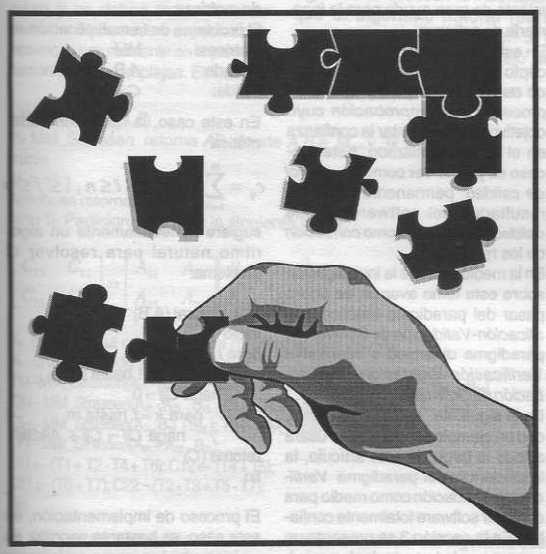
Comprobación probabilística de resultados de programas

Una herramienta para la ingeniería de software



- **Juan Francisco Díaz Frias,** Ph.D.
  Ingeniero de Sistemas
  Profesor Asociado
  Universidad del Valle
- **Mauricio Maca**
  Matemático
  Universidad del Valle
- **Vivian Uzuriaga**
  Matemática
  Universidad del Valle.

RESUMEN

Este artículo presenta una visión global del tema comprobación probabilística de resultados de programas, resaltando la necesidad de introducir la noción de comprobación en el contexto de desarrollo y depuración de software e ilustrando con un ejemplo sencillo la utilidad de esta teoría.

* Este texto se recibió, para ser publicado, en mayo de 1997

ABSTRACT

In this paper we present an overview of probabilistic result checking of programs. We point out the necessity of introducing these ideas in one more general frame of software debugging and software development. We illustrate the utility of this theory by means of a simple example.

1. INTRODUCCIÓN

Clásicamente los procesos de *verificación* y de *validación*, bastante estudiados en la literatura, son utilizados en el desarrollo y producción de software para garantizar su corrección.

El proceso de *verificación* se lleva a cabo en la etapa de diseño de los algoritmos para resolver un problema dado. Consiste en verificar la corrección de los algoritmos con respecto a una especificación dada. Cuando la especificación es formal, este proceso se puede hacer también formalmente y se reduce a demostrar, como se hace en matemáticas, una serie de teoremas a partir de un conjunto de axiomas e hipótesis.

Cuando la especificación no es formal, este proceso se hace de manera ad-hoc y su éxito (el del proceso) depende, en demasía, de la habilidad de las personas que lo lleven a cabo para interpretar acertadamente el lenguaje no formal en

que se especifica y, por consiguiente, es un proceso poco confiable.

El proceso de *validación* es posterior al de verificación; su objetivo es incrementar la confianza en el software desarrollado, independientemente del grado de formalidad con que haya sido realizado el proceso de verificación. El proceso de validación se realiza sobre el software mismo, a diferencia del proceso de verificación que se realiza sobre los algoritmos y no sobre las implementaciones de los mismos.

Una vez terminado el proceso de validación, dos escenarios son posibles:

1. Aquel en el cual todas las pruebas fueron exitosas.
2. Aquel en el cual algunas de las pruebas no tuvieron éxito.

En el primer escenario se confía en la corrección del software y se da el visto bueno para que éste sea utilizado. En el segundo escenario se deduce la no corrección del software, lo que implica devolver las pruebas fallidas a los desarrolladores, con el propósito de corregir los problemas detectados. Este proceso se denomina de *depuración*.

Desafortunadamente, aunque se le dé el visto bueno al software para ser utilizado, este visto bueno no es un certificado que garantice el perfecto funcionamiento del mismo; el visto bueno es solamente un instrumento para incrementar la confianza en que el software desarrollado funcionará correctamente.

Aquí es muy importante resaltar que el proceso de validación, anterior al de utilización del software, no es un proceso dinámico que esté permanentemente en funcionamiento durante la utilización del software, sino estático y termina justo antes de comenzarse a utilizar el software.

Aunque clásicamente existe un proceso de *mantenimiento*, que sí puede ser considerado dinámico, el objetivo de éste no es determinar fallas en el software, sino corregir las que se encuentren en su uso e incrementar su utilidad. En particular, el proceso de mantenimiento, aunque dinámico, no es interactivo, y no permite reaccionar en el mismo momento en que se produce un error en el software.

Teniendo en cuenta estas consideraciones, la existencia de un proceso paralelo al de la utilización, *interactivo,* es decir, que se active cada vez que se ejecute el software, y que permita incrementar la confianza en el software utilizado, ofreciendo una certificación permanente de los resultados que el software produce, sería una herramienta de gran ayuda para la ingeniería de software.

En este artículo se describe el concepto de *Comprobador probabilístico de resultados* y se propone un proceso de *Comprobación* cuyo objetivo es incrementar la confianza en el software utilizado. Este proceso se puede ver como un *control de calidad* permanente sobre los resultados del software, donde *calidad* se entiende como *corrección* de los resultados.

En la medida en que la investigación sobre este tema avance, se podrá pasar del paradigma estático *Verificación-Validación* de Software al paradigma dinámico e interactivo *Verificación-Validación-Comprobación* de Software.

En la siguiente sección se ilustra, con un ejemplo simple que se usará a todo lo largo de este artículo, la insuficiencia del paradigma *Verificación-Validación* como medio para construir software totalmente confiable. En la sección 3 se presenta un panorama de los principales resultados que permiten proponer el paradigma *Verificación-Validación-Comprobación* como un paradigma viable. Posteriormente, en la sección 4, se presentan diferentes alternativas para utilizar las ideas aquí expuestas durante el desarrollo del software, particularmente en el proceso de depuración. Las secciones 5 y 6 se han dedicado a explicar el contexto teórico de las ideas propuestas, así como su aplicación práctica. Por último, la sección 7 destaca algunos trabajos relacionados con el proceso de comprobación y presenta, someramente, los trabajos de investigación que adelantan actualmente, los autores, en esta línea.

2. INSUFICIENCIA DE LA COMBINACIÓN *VERIFICACIÓN-VALIDACIÓN*

Para ilustrar las diferentes ideas y conceptos que aparecen en este artículo, se ha escogido por su sencillez el problema de la *multiplicación de matrices*:

El problema de la multiplicación de matrices: MM
Entrada: $A,B \in Mnxn$
Salida: $C = AB$.

En este caso, la definición matemática

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} \ , 1 \le i \le n \ , 1 \le j \le p$$

sugiere inmediatamente un algoritmo natural para resolver el problema:

```
MM_natural (A,B)
inicio
para i=1 hasta n
   haga para j=1 hasta p
      haga Cij ¬ 0
         para k=1 hasta m
            haga Cij ¬ Cij + AikBkj
retorne (C)
fin
```

El proceso de implementación, en este caso, es bastante sencillo, en un lenguaje de programación como pascal, por ejemplo:

```
procedure producto (vara,b,c:matriz);
var
  i,j,k : integer;
begin
  with c do
    begin
      filas:= a.filas;
      columnas := b.columnas;
      for i:=1 to filas do
        for j:=1 to columnas do
          begin
            elementos[i,j] := 0;
```

```
      for k:=1 to a.columnas do
        elementos[i,j] := elementos[i,j] + a.elementos[i,k]*b.elementos[k,j]
    end;
  end;
end;
```

En este proceso, como en cualquier proceso de implementación de un algoritmo, se debe decidir cuáles son las estructuras de datos que se van a utilizar (en este caso, una matriz pascal), cuáles estructuras de control son las más adecuadas (en este caso se escogió un for, pero había podido ser un while o un repeat) y se deben tomar muchas otras decisiones sobre aspectos que aparecen naturalmente en implementaciones más complejas. Es natural, entonces, que en éste proceso de implementación se presenten errores.

Para ilustrar lo anterior, considere el mismo problema, pero resuelto con un algoritmo diferente: el algoritmo de Strassen. En la práctica se justifica utilizar esta solución pues el algoritmo de Strassen es más rápido que el algoritmo natural (para matrices de dimensión nxn, n^3 51), a pesar de ser más complejo de expresar.

MM_Strassen(A,B)
% MM_Strassen retorna AB, donde $A,B \in M_{nxn}$ y $\exists k : n = 2k$
inicio
si n =1
entonces retorne(AB)
sino % Particione A y B de la siguiente forma:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

% de tal forma que cada letra mayúscula representa
% una matriz de tamaño $2^{n/2} x 2^{n/2}$
$T1 \leftarrow$ MM_Strassen($A_{11} - A_{22}$, $B_{21} + B_{22}$); $T2 \leftarrow$ MM_Strassen($A_{11} - A_{22}$, $B_{22} + B_{22}$)
$T3 \leftarrow$ MM_Strassen($A_{11} - A_{21}$, $B_{11} + B_{12}$); $T4 \leftarrow$ MM_Strassen($A_{11} + A_{12}$, B_{22})
$T5 \leftarrow$ MM_Strassen(A_{11}, $B_{12} - B_{22}$); $T6 \leftarrow$ MM_Strassen(A_{22}, $B_{21} - B_{11}$)
$T7 \leftarrow$ MM_Strassen($A_{21} + A_{22}$, B_{11});
$C11 \leftarrow (T1 + T2 - T4 + T6)$; $C12 \leftarrow (T4 + T5)$;
$C21 \leftarrow (T6 + T7)$; $C22 \neg (T2 - T3 + T5 - T7)$;
Fin

El tiempo de ejecución de este algoritmo es del orden $O(n^{2.8})$ mejorando el tiempo del algoritmo natural que es $O(n^3)$ (n representa el tamaño de la entrada). Esto quiere decir que para n suficientemente grande (en este caso $N \geq 51$) el algoritmo de Strassen es, en teoría, mucho más rápido que el algoritmo natural. En la actualidad hay algoritmos que corren en tiempo del orden $O(n^{2.37})$, en la teoría. En la práctica, una mala implementación de un algoritmo teóricamente muy eficiente puede hacer que el tiempo de ejecución real del algoritmo sea superior al tiempo esperado en la teoría.

Por consiguiente, una implementación del algoritmo de Strassen que realmente se comporte en la práctica tan eficientemente como lo dice la teoría será, sin duda alguna, más compleja y de mayores retos que la sencilla implementación del algoritmo natural. La definición de aspectos tales cómo la estructura de datos a utilizar, qué se debe hacer si n no es potencia de dos o cómo manejar la recursión, dará origen a diferentes implementaciones del mismo algoritmo.

En resumen, para un sólo problema, se cuenta con múltiples algoritmos para resolverlo y con múltiples implementaciones de cada algoritmo. Los procesos de verificación y validación deberán repetirse, el primero por cada algoritmo, el segundo por cada implementación.

El proceso de verificación para el problema de multiplicar matrices consiste en demostrar matemáticamente que el algoritmo funciona, o sea, que en efecto resuelve el problema. Para el caso del algoritmo natural esto es evidente. Para el caso del algoritmo de Strassen, aunque menos evidente, no es difícil demostrar que esa sucesión de multiplicaciones y sumas, calculan AB. Sin embargo, esa demostración no asegura que la implementación del algoritmo sea una fiel copia del mismo y mucho menos que funcione bien en todos los casos. Ni siquiera si se trabaja con una metodología adecuada, se puede garantizar que el software funcione bien en todos los casos.

Sería entonces muy importante asegurar, de alguna manera, que los resultados obtenidos por el programa sean correctos; es aquí donde se aplica el proceso de validación que consiste en correr, para el caso del ejemplo, el programa de Strassen con unas entradas típicas (la cantidad, el tamaño, la generación de las entradas dependen de la metodología de validación) con el fin de comprobar que los resultados obtenidos son correctos (aquí aparece una pregunta: dado un resultado, cómo saber si es correcto? comparándolo con una respuesta precalculada a la mano por alguien o, automáticamente, por otro programa? Cómo saber que esos cálculos si están bien?). Este proceso, sin embargo, no garantiza que dadas cualquier par de matrices el resultado obtenido sea correcto, pues las pruebas realizadas siempre estarán limitadas a un número finito de casos, la mayoría de las

veces muy inferior al número de entradas posibles. En conclusión, con esta etapa se gana confianza pero no seguridad.

3. ESTADO DEL ARTE

El tema de *comprobación probabilística de resultados de programas* es relativamente reciente. Su historia comienza cuando M. Blum y S. Kannan definen el concepto de comprobador (checker, en inglés) en [6], construyen comprobadores para algunos problemas de teoría de grupos y para algunos problemas en la clase de complejidad *P*, a la que pertenecen el problema de hallar el máximo común divisor, o el problema de ordenamiento, entre otros.

Posteriormente, en [7], se incorpora el concepto de corrector y de programas que se auto-comprueban o auto-corrigen (self-testing/correcting programs, en inglés), y se presentan técnicas generales para la construcción de parejas de programas Comprobador-Corrector, para problemas numéricos tales como multiplicación entera, multiplicación modular, multiplicación e inversión de matrices, cálculo del determinante de una matriz, división entera, exponenciación modular y multiplicación polinomial.

En [13, 2] se extiende la aplicación de los conceptos anteriores a problemas sobre polinomios y funciones aproximadas, y en [11], se formaliza el concepto de función auto-reducible aleatoriamente (random-self reducible function, en inglés). A partir de estos artículos se comienza a dar mayor solidez e importancia al tema.

Dos artículos recientes le dan una mayor consistencia y madurez al tema: en [4], se hace una revisión global de los avances en el tema, se expresan de forma más consistente los conceptos y se proponen aplicaciones en la ingeniería de software; en [5], se aplican los conceptos de comprobador y corrector al problema que se presentó con el microprocesador Pentium, cuando se le detectaron errores en las operaciones aritméticas.

4. INTERACCIÓN CON LA INGENIERÍA DE SOFTWARE

En el estado actual, se puede asegurar que introducir dentro de las metodologías de desarrollo de software, el desarrollo de comprobadores probabilísticos para cierto tipo de programas (numéricos o de optimización, por ejemplo), sería una práctica muy conveniente que no solamente incrementaría la confiabilidad en el software mismo, sino que ayudaría a los desarrolladores en la etapa de depuración.

- En ese sentido, Blum y Wasserman en [4], proponen la utilización de las ideas siguientes:
- Comprobaciones parciales del resultado: consiste en comprobar solamente ciertos aspectos del resultado. Por ejemplo, si el resultado está compuesto de una estructura y un valor asociado a la estructura que se supone debe ser óptimo, se puede comprobar que dicho valor sea óptimo, sin entrar a comprobar que la estructura tenga asociado, efectivamente, ese valor.
- Comprobación utilizando pruebas interactivas: el concepto de pruebas interactivas (para un poco de historia al respecto ver [1]) es muy útil para hacer comprobación, si se tiene en cuenta que todo problema que pertenezca a la clase de complejidad *PSPACE* (clase de los problemas que se pueden resolver en espacio polinomial en el tamaño de la entrada) se puede verificar eficientemente por medio de un comprobador probabilístico. Por tanto, si se logra construir una prueba interactiva para un problema, que ejecute siempre el mismo tipo de interacciones, con diferentes instancias del mismo problema, entonces se puede construir directamente un comprobador probabilístico para el problema dado.
- Comprobación fuera de línea: consiste en almacenar las entradas y los resultados de la ejecución de un programa, para ser verificados posteriormente.
- Comprobaciones parciales de módulos: consiste en dotar de comprobadores solamente a ciertos módulos del software donde es más factible que se produzcan errores o donde se conoce cómo construir los comprobadores.

5. COMPROBACIÓN PROBABILÍSTICA DE RESULTADOS DE PROGRAMAS: CONTEXTO TEÓRICO

En esta sección se define formalmente el concepto de comprobador, de manera que se obtienen las características expuestas informalmente.

- Sea f una función tal que el mejor algoritmo conocido para calcularla corre en tiempo $O(T(n))$. Un comprobador probabilístico para f es un algoritmo probabilístico con las siguientes características (tomado de [4]):
- Recibe una pareja <x,y> como entrada.
- Devuelve «PASA" ($y=f(x)$) o «FALLA" ($y \neq f(x)$).
- La probabilidad de error en la decisión está acotada por una constante que se puede hacer tan cercana a cero como se quiera.
- El comprobador corre en tiempo $o(T(n))$, es decir en un tiempo considerablemente inferior al tiempo que toma el mejor algoritmo conocido para calcular f.

Esta última característica garantiza que el comprobador no sea un programa que calcule $f(x)$ (no tiene tiempo para hacerlo) y verifique simplemente si su respuesta coincide con la que se está comprobando; además garantiza que los cálculos realizados no se perciban en la complejidad total del problema, de manera que el comprobador se pueda integrar al programa que

calcula f sin perder realmente eficiencia.

6. APLICACIÓN AL PROBLEMA DE LA MULTIPLICACIÓN DE MATRICES

Como se mencionó anteriormente el concepto de comprobador probabilístico se ilustrará aplicado al problema de multiplicación de matrices, MM definido en la sección 1. En esta sección se presentan dos alternativas para comprobar que una matriz dada C es el resultado correcto de la multiplicación de A y B. La primera alternativa propuesta consiste en construir un comprobador exacto utilizando otro programa de multiplicación de matrices; la inconveniencia de esta solución justifica la segunda alternativa: un comprobador probabilístico en lugar del exacto.

6.1. Un comprobador exacto

La idea natural y más inmediata para resolver este problema consiste en realizar el producto de AB con un software bastante conocido y confiable y comparar el resultado con C. Esta solución toma un tiempo del orden $O(n^{2.37})$, utilizando el mejor algoritmo conocido hasta el momento para multiplicar matrices y, desafortunadamente, dependerá de haber verificado que esa implementación funcione perfectamente. Otra desventaja de esta solución es que el tiempo que toma la comprobación puede ser igual o superior al tiempo que tomó el cálculo mismo de C y este es un precio que muy seguramente no se está dispuesto a pagar, pues haría ineficiente la utilización del software.

6.2. Un comprobador probabilístico

Sea $D = AB - C$, la matriz de diferencias entre AB y C. Fácilmente se ve que $AB = C$ si y solo si D es idénticamente cero. El procedimiento descrito en esta sección está orientado a decidir si D es o no idénticamente cero, sin calcular AB. La clave de este procedimieto está en la siguiente observación: Si D no es idénticamente cero, y S es un subconjunto cualquiera de {1,2,...,n} escogido al azar, entonces la probabilidad de que $\sum_{i\in S} D_i$ no sea el vector nulo es superior a ½ (D_i, representa la i-ésima columna de D). Para justificar esta observación, suponga que D no es idénticamente cero y considere i un entero tal que la i-ésima columna de D contiene al menos un elemento distinto de cero. Ahora sea S' el mismo subconjunto S excepto que $i \in S'$ si y sólo si $i \in S$. Puesto que la i-ésima columna de D no es idénticamente cero, $\sum_{i\in S} D_i$ y $\sum_{i\in S'} D_i$ pueden ser simultáneamente cero. Como la probabilidad que S contenga a i es ½ e i era un índice cualquiera de una columna no nula de D, entonces la probabilidad que S contenga al menos un índice de una columna no idénticamente cero es superior a ½, y, por tanto, la probabilidad que $\sum_{i\in S} D_i$ no sea el vector nulo es superior a ½ (recuerde que todas las entradas de D son positivas).

A partir de la observación se puede construir un algoritmo que determine eficientemente si, dadas A, B, C, se tiene que $C \neq AB$, de la siguiente forma:

Freivalds_1

1. Generar S al azar. Defina R, el vector binario de n x 1 que representa a S así:
   $Rj = 1$ si y solo si $j \in S'$
2. Calcule DR es decir, así: $A(BR) - CR$
   %(obsérvese que no es necesario calcular AB)
3. Si $DR \neq 0$
4. Entonces "C no es el producto de A y B"
5. Sino "C es el producto de A y B"

Observe que el tiempo empleado para calcular $\sum_{i\in S} D_i$ es $O(n^2)$. Sin embargo, *Freivalds_1*, no da siempre el resultado correcto como lo ilustra el siguiente ejemplo:

$$A = \begin{bmatrix} 2 & 0 & 5 \\ 3 & 5 & 3 \\ 6 & 5 & 6 \end{bmatrix} \qquad B = \begin{bmatrix} 5 & 3 & 4 \\ 2 & 2 & 0 \\ 1 & 2 & 3 \end{bmatrix}$$

$$AB = \begin{bmatrix} 15 & 16 & 23 \\ 28 & 25 & 21 \\ 46 & 40 & 42 \end{bmatrix} \qquad C = \begin{bmatrix} 15 & 16 & 23 \\ 28 & 26 & 21 \\ 46 & 40 & 42 \end{bmatrix}$$

Aquí C es la respuesta que retorna el programa que calcula el producto de dos matrices, en este caso de forma incorrecta.

Los posibles vectores R, con los que va a calcular $A(BR)$-CR son:

$$\begin{bmatrix} 0 & 1 & 1 \end{bmatrix}^T \qquad \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T$$

$$\begin{bmatrix} 1 & 1 & 0 \end{bmatrix}^T \qquad \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T$$

$$\begin{bmatrix} 1 & 1 & 1 \end{bmatrix}^T \qquad \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}^T$$

$$\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^T \qquad \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$$

Con los vectores de la izquierda se obtiene que $A(BR)$ es diferente de CR y con los de la derecha se obtiene la igualdad. Es decir, el algoritmo *Freivalds_1* se equivoca con probabilidad (1/2) cuando AB es diferente de C. Para mejorar la probabilidad de obtener una respuesta correcta podemos repetir K veces el mismo algoritmo *Freivalds_1*. Se dirá, entonces, que $AB = C$ si y sólo si las K veces *Freivalds_1* responde esto mismo. Si se escoge $K = \lceil \log(1/\beta) \rceil$, se garantizará que la probabilidad de acierto en la respuesta sea por lo menos $1-\beta$, donde β, es un parámetro escogido según la necesidad.

Un algoritmo que utiliza esta idea lo planteó Freivalds también:

Comprobador_Freivalds (A,B,C,β)

```
inicio
para j←1 hasta élog(1/b)ù
  haga para i←1 hasta n
     haga R[i] ¬ Random(0,1)
   si CR ¹ A(BR)
     entonces retorne "FALLA" y termine
  fin
retorne "PASA"
  fin
```

El *Comprobador_Freivalds* es del orden $O(n^2 \lceil \log(1/\beta) \rceil)$ y retorna "FALLA" con probabilidad de al menos $1-\beta$ si $C \neq AB$; en caso contrario retorna "PASA".

6.3. Experimentación

Para experimentar, en la práctica, la eficiencia y corrección del comprobador, se implementaron 2 algoritmos: el natural y una modificación al natural que lo hacía equivocarse algunas veces, de manera aleatoria. El *Comprobador_Freivalds* se corrió para los resultados generados con la implementación que cometía errores.

Para cada β el *Comprobador_ Freivalds* se uso con los resultados de aplicar el software de multiplicación a una muestra compuesta por 100 matrices de orden 10x10 generadas aleatoriamente. Sobre cada respuesta a cada entrada en la muestra se corrió la comprobación 10000 veces, para poder calcular experimentalmente si la implementación estaba cumpliendo con la restricción impuesta por β es decir, que la probabilidad de error del *Comprobador_Freivalds* fuera inferior a β.

Los resultados se pueden ver en la tabla siguiente:

β	Errores	Producto correcto	Producto incorrecto	Probabilidad
0.5	153	171627	828373	0.171
0.4	162	170458	829542	0.170
0.3	163	80304	919696	0.080
0.2	157	84198	915802	0.084
0.1	156	38949	961051	0.038
0.05	145	20932	979068	0.0020

Las columnas se describen así:

- Errores: es el número de componentes en las 100 matrices de la muestra que hacen que C sea diferente de AB.
- Producto correcto: es el número de veces que en el 1.000.000 de ensayos el *Comprobador_Freivalds* reportó que C es igual a AB. En este caso se equivoca en la respuesta ya que siempre C es diferente de AB.
- Producto Incorrecto: es el número de veces en el 1.000.000 de ensayos que el *Comprobador_Freivalds* da la respuesta correcta.

Se puede concluir que la probabilidad con la que el *Comprobador_Freivalds* se equivoca en la práctica está efectivamente por debajo de b.

7. TRABAJOS RELACIONADOS Y TRABAJO FUTURO

Paralelo a la idea de comprobación de resultados, aparece la idea de corregir los resultados, en línea, cuando la comprobación los ha rechazado. Esta idea da origen a la definición formal de *corrector* y a la búsqueda de correctores, en particular para aquellos problemas que ya tienen comprobadores (ver [7, 4] para profundizar).

El problema que se presentó con el microprocesador Pentium, cuando se le detectaron errores en las operaciones aritméticas, dio origen a una investigación tendiente a construir comprobadores y correctores para las operaciones aritméticas básicas que se encuentran en un microprocesador, en general. Los resultados al respecto se encuentran en [5]. Las implementaciones de los algoritmos allí descritos se están realizando como parte de una práctica investigativa de un estudiante de pregrado en ingeniería de sistemas de la Universidad del Valle.

Por otro lado, la creciente necesidad en estos tiempos modernos, de resolver problemas de optimización donde las entradas son de un tamaño muy grande (optimización en recorrido de redes, minimización de desperdicios en grandes obras de construcción, maximización de ganancias en las ventas por subasta pública de compañías estatales, son algunos ejemplos) ha puesto en evidencia la fragilidad de las diferentes metodologías de validación. En este tipo de problemas, un error, así se presente sólo una vez en la vida, puede llevar a grandes pérdidas económicas que muchas veces pueden llevar al fracaso a una compañía. Por ello, el GEDI, dentro de la línea de investigación en *Teoría de la Computación*, está investigando en este tema de comprobación, para generar el conocimiento necesario para construir e implementar comprobadores probabilísticos para problemas de optimización.

Por último, y en un esfuerzo por que los resultados de investigación permeen el pregrado, el Grupo está trabajando en integrar estos mecanismos de comprobación, como parte de la calificación de los proyectos de los estudiantes dentro del curso de Análisis de Algoritmos. Actualmente se construyen los comprobadores para los diferentes proyectos que se han realizado en el curso.

8. BIBLIOGRAFÍA

[1.] BABAI. L, E-mail and the unexpected power of interaction. In *proceedings of the*, pages 30-43, 1990.

[2.] BABAI. L, FORTNOW. L., LEVIN. L. A., and SEEGEDY M., Checking computations in polygorithmic time. *Communications of the ACM*, pages 21-31,1991.

[3.] BALCÁZAR. J. L, DÍAZ J., and GABARRÓ. J., *Structural Complexity I.* Springer- Verlag, 1988.

[4.] BLUM. M., WASSERMAN. H., Software reliability via run-time result checking. TR 94-053, International Computer Science Institute, 1994.

[5.] BLUM. M., and WASSERMAN.H., Reflections on the pentium division bug. In *Proc. 8th International Software Quality Week*, 1995.

[6.] BLUM.M, and KANNAN.S, Designing programs that check their work. *Communications of the ACM*, pages 86-97, 1989. Computer Science Division. Uni-

versity of California at Berkeley.

[7.] BLUM. M, LUBY. M, and RONITT. R., Self-testing/correcting with aplications to numerical problems. *Communications of the ACM*, pages 73-83, 1990.

[8.] BOVET. D.P., and CRESCENZI. P, *Introduction to the theory of complexity.* Prentice Hall, International, 1994.

[9.] CORMEN. P., LEISERSON. Ch., and RIVEST. R., *Introduction to Algorithms.* MIT press, 1990.

[10.] DÍAZ. J. F., Verification probabiliste de problemes de graphes: Aplication a la planification avec incertitudes, doctoral dissertation. 1993.

[11.] FEIGENBAUM.J, and FORTNOW. L., On the random. Selfredicibility of complete sets. *IEEE*, pages 124-133, 1991. La versión final aparecerá en SIAM Journal on Computing.

[12.] GAREY. M.R, and JOHNSON. D. S, *Computers and intractability A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[13.] GEMMELL. P, LIPTON. R, RUBINFELD. R, and WIDGERSON. S. M., Selftesting/correcting for polynomials and for approximate functions. *Communications of the ACM*, pages 32-42, 1991.

[14.] HOPCROFT. J.E., and ULLMAN. J. D., Intractable problems. In *Introduction to Automata Theor, Languages, and Computation*, chapter 13, pages 320-376. Addison-Wesley, 1979.

[15.] JHONSON D., *A catalog of Complexity Classes*, pages 67-161. Elsevier Science publisher, 1990.

[16.] PAPADIMITRIOU. C, and YANNAKAKIS. M, Optimization, approximation, and complexity classes. *Journal of computer and system sciences*, 43:425-440, 1991.

[17.] PAPADIMITRIOU. C. H., *Computacional Complexity.* ADDISON WESLEY, Publishing Company, 1994

AUTORES

Juan Francisco Díaz Frias.

Matemático, Ingeniero de Sistemas, Ph.D. en Informática de la Universidad del Paris XI -Orsay; es Profesor Asociado del Departamento de Ciencias de la Computación, Facultad de Ingeniería de la Universidad del Valle, investigador principal de la línea de investigación en *Teoría de la Computación* del Grupo de Estudios Doctorales en Informática, GEDI de la Universidad del Valle.

Mauricio Maca. Matemático y actualmente realiza su tesis de grado en el Programa de Maestría en Matemáticas de la Facultad de Ciencias en la Universidad del Valle. Es integrante del grupo GEDI.

Vivian Uzuriaga,

Matemática y actualmente realiza su tesis de grado en el Programa de Maestría en Matemáticas de la Facultad de Ciencias de la Universidad del Valle. Es integrante del grupo GEDI.

Aunque recorramos el mundo en busca de la belleza, sino la sentimos dentro, nunca la encontraremos

Emerson