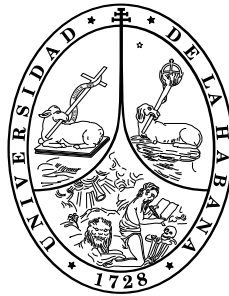


Universidad de La Habana
Facultad de Matemática y Computación
Departamento de Programación



Título

Generación automática de casos de prueba con aprendizaje de
la medida de calidad

Autor: **Marcel Ernesto Sánchez Aguilar**
Tutor: **Ludwig Leonard Méndez**

Trabajo de Diploma de presentado en opción al título de Licenciado en
Ciencia de la Computación

Glosario

- Software: Conjunto de programas y rutinas que permiten a la computadora realizar determinadas tareas.
- Casos de prueba: Conjunto de elementos de entrada que se utilizan para evaluar un programa.
- Algoritmo: Secuencia ordenada y finita de operaciones que permiten resolver un problema.

Capítulo 1

Introducción

La Ciencia de la Computación estudia los fundamentos teóricos de los procesos computacionales y su aplicación en la implementación de sistemas computacionales, en correspondencia con el desarrollo vertiginoso de la propia ciencia y las tecnologías computacionales, los cuales dan solución a la informatización que demanda la sociedad contemporánea. El origen de esta ciencia es anterior a la invención del primer computador moderno, pues desde la antigüedad han existido procedimientos y algoritmos para realizar cálculos, solo que estos eran llevados a cabo por personas y no por un ente digital. Gracias a los trabajos de Alan Turing [1] -considerado el padre de esta ciencia- y otros investigadores, se abrió el camino a nuevas ramas como la computabilidad y se profundizó en otras como la Inteligencia Artificial.

Hoy en día la computación ayuda al hombre en casi todas las áreas y muchas tareas que antes se hacían de forma manual, son responsabilidad ahora de un equipo de cómputo. Pero, ¿qué tan bueno puede ser un software y cómo expresamos la calidad de los mismos? El presente trabajo se enfoca en dar solución a un subconjunto de esta problemática: evaluar la calidad de una componente de un software.

Las componentes de un software son partes de una aplicación, se dividen en componentes para simplificar la complejidad que resulta la confección de un sistema computacional. Esto facilita el mantenimiento, desarrollo y operaciones que permiten a un mismo código ser usado en varios lugares. Específicamente, se abordará la componente lógica, la cual es vital para el correcto funcionamiento de cualquier sistema de cómputo.

Una vez fijado el objetivo de evaluación, quedaría por designar las métricas a utilizar para dicho fin. ¿Cómo evaluar una componente de un software? ¿En qué base a cuáles términos se mide la calidad en un sistema de cómputo? Se dará respuesta a las anteriores interrogantes haciendo referencia al estado

del arte de este problema.

1.1. Estado del Arte

La calidad podría resultar fácil de explicar. Sin embargo, no lo es ni en su propia definición. Algunos definen la calidad como el nivel de satisfacción del cliente. Otros dicen que se trata de cumplir con los requisitos del cliente. O bien el estado del software libre de defectos. Para ello se introduce una nueva definición: deuda técnica [2]. Esto no es más que una serie de aspectos que pueden puntuar la calidad de un software. Se han identificado ocho dimensiones de la deuda técnica del producto de software:

- Calidad del código fuente
- Usabilidad, interfaz de usuario y documentación
- Seguridad
- Actuación
- Lógica de negocios
- Calidad de la arquitectura
- Calidad de los datos
- Uso de código fuente abierto

Cada dimensión se mide según las métricas críticas y los niveles que deben alcanzar. Permite recibir una evaluación integral del producto con recomendaciones concretas. Con este enfoque, los programadores entienden cómo se siente realmente el producto. Ofrece una evaluación cuantitativa completa de la calidad del producto.

La deuda técnica no es solo acerca del código, este enfoque permite ejecutar un análisis profundo, el cual ha demostrado que las deudas técnicas se refieren a la calidad general del producto. Comprobar el código del software o de una de sus componentes no es suficiente para comprender la eficiencia del producto, se debe avanzar más.

1.2. Contexto del problema

El proceso de evaluación de las implementaciones de los algoritmos de los estudiantes se realiza de forma semiautomática en la Facultad de Matemática y Computación de la Universidad de La Habana. La generación de los casos de prueba a utilizar es manual y puede resultar engorrosa, pues es necesario cubrir la mayor cantidad de aspectos a evaluar sin incidir de más en algún rasgo para no afectar la calificación. Luego de realizada una generación abarcadora de casos de prueba, se debe escoger ese subconjunto que mejor represente la medida de calidad de efectividad de la componente de software evaluada.

La idea es proveer al sistema de algunas implementaciones para las cuales se conoce de antemano las evaluaciones y luego se debe encontrar el subconjunto de casos de prueba más cerca de la evaluación de referencia. Luego, si dos soluciones obtienen los mismos resultados al usar idénticos casos de prueba, es muy probable que ambas implementaciones deban obtener la misma calificación. Este es un problema de optimización combinatoria que será abordado con métodos de optimización meta-heurísticos.

1.3. Resultados esperados, importancia y herramientas a utilizar

El objetivo general de esta tesis es la generación automática de casos de prueba y la selección o ponderación que permita utilizarlos como medida de calidad para la evaluación de algoritmos. El objeto de investigación es de optimización combinatoria abordado por meta-heurísticas [3] para ser aplicado en la definición de los casos de pruebas de las evaluaciones de los estudiantes de Programación en la Facultad de Matemática y Computación. Ahora bien, ¿qué algoritmo implementar? ¿Por qué una meta-heurística? ¿Será necesario aplicar un modelo de optimización? ¿Es un problema de Inteligencia Artificial o de Combinatoria? A estas y otras interrogantes se dará respuesta a lo largo de este documento.

Para el diseño lógico de esta herramienta se utilizará la aplicación Visual Studio y el lenguaje de programación C#. Esta elección se debe a las facilidades que ambos brindan para la confección de una solución a este problema.

1.4. Ejemplo

Veamos un ejemplo. El algoritmo de Euclides para hallar el máximo común divisor entre dos números.

Definición 1. Máximo Común Divisor de dos números enteros: Es el mayor entero que los divide a ambos sin dejar resto.

Entrada: entero A , entero B

Método a implementar: Algoritmo de Euclides.

Salida: $MCD(A, b)$

En este ejemplo, nuestra propuesta debería generar un conjunto de entrada lo suficientemente abarcador, que permita evaluar de forma justa el algoritmo. Por ejemplo, la entrada debería contemplar el cero, los números primos, negativos, de Fibonacci, entre otros.

Capítulo 2

Marco Teórico

Los problemas de optimización combinatoria que involucren una extensa, pero finita, lista de posibles soluciones son muy comunes en la vida diaria. Por mencionar algunos, entre los más destacados se encuentran el diseño de redes de comunicación y la planificación de rutas de vuelo. Debido a la gran envergadura de estos problemas en cuanto a la cantidad de datos, resulta imposible enumerar las posibles soluciones y quedarnos con la mejor, pues es infactible en tiempo tal enumeración; incluso con los poderes de cómputo actuales, pues dicha lista crece de manera exponencial respecto al tamaño del problema.

En los últimos 50 años se han desarrollado varios métodos de búsqueda, los cuales arrojan una solución factible cercana al óptimo del problema sin necesidad de explorar cada alternativa. Esto es lo que se conoce como Optimización Combinatoria. Los resultados han sido notorios, avances significativos en problemas de importancia para ciencia como lo son el viajante y el enrutamiento de vehículos ya son palpables.

Sin embargo, una buena parte de los problemas encontrados son computacionalmente intratables por su naturaleza o porque son lo suficientemente grandes como para impedir el uso de algoritmos exactos. En tales casos, los métodos heurísticos generalmente se emplean para encontrar soluciones buenas, pero no necesariamente óptimas. La efectividad de estos métodos depende de su capacidad para adaptarse a un ambiente particular, evitar el atrapamiento en los óptimos locales y explotar la estructura básica del problema.

Sobre la base de estas nociones, se han desarrollado diversas técnicas de búsqueda basadas en heurísticas, las cuales han mejorado la capacidad de obtener buenas soluciones a diversos problemas de optimización combinatoria. Algunos de los principales métodos son: Recocido Simulado, Búsqueda Tabú, Algoritmos Genéticos y GRASP (Procedimientos de Búsqueda Adaptativa

Aleatoria Codiciosos).

2.1. Metaheurísticas

Algunas de las metaheurísticas que pueden dar solución al problema.

2.1.1. Greedy Randomized Adaptive Search Procedures

GRASP [4] es una técnica de muestreo aleatorio iterativo. Tiene la invariante de que en cada iteración proporciona una solución factible al problema en cuestión. Hay dos fases dentro de cada iteración del algoritmo: la primera construye inteligentemente una solución a través de una función codiciosa aleatoria; la segunda aplica un procedimiento de búsqueda local a la solución construida con la esperanza de encontrar una mejora. Esto se repite mientras no se alcance una condición de parada, que pudiera ser el cumplimiento de un número de iteraciones o el alcance de un valor en la función objetivo.

En la Figura 2.1 se muestra el procedimiento general del GRASP

```

procedure grasp()
1   InputInstance();
2   for GRASP stopping criterion not satisfied  $\rightarrow$ 
3       ConstructGreedyRandomizedSolution(Solution);
4       LocalSearch(Solution);
5       UpdateSolution(Solution, BestSolutionFound);
6   rof;
7   return(BestSolutionFound)
end grasp;

```

Figura 2.1: Algoritmo General GRASP

Ahora se construye iterativamente una solución factible. En cada iteración de construcción, la elección del siguiente elemento a agregar se determina respecto a una función codiciosa. Para reflejar los cambios provocados por la selección del elemento anterior, los beneficios asociados con cada elemento se actualizan en cada iteración. El componente probabilístico de un GRASP se caracteriza por elegir aleatoriamente uno de los candidatos de la lista, pero no necesariamente el mejor candidato. La lista de los candidatos se denomina lista de candidatos restringidos (RCL). Esta técnica de elección permite obtener diferentes soluciones en cada iteración GRASP.

La Figura 2.2 muestra el pseudocódigo para la fase de construcción de GRASP.

```

procedure ConstructGreedyRandomizedSolution(Solution)
1   Solution = {};
2   for Solution construction not done  $\rightarrow$ 
3       MakeRCL(RCL);
4        $s = \text{SelectElementAtRandom}(\text{RCL})$ ;
5       Solution = Solution  $\cup \{s\}$ ;
6       AdaptGreedyFunction( $s$ );
7   rof;
end ConstructGreedyRandomizedSolution;

```

Figura 2.2: Construcción de la solución GRASP

El algoritmo de búsqueda local es iterativo y reemplaza sucesivamente la solución actual por una mejor en la vecindad. Termina cuando no se encuentra una solución mejor en el vecindario. La clave del éxito para un algoritmo de búsqueda local consiste en la elección adecuada de una estructura de vecindario, técnicas eficientes de búsqueda y la solución inicial.

A continuación, la Figura 2.3 muestra dicho procedimiento.

```

procedure local( $P, N(P), s$ )
1   for  $s$  not locally optimal  $\rightarrow$ 
2       Find a better solution  $t \in N(s)$ ;
3       Let  $s = t$ ;
4   rof;
5   return( $s$  as local optimal for  $P$ )
end local;

```

Figura 2.3: Búsqueda local GRASP

2.1.2. Recocido Simulado

Kirkpatrick, Gelatt y Vecchi (1983) e independientemente Cerny (1985) propusieron un nuevo enfoque para la solución aproximada de problemas de optimización combinatoria. Este enfoque, Recocido Simulado [5] (Simulated Annealing) está motivado por una analogía con el comportamiento de los

sistemas físicos en presencia de un baño de calor. El enfoque no físico puede verse como una versión mejorada de la técnica de optimización local o mejora iterativa, en la que una solución inicial se mejora repetidamente haciendo pequeñas alteraciones locales hasta que dicha alteración no produzca una mejor solución. El Recocido Simulado aleatoriza este procedimiento de una manera que permite movimientos ascendentes ocasionales (cambios que empeoran la solución), en un intento de reducir la probabilidad de quedar atrapado en una solución localmente óptima. El Recocido Simulado se puede adaptar fácilmente a nuevos problemas (incluso en ausencia de una comprensión profunda de los problemas mismos) y, debido a su aparente capacidad para evitar los óptimos locales deficientes, ofrece la esperanza de obtener resultados significativamente mejores.

Para comprender el Recocido Simulado, primero se debe entender la optimización local. Se puede especificar un problema de optimización combinatoria identificando un conjunto de soluciones junto con una función de costo que asigna un valor numérico a cada solución. Una solución óptima es una solución con el mínimo costo posible (puede haber más de una solución de este tipo). Dada una solución arbitraria a tal problema, la optimización local intenta mejorar esa solución mediante una serie de cambios locales incrementales. Para definir un algoritmo de optimización local, primero se especifica un método para perturbar las soluciones para obtener diferentes. El conjunto de soluciones que se pueden obtener en uno de esos pasos a partir de una solución dada A se llama vecindario de A . El algoritmo luego realiza el ciclo simple que se muestra en la Figura 2.4, con los métodos específicos para elegir S y S' como detalles de implementación.

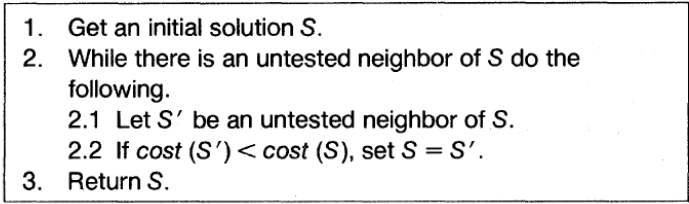
- 
- ```
graph TD; 1[1. Get an initial solution S.] --> 2[2. While there is an untested neighbor of S do the following.]; 2 --> 2.1[2.1 Let S' be an untested neighbor of S.]; 2.1 --> 2.2[2.2 If cost (S') < cost (S), set S = S'.]; 2.2 --> 2; 2 --> 3[3. Return S.];
```
1. Get an initial solution  $S$ .
  2. While there is an untested neighbor of  $S$  do the following.
    - 2.1 Let  $S'$  be an untested neighbor of  $S$ .
    - 2.2 If  $\text{cost}(S') < \text{cost}(S)$ , set  $S = S'$ .
  3. Return  $S$ .

Figura 2.4: Optimización local

Aunque  $S$  no necesita ser una solución óptima cuando finalmente se cierra el ciclo, será localmente óptima ya que ninguno de sus vecinos tiene un costo menor. La esperanza es que localmente óptimo sea lo suficientemente bueno.

La dificultad de la optimización local es que no tiene forma de retirarse

de los óptimos locales poco atractivos. Nunca se pasa a una nueva solución a menos que la dirección sea cuesta abajo, es decir, a un mejor valor de la función de costo. El Recocido Simulado es un enfoque que intenta evitar dicho atrapamiento. Esto se realiza bajo la influencia de un generador de números aleatorios y un parámetro de control llamado temperatura. Como se implementa típicamente, el enfoque de Recocido Simulado involucra un par de ciclos anidados y dos parámetros adicionales, una relación de enfriamiento  $r$ ,  $0 < r < 1$ , y una longitud de temperatura entera  $L$  (ver Figura 2.5). En el Paso 3 del algoritmo, el término congelado se refiere a un estado en el que no parece probable una mejora adicional en el costo ( $S$ ).

```

1. Get an initial solution S .
2. Get an initial temperature $T > 0$.
3. While not yet frozen do the following.
 3.1 Perform the following loop L times.
 3.1.1 Pick a random neighbor S' of S .
 3.1.2 Let $\Delta = \text{cost}(S') - \text{cost}(S)$.
 3.1.3 If $\Delta \leq 0$ (downhill move),
 Set $S = S'$.
 3.1.4 If $\Delta > 0$ (uphill move),
 Set $S = S'$ with probability $e^{-\Delta/T}$.
 3.2 Set $T = rT$ (reduce temperature).
4. Return S .

```

Figura 2.5: RecocidoSimulado

$e^{-\Delta/T}$  será un número en el intervalo  $(0, 1)$  donde  $\Delta$  y  $T$  son positivos. La probabilidad de que un movimiento cuesta arriba de tamaño  $\Delta$  sea aceptado disminuye proporcionalmente a la temperatura y, para una temperatura fija  $T$ , los movimientos ascendentes pequeños tienen mayores probabilidades de aceptación que los grandes. Este método particular de operación está motivado por una analogía física.

### 2.1.3. Algoritmos Genéticos

Un Algoritmo Genético [6] consiste en un conjunto de soluciones codificadas, que hacen una analogía con los cromosomas. Cada uno de estos, tendrá asociado un ajuste o valor de bondad, que expresa una medida su valor como solución al problema. En función de este valor se le darán más o menos oportunidades de “reproducción”. John Holland, investigador de la Universidad de Michigan, es uno de los principales precursores del desarrollo de los

Algoritmos Genéticos, sus trabajos a finales de la década de los 60 mostraron una técnica que imitaba en su funcionamiento a la selección natural.

La reproducción en estos algoritmos pueden darse de dos formas:

- Cruce: Se genera una descendencia a partir del mismo número de individuos (generalmente 2) de la generación anterior.
- Copia: Un determinado número de individuos pasa sin sufrir ninguna variación directamente a la siguiente generación.

La Figura 2.6 muestra el funcionamiento de un algoritmo genético

```

Inicializar población actual aleatoriamente
MIENTRAS no se cumpla el criterio de terminación
 crear población temporal vacía
 SI elitismo: copiar en población temporal mejores individuos
 MIENTRAS población temporal no llena
 seleccionar padres
 cruzar padres con probabilidad P_c
 SI se ha producido el cruce
 mutar uno de los descendientes (prob. P_m)
 evaluar descendientes
 añadir descendientes a la población temporal
 SINO
 añadir padres a la población temporal
 FIN SI
 FIN MIENTRAS
 aumentar contador generaciones
 establecer como nueva población actual la población temporal
FIN MIENTRAS

```

Figura 2.6: Algoritmo Genético

Algunos de los criterios de parada pudieran ser:

- Se ha alcanzado una población con individuos lo suficientemente buenos para darle solución al problema.
- Ha convergido la población, lo cual quiere decir que la media de bondad de la misma se aproxima a la bondad del mejor individuo.
- Se ha alcanzado el número de generaciones (iteraciones) especificado

A es algoritmo se le han definido numerosas variantes. Una de las más extendidas es aplicar los operadores genéticos de cruce y mutación directamente sobre la población genética. Pero en el caso de que se aplique cruce, no se puede insertar directamente la descendencia en la población, debido a que número de individuos de la población se ha de mantener constante. Es

decir, para permitir a los descendientes generados incorporarse a la población, se han de eliminar otros individuos. Trabajando con una sola población no se puede definir que la misma está llena, pues el número de sus miembros es constante. En este caso se pasará a la siguiente población cuando se hayan alcanzado un número determinado de cruzamientos especificados por el usuario, que deberán estar acordes al tamaño de la población.

En el esquema general, solo los descendientes originados a partir de un cruce son mutados. Otra opción habitual es la selección aleatoria del individuo a mutar entre todos los que forman parte de la población.





# Bibliografía

- [1] B. J. Copeland, *The Essential Turing*. Oxford University Press, 2004.
- [2] B. Kontsevoi, “The ultimate way to effective software evaluation,” *Unknown*, 2018.
- [3] J. P. García, “Optimizacioncombinatoria,” *Universidad Politécnica de Valencia*, 2006.
- [4] T. Feo and M. Resende, “Greedy randomized adaptive search procedures,” *Journal of Global Optimization*, vol. 6, pp. 109–133, 03 1995.
- [5] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, “Optimization by simulated annealing: An experimental evaluation; part i, graph partitioning,” *Operations research*, vol. 37, no. 6, pp. 865–892, 1989.
- [6] J. P. A. Marcos; Rivero Gestal (Daniel; Rabuñal, Juan Ramón; Dorado and M. Gestal, *Introducción a los algoritmos genéticos y la programación genética*. Universidade da Coruña, 2010.