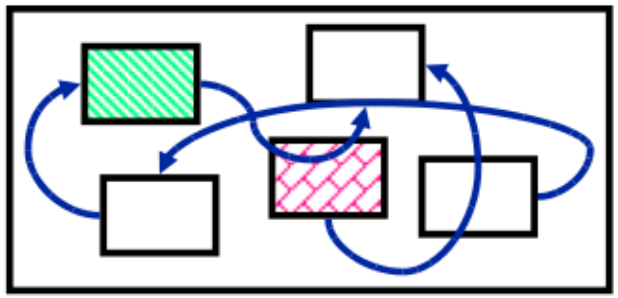
Jocelyn Simmonds

Departamento de Ciencias de la Computación

Enfoques para diseñar casos de prueba

- Caja negra – en base a requerimientos
  - clases de equivalencia
  - análisis de fronteras
  - ...

- Caja blanca – en base al código
  - caminos básicos
  - estructuras de control
  - ...



Caja blanca también se conoce como testing estructural ...

¿Cobertura?

- Queremos testear la mayor parte posible de la estructura del programa
- Dos categorías de cobertura:
  - en base al control de flujo
  - en case al flujo de datos

¿Cobertura?

- Queremos testear la mayor parte posible de la estructura del programa
- Dos categorías de cobertura:
  - en base al control de flujo
  - en case al flujo de datos
  - hay una tercera categoría (dependencias sintácticas), pero no entraremos en detalle en este curso

Control de flujo

- Un grafo de control de flujo (control flow graph) es un grafo dirigido (N, E, s, f) tal que
  - N es el conjunto de nodos, representan las instrucciones del programa
  - E es el conjunto de aristas, representan las posibles transferencias de flujo en el programa
  - $s \in N$ es el nodo inicial
  - $f \in N$ es el nodo final

$$E = \{(n_i, n_j) \mid \text{sintacticamente, la ejecucion de } n_j \text{ sigue la ejecucion de } n_i\}$$

Criterios de control de flujo

Cobertura de:

- Instrucciones (statement coverage)
- Decisiones (branch coverage)
- Caminos (path coverage)
- Caminos escondidos (hidden paths)
- Bucles (loops)

Statement coverage

- Cada instrucción del programa debe ser ejecutada al menos una vez
- Más formalmente:
  - un conjunto de caminos P donde
    $\forall n \in N, \exists p \in P$ tal que n esta en el camino p
- Queremos minimizar el numero de casos de prueba, pero preservando la cobertura deseada

Ejemplo

```
x = read();
y = read();
if (x > 0)
        write("1");
else
        write("2");
if (y > 0)
        write("3");
else
        write("4");
```

- Este test suite cubre el 100% de las instrucciones:
  $\{< x = 2, y = 3 >,$
  $< x = -13, y = 51 >,$
  $< x = 97, y = 17 >,$
  $< x = -1, y = -1 >\}$

Ejemplo

```
x = read();
y = read();
if (x > 0)
        write("1");
else
        write("2");
if (y > 0)
        write("3");
else
        write("4");
```

- Este test suite cubre el 100% de las instrucciones:
  $\{< x = 2, y = 3 >,$
  $< x = -13, y = 51 >,$
  $< x = 97, y = 17 >,$
  $< x = -1, y = -1 >\}$
- Pero este test suite es mínimo:
  $\{< x = -13, y = 51 >,$
  $< x = 2, y = -3 >\}$

¿Minimal = suficiente?

```
if (x < 0)
        x = -x;
z = x;
```

Test suite: $\{< x = -1 >\}$

¿Minimal = suficiente?

```
if (x < 0)
        x = -x;
z = x;
```

Test suite: $\{< x = -1 >\}$

Cubre todas las instrucciones, pero no todas las posibilidades de ejecución

En la practica ...

- Es fácil alcanzar una cobertura de un 85%, pero nunca 100%
- ¿Por qué?

En la practica ...

- Es fácil alcanzar una cobertura de un 85%, pero nunca 100%
- ¿Por qué?
  - código inalcanzable: instrucciones que no pueden ser ejecutadas en cualquier contexto, están "desconectadas" del control de flujo del programa
    - no confundir con código muerto: instrucciones que se ejecutan, pero nunca se usan sus resultados
  - secuencias complejas de instrucciones
- Empresas como Microsoft reportan una cobertura de 80-90%

Correctitud coincidental

La ejecución de una instrucción no garantiza que se revele una falla en ese camino.

Ejemplo:

```
y = x * 2
```

VS.

```
y = x ** 2
```

Correctitud coincidental

La ejecución de una instrucción no garantiza que se revele una falla en ese camino.

Ejemplo:

```
y = x * 2
```

VS.

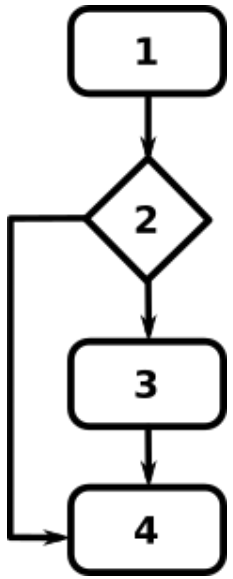
```
y = x ** 2
```

Si $\{< x = 2 >\}$, no se revela la falla

Branch coverage

- Cada transferencia de control de flujo del programa debe ser ejecutada al menos una vez
- $\forall e \in E, \exists p \in P$ tal que p ejercita e
- Más "fuerte" que statement coverage:

```
1  y = read();
2  if y >= 0
3          y--;
4  print y
```



statement:
$p_1 = (1, 2, 3, 4)$

branch:
$p_1 = (1, 2, 3, 4)$,
$p_2 = (1, 2, 4)$

Limitaciones de branch coverage

```
if (x != 0)
        y = 5;
else
        z = z - x;

if (z > 1)
        z = z / x;
else
        z = 0;
```

Test suite:
$\{< x = 0, z = 1 >,$
$< x = 1, z = 3 >\}$

Limitaciones de branch coverage

```
if (x != 0)
        y = 5;
else
        z = z - x;

if (z > 1)
        z = z / x;
else
        z = 0;
```

Test suite:
$\{< x = 0, z = 1 >,$
$< x = 1, z = 3 >\}$

Ejecuta todas las aristas del CFG, pero no expone la posibilidad de división por cero

Path coverage

- Para cada camino entre s y f del CFG, selecciona un caso de prueba
- Más fuerte que branch coverage . . . pero en la practica es infactible

```
n = read();
sum = 0;
for i = 1:n {
        x = read();
        sum += x;
}
```

¿Cómo seleccionamos los caminos?

Caminos escondidos

- $\{< x = 2, y = 5 >, < x = 1, y = 5 >\}$ nos dan branch coverage para:

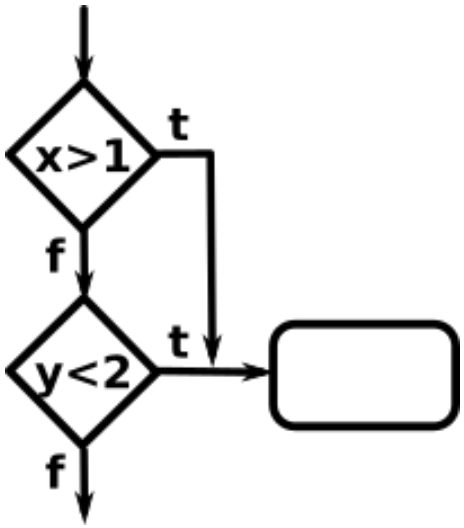
```
if (x > 1 || y < 2)
        ...
```

Caminos escondidos

- $\{< x = 2, y = 5 >, < x = 1, y = 5 >\}$ nos dan branch coverage para:

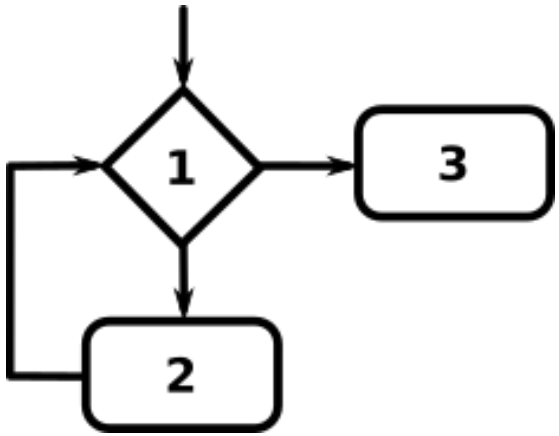
```
if (x > 1 || y < 2)
        ...
```

- Caminos escondidos: cada predicado en una condición compuesta debe ser testeado



Falta un caso de prueba para testear cuando `y < 2` es verdadero

Cobertura de bucles (loops)



Casos a probar:

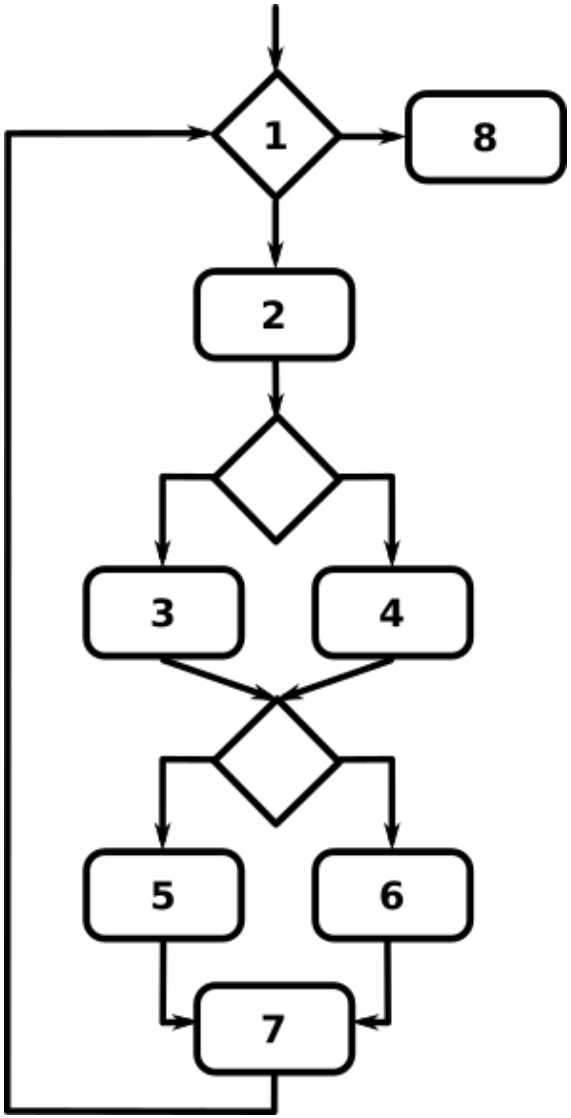
1. No entramos al loop (fall through case): $p_1 = (1, 3)$
2. Numero mínimo de iteraciones: $p_2 = (1, 2, 1, 3)$
3. Numero mínimo de iteraciones + 1: $p_3 = (1, 2, 1, 2, 1, 3)$
4. Numero máximo de iteraciones: $p_4 = ((1, 2)^n, 1, 3)$

En la practica, usualmente incluimos los casos 1, 2 y 3 en nuestras pruebas

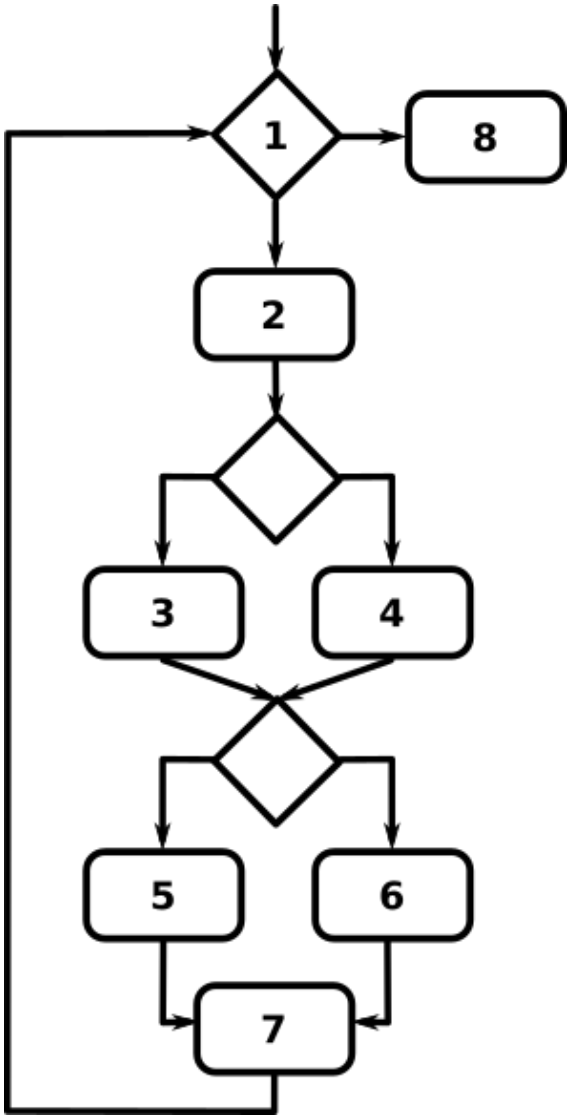
Criterio de frontera/interior (loops)

- Test de frontera (boundary): test que entra al loop, pero no itera sobre ella
- Test interior: test que ejercita el código del loop al menos una vez
- Hay que seleccionar tests de frontera e interior para cada camino distinto del loop

Ejemplo de frontera/interior



Ejemplo de frontera/interior



- caminos de frontera:
  - $a = (1, 2, 3, 5, 7)$
  - $b = (1, 2, 3, 6, 7)$
  - $c = (1, 2, 4, 5, 7)$
  - $d = (1, 2, 4, 6, 7)$
- caminos de interior: para 2 iteraciones del loop
  - (a, a)
  - (a, b)
  - (a, c)
  - (a, d)
  - (b, a)
  - (b, b)
  - ...

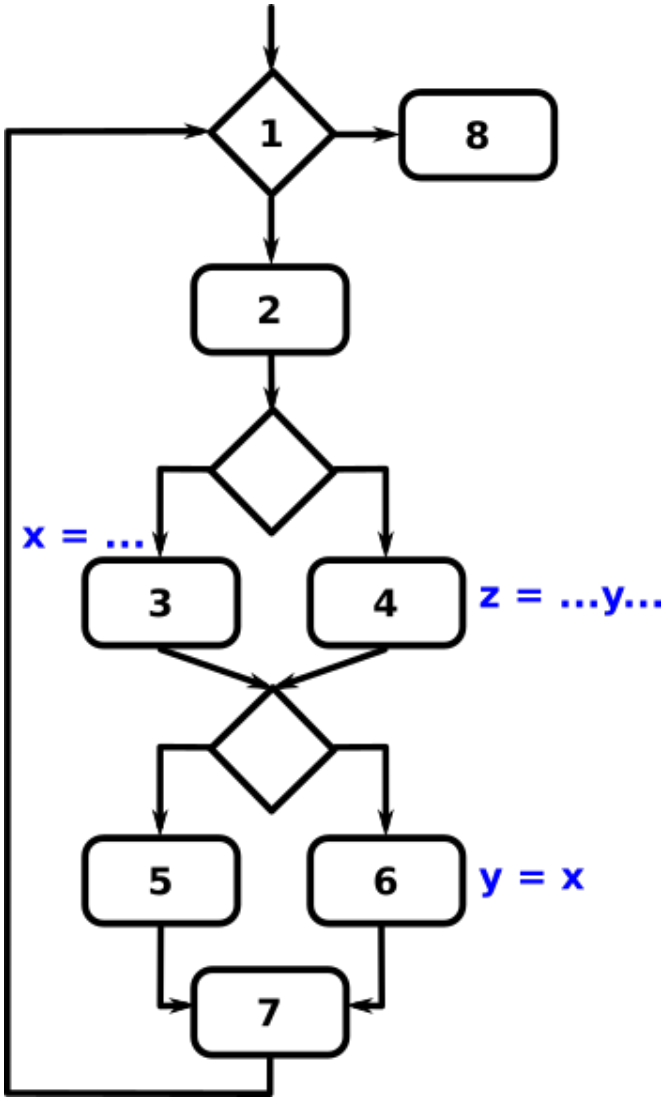
Limitaciones de criterios de cobertura

- Si seleccionamos los casos de prueba en forma estática ...
  - algunos de los caminos elegidos pueden ser infactibles
- Si queremos seleccionar casos en forma dinámica, tenemos que monitorear la cobertura de los casos de prueba
  - indicando las áreas que no tienen suficiente cobertura
- Aun así, los errores ...

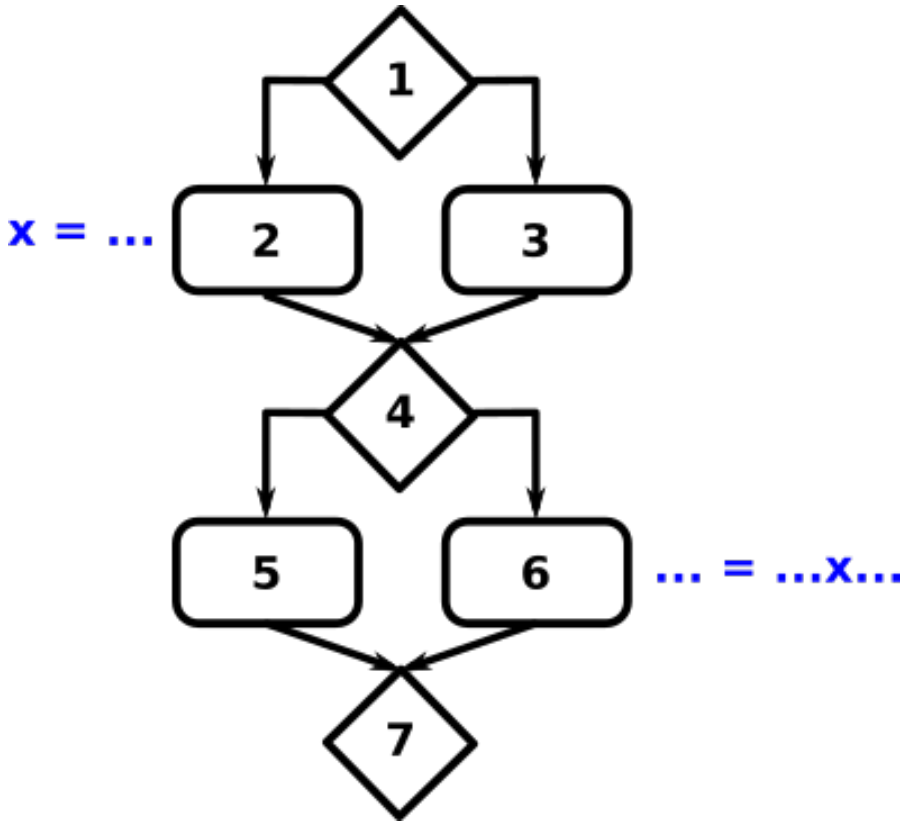
Limitaciones de criterios de cobertura

- Si seleccionamos los casos de prueba en forma estática ...
  - algunos de los caminos elegidos pueden ser infactibles
- Si queremos seleccionar casos en forma dinámica, tenemos que monitorear la cobertura de los casos de prueba
  - indicando las áreas que no tienen suficiente cobertura
- Aun así, los errores ...
  - dependen de combinaciones especificas de instrucciones, más que el nivel de cobertura
  - necesitamos elegir los datos de prueba en forma astuta

¿Realmente son necesarios tantos casos de prueba?



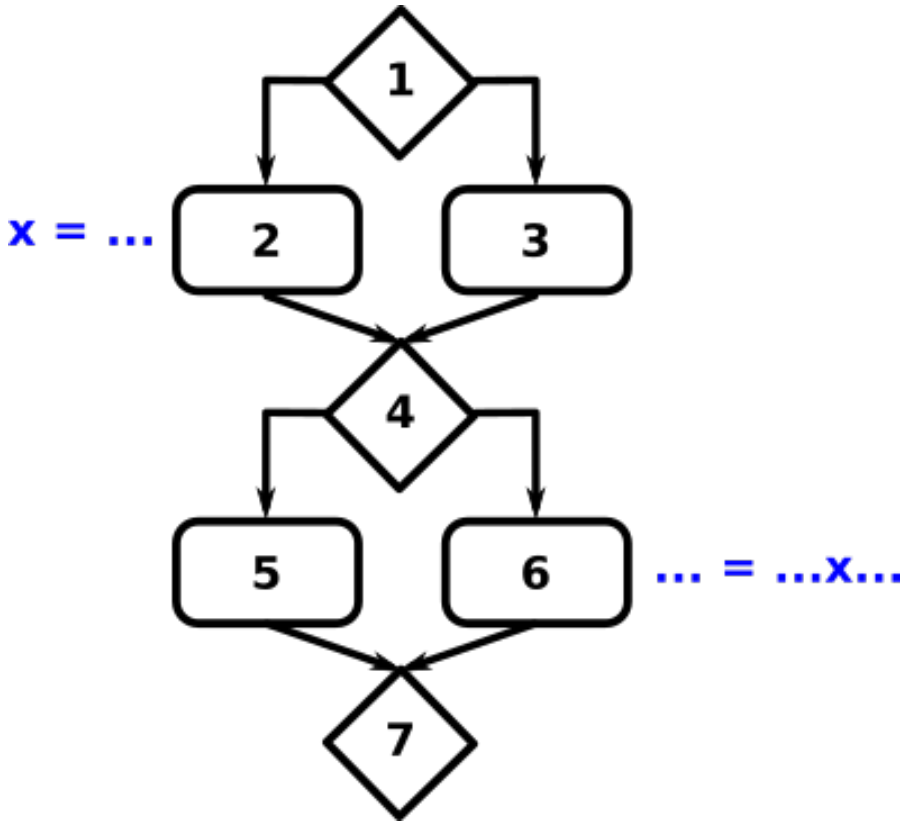
Revisando dependencias de datos



All branches:

$(1, 2, 4, 5, 7)$,
$(1, 3, 4, 6, 7)$

Revisando dependencias de datos



All branches:
$(1, 2, 4, 5, 7)$,
$(1, 3, 4, 6, 7)$

Pero estas pruebas no ejercitan la relación entre la definición de x en 2 y su uso en 6

Algunas definiciones

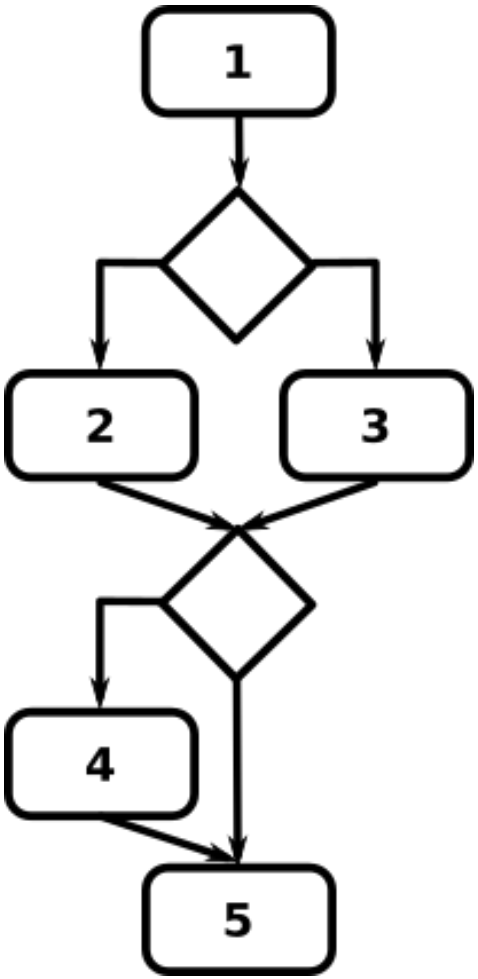
- $d_n(x)$: la variable x fue definida en el nodo n
- $u_m(y)$: la variable y fue usada/referenciada en el nodo m

Algunas definiciones

- $d_n(x)$: la variable x fue definida en el nodo n
- $u_m(y)$: la variable y fue usada/referenciada en el nodo m
- camino libre de definiciones de x (definition-clear path): un sub-camino del CFG que no define x
- una definición $d_m(x)$ alcanza a un uso $u_n(x)$ si y solo si existe un sub-camino (m, p, n) tal que p es un camino libre de definiciones de x

Nuevo objetivo: definir cobertura en base a caminos libres de definiciones entre las definiciones y usos de las variables

Revisando dependencias de datos



Definiciones: $d_1(x), d_2(x)$

Usos: $u_3(x), u_4(x)$

Caminos?

- (1, 3, 4, 5):
  $d_1(x) \rightarrow u_3(x), d_1(x) \rightarrow u_4(x)$
- (1, 2, 4, 4): $d_2(x) \rightarrow u_4(x)$

Criterios de cobertura

- all-defs: incluir un camino libre de definiciones desde cada definición a algún uso alcanzable desde la definición
- all-uses: incluir un camino libre de definiciones desde cada definición a cada uso alcanzable desde la definición
- all-du-paths: incluir todos los caminos libres de definiciones que sean libres de ciclos, o solo sean ciclos simples, desde cada definición a cada uso alcanzable desde la definición

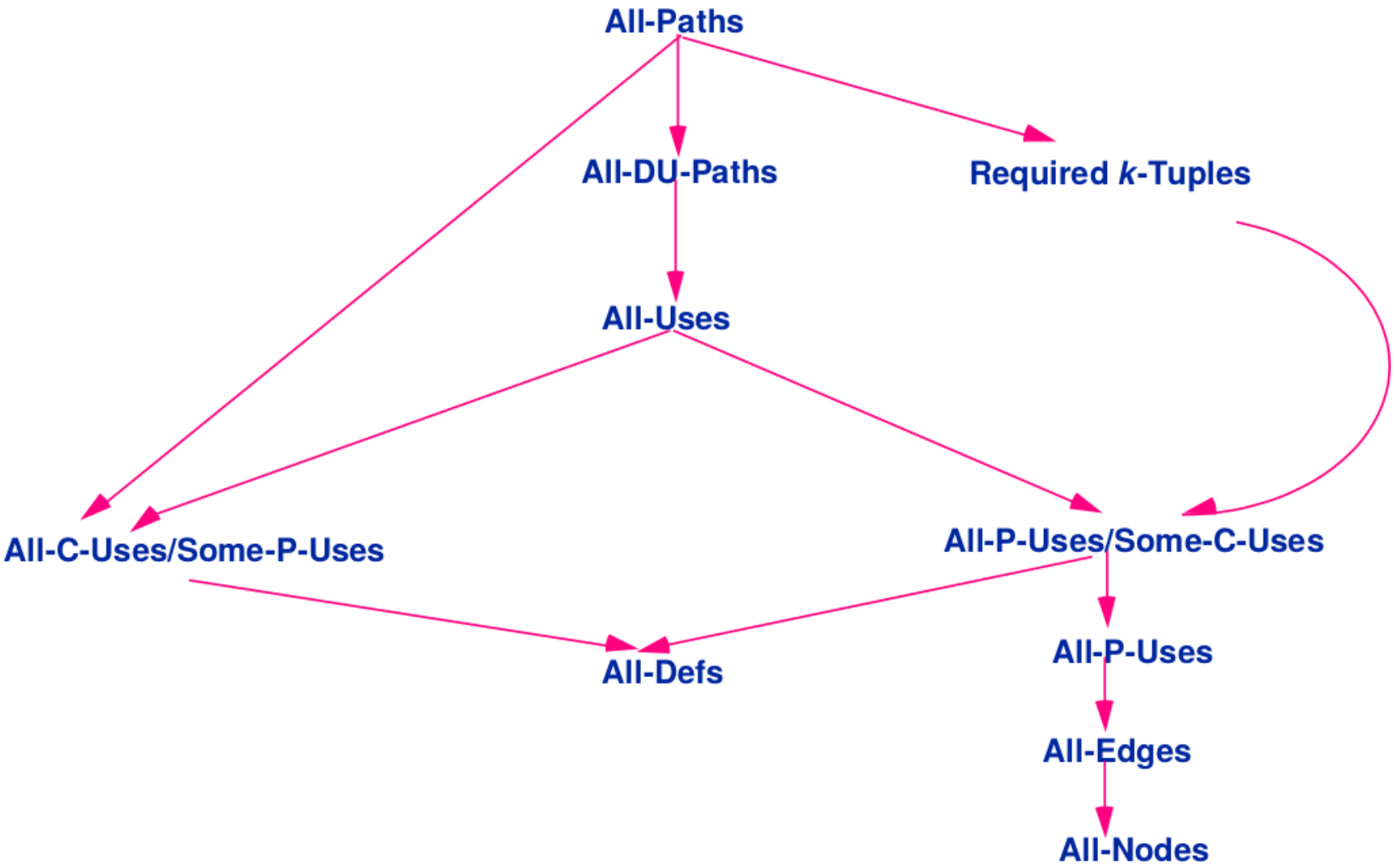
Criterios de cobertura

- También podemos distinguir por el tipo de uso:
  - c-use = uso en el calculo de un valor, p-use = uso en un predicado
  - all-c-uses, some-p-uses: para cada definición de una variable, incluir casos de prueba para todos los c-use de esta, o al menos un p-use
  - all-p-uses, some-c-uses: similar al anterior, pero ahora incluir casos de prueba para todos los p-use asociados a una definición, o al menos un c-use

Criterios de cobertura

- También podemos distinguir por el tipo de uso:
  - c-use = uso en el calculo de un valor, p-use = uso en un predicado
  - all-c-uses, some-p-uses: para cada definición de una variable, incluir casos de prueba para todos los c-use de esta, o al menos un p-use
  - all-p-uses, some-c-uses: similar al anterior, pero ahora incluir casos de prueba para todos los p-use asociados a una definición, o al menos un c-use
- k-tuplas: alternar entre variables
  - ej: tupla 2-dr $d_1(x) \rightarrow u_4(x)$ seguido por $d_5(y) \rightarrow u_{10}(y)$

Relaciones entre criterios de cobertura



Limitaciones de criterios de cobertura de datos

Estos criterios son mejores que los que solo consideran control de flujo:

- Pero son más "caros" ...
- ...y sigue el problema de selección de caminos infactibles
- En la practica, el criterio más usado es el de all-uses
- Podemos mejorar?

Limitaciones de criterios de cobertura de datos

Estos criterios son mejores que los que solo consideran control de flujo:

- Pero son más "caros" . . .
- . . . y sigue el problema de selección de caminos infactibles
- En la practica, el criterio más usado es el de all-uses
- Podemos mejorar?
  - dependencias semánticas: n_j es semánticamente dependiente de n_i si la ejecución de n_i puede afectar la ejecución de n_j
  - en general, no podemos determinar si una instrucción cualquiera depende semánticamente de otra
  - debemos conformarnos con dependencias sintácticas