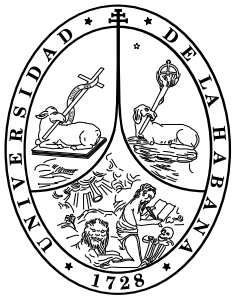
Universidad de La Habana
Facultad de Matemática y Computación
Departamento de Programación



Título

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

Autor: **Marcel Ernesto Sánchez Aguilar**
Tutores: **Ludwig Leonard Méndez**, **Carlos Fleitas**

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

Glosario

- Aplicación: Sistema computacional que tiene como objetivo servir de herramienta a uno o varios usuarios en la realización de diversas tareas.
- Aplicación de Consola: Es aquella que se ejecuta en una ventana mediante líneas de comandos.
- Software: Conjunto de programas y rutinas que permiten a la computadora realizar determinadas funciones.
- 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 cómputos, 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, se realizan ahora mediante un equipo de cómputo. Pero, ¿qué tan buena puede ser una aplicación y cómo expresamos la calidad de las mismas? El presente trabajo se enfoca en dar solución a un subconjunto de esta problemática: evaluar el correcto funcionamiento de un algoritmo presente en una aplicación.

En la actualidad existen numerosas herramientas que de alguna manera permiten expresar una métrica en cuanto a evaluación de un código. En lo adelante se hará mención de algunas de las mismas.

SonarQube es una plataforma desarrollada en Java que permite realizar análisis de la calidad de código de forma automatizada, para lo cual se usan diversas herramientas de análisis estático de código fuente como Checkstyle, PMD o FindBugs que permiten obtener métricas que ayudan a mejorar la calidad. Resulta una herramienta de utilidad en durante el testing de código en una aplicación.

Beyond Compare es una herramienta que permite comparar ficheros. Si

al examinar 2 códigos, resulta que tienen una estructura similar, es probable que que hayan seguido la misma idea pero con diferente ejecución para la solución de un problema.

HP Quality Center es una herramienta de gestión de pruebas que tiene como objetivo el control de la calidad software. Entre los mismo incluye la gestión de requisitos, gestión de pruebas y los procesos de negocio.

La aceptación de cualquier software pasa siempre por la lógica detrás del código. Esto es fundamental para el buen funcionamiento de cualquier programa. Un correcto diseño de la lógica del código puntúa de manera favorable el criterio de calidad. Siendo este nuestro objetivo de evaluación, quedaría por designar las métricas a utilizar para dicho fin. ¿Cómo evaluar la aplicación? ¿En base a cuáles aspectos se mide la calidad de estos sistemas de cómputo?

1.1. Estado del Arte

1.2. Contexto del problema

En la Facultad de Matemática y Computación de la Universidad de La Habana, los estudiantes de primer año que cursan la asignatura de Programación de la carrera Ciencia de la Computación se enfrentan a tres evaluaciones parciales y a una prueba final. Estos exámenes tienen la peculiaridad de que se realizan a través de un equipo de cómputo. Una vez entregada las soluciones digitales (código) de los estudiantes al problema propuesto, se procede a evaluar las implementaciones de los mismos. El proceso de calificación de un examen en ninguna asignatura es trivial, por ejemplo, muchas veces lo que separa un "buen"3 (a veces denominado 3+) de un "mal"4 (o 4−) es la subjetividad del evaluador. Esto es algo extremadamente difícil de definir formalmente , pues esa subjetividad depende de los objetivos a vencer las asignaturas y de experiencias en la enseñanza.

Otro aspecto a señalar en estas pruebas particulares, es que la evaluación no se realiza directamente sobre los códigos de los estudiantes. Profesores de la asignatura se empeñan en confeccionar casos de prueba representativos que puedan dar una medida de la correctitud del algoritmo propuesto. La anterior tarea puede resultar engorrosa, diseñar esos casos de prueba puede llegar a ser incluso más difícil que la solución al problema, pues es necesario cubrir la mayor cantidad de aspectos a evaluar sin incidir de más en algún rasgo para no afectar la calificación.

Entre los principales aspectos a evaluar están la correctitud del código,

el uso adecuado de los recursos del lenguaje de programación y el manejo eficiente del tiempo y la memoria (en términos computacionales).

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 de los mismos que permita utilizarlos como medida de calidad para la evaluación de algoritmos. El objeto de investigación es la optimización combinatoria abordado por meta-heurísticas [2] 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.

Esta solución sería extensible a cualquier algoritmo de programación. Existe abstracción sobre los algoritmos a evaluar, pero se diseñó particularmente para los tipos de exámenes que se aplican en primer año. Con solo tener en cuenta otros factores como el tiempo de ejecución y la memoria consumida, se podría ampliar el marco de algoritmos evaluables por esta propuesta.

1.4. Ejemplos

1.4.1. Funcionamiento General

Consideremos el algoritmo de Euclides para hallar el máximo común divisor entre dos números.

Definición. 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 en la fase de aprendizaje 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.

1.4.2. Funcionamiento Detallado

Consideremos el problema de la multiplicación de polinomios. Algunos aspectos a destacar serían los siguientes.

- La entrada al algoritmo serían 2 polinomios y su salida el resultado de su multiplicación.
- Se pasa como entrada a la propuesta un generador de polinomios y 3 respuestas al problema (implementaciones) asociadas a una nota (5, 4 o 3), además de los porcentajes de acierto esperados por cada solución. (5:100 %, 4:90 % y 3:75 % por ejemplo).
- La función objetivo será minimizar la sumatoria de los errores, donde error es el valor absoluto entre lo que se espera y lo obtenido en términos del porcentaje para cada nota.

Nuestra propuesta deberá generar n casos aleatorios con ayuda del generador y reducirlos a un $k <= n$ representativo que se ajuste a los porcentajes esperados por cada solución. Para ello se aplicarán algoritmos metaheurísticos que se describen posteriormente.

Capítulo 2

Marco Teórico

Los problemas de optimización combinatoria que involucran 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 no es factible 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 la 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

Definición. Heurística: técnica o método inteligente, para realizar una tarea que no es producto de un riguroso análisis formal, sino del conocimiento experto sobre un tema a solucionar, la cual aporta soluciones con cierto grado de confianza y calidad.

Definición. Método Heurístico: parte práctica del concepto de heurística. Es un enfoque para la resolución de problemas, aprendizaje o descubrimiento que emplea un método práctico no garantizado para ser óptimo o perfecto, pero suficiente para los objetivos inmediatos.

Definición. Metaheurística: son una clase de métodos aproximados que están diseñados para resolver problemas difíciles de optimización combinatoria, en los que los heurísticos clásicos no son efectivos. Las metaheurísticas proporcionan un marco general para crear nuevos algoritmos híbridos combinando diferentes conceptos derivados de la inteligencia artificial, la evolución biológica y los procedimientos estadísticos.

Las metaheurísticas generalmente se aplican cuando se desconoce de un algoritmo que resuelva de manera satisfactoria dichos problemas. Muchas veces lo anterior se debe a que no es factible explorar en su totalidad el campo de soluciones en busca de un óptimo global, por lo que se emplean heurísticas o métodos aproximados. Los problemas de optimización combinatoria son usuales escenarios en la aplicación de una metaheurística.

La optimización combinatoria se basa en encontrar una configuración de bits (selección), respecto a la entrada del problema, que maximice o minimice una función objetivo específica. A los pasos intermedios anteriores a la solución se les denomina estados, y al conjunto de todos los estados candidatos se le llama espacio de búsqueda. La función objetivo, los estados y el espacio de búsqueda son definidos en función del problema.

Existen metaheurísticas que mantienen un único estado actual durante cada instante de ejecución, el cual es actualizado en cada iteración. Este paso se conoce como función de transición. Otras metaheurísticas más sofisticadas mantienen, en vez de un único estado actual, un conjunto de estados candidatos. Así, la función de transición añade o elimina estados de este conjunto. Otros procedimientos pueden guardar información del óptimo actual, escogiendo el estado óptimo entre todos los óptimos locales obtenidos en varias

etapas del algoritmo.

Como se hizo mención anteriormente, el espacio de búsqueda puede resultar extremadamente grande o incluso infinito, por lo cual es necesario definir algunos criterios de parada para la ejecución del algoritmo. Entre los más usuales podemos encontrar el efectuar un número de iteraciones especificadas por el usuario, el alcance de un determinado tiempo de ejecución o el cumplimiento de una condición específica del problema.

Existen muchos métodos heurísticos con comportamientos y objetivos diferentes, por lo que resulta complicado clasificarlos. Esto se debe en parte a que muchos de ellos han sido diseñados para un problema específico sin posibilidad de generalización o aplicación a otros problemas similares. Sin embargo, algunas situaciones pueden resultar tener un parecido en cuanto al tipo de idea a seguir para su solución, de ahí surge una especie de clasificación entre estos algoritmos.

- Métodos de Descomposición: son aquellos aplicables a problemas que se descomponen en varios subproblemas más sencillos de resolver que el original.
- Métodos Inductivos: la idea es generalizar versiones pequeñas o más sencillas al caso completo. Propiedades o técnicas identificadas en estos casos más fáciles de analizar pueden ser aplicadas al problema completo.
- Métodos de Reducción: consiste en identificar propiedades que cumplen principalmente las buenas soluciones e introducirlas como restricciones del problema. El objetivo es restringir el espacio de soluciones al simplificar el problema. Se corre el riesgo de dejar fuera soluciones óptimas del problema original.
- Métodos Constructivos: se caracterizan por construir en cada paso una solución del problema. Usualmente son métodos deterministas y suelen estar basados en la mejor elección en cada iteración.
- Métodos de Búsqueda Local: los procedimientos de búsqueda o mejora local comienzan con una solución del problema y la mejoran progresivamente. El procedimiento realiza en cada paso un movimiento de una solución a otra con mejor valor. El método finaliza cuando, para una solución, no existe ninguna solución accesible que la mejore.

Los métodos constructivos y los de búsqueda local resaltan entre los procedimientos metaheurísticos más empleados y con mejores resultados en la práctica, por lo que para la confección de nuestra propuesta fueron seleccionados algunos de los que resultan aplicables a nuestro problema.

2.1.1. Greedy Randomized Adaptive Search Procedures

GRASP [3] 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 →
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.

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 la vecindad, técnicas eficientes de búsqueda y la solución inicial.

A continuación, la Figura 2.3 muestra dicho procedimiento.

```
procedure ConstructGreedyRandomizedSolution(Solution)
1    Solution = {};
2    for Solution construction not done →
3        MakeRCL(RCL);
4        s = SelectElementAtRandom(RCL);
5        Solution = Solution ∪ {s};
6        AdaptGreedyFunction(s);
7    rof;
end ConstructGreedyRandomizedSolution;
```

Figura 2.2: Construcción de la solución GRASP

```
procedure local(P,N(P),s)
1    for s not locally optimal →
2        Find a better solution t ∈ 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 [4] (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 otras. El conjunto de soluciones que se pueden obtener en uno de esos pasos a partir de una solución dada A se llama vecindad 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.

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 *cost* $(S') <$ *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).

$e^{-\Delta/T}$ será un número en el intervalo $(0, 1)$ donde Δ y T son positivos. La

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 $\Delta = cost\ (S') - cost\ (S)$.
     - 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

probabilidad de que un movimiento cuesta arriba de tamaño Δ 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 [5] 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 de 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 puede 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
Algunos de los criterios de parada pudieran ser:

```
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 Pc
        SI se ha producido el cruce
                mutar uno de los descendientes (prob. Pm)
                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

- 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 este algoritmo se le han definido numerosas variantes. Una de las más extendidas es aplicar los operadores genéticos de cruce y copia 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 el 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.

Capítulo 3

Diseño de la Propuesta

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. Pero pudiera haberse utilizado cualquier lenguaje de propósito general. Para poder aplicar alguna metahehurística al problema que se intenta dar solución, es necesario definir nuestro espacio de búsqueda, así como las entradas, salidas y adaptaciones hechas a las implementaciones.

Se provee al sistema de un conjunto inicial de casos de prueba brindados por el generador, los cuales inicialmente constituyen una solución, pues nuestro problema de optimización no tiene restricciones, solo una función objetivo $f(x, y)$ a minimizar:

$f(x, y) = |x - x_0| + |y - y_0|$

- x: Porcentaje de acierto obtenido por la solución con nota 3
- x_0: Porcentaje de acierto esperado de la solución con nota 3
- y: Porcentaje de acierto obtenido por la solución con nota 4
- y_0: Porcentaje de acierto esperado de la solución con nota 4

La solución que brinde el algoritmo debe consistir en un subconjunto del conjunto inicial de casos de prueba, que debería tener una evaluación menor o igual que la inicial. Se ha de aclarar que la salida puede estar sesgada por la entrada, es decir, que el conjunto que se reciba como entrada no sea una muestra representativa de la población de casos de prueba.

A continuación se explican las adaptaciones hechas a los dos algoritmos implementados: GRASP y Genético, los cuales reciben la misma entrada y aplican un proceder diferente.

3.1. Adaptación de GRASP

Como mencionábamos en (hacer referencia a la subsección), GRASP en cada iteración hace una búsqueda local, la cual brinda otra solución factible al algoritmo. En nuestro problema en particular, se tienen inicialmente los n casos de prueba, de los cuales se selecciona un subconjunto de tamaño $k, k \leq n$ que será la lista de candidatos de eliminar, los cuales representan una mejora (menor evaluación) de la función objetivo si no se les considerara en la solución final del problema. Una vez conformada dicha selección se toma uno de la misma al azar y se retira, posteriormente se hace otra iteración del algoritmo pero ahora con un conjunto de tamaño $n - 1$.

Como casos de parada tenemos los siguientes:

- Se ha alcanzado el valor esperado de la función objetivo.
- Se han cumplido un número de iteraciones especificadas.
- La lista de candidatos a extraer es vacía.

Para un mayor rendimiento de este algoritmo, se hacen k pasadas con el conjunto inicial de los n casos de prueba brindados por el generador, pues en la aleatoriedad de seleccionar el candidato a eliminar, puede que no siempre se esté caminando hacia el óptimo global. Por lo que varias corridas lanzarán un mejor rendimiento del algoritmo, el cual se quedará con la mejor de todas. La estimación del parámetro k se hace de forma experimental y es posible que dos pasadas brinden la misma solución, pero esto último es muy poco probable.

3.2. Adaptación del Genético

En (hacer referencia a la subsección), el Algoritmo Genético define varios operadores genéticos a efectuar sobre una población inicial de individuos, que serán métodos que reciben dos soluciones al problema. La entrada al procedimiento será el mismo subconjunto de tamaño n que recibe GRASP. La diferencia está en que este último solo da pasos de tamaño 1 en cada iteración, pues se visita una solución vecina que difiere en solo un caso (1 bit si se ven como máscaras booleanas sobre un conjunto).

La población inicial del algoritmo será de los n casos iniciales, $k, k > 4$ configuraciones diferentes de bits sobre esos n. Una vez conformada, se procede a la selección de 4 individuos que pudieran ser representativos.

- individuo o_1: el que mejor evalúa la función objetivo.

- individuo o_2: el segundo que mejor evalúa la función objetivo.
- individuos r_1, r_2: dos seleccionados de forma aleatoria.

Operadores genéticos definidos donde $i1, i2$ son individuos de la población, o sea, soluciones al problema.

- operador $op_1(i_1, i_2)$: mezcla la primera mitad de i_1 con la segunda mitad de i_2.
- operador $op_2(i_1, i_2)$: mezcla de forma alternada i_1 e i_2.
- operador $op_3(i_1, i_2)$: unión de miembros aleatorios de i_1 e i_2.
- operador $op_4(i_1, i_2, p)$: mezcla el $p\,\%$ de i_1 con el $(1-p)\,\%$ de i_2.

Una vez definida la población inicial y los operadores genéticos, se procede a aplicar estos últimos sobre los 4 individuos de la siguiente forma. Dados o_1 y o_2 se le pasan como parámetros cada operador genético, similar con r_1 y r_2. De esta forma se obtienen 8 nuevos individuos, de los cuales se seleccionan los 4 mejores y se añaden a la población. Esto se repite un número determinado de iteraciones y se devuelve el mejor individuo de la población, el cual minimiza la evaluación de la función objetivo.

Note que en cada iteración del algoritmo, se aumenta en 4 el número de soluciones, por lo que debe estar regulado el número de las mismas.

De esta forma se intenta cubrir varios caminos que nos puedan llevar al óptimo y se dan pasos de más de tamaño 1 (que difieran en más de un bit dos soluciones). Con un número de iteraciones lo suficientemente grande, este algoritmo debe acercarse a una buena solución del problema dado el balance entre exploración y explotación que posee.

Falta responder: ¿Cómo estos operadores genéticos resuelven el problema del estancamiento local?

Capítulo 4

Resultados Obtenidos

Los resultados en cuanto a la eficacia de los algoritmos en reducir la evaluación de la función objetivo, fueron analizados en base a un mismo conjunto de casos de prueba en varias ejecuciones del algoritmo, el cual fue pasado como entrada en varias corridas de los mismos. Dicho conjunto fue generado de manera aleatoria.

A continuación se muestran algunos detalles de 3 corridas, los casos están enumerados del 1 a n. Dada la aleatoriedad inherente a la metaheurística empleada, quedan distintos casos de prueba en el conjunto final. Sin embargo cada ejecución tiene los mismos valores finales en los indicadores que se expresan en la siguiente tabla:

Cantidad de casos iniciales	250
Porcentaje inicial/esperado del 3	19.60/30
Porcentaje inicial/esperado del 4	95.20/90
Evaluación inicial de la f.o.	15.60
Iteraciones del algoritmo	87
Cantidad de casos finales	163
Porcentaje final/esperado del 3	30.06/30
Porcentaje final/esperado del 4	92.63/90
Evaluación final de la f.o.	2.699

Cuadro 4.1: Indicadores iniciales y finales de las corridas 1, 2 y 3

Veamos otros aspectos a destacar para comprender por qué sucede esto. En la tabla 4.2 se muestra la evaluación de la función objetivo en cada una de las 3 corridas de GRASP con dicho conjunto de casos de prueba.

Iteración	f.o.
0	15.600
1	15.502
2	15.403
...	...
43	10.53
44	10.39
45	10.24
...	...
85	3.030
86	2.800
87	2.699

Cuadro 4.2: Valor de la función objetivo según las iteraciones

En cada una de las 3 ejecuciones de GRASP se hicieron exactamente 87 iteraciones. Como resultado se eliminaron conjuntos de casos disjuntos, pero la evaluación de la función objetivo en cada paso se redujo de manera idéntica. ¿Por qué?

Haciendo un análisis más profundo, dadas las peculiaridades del problema a resolver, podemos calificar en 3 grupos los casos de prueba:

- tipo *uno*: son de la forma $(0, 0)$, donde ninguna implementación acierta el caso
- tipo *dos*: son de la forma $(0, 1)$, donde solo acierta la implementación del 4.
- tipo *tres*: son de la forma $(1, 1)$, donde ambas implementaciones aciertan el caso
- tipo *cuatro*: son de la forma $(1, 0)$ donde solo acierta la implementación del 3.

Nota: Por lo general no existen los casos de la forma $(1, 0)$. En la práctica puede que surjan de manera muy aislada. Se supone que una implementación de 4 debería acertar en los mismos casos que una de 3 y más, que no tenga un aspecto en el que quede por detrás de otra que obtenga una nota inferior. Sin embargo es posible que no se provea al sistema de una implementación de 4 puntos, por lo que los casos de tipo *cuatro* serían los únicos presentes.

Tipo de caso	Cantidad Inicial	Cantidad Final
uno	12	12
dos	189	102
tres	49	49

Cuadro 4.3: Tipos de casos presentes en las corridas 1, 2 y 3

Una vez fijado el conjunto inicial de casos de prueba, ya la función objetivo toma un valor (ver definición de la misma en el Capítulo 3). Supongamos que el porciento de acierto de la implementación de 3 es 75 % de un 80 % esperado y la de 4 es 90 % de un 95 % esperado. El algoritmo debe reducir en alguna medida los casos en que fallan ambas implementaciones, por lo cual la lista de candidatos a eliminar definida por GRASP en cada iteración serán solamente los casos de la forma *cero*. Por tanto de esa lista, que no va a variar en cuanto a grupos de casos, pues el tipo de un caso es invariable, se seleccionarán tantos casos a eliminar como la evaluación de la función objetivo lo permita (mientras mejore) y según la cantidad de casos del tipo objetivo presentes en el conjunto inicial.

Por lo cual no importa cuál sea el conjunto inicial de casos de prueba, una vez fijado, siempre se se obtendrá un subconjunto final distinto en cuanto a casos de prueba pero con iguales evaluación de la función objetivo y cantidad de casos por tipo. En el cuadro 4.3 se aprecia lo anteriormente expuesto.

Como se ha mencionado, los casos de prueba se generan de forma aleatoria, por lo que desde ese inicio se puede estar comprometiendo el buen funcionamiento del algoritmo. Recibir como entrada una muestra donde esté representado el universo de casos de prueba permite una mejor evaluación de la función objetivo, pues el algoritmo es capaz de desestimar aquellos casos que no son iguales desde el punto de vista físico pero sí a nivel de evaluación (combinación lineal uno de otro). Tener toda la gama de casos de prueba le brinda la posibilidad de hacer ajustes para poder minimizar la función objetivo de manera óptima. De lo contrario podría estar condenada (que no se minimice todo lo posible) desde el principio. Al no poder garantizar esto, es posible que el algoritmo no pueda encontrar la solución óptima al problema si no pertenece al subconjunto que se recibe como entrada.

Sin embargo hay otros aspectos negativos inherentes al problema que afectan el desempeño del algoritmo. Por ejemplo, la información que se tiene de un caso de prueba es muy poca, solo tenemos información de en cuáles implementaciones acertó, no existe un criterio que permita comparar 2 casos

de prueba en cuanto a ser significativo para una implementación u otra.

Una idea pudiera ser tener varias implementaciones de nota, en vez de recibir solo una de 3 y una de 4. Con ello se podría tener más información, pues no necesariamente un caso de prueba tiene que acertarle a la totalidad del conjunto de implementaciones de 3 o de 4. Así se podría hacer algún tipo de ajuste, tener otra métrica para que el conjunto final de casos de prueba sea más heterogéneo. El inconveniente de esta propuesta es que se vería forzado el evaluador de proveer al sistema de dichas implementaciones extras.

Respecto a la idea del párrafo anterior, para que el evaluador no tenga que idear varias implementaciones, se podría tener una métrica en cuanto a dos implementaciones. Por ejemplo, asumiendo el porciento de acierto de las implementaciones como métrica, si la solución de un estudiante obtiene un resultado cercano al esperado por el evaluador, la misma se podría incorporar a lista de implementaciones de referencia. Pero esto no resulta siempre posible, depende de las características del problema que se pueda generar diferentes implementaciones de iguales en resultados.

Bibliografía

[1] B. J. Copeland, *The Essential Turing*. Oxford University Press, 2004.

[2] J. P. García, "Optimizacioncombinatoria," *Universidad Politécnica de Valencia*, 2006.

[3] T. Feo and M. Resende, "Greedy randomized adaptive search procedures," *Journal of Global Optimization*, vol. 6, pp. 109–133, 03 1995.

[4] 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.

[5] 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.