Capítulo 3 Caminos básicos

Los métodos de prueba analizados en el Capítulo 2 son de tipo funcional; su realización depende de las especificaciones del software. Por ello dejan margen para la existencia de defectos derivados de la implementación, especialmente en aquellos aspectos donde ésta se aleja de las especificaciones.

Para asegurar adecuadamente el software conviene aplicar, además de las pruebas funcionales, alguna prueba de tipo estructural (o de caja blanca), centrada en la implementación. Estas pruebas resultan complementarias, como se hace notar en (Watson y McCabe, 1996).

Existen varios tipos de pruebas estructurales, como son aquellas que ejercitan todas las proposiciones del software bajo análisis, las que ejercitan todas las condiciones y las que ejercitan los caminos básicos. Como se discute más adelante en este capítulo, la prueba estructurada de McCabe, basada en un conjunto básico de caminos, resulta la más conveniente. Así pues, en este capítulo se describe dicha prueba.

El método de McCabe opera así:

1. Convertir el código a un grafo de control
2. Calcular el número ciclomático y obtener la base de caminos
3. Preparar un caso de prueba para cada camino básico

En este capítulo se tratará el método de McCabe [Watson y McCabe 1996]. Para comenzar se presenta el concepto de grafo de control y la manera de construirlo a partir de un algoritmo o programa, después el concepto de complejidad ciclomática y su relación con los caminos de ejecución, luego se explica el método de obtención de una base de dichos caminos y por último la generación de casos de prueba a partir de la base. Se concluye con algunas consideraciones adicionales sobre sus ventajas y la mención de otros métodos relacionados.

3.1 Modelo de control

Los métodos estructurales o de caja blanca descansan sobre el código y la forma en que éste se ejecuta instrucción por instrucción. Por ello el modelo está relacionado con el concepto de control en la ejecución del software, expresado por un grafo de control, sobre el cual se aplican diversos métodos. Así pues, se presenta el grafo de control y cómo se construye.

3.1.1 Grafo de control

El modelo estructural construye un grafo (ver Figura 3.1) equivalente al artefacto bajo estudio, ya sea un procedimiento o un método, donde se cumple:

a) existe un solo nodo inicial
b) existe un solo nodo final

c) cada nodo representa una secuencia de instrucciones, que puede ser vacía.
d) La relación entre los nodos es una relación de “el control pasa a”.
e) Un nodo puede tener uno o dos sucesores,
f) Un nodo puede tener uno o muchos antecesores
g) Un nodo tendrá dos sucesores si existen dos caminos posibles, dependiendo de una condición
h) Los casos de for, while, until y case se pueden reducir a grupos de nodos con dos sucesores.

Por el tipo de relación, a veces se le llama **grafo de control**.

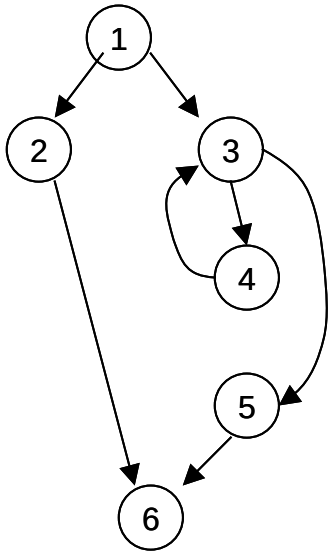


Figura 3.1 Grafo de control

A partir del grafo de control se puede establecer caminos de ejecución y también caminos entre la declaración y el uso de un dato, de donde surgen diversos métodos de prueba.

3.1.2 Construcción del grafo de control

La mayoría de las pruebas estructurales parten de convertir el código de la unidad en un grafo dirigido, llamado grafo de control, donde las instrucciones individuales, o secuencias lineales de ellas, forman nodos y los arcos representan paso de control de la ejecución. En esta sección se describe cómo construir un grafo de control a partir de un programa o algoritmo. En adelante se hablará de programa, pero eso incluye también a un algoritmo.

La ejecución de cualquier programa se realiza en forma secuencial, instrucción por instrucción, a menos que se realice un cambio de dirección[1], debida a una orden específica (salto o fin de iteración) o a la evaluación de una condición. Para

[1] En general, la “dirección” es la siguiente instrucción.

muchos fines, las secuencias de instrucciones no resultan notables pero sí los cambios. Un grafo de control representa un programa de modo que se observan con claridad los diferentes puntos donde existen cambios de dirección, permitiendo analizar los diversos caminos que llevan del inicio al fin del mismo.

El grafo contendrá dos tipos de elementos:

a) nodos que representan una o más instrucciones secuenciales y
b) arcos que representan cambios de dirección.

En principio cada instrucción se representará como un nodo. Ahora bien, existen dos tipos de instrucción:

a) las que tienen un sucesor único, como las asignaciones y las instrucciones de lectura;
b) las que tienen dos o más sucesores posibles, como el if, la condición del for y el case.

Las instrucciones con sucesor único, es decir las secuenciales, pueden representarse una sola en cada nodo o varias de ellas en uno solo, ya que si se ejecuta la primera se ejecutarán las siguientes una tras otra.

La Figura 3.2 muestra varias formas de representar una sucesión de instrucciones, siendo todas ellas equivalentes para los propósitos de cálculo de complejidad y de preparación de pruebas. La diferencia es únicamente de nivel de abstracción, siendo mayor el último caso (el recomendable).

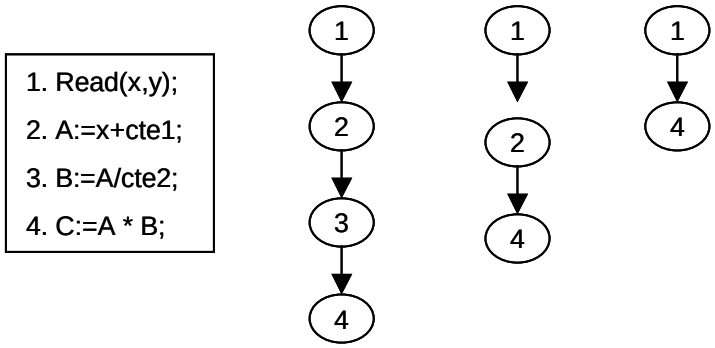


Figura 3.2 Varias formas de representar una secuencia de instrucciones

En cambio, las instrucciones con sucesor múltiple no pueden unirse a una posterior, ya que hay varias alternativas. Sin embargo sí pueden unirse a un grupo secuencial anterior a ellas.

En la Figura 3.3 se muestra un ejemplo con un segmento de código que contiene una instrucción condicional y pueden apreciarse dos formas de representarlo. Note que en ambos casos el nodo que contiene el **if** es el punto de partida de dos caminos alternos. Se incluye una forma alterna donde se cierran los dos caminos alternos en un solo punto de terminación.

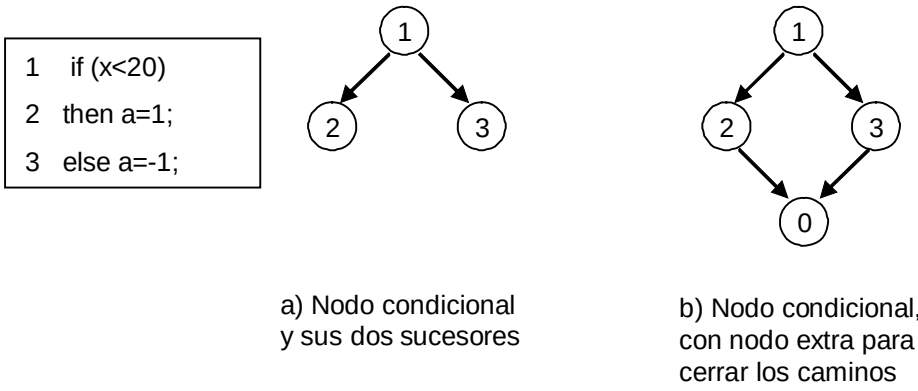


Figura 3.3 Ejemplo de nodos condicionales

La construcción del grafo de control puede emprenderse de dos maneras generales, ambas sistemáticas: descendente y ascendentemente. Ambas se auxilian de las estructuras típicas de programación estructurada, aunque puede usarse en casos no estructurados también. Las estructuras se muestran en la Figura 3.4

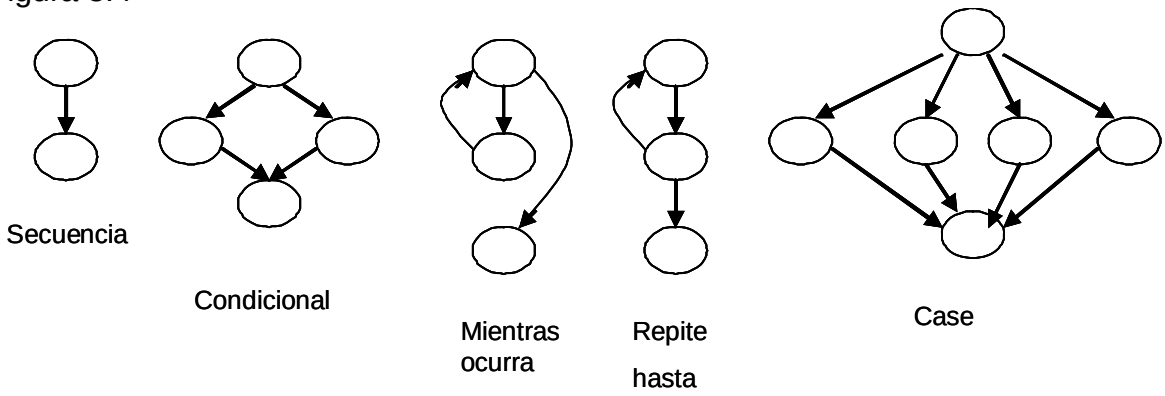


Figura 3.4 Estructuras básicas de la programación

Al ir formando el grafo es de utilidad numerar las instrucciones y utilizar esos números como etiquetas de los nodos, para así simplificar el trabajo. Con cualquier estrategia que se utilice, se recomienda utilizar un nodo inicial y otro terminal únicos, ya que facilita la identificación de los caminos.

Para la explicación de los dos enfoques se utilizará el código de ejemplo de la Figura 3.5, que corresponde a una búsqueda binaria de un valor entero en una lista de valores previamente almacenados. La salida será el índice de la lista donde se encontró o un menos uno si no se encontró. La lista va del lugar lmin hasta el lugar lmax.

3.1.2.1 Paso común

Un primer paso, común a ambos enfoques, permite reducir los elementos a considerar. Como se dijo antes, las secuencias de instrucciones donde no hay alternativas pueden considerarse como un solo nodo; así pues, en este paso se dejará un solo nodo por cada secuencia de este tipo, que es el primer nodo que no

sea secuencial y que sea posterior a todos los elementos de la secuencia. Por ejemplo, en la Figura 3.5 las instrucciones numeradas del 1 al 4 son de tipo secuencial y la 5, un mientras, incluye una bifurcación; entonces, la secuencia entera se puede representar con un nodo con el número 5.

Cuando no se llega a un nodo condicional, pero sí al fin del algoritmo o a un arco que regresa o cambia de dirección, se deja el último nodo de la secuencia. Por ejemplo, para el mismo código, la sucesión 8, 9 y 10 se dejaría como un nodo marcado con el 10.

```
 1  Lee llave
 2  x1=lmin
 3  x2=lmax
 4  resp=-1
 5  mientras (x1<x2-1)
 6     ix=(x1+x2)/2
 7     si (Lista[ix] == llave) entonces
 8        resp = ix
 9        x1 = ix
10        x2 = ix
11     de otro modo si (Lista[ix]<llave) entonces
12           x1 = ix
13        de otro modo x2 = ix
14  fin del mientras
15  regresa resp
```

Figura 3.5 Código de búsqueda binaria

3.1.2.2 Descendente

Con esta estrategia el programa será inicialmente un solo nodo, que se detallará cada vez más hasta donde sea necesario. A cada paso se expande un nodo en varios que lo forman, de acuerdo con las estructuras de la Figura 3.4. Se comienza con los nodos que forman las estructuras iterativas (mientras, while, for) principales, comenzando por el principio del algoritmo. Luego se expanden las estructuras condicionales (if, case). En la Figura 3.6 se muestran varias etapas para el código de la Figura 3.5, los cuales se explican a continuación.

Como primer paso, se identifica la estructura principal que es un *mientras*, que abarca de la línea 6 a la 14. Cuando el *mientras* concluya, seguirá la instrucción 15. De acuerdo a las figuras básicas, se obtiene el grafo de la Figura 3.6 a).

En un segundo paso, el *mientras* (línea 5) se estructura de acuerdo a los patrones mostrados en la Figura 3.4, quedando como se muestra en la Figura 3.6 b).

En el tercer paso, el *mientras* tiene un *if* principal (línea 7) con dos ramas: la línea 10 y las líneas 11 a 13. Al concluir el *if* regresa al inicio el mientras. Note que todas las líneas de control regresan al nodo 5 y cuando falle el mientras, del nodo 5 seguirá el 15. El resultado se muestra en la Figura 3.6 c).

Un cuarto paso expande el nodo del if más interno (11 a 13) en dos ramas: 12 y 13. Al terminar, ambas regresan al nodo 5. El resultado se muestra en la Figura 3.6 d).

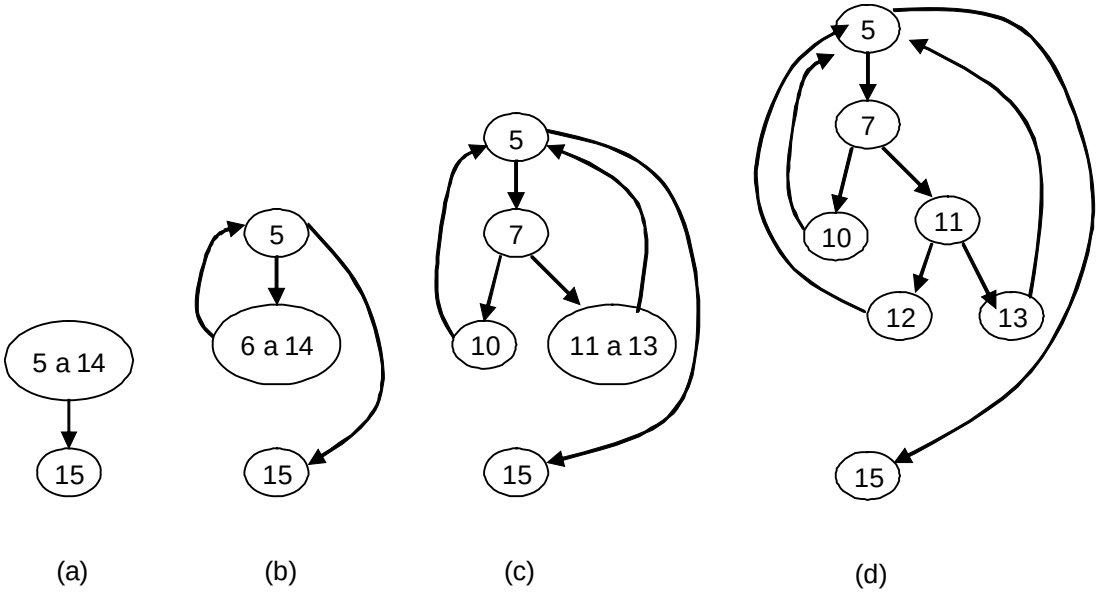


Figura 3.6 Creación descendente del grafo

Al observar las Figuras 3.6 c) y d) a veces resulta desagradable o confuso tener tantos regresos a un sólo nodo (el 5 en éste caso) y algunos prefieren agregar nodos redundantes que sirven para ligar, pero no representan ninguna operación. Tales nodos se etiquetan con un 0, como se muestra en la Figura 3.7.

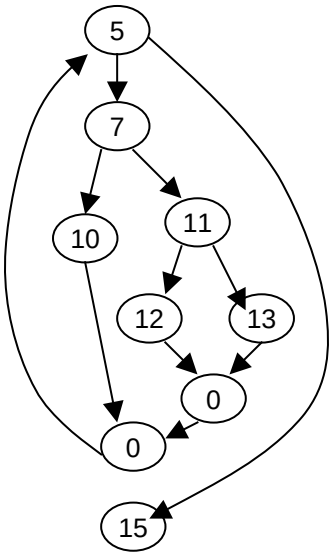


Figura 3.7 Uso de nodos conectores

3.1.2.3 Ascendente

Esta otra estrategia comienza por las instrucciones individuales y se van agrupando formando las estructuras mostradas en la Figura 3.4. Usualmente se comienza por instrucciones que están dentro de las iteraciones o condiciones más profundamente anidadas y se avanza hacia el exterior. Si se identifica un condicional, se reemplaza por la estructura correspondiente. En forma similar se aplica a iteraciones y case. En las iteraciones siempre existirá un nodo dentro de la iteración.

En la Figura 3.8 se muestran varios pasos de la formación ascendente, que se describen a continuación.

En el primer paso se tiene una serie de nodos individuales. Recorriendo el grafo hasta niveles que no tengan anidamiento, se podrá identificar un grupo, correspondiente al Si de la línea 11, formando un grupo con los nodos 11, 12 y 13 (Figura 3.8 a). En este caso el uso de nodos conectores es muy útil, por lo cual se usarán en éste y los siguientes pasos.

En el siguiente paso se identifica el otro Si de la línea 7, que forma una estructura con el nodo 10 y el grupo previamente identificado (11 a 13), dando el grafo de la Figura 3.8 b).

En el cuarto paso identificamos el mientras de la línea 5, que incluye los grupos antes identificados, con su regreso, como se muestra en la Figura 3.8 c).

Finalmente, se conecta el mientras con el nodo final, quedando como en 3.8 d).

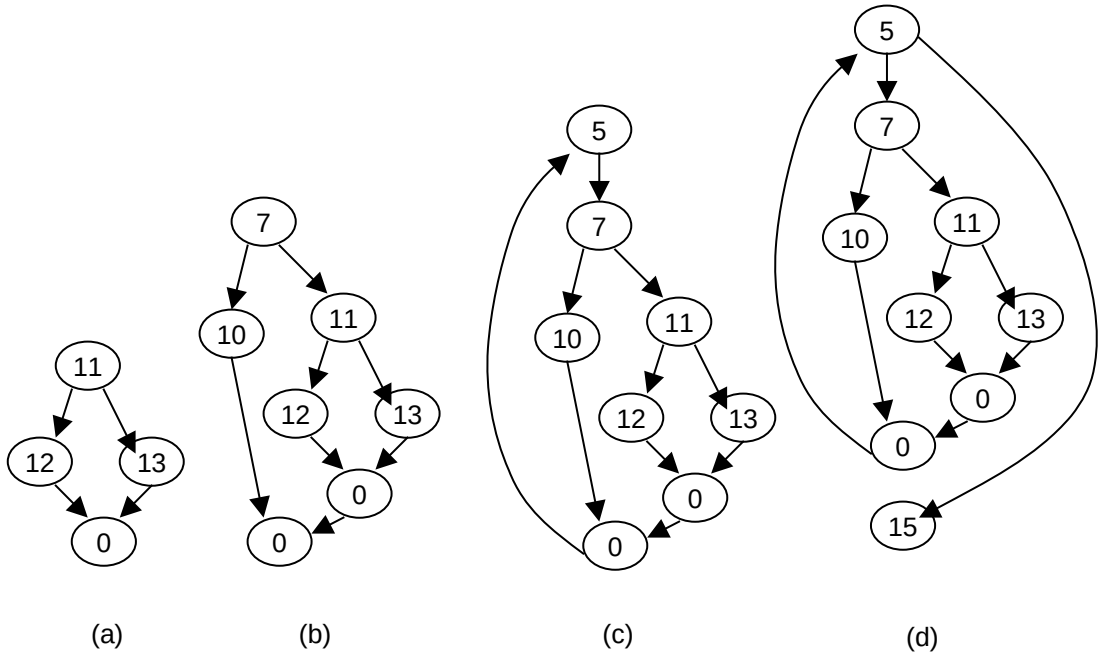


Figura 3.8 Construcción ascendente del grafo

3.1.2.4 Condiciones con varios predicados

Hasta aquí se tiene un grafo de control que puede llamarse “en bruto”. Algunos autores se conforman con él. Sin embargo, es posible que este grafo aún esconda

parte de la complejidad y origine problemas al tiempo de realizar las pruebas. El problema reside en las condiciones con varios predicados que puedan existir, tanto en instrucciones condicionales como en las iteraciones. La razón es la siguiente: aún cuando se tenga una condición "única" como por ejemplo (Alfa>234 AND Beta<0.1), a nivel de ejecución se descompone en varias instrucciones, realizándose primero una instrucción condicional (con Alfa>234 en el ejemplo) y después, si se cumple, se ejecuta otra instrucción condicional (con Beta<0.1 en el ejemplo). En caso de marcar Falso en la primera, la segunda no se realiza, lo cual deja segmentos de código sin probar, a menos que se consideren ambas condiciones. Algo similar ocurre con la alternación.

Para evitar los problemas mencionados, las condiciones deben separarse en un nodo para cada predicado, como se ilustra en la Figura 3.9. Nótese que se acostumbra representar el "entonces" a la izquierda y el "de otro modo" a la derecha.

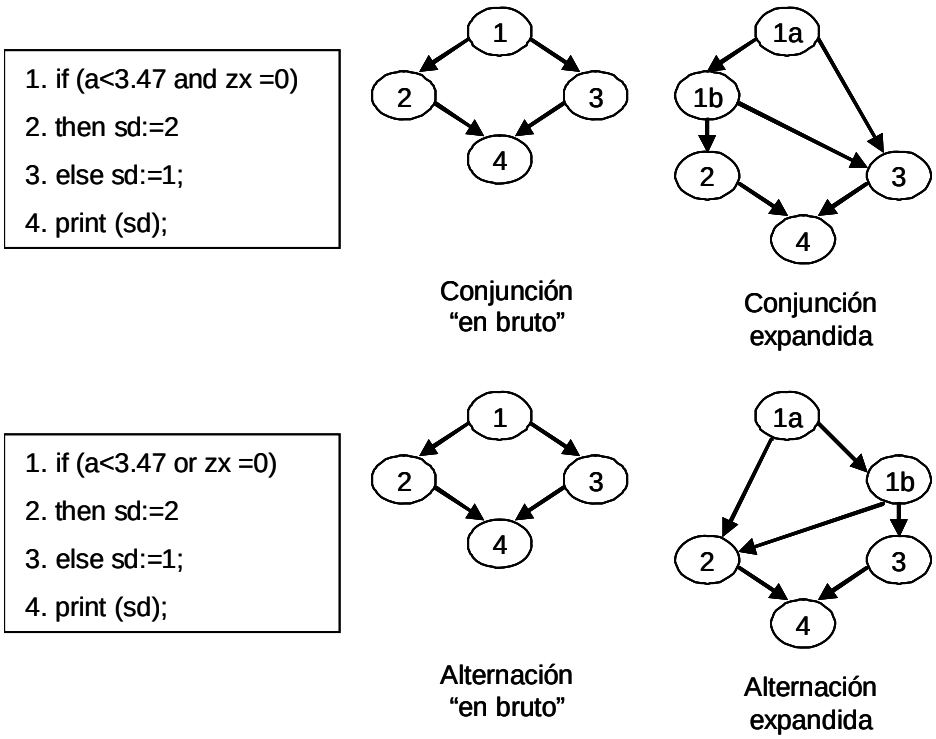


Figura 3.9 Expansión de condiciones con predicados múltiples

3.1.2.5 Ejemplos

En el código de ejemplo mostrado en la Figura 3.12 (ver más adelante) la línea 2 dice:

Si (x<=0) o (y<=0) entonces...

En esa línea existe una condición con dos predicados simples conectados por una disyunción: la primera es (x<=0) y la segunda (y<=0). De acuerdo a lo anterior, le corresponde una expansión de la forma que se muestra en la Figura 3.10.

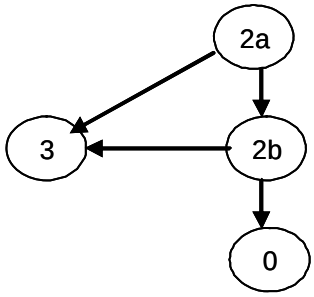


Figura 3.10 Expansión de una condición con dos predicados simples

3.2 Complejidad

Todo aquel que ha programado sabe que el software puede ser más o menos complejo, dependiendo de la naturaleza del problema y de la implementación particular. También se da cuenta de que el software más complejo origina más problemas: falla con mayor frecuencia, es más lento de desarrollar, contiene más defectos. Si, además de desarrollarlo, ha realizado pruebas, sabe que el software más complejo resulta más difícil de probar.

Ahora bien, la complejidad debida al problema que el software pretende resolver no se podrá reducir, por lo cual no se tratará aquí. Sin embargo, la complejidad asociada con una implementación particular sí se puede analizar y aún reducir. Por el momento se tratará su análisis.

Una manera de medir la complejidad del software es el uso de la métrica conocida como “complejidad ciclomática”, propuesta por McCabe (McCabe, 1976) y desarrollada a lo largo de los años. Una buena descripción y discusión es [Watson y McCabe, 1996][2].

La complejidad ciclomática se calcula a partir del grafo de control del software bajo análisis y mide, a grandes rasgos, el número de caminos independientes que pueden ocurrir en la ejecución, entre el inicio y fin del software. En las secciones siguientes se detalla su cálculo.

3.2.1 Cálculo de la complejidad ciclomática

A partir de un grafo de control G=(N, V), donde N es un conjunto de nodos y V un conjunto de arcos entre los nodos, se calcula la complejidad ciclomática de varias maneras que resultan equivalentes. Una es más cómoda para el trabajo manual y las otras para el proceso automático. A continuación se definen las tres formas:

> **Definición 1**: Sean a: número de arcos y n: número de nodos; entonces la complejidad ciclomática o número ciclomático se define como:
>
> $$\nu(G) = a - n + 2$$

Esta es la forma clásica de calcular la complejidad ciclomática, adecuada en programas que lo hacen en forma automática.

[2] Puede localizarse en http://www.mccabe.com y en la página del NIST: http://www.nist.org.

Definición 2: Sea nps el número de nodos con predicado simple (es decir, nodos de los que parten dos caminos); entonces la complejidad ciclomática o número ciclomático se define como:

$$\nu(G) = 1 + nps$$

Esta forma es muy útil, ya que permite omitir la construcción del grafo, centrándose únicamente en los predicados simples del programa. Note que se dice "predicados simples", es decir, supone que ya se expandieron los predicados múltiples.

Definición 3: Sea rr el número de regiones rodeadas completamente por arcos del grafo; entonces la complejidad ciclomática o número ciclomático se define como:

$$\nu(G) = rr + 1$$

Esta definición es de utilidad cuando se calcula manualmente la complejidad ciclomática, ya que se identifican las áreas cerradas en forma visual. En cálculo automatizado no es muy conveniente.

Existe una manera más formal de definir $\nu(G)$, como la cardinalidad de la base del conjunto de caminos posibles en el grafo, pero este concepto se usa, más bien como dato para saber cuántos son los caminos básicos. Ésto se utilizará en lo que sigue.

3.2.1.1 Ejemplos

Ejemplo 1. Para el código de la Figura 3.5, cuyo grafo se muestra en la Figura 3.6, aplicando las definiciones anteriores se tendrá:

Definición:	Cálculo
1	a= 9, n= 7, $\nu(G) = 9 – 7 + 2 = 4$
2	Hay 3 predicados simples; $\nu(G) = 1 + 3 = 4$
3	Hay tres regiones cerradas; $\nu(G) = 3 + 1 = 4$

Si acaso le queda la duda de que pueda cambiar la complejidad en caso de usar nodos conectores, puede aplicarse el mismo cálculo usando la Figura 3.8, que sólo afecta la primera forma de calcular la complejidad:

Definición:	Cálculo
1	a= 11, n= 9, $\nu(G) = 9 – 7 + 2 = 4$

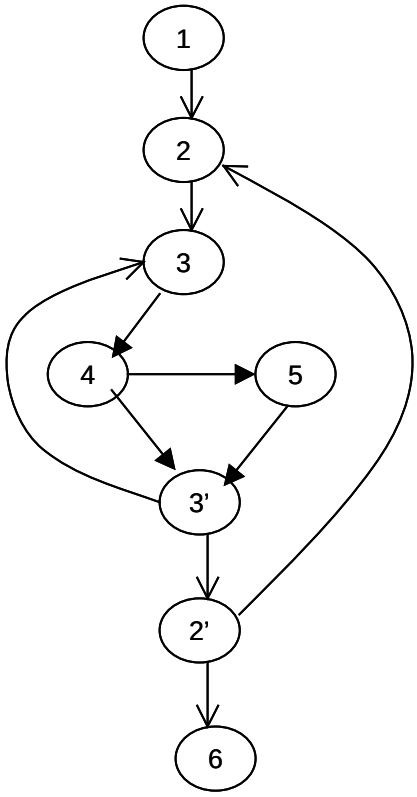
La comprobación de las otras formas queda para el lector.

Ejemplo 2. Considere el código de la Figura 3.11, con su grafo correspondiente. La complejidad ciclomática se puede calcular con cualquiera de las definiciones:

Definición:	Cálculo
1	a= 10, n= 8, $\nu(G) = 10 – 8 + 2 = 4$
2	Hay 3 predicados simples; $\nu(G) = 1 + 3 = 4$
3	Hay tres regiones cerradas; $\nu(G) = 3 + 1 = 4$

```
Calcula cohesión (int nt1, nt2; String tok1[ ],
tok2[ ])

1 numAdh = 0
2 Para i de 0 hasta nt1-1
3     Para j de 0 hasta nt2-1
4         Si tok1[i]=tok2[j] entonces
5             numAdh = numAdh +1
6 total = tok1 + tok2 – numAdh
7 cohesión = numAdh / total
8 regresa cohesión
```



(a) Código (b) Grafo correspondiente

Figura 3.11 Ejemplo de cálculo de cohesión funcional

Ejemplo 3. Considere el código de la Figura 3.12, con su grafo correspondiente. La complejidad ciclomática se puede calcular con cualquiera de las definiciones:

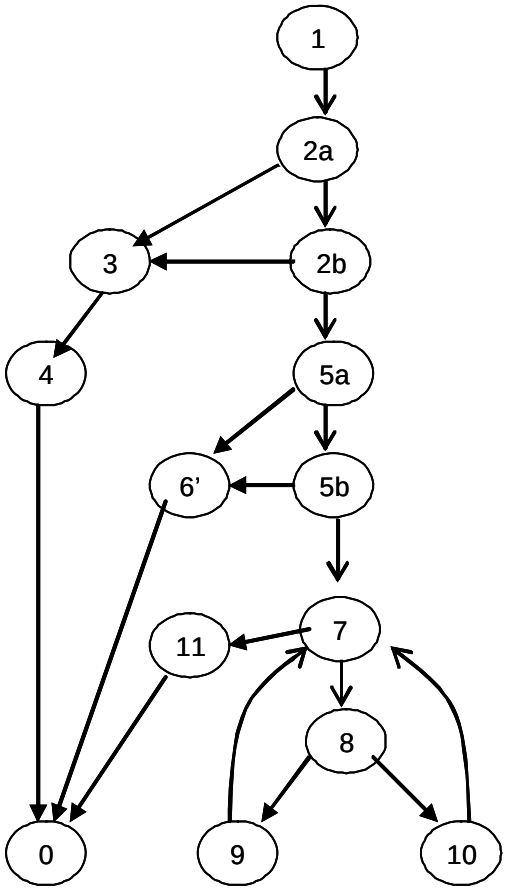
Definición:	Cálculo
1	a= 19, n= 14, ν(G) = 19 – 14 + 2 = 7
2	Hay 6 predicados simples; ν(G) = 1 + 6 = 7
3	Hay seis regiones cerradas; ν(G) = 6 + 1 = 7

3.3 Selección de caminos con la prueba estructurada

Empleando el grafo de control se pueden realizar diversas pruebas. Algunas buscan asegurar que se ejecuten todos los nodos, otros se concentran en los nodos de decisión. Otro enfoque consiste en verificar caminos. En un grafo de control puede haber un número pequeño de caminos, pero también puede existir un número infinito de ellos. Un grafo que incluya un ciclo puede generar éste último caso.

Para evitar el problema de un número grande o infinito de caminos McCabe propuso, a fines de la década de 1970, un método de prueba que consiste en probar un conjunto de caminos que formen una base (en sentido de álgebra lineal) para todos los posibles caminos en el grafo. A tal método llamó “prueba estructurada”.

```
1  lee x, y
2  Si (x<=0) o (y<=0) entonces
3      Escribe “deben ser no negativos;
4      regresa -1
5  Si (x=1) o (y=1) entonces
6       regresa 1
7  Mientras (x <> y)
8       Si (x>y) entonces
9            x = x - y
10     de otro modo y = y – x
11 regresa x
```



(a) Código de máximo común divisor (b) Grafo correspondiente

Figura 3.12 Ejemplo de cálculo del máximo común divisor

Cuando se habla de caminos en un grafo es necesario considerar el inicio y el final, ya que existen muchos posibles caminos. Los que interesan para pruebas de software son caminos que vayan del nodo inicial al nodo final del grafo. Recuerde que un algoritmo tiene un punto de inicio y uno de terminación, que corresponderán con los del grafo. En la práctica puede existir un grafo con varios nodos terminales, pero se les puede reunir con un nodo adicional sin ninguna función.

En los apartados siguientes se discute la prueba estructurada.

3.3.1 Concepto de base de caminos[3]

Cada camino en un grafo puede convertirse en un vector de dimensión igual al número de arcos en el grafo. Cada elemento del vector corresponde a un arco y puede tomar los valores cero o uno, dependiendo de si el arco correspondiente participa en el camino o no.

Por ejemplo, considere el grafo de la Figura 3.11. El camino 1-2-3-4-3’-2’-6 corresponde a un vector con 10 dimensiones o columnas, donde cada arco es una dimensión, como se muestra (se indican los arcos para mayor claridad)

[3] Esta sección se agrega para entender la base algebraica del método, pero puede omitirse en un primer curso de pruebas.

1-2	2-3	2'-2	2'-6	3-4	3'-2'	3'-3	4-3'	4-5	5-3'
1	1		1	1	1		1		

Si se reúnen todos los caminos de un grafo, se forma una matriz. Para el ejemplo, la matriz sería la siguiente:

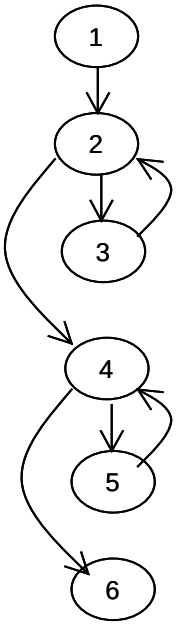
1-2	2-3	2'-2	2'-6	3-4	3'-2'	3'-3	4-3'	4-5	5-3'
1	1		1	1	1		1		
1	1		1	1	1			1	1
1	1		1	1	1	1	1		
1	1	1	1	1	1		1		

A partir de la matriz, empleando métodos algebraicos, se puede obtener su base, es decir un conjunto de vectores que generan todos los demás de la matriz[4].

3.3.2 Prueba estructurada

Sabiendo que la complejidad de un módulo es $\nu(G)$, se sabe que habrá $\nu(G)$ caminos básicos y que bastará probar éstos para tener la seguridad de haber ejercitado todas las proposiciones y todas las condiciones del módulo que se está probando. Sin embargo, el obtener la base de caminos no siempre es tarea fácil y surge la tentación de usar cualquier grupo de caminos. De hacerse así, se corre el riesgo de dejar algún aspecto sin probar.

```
1  i=1     j=1
2  While not eof(a)
3      lee ARRA[i]     i=i+1
4  While not eof(b)
5     lee ARRB[j]     j=j+1
6  regresa
```



(a) Fragmento de código (b) Grafo correspondiente

Figura 3.13 Código de un fragmento de código

[4] Para el manejo algebráico consulte un texto de Álgebra Lineal.

Por ejemplo, considere el fragmento de código de la Figura 3.13 y su grafo correspondiente, que se emplea mucho en diversas aplicaciones.

En ese caso, la complejidad ciclomática es 3, por lo cual se requieren tres caminos. Los siguientes son tres caminos:

1-2-3-2-4-5-4-6
1-2-3-2-3-2-4-5-4-6
1-2-3-4-5-4-5-4-6

Sin embargo, esos caminos no forman una base y dejan sin probar varios caminos importantes, como 1-2-4-6, 1-2-3-2-4-6 y 1-2-4-5-4-6. Estos caminos corresponden a casos donde los archivos están vacíos, lo cual es una situación de excepción, pero de gran relevancia para la operación del software.

Para asegurarse de no cometer errores como esos, conviene seguir un método. Para obtener un conjunto de caminos básicos pueden seguirse al menos dos métodos, conocidos como “simplificado” y “general”. El primero es bueno en algunos casos, pero el general resulta más flexible y más centrado en las funciones importantes del módulo que se prueba. El método simplificado, en términos algebraicos, tiende a hallar una base correspondiente a una matriz triangular inferior, ya que comienza por el camino más corto, que usualmente corresponde a un error. En cambio, el método general tiende a formar una base correspondiente a una matriz triangular superior, ya que comienza con caminos más bien largos. A continuación se describen ambos métodos.

3.3.3 Método simplificado

Como se dijo, éste método comienza seleccionando el camino más corto de principio a fin y luego va buscando segmentos no recorridos hasta completar el número de caminos necesarios. El procedimiento es como sigue:

0: asegurarse que haya sólo un nodo inicial y sólo un nodo final
1: seleccionar el camino más corto entre inicio y fin y agregarlo a la base
2: fijar num-caminos en 1
3: Mientras existan nodos de decisión con salidas no utilizadas y num-caminos<ν(G),
    3.1: seguir un camino básico hasta uno de tales nodos
    3.2: seguir la salida no utilizada y buscar regresar al camino básico tan pronto sea posible
    3.3: agregar el camino a la base e incrementar num-caminos en uno.

Este método tiende a recorrer primero los caminos de excepción, los errores, las salidas de emergencia. Como se verá en el primer ejemplo, primero elige la salida correspondiente a que ambos archivos están vacíos.

Ejemplos.

Para el grafo de la Figura 3.13 se tiene (subrayando los cambios entre dos caminos):

a) el camino más corto es 1-2-4-6
b) si se inicia con ese camino, el nodo 2 tiene una salida no ejercitada: 3, dando el camino 1-2-3-2-4-6
c) de nuevo siguiendo el primer camino, el nodo 4 tiene una salida no usada hacia 5, dando el camino 1-2-4-5-4-6

Para el grafo de la Figura 3.12 con complejidad ciclomática de 7, se tendrán los siguientes caminos:

número	Camino
1	1-2a-3-4-0
2	1-2a-2b-3-4-0
3	1-2a-2b-5a-6-0
4	1-2a-2b-5a-5b-6-0
5	1-2a-2b-5a-5b-7-11-0
6	1-2a-2b-5a-5b-7-8-9-7-11-0
7	1-2a-2b-5a-5b-7-8-10-7-11-0

Por último, para el grafo de la Figura 3.11 con complejidad ciclomática de 4, se tendrán los siguientes caminos:

número	Camino
1	1-2-3-4-3'-2'-6
2	1-2-3-4-5-3'-2'-6
3	1-2-3-4-5-3'-3-4-3'-2'-6
4	1-2-3-4-5-3'-3-4-3'-2'-2-3-4-3'-2'-6

3.3.4 Método general

En este método se elige el primer camino como uno que tenga sentido funcional, es decir, que represente la operación normal del software. Estos caminos omiten, en primera instancia, las salidas de error y las excepciones y recorre al menos una vez cada ciclo. A partir de ese camino inicial, se sigue la misma idea del método simplificado, es decir, ir variando una parte cada vez hasta completar los caminos necesarios. El procedimiento se puede describir como sigue:

0: asegurarse que haya sólo un nodo inicial y sólo un nodo final
1: seleccionar un camino funcional, que no sea salida de error, que sea el más importante a ojos del probador y agregarlo a la base
2: fijar num-caminos en 1
3: Mientras existan nodos de decisión con salidas no utilizadas y num-caminos<ν(G),
    3.1: seguir un camino básico hasta uno de tales nodos
    3.2: seguir la salida no utilizada y buscar regresar al camino básico tan pronto sea posible
    3.3: agregar el camino a la base e incrementar num-caminos en uno.

Ejemplos.

Para el grafo de la Figura 3.13 se tiene:

a) el camino más significativo puede ser 1-2-3-2-4-5-4-6
b) si se inicia con ese camino, el nodo 2 tiene una salida no ejercitada: 4, dando el camino 1-2-4-5-4-6
c) de nuevo siguiendo el primer camino, el nodo 4 tiene una salida no usada hacia 6, dando el camino 1-2-3-2-4-6

Para el grafo de la Figura 3.12 con complejidad ciclomática de 7, se tendrán los siguientes caminos:

número	Camino
1	1-2a-2b-5a-5b-7-8-9-7-11-0
2	1-2a-2b-5a-5b-7-8-10-7-11-0
3	1-2a-2b-5a-5b-7-11-0
4	1-2a-2b-5a-5b-6-0
5	1-2a-2b-5a-6-0
6	1-2a-2b-3-4-0
7	1-2a-3-4-0

Por último, para el grafo de la Figura 3.11 con complejidad ciclomática de 4, se tendrán los siguientes caminos:

número	Camino
1	1-2-3-4-3'-4-5-3'-2'-2-3-4-3'-2'-6
2	1-2-3-4-3'-4-5-3'-2'-6
3	1-2-3-4-3'-2'-6
4	1-2-3-4-5-3'-2'-6

3.4 Generación de casos de prueba

Los caminos básicos identificados en el grafo de control se usan de dos formas durante las pruebas:

a) como guía para verificar que todos los caminos han sido recorridos al menos una vez al ejecutar casos de prueba generados con otro método

b) para generar casos de prueba nuevos

En el primer caso se pueden asociar con una herramienta de trazado de la ejecución de un programa, donde se irán marcando los caminos recorridos. Si todos los caminos básicos están cubiertos por los casos de prueba disponibles, no es necesario crear más.

La otra posibilidad puede emplearse para completar los casos de prueba para caminos no recorridos o bien para generar los casos de prueba desde un inicio. Esta forma tiene sus dificultades, ya que a veces existen caminos para los cuales no es posible generar un caso de prueba. Generalmente esto indica un defecto en el diseño del programa.

Un caso donde puede ocurrir el problema mencionado es el que se ilustra en la Figura 3.14, donde cualquier camino que siga el caso afirmativo para el primer Si nunca podrá pasar por el caso afirmativo del segundo Si. Entonces se tiene un camino imposible. Por supuesto, eso representa un defecto en la estructuración del programa o un error de comprensión del programador.

```
Leer alfa
Si alfa>20 entonces
   begin
      beta:= alfa * 2.5;
      Si beta <=10 entonces
         Avisa(“rama imposible”);
…
```

Figura 3.14 Código con caminos imposibles

El proceso para generar los casos de prueba parte de una observación que quizá el lector ya haya realizado al preparar el grafo de caminos: los elementos importantes para definir los caminos se encuentran en las condiciones, ya sea que correspondan a un *Si*, un *Mientras*, un *Case,* un *For* o un *Repeat*. Así pues, el proceso será como sigue:

a) Seleccione un camino de los que identificó
b) Observe la primera condición e identifique las variables que intervienen
c) Si las variables corresponden a parámetros o se leen directamente, entonces esas variables y los valores contra los que se comparan definen los valores de entrada de los casos de prueba
d) Si las variables son internas, habrá que retroceder hasta encontrar las variables externas (parámetros o leídas) de las que dependen y aplicar la misma idea
e) Si en el camino hay otras condiciones deberá repetirse el proceso, agregando más variables a la parte de entrada del caso de prueba
f) Cuando no haya más condiciones, agregue la parte de salidas esperadas del caso de prueba, de acuerdo a las especificaciones del programa
g) Repita el proceso para los otros caminos.

A continuación se ilustra el proceso con varios ejemplos.

Ejemplo 1: Compra con descuento

Un negocio vende un producto y realiza envíos por mensajería. El producto tiene un precio base de $125 por unidad, pero si se adquieren más de 100 unidades se ofrece un descuento del 5%; si se adquieren más de 1000 unidades el descuento será de 10%. Sobre el precio se agrega el flete que depende de cuantas cajas se requieran. Cada caja puede almacenar hasta 4 unidades y tiene un costo de $50. El programa lee la cantidad pedida y debe calcular el precio total, el número de cajas y el descuento que se aplicó. Así se tiene que al pedir 150 unidades se requieren 38 cajas y alcanza 5% de descuento, por lo que el costo del producto será $125 X 150 = $18750; el descuento será de $937.50, por lo que deberá pagar $17812.50 más el flete: 38 X $50 = $1,900; El total será $19712.50. Expresado como caso de prueba será:

Entrada	**Salida**
Cantidad=150	Costo=19712.50; núm. cajas= 38; descuento 937.50

En la Figura 3.15 se muestra una forma de codificarlo, su grafo de control y los caminos básicos.

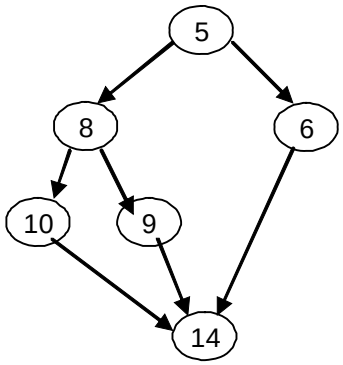
1 Leer cant 2 costo1=cant*125 3 numcaj=redondea(0.5+cant/4) 4 flete=numcaj*50 5 Si cant > 1000 entonces 6 desc = costo1*.1 7 de otro modo 8 Si cant>100 entonces 9 desc = costo1*0.05 10 de otro modo 11 desc = 0 12 costoTot =costo1+flete-desc 13 Escribe costoTot, numcaj, desc		5 – 8 - 9 – 14 5 – 8 – 10 – 14 5 – 6 – 14
(a) Código	(b) Grafo	(c) Caminos

Figura 3.15 Ejemplo de ventas

Con el ejemplo de la Figura 3.15 se toma el primer camino, donde hay una primera condición (línea 5): “Si cant>1000”; retrocediendo, se observa que es una entrada; así pues, como el camino lleva por el lado negativo, se sabe que cant debe ser menor o igual a 1000. Analizando la siguiente condición (línea 8): “Si cant > 100”, opera sobre la misma variable y ahora en caso afirmativo, es decir, x es mayor a cien. Entonces un valor aceptable para el caso de prueba será cant=300. Para esa entrada las salidas son las que se indican en el siguiente caso de prueba:

Entrada	**Salida**
Cantidad=300	Costo=39375; núm. cajas= 75; descuento 1875

En forma similar se calculan los otros dos caminos, dando los siguientes casos de prueba:

Entrada	**Salida**
Cantidad=60	Costo=8250; núm. cajas= 15; descuento 0
Cantidad=1200	Costo=150000; núm cajas=300; descuento 15000

Ejemplo 2: Máximo común divisor (Figura 3.12)

En la Figura 3.12 se presentó el caso de un algoritmo para máximo común divisor, para el cual se identificaron siete caminos. A continuación se describe la obtención de casos de prueba para cada camino.

- Primer camino: 1-2a-2b-5a-5b-7-8-9-7-11-0: La primera condición (2a y 2b) indican que deberá cumplirse x>0 y y>0; la siguiente condición (5a y 5b) indican que x<>1 y y<>1; la condición 8 lleva a x>y y ya no debe volver al ciclo. Por lo tanto, dos posibles valores de entrada serán x=4, y=2.

- Segundo camino: 1-2a-2b-5a-5b-7-8-10-7-11-0: Es similar al anterior, pero debe cumplirse x<y; entonces puede tomarse x=3, y=6.

- Tercer camino: 1-2a-2b-5a-5b-7-11-0: Es similar, pero x=y; entonces se usará x=37, y=37

- Cuarto camino: 1-2a-2b-5a-5b-6-0: El inicio es igual, pero y=1; se propone x=7, y=1

- Quinto camino: 1-2a-2b-5a-6-0: Similar al anterior, x=1 y>1; se propone x=1, y=13

- Sexto camino: 1-2a-2b-3-4-0: En este camino x>0, y<=0; se propone x=18, y=0

- Séptimo camino: 1-2a-3-4-0: Aquí x<=0, y>0; se propone x=-1, y=3

Resumiendo los casos de prueba se tiene:

Entrada	**Salida**
x = 4, y = 2	2
x = 3, y = 6	3
x = 37, y = 37	37
x = 7, y = 1	1
x = 1, y = 13	1
x = 18, y = 0	Mensaje “deben ser no negativos”
x = -1, y = 3	Mensaje “deben ser no negativos”

Debe notarse que se revisaron los caminos básicos, pero no se ha probado el algoritmo cuando debe pasar por varias iteraciones, por lo cual hacen falta otros casos de prueba obtenidos por otro método.

Ejemplo 3 Búsqueda binaria (Figuras 3.4 y 3.6)

El caso de la búsqueda binaria arrojó cuatro caminos básicos. A continuación se describe la preparación de casos de prueba.

- Camino 1: 5-7-11-12-5-15: El número debe no encontrarse en el segundo intento en la segunda mitad del arreglo y no continuar la búsqueda. Esto nos lleva a un arreglo pequeño de tres elementos: suponga que el arreglo contiene los valores 20, 31 y 46. Se propone la llave=42.

- Camino 2: 5-7-11-13-5-15: Similar al anterior, pero en la primera mitad; se propone llave=23.

- Camino 3: 5-7-10-5-15: Lo halla a la primera y sale; se propone llave=31.

- Camino 4: 5-15: Se requiere un camino con un solo elemento, como 20, y cualquier llave diferente al elemento; se propone llave=87.

Note que, al igual que el ejemplo pasado, se han probado casos muy cortos y es probable que se requiera probar caminos más largos con arreglos mayores. También observe que hay problemas en el algoritmo y la preparación de los casos de prueba ya lo indican, aún sin ejecutarlos.

Un resumen de las pruebas será:

Entrada **Arreglo**	**llave**	**Salida**
[20, 31, 46]	42	-1
[20, 31, 46]	23	-1
[20, 31, 46]	31	2
[20]	87	-1

3.5 Pruebas relacionadas

La prueba de caminos básicos está relacionada con otras pruebas que utilizan la misma estructura del grafo. Algunas de ellas son:

a) Todas las ramas
b) Todas las condiciones
c) Todas las instrucciones
d) Caminos de datos

A continuación se describen brevemente tales pruebas, únicamente como complemento, ya que no se tratarán en éste capítulo. Para más información puede consultarse el trabajo de [Beizer, 1995]

3.5.1 Todas las ramas

Esta prueba busca ejecutar suficientes casos de prueba para asegurarse de haber recorrido todas las ramas del programa. Muchas veces se implementa como verificación usando casos de prueba generados por otros medios, en forma semejante al de caminos básicos.

Si se analiza el concepto se verá que está incluido en el de caminos básicos, pero a veces se prefiere para no tener que precisar tales caminos.

3.5.2 Todas las condiciones

Esta prueba es un refinamiento de la anterior, para asegurar que se revisen todas las condiciones individuales, ya que una decisión se puede tomar basándose en una condición múltiple (como (a==b) and (c==d)or (d<s)). Al igual que la anterior, está contenida en el método de caminos básicos, pero algunos consideran más fácil tratar únicamente con las condiciones.

3.5.3 Todas las instrucciones

Esta prueba pretende asegurarse de que todas las instrucciones de un programa se hayan ejecutado al menos una vez. La razón detrás de esa condición es el evitar que exista código no probado, que puede ocultar funciones maliciosas. Existen herramientas que usan el trazado de las ejecuciones para asegurarse que se han usado todas las instrucciones. Si se analiza con cuidado, la prueba de todas las condiciones y la prueba de caminos proporcionan la misma seguridad de manera más directa.

3.5.4 Caminos de datos

Un cambio de enfoque sobre el significado del grafo lo proporciona observar el uso de los datos más que de las instrucciones. Los datos contenidos en variables y constantes tienen dos grandes usos:

1. Cuando se definen (se almacena un valor, ya sea por lectura, copia de parámetros o asignación directa) y
2. Cuando se emplean para producir una salida o emplearlos en un cálculo.

Todo dato bien comportado debe tener un punto donde se declara y al menos uno donde se usa. Entre el primero y el segundo se establece un **camino de datos**. Si se asegura que no haya caminos inconclusos o sin inicio y que se hayan probado todos los caminos de datos, se podrá tener tranquilidad por el algoritmo o programa analizado. Para más información, consulte [Beizer, 1995].

3.6 Referencias bibliográficas

Además de las referencias específicas marcadas en el capítulo, el lector interesado puede consultar el libro de Kit o el de Jorgensen. Esta prueba está incluida en los estándares de pruebas como el de la British Computer Society y el de IEEE.

Beizer, B. "*Black-box testing*", John Wiley & Sons, 1995.

British Computer Society, "*Standard for Software Component Testing*", version 3.3, 1997.

IEEE "*IEEE Std. 1008-1987: Standard for Software Unit Testing*", EEUU, 1987.

Jorgensen, P.C. "*Software Testing: a Craftsman's Approach*", CRC Press, 1995.

Kit, E. "*Software Testing in the real world*", ACM Press – Addison Wesley, Reading, Mass. USA, 1995.

McCabe, T.J. "*Complexity Measure*", IEEE Transactions on Software Engineering, vol. 2, n. 4, pp 308-320, December 1976.

Watson, A.H. and T.J. McCabe, "*Structured testing: a testing methodology using the cyclomatic complexity metric*", Technical Report NIST 500-225, 1996.

Actualizado el 23/11/2006