Pruebas de Software 2017-18

Solución de la PEC-1:

Cálculo de los valores máximo y mínimo de un vector

Dpto. de Ingeniería de Software y Sistemas Informáticos

Rubén Heradio, rheradio@issi.uned.es

ETSI Informática, Universidad Nacional de Educación a Distancia

Índice

1. Contexto de la práctica: determinación de los valores mínimo y máximo de un vector

Se desea construir un algoritmo para identificar los valores mínimo y máximo de un vector. Suponiendo que los índices de un vector van de 0 a su longitud-1, el método obvio para ello sería:

1. Comenzar asumiendo que el mínimo y el máximo están en la primera posición, i.e., min = max = v_0
2. Iterar sobre cada elemento v_i desde la posición i=1 hasta la i=longitud-1, haciendo la siguiente comprobación:
   - Si $v_i <$ min, entonces min pasa a valer v_i
   - Si $v_i >$ max, entonces max pasa a valer v_i

Imaginemos que estamos trabajando en un método más rápido, que recorra los elementos del vector de dos en dos, en lugar de uno en uno. La función de la Figura 1 sería nuestra implementación de dicho método en el lenguaje de programación Ruby.

La Tabla 1 resume la ejecución del código para el vector [2,5,6,4,1,9,8].

Ejemplo de ejecución de Fig. 1: `calcular_min_max( [2,5,6,4,1,9,8] )`	
Línea en Fig. 1	**Resultado**
6	`min=2, max=2`
8	`i=1`, por tanto: `v[i]=5, v[i+1]=6`
14-16	`max=6`
25	`i=3`, por tanto: `v[i]=4, v[i+1]=1`
18-20	`min=1`
25	`i=5`, por tanto: `v[i]=9, v[i+1]=8`
21-23	`max=9`
28	`min=1, max=9`

Tabla 1: Ejemplo de ejecución del código de la Fig. 1

```
def calcular_min_max(vector)
  if vector == []
    return nil
  else
    longitud = vector.length
    min = max = vector[0]
    if longitud > 1
      i = 1
      while i < (longitud-1)
        if vector[i] < vector[i+1]
          if min > vector[i]
            min = vector[i]
          end
          if max < vector[i+1]
            max = vector[i+1]
          end
        else
          if min > vector[i+1]
            min = vector[i+1]
          end
          if max < vector[i]
            max = vector[i]
          end
        end
        i = i + 2
      end
    end
    return min, max
  end
end
```

Figura 1: Implementación de un algoritmo eficiente para determinar los valores mínimo y máximo de un vector

2. Enunciado

Se trata de comprobar la corrección del código de la Figura 1. Para ello, proponga un juego de pruebas de:

1. Caja negra
2. Caja blanca. En concreto, las pruebas deberán lograr una cobertura del 100 % de las sentencias.

3. Solución

3.1. Prueba de caja negra

En la prueba de caja negra, el *tester* ignora la implementación del algoritmo. En este ejemplo, la única *dimensión principal* que merece consideración es la longitud del vector, por que es la principal característica de un vector que puede variar el comportamiento del algoritmo (por ejemplo, si el vector no tiene elementos, la función no debería devolver ningún valor mínimo ni máximo).

Como el dominio de las longitudes del vector es infinito, lo particionaremos para evitar tests redundantes. La idea que subyace en la partición de un dominio en clases de equivalencia es: "si parece que el SUT procesa de la misma manera cierto conjunto de valores, debería bastar con probar cualquiera de dichos valores". La Tabla 2 muestra una posible partición en clases de equivalencia y propone valores de prueba para cada una de esas clases.

Clase válida	Clase inválida	Valor de prueba	Resultado esperado
`longitud` ≥ 1		`longitud`=1 $\Rightarrow$ `vector` = [2]	`min=2, max=2`
		`longitud`=5 $\Rightarrow$ `vector` = [3,5,6,4,1]	`min=1, max=6`
	`longitud` = 0	`vector` = []	`nil`

Tabla 2: Clases de equivalencia para la longitud del vector

Puede comprobarse que ninguno de los valores de prueba detecta errores.

3.2. Prueba de caja blanca: cobertura total de sentencias

Para lograr una cobertura del 100 % del código hay que seguir los siguientes pasos:

Paso 1 *Identificar las decisiones.* La Tabla 3 resume todas las decisiones existentes en la Figura 1, indicando las líneas de código donde aparecen.

Paso 2 *Identificar las ramas de las decisiones.* La Figura 2 representa todas las ramas posibles que surgen al hacer verderas (T=true) o falsas (F=false) las decisiones.

Paso 3 *Definir valores de prueba para cada rama.* La Tabla 4 propone un caso de prueba para cada rama identificada en la Figura 2. En la tabla se ha resaltado el caso de prueba T3, que produce un error.

La Figura 3 muestra cómo corregir el error detectado añadiendo las líneas 8-12 .

Decisión	Línea en Fig. 1
D1: `vector == []`	2
D2: `longitud>1`	7
D3: `i<(longitud-1)`	9
D4: `vector[i]<vector[i+1]`	10
D5: `min>vector[i]`	11
D6: `max<vector[i+1]`	14
D7: `min>vector[i+1]`	18
D8: `max<vector[i]`	21

Tabla 3: Decisiones en la Figura 1

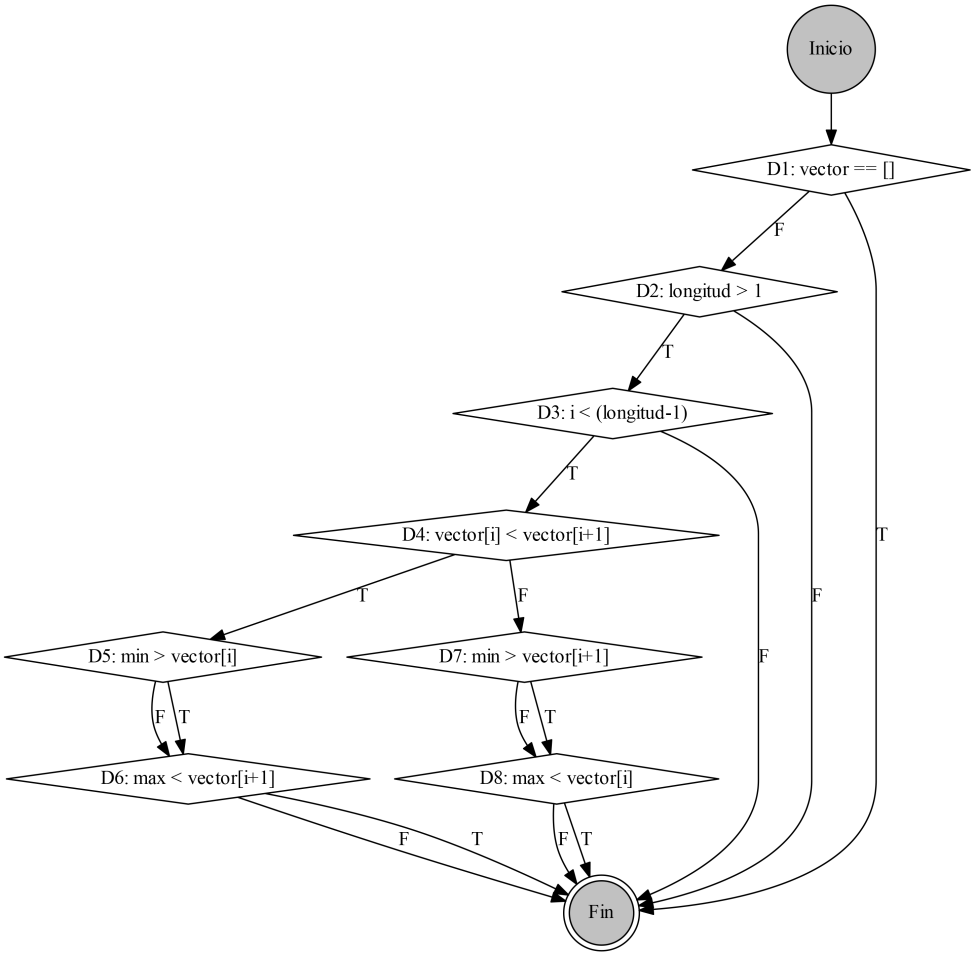

Figura 2: Representación gráfica de todas las ramas de la Fig. 1

Camino	Caso de pr.	Id.	Res. esperado	Res. obtenido
D1=T	[]	T1	nil	nil
D1=F, D2=F	[8]	T2	min=8, max=8	min=8, max=8
D1=F, D2=T, D3=F	[3,2]	T3	min=2, max=3	min=2, max=2
D1=F, D2=T, D3=T, D4=T, D5=F, D6=T	[2,3,4]	T4	min=2, max=4	min=2, max=4
D1=F, D2=T, D3=T, D4=T, D5=T, D6=F	[5,3,4]	T5	min=3, max=5	min=3, max=5
D1=F, D2=T, D3=T, D4=F, D7=F, D8=T	[2,4,3]	T5	min=2, max=3	min=2, max=3
D1=F, D2=T, D3=T, D4=T, D7=T, D8=F	[5,4,3]	T5	min=3, max=5	min=3, max=5

Tabla 4: Valores de prueba según las ramas representadas en la Fig. 2

```
def calcular_min_max(vector)
  if vector == []
    return nil
  else
    longitud = vector.length
    min = max = vector[0]
    if longitud > 1
      if longitud == 2
        i = 0
      else
        i = 1
      end
      while i < (longitud-1)
        if vector[i] < vector[i+1]
          if min > vector[i]
            min = vector[i]
          end
          if max < vector[i+1]
            max = vector[i+1]
          end
        else
          if min > vector[i+1]
            min = vector[i+1]
          end
          if max < vector[i]
            max = vector[i]
          end
        end
        i = i + 2
      end
    end
    return min, max
  end
end
```

Figura 3: Código modificado para corregir el error detectado por el valor de prueba T3

3.3. Conclusiones

En la Sección 3.1 hemos abordado la detección de errores usando pruebas de caja negra, es decir, teniendo en cuenta exclusivamente la entrada y salida del algoritmo. Con este enfoque no hemos conseguido encontrar errores. Sin embargo, en la Sección 3.2 sí hemos logrado detectar errores examinando detenidamente el código.

Podemos concluir que cuanta más información consideremos para diseñar el juego de pruebas, mayor será la sensibilidad de éste para detectar errores (y, por supuesto, también será mayor el trabajo que deberá realizar el tester).

Como última reflexión, interesa señalar que el criterio de cobertura de sentencias no es exhaustivo ni infalible. De hecho, el código de la Figura 3 sigue funcionando mal para otros vectores, como por ejemplo [3, 2, 7, 8], pues indica que el mínimo es 2 pero el máximo 7.