ALGORITMO PARA LA CONSTRUCCIÓN DE ARREGLOS DE CUBRIMIENTO BASADO EN LA MEJOR BÚSQUEDA ARMÓNICA GLOBAL Y TÉCNICAS DE OPTIMIZACIÓN LOCAL

JIMENA ADRIANA TIMANÁ PEÑA

Tesis de Maestría en Computación

Director: PhD. Carlos Alberto Cobos Lozada
Co-Director: PhD. Martha Eliana Mendoza Becerra

Asesor: PhD. José Torres Jiménez
(CINVESTAV, Tamaulipas, México)

Universidad del Cauca
Facultad de Ingeniería Electrónica y Telecomunicaciones
Departamento de Sistemas
Grupo de I+D en Tecnologías de la Información (GTI)
Línea de Investigación: Sistemas Inteligentes e Ingeniería de Software
Popayán, Noviembre de 2017

JIMENA ADRIANA TIMANÁ PEÑA

ALGORITMO PARA LA CONSTRUCCIÓN DE ARREGLOS DE CUBRIMIENTO BASADO EN LA MEJOR BÚSQUEDA ARMÓNICA GLOBAL Y TÉCNICAS DE OPTIMIZACIÓN LOCAL

Tesis presentada a la Facultad de Ingeniería Electrónica y Telecomunicaciones de la Universidad del Cauca para la obtención del Título de

Magíster en Computación

Director
Director: Ph.D. Carlos Alberto Cobos Lozada
Co-Director: Ph.D. Martha Eliana Mendoza Becerra

Asesor: Ph.D. José Torres Jiménez
(CINVESTAV, Tamaulipas, México)

Popayán
2017

Dedicatoria

Al Padre Celestial por brindarme una nueva oportunidad de estudio y poderlo culminar satisfactoriamente.

A mis padres por su apoyo incondicional, por su constante motivación e interés en mi crecimiento personal y profesional.

A Lisandro y a Matías por su amor, comprensión y paciencia.

A mi hermano, sobrinos, familia y amigos que siempre estuvieron pendientes de mí y del proceso vivido durante la maestría.

Agradecimientos

Al Doctor Carlos Alberto Cobos Lozada, por su excelente y sabia dirección, por su ayuda incondicional en todo momento, por su increíble comprensión y paciencia y sobre todo por su invaluable amistad.

A la Doctora Martha Eliana Mendoza Becerra, por su colaboración en la co-dirección de este trabajo de grado.

Al Doctor José Torres Jiménez y al Centro de Investigación y Estudios Avanzados del Instituto Politécnico Nacional (CINVESTAV – Tamaulipas, México) por su asesoramiento y apoyo constante durante el desarrollo de la Maestría y la pasantía de investigación.

A los docentes y compañeros de la Maestría en Computación por todo lo enseñado, por todo lo vivido.

Resumen Estructurado

Los Covering Arrays son objetos matemáticos que han sido ampliamente utilizados dentro del diseño de experimentos en diversos sectores y áreas de aplicación como agricultura, medicina, biología, diseño de materiales y manufactura, y más recientemente para realizar pruebas funcionales de componentes software ya que permiten probar todas las interacciones de un tamaño determinado de los parámetros de entrada de un procedimiento, función o unidad lógica en general, utilizando el mínimo número de casos de prueba.

La construcción de un CA es una tarea compleja que conlleva un tiempo de ejecución elevado e involucra una alta carga computacional. El objetivo de los métodos y algoritmos para construir Covering Arrays es satisfacer las propiedades de cobertura con el menor número posible de filas, buscando de esta forma CAs óptimos.

A la fecha se han realizado investigaciones sobre diferentes métodos eficaces para construir CAs, entre ellos: métodos exactos, métodos algebraicos, métodos Greedy y métodos basados en meta-heurísticas. Estos últimos han reportado a la fecha los mejores resultados.

Como resultado de la presente investigación se diseñó e implementó un algoritmo híbrido que haciendo uso de una adaptación del algoritmo meta-heurístico de la Mejor Búsqueda Armónica Global como técnica de optimización global y del algoritmo de Recocido Simulado como técnica de optimización local, construye Covering Arrays uniformes y mixtos con diferentes niveles de fuerza. Este algoritmo denominado GHSSA logró generar resultados competitivos frente a los mejores resultados (cotas) reportados por otros algoritmos del estado del arte.

Palabras claves: Covering Array, Meta-heurísticas, Mejor Búsqueda Armónica Global, Recocido Simulado.

Structured Abstract

Covering Arrays are mathematical objects which have been broadly implemented in the design of experiments in different areas and purposes such as agriculture, medicine, biology, material design and manufacture. More recently, they have been used to perform software component tests, since they allow for the testing of all the interactions in a fixed-sized block of the input parameters in a procedure, function or logic unit by using the minimum amount of test cases.

Building a CA is a complex task which takes a long execution time and involves a heavy computational load. The objective of the methods and algorithms to build Covering Arrays is to fulfill the coverage properties by using the minimum possible rows, achieving more efficient CAs.

To date, research about the different efficient methods to build CAs has been conducted, namely: exact methods, algebraic methods, Greedy methods, and metaheuristic based methods, among others. The latter have reported the best outcomes this far.

As a result of this research, a hybrid algorithm was designed and implemented in order to build uniform and mixed Covering Arrays with different power values. It uses an adaptation of the meta-heuristic Global Best Harmony Search method as a global optimization technique, and the Simulated Annealing algorithm for local optimization. This algorithm, referred to as GHSSA, achieved competitive results when compared to the best results reported by other state of the art algorithms.

Keywords**:** Covering Array, Metaheuristic, Global Best Harmony Search, Simulated Annealing.

Tabla de Contenido

Lista de Figuras

Lista de Tablas

Lista de Ecuaciones

Capítulo 1

Introducción

Desde hace varias décadas, las pruebas de software son el método más ampliamente usado para asegurar la calidad del software [1]. Las pruebas pueden verse como un conjunto de procesos que se diseñan, desarrollan y aplican a lo largo de la vida del desarrollo del software, para detectar y corregir defectos y fallas encontrados durante y después de que el código es implementado y de esta manera, elevar la calidad y fiabilidad del programa entregado al usuario final [2].

Sin embargo, aplicar pruebas de software exhaustivas que permitan probar el componente software en su totalidad, además de ser una tarea costosa en dinero y que consume demasiado tiempo, es una tarea prácticamente imposible. Inclusive, tratar de probar el software más simple, puede llegar a generar un número elevado de casos de pruebas debido a las múltiples posibilidades de combinaciones en sus parámetros de entrada. La creación de casos de prueba para todas esas posibilidades es un proceso poco factible y poco práctico [1, 2].

Por ejemplo, si se desea probar un componente software que recibe como parámetros de entrada los valores de 5 sensores y cada sensor puede tomar 10 valores diferentes, serían requeridas entonces, 10^5 o 100.000 casos de pruebas diferentes para probar de forma exhaustiva dicho componente.

Como propuesta alternativa a las pruebas exhaustivas se encuentran las pruebas combinatorias, las cuales, seleccionan casos de prueba con un mecanismo de muestreo, que sistemáticamente cubre las combinaciones de valores de los parámetros de entrada, definiendo un conjunto de pruebas más pequeño, que es relativamente fácil de administrar y ejecutar [3]. Este tipo de pruebas reduce los costos y aumenta significativamente la efectividad de las pruebas de software. Sin embargo, el desafío de las pruebas combinatorias radica precisamente en encontrar el mínimo número de casos de prueba, los cuales logren la máxima cobertura con la mínima cardinalidad [4, 5].

Uno de los objetos combinatorios que satisfacen el criterio de cobertura para este tipo de situación son los Arreglos de Cobertura o Covering Arrays (CA) [5].
Los Arreglos de Cobertura son objetos matemáticos que han sido ampliamente utilizados dentro del diseño de experimentos en diversos sectores y áreas de aplicación como agricultura, medicina, biología, diseño de materiales y manufactura, ya que cuentan con las propiedades necesarias para definir óptimamente una serie de experimentaciones, debido a su naturaleza combinatoria [5].

Con aplicación más reciente en el área de pruebas de software, los CA pueden ser usados y aplicados para realizar pruebas funcionales de software. Una de las principales aplicaciones de estos objetos, es generar el menor número de conjuntos o casos de pruebas de software para cubrir todos los conjuntos de interacciones de los parámetros de entrada de un procedimiento, función o unidad lógica de software [5].

1.1 Definición del Problema

Construir CAs óptimos (CAs con el menor tamaño de filas posible) es una tarea compleja que involucra una alta carga computacional. Para generar CAs de tamaño considerable, generalmente se requiere el uso de muchos procesadores en paralelo durante varios meses, pero el poder lograr su definición, tiene un impacto muy significativo en los procesos que se adelantan durante las pruebas de software ya que al definir conjuntos de pruebas más pequeños aumenta significativamente la efectividad de las mismas, reduce tiempos de aplicación y costos. [6]. Debido a la importancia de los CAs, hasta la fecha se han desarrollado varias investigaciones sobre el desarrollo de algoritmos y métodos eficaces para construirlos.

Entre los métodos más significativos reportados en la literatura se encuentran: métodos exactos, métodos algebraicos, métodos Greedy y métodos basados en meta-heurísticas [7].

A la fecha, los métodos basados en meta-heurísticas son los que han proporcionado mejores resultados para la construcción de CAs. Entre los algoritmos meta-heurísticos más destacados se encuentran: Recocido Simulado, Búsqueda Tabú, algoritmos genéticos y algoritmos de colonias de hormigas [5]. La mayoría de estos algoritmos debe su efectividad a que hacen uso adicional de técnicas de optimización local que potencializan su operación. Entre las técnicas de

optimización local más empleadas se incluyen búsqueda en vecindario variable, búsqueda local guiada e iterada, entre otras.

A pesar de que existen otros algoritmos meta-heurísticos que han reportado excelentes resultados en la resolución de problemas complejos en distintas áreas del conocimiento, como por ejemplo el algoritmo Global Best Harmony Search (GHS), no han sido explorados en la construcción de CA. Teniendo en cuenta lo anterior, y que el algoritmo GBHS reporta excelentes resultados en la solución de problemas continuos y discretos en diferentes entornos, en la presente investigación se busca dar respuesta a la siguiente pregunta: ¿Es posible crear Covering Arrays óptimos usando un algoritmo basado en la mejor búsqueda armónica global y técnicas de optimización local?

1.2 Justificación

Estudios realizados por el National Institute for Standards and Technology (NIST) estiman que el costo de las pruebas inadecuadas de software afectan la economía de los EE.UU. en el rango de los 22,2 hasta 59,5 billones de dólares por año, a pesar de la asignación de importantes recursos de los proyecto de software para el diseño y aplicación de las pruebas [8]. Estiman, además, que aproximadamente 22 billones de dólares podrían ser reducidos mediante la realización de pruebas más efectivas.

Desde el punto de vista de investigación, con la definición e implementación del algoritmo propuesto se busca aportar nuevo conocimiento en el área de la computación inteligente, ya que se realizará el modelado de un nuevo algoritmo de optimización discreta basado en la Mejor Búsqueda Armónica Global y estrategias de optimización local que permita la creación de CA óptimos.

La construcción de nuevos CA óptimos con el algoritmo propuesto podrá impactar potencialmente en el área de calidad de software específicamente en escenarios de aplicación como lo son las pruebas de software ya que se generarán casos de prueba de menor tamaño que sistemáticamente cubrirán las combinaciones de valores de parámetros de alguna función o componente software y que serán relativamente más fáciles de administrar y ejecutar.

1.3 Objetivos

A continuación, se presentan los objetivos de la presente tesis como fueron aprobados por el Consejo de Facultad de la Facultad de Ingeniería Electrónica y Telecomunicaciones.

Objetivo General

Proponer un algoritmo para la construcción de arreglos de cubrimiento (CA) uniformes (igual alfabeto para las diferentes variables) y con diferentes niveles de fuerza, basado en la meta-heurística de la Mejor Búsqueda Armónica Global como estrategia de optimización global y diversas estrategias de optimización local.

Objetivos específicos

- Definir el estado del arte sobre métodos globales de optimización de sistemas discretos, su hibridación con estrategias de optimización local en la creación de arreglos de cubrimiento óptimos.
- Proponer un algoritmo que combine la meta-heurística de la mejor búsqueda armónica global y estrategias de optimización local para definir (construir) arreglos de cubrimiento uniformes a diferentes niveles de fuerza.
- Evaluar el comportamiento del algoritmo contra otros del estado del arte, teniendo en cuenta el número de filas definidas para el arreglo (que tan óptimo es el resultado) en comparación con los reportados en el estado del arte, teniendo en cuenta la misma configuración del CA (número de parámetros, alfabeto y fuerza).

1.4 Resultados Obtenidos

- Monografía del trabajo de grado, cuya estructura del documento se detalla en la siguiente sección.
- Código fuente del algoritmo híbrido GHSSA implementado en lenguaje C# y desarrollado en el entorno de desarrollo Microsoft Visual Studio 2015.
- Artículo publicado en una revista indexada categoría A2 según PUBLINDEX de Colciencias con la siguiente referencia: J.Timana-Peña, C.Cobos-Lozada, J.Torres-Jimenez."Metaheuristic algorithms for building Covering Arrays: A review".Revista Facultad de Ingeniería (Rev. Fac. Ing.) Vol. 25 (43), pp. 31-45.

Septiembre-Diciembre, 2016. Tunja-Boyacá, Colombia. ISSN Impreso 0121-1129, ISSN Online 2357-5328.

- Artículo que resume el desarrollo de toda la investigación y los resultados obtenidos, el cuál será presentado en el evento “European Conference on the Applications of Evolutionary Computation EvoApplications – LNCS” a desarrollarse en Parma, Italia en el mes de abril de 2018.

1.5 Organización del documento

El documento de la monografía está estructurado de la siguiente manera:

Capítulo 2. Presenta el marco teórico, que incluye los conceptos básicos necesarios para entrar en contexto a la temática a tratar.

Capítulo 3. Presenta el estado del arte, donde se analizan los métodos y los algoritmos más significativos reportados en la literatura para la construcción de CAs. Se presenta en mayor detalle los algoritmos basados en meta heurísticas o hibridación de las mismas, reportados a la fecha como los de mejores resultados.

Capítulo 4. Presenta el algoritmo híbrido GHSSA, que construye CAs uniformes y mixtos, con diferentes niveles de fuerza, basado en la metaheurística de la Mejor Búsqueda Armónica Global, como estrategia de optimización global y Recocido Simulado como estrategia de optimización local. Este capítulo también referencia la arquitectura utilizada para la ejecución en paralelo del algoritmo.

Capítulo 5. Presenta los resultados obtenidos durante el proceso de comparación del algoritmo GHSSA frente a otros del estado el arte.

Capítulo 6. Presenta las conclusiones y trabajo futuro.

Capítulo 2

Marco Teórico

2.1 Covering Array

Un Covering Array o Arreglo de Cubrimiento denotado por CA(N; k, v, t), es una matriz M de tamaño N × k, donde N (filas de la matriz) es el número de experimentos o casos de pruebas, k es el número de factores o parámetros (del método, función o unidad de procesamiento a evaluar), v es el número de símbolos (valores posibles) por cada parámetro, denominado como alfabeto y t es el grado de interacción entre los parámetros, denominado fuerza [9]. Cada submatriz de tamaño N × t contiene cada tupla de símbolos de tamaño t (t-tupla) al menos una vez [5]. Un CA es óptimo, si contiene el mínimo número posible de filas [4] y se conoce como CAN.

En la Tabla 1, se muestra un ejemplo de un CA (5; 4, 2, 2). La fuerza de este CA es 2 (t=2), con 4 parámetros o factores (k=4) y un alfabeto de 2 (v=2) valores que está definido por los símbolos {0,1}. Obsérvese que en cada par de columnas aparecen las combinaciones {0,0}, {0,1}, {1,0} y {1,1} al menos una vez [5].

1	1	1	0
0	0	0	0
0	1	0	1
1	0	0	1
0	0	1	1

Tabla 1. Ejemplo de CA (5; 4, 2, 2).

Para ilustrar el enfoque del CA aplicado al diseño de las pruebas de software, a continuación, se presenta un ejemplo de un Sistema Web que se espera probar con los parámetros de la Tabla 2.

	Navegador	S.O	BDMS	Conexión
0	IE	Windows 10	MySQL	ISDN
1	Chrome	Ubuntu	Oracle	ADSL
2	Mozilla	Red Hat	SQL Server	Cable

Tabla 2. Parámetros y valores de un Sistema Web

En este ejemplo se tienen 4 factores o parámetros (k = 4) que corresponden al Navegador, Sistema Operativo, Sistema Manejador de Base de Datos o BDMS y Conexión, cada uno con 3 valores posibles (v = 3). A continuación, en la Tabla 3 se muestra parte de una prueba exhaustiva, en la que se generan todas las posibles combinaciones entre todos los parámetros (definiendo una fuerza de interacción de t = k, que en este caso es 4). Entre mayor sea el grado de la fuerza (t) y el número de valores para cada parámetro (v) es mayor el número de pruebas exhaustivas que se deben realizar. Este número de pruebas es igual a v^t. Para el ejemplo, si se quiere realizar una prueba exhaustiva, se tendrán que definir un total de 3^4 o 81 casos de prueba y con esto se logra el cubrimiento del 100% de los casos.

	Navegador	S.O	BDMS	Conexión
0	IE	Windows 10	MySQL	ISDN
1	IE	Windows 10	MySQL	ADSL
2	IE	Windows 10	MySQL	Cable
3	IE	Windows 10	Oracle	ISDN
4	IE	Windows 10	Oracle	ADSL
5	IE	Windows 10	Oracle	Cable
6	IE	Windows 10	SQL Server	ISDN
7	IE	Windows 10	SQL Server	ADSL
8	IE	Windows 10	SQL Server	Cable
9	IE	Ubuntu	MySQL	ISDN
10	IE	Ubuntu	MySQL	ADSL
11	IE	Ubuntu	MySQL	Cable
12	IE	Ubuntu	Oracle	ISDN
…	…	…	…	…
79	Mozilla	Red Hat	SQL Server	ISDN
80	Mozilla	Red Hat	SQL Server	ADSL
81	Mozilla	Red Hat	SQL Server	Cable

Tabla 3. Parte de una prueba exhaustiva para el Sistema Web

Sin embargo, si la interacción entre dichos parámetros se lleva a fuerza 2 (t=2), el número de posibles combinaciones se reduce a 3^2 o 9 casos de prueba, garantizando un amplio cubrimiento o cobertura en la prueba con el mínimo esfuerzo posible [6].

La Tabla 4 muestra el CA (9; 4, 3, 2) resultante del ejemplo anterior con fuerza 2 (t=2) y alfabeto 3 (v=3). Para hacer el mapeo entre el sistema Web y el CA, cada posible valor de cada parámetro de la Tabla 2, es etiquetado por el número de fila, para este caso 0, 1 y 2. Las combinaciones que deben aparecer al menos una vez en cada subconjunto de tamaño N×t son: {0,0}, {0,1}, {0,2}, {1,0}, {1,1}, {1,2}, {2,0}, {2,1}, {2,2}.

0	0	0	0
0	1	1	1
0	2	2	2
1	0	1	2
1	1	2	0
1	2	0	1
2	0	2	1
2	1	0	2
2	2	1	0

Tabla 4. CA resultante para el sistema Web

La Tabla 5 muestra específicamente los casos de prueba o experimentos a realizar. Obsérvese que cada uno de los nueve experimentos es análogo a cada una de las filas del CA.

IE	Windows 10	MySQL	ISDN
IE	Ubuntu	Oracle	ADSL
IE	Red Hat	SQL Server	Cable
Chrome	Windows 10	Oracle	Cable
Chrome	Ubuntu	SQL Server	ISDN
Chrome	Red Hat	MySQL	ADSL
Mozilla	Windows 10	SQL Server	ADSL
Mozilla	Ubuntu	MySQL	Cable
Mozilla	Red Hat	Oracle	ISDN

Tabla 5. Casos de Prueba para el sistema Web

2.2 Búsqueda Armónica

El algoritmo de la Búsqueda Armónica (Harmony Search, HS), propuesto en 2001 por Zong Woo Geem y Kang Seo Lee [10], es un algoritmo meta heurístico que simula el proceso de improvisación musical y cuyo objetivo es encontrar un perfecto estado de armonía. Esta armonía en música es análoga a encontrar un óptimo en

un proceso de optimización. Un músico siempre intentará producir una pieza musical con perfecta armonía. Una solución óptima en un problema de optimización debe ser siempre la mejor solución disponible para el problema, bajo los objetivos dados y ciertas restricciones. Ambos procesos intentar generar el mejor u óptimo [11].

Durante el proceso de improvisación musical que realiza un músico puede presentarse tres posibles opciones a saber: (1), que el músico toque exactamente una melodía que ya conoce. (2), que toque algo similar a la melodía conocida pero con un ligero ajuste de tono y tercero, que componga una nueva melodía a partir de notas seleccionadas aleatoriamente. Gemm *et al*. formalizaron estas tres opciones en los tres componentes generales del algoritmo: el uso de la memoria armónica, el ajuste de tono y la asignación al azar o aleatorización. [11].

Para el primer componente, un correcto manejo de la memoria armónica asegura que buenas armonías sean consideradas como elementos de nuevos vectores solución. Para hacer un uso efectivo de la memoria, el algoritmo hace uso de un parámetro denominado tasa de consideración de la memoria armónica o harmony memory considering rate (HMCR). Si ésta tasa es muy baja, solo unas cuantas armonías élite serán seleccionadas y la convergencia del algoritmo será lenta. Si la tasa es muy alta, cercana a uno, conduce a generar no buenas soluciones o armonías. Usualmente el valor de HMCR está comprendido entre 0.7 y 0.95.

El segundo componente, el ajuste de tono tiene dos parámetros a saber: el tono de ancho de banda o pitch bandwidth(BW) y la tasa de ajuste del tono o pitch adjusting rate (PAR). En términos musicales, un ajuste de tono significa cambiar la frecuencia, lo que significa generar un valor ligeramente diferente en el algoritmo de Búsqueda Armónica. El valor generalmente usado para PAR está comprendido entre 0.1 y 0.5.

El tercer componente, la aleatorización permite la diversidad en las soluciones. Aunque el ajuste de tono tiene un papel similar, está limitado a cierta área y por lo tanto corresponde a una búsqueda local. El uso de la aleatorización permite llegar más lejos en la exploración de diversas soluciones para alcanzar la optimalidad global.

Existen diversas variaciones de esta propuesta, entre ellas la búsqueda armónica mejorada o Improve Harmony Search (IHS) [12], la cual genera sus vectores solución a través del ajuste dinámico de los parámetros PAR y BW, logrando

mejoras a nivel de precisión y velocidad del convergencia. La mejor búsqueda armónica global o Global-Best Harmony Search (GHS,) [13], la cual hibrida la búsqueda armónica original con el concepto de inteligencia de enjambre propuesto en PSO [13] y la nueva búsqueda armónica global (NGHS) [14], basado en PSO y mutación genética.

2.2.1 Mejor Búsqueda Armónica Global

El algoritmo de la Mejor Búsqueda Armónica Global (GHS) combina la Búsqueda Armónica con el concepto de inteligencia de enjambre planteado en el algoritmo de Optimización por Enjambre de Partículas (Particle Swarm Optimization, PSO). Generalmente en un sistema PSO, las partículas vuelan en un espacio de búsqueda y cada una de ellas representa una solución candidata al problema de optimización. La mejor posición visitada anteriormente por la partícula y la posición de la mejor partícula en el enjambre repercute en la posición actual de la partícula. GHS modifica el paso en el algoritmo HS relacionado con el ajuste de tono, de modo que la nueva armonía pueda imitar la mejor armonía en la memoria armónica, adicionando así una dimensión social al HS. El parámetro BW es eliminado. Esta modificación permite que el algoritmo GHS trabaje eficientemente en problemas discretos y continuos.

2.3 Recocido Simulado

El Recocido Simulado o Simulated Annealing (SA) [15] es una técnica de optimización estocástica de propósito general, basada en los pasos para el recocido de metales, usado en la industria para obtener materiales más resistentes y con mejores cualidades. El método inicia con un proceso de calentamiento, que consiste básicamente en fundir el material a una alta temperatura alta hasta llegar a su estado líquido. En ese momento, los átomos aumentan considerablemente su movilidad dentro de la estructura del material. Luego comienza un proceso de enfriamiento, donde la temperatura se desciende gradualmente por etapas, hasta que los átomos se configuren de manera adecuada antes de perder totalmente su movilidad y se logre así un equilibrio térmico. Al finalizar el proceso, se logra obtener una estructura altamente regular y estable. Estudios han demostrado que a descensos abruptos de temperatura o el no esperar el tiempo suficiente en cada etapa, la estructura resultante no es óptima.

Capítulo 3

Estado del Arte

3.1 Métodos y algoritmos para construir Covering Arrays

A continuación se analizan los métodos que han sido desarrollados a la fecha para construir CAs clasificados en cuatro grupos principales, a saber [9]: métodos exactos, métodos algebraicos, métodos Greedy y métodos basados en meta heurísticas [6].

3.1.1 Métodos exactos

Son métodos que permiten la construcción de CAs óptimos, pero que sólo son prácticos para la construcción de arreglos de cobertura pequeños [9]. En 2002 Meagher presenta una propuesta para la generación de CAs no isomorfos usando un algoritmo de backtracking que construye recursiva e independientemente cada fila del CA. El algoritmo construye todos los CAs posibles de un tamaño específico pero solo trabaja con alfabeto binario [16]; EXACT (EXhaustive seArch of Combinatorial Test suites) propuesto en 2006, que permite la generación de arreglos de cobertura mixtos (Mixed Covering Arrays, MCA, arreglos que tienen diferente alfabeto en sus columnas) reduciendo el espacio de búsqueda al evitar explorar arreglos isomorfos. Utiliza técnicas de backtracking para devolverse en el proceso de búsqueda cuando se estanca en el proceso de construcción del MCA [4]. En esta propuesta el tiempo de cómputo se incrementa considerablemente cuando el número de filas (N) del CA se acerca al óptimo, además, no encuentra la solución para ciertas configuraciones. En 2009 [17], se presenta un algoritmo que permite construir CAs con una técnica de backtracking pero que crea el CA con un orden lexicográfico de filas y columnas para evitar búsquedas en arreglos simétricos. El algoritmo propuesto logró igualar las mejores soluciones encontradas por EXACT para CAs binarios pequeños empleando menor tiempo. También reportó mejores CAs que los obtenidos con IPOG-F. El algoritmo está limitado para alfabeto

binario y fuerza $3 \leq t \leq 5$. En 2006 se realizó una propuesta de construcción de CAs usando cuatro modelos de programación por restricciones codificados como un problema SAT (propositional satisfiability problem) Los modelos propuestos son capaces de igualar límites existentes y encuentran nuevos valores óptimos para problemas de tamaño relativamente moderado. Sin embargo, cuando la complejidad del CA a construir aumenta, ya sea en términos del alfabeto o la fuerza, el desempeño decae [18]; En esta misma línea, en 2008, López-Escogido et al crean CAs basado en SAT. La principal contribución de esta propuesta se centra en un solucionador eficiente de non-CNF (forma normal no conjuntiva) SAT que usa búsqueda local y es capaz de igualar o incluso mejorar algunos resultados generados por otras aproximaciones reportadas previamente, sin embargo, el algoritmo solo trabaja con fuerza $t = 2$ [19]. Torres-Jiménez et al en 2015 propusieron un método directo para crear CAs óptimos de fuerza 2 basado en coeficientes binomiales y en un algoritmo basado en ramificación y poda que une los resultados de los coeficientes, permitiendo llegar a CAs con gran números de columnas [20]. El algoritmo fue comparado con IPOG [21], uno de los mejores algoritmos greedy para construcción de CAs y superó a IPOG para valores pequeños de k, pero a medida que k crece IPOG proveyó mejores resultados. Las experimentaciones se vieron limitadas respecto a que IPOG solo trabaja con fuerza $t \leq 6$, mientras que el algoritmo propuesto trabaja con fuerzas mayores $t \geq 7$.

3.1.2 Métodos algebraicos

Los algoritmos basados en métodos algebraicos construyen CAs en tiempo polinomial usando reglas predefinidas. El cálculo de los CAs usa directamente algunas funciones matemáticas u otros procedimientos algebraicos [6]. Los métodos algebraicos son aplicables para casos muy particulares y ofrecen construcciones eficientes en cuanto a tiempo. Sin embargo, actualmente, es difícil generar resultados precisos en una amplia variedad en los valores de las entradas [7, 22]. Por ejemplo para el caso de fuerza 2 y alfabeto 2 en 1973 se propusieron dos algoritmos que crean CAs óptimos, el primero propuesto por Katona [23] y el segundo por Kleitman y Spencer [24]. Chateauneuf et al en 1999 [25] propusieron un algoritmo para crear CAs de fuerza tres usado acciones de grupos sobre los símbolos/alfabetos y luego en 2005, Meagher y Stevens adaptaron este método para crear CAs de fuerza 2 usando vectores inicializadores que se rotan y trasladan para obtener el CA [26]. Hartman en 2005 propone un método para elevar al cuadrado el número de columnas de un CA usando como herramienta intermedia un arreglo ortogonal [27]. Colbourn et al en 2006 proponen un método para

multiplicación de dos CAs de fuerza dos que da como resultado un nuevo CA con la multiplicación de las columnas originales y la suma de las filas originales [28]. Y Colbourn en 2010 propone la construcción de CAs a partir del análisis de matrices ciclotómicas [29].

3.1.3 Métodos Greedy

Se ha encontrado que los algoritmos basados en métodos Greedy son relativamente más eficientes y rápidos en cuanto a tiempo de ejecución. También han sido precisos para una amplia variedad de entradas. Sin embargo, los CAs generados no son siempre los de menor tamaño, comparados con los CA generados con otros métodos [7]. Entre las propuestas greedy se pueden mencionar, el algoritmo de densidad determinista (Deterministic Density Algorithm, DDA) propuesto por Bryce y Colbourn en 2007 que crea CAs de fuerza 2 una fila a la vez [30]. En 1998 [31], Lei y Tai proponen el algoritmo In-Parameter-Order (IPO) extendiendo el arreglo tanto en filas como columnas, es decir, se parte de un CA de un número k de columnas y obtiene uno de k+1 columnas. Luego este algoritmo IPO es generalizado para fuerzas superiores a 2 en IPOG en 2007 [21], que luego es mejorado en IPOG-F en 2008 [32]. Además en 2012, Calvagna y Gargantini extienden las ideas de IPO para construir MCAs de cualquier fuerza en [33].

3.1.4 Métodos basados en Meta Heurísticas

Los algoritmos basados en meta heurísticas son computacionalmente intensivos (demandan mucho tiempo de cómputo) y han proporcionado los resultados más precisos y competitivos a la fecha. Han logrado generar los CAs de tamaño más pequeños conocido (óptimos) a un costo significativo en tiempo de ejecución [7, 34].

Entre los algoritmos más destacados para la construcción de CAs se encuentran:

- Búsqueda Tabú, que se ha aplicado con éxito a una variedad de problemas de optimización combinatorial [35, 36].
- Recocido Simulado, el cual ha presentado los resultados más precisos y además ha encontrado nuevas y mejores soluciones para los parámetros de entradas con varios valores [37].
- Algoritmos genéticos, que toman como modelo de base la supervivencia de los individuos más aptos. Estos algoritmos usan normalmente como operadores la selección, cruce, mutación y remplazo. Estos algoritmos reciben un conjunto de casos de prueba candidatos y proporciona como salida el subconjunto de casos

de prueba de más alto valor [38].

- Colonias de Hormigas, este algoritmo simula computacionalmente la comunicación indirecta que realizan las hormigas para establecer el camino más corto entre su ubicación inicial y una fuente de comida. Un caso de prueba en este algoritmo se representa como una ruta desde un punto de inicio hasta un punto final objetivo. Cuando una hormiga alcanza el nodo objetivo, se deposita en cada camino de los que ha visitado, una cantidad de feromonas proporcional a la calidad de la solución. Si una hormiga tiene que elegir entre diferentes caminos, se va por aquel que tenga más feromonas [39].
- Optimización por Enjambre de Partículas, este algoritmo está inspirado en el comportamiento o movimientos de ciertos organismos de la naturaleza como enjambres de abejas, bandadas de aves o bancos de peces. Los cuales, intentan después de explorar en varias áreas, localizar aquellas regiones del espacio donde se concentra la mayor cantidad de alimento. Finalmente todo el enjambre orientará la búsqueda hacia esa nueva dirección[40].
- Búsqueda Amónica, este algoritmo basa su funcionamiento en el proceso de improvisación musical que ocurre cuando un músico busca producir una armonía agradable, tal como lo que ocurre durante una improvisación de jazz. Para este proceso hay tres posibles opciones: Uno, tocar una melodía exactamente como la conoce, como está en su memoria. Dos, tocar algo similar a la melodía anteriormente mencionada con un ligero ajuste en el tono. Tres, componer una nueva melodía con notas seleccionadas aleatoriamente. Estas tres opciones se formalizan en [10] y corresponden a los componentes del algoritmo: Uso de la memoria armónica, ajuste de tono y aleatoriedad.

3.2 Algoritmos basados en Metaheurísticas

A continuación, se presenta en mayor detalle las investigaciones de los algoritmos basados en metaheurísticas o hibridación de las mismas para la construcción de CAs, reportados a la fecha con los de mejores resultados.

3.2.1 Recocido Simulado

En el año 2003, Cohen *et al.* [41] aplican la meta heurística del SA para la construcción de CAs. En esta propuesta, se inicia escogiendo aleatoriamente una solución factible inicial S que corresponde a un array Nxk con símbolos

aleatoriamente escogidos de la especificación del CA deseado. A través de una secuencia de pruebas, se selecciona aleatoriamente una celda del array y se cambia el símbolo por uno nuevo. Si la trasformación resulta en una solución factible S', donde el costo c(S') sea menor o igual al costo de c(S), entonces ésta es aceptada como la nueva y actual solución factible. De lo contrario, si el resultado es una solución factible de mayor costo, S' es aceptado con una probabilidad $e^{-(c(S')-c(S))/T}$, donde T es la temperatura de control de la simulación. Tener la posibilidad de elegir una solución peor que la actual, le permite al algoritmo salir de óptimos locales. La temperatura se decrementa en pequeños pasos para lograr el equilibrio del sistema. Esto se logra ajustando T en cada iteración a un valor αT, donde α es un número real menor que uno. El algoritmo termina cuando la función objetivo tiene un costo de cero, lo que significa que se tiene un CA. El algoritmo también termina cuando el costo de la solución actual no cambia después de cierto número de pruebas. Los resultados de esta investigación indican que el SA permite construir CAs más pequeños que los métodos algebraicos, sin embargo, falla para problemas más grandes especialmente cuando la fuerza t es igual o superior a 3.

En 2008, Cohen [42] realiza ciertas modificaciones a la propuesta inicial, incluyendo un refinamiento al algoritmo de Recocido Simulado que permitió encontrar casos de prueba más rápidamente y la inclusión de construcciones algebraicas que permitieron construir casos de prueba mucho más pequeños. Además encontró nuevas cotas para algunos CAs de fuerza tres. Este enfoque híbrido se denominó Augmented Annealing.

En 2010, Torres-Jiménez y Rodríguez-Tello [43] presentaron una nueva implementación del algoritmo SA para construir CAs binarios de fuerza variable hasta t = 5, el cual integra tres características importantes que determinan su desempeño. Primero, incorpora una eficiente heurística para generar soluciones iniciales de buena calidad. Segundo, el diseño de una función de vecindario compuesta que permite a la búsqueda reducir rápidamente el costo total de las soluciones candidatas, evitando al mismo tiempo, caer en mínimos locales. Tercero, un calendario de enfriamiento (cooling Schedule) que de forma efectiva permite al algoritmo converger mucho más rápido, generando soluciones de calidad al mismo tiempo. El algoritmo fue comparado con los algoritmos Deterministic Density Algorithm (DDA), Búsqueda Tabú e IPOG-F [32]. Los resultados demostraron que SA encontró nuevas cotas e igualó otras soluciones previamente conocidas de las referencias seleccionadas.

En 2012, Torres-Jiménez y Rodríguez-Tello [44] presentan una mejora a la implementación del algoritmo SA llamada ISA, para construir CA de fuerza t $\in \{3 - 6\}$ sobre un alfabeto binario. El algoritmo integra dos características claves que determinan su desempeño: Una heurística eficiente que genera soluciones iniciales de buena calidad que contienen un número equilibrado de símbolos en cada columna y una función de vecindario cuidadosamente diseñada que permite buscar de forma rápida y reducir el costo de soluciones candidatas evitando caer en mínimos locales.

El desempeño de ISA fue evaluado a través de una amplia experimentación sobre un conjunto de casos de referencia conocidos, que incluye 127 CA binarios de fuerza 3 a 6 y comparados con varios algoritmos del estado del arte entre los que se encuentran SA, un método Greedy y TS. Los resultados computacionales muestran que el algoritmo ISA tiene la ventaja de producir CAs más pequeños que los otros métodos de comparación, a un costo computacional moderado y sin tener fluctuaciones importantes en su rendimiento promedio.

En 2012, Rodríguez-Cristerna y Torres-Jiménez [45] presentan una aproximación híbrida denominada SA-VNS para la construcción de covering arrays mixtos (Mixed Covering Array, MCA, CAs con diferentes valores posibles en las columnas) basada en Recocido Simulado y una función de búsqueda con vecindarios variables (Variable Neighborhood Search function, VNS). La búsqueda de vecindario variable VNS, es una meta-heurística que intenta evitar quedar atrapada en óptimos locales cambiando la estructura de vecindad donde se realiza la búsqueda. Por lo tanto para cada solución se define un conjunto de vecindades, las cuales se construyen mediante una o varias métricas que serán lógicamente dependientes del problema [46]. Las soluciones obtenidas por el algoritmo híbrido SA-VNS fueron comparadas con otros algoritmos, entre ellos In Parameter Order General(IPOG) [21], IPOG-F y MITs [47], logrando igualar 12 de las mejores soluciones conocidas y mejorando 6 de ellas. El tiempo consumido por el SA-VNS es mayor que el tiempo gastado por los algoritmos mencionados anteriormente, sin embargo, los resultados mejorados justifican el tiempo empleado.

En 2015, Rodríguez-Cristerna y Torres-Jiménez [48] presentan el algoritmo SAVNS, como una mejora a la propuesta SA-VNS. Entre las características principales de esta nueva versión del algoritmo se encuentran: un mecanismo para cambiar el tamaño del vecindario en un rango acorde a movimientos aceptados, un mecanismo para incrementar la temperatura y decrementar las probabilidades de una

convergencia prematura y una mezcla probabilística de dos funciones de vecindario que trabajan como una búsqueda local, diversificando así la estrategia de búsqueda. El algoritmo fue sometido a un proceso de afinamiento que permitió establecer la mejor configuración de los parámetros del algoritmo. Los resultados obtenidos de las pruebas de rendimiento entre SAVNS y otros algoritmos como IPOG-F, MiTS, SA, TS y SA-VNS indican que, bajo un punto de referencia de nueve casos o instancias, SAVNS logró mejorar cinco instancias, de las mejores soluciones conocidas de los algoritmos anteriormente mencionados con los que fue comparado.

En 2016 Torres-Jiménez [49] presenta un algoritmo basado en ISA [44] que iguala o mejora el tamaño de los CAs reportados por Colbourn en [29] para fuerza 3, alfabeto 3 y $105 \leq N \leq 223$, gracias a dos nuevas funciones de vecindario que adiciona. La propuesta definió además un punto de referencia para 25 MCAs y logró encontrar 579 nuevos límites inferiores e igualar 13 soluciones previamente conocidas. El algoritmo está limitado para una fuerza y alfabeto específico.

3.2.2 Búsqueda Tabú

La Búsqueda Tabú (Tabu Search, TS) es un enfoque de optimización de búsqueda local que hace frente a diferentes problemas de optimización combinatoria. Esta estrategia propuesta en 1998 por Glover y Laguna [35], busca evitar caer en ciclos y en mínimos locales, prohibiendo o penalizando movimientos que tiene la solución en la siguiente iteración, a puntos en el espacio de soluciones visitados con anterioridad, de ahí el término tabú. La idea básica de TS es realizar búsqueda local evitando caer en un mínimo local escogiendo movimientos que no mejoran la solución, con la suposición que una mala estrategia escogida puede dar más información que una buena elección al azar. Para evitar regresar a soluciones pasadas y quedar en ciclos se utiliza una memoria temporal llamada lista tabú la cual almacena la historia reciente de la búsqueda.

En 2004, Nurmela [50] usa TS para encontrar CAs. El algoritmo inicia con una matriz M generada aleatoriamente de tamaño Nxk, donde las filas corresponden al alfabeto del CA. El costo de la matriz está definido por el número de combinaciones faltantes. Luego, se selecciona una combinación faltante de forma aleatoria. Se verifica que filas requieren solo el cambio de un único elemento tal que la fila cubra la combinación seleccionada. Esos cambios son los movimientos del actual vecino. El costo es calculado de acuerdo a cada movimiento del vecino y se selecciona el que

genere el menor costo, siempre y cuando el movimiento no sea tabú, es decir, este no se encuentre dentro de la lista tabú. Si existe más de un movimiento con el mismo costo, se selecciona uno de los movimientos realizados en forma aleatoria. El proceso se repite hasta que el costo de la matriz M sea cero o el número máximo de movimientos sea alcanzado.

Los resultados demostraron que la implementación mejoró algunas de las mejores soluciones previamente conocidas. Sin embargo, un inconveniente importante de este algoritmo es que consume considerablemente mucho más tiempo de cálculo que otros algoritmos. Además, en ciertos casos, llegó a requerir varios meses de cálculo para resolver las diferentes instancias de prueba.

En 2009, Walker y Colbourn [51] emplearon el algoritmo de Búsqueda Tabú para generar CAs a partir de permutación de vectores y objetos matemáticos conocidos como Covering Perfect Hash Families (CPHF). Esta representación de un CA es capaz de encontrar eficientemente, arrays más pequeños para fuerzas t más grandes. Además, permitió búsquedas para $t \geq 5$.

En 2010 González-Hernández *et al.* [52] presentan un enfoque basado en Búsqueda Tabú referido como TSA para la construcción de Covering Arrays Mixtos (MCA) de fuerza variable. Las características principales de esta aproximación radican en los siguientes aspectos: primero, el algoritmo escoge entre un conjunto de funciones de vecindario predefinidas, en donde cada una de ellas, tiene asignada una probabilidad de ser seleccionada para crear un nuevo vecino. Segundo, cuenta con un cálculo eficiente de la función objetivo para determinar la mejor probabilidad de selección para cada función de vecindario y tercero, cuenta con una nueva función de inicialización. Dado que el rendimiento del TS depende de los valores de las probabilidades asignadas, un proceso de afinamiento se llevó a cabo sobre las configuraciones de dichas probabilidades. La configuración utilizada por TSA permitió generar MCAs de tamaño más pequeño y en menos tiempo. TSA mejoró el tamaño de los MCAs en comparación con IPOG-F y encontró la solución óptima en 15 instancias, de las 18 que constituyen el conjunto completo. Estas instancias oscilan entre un alfabeto de 2 a 11, el número de columnas de 2 a 20 y la fuerza de 2 a 6.

En 2013 González-Hernández [47] presenta un algoritmo de optimización combinatorial para la construcción de MCAs de fuerza variable denominado MiTS (Mixed Tabu Search), el cual utiliza la estrategia meta heurística de la Búsqueda

Tabú y tiene como principal característica la mezcla de diferentes funciones de vecindad, cada una de ellas con cierta probabilidad de ser elegida. Otros aspectos importantes incluidos en el diseño del MiTS son el tamaño de la lista tabú empleada y la función de inicialización para la creación de la solución inicial. Para cada uno de estos aspectos se propusieron tres diferentes alternativas. Respecto a las funciones de inicialización, se han incorporado en el algoritmo tres (3) diferentes alternativas de hacerlo: de forma aleatoria, utilizando la distancia de Hamming y a través de un subconjunto de t columnas. Los resultados experimentales indicaron que el MiTS fue capaz de establecer 91 nuevas cotas óptimas e igualó 36 de los mejores casos reportados en la literatura, de los cuales 31 ya eran el óptimo, por lo que no eran susceptibles de mejora. El algoritmo no mostró una diferencia significativa en el rendimiento al usar los diferentes tamaños propuestos de las listas tabú.

En 2015, González-Hernández [53] desarrolla una metodología que usa MiTS para construir MCAs más pequeños que los mejores conocidos, con fuerza uniforme para valores de t $\in\{2 - 6\}$. La metodología propuesta utiliza un proceso de afinamiento que haciendo uso de pruebas estadísticas, identifica los valores que afectan significativamente el desempeño de MiTS. Para verificar la eficiencia de la metodología propuesta, se comparó y analizó estadísticamente el desempeño del MITS frente a una robusta selección de conjuntos que incluían las mejores cotas o límites de MCAs con fuerza t $\in\{2 - 6\}$ que han sido reportadas a la fecha para los algoritmos SA, DDA entre otros. La metodología basada en MITS demostró que hubo diferencias significativas entre las soluciones obtenidas y los mejores límites anteriormente reportados.

3.2.3 Algoritmos Genéticos

Un Algoritmo Genético (Genetic Algorithm, GA) es un método adaptativo que se usa para resolver problemas de búsqueda y optimización. Está basado en el proceso de evolución biológica y usa la analogía de la supervivencia de los individuos más aptos [54]. Se parte de una población inicial, la cual evoluciona a través de las generaciones representadas por iteraciones. En cada generación, los individuos son evaluados a través de alguna función de aptitud y en cada generación sobreviven los mejores individuos de la población. Las generaciones siguientes son generadas aplicando sucesivamente operadores genéticos tales como selección, cruce, mutación y reemplazo. Si se llega a la condición de parada y no se ha encontrado una solución, se puede optar por mutar la población masivamente.

En 2001, Stardom [55] presenta los resultados de comparar los algoritmos de optimización de SA, TS y GA para construir CAs. Los tres enfoques lograron encontrar nuevas cotas. Sin embargo, los algoritmos genéticos no fueron efectivos para encontrar CAs de calidad. No solo tomó más tiempo para ejecutar movimientos sino también para encontrar un buen CA. TS fue el algoritmo que arrojó los mejores resultados.
En 2010, Rodríguez-Tello y Torres-Jiménez [56] presentaron un algoritmo memético (algoritmo genético que incluye conocimiento del problema para encontrar mejores soluciones a través de un optimizador local) para encontrar soluciones óptimas para la construcción de CAs binarios de fuerza t = 3, que incorporó características importantes como incluir una heurística eficiente para generar una población inicial de buena calidad y un operador de búsqueda local basado en el afinamiento del algoritmo SA. Los resultados computacionales comparados con otros de la literatura entre los que se encuentran IPOG-F y TS, evidenciaron que el algoritmo propuesto mejoró en 9 casos las mejores soluciones conocidas e igualó el resto de resultados.

En 2016, Sabharwal et al [57] presentan G-PWiseGen, una propuesta generalizada de una herramienta existente de código abierto denominada PWiseGen, la cual es usada para la generación de casos de prueba con fuerza 2. El principal inconveniente de PWiseGen es que requiere conocer de antemano el tamaño N del caso de prueba como entrada. G-PWiseGen genera CAs para fuerza $t \geq 2$ y al incorporar un algoritmo de búsqueda binaria permite fijar límites bajos y altos para el tamaño del CA, eliminando la necesidad de conocer N. G-PWiseGen fue comparada con otras herramientas de código abierto para generar CAs entre las que se encuentran ACTS, Jenny, TVG y CASA. Los resultados evidenciaron que la propuesta requiere mucho más tiempo para generar CAs respecto a las otras iniciativas, debido a la complejidad en las operaciones de cruce y mutación cuando la fuerza t se incrementa, sin embargo, los tamaños de los CAs generados compensan el tiempo empleado en su construcción.

3.2.4 Colonia de Hormigas

El algoritmo de Colonia de Hormigas (Ant Colony Algorithm, ACA) propuesto en 1999 por Dorigo [58], está basado en el comportamiento estructurado que presenta una colonia de hormigas reales para encontrar alimento. En este enfoque, cada camino desde un punto de partida a un punto final se asocia con una solución candidata para un problema dado. Cuando una hormiga alcanza el punto final, la cantidad de feromona depositada en cada borde de la trayectoria seguida por dicha

hormiga, es proporcional a la calidad de la solución candidata correspondiente. Cuando una hormiga tiene que elegir entre los diferentes bordes en un punto dado, el borde con una mayor cantidad de feromona se elige con mayor probabilidad. Como resultado, las hormigas finalmente convergen a un camino más corto.

En 2004, Shiba [59] presentó un algoritmo generador de pruebas basado en algoritmos genéticos y ACA, y los comparó con otros algoritmos entre los que se incluyen SA, IPO y The Automatic Efficient Test Generator (AETG) [60] que usa estrategia greedy. Los resultados para fuerza t $\in\{2 - 3\}$ evidenciaron un buen desempeño a nivel general con respecto al tamaño de los casos de prueba (valor N en un CA) y la cantidad de tiempo requerido para ello. Sin embargo, los resultados de algoritmo genético no siempre fueron óptimos. Los resultados obtenidos de este algoritmo fueron mejores que los generados por el GA.

En 2009, Chen [61] adaptó el algoritmo ACA para construir un conjunto de pruebas priorizadas para interacción de parejas, un CA y un MCA de fuerza t = 2. Propone concretamente cuatro algoritmos para la generación de pruebas basadas en ACA, en busca de una implementación más efectiva. Aunque los resultados obtenidos fueron competitivos, el rendimiento de los mismos no pudo ser generalizado para instancias con diferentes características.

3.2.5 Optimización por Enjambre de Partículas

El algoritmo de Optimización por Enjambre de Partículas (Particle Swarm Optimization, PSO) propuesto en 1995 por Kennedy[40] es un popular método de optimización. PSO intenta optimizar un problema a partir de la manipulación de un cierto número de soluciones candidatas. Cada solución es representada por una partícula que trabaja en un espacio de búsqueda para encontrar una mejor posición o solución del problema. La población entera es llamada enjambre. Por lo tanto cada partícula tiene una posición aleatoria y actualiza su posición iterativamente con la esperanza de encontrar mejores soluciones. Cada partícula mantiene también información esencial sobre sus movimientos.

Aplicado a la construcción de CAs, una partícula representaría generalmente un caso de prueba. A cada partícula se le asocia un factor de peso que representa el número de interacciones cubiertas por el caso de prueba. Cuando la evaluación de todas las partículas finalice, las de mayor peso serán escogidas para estar en los conjuntos de pruebas.

En 2011, Ahmed y Kamal [62] presentan el desarrollo de una nueva estrategia para la generación de datos de prueba en pares basada en PSO, denominada Pairwise Particle Swarm based Test Generator(PPSTG). En el estudio se evaluó el desempeño de esta propuesta en términos del tamaño de las pruebas generadas contra otras estrategias y herramientas. En un primer escenario, se comparó a PPSTG con los resultados publicados en la literatura para los algoritmos GA, ACA, AETG e IPO[31] un algoritmo Greedy. PPSTG generó conjuntos de pruebas con resultados satisfactorios en la mayoría de los experimentos. Sin embargo, GA y ACA generaron ligeramente mejores tamaños que PPSTG, pero este tuvo mejor desempeño que AETG. SA generó los resultados más óptimos.

En 2011, Ahmed y Kamal [63] presentan VS Particle Swarm Test Generator (VS-PSTG) una propuesta para generar casos de prueba en interacciones de fuerza variable o variable-strength (VS). VS-PSTG adopta a PSO para asegurar la reducción óptima del tamaño de las pruebas. Resultados comparativos con otras estrategias y configuraciones, evidenciaron resultados competitivos. Un caso de estudio empírico fue conducido sobre un sistema software no trivial para mostrar su aplicabilidad y determinar la eficiencia en la generación de casos de prueba, los resultados fueron promisorios.

En 2012, Ahmed *et al*. [64] demuestran la eficiencia de Particle Swarm-based t-way Test Generator (PSTG), una estrategia para generar CAs uniformes y de fuerza variable, que soporta altas fuerzas de interacción de hasta t = 6. PSTG es computacionalmente más liviano en comparación con otros métodos de optimización, debido a la simplicidad en la estructura del algoritmo PSO en el cual se basa y además, supera a otras estrategias en función de los tamaños generados para los CA.

En 2015, Mahmoud y Ahmed [65] presentan una estrategia para construir CAs usando lógica difusa para afinar los parámetros heurísticos que utiliza el algoritmo PSO. El algoritmo fue evaluado con diferentes propuestas entre las que se encuentran SA y Hill Climbing. Los resultados mostraron una mejora significativa en términos del tamaño del CA generado, sin embargo, el mecanismo difuso demandó requerimientos computacionales adicionales. En consecuencia, la estrategia es comparativamente lenta para generar CAs con fuerza t >4.

3.2.6 Búsqueda Armónica

En 2012, Rahman *et al.* [66] proponen y evalúan una estrategia llamada Pairwise Harmony Search algorithm-based Strategy (PHSS) para la generación de datos de prueba en pares. PHSS fue evaluada en dos partes. En una primera parte, se tomó una configuración del sistema con 10 parámetros de entrada de *v* valores, donde *v* variaba de 3 a 10. Luego tomaron otra configuración del sistema con *p* parámetros de entrada de 2 valores, donde *p* variaba de 3 a 15. El objetivo era investigar como PHSS se comportaba respecto a la variación de *v* y *p*. Los resultados mostraron que su desempeño no se vio afectado por el número creciente de *v* y *p*. Además el algoritmo generó en la mayoría de los casos, tamaños de conjuntos de prueba más pequeños que otras estrategias con las que fue comparada como PPSTG, IPOG, TConfig, Jenny, TVG entre otras. En una segunda parte, se generaron otras configuraciones del sistema con el fin de comparar el rendimiento de PHSS contra otras estrategias representativas de la literatura como SA, GA, ACA. Los resultados de PHSS fueron competitivos.

En 2012, Rahman et al. [66] diseñan, implementan y evalúan un algoritmo basado en HS para fuerza variable denominado Harmony Search Strategy (HSS), el cual se compone de dos algoritmos principales: el primero, es un algoritmo de generación de interacción que genera parámetros, tuplas y valores de interacción basados en una fuerza especificada como también en una lista de restricciones. Segundo, un algoritmo para la generación de los casos de prueba en el cual se especifica el tamaño de la memoria armónica, la tasa de consideración de la memoria armónica, la tasa de ajuste del tono y los criterios de parada. Los resultados del algoritmo fueron evaluados en dos partes: en una primera parte, se evaluó el desempeño de HSS en comparación con otras estrategias de fuerza variable entre las que sobresalen VS-PSTG, ACS, SA, IPOG. Dependiendo de la fuerza, parámetros y valores definidos, HSS generó los resultados más óptimos para fuerzas muy altas ya que es capaz de manejar fuerzas de interacción de hasta $t = 15$. Superando a otros algoritmos como ACS y SA que generan casos de prueba con fuerza de interacción $t \leq 3$ y VS-PSTG que genera casos de prueba con fuerza de interacción $t \leq 6$. Sin embargo, SA generó los resultados más óptimos con bajos valores de interacción $t \leq 3$. HSS, VS-PSTG, and ACS obtuvieron resultados iguales o cercanos a los de SA. IPOG no soportó fuerzas superiores a $t = 6$. En una segunda parte, HSS se comparó con otras estrategias que soportan restricciones como SA_SAT, PICT, TestCover y mAETG_SAT. Sin embargo, estas dos últimas propuestas a pesar de que soportan el manejo de restricciones no lo hacen para la generación de

pruebas de fuerza variable, por lo tanto, no se consideraron en los experimentos de fuerza variable. PICT reportó los peores resultados. Finalmente, los resultados obtenidos por HSS fueron competitivos frente a los entregados por SA_SAT ya que en la mayoría de los casos se logró igualar los resultados y en solo algunas configuraciones los logró superar.

En 2015, Bao *et al*. [67] presentan el algoritmo Improved Harmony Search (IHS) una propuesta que busca mejorar la velocidad de convergencia del algoritmo estándar HS. IHS hace uso de un algoritmo Greedy para generar un conjunto óptimo de soluciones iniciales para la inicialización de la memoria armónica. Para evitar que el algoritmo caiga en óptimos locales se ajustaron dinámicamente los valores de HMCR y PAR. Los resultados de los experimentos lograron evidenciar que el tamaño de los casos de prueba generados por IHS es menor que los generados por HS. Se demostró además que cuando la fuerza *t* es pequeña, el tiempo de ejecución de IHS es muy similar al de HS. Sin embargo, cuando *t* aumenta, el tiempo de ejecución de IHS decrementa significativamente comparado con HS. Al comparar IHS con otros algoritmos inteligentes como GA, ACA and SA se observó que en la mayoría de los experimentos los tamaños de los casos de prueba de IHS fueron más óptimos.

Capítulo 4

Algoritmo GHSSA

GHSSA es un algoritmo híbrido que construye arreglos de cubrimiento o Covering Arrays (CA) uniformes[1] y mixtos[2], con diferentes niveles de fuerza ($2 \leq t \leq 6$), basado en la metaheurística de la Mejor Búsqueda Armónica Global, como estrategia de optimización global y Recocido Simulado como estrategia de optimización local.

El procedimiento GHSSA recibe como parámetros el valor de varias constantes a saber: NI, que representa el número de iteraciones o improvisaciones. ParMin, que representa la tasa mínima de ajuste del tono. ParMax, que representa la tasa máxima de ajuste del tono. HMCR, que representa la tasa de consideración de la memoria armónica y HMS que representa el tamaño que tendrá la memoria armónica.

La Tabla 6 muestra el procedimiento de GHSSA, que luego se explica en detalle.

```
    procedimiento GHSSA(entero NI, real ParMin, real ParMax, real HMCR, entero HMS)
                                HMS)
1       Vector<Armonia> HM[HMS] //declara la memoria armónica
2       inicializarMemoriaArmonica()
3       entero posPeor ←buscarPeor()
4       entero posBest ←buscarMejor()
5       Armonia best ← HM[posBest]
6       for i ←0 to NI-1 do
7           if best.Fitness = 0 then
8               break
9           end_if
10          par ← ParMin + (((ParMax - ParMin) / NI) * i)
11          Armonia improviso // declara el nuevo improviso
12          if Random(0,1)< HMCR then
13              if Random(0,1)< par then
14                  improviso← HM[posBest]
15              else
16                  improviso← HM[Random(HMS)]
```

[1] Igual alfabeto para cada uno de los K parámetros del CA
[2] Diferente alfabeto para cada uno de los K parámetros del CA

```
17              end_if
18          else
19              real filas ←0.10 * N
20              improviso← mutarImproviso(HM[Random(HMS)], entero(filas))
21          end_if
22          calcularFitnessCA (improviso)
23          optimizarCAConSA(improviso)
24          if improviso.Fitness < HM[posPeor].Fitness and not(existe(improviso)) then
25              HM[posPeor]← improviso
26              posPeor ←buscarPeor()
27              posBest ←buscarMejor()
28              best← HM[posBest]
29          end_if
30      end_for
    fin_procedimiento
```

Tabla 6 Procedimiento GHSSA

En la línea 1, se crea la memoria armónica HM, representada como un vector o lista de armonías. Una armonía es un objeto que a nivel general almacena tres elementos: un Covering Array, el Fitness que es el valor de la función objetivo o de aptitud del CA y un objeto P, utilizado para calcular el Fitness. Estos elementos se explican detalladamente en una sección posterior. En la línea 2 se llama al procedimiento que permite inicializar la memoria armónica y que se muestra en la Tabla 7. En la línea 3, la variable posPeor guarda el valor que retorna la función buscarPeor, la cual busca la posición en la memoria armónica, donde se encuentra la armonía que registra el peor fitness que en este caso es el valor más alto del fitness. La función buscarPeor se muestra en la Tabla 8. En la línea 4, la variable posBest guarda el valor que retorna la función buscarMejor, la cual busca la posición en la memoria armónica, donde se encuentra la armonía que registra el mejor fitness que en este caso es el valor más bajo del fitness. La función buscarMejor se muestra en la Tabla 9. En la línea 5, se obtiene de la memoria armónica, la armonía que se encuentra en la posición posBest y se asigna a un objeto de tipo Armonia denominado best.

De la línea 6 a la línea 30, se realiza el proceso iterativo de improvisación, enmarcado en un número de improvisaciones definida en la constante NI. En la línea 7 se pregunta si el fitness de best es cero. Si es así, termina el ciclo de improvisaciones dado a que ya se logró construir un CA para la configuración solicitada. En la línea 10 se define la variable par, la cual representa la tasa de ajuste del tono y cuyo valor es asignado a través de una expresión matemática predefinida dentro del algoritmo original de la Mejor Búsqueda Armónica Global. La variable par

cambia dinámicamente con cada iteración o improvisación. En la línea 11 se crea un objetivo de tipo Armonía denominado improviso. Luego en la línea 12, se pregunta si el valor de un número aleatorio generado uniformemente entre 0 y 1 es menor que el valor de la constante HMCR. Si es así, en la línea 13 se pregunta si el valor de un nuevo número aleatorio uniforme entre 0 y 1 es menor que el valor de la variable par. Si es así, en la línea 16, se guardará en el objeto improviso, la armonía con el mejor fitness registrado. Si no, en la línea 18, el objeto improviso guardará una armonía escogida aleatoriamente de la memoria armónica. Este bloque de acciones es similar pero no igual a las tres reglas definidas en el algoritmo original de la Mejor Búsqueda Armónica Global, las reglas originales no se pudieron usar ya que el algoritmo no llegaba a construir los CA solicitados usando esas reglas.

Si el valor aleatorio no fue menor que el valor de la constante HMCR, en la fila 19, la variable real filas almacena un valor que representará el número de filas que se mutarán o cambiarán sobre el objeto improviso. En la línea 20, el objeto improviso se generará con base en el resultado devuelto por la función mutarImproviso, la cual recibe como parámetro una armonía que ha sido seleccionada aleatoriamente de la memoria armónica y al valor de la variable filas. La función retorna la armonía modificada en ciertas filas y en ciertas posiciones de dichas filas. La función mutarImproviso se muestra en la Tabla 10.

En la línea 22 se llama al procedimiento calcularFitnessCA que se muestra en la Tabla 16 y el cual permite calcular el valor del fitness para el improviso. El funcionamiento del procedimiento calcularFitnessCA se detalla en una sección posterior. En la línea 23 se llama al procedimiento optimizarCAconSA que se muestra en la Tabla 27 y el cual permite aplicar el algoritmo de Recocido Simulado para optimizar o mejorar el fitness para el improviso. El funcionamiento del procedimiento optimizarCAconSA se detalla en una sección posterior. En la línea 24 se realizan dos preguntas, la primera, si el fitness del improviso es menor que el fitness de la armonía dentro de la memoria armónica que registra el peor fitness y la segunda, si el improviso que se generó no está ya dentro de la memoria armónica. Esta segunda pregunta se comprueba a través de la función existe que se muestra en la Tabla 11. Si ambas condiciones se cumplen, entonces en la línea 25, el improviso entra a reemplazar en la memoria armónica, el lugar donde se encontraba la armonía con el peor fitness. En la línea 26 y línea 27, se calculan nuevamente las posiciones en la memoria armónica donde se encuentran las armonías con el peor

y mejor fitness. En la línea 30 se obtiene de la memoria armónica, la armonía que se encuentra en la posición posBest y se asigna al objeto best.

Procedimiento inicializarMemoriaArmonica

En la Tabla 7 se muestra el método inicializarMemoriaArmonica, que permite la inicialización de la Memoria Armónica (Harmony Memory, HM), representada como un vector de Armonías y cuyo tamaño es definido de acuerdo a un parámetro denominado HMS (Harmony Memory Size). Para cada armonía creada se llena su CA y se calcula su fitness respectivo. Luego, se optimiza la armonía y finalmente se almacena en la memoria armónica.

```
    procedimiento inicializarMemoriaArmonica ()
1       entero filasCandidatas←valor
2       for i ← 0 to HMS-1 do
3           Armonia armonía
4           inicializarArmonia(armonia)
5           llenarCAConGreedy(armonia, filasCandidatas)
6           calcularFitnessCA (armonia)
7           optimizarCAConSA(armonia)
8           HM[i] ← armonia
9       end_for
    fin_procedimiento
```

Tabla 7 Procedimiento inicializarMemoriaArmonica

En la línea 1 a la variable entera filasCandidatas se le asigna un valor que es definido por el usuario y que se utiliza posteriormente dentro del procedimiento llenarCAConGredy. De la línea 2 a la línea 9, se crea una a una, las armonías que llenarán a la memoria armónica de acuerdo al tamaño definido en HMS. Para cada armonía se llena su CA y se calcula su fitness respectivo. Luego, cada armonía se optimiza con el Algoritmo de Recocido Simulado y finalmente se almacena en la memoria armónica. Específicamente en la línea 3 se crea una armonía. Cada objeto armonía almacena tres elementos a saber: el primer elemento, es un objeto CA (Covering Array), representado internamente por una matriz de enteros denominada matriz, un valor entero N, un valor entero K, un valor entero t y un vector de enteros V. El segundo elemento, es el Fitness, representado por un valor entero. El tercer elemento es un objeto P que permite calcular el valor del Fitness. El objeto P está representado internamente por una matriz de enteros denominada matrizP, un número entero MaxJ, una matriz J y un vector de enteros Vt. Los elementos

anteriormente descritos se explican a detalle en una sección posterior. La Figura 1 muestra un objeto armonía.

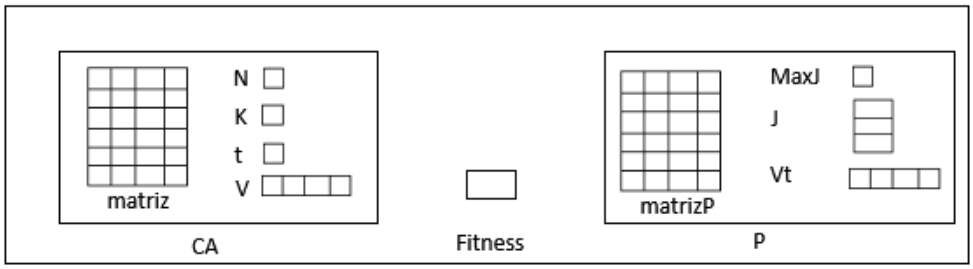


Figura 1 Representación de un objeto armonía

Una vez creada una armonía, en la línea 4 se procede a inicializarla. El procedimiento que permite inicializar la armonía se muestra luego en la Tabla 12. En la línea 5 se llena para cada armonía, su respectiva matriz del CA mediante un método Greedy, el cual, va creando fila por fila, buscando incluir filas diferentes a las ya existentes. Este método utiliza el parámetro filasCandidatas que es definido por el usuario y cuyo uso es explicado posteriormente. El procedimiento que permite llenar la matriz del CA con el método Greedy se muestra en la Tabla 15. En la línea 6, se calcula el fitness del CA, que corresponde a las t-adas faltantes para que se obtenga la matriz del CA que se está buscando. El procedimiento que permite calcular el fitness se muestra más adelante en la Tabla 16. En la línea 7, se optimiza la matriz del CA a través de la aplicación del algoritmo de Recocido Simulado (Simulated Annealing, SA). El procedimiento que optimiza la matriz del CA se muestra luego en la Tabla 27. Finalmente, en la línea 8 se guarda la armonía en la Memoria Armónica.

A continuación, en la Figura 2 se muestra la representación de la Memoria Armónica.

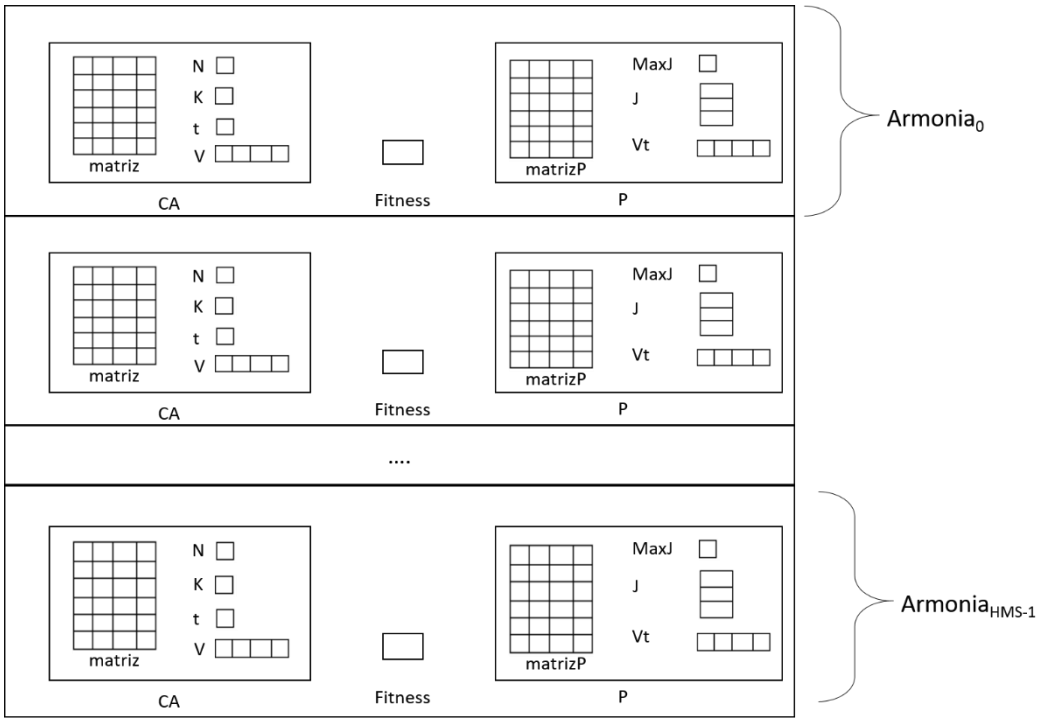


Figura 2 Memoria Armónica o Harmony Memory (HM)

Función buscarPeor

La Tabla 8 muestra la función buscarPeor, que retorna la posición en la memoria armónica donde se encuentra la armonía que registra el peor fitness que en este caso es el valor más alto de fitness.

```
    entero función buscarPeor()
1       entero posPeor←0
2       entero faltanPeor←HM[0].Fitness
3       for i←1 to HMS-1 do
4           if HM[i].Fitness > faltanPeor then
5               faltanPeor ← HM[i].Fitness
6               posPeor ← i
7           end_if
8       end_for
9       return posPeor
    fin_función
```

Tabla 8 Función buscarPeor

En la línea 1, se crea la variable entera posPeor que se inicializa en cero y que indica la posición donde se encuentra la armonía con el peor fitness. En la línea 2, la variable entera faltanPeor almacena el fitness de la armonía que se encuentra en

la posición cero dentro de la memoria armónica. De la línea 3 a la línea 8 se recorre toda la memoria amónica para encontrar el peor fitness. En la línea 4, se pregunta si el fitness de la armonía en una posición i es mayor que el valor de la variable faltanPeor. Si es así, en la línea 5, el nuevo valor de la variable faltanPeor será igual al fitness de la armonía que se estaba evaluando. En la línea 6, la variable posPeor será igual a la posición i. Finalmente, en la línea 9 se retorna el valor de la variable posPeor.

Función buscarMejor

La Tabla 9 muestra la función buscarMejor, que retorna la posición en la memoria armónica, donde se encuentra la armonía que registra el mejor fitness, en este caso es el valor más bajo de fitness. La explicación de esta función es similar a buscarPeor() previamente explicada.

```
    entero función buscarMejor()
1       entero posMejor←0
2       entero faltanMejor←HM[0].Fitness
3       for i←1 to HMS-1 do
4           if HM[i].Fitness < faltanMejor then
5               faltanMejor ← HM[i].Fitness
6               posMejor ← i
7           end_if
8       end_for
9       return posMejor
    fin_función
```

Tabla 9 Función buscarMejor

Función mutarImproviso

La Tabla 10 muestra la función mutarImproviso, la cual recibe una armonía y el valor del número de filas a modificar en dicha armonía. La función retorna la armonía modificada en ciertas filas y en ciertas posiciones de dichas filas.

```
     Armonia función mutarImproviso(Armonia armonia, entero filasMutar)
1        entero valor
2        for i←0 to filasMutar-1 do
3            entero posFila←Random(armonia.CA.N)
4            entero posColumna←Random(armonia.CA.K)
5            Do
6                valor←Random(amonia.CA.V[posColumna])
7            while(armonia.CA.matriz[posFila][posColumna]=valor)
```

```
8           armonia.CA.matriz[posFila][posColumna]←valor
9       end_for
10      return armonia
    fin_función
```

Tabla 10 Función mutarImproviso

En la línea 1, se crea la variable entera valor que será utilizada para ir guardando los diferentes valores del alfabeto para un parámetro específico dentro del vector V. En la línea 2 se da inicio al proceso iterativo que permite cambiar ciertas filas y los valores en ciertas posiciones de dichas filas. En la línea 3, la variable entera posFila almacena un número aleatoriamente generado con base en el número de filas N del CA. En la línea 4, la variable entera posColumna almacena un número aleatoriamente generado con base en el número de parámetros K del CA. En la línea 6, la variable valor tomará aleatoriamente un valor entre los valores del alfabeto que se encuentran en la posición posColumna del vector V, repitiendo esta operación mientras que el valor almacenado en la matriz de la armonía en las posiciones posFila y posColumna sea igual a la variable valor. Cuando sea diferente, en la línea 8 el valor almacenado en la matriz de la armonía en las posiciones posFila y posColumna se cambia por la variable valor. Finalmente, en la línea 10 se retorna la armonía modificada.

Función existe

La Tabla 11 muestra la función existe, la cual retorna un valor booleano de acuerdo si la armonía que llega como parámetro se encuentra o no en la memoria armónica.

```
    booleano función existe(Armonia armonia)
1       for i←0 to HMS-1 do
2           if armonia = HM[i] then
3               return true
4           end_if
5       end_for
6       return false
    fin_función
```

Tabla 11 Función existe

En la línea 1, se da inicio al proceso iterativo que permite ir comparando la armonía con cada una de las armonías registradas en la memoria armónica. Específicamente en la línea 2 se pregunta si la armonía es igual a la armonía que está almacenada en la memoria armónica. Si es así, se retorna el valor de verdadero y la función termina. En la línea 6, se retorna el valor de falso si no se encontró repetida la

armonía con alguna ya registrada en la memoria armónica. La comparación de igualdad se realiza por valor, es decir, si algún dato de la matriz del CA de la armonía es diferente de la matriz del elemento i de la memoria armónica entonces los objetos son diferentes.

A continuación, se describe a detalle cada uno de los procedimientos internos del método inicializarMemoriaArmonica que se describió anteriormente y que se presentó en la Tabla 7.

Procedimiento para inicializar la Armonía

La Tabla 12 presenta el procedimiento inicializarArmonia el cual permite inicializar cada uno de los elementos de una Armonía: el CA y el fitness del CA.

```
    procedimiento inicializarArmonia (Armonia armonia)
1       nombreCA ← leerCAdeBD()
2       inicializarCA(armonia, nombreCA)
3       armonia.Fitness←Integer.MaxValue
    fin_procedimiento
```

Tabla 12 Procedimiento inicializarArmonia

En la línea 1, la variable tipo cadena nombreCA almacena la configuración del CA que se va a construir, la cual es recuperada a través de una consulta a una base de datos que almacena diferentes configuraciones de CAs que se esperan construir con la ejecución del algoritmo. En la línea 2, se inicializa el CA de la Armonía con base en la cadena nombreCA. En la línea 3, se inicializa el fitness con un valor por defecto (el máximo valor que toma un número entero en un lenguaje de programación seleccionado). La Figura 3 muestra un ejemplo de una configuración de un CA.

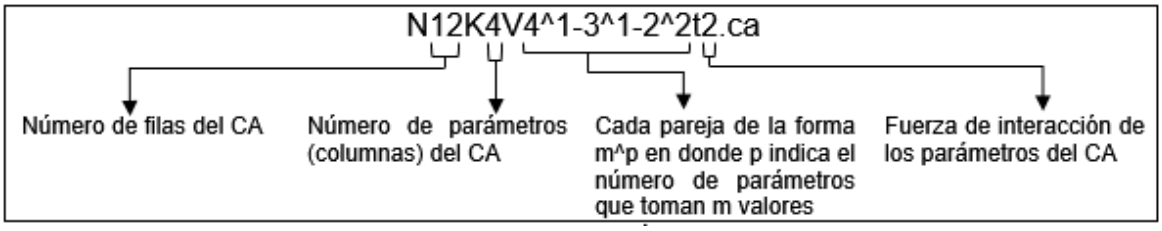


Figura 3 Configuración del CA

Procedimiento para inicializar el CA

El objeto CA almacenado en la Armonía tiene definido internamente varios elementos a saber: un número entero N que representa el número de filas del CA, un número entero K que indica el número de parámetros o columnas que tendrá el CA, un vector V de tamaño K que almacena en cada columna el alfabeto o conjunto de valores que toma cada parámetro del CA y un número entero T que referencia la fuerza de interacción entre los parámetros.

La Tabla 13 muestra el procedimiento inicializarCA el cual permite que cada uno de los elementos del CA tome su correspondiente valor de acuerdo a una configuración que está establecida en una cadena de caracteres.

```
     procedimiento inicializarCA(Armonia armonia, cadena nombreCA)
1        inicio←posicion(nombreCA, 'N')
2        fin←posicion(nombreCA, 'K')
3        longitud← fin – inicio
4        cadenaAuxiliar ←subString(nombreCA, inicio+1,longitud)
5        armonia.CA.N ←entero(cadenaAuxiliar)
6        inicio←posicion(nombreCA, 'K')
7        fin←posicion(nombreCA, 'V')
8        longitud← fin – inicio
9        cadenaAuxiliar ←subString(nombreCA, inicio+1,longitud)
10       armonia.CA.K ←entero(cadenaAuxiliar)
11       inicio←posicion(nombreCA, 't')
12       fin←posicion(nombreCA, '.')
13       longitud← fin – inicio
14       cadenaAuxiliar ←subString(nombreCA, inicio+1,longitud)
15       armonía.CA.t ←entero(cadenaAuxiliar)
16       construirV(armonia, nombreCA)
     fin_procedimiento
```

Tabla 13 Procedimiento inicializarCA

El ejemplo a continuación describe el proceso que se hace para inicializar cada elemento del CA.

En este ejemplo se espera construir un CA con la configuración “N12K4V4^1-3^1-2^2t2.ca”. En la Figura 4 se muestra como la cadena con la configuración del CA será manejada como un vector.

	N	1	2	K	4	V	4	^	1	-	3	^	1	-	2	^	2	t	2	.	c	a
i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*

Figura 4 Configuración del CA representada en un vector

En la línea 1, la variable *inicio* guarda la posición donde empieza la letra 'N', tomando el valor de 0. En la línea 2, la variable *fin* guarda la posición donde empieza la letra 'K', tomando el valor de 3. En la línea 3, la variable *longitud* tomará el valor de 3 - 0 que es igual a 3.

En la línea 4, se utiliza la función predefinida *subString* para obtener una subcadena que por definición inicia en un índice hasta una longitud-1. La variable *cadenaAuxiliar* almacena la subcadena comprendida a partir de la variable *inicio* + 1 hasta el tamaño definido en la variable *longitud*-1. La variable *cadenaAuxiliar* toma el valor de "12".

En la línea 5, el resultado obtenido se convierte a entero y se almacena en el elemento N del CA. En la línea 6, la variable *inicio* almacena la posición donde empieza la letra 'K' y en la línea 7, la variable fin almacena la posición donde empieza la letra 'V'. Las variables *inicio* y *fin* toman respectivamente los valores de 3 y 5. En la línea 8, la variable longitud toma el valor de 2. En la línea 9, la variable *cadenaAuxiliar* toma el valor de "4". En la línea 10, el resultado obtenido se convierte a entero y se almacena en el elemento K del CA. En la línea 11, la variable *inicio* almacena la posición donde empieza la letra 't' y en la línea 12, la variable fin almacena la posición donde empieza el punto '.'. Las variables *inicio* y *fin* toman respectivamente los valores de 17 y 19. En la línea 13, la variable longitud toma el valor de 2. En la línea 14, la variable *cadenaAuxiliar* toma el valor de "2". En la línea 15, el resultado obtenido se convierte a entero y se almacena en el elemento T del CA. Y en la línea 16, se llama al procedimiento construirV que permite construir el vector V de tamaño K, que almacena en cada posición el número de valores que toma cada parámetro K del CA.
A continuación, en la Tabla 14 se presenta el procedimiento que permite llenar el vector V con los alfabetos de las columnas del CA.

```
     procedimiento construirV(Armonia armonia, cadena nombreCA)
1        pos←0
2        Vector<cadena> vectorAuxiliar [ ]
3        Vector <cadena>vectorAuxiliar2 [ ]
4        inicio←posicion(nombreCA, 'V')
5        fin←posicion(nombreCA, 't')
6        longitud← fin – inicio
7        cadenaAuxiliar ←subString(nombreCA, inicio+1,longitud)
8        vectorAuxiliar ← cadenaAuxiliar.split('-')
9        for i ←0 to vectorAuxiliar.length()-1 do
10           vectorAuxiliar2 ←vectorAuxiliar[i].split('^')
11           valoresAlfabeto ← vectorAuxiliar2[0]
```

```
12          parametros← vectorAuxiliar2[1]
13          for j ←0 to parametros-1 do
14              armonia.CA.V[pos]← valoresAlfabeto
15              pos←pos + 1
16          end_for
17      end_for
    fin_procedimiento
```

Tabla 14 Procedimiento construirV

En la línea 1, se crea e inicializa la variable entera *pos*, la cual representa la posición que toma cada elemento del vector V. En las líneas 2 y 3 se crean respectivamente dos vectores de cadenas vectorAuxiliar y vectorAuxiliar2, los cuales almacenan cadenas temporales que surgen cuando se divide la cadena con el nombre del CA, con base en un carácter en particular. En las líneas 4 y 5, la variable entera inicio toma el valor de la posición donde está el carácter 'V' y la variable entera fin toma el valor de la posición donde está el carácter 't'. En la línea 6, se calcula la variable entera longitud que será usada en la línea 7 para obtener la variable tipo cadena cadenaAuxiliar que almacena la subcadena que relaciona el número de valores del alfabeto que toma uno o varios parámetros. En la línea 8, la variable cadenaAuxiliar se divide con base en el carácter '-', generando el vector de cadenas vectorAuxiliar, en donde cada posición tiene una representación de la forma m^p, en donde p indica el número de parámetros que toman m valores. De la línea 9 a la 17, cada elemento del vector de cadenas vectorAuxiliar se divide nuevamente con base en el carácter '^'. Generando el vector de cadenas vectorAuxiliar2. En la posición cero de dicho vector, queda almacenado el número de valores del alfabeto que toman un determinado número de parámetros. En la posición uno, se almacena cuántos parámetros tienen el mismo número de valores del alfabeto. Finalmente, se extrae el número almacenado en la posición cero del vector vectorAuxiliar2 y se almacenan en el vector V.

Por ejemplo, retomando la configuración del CA de la Figura 4, la variable inicio guarda la posición donde empieza la letra 'V', tomando el valor de 5. La variable fin guarda la posición donde empieza la letra 't', tomando el valor de 17. La variable longitud tomará el valor de 17 - 5 que es igual a 12.

La variable cadenaAuxiliar almacena la subcadena comprendida a partir de la variable inicio + 1 hasta el tamaño definido en la variable longitud -1, tomando el valor de “4^1-3^1-2^2”.

Paso siguiente, se divide cadenaAuxiliar con base en el carácter '-' creando así a vectorAuxiliar, representado como aparece en la Figura 5.

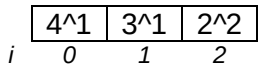


Figura 5 Representación de vectorAuxiliar

La cadena “4^1” indica que hay 1 parámetro (en este caso el primero) que toma 4 valores como alfabeto, la cadena “3^1” indica que hay 1 parámetro (el segundo en este caso) que toma 3 valores como alfabeto y la cadena “2^2” indica que hay 2 parámetros (el tercero y el cuarto en este caso) donde cada uno toma 2 valores como alfabeto.

La estructura repetitiva externa for permite dividir cada elemento i del vectorAuxiliar con base en el carácter '^', generando el vector vectorAuxiliar2, representado como aparece en la Figura 6.

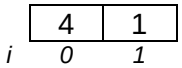


Figura 6 Primera Representación de vectorAuxiliar2

Donde la variable valoresAlfabeto tiene asignado el valor de 4 obtenido de la posición cero y la variable parametros tiene asignado el valor de 1, obtenido de la posición 1.

En la estructura repetitiva interna for el vector V se llenará parámetros-1 veces con el valor almacenado en la variable valoresAlfabeto. Para el primer ciclo, la posición 0 del vector V toma el valor de 4. Ver Figura 7.

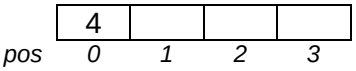


Figura 7 Vector V con valor almacenado en la posición cero

El proceso se repite tomando ahora la cadena “3^4”. Se divide con base en el carácter '^', generando a vectorAuxiliar2, representado como se observa en la Figura 8:

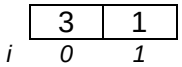


Figura 8 Segunda Representación de vectorAuxiliar2

Donde la variable valoresAlfabeto tiene asignado el valor de 3 correspondiente a la posición cero y la variable parametros tiene asignado el valor de 4, correspondiente a la posición 1.

En la estructura repetitiva interna for el vector V se llenará parámetros-1 veces con el valor almacenado en la variable valoresAlfabeto partiendo de la posición pos donde quedó la iteración anterior. El vector V queda definido hasta el momento como se observa en la Figura 9.

4	3		
Pos *0*	*1*	*2*	*3*

Figura 9 Vector V con valor almacenado en la posición uno

El proceso nuevamente se repite tomando ahora la cadena “2^2”. Se divide con base en el carácter '^', generando a vectorAuxiliar2, representado como aparece en la Figura 10.

2	2
i *0*	*1*

Figura 10 Tercera Representación de vectorAuxiliar2

Donde la variable valoresAlfabeto tiene asignado el valor de 2 obtenido de la posición cero y la variable parametros tiene asignado el valor de 2, obtenido de la posición 1.

Finalmente, y después de realizar la última iteración, el vector V queda definido como aparece en la Figura 11.

4	3	2	2
pos *0*	*1*	*2*	*3*

Figura 11 Vector V lleno

Procedimiento llenarCAConGredy

La Tabla 15 presenta el procedimiento llenarCAConGreedy el cual permite llenar el Covering Array a través de un método Greedy que busca agregar filas a la matriz del CA que son diferentes a las que ya se han incluido previamente, esto se realiza seleccionando de un conjunto de renglones candidatos generados al azar, el más diferente a los que ya se han incluido en la matriz del CA. El renglón más diferente es aquel que tiene una distancia de Hamming mayor, lo que significa que tiene menor similitud.

El procedimiento inicia declarando en la línea 1 a renglonElegido, un vector de tamaño k (número de parámetros definidos en la configuración del CA) el cual será utilizado para llenar cada fila de la matriz del CA hasta completar el total de N filas

de la matriz del CA que se espera construir. En la línea 2 se declara la lista ListaRenglonesCandidatos que almacena vectores candidatos para poblar la matriz del CA y que se usa a partir del segundo renglón que será incluido en la matriz del CA.

De la línea 5 a la línea 7, se crea la primera fila o renglón de la matriz del CA de forma aleatoria. Donde cada posición j de renglonElegido se llena con un número aleatorio escogido entre los valores que toma cada posición j del vector V, el cual define el alfabeto de cada columna de la matriz del CA. Luego, en la línea 20 se almacena el primer renglón en la matriz del CA.

```
     procedimiento llenarCAConGreedy (Armonia armonia, entero filasCandidatas)
1        Vector renglonElegido [armonia.CA.K]
2        Lista<Vector> listaRenglonesCandidatos
3        for i ← 0 to armonia.CA.N-1 do
4            if i = 0 then
5                for j ← 0 to armonia.CA.K-1 do
6                    renglonElegido [j] ←Random(armonia.CA.V[j])
7                end_for
8            else
9                for j ← 0 to filasCandidatas-1 do
10                   for j ← 0 to k-1 do
11                       Lista<Vector, entero> renglonAuxiliar
12                       renglonAuxiliar.vectorRenglon[j] ←Random(armonia.CA.V[j])
13                   end_for
14                   calcularSimilitudes(renglonAuxiliar, armonía.CA.matriz)
15                   ListaRenglonesCandidatos.Adicionar(renglonAuxiliar)
16               end_for
17               ListaRenglonesCandidatos.Ordenar()
18               renglonElegido←ListaRenglonesCandidatos.PrimerRenglon()
19           end_if
20           armonia.CA.matriz.adicionarRenglon(renglonElegido)
21       end_for
     fin_procedimiento
```

Tabla 15 Procedimiento llenarCAConGreedy

A partir de la línea 9 hasta la línea 19 se utiliza un método Greedy que permite que las siguientes filas (de la segunda en adelante) incluidas en la matriz del CA sean lo más diferentes posible a las previamente almacenadas.

El número de filas candidatas a crear entre las líneas 10 y 16, está dado por la variable filasCandidatas que es un valor determinado por el usuario y que llega como parámetro al método llenarCAConGreedy. En la línea 11 se declara a

renglonAuxiliar, una lista con un vector denominado vectorRenglon de tamaño K y un número entero denominado similitud.

De la línea 10 a la 13 se llena aleatoriamente el vector vectorRenglon, basado también en los valores que toma cada posición del vector V (alfabeto de cada columna de la matriz del CA). Una vez lleno, en la línea 14 se envía la lista renglonAuxiliar junto con la matriz del CA al método calcularSimilitudes que calcula qué tan similar es cada vectorRenglon con las filas ya almacenadas en la matriz del CA y fija dicho valor en el campo similitud de la lista. Como se mencionó esta similitud se basa en la distancia de Hamming del vectorRenglon y los renglones incluidos previamente en la matriz del CA.

En la línea 15 se adiciona renglonAuxiliar a la lista de vectores denominada ListaRenglonesCandidatos. Una vez se llene la lista con todos los renglones candidatos generados, se procede a ordenarla y se obtiene el primer elemento de la lista, que será aquél que menos similitud (mayor distancia) tuvo con los renglones previamente incluidos en la matriz del CA. Este renglón elegido entra a ser parte de la matriz del CA en la línea 20. Este proceso se repite N-1 veces, donde N es el número de fila que se espera tenga la matriz del CA.

Por ejemplo, si se quiere construir un CA con configuración N4K3V2^3t2.ca con número de filas N = 4, alfabeto *v* = {0,1} para todas las columnas o parámetros, número de parámetros K = 3 y fuerza *t* =2. Entonces el vector V queda definido como V = [2, 2, 2] ya que cada uno de los 3 parámetros (columnas de la matriz del CA) puede tomar dos valores, 0 o 1.

Para poblar el primer renglón (fila 0) de la matriz del CA, se genera aleatoriamente un renglonElegido con los valores definidos en el vector *V* y se almacena directamente en la matriz del CA (ver Figura 12).

Figura 12 Generación renglonElegido para i = 0

Para el segundo renglón en adelante (fila > 0), se genera aleatoriamente a vectorRenglon, el vector de la lista renglonAuxiliar. Posteriormente se calcula el número de similitudes que hay con el renglón o renglones ya registrados en la matriz del CA y se fija dicho valor al campo similitud de la lista renglonAuxiliar. Luego, se procede a guardar a renglonAuxiliar en la lista ListaRenglonesCandidatos (ver Figura 13).

Posteriormente, se generan los otros vectores vectorRenglon, se calcula su similitud con base en los renglones que se encuentran en la matriz del CA y se adiciona cada renglonAuxiliar a la lista ListaRenglonesCandidatos que tiene definido para el ejemplo el número de filasCandidatas en 5 (ver Figura 14).

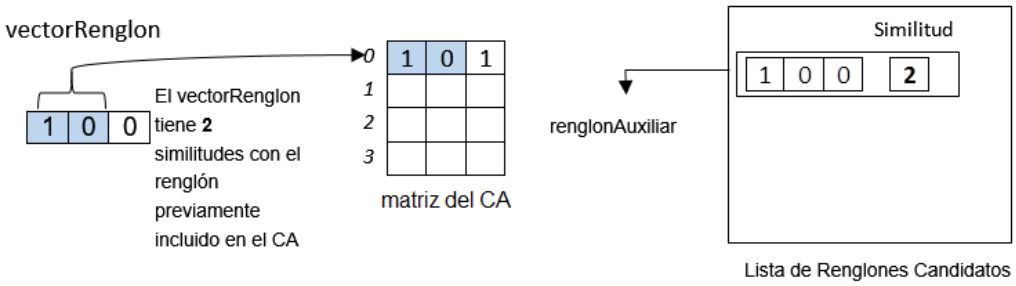


Figura 13 Adición renglonAuxiliar a ListaRenglonesCandidatos

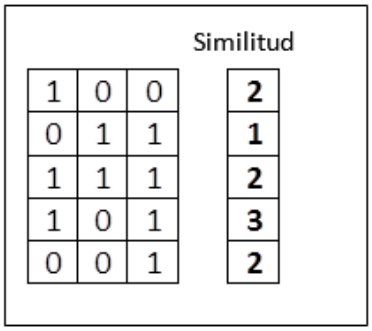


Figura 14 ListaRenglonesCandidatos llena

Finalmente, se ordena la lista ListaRenglonesCandidatos por el campo similitud de cada renglonAuxiliar, se obtiene el vector del primer elemento que será el RenglonElegido y se adiciona a la matriz del CA (ver Figura 15). El proceso se repite hasta generar todos los renglones de la matriz del CA.

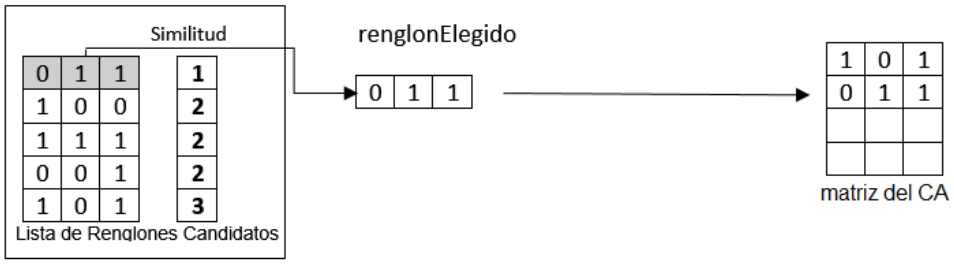


Figura 15 renglonElegido almacenado en el CA

Procedimiento para calcular el Fitness de CA

La Tabla 16 muestra el procedimiento calcularFitnessCA, el cual permite calcular las t-adas o ceros faltantes para que la matriz que representa el CA sea realmente el CA que se está buscando.

```
    procedimiento calcularFitnessCA (Armonia armonia)
1       construirP(armonia)
2       llenarMatrizP(armonia)
3       armonia.Fitness ←armonia.P.contarCerosenP( )
    fin_procedimiento
```

Tabla 16 Procedimiento calcularFitnessCA

Para calcular el fitness del CA se hace necesario construir el objeto P y sus elementos internos. Dicho objeto se construye en la línea 1 llamando al procedimiento construirP que se muestra en la Tabla 18. En la línea 2, la matrizP registrará, para cada combinación de parámetros, todas las repeticiones de una misma combinación de valores del alfabeto. El procedimiento llenarMatrizP que se muestra en la Tabla 24. En la línea 3, se cuenta cuántos ceros hay registrados en la matrizP, dicho valor representa al fitness de la matriz del CA dentro de la armonía y corresponde a las t-adas faltantes para que se encuentre el CA requerido. La función contarCerosenP se detalla en la Tabla 26.

El siguiente ejemplo muestra la forma de calcular el fitness para una configuración de CA específica.

La Tabla 17, presenta un CA de configuración N4K3V2^3t2.ca, descrito por número de filas N = 4, número de parámetros K = 3, donde cada parámetro representa una

columna en la matriz del CA y puede tomar un valor binario entre 0 y 1. La fuerza de interacción está definida como t = 2.

1	1	0
0	1	0
0	0	0
0	1	1

Tabla 17 CA de configuración N4K3V2^3t2.ca

La Figura 16 muestra la MatrizP asociada al CA. Cada fila de la MatrizP representa una combinación de parámetros o columnas de la matriz del CA, aplicada una fuerza t. Como hay K= 3 parámetros (p_0, p_1, p_2) y fuerza de interacción t = 2, las combinaciones resultantes de los parámetros están dada por el conjunto = {(p_0, p_1), (p_0, p_2), (p_1, p_2)} que se resume como {(0,1), (0,2), (1,2)}, conformando las filas de la matrizP. Cada columna en la MatrizP registra todas las combinaciones sin repetición que se puedan dar entre los alfabetos de los parámetros relacionados, aplicada una fuerza t.

Dado a que cada parámetro tiene un alfabeto binario entre 0 y 1, se identificarán todas las combinaciones posibles y sin repetición de los valores del alfabeto para cada interacción de parámetros. La combinación de los valores del alfabeto está dada por el conjunto = {(0,0), (0,1), (1,0), (1,0)}.

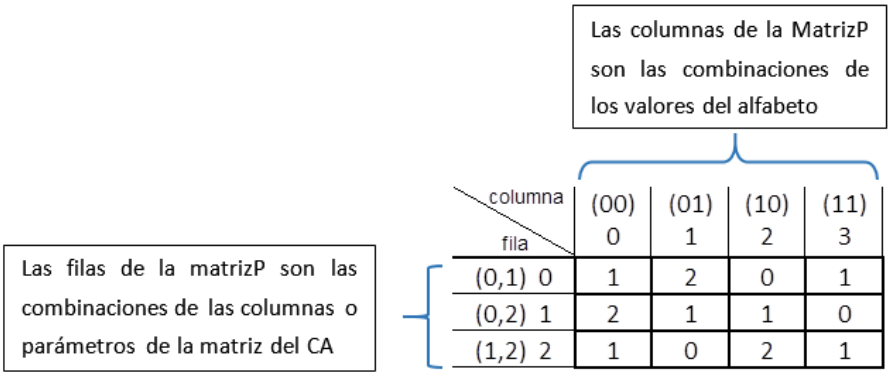
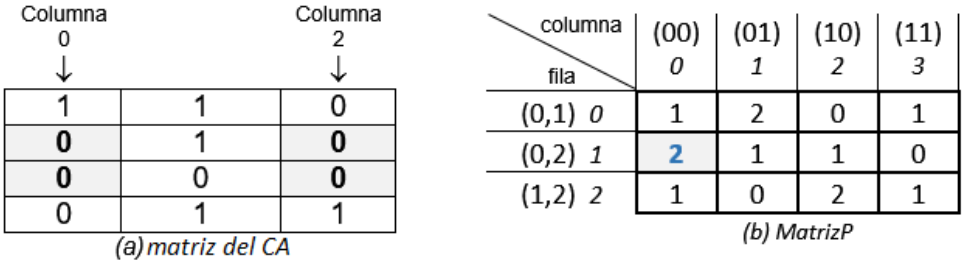


Figura 16 matrizP asociada a la configuración N4K3V2^3t2.ca

En la Figura 17(a) se muestra que al combinar el parámetro 0 y el parámetro 2 de la matriz del CA (representados como la columna 0 y la columna 2 de la matriz del CA), aparece 2 veces las combinaciones del alfabeto 0 y 0 (en fondo gris). El la

Figura 17(b) el valor de 2 es registrado en la matrizP en la fila 1, que representa la combinación de la columna 0 y 2 de la matriz del CA y columna 0, que representa la combinación del alfabeto (00).



Columna 0 ↓		Columna 2 ↓
1	1	0
0	1	**0**
0	0	**0**
0	1	1

(a) matriz del CA

columna / fila	(00) 0	(01) 1	(10) 2	(11) 3
(0,1) 0	1	2	0	1
(0,2) 1	**2**	1	1	0
(1,2) 2	1	0	2	1

(b) MatrizP

Figura 17 CA de configuración N4K3V2^3t2.ca y MatrizP asociada

Los ceros registrados en la MatrizP representan las t-adas faltantes o combinaciones del alfabeto faltantes para determinadas combinaciones de t-columnas. El fitness del CA se define como el número total de ceros registrados en la MatrizP. Para el ejemplo anterior, el fitness es igual a 3.

Procedimiento construirP

La Tabla 18 muestra el procedimiento construirP. La construcción del objeto P involucra internamente la construcción de otros elementos a saber: el número entero MaxJ, la matriz de enteros J, el vector de enteros Vt y una matriz de enteros denominada matrizP.

```
    procedimiento construirP(Armonia armonia)
1       MaxJ ← Combinatoria (armonia.CA.K, armonia.CA.t)
2       construirJ(armonia, armonia.CA.K, armonia.CA.t)
3       construirVt(armonia)
4       definirMatrizP(armonia)
5       inicializarMatrizP(armonia)
    fin_procedimiento
```

Tabla 18 Procedimiento construirP

En la línea 1, se define la variable entera MaxJ, que almacena el número de combinaciones posibles de los K parámetros, dada una fuerza t. En la línea 2, se construye a J, una matriz que almacena en cada fila, un vector de tamaño t con las combinaciones de los K parámetros, aplicada una fuerza de interacción t. En la línea 3, se construye a Vt, un vector que almacena en cada columna el número de combinaciones del alfabeto generadas para una interacción de parámetros

específica. En la línea 4, se define a matrizP, una matriz que se construye con base en el valor MaxJ y el vector Vt. En la línea 5, se inicializa a P en un valor por defecto, que es cero para todas las celdas de la matriz.

A continuación, se describe la utilidad y construcción de cada elemento del objeto P.

Definición de MaxJ

MaxJ representa el número de combinaciones resultantes de los K parámetros, aplicada una fuerza t. Se calcula con base en una combinatoria entre k y t y está dada por la Ecuación 1:

$$C_{k,t} = \binom{k}{t} = \frac{k!}{(k-t)! \times t!}$$

Ecuación 1 Cálculo del número combinatorio $C_{k,t}$

Por ejemplo, el CA de configuración N4K3V2^3t2.ca tiene definido K = 3 parámetros, donde k = {p_0, p_1, p_2} y se aplicará sobre ellos una fuerza de interacción t = 2. El cálculo del número de combinaciones resultantes se obtiene al aplicar la Ecuación 2.

$$C_{3,2} = \binom{3}{2} = \frac{3!}{(3-2)! \times 2!} = 3$$

Ecuación 2 Cálculo del número combinatorio $C_{3,2}$

MaxJ será igual a 3 que indica todas las combinaciones resultantes y sin repetición de los 3 parámetros con fuerza t = 2. Para este caso (p_0, p_1), (p_0, p_2) y (p_1, p_2).

La Tabla 19 muestra la función Combinatoria que permite calcular el valor de MaxJ.

```
    entero función Combinatoria (entero k, entero t)
1       comb ←1.0
2       menor ← k – t
3       if t < menor then
4           menor ← t
5       end_if
6       for i ← 1 to menor do
7           comb ← (k * 1.0 / i) * comb
8           k ← k -1
9       end_for
10      return (entero)comb
    fin_función
```

Tabla 19 Función Combinatoria

Procedimiento construirJ

En la matriz J, cada una de sus filas almacena un vector que representa la combinación de los K parámetros con una fuerza de interacción aplicada t. Los elementos de la matriz J se encuentran directamente relacionados con los índices (posiciones) de las filas de la matriz P. La Tabla 20 muestra el procedimiento que permite construir la matriz J.

```
    procedimiento construirJ( Armonia armonia, entero k, entero t)
1       pos←0
2       Matriz<entero> J [armonia.P.MaxJ][ ]
3       lineaDeJ [t]
4       for i←0 to t-1 do
5           lineaDeJ[i] ← i
6       end_for
7       J[pos]←lineaDeJ
8       pos←pos + 1
9       iMax←t-1
10      for i←0 to t-1 do
11          if lineaDeJ[i] == k – t + i then
12              iMax←i
13              Break
14          end_if
15      end_for
16      Do
17          lineaDeJ[t-1]← lineaDeJ[t-1] + 1
18          if lineaDeJ[t-1] == k then
19              if iMax == 0 then
20                  Break
21              end_if
22              lineaDeJ[iMax - 1]← lineaDeJ[iMax - 1] + 1
23              for i←iMax to t-1 do
24                  lineaDeJ[i] ← lineaDeJ[i - 1] + 1
25              end_for
26              if lineaDeJ[iMax - 1] == k - t + iMax – 1 then
27                  iMax ← iMax – 1
28              Else
29                  iMax ← t – 1
30              end_if
31          end_if
32          J[pos]←lineaDeJ
33          pos←pos + 1
34      while(true)
35      armonia.P.J ←J
    fin_procedimiento
```

Tabla 20 Procedimiento construirJ

Por ejemplo, para el CA de configuración N4K3V2^3t2.ca que tiene definido K = 3 parámetros y fuerza de interacción t = 2, la matriz J tendrá MaxJ filas, donde MaxJ es 3, valor obtenido al aplicar la función Combinatoria. Ver Figura 18.

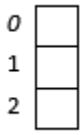


Figura 18 Matriz J vacía

LineaDeJ es un vector de tamaño t, que inicialmente se llena con los valores comprendidos entre cero y t-1. Dado a que en el ejemplo la fuerza es t = 2, LineaDeJ tendrá los valores que se muestra en la Figura 19.

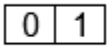


Figura 19 LineaDeJ

Posteriormente, se guarda LineaDeJ en la primera fila de la matriz J. Ver Figura 20.

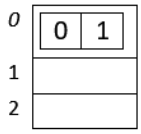


Figura 20 LineadeJ almacenada en matriz J

El algoritmo luego realiza las combinaciones sin repetición basadas en los K parámetros y fuerza t. Finalmente, la matriz J quedará definida como se observa en la Figura 21.

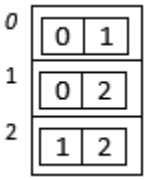


Figura 21 Matriz J llena

Procedimiento construirVt

Vt es un vector de tamaño MaxJ que almacena en cada columna el número de combinaciones del alfabeto generadas para una interacción de parámetros específica. Cada uno de sus elementos indica también el tamaño máximo de columnas que tendrá cada fila i de la matrizP.

La Tabla 21 muestra el procedimiento construirVt que permite llenar el vector Vt. Para llenar el vector Vt se hace uso del vector V del CA que se llenó previamente a través del procedimiento presentado en la Tabla 14. El vector V almacena en cada posición el número de valores que toma cada parámetro K de la matriz del CA.

```
    procedimiento construirVt(Armonia armonia)
1       Vector<entero> Vt[armonia.P.MaxJ]
2       for i ←0 to armonia.P.MaxJ-1 do
3           mul ←1
4           for j←0 to armonia.t-1 do
5               mul ← mul * armonia.CA.V[armonia.P.J[i]]
6           end_for
7           armonia.P.Vt[i] ← mul
8       end_for
9       armonia.P.Vt←Vt
    fin_procedimiento
```

Tabla 21 Procedimiento construirVt

El siguiente ejemplo, muestra la generación del vector Vt para un CA de configuración N12K4V4^1-3^1-2^2t2.ca. Para dicha configuración, el número MaxJ tiene el valor de 6, la matriz J está definida como se muestra en la Figura 22 y el vector V está definido como se muestra en la Figura 23.

0	0	1
1	0	2
2	0	3
3	1	2
4	1	3
5	2	3

Figura 22 Matriz J para el CA N12K4V4^1-3^1-2^2t2.ca

	4	3	2	2
pos	*0*	*1*	*2*	*3*

Figura 23 vector V para el CA N12K4V4^1-3^1-2^2t2.ca

Para ampliar la interpretación del vector V se hace uso de la representación de la Figura 24. En ella se observa que las columnas (parámetros) no tienen siempre el mismo alfabeto.

parámetro / *valores*	*p0*	*p1*	*p2*	*p3*
	0	0	0	0
	1	1	1	1
	2	2		
	3			

Figura 24 Interpretación del Vector V

El parámetro *p0*, que está representado en el vector V en la posición cero tiene un valor de 4 y hace referencia a que dicho parámetro toma 4 valores del alfabeto (0, 1, 2 y 3). El parámetro *p1,* representado con el parámetro uno en el vector V tiene un valor de 3, que hace referencia a que dicho parámetro toma 3 valores del alfabeto (0, 1 y 2). Los parámetros *p2 y p3* toman cada uno de ellos 2 valores del alfabeto (0 y 1).

En la Figura 22 cada fila de la matriz J, representa las combinaciones que se pueden hacer con los K = 4 parámetros de la matriz del CA. La Figura 25 muestra el vector almacenado en la posición cero de la matriz J. Donde la posición cero de dicho vector hace referencia al parámetro p0 y la posición uno, hace referencia al parámetro p1.

0	0	1

Figura 25 Vector almacenado en la posición i = 0 Matriz J

Teniendo en cuenta el vector V de la Figura 23, el parámetro p0 toma 4 valores del alfabeto y el parámetro p1 toma 3 valores del alfabeto. Por lo tanto, se realizan todas las combinaciones sin repetición, a las que haya lugar entre los valores del parámetro p0 y los valores del parámetro p1 (ver Figura 26).

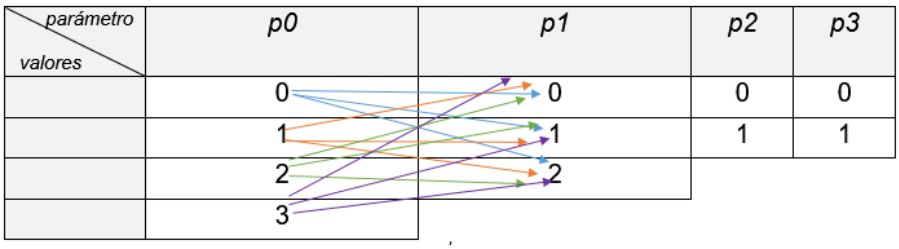


Figura 26 Combinaciones del parámetro 0 y parámetro 1

En la figura anterior, las flechas representan todas las combinaciones de valores que se pueden hacer entre las columnas p0 y p1. En total 12, generando las siguientes parejas: 00, 01, 02, 10, 11, 12, 20, 21, 22, 30, 31 y 32.

En la posición i = 0 del vector Vt se registra el valor de 12, que indica que entre los parámetros p0 y p1 se han generado doce combinaciones de valores. El proceso se repite para cada fila de la matriz J y se registra el valor en el vector Vt. Finalmente, el procedimiento construirVt, permite que el vector Vt queda definido como se muestra en la Figura 27.

	12	8	8	6	6	4
pos	*0*	*1*	*2*	*3*	*4*	*5*

Figura 27 Vector Vt lleno

Procedimiento definirMatrizP

La matrizP es una matriz donde el número de filas está definido por MaxJ y donde el número de columnas para cada fila, se crea de acuerdo al valor de cada elemento del vector Vt. La Tabla 22, muestra el procedimiento definirMatrizP que define el tamaño para la matrizP con base en el valor MaxJ y el vector Vt.

```
    procedimiento definirMatrizP(Armonia armonia)
1       Matriz<entero> matrizP[armonia.P.MaxJ][ ]
2       for i ←0 to armonia.P.MaxJ-1 do
3           matrizP[i] ← new [armonia.P.Vt[i]]
4       end_for
5       armonia.P.matrizP←matrizP
    fin_procedimiento
```

Tabla 22 Procedimiento definirMatrizP

La Figura 28, representa gráficamente como queda definida la matrizP para el CA de configuración N12K4V4^1-3^1-2^2t2.ca. Teniendo en cuenta que K = 4, las combinaciones de columnas con fuerza 2 son cinco, a saber: {(0,1), (0,2), (0,3), (1,2), (1,3) (2,3)}.

Para la primera fila se tienen 12 espacios porque los valores de la columna 0 y 1 son de 4 y 3 que al combinarlo genera 12 parejas como se explicó previamente. Para la segunda fila como se combinan las columnas 0 y 2 cuyos alfabetos son 4 y 2 se obtienen 8 parejas (4 x 2) y así sucesivamente con cada fila de la matriz.

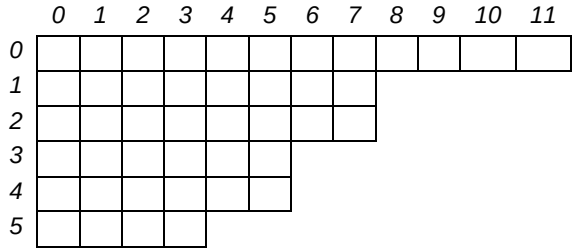


Figura 28 Representación de la matriz P

Procedimiento inicializarMatrizP

La Tabla 23 muestra el procedimiento que inicializa la matrizP con ceros.

```
    procedimiento inicializarMatrizP(Armonia armonia)
1   for i ←0 to armonia.P.MaxJ-1 do
2       for j ←0 to armonia.P.Vt[i]-1 do
3           armonia.P.matrizP[i][j] ←0
4       end_for
5   end_for
    fin_procedimiento
```

Tabla 23 Procedimiento inicializarMatrizP

Procedimiento para llenarMatrizP

La Tabla 24 muestra el procedimiento que permite llenar la matrizP. La matrizP se va llenando de acuerdo a las ocurrencias que vayan apareciendo de una combinación específica de valores del alfabeto para una combinación de parámetros específica en la matriz del CA.

```
    procedimiento llenarMatrizP(Armonia armonia)
1       for i ←0 to armonia.P.MaxJ-1 do
2           for j ←0 to armonia.CA.N-1 do
3             columnaEnP ←divisionSintetica(armonia.CA.matriz[j], armonia.P.J[i],
                              armonia.CA.V, armonia.CA.t)
4             if columnaEnP < > -1 then
5               armonia.P.matrizP[i][columnaEnP] ← armonia.P.matrizP [i][columnaEnP]+1
6             end_if
7           end_for
8       end_for
    fin_procedimiento
```

Tabla 24 Procedimiento llenarMatrizP

El procedimiento llenarMatrizP hace uso de la función divisionSintetica, la cual dada una combinación del alfabeto y la combinación de parámetros que se están evaluando, devuelve el número de columna asociada en la matrizP donde aparece relacionada dicha combinación de alfabeto. La función divisionSintetica se muestra en la Tabla 25.

```
    entero función divisionSintetica(entero filaCA[], entero filaJ[], entero V[], entero t )
1       suma←filaCA[filaJ[0]]
2       for i ←1 to t-1 do
3           suma← suma * V[filaJ[i]] + filaCA[filaJ[i]]
4       end_for
5       retornar suma
    fin_función
```

Tabla 25 Función divisionSintetica

En el siguiente ejemplo se explica la forma cómo se llena la matrizP. La Figura 29, muestra un CA de configuración N12K4V4^1-3^1-2^2t2.ca.

0	0	0	0	1
1	1	1	1	1
2	3	0	1	0
3	1	2	0	0
4	1	2	1	0
5	3	1	0	0
6	2	0	0	0
7	3	2	1	0
8	2	1	0	1
9	1	2	0	1
10	1	1	1	0
11	3	0	0	0

Figura 29 CA de configuración N12K4V4^1-3^1-2^2t2

La Matriz J está definida como se muestra en la Figura 30. Observe que el CA tiene 4 parámetros (columnas) y es de fuerza 2.

0	0	1
1	0	2
2	0	3
3	1	2
4	1	3
5	2	3

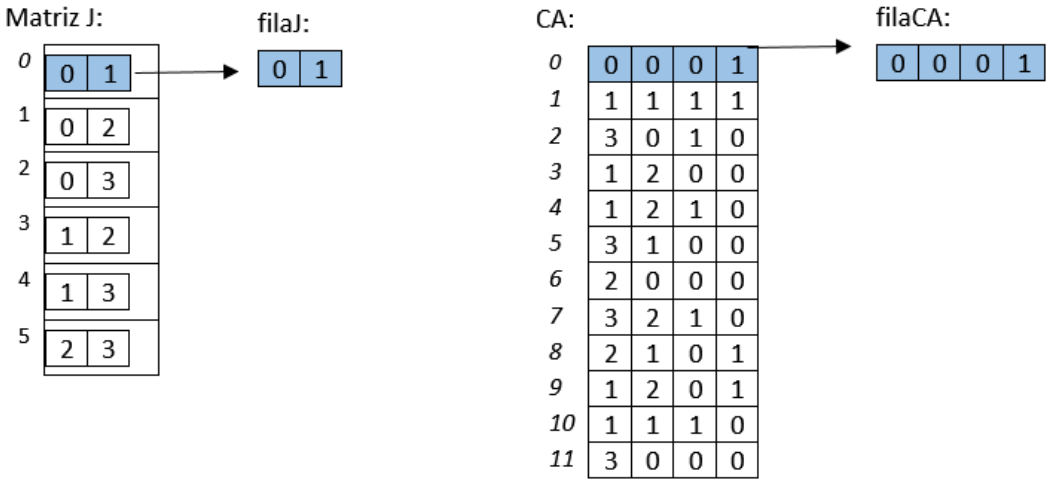
Figura 30 Matriz J del CA de configuración N12K4V4^1-3^1-2^2t2

El Vector V está definido como se muestra en la Figura 31.

	4	3	2	2
pos	*0*	*1*	*2*	*3*

Figura 31 Vector V para el CA de configuración N12K4V4^1-3^1-2^2t2

El valor de MaxJ es igual a 6 y fuerza t = 2. El algoritmo inicia seleccionando cada fila de la matriz J y cada fila j de la matriz del CA. Para la primera iteración se obtiene fila i = 0 de la matriz J denominada filaJ y la fila j = 0 de la matriz del CA denominada FilaCA. Ver Figura 32.



0	0	1		
1	0	2		
2	0	3		
3	1	2		
4	1	3		
5	2	3		

0	0	0	0	1
1	1	1	1	1
2	3	0	1	0
3	1	2	0	0
4	1	2	1	0
5	3	1	0	0
6	2	0	0	0
7	3	2	1	0
8	2	1	0	1
9	1	2	0	1
10	1	1	1	0
11	3	0	0	0

Figura 32 Obtención de filaJ y filaCA para i = 0 y j = 0

filaJ representa la combinación de parámetros de la matriz del CA que se analizará, para este caso se referencia el parámetro 0 (columna 0 de la matriz del CA) y el parámetro 1 (columna 1 de la matriz del CA). Luego, dichos vectores se pasan cómo parámetro a la función *divisionSintetica* junto con el vector V y la fuerza t.

Hay que tener en cuenta, que la posición 0 del vector *V* indica que el parámetro 0 representado por la columna 0 de la matriz del CA toma 4 valores y la posición 1 del vector V indica que el parámetro 1 representado en la columna 1 de la matriz del CA toma 3 valores (ver Figura 33). El parámetro *0* denotado por *p0* tomará entonces los valores *p0* = {0, 1, 2, 3} y el parámetro *1* denotado por *p1* tomará los valores *p1* = {0, 1, 2}. De tal forma que, si se combina *p0* y *p1*, se obtienen 12 combinaciones de valores sin repetición, generando las siguientes parejas: 00, 01, 02, 10, 11, 12, 20, 21, 22, 30, 31, 32.

4	3	2	2
pos 0	*1*	*2*	*3*

Figura 33 Vector V con posición 0 y 1

En la Figura 34, se observa como cada fila i de la matrizP es el elemento i de la matriz J. Cada una de las columnas de la matriz P tiene correspondencia directa con cada una de las parejas que se formaron al combinar el parámetro *p0* y el parámetro *p1,* representados como el vector [0,1] en la fila i = 0 de la matriz J.

	Columna→	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
	Pareja→	***00***	***01***	***02***	***10***	***11***	***12***	***20***	***21***	***22***	***30***	***31***	***32***
0	***0*** ***1***	0	0	0	0	0	0	0	0	0	0	0	0
1	*0* *2*	0	0	0	0	0	0	0	0				
2	*0* *3*	0	0	0	0	0	0	0	0				
3	*1* *2*	0	0	0	0	0	0						
4	*1* *3*	0	0	0	0	0	0						
5	*2* *3*	0	0	0	0								

Figura 34 parejas generadas para la fila i = 0 de la matriz P

FilaCA está definida como se muestra en la Figura 35 y la FilaJ está definida como se muestra en la Figura 36.

0	0	0	1
0	*1*	*2*	*3*

Figura 35 FilaCA

0	1

Figura 36 FilaJ

Se analiza entonces qué pareja se forma en FilaCA teniendo en cuenta los valores definidos en FilaJ. Como en FilaJ aparecen los valores 0 y 1, se mira en filaCA que pareja se forma en las posiciones 0 y 1 (ver Figura 37).

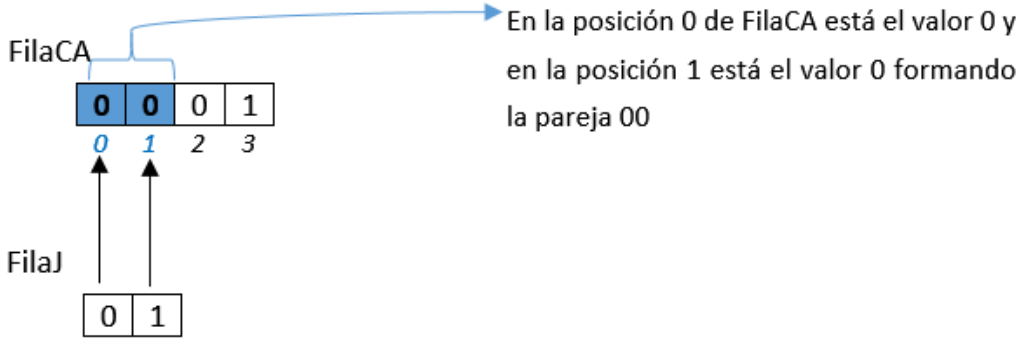


Figura 37 Pareja obtenida en FilaCA con base en FilaJ para fila i = 0 del CA

La función divisionSintetica definida en la Tabla 25, tiene como objetivo retornar el número de columna en la MatrizP que coincide con la pareja formada. Si se observa la Figura 34, la pareja 00 corresponde a la columna 0 de la MatrizP en dicha combinación de columnas. De tal manera que el valor de 0 será retornado por la función divisionSintetica y almacenado en la variable columnaEnP. Luego, se incrementa en uno el valor registrado en las posiciones de la matrizP[i][columnaEnP], donde i = 0 y columnaenP = 0 (ver Figura 38).

		Columna→	***0***	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
		Pareja→	*00*	*01*	*02*	*10*	*11*	*12*	*20*	*21*	*22*	*30*	*31*	*32*
0	*0 1*		**1**	0	0	0	0	0	0	0	0	0	0	0
1	*0 2*		0	0	0	0	0	0	0	0				
2	*0 3*		0	0	0	0	0	0	0	0				
3	*1 2*		0	0	0	0	0	0						
4	*1 3*		0	0	0	0	0	0						
5	*2 3*		0	0	0	0								

Figura 38 Primer valor incrementado en la matrizP[0][0]

El proceso continúa con el ciclo interno del procedimiento ahora con j = 1. Donde filaJ se mantiene igual con los valores 0 y 1 y filaCA será ahora la fila 1 del CA. Ver Figura 39.

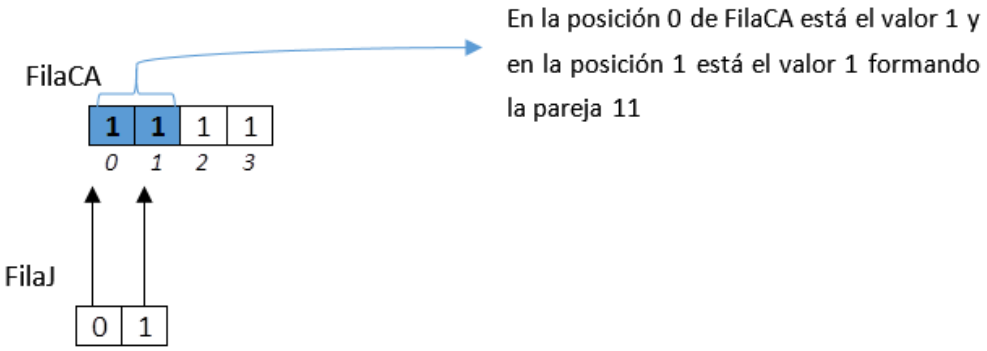


Figura 39 Pareja obtenida en FilaCA con base en FilaJ para fila = 1 del CA

Si se observa en la Figura 40, la pareja 11 corresponde a la columna 4 de la MatrizP. El valor de 4 será retornado por la función divisionSintetica y almacenado en la variable columnaEnP. Luego, se incrementa en uno el valor registrado en las posiciones de la matriz P[i][columnaEnP], donde i = 0 y columnaenP = 4.

	Columna→	0	1	2	3	**4**	5	6	7	8	9	10	11
	Pareja→	00	01	02	10	11	12	20	21	22	30	31	32
0	0 1	1	0	0	0	**1**	0	0	0	0	0	0	0
1	0 2	0	0	0	0	0	0	0	0				
2	0 3	0	0	0	0	0	0	0	0				
3	1 2	0	0	0	0	0	0						
4	1 3	0	0	0	0	0	0						
5	2 3	0	0	0	0								

Figura 40 Segundo valor incrementado en la matriz P[0][4]

La Figura 41, muestra cómo queda la matrizP después de haber terminado el ciclo interno (líneas 2 a 7) del procedimiento llenarMatrizP, cuando se han recorrido todas las filas de la matriz del CA para la primer combinación de columnas en la matriz J o la primer fila de P. Quedan así registradas en la matrizP todas las ocurrencias de parejas que aparecieron al combinar los parámetros indicados en la fila i = 0 de la matrizP, en este caso el parámetro 0 y parámetro 1.

	Columna→	0	1	2	3	4	5	6	7	8	9	10	11
	Pareja→	00	01	02	10	11	12	20	21	22	30	31	32
0	0 1	**1**	**0**	**0**	**0**	**2**	**3**	**1**	**1**	**0**	**2**	**1**	**1**
1	0 2	0	0	0	0	0	0	0	0				
2	0 3	0	0	0	0	0	0	0	0				
3	1 2	0	0	0	0	0	0						
4	1 3	0	0	0	0	0	0						
5	2 3	0	0	0	0								

Figura 41 columnas llenas para la fila i = 0 de la Matriz P

Para la siguiente iteración se evalúa la fila i = 1 de la matriz J con cada una de las filas de la matriz del CA (ver Figura 42).

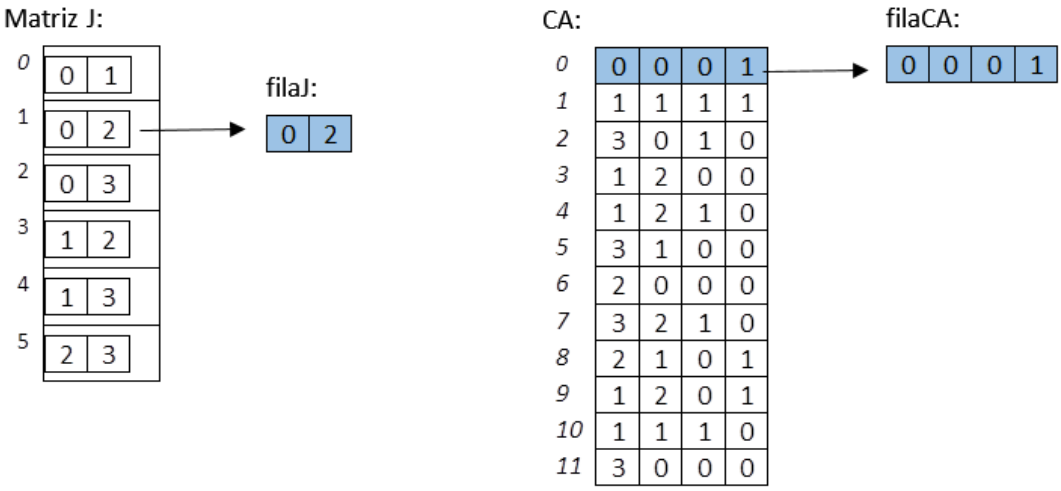


Figura 42 Obtención de filaJ y filaCA para i = 1 y j = 0

La filaJ indica ahora que se trabajará con el parámetro 0 y el parámetro 2 de la matriz del CA. Por lo tanto, en el vector V se identifica el número de valores que toman cada uno de dichos parámetros (ver Figura 43).

pos	0	1	2	3
	4	3	2	2

Figura 43 Vector V con posición 0 y 2

El parámetro 0 (representado en la posición 0 del vector V) toma entonces los valores p0 = {0, 1, 2, 3} y el parámetro 2 (representado en la posición 2 del vector V) toma los valores p2 = {0, 1}.

De tal forma que cuando se combina p0 y p2, se obtienen 8 combinaciones de valores sin repetición, generando las siguientes parejas: 00, 01, 10, 11, 20, 21, 30,

31. En la Figura 44, Cada uno de las columnas de la matrizP tiene correspondencia directa con cada una de las parejas que se formaron al combinar el parámetro p0 y el parámetro p2, representados en la fila i = 1 de la matriz J.

	Columna→	0	1	2	3	4	5	6	7				
	Pareja→	**00**	**01**	**10**	**11**	**20**	**21**	**30**	**31**				
0	0 1	1	0	0	0	2	3	1	1	0	2	1	1
1	0 2	0	0	0	0	0	0	0	0				
2	0 3	0	0	0	0	0	0	0	0				
3	1 2	0	0	0	0	0	0						
4	1 3	0	0	0	0	0	0						
5	2 3	0	0	0	0								

Figura 44 parejas generadas para la fila i = 1 de la matriz P

El proceso continúa, donde filaJ se mantiene igual con los valores 0 y 2 y filaCA será ahora la fila 0 de la matriz del CA. Se analiza nuevamente qué pareja se forma en filaCA teniendo en cuenta los valores definidos en filaJ. Como en filaJ aparecen los valores 0 y 2, se mira en filaCA que pareja se forma en las posiciones 0 y 2, en este caso la pareja 00 (Ver Figura 45).

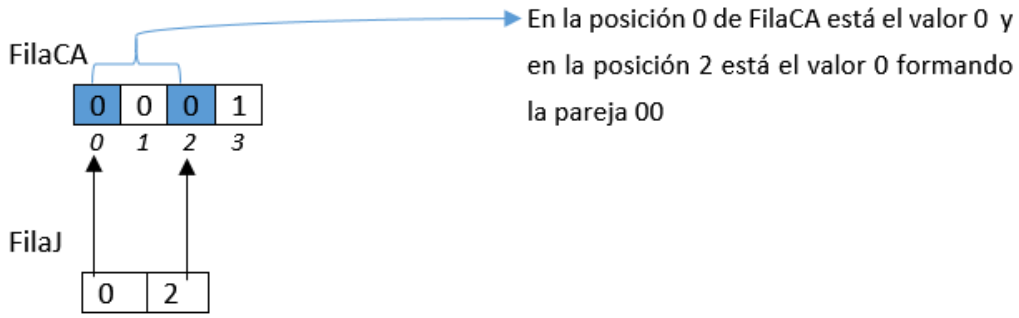


Figura 45 Pareja obtenida en FilaCA con base en FilaJ para fila = 0 del CA

Si se observa en la Figura 46, la pareja 00 corresponde a la columna 0 de la MatrizP. El valor de 0 será retornado por la función y almacenado en columnaEnP. Luego, se incrementa en uno el valor registrado en las posiciones de la matrizP[i][columnaEnP], donde i = 1 y columnaenP = 0.

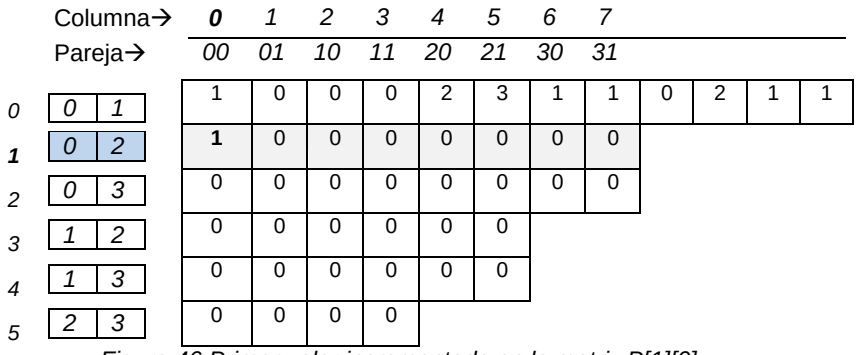


Figura 46 Primer valor incrementado en la matriz P[1][0]

La Figura 47, muestra cómo queda la matrizP después de haber terminado el ciclo interno del procedimiento llenarMatrizP, donde todas las filas de la matriz del CA se han recorrido para la combinación de columnas (parámetros) 0 y 2. Quedan así registradas en la matrizP todas las ocurrencias de parejas que aparecieron al combinar los parámetros indicados en la fila i = 1 de la matrizP, en este caso el parámetro 0 y parámetro 2.

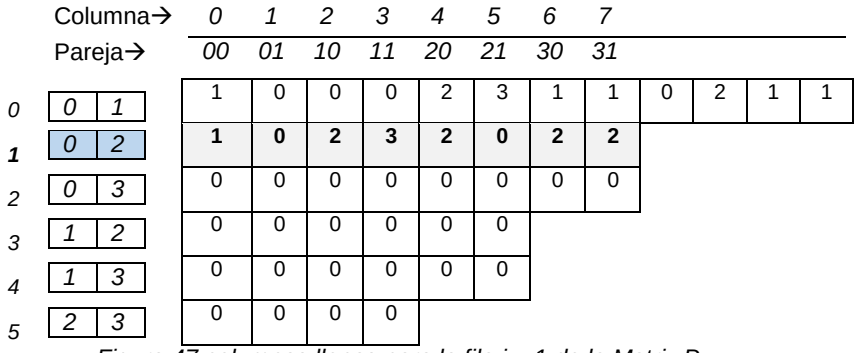


Figura 47 columnas llenas para la fila i = 1 de la Matriz P

El proceso continúa con la evaluación de cada fila de la matriz J y cada una de las filas del CA. Finalmente, la Figura 48 muestra la matrizP llena. Las celdas de P con cero son t-adas faltantes y las que tienen un valor mayor o igual a 2 significan t-adas repetidas.

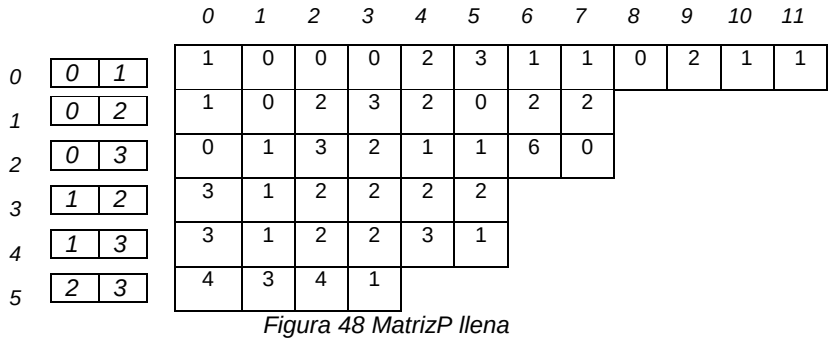


Figura 48 MatrizP llena

Función contarCerosenP

La función contarCerosenP, permite calcular cuántos ceros hay registrados en la matriz P. La función se muestra en la Tabla 26. En la línea 1, se define una variable entera denominada total que actúa como un contador y su valor se inicializa en cero. De la línea 2 a la línea 8 se recorre la matriz P de tal forma que se incrementa en uno el valor de la variable total cuando se encuentra un cero. Finalmente, en la línea 9 el valor de total es retornado para ser asignado al fitness del CA dentro de la armonía, el cual representa el número de t-adas faltantes para encontrar el CA requerido.

```
    entero función contarCerosenP()
1       total←0
2       for i←0 to armonia.P.MaxJ-1 do
3           for j←0 to armonia.CA.Vt[i]-1 do
4               if armonia.P.matrizP[i][j] = 0 then
5                   total←total+1
6               end_if
7           end_for
8       end_for
9       retornar total
    fin_función
```

Tabla 26 Procedimiento contarCerosenP

Procedimiento optimizarCAConSA

La Tabla 27 muestra el procedimiento optimizarCAconSA, el cual optimiza el CA de una armonía aplicándole el algoritmo de Recocido Simulado (Simulated annealing, SA).

```
     procedimiento optimizarCAConSA(Armonia armonia)
1        entero renglonDelCA
2        Armonia best←CopiaDe(armonia)
3        for i←0 to MaxIteracionesSA - 1 do
4            if armonia.Fitness = 0 then
5                Break
6            end_if
7            temperatura ← (1.0 * (MaxIteracionesSA - i)) / MaxIteracionesSA
8            Vector renglonOriginal[K]
9            Vector renglonOptimizado [K]
10           if Random(0,1) < 0.7 then
11               renglonOptimizado ← PMisLocal1RepetidoEnP(armonia, armonia.Fitness,
                                        out renglonDelCA)
12               renglonOriginal← armonia.CA.Matriz[renglonDelCA]
```

```
13        else
14            renglonDelCA ←Rnd(armonia.CA.N)
15            renglonOriginal← armonia.CA.Matriz[renglonDelCA]
16            renglonOptimizado ← PMisLocal2RepetidoEnTodasColumnas(armonia,
                                      renglonOriginal, armonia.Fitness)
17        end_if
18        nuevoFitness ←quitarPonerRenglon(renglonOriginal, renglonOptimizado,
                                armonia.P, armonia.CA.V, armonia.CA.t, false)
19        if nuevoFitness < best.Fitness then
20            Armonia best←CopiaDe(armonia)
21            for j ←0 to Armonia.CA.K-1 do
22                best.CA.Ca[renglonDelCA][j] ← renglonOptimizado[j]
23            end_for
24            best.Fitness←nuevoFitness
25        end_if
26        min ← minimo(nuevoFitness, armonia.Fitness, temperatura)
27        if Random(0,1) < min or nuevoFitness < armonia.Fitness then
28            for j ←0 to Armonia.CA.K-1 do
29                armonia.CA.Matriz[renglonDelCA][j] ← renglonOptimizado[j]
30            end_for
31            armonia.Fitness←quitarPonerRenglon(renglonOriginal,renglonOptimizado,
                                    P,armonia.CA.V, armonia.CA.t, true)
32        end_if
33    end_for
34    armonia ← best
35    calcularFitnessCA(armonia)
  fin_procedimiento
```

Tabla 27 Procedimiento optimizarCAConSA

En la línea 1, renglonDelCA es una variable entera que almacena la posición de una fila en la matriz Ca del CA y puede tomar un valor entre 0 y N-1. En la línea 2, se realiza una copia de la armonía original que llega como parámetro al procedimiento y se guarda en el objeto best, al cual se le procederá a optimizar su Ca.

Entre las líneas 3 y 33 y enmarcado en un número máximo de iteraciones de optimización MaxIteracionesSA se da inicio al proceso de optimización. Inicialmente en la línea 4 se pregunta si el fitness del CA dentro de la armonía original es cero, si es así el ciclo termina al igual que el proceso de optimización. Cabe resaltar que es poco probable que cuando llegue al procedimiento una armonía como parámetro, ésta ya tenga su fitness en cero, por lo general siempre habrá t-adas faltantes en los CA que se generan es decir, con fitness diferente de cero.

En la línea 7, se inicia la temperatura del algoritmo SA en 1.0 y se empieza así el proceso de enfriamiento. En las líneas 8 y 9 se crean respectivamente los vectores

de tamaño k renglonOriginal y renglonOptimizado, los cuales almacenarán una fila o renglón del CA. En la línea 10, se genera un número aleatorio entre 0 y 1, y dependiendo del valor que tome dicho número se aplicará una de las dos funciones de optimización local PMisLocal1RepetidoEnP (con una probabilidad de escogencia casi del 70%) o PMisLocal2RepetidoEnTodasColumnas (con una probabilidad de escogencia casi del 30%). Estos porcentajes pueden ser un parámetro del algoritmo, pero se fijaron así porque son los que mejores resultados reportaron en la experimentación. Las funciones de optimización local tienen como objetivo generar un vecino que se representa en el vector denominado renglonOptimizado.

Si el número aleatorio es menor de 0.7, en la línea 11 se asigna al vector renglonOptimizado la fila de la matriz dentro del CA que ha sido optimizada a través de la función de optimización local PMisLocal1RepetidoEnP (que se explica a detalle en una sección posterior y se referencia en la Tabla 30). Esta función de optimización toma cada fila de la matriz del CA, le hace un cambio con base en unas posiciones y valores específicos. Luego, calcula el fitness para el renglón modificado, lo compara con el fitness original y después de un ciclo iterativo, finalmente retorna el renglón que permitió obtener el mejor fitness. La función PMisLocal1RepetidoEnP tiene un parámetro de salida que guarda la posición del renglón optimizado. En la línea 12, en el vector renglonOriginal se guarda aquel renglón o fila de la matriz dentro del CA que se encuentre en la posición renglonDelCA.

En caso contrario, si el número aleatorio es mayor o igual de 0.7, en la línea 14 se guarda en la variable renglonDelCA la posición de una fila de la matriz del CA seleccionada aleatoriamente. En la línea 15, en el vector renglonOriginal se almacena esa fila de la matriz dentro del CA que está en la posición renglonDelCA. En la línea 16, en el vector renglonOptimizado se guarda la fila que ha sido optimizada a través de la función de optimización local PMisLocal2RepetidoEnTodasColumnas (que se explica a detalle en una sección posterior y se referencia en la Tabla 31. Esta función va calculando el fitness de cada valor del alfabeto que toma una posición específica del renglón a analizar y lo va comparando con el fitness de un renglón original. El Renglón con el mejor fitness es retornado por la función.

En la línea 18, la variable entera nuevoFitness almacena el fitness o número de ceros registrados en la matrizP después de aplicar la función denominada QuitarPonerRenglon. La función quita de la matrizP las ocurrencias o repeticiones

del alfabeto a las que da lugar el quitar el renglonOriginal. Luego, adiciona a la matrizP las ocurrencias o repeticiones del alfabeto a las que da lugar el poner el renglonOptimizado. Posteriormente se recalcula cuántos ceros quedan en la matrizP después de quitar el renglonOriginal y poner el renglonOptimizado, ese nuevo fitness es el retornado por la función. La función QuitarPonerRenglon se explicará en una sección posterior y se referencia en la Tabla 28.

Cabe resaltar que el último parámetro que recibe la función quitarPonerRenglon es un valor booleano, que tiene valor false y que indica que una vez se haya recalculado el número de ceros en la matrizP después de aplicar la función, se debe deshacer los cambios sobre la matrizP y dejarla como estaba inicialmente antes de aplicar dicha función. Esta función es una forma incremental de calcular el fitness y ahorrar tiempo de ejecución.

En la línea 19 se pregunta si el nuevoFitness es menor que el fitness de best. Si es menor, en la línea 20 se crea nuevamente una nueva copia de la armonía denominada best. Luego en las líneas 21 y 23, a la matriz dentro del CA de best se le cambia el renglonOriginal por el renglonOptimizado. Posteriormente en la línea 24 al fitness de best se le asigna el valor de la variable nuevoFitness.

En la línea 26 a la variable min se le asigna el valor que retorna la función minimo (que se explica en una sección posterior y se referencia en la Tabla 29), la cual retorna un número que indica la probabilidad de aceptación que tendría una armonía con un nuevo fitness no tan bueno en relación con el fitness de la armonía original.

En la línea 28 se establece el umbral de aceptación para la armonía con el nuevo fitness, basado en el cumplimiento de cualquiera de las dos siguientes condiciones. La primera condición evalúa si un valor aleatorio entre 0 y 1 es menor a la variable min. Con esta pregunta lo que se busca es aceptar una armonía con un fitness mayor (malo) respecto al fitness de la armonía original. En las primeras iteraciones del algoritmo el valor de min es alto, lo que permite que se acepten armonías con fitness malos. Sin embargo, con el trascurso del tiempo, min tiende a cero dejando así de aceptar armonías con dichas características. La segunda condición pregunta si el nuevoFitness es menor al fitness de la armonía original. Si se cumple cualquiera de las dos anteriores condiciones se procede a actualizar la matriz del CA en la armonía original, con el renglón optimizado obtenido con la aplicación de una de las dos funciones de optimización local. De la línea 28 a la línea 30, se reemplaza el renglón de la matriz del CA en la armonía original, con el renglón optimizado.

En la línea 31 se actualiza el fitness de la armonía original con el valor que retorna la función quitarPonerRenglon descrita en la Tabla 28. El último de los parámetros que recibe la función, se pasa una variable booleana en true, que indica que una vez se haya recalculado el número de ceros en la matrizP, después de hacer el cambio de renglón, los cambios realizados sobre la matrizP deben persistir. Finalmente, en la línea 34 se realiza una copia de best y se asigna a la variable armonía y en la línea 35 se recalcula el fitness de la armonía actualizando internamente su matrizP. Esta variable armonía es la que retorna la función.

Función quitarPonerRenglon

A continuación, en la Tabla 28 se presenta la función quitarPonerRenglon. La función retorna un número entero que hace referencia al fitness o número de ceros registrados en la matrizP en P, después de quitar el renglonOriginal (fila o renglón escogido aleatoriamente de la matriz del CA) y poner el renglonOptimizado (renglón o fila optimizada con alguna de las dos funciones de optimización local).

```
    entero función quitarPonerRenglon(Vector<entero> renglonOriginal,
                                      Vector<entero> renglonOptimizado,
                                      P p, Vector<entero>V,
                                      entero t, booleano modificarP)
1       Vector <entero> logSumados [MaxJ]
2       Vector <entero> logRestados [MaxJ]
3       for i←0 to MaxJ - 1 do
4           j ← DivisionSintetica (renglonOriginal, p.J[i], V, t)
5           decrementarCelda(p, i, j)
6           logRestados[i] ← j
7       end_for
8       for i←0 to MaxJ - 1 do
9           j ← DivisionSintetica(renglonOptimizado, p.J[i], V, t)
10          incrementarCelda(p, i, j)
11          logSumados[i] ← j
12      end_for
13      ceros ← contarCerosEnP()
14      if modificarP = false then
15          for i←0 to MaxJ - 1 do
16              decrementarCelda(p, i, logSumados[i])
17          end_for
18          for i←0 to MaxJ - 1 do
19              incrementarCelda(p, i, logRestados[i])
20          end_for
21      end_if
```

```
22      retornar ceros
    fin_función
```

Tabla 28 Función quitarPonerRenglon

En la línea 1 se crea a logSumados, un vector de enteros de tamaño MaxJ el cual almacena en cada posición, el número de columna en la matrizP donde se realizará un incremento. En la línea 2 se crea a logRestados, un vector de enteros de tamaño MaxJ el cual almacena en cada posición, el número de columna en la matrizP donde se realizará un decremento.

De la línea 3 a la línea 7 se decrementa en la matrizP una ocurrencia de una combinación de alfabetos específica, para lo cual se tiene en cuenta la posición i de un elemento de la matriz J y a la posición j de una columna en la matrizP. Particularmente en la línea 4, en la variable j se almacena el valor retornado por la función divisionSintetica descrita anteriormente en la Tabla 25, el cual devuelve la posición de la columna en la matrizP que coincide con la combinación de alfabetos a evaluar. En la línea 5 se decrementa en uno en la matrizP, una ocurrencia en la combinación de alfabetos que se encuentre en las posiciones i y j respectivas. En la línea 6, en el vector logRestados se almacena el valor de j, que corresponde a la posición de la columna en la matrizP donde se hizo un decremento.

De la línea 8 a la línea 12 se incrementa en la matrizP una ocurrencia de una combinación de alfabetos específica, para lo cual se tiene en cuenta la posición i de un elemento de la matriz J y a la posición j de una columna en la matrizP. Particularmente en la línea 9, en la variable j se almacena el valor retornado por la función divisionSintetica, descrita anteriormente en la Tabla 25, la cual devuelve la posición de la columna en la matrizP que coincide con la combinación de alfabetos a evaluar. En la línea 10 se incrementa en uno en la matrizP, una ocurrencia en la combinación de alfabetos que se encuentre en las posiciones i y j respectivas.

En la línea 11, en el vector logSumados se almacena el valor de j, que corresponde a la posición de la columna en la matrizP donde se hizo un incremento. En la línea 13 en la variable entera ceros se guarda el número de ceros que hay registrados en la matrizP después de haber decrementado e incrementado ocurrencias para una combinación del alfabeto específica.

En la línea 14 se pregunta si el valor de la variable booleana modificarP es falsa. Si es así, de las líneas 15 a la 20, se deshacen los cambios sobre la matrizP,

incrementando las ocurrencias que se decrementaron y decrementando las ocurrencias que se incrementaron. Finalmente, en la línea 22 se retorna el valor que obtuvo la variable ceros correspondiente a las t-adas faltantes en el CA después de quitar el renglón original y poner el renglón optimizado.

A continuación, se presenta un ejemplo que retoma el CA y matrizP de la Figura 17, descrita en una sección anterior, para explicar el funcionamiento de la función quitarPonerRenglon.

En la Figura 17(a) se muestra el CA de Configuración N4K3V2^3t2.ca y en la Figura 17(b) se muestra la matriz P asociada a dicho CA.

En la Figura 49 (a) se muestra el RenglonOriginal, que será utilizado para quitar en la matrizP las ocurrencias del alfabeto para las combinaciones de columnas a las que hay lugar en dicho renglón y en la Figura 49(b) se muestra el RenglonOptimizado, que será utilizado para poner en la matrizP las ocurrencias del alfabeto para las combinaciones de columnas a las que hay lugar en dicho renglón.

1	1	0
0	1	0
0	0	0
0	1	1

(a) CA

columna / fila	(00) *0*	(01) *1*	(10) *2*	(11) *3*
(0,1) *0*	1	2	0	1
(0,2) *1*	2	1	1	0
(1,2) *2*	1	0	2	1

(b) MatrizP asociada al CA

Figura 17 CA de configuración N4K3V2^3t2.ca y MatrizP asociada

pos	*0*	*1*	*2*
	0	0	0

(a) RenglonOriginal

pos	*0*	*1*	*2*
	1	0	1

(b) RenglonOptimizado

Figura 49 RenglonOriginal y RenglonOptimizado

La Figura 50 muestra el proceso de decrementar una celda en la matrizP cuando se evalúa en el renglonOriginal las posiciones 0 y 1. Dichas posiciones tienen correspondencia con el vector [0,1] que se almacena en la posición 0 de la matriz J. Cada fila de la matriz J tiene correspondencia con una fila en la matrizP.

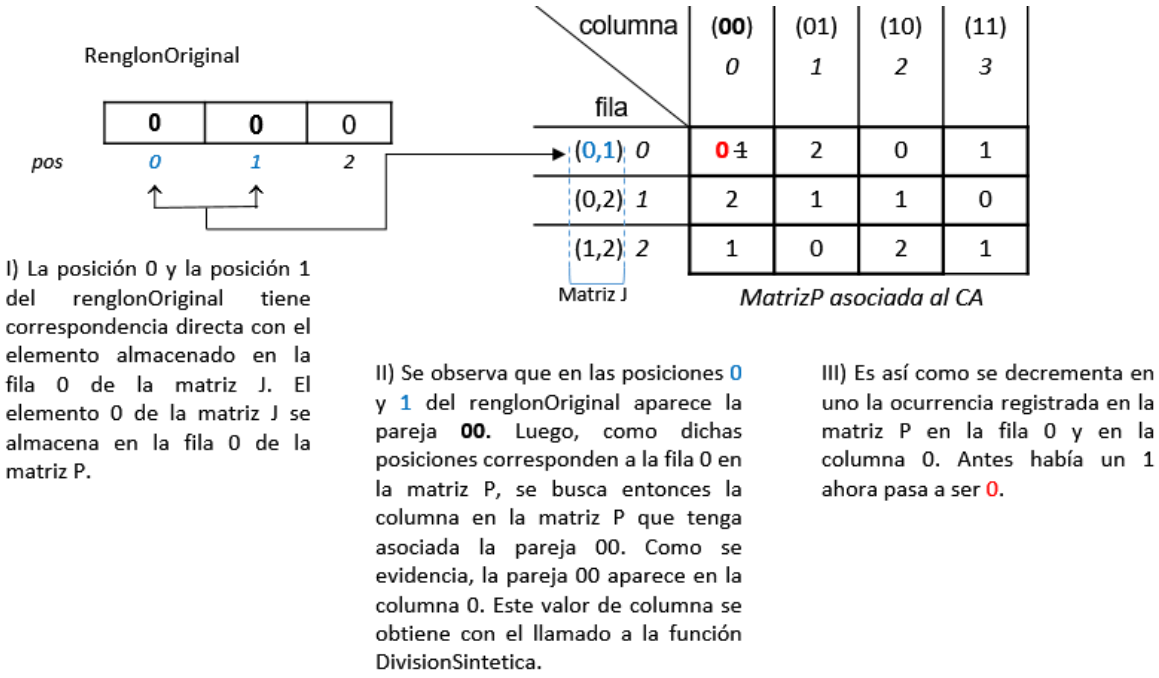
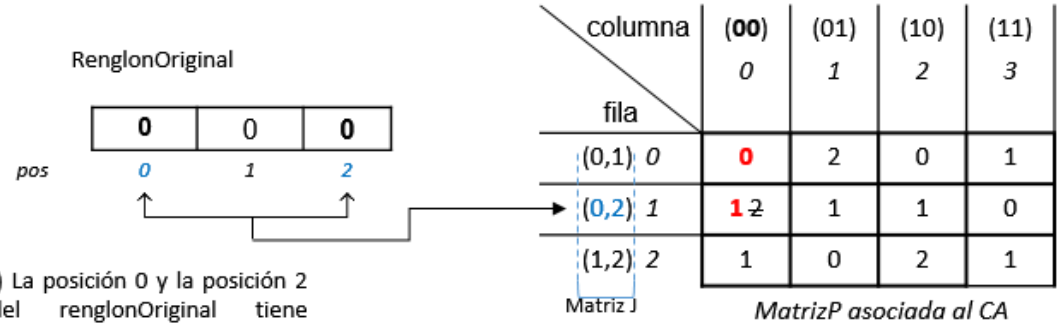


Figura 50 Decrementar celda en matrizP para posición 0 y 1 en RenglonOriginal

La Figura 51 muestra el proceso de decrementar una celda en P cuando se evalúa en el renglonOriginal las posiciones 0 y 2. Dichas posiciones corresponden con la fila 1 de la matriz J. La fila 1 de la matriz J es la fila 1 de la matrizP.

La Figura 52 muestra el proceso de decrementar una celda en la matrizP cuando se evalúa en el renglonOriginal las posiciones 1 y 2. Dichas posiciones corresponden con la fila 2 de la matriz J. La fila 2 de la matriz J es la fila 2 de la matrizP.

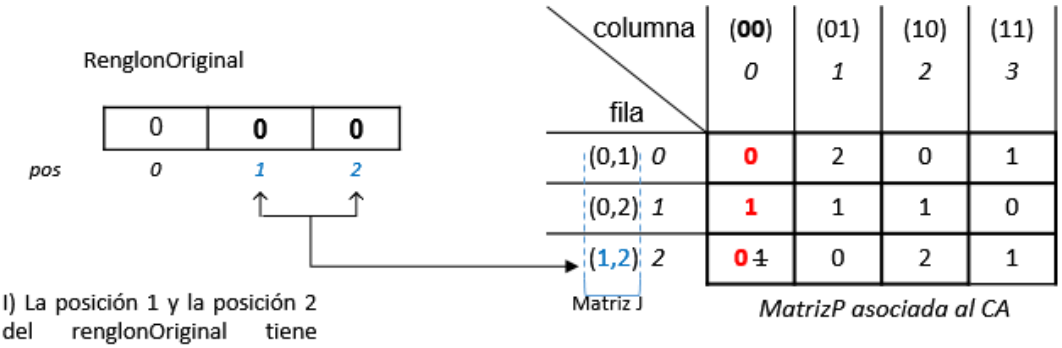


I) La posición 0 y la posición 2 del renglonOriginal tiene correspondencia directa con el elemento almacenado en la fila 1 de la matriz J. El elemento 1 de la matriz J se almacena en la fila 1 de la matriz P.

II) Se observa que en las posiciones 0 y 2 del renglonOriginal aparece la pareja **00.** Luego, como dichas posiciones corresponden a la fila 1 en la matriz P, se busca entonces la columna en la matriz P que tenga asociada la pareja 00. Como se evidencia, la pareja 00 aparece en la columna 0. Este valor de columna se obtiene con el llamado a la función DivisionSintetica.

III) Es así como se decrementa en uno la ocurrencia registrada en la matriz P en la fila 1 y en la columna 0. Antes había un 2 ahora pasa a ser 1.

Figura 51 Decrementar celda en matrizP para posición 0 y 2 en RenglonOriginal



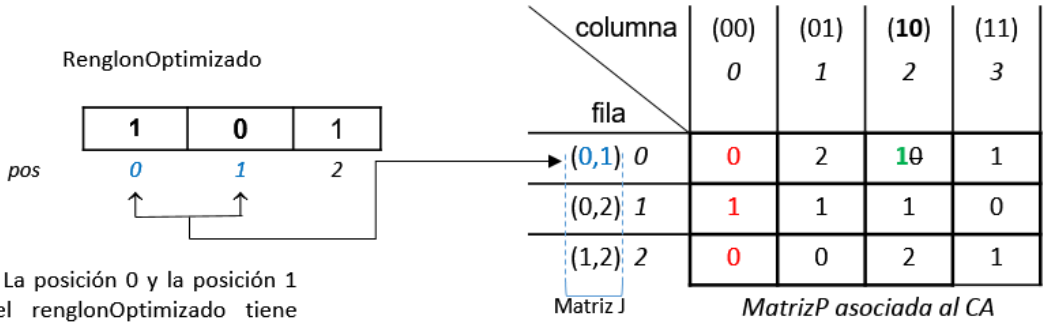
I) La posición 1 y la posición 2 del renglonOriginal tiene correspondencia directa con el elemento almacenado en la fila 2 de la matriz J. El elemento 2 de la matriz J se almacena en la fila 2 de la matriz P.

II) Se observa que en las posiciones 1 y 2 del renglonOriginal aparece la pareja **00.** Luego, como dichas posiciones corresponden a la fila 2 en la matriz P, se busca entonces la columna en la matriz P que tenga asociada la pareja 00. Como se evidencia, la pareja 00 aparece en la columna 0. Este valor de columna se obtiene con el llamado a la función DivisionSintetica.

III) Es así como se decrementa en uno la ocurrencia registrada en la matriz P en la fila 2 y en la columna 0. Antes había un 1 ahora pasa a ser 0.

Figura 52 Decrementar celda en matrizP para posición 1 y 2 en RenglonOriginal

Una vez realizados los decrementos en las celdas en la matrizP como resultado de quitar el renglonOriginal, se procede a realizar los incrementos en las celdas de la matrizP cuando se adicione el renglonOptimizado. La Figura 53 muestra el proceso de incrementar una celda en la matrizP cuando se evalúa en el renglonOptimizado las posiciones 0 y 1. Luego, en la Figura 54 y la Figura 55 se muestran respectivamente el proceso de incrementar una celda en la matrizP cuando se evalúa en el renglonOptimizado las posiciones 0 y 2 y luego las posiciones 1 y 2.

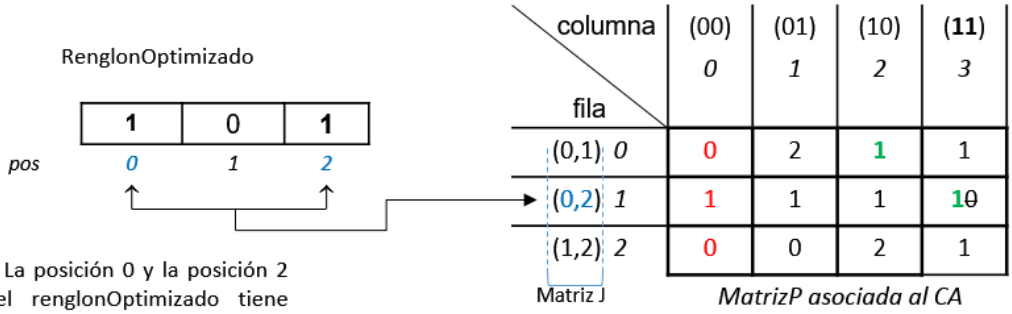


I) La posición 0 y la posición 1 del renglonOptimizado tiene correspondencia directa con el elemento almacenado en la fila 0 de la matriz J. El elemento 0 de la matriz J se almacena en la fila 0 de la matrizP.

II) Se observa que en las posiciones 0 y 1 del renglonOptimizado aparece la pareja **10.** Luego, como dichas posiciones corresponden a la fila 0 en la matrizP, se busca entonces la columna en la matrizP que tenga asociada la pareja 10. Como se evidencia, la pareja 10 aparece en la columna 2. Este valor de columna se obtiene con el llamado a la función divisionSintetica.

III) Es así como se incrementa en uno la ocurrencia registrada en la matrizP en la fila 0 y en la columna 2. Antes había un 0 ahora pasa a ser 1.

Figura 53 Incrementar celda en matrizP para posición 0 y 1 en RenglonOptimizado

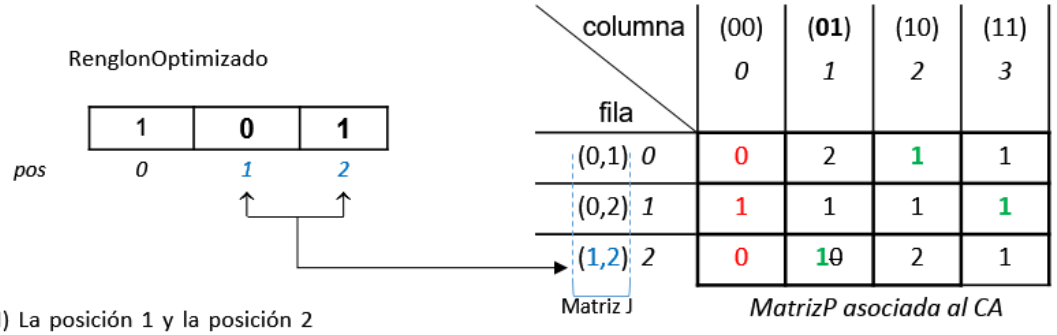


I) La posición 0 y la posición 2 del renglonOptimizado tiene correspondencia directa con el elemento almacenado en la fila 1 de la matriz J. El elemento 1 de la matriz J se almacena en la fila 1 de la matrizP.

II) Se observa que en las posiciones 0 y 2 del renglonOptimizado aparece la pareja **11.** Luego, como dichas posiciones corresponden a la fila 1 en la matrizP, se busca entonces la columna en la matrizP que tenga asociada la pareja 11. Como se evidencia, la pareja 11 aparece en la columna 3. Este valor de columna se obtiene con el llamado a la función divisionSintetica.

III) Es así como se incrementa en uno la ocurrencia registrada en la matrizP en la fila 1 y en la columna 3. Antes había un 0 ahora pasa a ser **1**.

Figura 54 Incrementar celda en matrizP para posición 0 y 2 en RenglonOptimizado



I) La posición 1 y la posición 2 del renglonOptimizado tiene correspondencia directa con el elemento almacenado en la fila 2 de la matriz J. El elemento 2 de la matriz J se almacena en la fila 1 de la matrizP.

II) Se observa que en las posiciones 1 y 2 del renglonOptimizado aparece la pareja **01.** Luego, como dichas posiciones corresponden a la fila 2 en la matrizP, se busca entonces la columna en la matrizP que tenga asociada la pareja 01. Como se evidencia, la pareja 01 aparece en la columna 1. Este valor de columna se obtiene con el llamado a la función divisionSintetica.

III) Es así como se incrementa en uno la ocurrencia registrada en la matrizP en la fila 2 y en la columna 1. Antes había un 0 ahora pasa a ser **1**.

Figura 55 Incrementar celda en matrizP para posición 1 y 2 en RenglonOptimizado

Finalmente, en la Figura 56 se observa la matrizP resultante después de quitar el renglón original [0, 0, 0] y poner el renglón optimizado [0, 1, 0]. La matrizP quedó con 2 ceros, este es el valor que retorna la función quitarPonerRenglon.

columna / fila	(00) *0*	(01) *1*	(10) *2*	(11) *3*
(0,1) *0*	**0**	2	1	1
(0,2) *1*	1	1	1	1
(1,2) *2*	**0**	1	2	1

Matriz J

MatrizP asociada al CA

Figura 56 matrizP resultante después de aplicar la función quitarPonerRenglon

Retomando el algoritmo optimizarCAconSA que se muestra en la Tabla 27, en la línea 26 se llama a la función minimo. Esta función tiene como objetivo encontrar la probabilidad de pasar de un estado a otro con base en una temperatura específica. Cabe recalcar que a temperaturas iniciales es posible que se acepten soluciones no muy buenas esto con el fin de evitar caer en óptimos locales. La función minimo se muestra en la Tabla 29.

```
    real función minimo(entero nuevoFitness, entero viejoFitness, real
                        temperatura)
1       exponente ← -1.0 * (nuevoFitness- viejoFitness)/ temperatura
2       r←exp(exponente)
3       if r < 1.0 then
4           return r
5       end_if
6       return 1.0
    fin_función
```

Tabla 29 Función minimo

Función de optimización local PMisLocal1RepetidoEnP

La Tabla 30 muestra la función de optimización local PMisLocal1RepetidoEnP. Esta función toma cada fila o renglón de la matriz del CA, le hace un cambio con base en unas posiciones y valores específicos. Luego, calcula el fitness para el renglón modificado, lo compara con el fitness original y finalmente retorna el renglón que permitió obtener el mejor fitness.

```
    Vector<entero> función PMisLocal1RepetidoEnP(Armonia armonia, entero fitness,
                                                 out entero renglonElegido)
1       renglonElegido←Random(armonia.CA.N)
2       totalCeros←armonia.P.contarCerosEnP()
3       Matriz<entero> lista[totalCeros][ ]
4       lista ←armonia.P.FaltanEnP()
5       posCero←Random(totalCeros)
6       Vector<entero> mejorRenglon[armonia.CA.K]
7       booleano cambio←false
8       Vector<entero> posiciones[armonia.CA.N]
9       for i←0 to armonia.CA.N-1 do
10          posiciones.add(i)
11      end_for
12      while posiciones.size() > 0 do
13          seleccionado← Random(posiciones.size())
14          pos←posiciones[seleccionado]
15          posiciones.Remove(seleccionado)
16          Vector<entero> renglonOriginal← armonia.CA.Matriz[pos]
17          Vector<entero> nuevoRenglon←armonia.CA.Matriz[pos]
18          filaP←lista[posCero][0]
19          columnaP← lista[posCero][1]
20          Vector<entero> valoresP
21          valoresP←multiplicacionSintetica(columnaP, armonia.CA.V, P.J[filaP],
                        armonia.CA.t)
22          for columna←0 to armonia.CA.t-1 do
23              nuevoRenglon[armonia.P.J[filaP][columna]]←valoresP[columna]
24          end_for
25          faltan←quitarPonerRenglon(renglonOriginal, nuevoRenglon, armonia.P,
                    armonia.CA.V, armonia.CA.t, false)
26          if faltan < fitness then
27              renglonElegido ←pos
28              fitness ← faltan
29              mejorRenglon ← nuevoRenglon
30              cambio ←true
31          end_if
32      end_while
33      if cambio = false then
34          mejorRenglon←armonia.CA.Matriz[renglonElegido]
35          shake(armonia, mejorRenglon)
36      end_if
37      return mejorRenglon
    fin_función
```

Tabla 30 Función de Optimización Local PMisLocal1RepetidoEnP

En la línea 1, a renglonElegido que es la variable que saldrá como parámetro de la función, se le asigna un valor aleatorio comprendido entre cero y el número de filas N del CA. En la línea 2, a totalCeros se le asigna el valor que retorna la función

contarCerosEnP, la cual devuelve el número de ceros que hay registrados en la matriz de P dentro de la armonía. En la línea 3, se crea a la matriz de enteros denominada lista. El número de filas de la matriz lista está definido por el valor de la variable totalCeros. En la línea 4, la matriz lista almacena una matriz que devuelve la función FaltanEnP. En dicha función, se crea una matriz donde cada fila almacena un vector de enteros de dos posiciones que referencian la fila y columna donde hay un cero en la matriz de P.

En la línea 5 la variable posCero toma un valor aleatorio entre cero y la variable totalCeros. En la línea 6 se crea a mejorRenglon, un vector de enteros de tamaño K. Este será el vector que retornará la función una vez haya sido optimizado. En la línea 7 se inicializa una variable booleana con el valor de falso y en la línea 8 se crea al vector posiciones, cuyo tamaño está dado por el número de filas N del CA.

De la línea 9 a la 11, el vector posiciones se llena con los números comprendidos entre 0 y N-1. En la línea 12, mientras el tamaño del vector posiciones sea mayor que cero se realizarán un conjunto de acciones que se realizarán repetidamente hasta la línea 32 y que se describen a continuación. En la línea 13, la variable seleccionado toma un valor entero aleatorio entre cero y el tamaño del vector posiciones. En la línea 14, la variable pos guarda el número que se encuentra almacenado dentro del vector posiciones en la posición definida por la variable denominada seleccionado. En la línea 15 se remueve del vector posiciones el elemento que se encuentra ubicado en la posición definida por la variable denominada seleccionado. En la línea 16, se crea el vector de enteros renglonOriginal y al cual se le asigna la fila de la matriz del CA en la armonía que se encuentra en la posición pos. De igual manera en la línea 17, se crea el vector de enteros nuevoRenglon y al cual se le asigna la fila de la matriz del CA en la armonía que se encuentra también en la posición pos.

En la línea 18 en la variable filaP se almacena el elemento que se encuentra en la posición cero del vector de enteros almacenado en la lista en la fila posCero. En la línea 19 en la variable columnaP se almacena el elemento que se encuentra en la posición uno del vector de enteros almacenado en la lista en la fila posCero. En la línea 20 se crea el vector de enteros valoresP y en la línea 21, valoresP almacena el vector que devuelve la función multiplicacionSintetica. La variable filaP y columnaP que hacen parte de los parámetros que recibe la función, representan respectivamente la posición de una fila y una columna en la matrizP. En dicha coordenada hay registrado un cero en la matrizP. La función multiplicacionSintetica

que se muestra en la Tabla 31 devuelve un vector de números enteros que representa la combinación del alfabeto faltante en la matriz de CA en la combinación de parámetros descritos en la matriz J en su posición filaP.

De la línea 22 a la 24 en el nuevoRenglon se cambian cada uno de los elementos que se encuentran en las posiciones descritas en la matriz J en su posición filaP, por los elementos que se encuentran en el vector valoresP. En la línea 25 en la variable entera faltan se almacena el valor que devuelve la función quitarPonerRenglon descrita en la Tabla 28, la cual retorna el fitness o número de ceros registrados en matrizP en P una vez se quite el renglonOriginal y se ponga el nuevoRenglon.

En la línea 26 se pregunta si el valor de la variable faltan (fitness calculado anteriormente con la función quitarPonerRenglon) es menor que el valor de la variable fitness que llega como parámetro a la función de optimización local y que representa el fitness de la armonía original. Solo si es menor, en la línea 27 la variable renglonElegido se actualiza con el valor de la variable pos. En la línea 28 el valor de la variable fitness se actualiza con el valor de la variable faltan. En la línea 29 en el vector mejorRenglon se copia el vector nuevoRenglon y en la línea 30 la variable booleana cambio toma el valor de verdadero.

Una vez termine el ciclo mientras, en la línea 33 se pregunta si la variable booleana denominada cambio quedó con el valor de falso. Si esto ocurrió, es porque en las repetidas ocasiones en las que se intentó actualizar el renglonOriginal con el nuevoRenglon, el fitness del nuevoRenglon nunca fue mejor que el fitness original. Por consiguiente, en la línea 34, en el vector mejorRenglon se almacena un renglón escogido aleatoriamente de las N filas de la matriz en el CA.

En la línea 35 se le aplica al vector mejorRenglon el método shake descrito en la Tabla 32, que cambia al azar el valor de una columna de dicho vector, teniendo en cuenta el alfabeto que toma cada columna, el cual está definido en el vector V del CA. Finalmente, en la línea 37 se retorna el vector mejorRenglon.

A continuación, en la Figura 57 se presenta un ejemplo gráfico que permite mostrar la creación e inicialización de varios objetos dentro de la función de optimización local PMisLocal1RepetidoEnP.

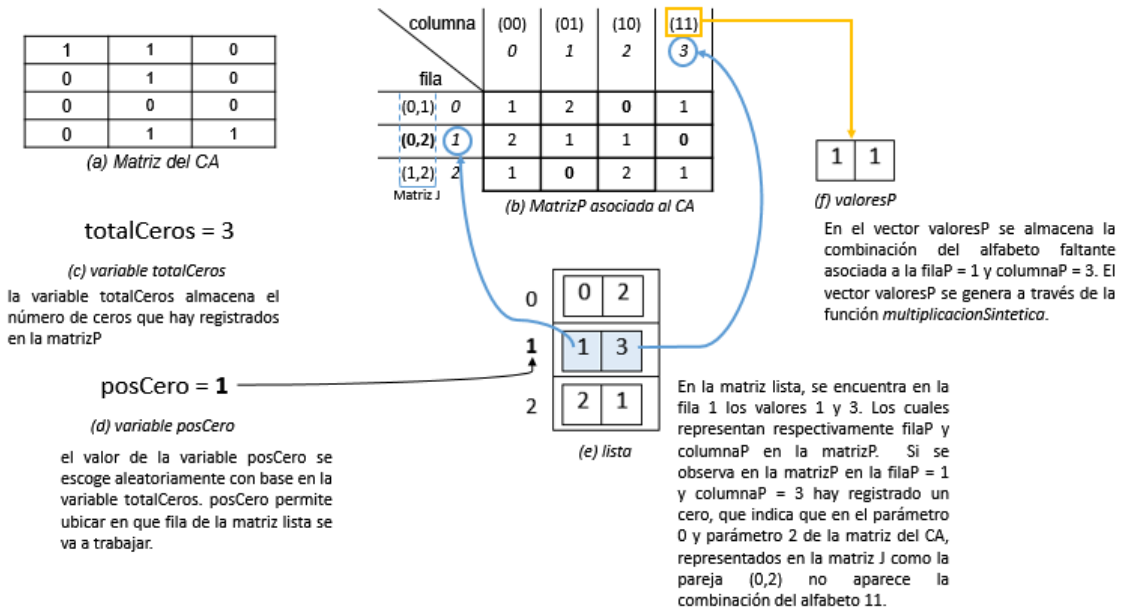


Figura 57 Inicialización vectores lista y valoresP y variables totalCeros y posCero

Una vez se hayan creado e inicializado los vectores lista y valoresP y las variables totalCeros y posCero, se procede a realizar un número específico de iteraciones (definido por el número de filas de la matriz del CA) para encontrar el mejor renglón. Si se observa la Figura 57(a) la matriz del CA tiene 4 filas, así que se realizarán 4 ciclos o iteraciones.

En el ciclo 1, se escoge una fila aleatoria de la matriz del CA, en este caso se escogió el renglón que se muestra en la Figura 58(a) y que hace referencia al renglonOriginal. La Figura 58(b) muestra el vector nuevoRenglon, que inicialmente es igual al renglonOriginal. El nuevoRenglon actualizará sus valores en las posiciones indicadas en la Figura 58(c) con los valores indicados en la Figura 58(d). La Figura 58(e) muestra el nuevoRenglon con sus valores actualizados.

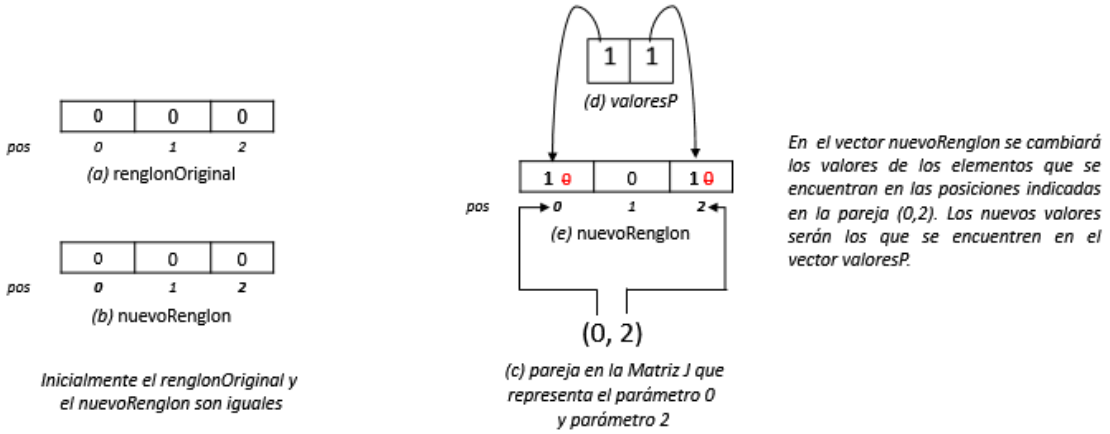


Figura 58 Creación del vector nuevoRenglon para el ciclo 1

En la Figura 59(a) y en la Figura 59(b) se muestra respectivamente los vectores renglonOriginal y nuevoRenglon. La Figura 59(c) indica el valor de la variable fitness. Esta variable llega como parámetro a la función de optimización local.

Los vectores renglonOriginal y nuevoRenglon son enviados a la función quitarPonerRenglon indicada en la Figura 59(d), la cual retorna un valor que es almacenado en la variable faltan que se muestra en la Figura 59(e) y que representa el número de ceros registrados en la matrizP al quitar el renglonOriginal y al poner el nuevoRenglon. Como el valor de la variable faltan que es igual a 2, es menor que el valor de la variable fitness que es igual a 3, el valor de la variable fitness será actualizado con el valor de la variable faltan como se muestra en la Figura 59(f). En la Figura 59(g), se crea un vector denominado mejorRenglon que será igual al vector nuevoRenglon.

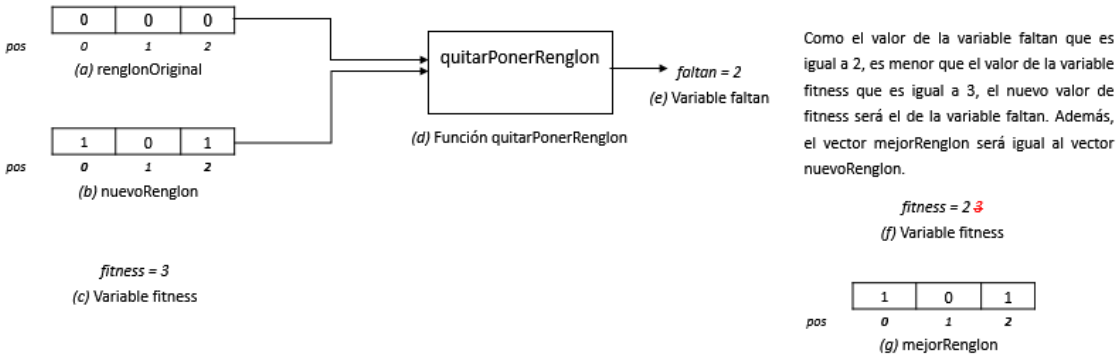


Figura 59 Aplicación de la función quitarPonerRenglon en el ciclo 1

En el ciclo 2, se escoge una fila aleatoria de la matriz del CA, en este caso se escogió el renglón que se muestra en la Figura 60(a) y que hace referencia al renglonOriginal. La Figura 60(b) muestra el vector nuevoRenglon, que inicialmente es igual al renglonOriginal. El nuevoRenglon actualizará sus valores en las posiciones indicadas en la Figura 60(c) con los valores indicados en la Figura 60(d). La Figura 60(e) muestra el nuevoRenglon con sus valores actualizados.

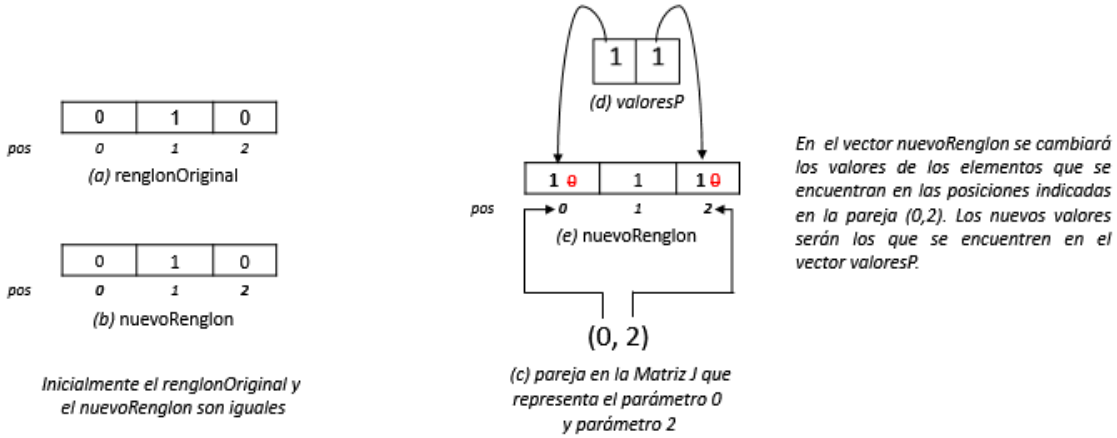


Figura 60 Creación del vector nuevoRenglon para el ciclo 2

En la Figura 61(a) y en la Figura 61(b) se muestra respectivamente los vectores renglonOriginal y nuevoRenglon. La Figura 61(c) indica el valor de la variable fitness. Los vectores renglonOriginal y nuevoRenglon son enviados a la función quitarPonerRenglon indicada en la Figura 61(d), la cual retorna un valor que es almacenado en la variable faltan que se muestra en la Figura 61(e) y que representa el número de ceros registrados en la matrizP al quitar el renglonOriginal y al poner el nuevoRenglon. Como el valor de la variable faltan es igual al valor de la variable fitness, no se realiza actualización alguna sobre la variable fitness ni sobre el vector mejorRenglon.

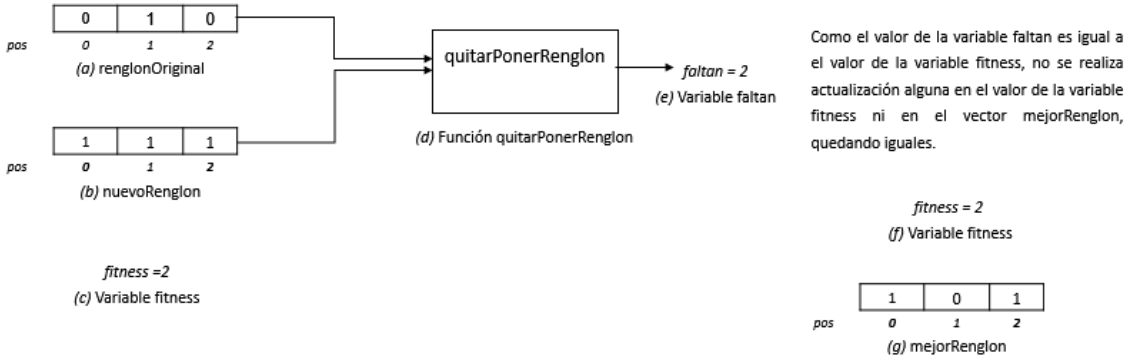


Figura 61 Aplicación de la función quitarPonerRenglon en el ciclo 2

En el ciclo 3, se escoge una fila aleatoria de la matriz del CA, en este caso se escogió el renglón que se muestra en la Figura 63(a) y que hace referencia al renglonOriginal. La Figura 63(b) muestra el vector nuevoRenglon, que inicialmente es igual al renglonOriginal. El nuevoRenglon actualizará sus valores en las posiciones indicadas en la Figura 63(c) con los valores indicados en la Figura 63(d). La Figura 63(e) muestra el nuevoRenglon con sus valores actualizados.

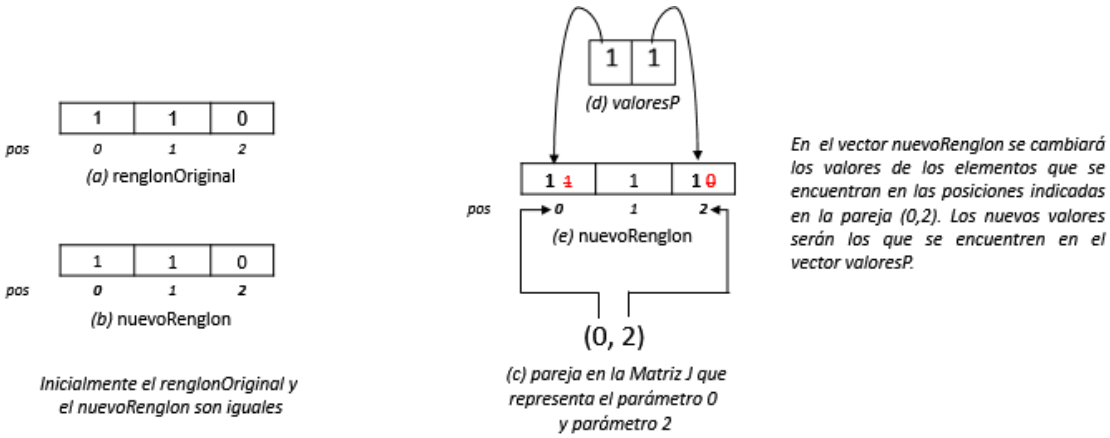


Figura 62 Creación del vector nuevoRenglon para el ciclo 3

En la Figura 63(a) y en la Figura 63(b) se muestra respectivamente los vectores renglonOriginal y nuevoRenglon. La Figura 63 (c) indica el valor de la variable fitness. Los vectores renglonOriginal y nuevoRenglon son enviados a la función quitarPonerRenglon indicada en la Figura 63(d), la cual retorna un valor que es almacenado en la variable faltan que se muestra en la Figura 63(e) y que representa el número de ceros registrados en la matrizP al quitar el renglonOriginal y al poner el nuevoRenglon. Como el valor de la variable faltan es mayor al valor de la variable fitness, no se realiza actualización alguna sobre la variable fitness ni sobre el vector mejorRenglon.

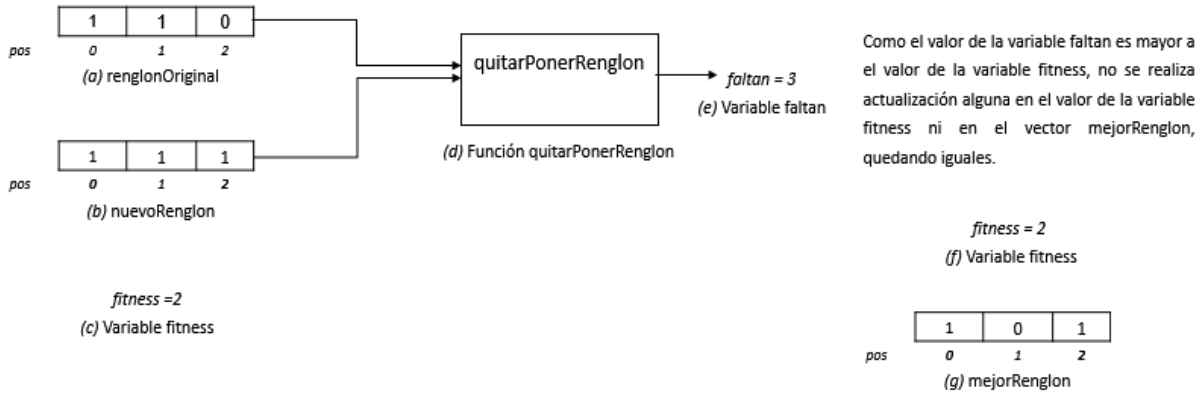


Figura 63 Aplicación de la función quitarPonerRenglon en el ciclo 3

En el ciclo 4, se escoge una fila aleatoria de la matriz del CA, en este caso se escogió el renglón que se muestra en la Figura 64(a) y que hace referencia al renglonOriginal. La Figura 64(b) muestra el vector nuevoRenglon, que inicialmente es igual al renglonOriginal. El nuevoRenglon actualizará sus valores en las posiciones indicadas en la Figura 64(c) con los valores indicados en la Figura 64(d). La Figura 64(e) muestra el nuevoRenglon con sus valores actualizados.

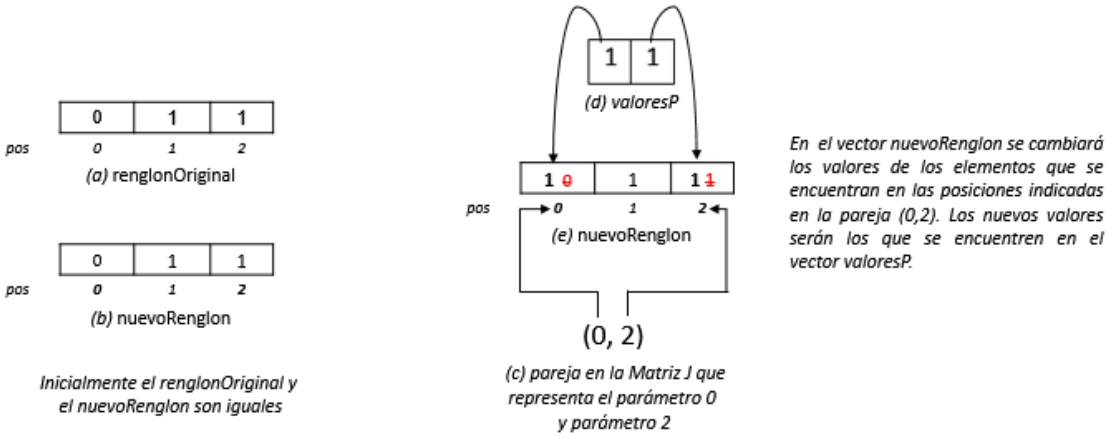


Figura 64 Creación del vector nuevoRenglon para el ciclo 4

En la Figura 65(a) y en la Figura 65 (b) se muestra respectivamente los vectores renglonOriginal y nuevoRenglon. La Figura 65(c) indica el valor de la variable fitness. Los vectores renglonOriginal y nuevoRenglon son enviados a la función quitarPonerRenglon indicada en la Figura 65(d), la cual retorna un valor que es almacenado en la variable faltan que se muestra en la Figura 65(e) y que representa el número de ceros registrados en la matrizP al quitar el renglonOriginal y al poner el nuevoRenglon. Como el valor de la variable faltan es mayor al valor de la variable fitness, no se realiza actualización alguna sobre la variable fitness ni sobre el vector mejorRenglon.

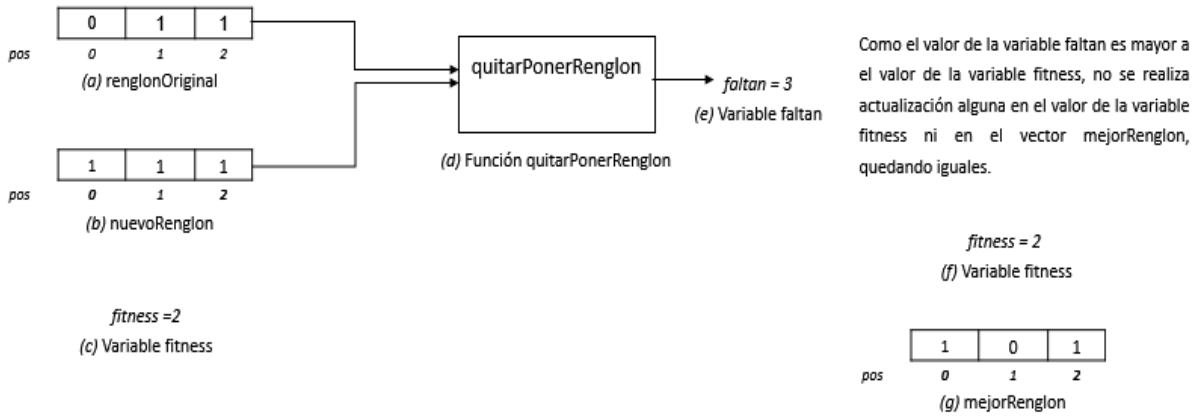


Figura 65 Aplicación de la función quitarPonerRenglon en el ciclo 4

Al finalizar las 4 iteraciones, la función de optimización local PMisLocal1RepetidoEnP retorna el vector mejorRenglon que se muestra en la Figura 65(g).

Función multiplicacionSintetica

La Tabla 31 muestra la función multiplicacionSintetica. La función multiplicacionSintetica devuelve un vector de números enteros que representa la combinación del alfabeto faltante en la matriz del CA, en la combinación de parámetros descritos en la matriz J en su posición filaP.

```
   Vector<entero> función multiplicacionSintetica(entero columnaP, Vector<entero>V,
                                                  Vector<entero> lineaDeJ, entero t)
1      Vector<entero> valores [t]
2      for i←t-1 to i>0 decr 1 do
3          valores[i]← columnaP % V[lineaDeJ[i]]
4          columnaP← columnaP / v[lineaDeJ[i]]
5      end_for
6      valores[0] ← columnaP
7      return valores
   fin_función
```

Tabla 31 Función multiplicacionSintetica

A continuación, se presenta un ejemplo donde se explica la funcionalidad de la función multiplicacionSintetica. Para ello, se retoma el CA de configuración N12K4V4^1-3^1-2^2t2.ca que se muestra en la Figura 29 y la matrizP llena, asociada al CA que se muestra en la Figura 48.

0	0	0	0	1
1	1	1	1	1
2	3	0	1	0
3	1	2	0	0
4	1	2	1	0
5	3	1	0	0
6	2	0	0	0
7	3	2	1	0
8	2	1	0	1
9	1	2	0	1
10	1	1	1	0
11	3	0	0	0

Figura 29 CA de configuración N12K4V4^1-3^1-2^2t2

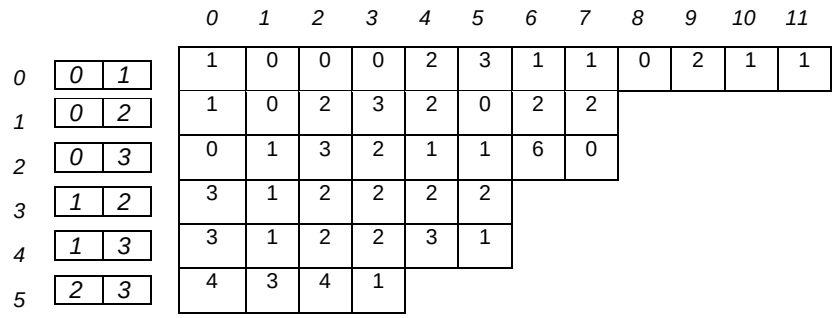


Figura 48 MatrizP llena

La función multiplicacionSintetica recibe entre sus parámetros a columnaP y una fila de la matriz J en la matrizP denominada filaP. Se asume para el ejemplo que filaP es igual a 2 y columnaP es igual a 7. El objetivo de la función es devolver en forma de vector la combinación de parámetros faltantes en el CA de acuerdo al valor de filaP y columnaP.

Para el ejemplo, en la Figura 66 se observa que, en el CA, en la combinación de parámetros 0 y parámetro 3 no aparece la pareja 00 ni la pareja 31. Es por eso que en la Figura 67 en la matrizP aparecen 2 ceros registrados en filaP igual a 2. Dado a que columnaP es igual a 7, la función multiplicacionSintetica retornará para este caso el vector con los elementos 3 y 1.

	p0	*p1*	*p2*	***p3***
0	0	0	0	1
1	1	1	1	1
2	3	0	1	0
3	1	2	0	0
4	1	2	1	0
5	3	1	0	0
6	2	0	0	0
7	3	2	1	0
8	2	1	0	1
9	1	2	0	1
10	1	1	1	0
11	3	0	0	0

Figura 66 Columnas p0 y p3 del CA de configuración N12K4V4^1-3^1-2^2t2

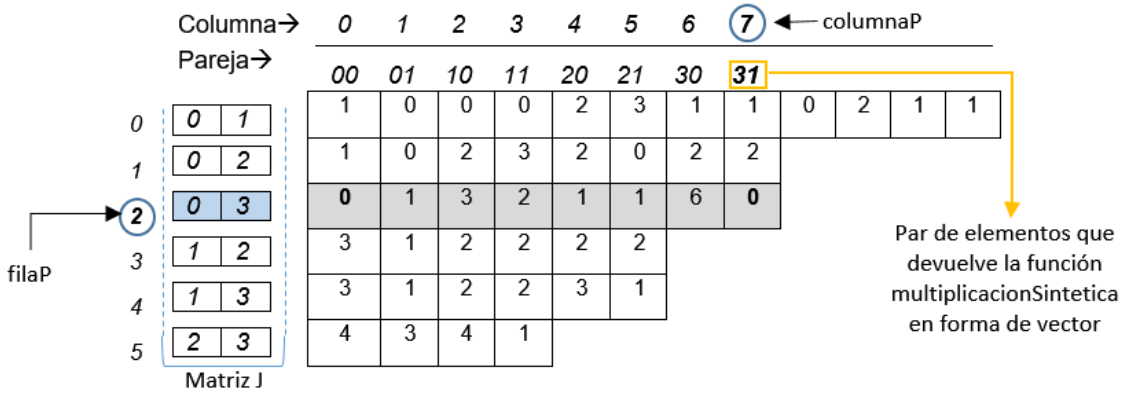


Figura 67 Funcionamiento de la función multiplicacionSintetica

Internamente la función multiplicacionSintetica lo que hace es dividir columnaP entre el número de valores del alfabeto registrados en el vector V para cada combinación de parámetros representados en filaP.

Si se observa en la Figura 68, el parámetro p0 ubicado en la posición 0 de filaP, tiene 4 valores como alfabeto y que corresponde con el número almacenado en la posición 0 del vector V. El parámetro p3 ubicado en la posición 1 de filaP, tiene 2 valores como alfabeto y que corresponde con el número almacenado en la posición 3 del vector V.

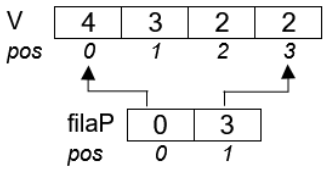


Figura 68 correspondencia filaP y vector V

En la Figura 69 se muestra el funcionamiento interno de la función multiplicacionSintetica. Se procede a dividir columnaP que es igual a 7 entre el número almacenado en la posición 3 del vector V, en este caso el número 2. El valor del residuo que es igual a 1 se almacena en la posición 1 del vector valores (creado dentro de la función multiplicacionSintetica), que corresponde a la posición t-1, siendo t = 2. Luego, el resultado de la división entre 7 y 2 que es igual a 3, se divide entre el valor almacenado en la posición 0 del vector V, en este caso un 4. El valor del residuo que es igual a 3 se almacena en la posición 0 del vector valores ya que

se decrementa en uno dicha posición. Finalmente, el vector valores es retornado por la función.

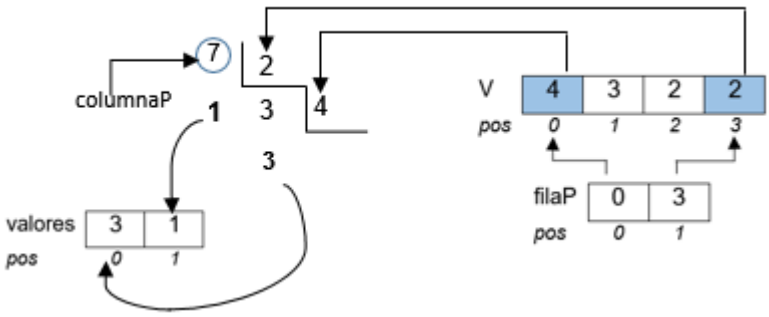


Figura 69 funcionamiento interno multiplicacionSintetica

Procedimiento shake

El procedimiento shake se muestra en la Tabla 32 el cual cambia el valor de un elemento que se encuentra en una posición escogida aleatoriamente en el vector denominado renglon. Dado a que dicho vector se pasa como parámetro por referencia, su modificación dentro del método shake se evidencia dentro de la función de optimización donde se hizo el llamado.

```
    procedimiento shake(Armonia armonia, Vector<entero> renglon)
1       col ←Random(armonia.CA.K)
2       renglon[col] ← armonia.CA.V[col] - renglon[col] – 1
    fin_procedimiento
```

Tabla 32 Función shake

La función shake recibe como parámetro una fila de la matriz CA que ha sido escogida aleatoriamente. En la línea 1, se amacena en la variable entera col un valor aleatorio escogido entre el elemento K del CA que representa el número de parámetros o columnas definido en la matriz en el CA. En la línea 2, se selecciona el elemento que se encuentra la posición pos del vector renglon y se le resta el valor de 1. Luego, se escoge el valor del elemento que se encuentra en el vector V en la posición pos y se le resta la operación realizada anteriormente. El resultado es asignado al vector renglon en la columna indicada.

Función de optimización local PMisLocal2RepetidoEnTodasColumnas

La Tabla 33 muestra la función de optimización Local PMisLocal2RepetidoEnTodasColumnas. La función parte de un renglón denominado nuevoRenglon que es copia de un renglón original que llega como parámetro. Para cada posición del nuevoRenglon se calculan el fitness de cada uno de los valores del alfabeto que puede tomar esa posición específica. En un ciclo iterativo, el fitness calculado de cada valor del alfabeto para la posición específica es comparado con el fitness del renglón original. Al final se retorna el renglón que reportó el mejor fitness.

```
    Vector<entero> función PMisLocal2RepetidoEnTodasColumnas(Armonia armonia,
                                        Vector<entero> renglonOriginal, entero fitness)
1       Vector<entero> nuevoRenglon ← renglonOriginal
2       booleano cambio← false
3       for i←0 to armonia.CA.K-1 do
4           entero mejorValor←nuevoRenglon[i]
5           for pos←0 to armonia.CA.V[i]-1 do
6               nuevoRenglon[i] ← pos
7               if renglonOriginal[i] <> nuevoRenglon[i] then
8                   faltan ← Validador.quitarPonerRenglon(renglonOriginal, nuevoRenglon,
                            armonia.P, armonia.CA.V, armonia.CA.t, false)
9                   if faltan < fitness then
10                      mejorValor←pos
11                      fitness← faltan
12                      cambio← true
13                  end_if
14              end_if
15          end_for
16          nuevoRenglon[i] ← mejorValor
17      end_for
18      if cambio = false then
19          shake(armonia, nuevoRenglon)
20      end_if
21      return nuevoRenglon
    fin_función
```

Tabla 33 Función de Optimización Local PMisLocal2RepetidoEnTodasColumnas

En la línea 1 en el vector de enteros nuevoRenglon se hace una copia del vector renglonOriginal que llega como parámetro a la función. En la línea 2 se crea una variable booleana denominada cambio que se inicializa con valor falso y que permite controlar si un renglón es mejor que otro.

La estructura repetitiva externa comprendida entre las líneas 3 y 17, permite ir calculando el fitness para cada valor del alfabeto de un parámetro específico, de los K parámetros del CA. En la línea 4, la variable entera mejorValor almacena el valor actual del elemento que se encuentra en la posición i del nuevoRenglon.

La estructura repetitiva interna, comprendida desde la línea 5 a la línea 15 permite ir calculando el fitness para los diferentes valores del alfabeto para una posición i específica en el nuevoRenglon. Si ese fitness calculado es mejor que el fitness original que llegó como parámetro a la función, se actualiza el valor del fitness original por el calculado. En la línea 6, el valor de la variable pos se asigna al vector nuevoRenglon en la posición i. Luego, en la línea 7 se pregunta si el valor del elemento en el renglonOriginal en la posición i es diferente al valor del elemento en el nuevoRenglon en la misma posición i. Si son diferentes, en la línea 8, la variable faltan almacena el valor que retorna la función quitarPonerRenglon que se muestra en la Tabla 28. El valor retornado hace referencia al fitness o número de ceros registrados en la matrizP en P, después de quitar el renglonOriginal y poner el nuevoRenglon.

En la línea 9 se pregunta si el fitness registrado en la variable faltan es menor al fitness que llega como parámetro a la función de optimización local. Si es así, en la línea 10, la variable mejorValor guarda el valor de la variable pos. En la línea 11, la variable fitness toma el valor de la variable faltan y en la línea 12 la variable booleana cambio toma el valor de verdadero, indicando que el fitness del nuevoRenglon fue mejor que el renglonOriginal.

Una vez se ha calculado para un parámetro específico el fitness de cada uno de sus valores del alfabeto y se haya encontrado el mejor, en la línea 16 se cambia en el nuevoRenglon, el valor del alfabeto que reportó el mejor fitness.

En la línea 18 se pregunta si la variable booleana cambio es falsa. Si es así, indica que ningún fitness calculado, cuando se probó para cada parámetro sus diferentes valores del alfabeto llegó a ser mejor que el fitness del renglonOriginal. Por consiguiente, se procede a llamar al procedimiento shake, descrito en la Tabla 32, que permite cambiar en el nuevoRenglon el valor de un elemento que se encuentra en una posición escogida aleatoriamente. Finalmente, en la línea 21, se retorna el vector nuevoRenglon.

A continuación, en la se presenta un ejemplo gráfico que muestra el funcionamiento de la función de optimización local PMisLocal2RepetidoEnTodasColumnas.

Si se observa la Figura 70(a), la matriz del CA tiene 3 parámetros a saber: parámetro p0, p1 y p2. Por tal razón se realizará un total de 3 ciclos o iteraciones para encontrar el renglón que reporte el mejor fitness.
Previamente, se ha escogido aleatoriamente un renglón de la matriz del CA y éste ha sido enviado como parámetro a la función de optimización local PMisLocal2RepetidoEnTodasColumnas. Este vector es el que se muestra en la Figura 70(b). La Figura 70(c) muestra el vector nuevoRenglon, que inicialmente es igual al renglonOriginal. La Figura 70(d) muestra el vector V el cual indica en cada uno de sus elementos, el número de valores del alfabeto que toma cada uno de los parámetros de la matriz del CA. La Figura 70(e) describe más detalladamente los valores que toma cada parámetro.

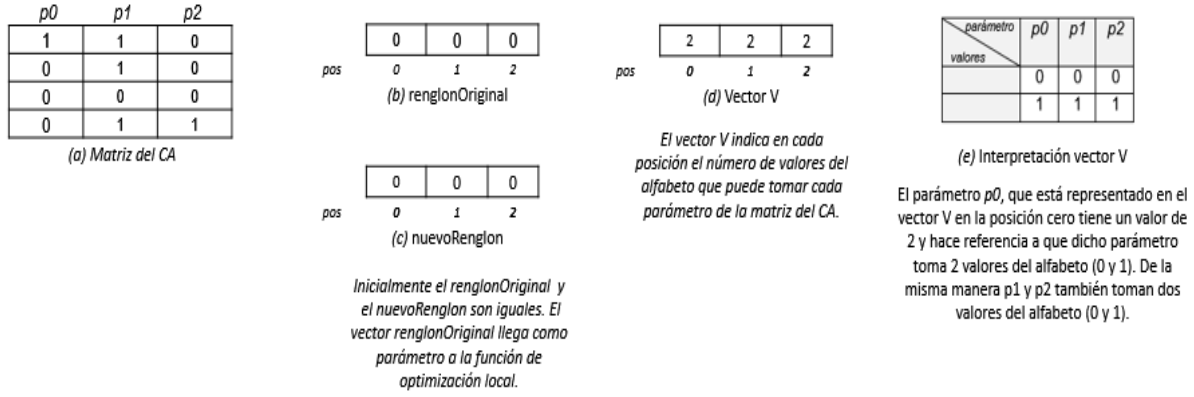


Figura 70 Inicialización objetos en función PMisLocal2RepetidoEnTodasColumnas

En el ciclo 1, se almacena en la variable mejorValor el valor del elemento que se encuentra en la posición cero del vector nuevoRenglon, tal como se muestra en la Figura 71(a). En la Figura 71(b) se muestra el vector nuevoRenglon que se quiere actualizar en su posición 0 con los valores descritos en la Figura 71(c). Si se observa, hay dos valores 0 y 1. Sin embargo, al reemplazar el 0 en el nuevoRenglon da como resultado el mismo vector nuevoRenglon por consiguiente no se tiene en cuenta. Luego, se reemplaza el 1 en nuevoRenglon. La Figura 71(d) muestra el nuevoRenglon con su valor actualizado.

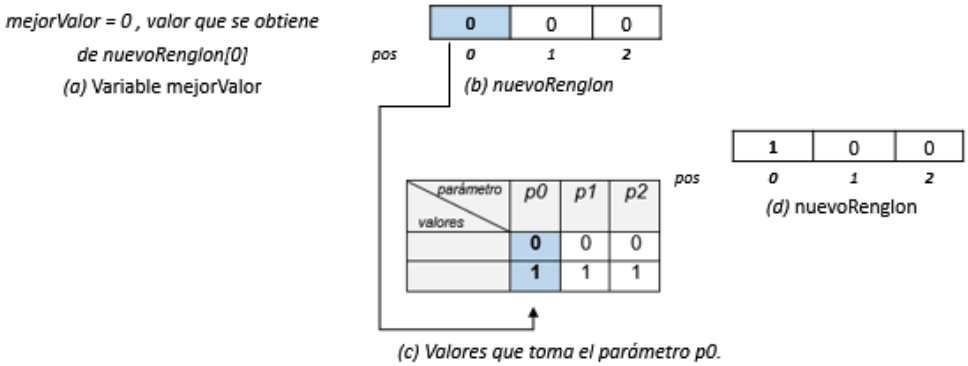


Figura 71 Inicialización vector nuevoRenglon en ciclo 1

El vector renglonOriginal que se muestra en la Figura 72(a) y el vector nuevoRenglon que se muestra en la Figura 72(b) se envían a la función quitarPonerRenglon indicada en la Figura 72(e), la cual retorna un valor que es almacenado en la variable faltan que se muestra en la Figura 72(f) y que representa el número de ceros registrados en la matrizP al quitar el renglonOriginal y al poner el nuevoRenglon. Como el valor de la variable faltan es igual al valor de la variable fitness que es igual a 3, no se realiza actualización alguna al valor de la variable fitness. En la Figura 72(g), el vector nuevoRenglon guarda en su posición cero, el valor que se había almacenado previamente en la variable mejorRenglon.

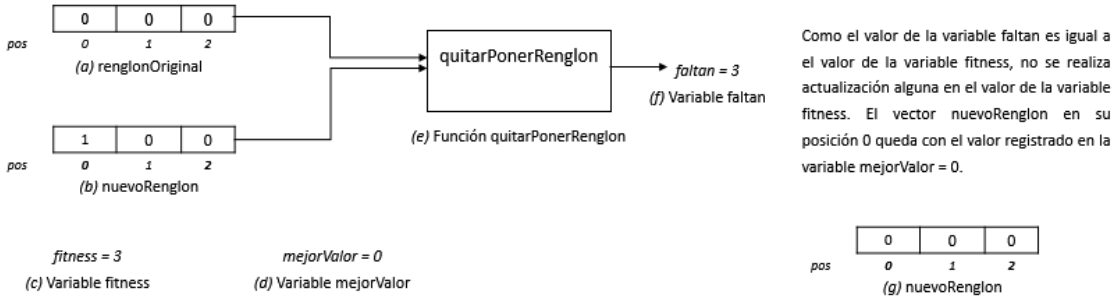


Figura 72 Aplicación de la función quitarPonerRenglon en el ciclo 1 para PMisLocal2RepetidoEnTodasColumnas

En el ciclo 2, se almacena en la variable mejorValor el valor del elemento que se encuentra en la posición uno del vector nuevoRenglon, tal como se muestra en la Figura 73(a). En la Figura 73(b) se muestra el vector nuevoRenglon que se quiere actualizar en su posición 1 con los valores descritos en la Figura 73(c). Si se observa, hay dos valores 0 y 1. Sin embargo, al reemplazar el 0 en el nuevoRenglon da como resultado el mismo vector nuevoRenglon por consiguiente no se tiene en cuenta. Luego, se reemplaza el 1 en nuevoRenglon. La Figura 73(d) muestra el nuevoRenglon con su valor actualizado.

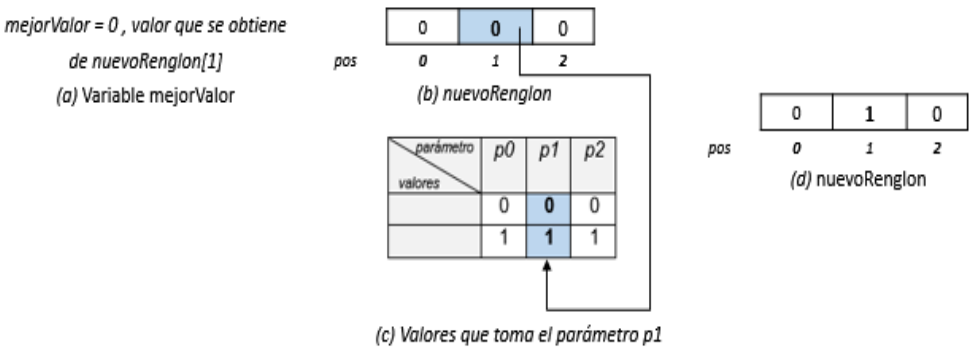


Figura 73 Inicialización vector nuevoRenglon en ciclo 2

El vector renglonOriginal que se muestra en la Figura 74(a) y el vector nuevoRenglon que se muestra en la Figura 74(b) se envían a la función quitarPonerRenglon indicada en la Figura 74(e), la cual retorna un valor que es almacenado en la variable faltan que se muestra en la Figura 74(f) y que representa el número de ceros registrados en la matrizP al quitar el renglonOriginal y al poner el nuevoRenglon. Como el valor de la variable faltan es mayor al valor de la variable fitness que es igual a 3, no se realiza actualización alguna al valor de la variable fitness. En la Figura 74(g), el vector nuevoRenglon guarda en su posición uno, el valor que se había almacenado previamente en la variable mejorRenglon.

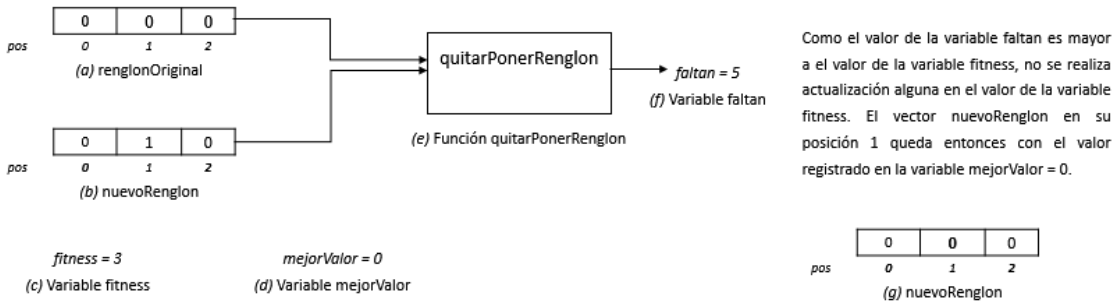


Figura 74 Aplicación de la función quitarPonerRenglon en el ciclo 2 para PMisLocal2RepetidoEnTodasColumnas

En el ciclo 3, se almacena en la variable mejorValor el valor del elemento que se encuentra en la posición dos del vector nuevoRenglon, tal como se muestra en la Figura 75(a). En la Figura 75(b) se muestra el vector nuevoRenglon que se quiere actualizar en su posición 2 con los valores descritos en la Figura 75(c). Si se observa, hay dos valores 0 y 1. Sin embargo, al reemplazar el 0 en el nuevoRenglon da como resultado el mismo vector nuevoRenglon por consiguiente no se tiene en

cuenta. Luego, se reemplaza el 1 en nuevoRenglon. La Figura 75(d) muestra el nuevoRenglon con su valor actualizado.

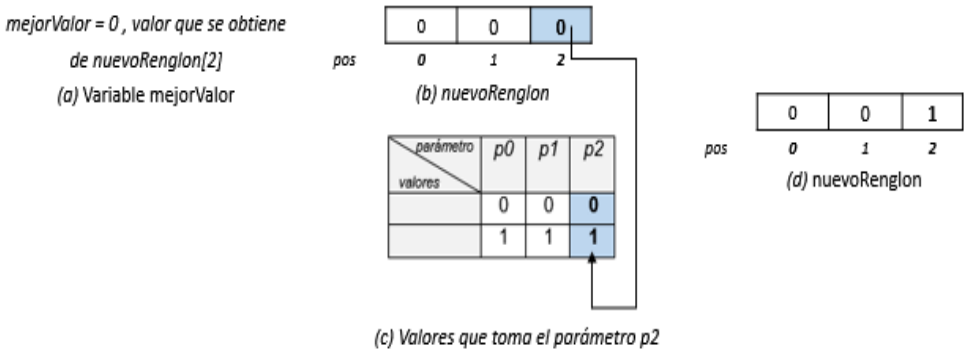


Figura 75 Inicialización vector nuevoRenglon en ciclo 3

El vector renglonOriginal que se muestra en la Figura 76(a) y el vector nuevoRenglon que se muestra en la Figura 76(b) se envían a la función quitarPonerRenglon indicada en la Figura 76(e), la cual retorna un valor que es almacenado en la variable faltan que se muestra en la Figura 76(f) y que representa el número de ceros registrados en la matrizP al quitar el renglonOriginal y al poner el nuevoRenglon. Como el valor de la variable faltan es igual al valor de la variable fitness que es igual a 3, no se realiza actualización alguna al valor de la variable fitness. En la Figura 76(g), el vector nuevoRenglon guarda en su posición dos, el valor que se había almacenado previamente en la variable mejorRenglon.

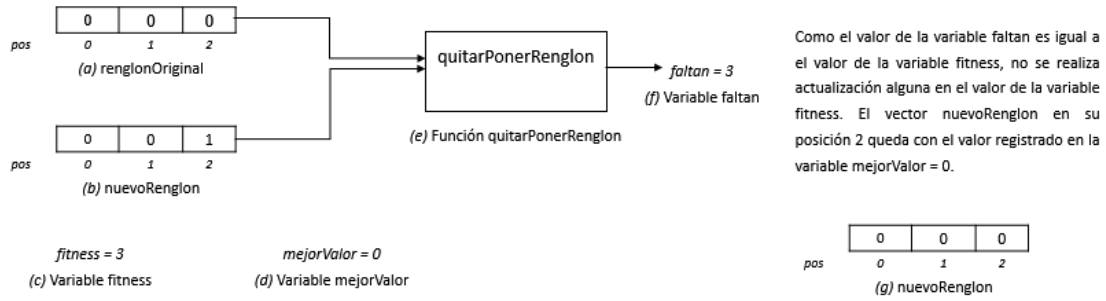


Figura 76 Aplicación de la función quitarPonerRenglon en el ciclo 3 para PMisLocal2RepetidoEnTodasColumnas

Para este ejemplo en particular, ninguno de los fitness calculados fue mejor que el fitness original que llegó a la función como parámetro. Por consiguiente, se hace el llamado a la función shake que se muestra en la Tabla 32, la cual cambia el valor

de un elemento que se encuentra en una posición escogida aleatoriamente en el vector nuevoRenglon.

La función de optimización local PMisLocal2RepetidoEnTodasColumnas finalmente retorna el vector nuevoRenglon.

El algoritmo híbrido GHSSA está diseñado para ser ejecutado en forma paralela. La arquitectura empleada para lograr la paralelización del algoritmo se muestra en el anexo A.

Capítulo 5

Resultados obtenidos

Para medir el desempeño del algoritmo propuesto, se realizó tres comparaciones experimentales a saber: la primera, compara el algoritmo propuesto GHSSA con los algoritmos de la Mejor Búsqueda Armónica Global (GHS) y Recocido Simulado (SA) en la construcción de CA uniformes (igual alfabeto para cada uno de los K parámetros del CA) y mixtos (diferente alfabeto para cada uno de los K parámetros del CA) de fuerza variable. La segunda, compara GHSSA con algoritmos del estado del arte que construyen Covering Arrays uniformes, binarios y de fuerza variable. La tercera, compara GHSSA con algoritmos del estado del arte que construyen Covering Arrays mixtos y de fuerza variable.

El primer experimento se llevó a cabo en un equipo de configuraciones básicas descritas a continuación: DELL Vostro 460, con 4GB de memoria RAM, procesador Intel Core i5-2400 3.1 GHz y sistema Operativo Windows 7 Home Premium. Los algoritmos GHS, SA y GHSSA se ejecutaron en la misma máquina, para garantizar que corrieran bajo las mismas condiciones. Por tal razón fue relevante registrar el tiempo en segundos que cada algoritmo empleó para construir el CA.

El segundo y tercer experimento se llevaron a cabo en distintos equipos con diferentes configuraciones hardware y software, con el único propósito de construir los CAs indicados por parte del algoritmo GHSSA. Para estos experimentos no se tuvo en cuenta el tiempo empleado en la construcción de los CAs, debido a que los algoritmos del estado del arte utilizados en las comparaciones fueron construidos la mayoría de las veces en supercomputadoras. Solo se tuvo en cuenta el valor de N que se logró obtener en la construcción de los CAs.

Las configuraciones de los equipos empleados en el segundo y tercer experimento fueron:

- DELL Vostro 460, con 4 GB de memoria RAM, procesador Intel Core i5-2400 3.1 GHz, 256 GB DD y sistema Operativo Windows 7 Home Premium de 64 bits.
- Dell Latitude 3470 con 16 GB de memoria RAM, procesador Intel Core i7 2.7 GHz, 1 TB DD y sistema operativo Windows 10 Pro de 64 bits.

- Lenovo Z50 con 16 GB de memoria RAM, procesador Intel Core i7-4510U 2.6 GHz. 1 TB DD y sistema operativo Windows 10 de 64 bits.
- Mac Book Air con 8 GB de memoria RAM, procesador Intel Core i5 1.6 GHz. 256 SSD y sistema operativo Windows 10 de 64 bits.

A continuación, se presenta cada uno de los resultados obtenidos para cada experimento.

Comparación experimental 1:

Cuando se ejecuta el programa para construir una configuración del CA específica y con un algoritmo en particular (GHS, SA o GHSSA), éste lanza entre 4 y 8 procesos que corren en paralelo, cada uno con diferente semilla para tener así diferentes espacios de búsqueda que permitan encontrar la solución para el CA. El proceso que obtenga el menor tiempo de construcción será el reportado para dicho CA.

En la Tabla 34 se muestra la primera comparación experimental realizada entre los algoritmos GHS y SA frente al algoritmo propuesto GHSSA. Fueron construidos 30 CAs, entre uniformes y mixtos, donde cada configuración del CA se construyó 3 veces con diferente algoritmo para un total de 90 construcciones.

La primera columna de la Tabla 34 presenta el identificador del CA. La segunda columna indica la configuración del CA a construir. La tercera columna indica el algoritmo con el que se construyó el CA. La cuarta columna indica el estado alcanzado al ejecutar el algoritmo. Este estado puede ser (S), indicando que la construcción del CA fue exitosa, (P) para construcción en proceso o en ejecución y (F) para construcción fallida. La quinta columna indica el fitness alcanzado por cada algoritmo dentro de un número de iteraciones predefinido cuyo valor fue de 25 millones. La sexta columna indica el tiempo en segundos que demoró la construcción del CA por un algoritmo específico (si lo construyó). La séptima columna denominada N' indica el número que pudo construir el algoritmo para una configuración del CA no construida inicialmente.

Id	MCA	Algoritmo	Estado	Fitness Alcanzado	T (seg)	N'
1	N5K4V2^4t2.ca	GHS	S	0	0,020	-
		GHSSA	S	0	0,011	-

		SA	S	0	0,020	-
2	N15K11V5^1-3^8-2^2t2.ca	GHS	P	-	-	20
		GHSSA	S	0	11,752	-
		SA	S	0	36,568	-
3	N19K9V4^5-3^4t2.ca	GHS	P	-	-	21
		GHSSA	S	0	37,393	-
		SA	S	0	186,513	-
4	N36K20V6^2-4^9-2^9t2.ca	GHS	P	-	-	44
		GHSSA	S	0	98,248	-
		SA	S	0	15,886	-
5	N37K16V6^4-4^5-2^7t2.ca	GHS	P	-	-	50
		GHSSA	S	0	25373,737	-
		SA	F	2	22090,729	-
6	N42K19V7^1-6^1-5^1-4^5-3^8-2^3t2.ca	GHS	P	-	-	47
		GHSSA	S	0	5,148	-
		SA	S	0	6,580	-
7	N30K61V4^15-3^17-2^29t2.ca	GHS	P	-	-	37
		GHSSA	S	0	3190,504	-
		SA	S	0	1723,872	-
8	N36K10V6^2-5^2-4^2-3^2-2^2t2.ca	GHS	P	-	-	41
		GHSSA	S	0	0,156	-
		SA	S	0	0,30	-
9	N64K8V8^2-7^2-6^2-5^2t2.ca	GHS	P	-	-	90
		GHSSA	S	0	968,846	-
		SA	S	0	445,215	-
10	N49K12V7^2-6^2-5^2-4^2-3^2-2^2t2.ca	GHS	P	-	-	57
		GHSSA	S	0	81,222	-
		SA	S	0	13,540	-
11	N100K5V10^2-8^3t2.ca	GHS	P	-	-	160
		GHSSA	S	0	11,656	-
		SA	S	0	9,480	-
12	N8K4V2^4t3.ca	GHS	P	-	-	10
		GHSSA	S	0	0,038	-
		SA	S	0	0,040	-
13	N10K5V2^5t3.ca	GHS	S	0	5,539	-
		GHSSA	S	0	0,052	-
		SA	S	0	0,010	-
14	N12K6V2^6t3.ca	GHS	S	0	1,833	-
		GHSSA	S	0	0,040	-
		SA	S	0	0,090	-
15	N12K7V2^7t3.ca	GHS	P	-	-	13
		GHSSA	S	0	0,01	-
		SA	S	0	0,125	-
16	N12K11V2^11t3.ca	GHS	S	0	47,158	-
		GHSSA	S	0	0,230	-

		SA	S	0	0,281	-
17	N15K12V2^12t3.ca	GHS	P	-	-	18
		GHSSA	S	0	0,320	-
		SA	S	0	0,370	-
18	N100K6V5^2-4^2-3^2t3.ca	GHS	P	-	-	130
		GHSSA	S	0	205,101	-
		SA	S	0	2398,327	-
19	N16K14V2^14t3.ca	GHS	P	-	-	19
		GHSSA	S	0	207,501	-
		SA	S	0	199,888	-
20	N13K11V2^11t3.ca	GHS	S	0	26,012	
		GHSSA	S	0	0,028	-
		SA	S	0	0,042	-
21	N360K7V10^1-6^2-4^3-3^1t3.ca	GHS	P	-	-	800
		GHSSA	S	0	350,699	-
		SA	S	0	75,169	-
22	N400K12V10^2-4^1-3^2-2^7t3.ca	GHS	P	-	-	710
		GHSSA	S	0	35,305	-
		SA	S	0	34,659	-
23	N16K5V2^5t4.ca	GHS	S	0	17,050	-
		GHSSA	S	0	0,162	-
		SA	S	0	0,032	-
24	N21K6V2^6t4.ca	GHS	P	-	-	22
		GHSSA	S	0	1,817	-
		SA	S	0	10,546	-
25	N24K12V2^12t4.ca	GHS	P	-	-	48
		GHSSA	S	0	0,848	-
		SA	S	0	1,040	-
26	N32K6V2^6t5.ca	GHS	P	-	-	33
		GHSSA	S	0	0,071	-
		SA	S	0	0,436	-
27	N42K7V2^7t5.ca	GHS	P	-	-	61
		GHSSA	S	0	0,034	-
		SA	S	0	0,423	-
28	N52K8V2^8t5.ca	GHS	P	-	-	60
		GHSSA	S	0	29,875	-
		SA	S	0	371,133	-
29	N64K7V2^7t6.ca	GHS	P	-	-	97
		GHSSA	S	0	0,043	-
		SA	S	0	0,084	-
30	N85K8V2^8t6.ca	GHS	P	-	-	160
		GHSSA	S	0	5,340	-
		SA	S	0	61,932	-

Tabla 34 Comparación experimental 1 entre los algoritmos GHS, GHSSA y SA

En la Tabla 34 se pueden evidenciar varias situaciones. Por ejemplo, al analizar los resultados de la Tabla 35 (extraída de la Tabla 34) , se puede observar que para la configuración de CA respectiva, los tres algoritmos lograron construir dicho CA, alcanzando un fitness de cero, que indica una construcción de CA exitosa. Se observa además que el algoritmo propuesto GHSSA construyó el algoritmo más rápido que GHS y SA. En 20 de las 30 pruebas realizadas, GHSSA logró ser más rápido que los otros dos algoritmos.

id	MCA	Algoritmo	Estado	Fitness Alcanzado	T (seg)	N'
16	N12K11V2^11t3.ca	GHS	S	0	47,158	-
		GHSSA	S	0	0,230	-
		SA	S	0	0,281	-

Tabla 35 Comparación individual para el CA N12K11V2^11t3.ca

Al analizar la tabla 36 (extraída de la Tabla 34), se observa que los tres algoritmos construyeron la configuración de CA propuesta. En 10 de las 30 pruebas realizadas, SA logró ser más rápido que los otros dos algoritmos. Sin embargo, la diferencia entre SA y GHSSA en la mayoría de los casos es mínima ya que se da en milisegundos.

id	MCA	Algoritmo	Estado	Fitness Alcanzado	T (seg)	N'
23	N16K5V2^5t4.ca	GHS	S	0	17,050	-
		GHSSA	S	0	0,162	-
		SA	S	0	0,032	-

Tabla 36 Comparación individual para el CA N16K5V2^5t4.ca

Al analizar la Tabla 37 (extraída de la Tabla 34), se observa que los algoritmos GHSSA y SA lograron construir el CA propuesto. Sin embargo, al ejecutar el algoritmo GHS, éste cayó en un óptimo local quedando así estancado dentro del número de iteraciones predefinido, lo que provocó que de forma intencional se detuviera el proceso. Al detener el proceso y observar que no era posible construir el CA indicado, el valor de N se incrementó y nuevamente se ejecutó el algoritmo. Para el ejemplo en cuestión, se pasa de probar con N = 15 a probar y ejecutar el algoritmo con N'=16. En caso de que caiga nuevamente en un óptimo local se detiene el proceso y nuevamente se incrementa N. Para este ejemplo, el algoritmo GHS logró construir la configuración con N' = 18.

De las 30 pruebas realizadas, GHS solo pudo construir dentro del número de iteraciones predefinido a 6 configuraciones de CA de manera exitosa. Mientras que en las 24 configuraciones restantes no logró construir el CA propuesto, por lo tanto se procedió a encontrar el N' respectivo. De lo anterior se puede observar claramente que el desempeño del algoritmo GHS para la construcción de CAs no es bueno.

id	MCA	Algoritmo	Estado	Fitness Alcanzado	T (seg)	N'
17	N15K12V2^12t3.ca	GHS	P	-	-	18
		GHSSA	S	0	0,320	-
		SA	S	0	0,370	-

Tabla 37 Comparación individual para el CA N15K12V2^12t3.ca

Al analizar la Tabla 38 (extraída de la Tabla 34), se observa que el algoritmo propuesto GHSSA construyó la configuración de CA indicada, mientras que el algoritmo SA falló en la construcción del CA para el número de iteraciones predefinido ya que el fitness final fue igual a 2, que indica el número de t-adas faltantes para que sea un CA. De las 30 construcciones de CAs realizadas, SA falló en una construcción.

En la Tabla 38 también se observa que el algoritmo GHS, dentro del número de iteraciones predefinidas, cayó en un óptimo local por consiguiente se detuvo intencionalmente el proceso. GHS logró construir el CA para un N' = 50.

id	MCA	Algoritmo	Estado	Fitness Alcanzado	T (seg)	N'
5	N37K16V6^4-4^5-2^7t2.ca	GHS	P	-	-	50
		GHSSA	S	0	25373,737	-
		SA	F	2	22090,729	-

Tabla 38 Comparación individual para el CA N100K6V5^2-4^2-3^2t3.ca

Comparación experimental 2:

Esta comparación se realizó para un marco de referencia de 12 configuraciones de CAs, con los mejores límites reportados por algunos algoritmos del estado del arte que construyen CAs uniformes y de alfabeto binario entre los que se encuentran DDA [68], TS [51], IPOG-F [32] y un algoritmo mejorado de SA [43]. La Tabla 39 muestra los resultados obtenidos.

La primera columna de la tabla muestra el Id del CA. La segunda columna muestra la configuración del CA binario a construir. De la columna 3 a la 6, se muestran los algoritmos con los que se comparó el algoritmo propuesto. La séptima columna de la tabla, denotada como Best (β) presenta la mejor cota reportada en la literatura. La octava columna muestra la cota obtenida por el algoritmo propuesto GHSSA (θ) y la novena columna presenta la diferencia entre el resultado obtenido por GHSSA y la mejor solución o cota reportada $\Delta = \theta - \beta$.

Id	CA binario	DDA	TS	IPOG-F	SA	Best (β)	GHSSA (θ)	Δ
1	N8K4V2^4t3.ca	8	8	8	8	8	8	0
2	N10K5V2^5t3.ca	10	10	11	10	10	10	0
3	N12K11V2^11t3.ca	20	12	18	12	12	12	0
4	N15K12V2^12t3.ca	21	15	19	15	15	15	0
5	N16K14V2^14t3.ca	27	16	21	16	16	16	0
6	N17K16V2^16t3.ca	27	17	22	17	17	19	+2
7	N16K5V2^5t4.ca	16	16	22	16	16	16	0
8	N21K6V2^6t4.ca	26	21	26	21	21	21	0
9	N24K12V2^12t4.ca	52	48	47	24	24	24	0
10	N32K6V2^6t5.ca	32	32	42	32	32	32	0
11	N42K7V2^7t5.ca	52	56	57	42	42	42	0
12	N52K8V2^8t5.ca	76	56	68	52	52	52	0

Tabla 39 Comparación experimental 2 entre DDA, TS, IPOG-F, SA mejorado y GHSSA

En 11 casos de 12, el algoritmo GHSSA logró construir las configuraciones establecidas. Solo para la configuración con Id = 6, el resultado obtenido fue cercano a la mejor cota reportada.

Comparación experimental 3:

Esta comparación se realizó para un marco de referencia de 19 configuraciones de CAs, con los mejores límites reportados por algunos de los más sobresalientes algoritmos del estado del arte que construyen CAs mixtos y de fuerza variable tales como IPOG[21], IPOG-F[32], SA-VNS[45], MiTS[47] y SA-H[69].
En la Tabla 40 se muestra los resultados obtenidos de la comparación. La primera columna muestra el Id del CA. La segunda columna muestra la configuración del CA a construir. De la columna 3 a la 7, se muestran los algoritmos con los que se

comparó el algoritmo propuesto. La octava columna de la tabla denotada como Best (β) presenta la mejor cota reportada en la literatura. La novena columna muestra la cota obtenida por GHSSA (θ) y la décima columna presenta la diferencia entre el resultado obtenido por GHSSA y la mejor solución o cota reportada en el estado del arte $\Delta = \theta - \beta$.

En 10 casos de 19, se observa que el algoritmo GHSSA fue capaz de igualar la mejor cota reportada en el estado del arte. En los 9 casos restantes la solución encontrada estuvo cercana a la mejor solución.

Id	MCA	IPOG	IPOG-F	MiTS	SA-H	SA-VNS	Best (β)	GHSSA (θ)	Δ
1	N15K11V5^1-3^8-2^2t2.ca	17	20	15	15	15	15	15	**0**
2	N19K9V4^5-3^4t2.ca	24	22	19	19	19	19	19	**0**
3	N20K75V4^1-3^39-2^35t2.ca	26	28	22	20	20	20	22	**+2**
4	N21K21V5^1-4^4-3^11-2^5t2.ca	25	27	22	21	21	21	23	**+2**
5	N30K61V4^15-3^17-2^29t2.ca	34	35	30	30	30	30	30	**0**
6	N28K19V6^1-5^1-4^6-3^8-2^3t2.ca	36	34	30	29	28	28	30	**+2**
7	N36K20V6^2-4^9-2^9t2.ca	38	39	36	36	36	36	36	**0**
8	N37K16V6^4-4^5-2^7t2.ca	44	42	38	38	37	37	37	**0**
9	N42K19V7^1-6^1-5^1-4^5-3^8-2^3t2.ca	42	42	42	42	42	42	42	**0**
10	N46K15V6^6-5^5-3^4t2.ca	55	53	50	-	46	46	50	**+4**
11	N45K18V6^7-4^8-2^3t2.ca	55	55	47	47	45	45	49	**+4**
12	N50K19V6^9-4^3-2^7t2.ca	64	60	51	51	50	50	53	**+3**
13	N64K8V8^2-7^2-6^2-5^2t2.ca	68	64	64	64	64	64	64	**0**
14	N80K9V4^5-3^4t3.ca	103	100	85	80	83	80	93	**+13**
15	N100K6V5^2-4^2-3^2t3.ca	106	103	100	100	100	100	100	**0**
16	N360K7V10^1-6^2-4^3-3^1t3.ca	372	361	360	360	360	360	360	**0**
17	N400K12V10^2-4^1-3^2-2^7t3.ca	405	402	400	400	400	400	400	**0**
18	N376K15V6^6-5^5-3^4t3.ca	452	426	-	-	376	376	426	**+50**
19	N500K8V8^2-7^2-6^2-5^2t3.ca	593	554	540	535	500	500	578	**+78**

Tabla 40 Comparación experimental 3 entre GHSSA y los enfoques IPOG, IPOG-F, MiTS, SA-H y SA-VNS

Capítulo 6

Conclusiones y Trabajo futuro

6.1 Conclusiones

Las siguientes conclusiones se obtuvieron del análisis realizado a los diferentes trabajos e investigaciones del estado del arte:

- Lo más complejo en la construcción de un CA es lograr encontrar el tamaño mínimo de filas del CA, lo que se conoce como CA óptimo. Por lo que la escogencia o aplicación de un método de construcción en particular dependerá de las características específicas que se disponga. Por ejemplo, Si no se dispone de tiempo para la construcción de los casos de prueba del Covering Array, se recomienda utilizar un método greedy, si hay disponibilidad de tiempo, los más indicados son los métodos meta-heurísticos.

- A pesar de que los métodos algebraicos proporcionan a menudo buenos resultados para la construcción de Covering Arrays, sólo son aplicables para casos muy particulares y generalmente pequeños. Si se usan para encontrar CAs de gran tamaño, se requiere mucho tiempo de ejecución y recursos computacionales.

- Los métodos greedy son más versátiles que los métodos algebraicos, ya que construyen CAs más grandes respecto al número de filas, columnas, alfabeto y fuerza. Sin embargo, raramente obtienen Covering Arrays óptimos y requieren menos tiempo computacional que los algoritmos metaheurísticos.

- Los últimos avances en métodos exactos, específicamente en el uso de coeficientes binomiales, ramificación y poda establecen una estrategia eficiente para construir CAs de fuerza 2 y señalan un camino promisorio de investigación en el uso de coeficientes trinomiales para obtener CAs de fuerza 3.

- Actualmente, los métodos basados en metaheurísticas son el enfoque más utilizado, debido en gran medida, a que generan los mejores resultados en la construcción de Covering Arrays. Estos métodos construyen generalmente CAs óptimos y trabajan con un número muy grande de filas, variables, alfabetos y con diferentes niveles de fuerza. Sin embargo, demandan mucho tiempo de cómputo y consumen más recursos de máquina que los otros métodos.

- La metaheurística más exitosa reportada a la fecha para la construcción de Covering Arrays uniformes y mixtos ha sido Recocido Simulado (Simulated Annealing). Las principales razones son: la posibilidad de escapar de óptimos locales gracias a la aceptación condicional de movimientos que no necesariamente mejoran la solución actual. Al no ser un algoritmo poblacional, logra evolucionar rápidamente hacía mejores regiones del espacio de búsqueda y permite una mezcla de varias funciones de vecindad.

- Debido al éxito que ha tenido Recocido Simulado en la construcción de CAs, se considera que, en los próximos años, esta metaheurística seguirá siendo estudiada e hibridada con diversas técnicas (metaheurísticas o no) para encontrar CAs óptimos o cercanos al óptimo.

- La extensa revisión de la literatura indicó que la metaheurística de la Búsqueda Armónica ha sido uno de los algoritmos menos explorados para la construcción de Covering Arrays.

- Teniendo en cuenta que día a día la complejidad de las tareas en las que se usan los CAs crece y que los CAs que se requieren construir tienen mayores alfabetos (V), mayor número de columnas (K) y mayor nivel de interacción (t), es preciso que la comunidad académica y científica del área, trabaje en formas eficientes de paralelizar el uso de las meta heurísticas que construyen CAs y los transfieran a las diferentes áreas de aplicación (pruebas de software, pruebas de hardware, seguridad, criptografía, entre otras).

A continuación, se presentan las conclusiones obtenidas con base en los experimentos realizados con el algoritmo propuesto y su comparación con métodos del estado del arte:

- El algoritmo de la Mejor Búsqueda Armónica Global (GHS) no tiene buenos resultados en la construcción de CAs para configuraciones donde el número de variables, alfabeto y fuerza es elevado.

- Ciertas configuraciones hardware y software de algunas máquinas donde se ejecutaron las pruebas, fueron un limitante a la hora de construir CAs mixtos de configuración compleja y de fuerza muy alta. Algunas pruebas realizadas para la construcción de CAs, sobrepasaron la carga de CPU y memoria RAM, lo que generó que el sistema operativo interrumpiera las construcciones que llevaban varias horas e incluso semanas en ejecución. El equipo Dell Vostro con 4 GB de RAM fue el que más presentó estas situaciones seguido por el equipo Lenovo. Sin embargo, se resalta que el algoritmo propuesto GHSSA logró construir diversas configuraciones de CAs en máquinas de especificaciones básicas teniendo en cuenta que la mayoría de CAs construidos por otros algoritmos del estado del arte, emplean supercomputadoras.

- Implementaciones iniciales del algoritmo propuesto GHSSA que no hacían uso de técnicas de optimización local, lograron construir ciertos CAs mixtos, sencillos, que se caracterizaban por tener pocas variables y de fuerza y alfabetos pequeños. Sin embargo, no logró construir configuraciones de CAs mixtos de configuraciones complejas, con un número considerable de variables y alfabetos y donde la fuerza de interacción era alta. Sin embargo, el desempeño del algoritmo GHSSA mejoró considerablemente al incluir al Recocido Simulado como técnica de optimización local y se logró construir configuraciones que inicialmente no fueron posible por las primeras versiones del algoritmo.

- La hibridación lograda en GHSSA (entre los algoritmos GHS y SA) fue acertada ya que permitió construir Covering Arrays uniformes y mixtos, de configuración simple y compleja, con diferentes niveles de fuerza ($2 \leq t \leq 6$) y cuyos resultados obtenidos fueron competitivos respecto a los reportados en el estado del arte.

- Finalmente dando respuesta a la pregunta de investigación formulada al principio de esta investigación, sí es posible crear Covering Arrays óptimos usando el algoritmo de la Mejor Búsqueda Armónica Global utilizando técnicas de optimización local.

- A nivel de recomendación se sugiere correr el algoritmo GHSSA utilizando un rango más amplio de semillas que no sean tan cercanas o vecinas, para

garantizar un espacio de búsqueda mucho más amplio. Esto debido a que en ciertas ocasiones que se corrió el algoritmo con semillas no tan cercanas, se logró encontrar la solución en un tiempo más rápido.

6.2 Trabajo Futuro

Incorporar el algoritmo propuesto GHSSA dentro de un aplicativo software que permita obtener los casos de prueba puntuales que se deben aplicar durante las pruebas funcionales para probar un método o un componente software.

Al observar que el uso de diferentes técnicas para manejar vecindad variable o búsquedas locales con distintos esquemas de vecindarios reportan buenos resultados para la construcción de CAs, los esquemas hyper-heurísticos [70] y de muestreo de múltiples descendientes (Multiple Offspring Sampling) [71] pueden ser el futuro en la construcción de CAs y MCAs de fuerza variable y diferentes variables y alfabetos y pueden ser tema de nuevas investigaciones y de mejora para el algoritmo GHSSA.

Referencias

[1] S. Maity, "Software Testing with Budget Constraints," in *Information Technology: New Generations (ITNG), 2012 Ninth International Conference on*, 2012, pp. 258-262.

[2] G. J. Myers*, et al.*, *The art of software testing, third edition*. Hoboken, N.J.: John Wiley & Sons, 2012.

[3] N. Changhai and H. Leung, "A Survey of Combinatorial Testing," *ACM Computing Surveys,* vol. 43, pp. 11:1-11:29, 2011.

[4] Y. Jun, "Backtracking Algorithms and Search Heuristics to Generate Test Suites for Combinatorial Testing," 2006, pp. 385-394.

[5] H. A. George*, et al.*, *Verificación de Covering Arrays*: Lambert Academic Publishing, 2010.

[6] H. Avila George, "Constructing Covering Arrays using Parallel Computing and Grid Computing," Ph.D., Departamento de Sistemas Informáticos y Computación, Universitad Politécnica de Valencia, Valencia, Spain, 2012.

[7] R. C. Turban and C. Adviser-Colbourn, *Algorithms for covering arrays*: Arizona State University, 2006.

[8] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology, RTI Project,* vol. 7007, 2002.

[9] J. Torres-Jimenez and I. Izquierdo-Marquez, "Survey of Covering Arrays," in *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on*, 2013, pp. 20-27.

[10] Z. Geem*, et al.*, "A New Heuristic Optimization Algorithm: Harmony Search," *SIMULATION,* vol. 76, pp. 60-68, 2001.

[11] Z. Geem and X.-S. Yang, "Harmony Search as a Metaheuristic Algorithm," in *Music-Inspired Harmony Search Algorithm*. vol. 191, ed: Springer Berlin / Heidelberg, 2009, pp. 1-14.

[12] M. Mahdavi*, et al.*, "An improved harmony search algorithm for solving optimization problems," *Applied Mathematics and Computation,* vol. 188, pp. 1567-1579, 2007.

[13] M. G. H. Omran and M. Mahdavi, "Global-best harmony search," *Applied Mathematics and Computation,* vol. 198, pp. 643-656, 2008.

[14] D. Zou*, et al.*, "A novel global harmony search algorithm for reliability problems," *Computers & Industrial Engineering,* vol. 58, pp. 307-316, 15th November, 2009 2009.

[15] S. Kirkpatrick and M. Vecchi, "Optimization by simmulated annealing," *science,* vol. 220, pp. 671-680, 1983.

[16] K. Meagher, "Non-Isomorphic Generation of Covering Arrays," University of Regina2002.

[17] J. Bracho-Rios*, et al.*, "A New Backtracking Algorithm for Constructing Binary Covering Arrays of Variable Strength," in *MICAI 2009: Advances in Artificial Intelligence*. vol. 5845, A. Aguirre*, et al.*, Eds., ed: Springer Berlin Heidelberg, 2009, pp. 397-407.

[18] B. Hnich*, et al.*, "Constraint Models for the Covering Test Problem," *Constraints,* vol. 11, pp. 199-219, 2006/07/01 2006.

[19] D. Lopez-Escogido*, et al.*, "Strength Two Covering Arrays Construction Using a SAT Representation," in *MICAI 2008: Advances in Artificial Intelligence*. vol. 5317, A. Gelbukh and E. Morales, Eds., ed: Springer Berlin Heidelberg, 2008, pp. 44-53.

[20] J. Torres-Jimenez*, et al.*, "A Branch & Bound Algorithm to Derive a Direct Construction for Binary Covering Arrays," in *Advances in Artificial Intelligence and Soft Computing*. vol. 9413,

G. Sidorov and S. Galicia-Haro, Eds., ed: Springer International Publishing, 2015, pp. 158-177.

[21] Y. Lei*, et al.*, "IPOG: A General Strategy for T-Way Software Testing," presented at the Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, 2007.

[22] J. Zhang*, et al.*, *Automatic Generation of Combinatorial Test Data*: Springer Publishing Company, Incorporated, 2014.

[23] G. O. H. Katona, "Two applications (for search theory and truth functions) of Sperner type theorems," *Periodica Mathematica Hungarica,* vol. 3, pp. 19-26, 1973/03/01 1973.

[24] D. J. Kleitman and J. Spencer, "Families of k-independent sets," *Discrete Mathematics,* vol. 6, pp. 255-262, 1973.

[25] M. A. Chateauneuf*, et al.*, "Covering Arrays of Strength Three," *Designs, Codes and Cryptography,* vol. 16, pp. 235-242, 1999/05/01 1999.

[26] K. Meagher and B. Stevens, "Group construction of covering arrays," *Journal of Combinatorial Designs,* vol. 13, pp. 70-77, 2005.

[27] A. Hartman, "Software and Hardware Testing Using Combinatorial Covering Suites," in *Graph Theory, Combinatorics and Algorithms*. vol. 34, M. Golumbic and I.-A. Hartman, Eds., ed: Springer US, 2005, pp. 237-266.

[28] C. J. Colbourn*, et al.*, "Products of mixed covering arrays of strength two," *Journal of Combinatorial Designs,* vol. 14, pp. 124-138, 2006.

[29] C. Colbourn, "Covering arrays from cyclotomy," *Designs, Codes and Cryptography,* vol. 55, pp. 201-219, 2010/05/01 2010.

[30] R. C. Bryce and C. J. Colbourn, "The density algorithm for pairwise interaction testing," *Software Testing, Verification and Reliability,* vol. 17, pp. 159-182, 2007.

[31] Y. Lei and K.-C. Tai, "In-Parameter-Order: A Test Generation Strategy for Pairwise Testing," presented at the The 3rd IEEE International Symposium on High-Assurance Systems Engineering, 1998.

[32] M. Forbes*, et al.*, "Refining the in-parameter-order strategy for constructing covering arrays," *Journal of Research of the National Institute of Standards and Technology,* vol. 113, pp. 287-297, 2008.

[33] A. Calvagna and A. Gargantini, "T-wise combinatorial interaction test suites construction based on coverage inheritance," *Software Testing, Verification and Reliability,* vol. 22, pp. 507-526, 2012.

[34] R. N. Kacker*, et al.*, "Combinatorial testing for software: An adaptation of design of experiments," *Measurement,* vol. 46, pp. 3745-3752, 2013.

[35] F. Glover and M. Laguna, *Tabu Search*: Kluwer Academic Publishers, 1997.

[36] M. Gendreau, "Recent Advances in Tabu Search," in *Essays and Surveys in Metaheuristics*. vol. 15, ed: Springer US, 2002, pp. 369-377.

[37] M. Gendreau and J. Y. Potvin, *Handbook of Metaheuristics*: Springer US, 2010.

[38] S. Luke. (2013). *Essentials of Metaheuristics (Second ed.)*.

[39] M. Dorigo and T. Stützle, *Ant Colony Optimization*: Bradford Company, 2004.

[40] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Neural Networks, 1995. Proceedings., IEEE International Conference on*, 1995, pp. 1942-1948 vol.4.

[41] M. B. Cohen*, et al.*, "Constructing test suites for interaction testing," presented at the Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, 2003.

[42] M. B. Cohen*, et al.*, "Constructing strength three covering arrays with augmented annealing," *Discrete Mathematics,* vol. 308, pp. 2709-2722, 2008.
[43] J. Torres-Jimenez and E. Rodriguez-Tello, "Simulated Annealing for constructing binary covering arrays of variable strength," in *Evolutionary Computation (CEC), 2010 IEEE Congress on*, 2010, pp. 1-8.
[44] J. Torres-Jimenez and E. Rodriguez-Tello, "New bounds for binary covering arrays using simulated annealing," *Information Sciences,* vol. 185, pp. 137-152, 2012.
[45] A. Rodriguez-Cristerna and J. Torres-Jimenez, "A Simulated Annealing with Variable Neighborhood Search Approach to Construct Mixed Covering Arrays," *Electronic Notes in Discrete Mathematics,* vol. 39, pp. 249-256, 2012.
[46] A. Muñoz Duarte, *Metaheurísticas*: S.L. - DYKINSON, 2007.
[47] A. L. González Hernández, "Un Algoritmo de Optimizacion Combinatoria para la Construccion de Covering Arrays Mixtos de Fuerza Variable," PhD., Laboratorio de Tecnologías de la Información, Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, 2013.
[48] A. Rodriguez-Cristerna*, et al.*, "Construction of Mixed Covering Arrays Using a Combination of Simulated Annealing and Variable Neighborhood Search," *Electronic Notes in Discrete Mathematics,* vol. 47, pp. 109-116, 2015.
[49] J. Torres-Jimenez*, et al.*, "A two-stage algorithm for combinatorial testing," *Optimization Letters,* pp. 1-13, 2016.
[50] K. J. Nurmela, "Upper bounds for covering arrays by tabu search," *Discrete Applied Mathematics,* vol. 138, pp. 143-152, 2004.
[51] R. A. Walker Ii and C. J. Colbourn, "Tabu search for covering arrays using permutation vectors," *Journal of Statistical Planning and Inference,* vol. 139, pp. 69-80, 2009.
[52] L. Gonzalez-Hernandez*, et al.*, "Construction of Mixed Covering Arrays of Variable Strength Using a Tabu Search Approach," in *Combinatorial Optimization and Applications*. vol. 6508, W. Wu and O. Daescu, Eds., ed: Springer Berlin Heidelberg, 2010, pp. 51-64.
[53] L. Gonzalez-Hernandez, "New bounds for mixed covering arrays in t-way testing with uniform strength," *Information and Software Technology,* vol. 59, pp. 17-32, 2015.
[54] X. Yu and M. Gen. (2010). *Introduction to Evolutionary Algorithms*.
[55] J. Stardom, *Metaheuristics and the Search for Covering and Packing Arrays [microform]*: Thesis (M.Sc.)--Simon Fraser University, 2001.
[56] E. Rodriguez-Tello and J. Torres-Jimenez, "Memetic Algorithms for Constructing Binary Covering Arrays of Strength Three," in *Artifical Evolution*. vol. 5975, P. Collet*, et al.*, Eds., ed: Springer Berlin Heidelberg, 2010, pp. 86-97.
[57] S. Sabharwal*, et al.*, "Construction of t-way covering arrays using genetic algorithm," *International Journal of System Assurance Engineering and Management,* pp. 1-11, 2016.
[58] M. Dorigo*, et al.*, "Ant system: optimization by a colony of cooperating agents," *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on,* vol. 26, pp. 29-41, 1996.
[59] T. Shiba*, et al.*, "Using artificial life techniques to generate test cases for combinatorial testing," in *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, 2004, pp. 72-77 vol.1.
[60] D. M. Cohen*, et al.*, "The Automatic Efficient Test Generator (AETG) system," in *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, 1994, pp. 303-309.

[61] C. Xiang*, et al.*, "Building Prioritized Pairwise Interaction Test Suites with Ant Colony Optimization," in *Quality Software, 2009. QSIC '09. 9th International Conference on*, 2009, pp. 347-352.

[62] B. S. Ahmed*, et al.*, "The development of a particle swarm based optimization strategy for pairwise testing," *Journal of Artificial Intelligence,* vol. 4, pp. 156-165, 2011.

[63] B. S. Ahmed and K. Z. Zamli, "A variable strength interaction test suites generation strategy using Particle Swarm Optimization," *Journal of Systems and Software,* vol. 84, pp. 2171-2185, 2011.

[64] B. S. Ahmed*, et al.*, "Application of Particle Swarm Optimization to uniform and variable strength covering array construction," *Applied Soft Computing,* vol. 12, pp. 1330-1347, 2012.

[65] T. Mahmoud and B. S. Ahmed, "An efficient strategy for covering array construction with fuzzy logic-based adaptive swarm optimization for software testing use," *Expert Systems with Applications,* vol. 42, pp. 8753-8765, 2015.

[66] A. R. A. Alsewari and K. Z. Zamli, "A harmony search based pairwise sampling strategy for combinatorial testing," *International Journal of the Physical Sciences,* vol. 7, pp. 1062-1072, 2012.

[67] X. Bao*, et al.*, "Combinatorial Test Generation Using Improved Harmony Search Algorithm," *International Journal of Hybrid Information Technology,* vol. 8, pp. 121-130, 2015.

[68] R. C. Bryce and C. J. Colbourn, "A density-based greedy algorithm for higher strength covering arrays," *Softw. Test. Verif. Reliab.,* vol. 19, pp. 37-53, 2009.

[69] H. Avila-George*, et al.*, "Simulated Annealing for Constructing Mixed Covering Arrays," in *Distributed Computing and Artificial Intelligence: 9th International Conference*, S. Omatu*, et al.*, Eds., ed Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 657-664.

[70] E. K. Burke*, et al.*, "Hyper-heuristics: A survey of the state of the art," *Journal of the Operational Research Society,* vol. 64, pp. 1695-1724, 2013.

[71] A. LaTorre*, et al.*, "Multiple Offspring Sampling in Large Scale Global Optimization," in *2012 IEEE Congress on Evolutionary Computation*, 2012, pp. 1-8.