# Aplicación web para la automatización de pruebas de software de caja negra basado en arreglos de cobertura mixtos a través de un enfoque de software como servicio


**Andrés Rodrigo López Realpe**

**Jorge Armando Muñoz Ordoñez**

**Director: PhD. Carlos Alberto Cobos Lozada**

**Universidad del Cauca**
**Facultad de Ingeniería Electrónica y Telecomunicaciones**
**Departamento de Sistemas**
**Grupo de I+D en Tecnologías de la Información (GTI)**
**Área de interés en Gestión de la Información**
**Popayán, mayo de 2022**

# RESUMEN

Actualmente la humanidad se ha visto inmersa en el mundo tecnológico debido a diferentes factores, ya sea por la expansión del internet, la globalización o acontecimientos importantes tales como, la cuarentena obligatoria por la crisis sanitaria debido al COVID-19, lo cierto es que la sociedad se ha volcado a ser estrechamente aliada de la tecnología. Teniendo en cuenta lo mencionado es preciso afirmar que el sector tecnológico abarca una gran porción del mercado actual, haciéndolo cada vez más dependiente de las herramientas software, esto implica una fuerte necesidad de desarrollar software de calidad que soporte dicha dependencia. El desarrollo de software tiene que responder a los retos impuestos por el sector ya que se encarga de controlar importantes operaciones económicas, industriales y de salud, en las cuales un error podría costar grandes pérdidas de dinero o de hasta vidas humanas. Eso se convirtió en motivo de interés para este trabajo, enfocar los esfuerzos en reducir los errores del producto software, automatizando la creación de interacciones de prueba que a su vez al someter dicho producto a estas pruebas se puede garantizar la calidad en la funcionalidad de un software.

En este trabajo se presenta la elaboración de una aplicación web y dos algoritmos que permiten automáticamente diseñar casos de prueba que garanticen una cobertura determinada por el usuario tester, teniendo en cuenta que, si hay una mayor cobertura en las pruebas, el margen de error en el producto final será menor. Para este fin se buscó realizar pruebas funcionales de caja negra, con la ayuda de los Arreglos de Cobertura Mixta (MCA). Utilizando MCA es posible tener la mayor cobertura en las pruebas con una menor cantidad de esfuerzo. Alineando el proyecto a las demandas actuales del mercado la aplicación Web se diseñó orientada a microservicios, lo cual le brinda escalabilidad y estabilidad a largo plazo, en el aplicativo el usuario tester tiene la oportunidad de interactuar con una interfaz sencilla e intuitiva que permite generar proyectos y en ellos casos su prueba.

La aplicación web fue sometida a evaluación por usuarios interesados en su uso y posible comercialización y además de evaluarla positivamente, dieron recomendaciones y sugerencias valiosas, algunas de las cuales se incluyeron en la aplicación y otras se dejaron como trabajo futuro. Con los resultados se puede

observar la gran importancia que tiene el proceso de prueba en el desarrollo de un producto o aplicativo software.

## TABLA DE CONTENIDO

## LISTA DE FIGURAS

## LISTA DE TABLAS



## LISTA DE ANEXOS

Esta página ha sido dejada intencionalmente en blanco.

# CAPÍTULO 1

## 1 INTRODUCCIÓN

### 1.1 PLANTEAMIENTO DEL PROBLEMA

El constante crecimiento de la industria y las necesidades de procesamiento de grandes cantidades de datos hacen cada vez más intrincado el mundo laboral, por ello día a día se hace más necesario contar con herramientas software para organizar y facilitar el manejo de la información. De acuerdo con Kovačević [1], actualmente los sistemas software controlan y administran grandes sistemas económicos, de salud, transporte, entre otros, y un error en dichos sistemas podría costar grandes cantidades de dinero e incluso la pérdida de vidas humanas. Por lo anterior, los errores en dichos sistemas son inadmisibles y como consecuencia, las empresas actualmente invierten grandes cantidades de dinero en pruebas de software para detectar errores de diseño y codificación, entre otras, para garantizar la fiabilidad del software que se entrega como producto final [2], [3].

Estudios previos han podido estimar que, del costo total de la producción de software, aproximadamente el 50% es destinado para la realización de pruebas [4], lo cual significa un incremento considerable en el costo de producción para las empresas desarrolladoras de software, dicho costo de pruebas oscila entre los 22.2 a 59.5 mil millones de dólares estadounidenses al año [5]. Una cifra extremadamente alta para las empresas, además no simplemente se trata de este incremento en costos, el tiempo necesario para la realización de estas, retarda el tiempo de entrega, repercutiendo negativamente en la eficiencia y rentabilidad de la empresa. Mitigar los retrasos y reducir el costo de las pruebas software es un reto muy importante para los investigadores de este campo.

La literatura reporta diferentes técnicas para probar software a través de pruebas funcionales y no funcionales. Las pruebas no funcionales se centran en aspectos importantes del comportamiento del producto, pero que no están relacionados con las funciones que realiza el sistema. Por el contrario, las pruebas funcionales se definen teniendo como fuente los requisitos del sistema, por medio de estas se puede validar y verificar que el producto cumple con lo especificado. Dentro de las pruebas funcionales se puede encontrar una gran variedad de técnicas, entre ellas las pruebas de caja negra, que a su vez se pueden realizar de diversas maneras, una alternativa la constituyen las pruebas exhaustivas que consideran la totalidad

de los casos de prueba, sin embargo, son las más costosas de realizar, en tiempo y dinero.

En el enfoque exhaustivo se evalúan todas las combinaciones posibles de valores en los parámetros de entrada de un método (procedimiento o función), por ejemplo, si se espera probar un método que tiene 4 parámetros o argumentos con 5 posibles valores cada uno, se tendrían que realizar 1024 ($4^5$) casos de prueba. El crecimiento exponencial del número de casos de prueba en el enfoque exhaustivo lo hace inviable para la gran mayoría de compañías desarrolladoras de software, por su costo en tiempo y dinero. Una estrategia para manejar esta situación es reducir la cantidad de casos de prueba, de manera que permita reducir la cantidad de tiempo y dinero invertido en el personal que hace el diseño y ejecución de estas pruebas [6].

Los casos de prueba son diseñados por los Testers (o probadores de software, encargados de planificar y llevar a cabo pruebas de software para comprobar si una determinada funcionalidad de una aplicación opera correctamente) quienes con su conocimiento y experiencia buscan aumentar la cobertura en la detección de fallos y disminuir el costo de las pruebas. Anand et al. [7] describen varias técnicas para realizar diferentes pruebas, entre ellas: pruebas al azar, pruebas adaptativas al azar, ejecución simbólica y programa de pruebas de cobertura estructural, generación de casos de prueba basada en modelos, pruebas basadas en la búsqueda y pruebas combinatorias. En el contexto de las pruebas de caja negra, las pruebas combinatorias son las más eficientes en el diseño de casos de prueba, dentro de estas se destaca el uso de arreglos de cobertura (CA) y más específicamente el uso de arreglos de cobertura mixtos (MCA), los cuales han demostrado con éxito la disminución en el número de casos de prueba (menor costo y tiempo), garantizando la evaluación de la interacción de un conjunto previamente definido de parámetros a probar con la mayor cobertura posible y basado en un concepto denominado fuerza que se usa para expresar la interacción esperada entre los parámetros del método que se va a evaluar [8],[9].

En el mercado se encuentran varias herramientas que permiten soportar el desarrollo de pruebas de caja negra, como Junit [1] y Nunit[2]. Sin embargo, dichas herramientas no proporcionan el soporte requerido para diseñar los casos de prueba, es decir el usuario es quien define los casos de prueba con base en su experiencia y no cuenta con alguna guía o herramienta que le ayude a definir el menor número de casos que permita detectar el mayor número de fallos.

Por lo anterior, un gran reto en el proceso de desarrollo de software específicamente en la etapa de pruebas consiste en identificar ¿Cuáles deben ser las características de un servicio disponible en la Web para soportar el diseño de

[1] https://junit.org/junit5/
[2] https://nunit.org/

casos de prueba en el contexto de pruebas de caja negra en Junit y Nunit que otorgue un tiempo de respuesta promedio no mayor a un segundo? Conforme se mencionó previamente, Disminuir el tiempo y esfuerzo (medido en dinero u horas hombre) que conlleva el proceso de pruebas es un aspecto de vital importancia para las empresas desarrolladoras de software.

Con el objetivo de delimitar el alcance de la presente investigación, este proyecto se enfocó en el uso de los arreglos de cobertura mixtos o MCA para la construcción de casos de prueba orientados a pruebas de caja negra, los cuales han mostrado según la literatura ser la opción combinatoria más eficiente. Adicionalmente se presenta un aplicativo Web construido con arquitectura de microservicios, que es de fácil uso para los testers de cualquier compañía que se dedique a desarrollar y probar software usando pruebas de caja negra.

## 1.2 APORTES DEL PROYECTO

A nivel de investigación, la principal contribución se realizó en el área de pruebas de software de caja negra, generando conocimiento en la adaptación e implementación de algoritmos para construcción y post optimización de arreglos de cobertura mixtos con base en el estado del arte.

De acuerdo con la literatura las alternativas a la hora de generar CA o MCA considera diferentes factores que en un sentido de implementación practico no aportan claramente al objetivo planteado en este trabajo, por ejemplo, en [2] se mencionan configuraciones y algoritmos que buscan encontrar u optimizar límites superiores de MCA de grandes dimensiones, lo cual no contribuye a la realización practica de casos de prueba, ya que son demasiadas interacciones (fuerza muy alta para el contexto de pruebas de software) y el tiempo de respuesta para crear estas configuraciones de MCA sería extremadamente alto con respecto a los requerido en un contexto real de pruebas de software en la actualidad.

Por lo mencionado anteriormente en este trabajo se buscó orquestar ciertas características en los algoritmos que permiten disminuir los tiempos de procesamiento y optimizar la calidad de los arreglos de cobertura que se usan en la definición de los casos de prueba. Adicionalmente la adaptación realizada permite distribuir la carga de las solicitudes de los usuarios en dos tareas, una con un tiempo de respuesta rápida basado en la post optimización de MCA que subsumen (un MCA que tiene más parámetros y/o un mayor alfabeto en los parámetros de lo que requiere un tester en una prueba de software especifica) la solicitud del tester y otra que se ejecuta en el fondo con una respuesta menos rápida pero que busca crear un MCA tan óptimo como sea posible, y que sirve para atender solicitudes futuras con requisitos similares.

En el ámbito de innovación y desarrollo, se construyó una aplicación web escalable con una arquitectura orientada a microservicios que automatiza el

proceso de diseño de casos de prueba de caja negra. Esta aplicación se constituye en una herramienta importante para los testers (probadores) que usan Junit y Nunit, que además es capaz de dar respuestas en un tiempo promedio menor a un segundo.

Con el fin que la aplicación sea usada libremente y con el tiempo convertirla en un emprendimiento, se dejara el código fuente en un repositorio público de GitHub.

## 1.3 Objetivos

A continuación, se presentan los objetivos planteados y aprobados por el Concejo de Facultad de la Facultad de Ingeniería Electrónica y Telecomunicaciones al inicio del proyecto.

### 1.3.1 Objetivo General

Proponer una aplicación web para soportar la automatización de pruebas de software de caja negra basado en arreglos de cobertura mixtos a través de un enfoque de software como servicio y una arquitectura orientada a servicios.

### 1.3.2 Objetivos Específicos

- Desarrollar una aplicación web con una arquitectura basada en microservicios que permita la construcción de arreglos de cobertura mixtos y su post optimización de acuerdo con las solicitudes de los usuarios Testers.
- Adaptar un algoritmo de construcción de arreglos de cobertura mixtos del estado del arte e integrarlo dentro de la arquitectura propuesta para alimentar un repositorio de estos arreglos.
- Adaptar un algoritmo de optimización de arreglos de cobertura mixtos del estado del arte e integrarlo dentro de la arquitectura propuesta.
- Determinar el nivel de satisfacción de los usuarios Testers (probadores) de software al usar el complemento propuesto en dos empresas de la región o el país.

## 1.4 Resultados Obtenidos

A continuación, se resumen los resultados obtenidos en la investigación de acuerdo con los objetivos planteados:

1. **Monografía de trabajo de grado**: Hace referencia al presente documento en el cual se plasma el estudio, proceso y desarrollo de los objetivos presentados previamente, con los cuales se busca la creación de una aplicación Web que facilita la implementación de pruebas unitarias haciendo uso de los arreglos de cobertura mixtos (MCA). En este documento también se encuentra descrita la arquitectura de la aplicación Web y los algoritmos implementados para la mejora del proceso de creación y post optimización de los MCA. Además, los

resultados de la evaluación del nivel de satisfacción de un conjunto de testers. Finalmente, presenta las conclusiones de la investigación y el trabajo futuro que el grupo de investigación espera realizar con lo avanzado en el tema.

2. **Aplicación Web** para la Creación de casos de prueba basado en MCA, compuesto por:
   - **Código Fuente:** Hace referencia al código fuente con el que se desarrollaron los componentes de la aplicación. Estos componentes de código fueron implementados en Angular para el Frontend y .NET Core para el Backend usando C# (**Anexo A**).
   - **Documentación del código fuente:** Hace referencia a la documentación realizada sobre el código y los componentes de la aplicación desarrollada. Esta documentación fue generada con ayuda de la herramienta Doxygen (**Anexo B**).
   - **Repositorio de MCA:** Hace referencia al repositorio de arreglos de cobertura mixtos generados tanto por el algoritmo de construcción (y que se usa inicialmente para poblar el repositorio con un conjunto de casos de prueba comunes, y luego, con el uso de la aplicación para crear nuevos MCA que son solicitados por los testers) como por el algoritmo de post optimización (**Anexo C**).
   - **Reportes de Serenity:** Hace referencia a los reportes HTML correspondientes a los consumos de las APIs donde se encuentra los resultados de las pruebas a los algoritmos propuestos, generado por la herramienta SerenityBdd mediante Maven y Junit (**Anexo D**).
   - **Tablas de resultados:** Hace referencia a las tablas en Excel concernientes a los resultados obtenidos en las pruebas a los algoritmos propuestos (**Anexo E**).

3. **Articulo**: articulo con la descripción detallada del algoritmo de creación de MCA y los resultados de su uso en la construcción de casos de prueba comúnmente usados por los Testers (**Anexo F**).

### 1.5 Estructura de la monografía

A continuación, se describe de manera general el contenido y organización de la presente monografía:

**CAPITULO 1: INTRODUCCIÓN**: Hace referencia al presente capitulo que introduce el tema de investigación, presenta la pregunta de investigación que originó el trabajo, resume los aportes realizados por la investigación, presenta los objetivos (general y específicos) definidos previamente en el proyecto, resume los resultados obtenidos y finalmente presenta la organización de la monografía.

**CAPITULO 2: CONTEXTO TEÓRICO Y ESTADO DEL ARTE**: En este capítulo se presentan la contextualización especifica de los conceptos principales, como CA,

MCA y pruebas de caja negra, también los algoritmos que se implementan para su construcción y su post optimización.

**CAPITULO 3: APLICACIÓN WEB:** En este capítulo se presenta el diseño detallado del sistema desarrollado describiendo su arquitectura general, así como especifica de cada microservicio, incluyendo los flujos, así como sus aspectos más importantes.

**CAPITULO 4: GENERADOR MCA:** En este capítulo se muestra el proceso de implementación y optimización del generador de arreglos de cobertura implementado, incluyendo su funcionamiento y acople con el resto del sistema.

**CAPITULO 5: POST-OPTIMIZADOR:** En este capítulo se muestra el proceso de implementación y optimización del algoritmo de Post-Optimización de arreglos de cobertura implementado, incluyendo su funcionamiento y acople con el resto del sistema.

**CAPITULO 6: ANALISIS DE RESULTADOS Y EVALUACIÓN DE SATISFACCIÓN:** En este capítulo se presenta la evaluación de la satisfacción de los usuarios por la aplicación web.

**CAPITULO 7: CONCLUSIONES Y TRABAJOS FUTUROS**: En este capítulo se presentan las conclusiones obtenidas al finalizar el trabajo de grado e ideas que el grupo de investigación espera realizar en un trabajo futuro.

**CAPITULO 8: BIBLIOGRAFIA**: Este último capítulo contiene las referencias bibliográficas de los artículos y libros consultados para la realización del proyecto.

# CAPÍTULO 2

## 2 CONTEXTO TEÓRICO Y ESTADO DEL ARTE

A continuación, se presenta una breve contextualización de la temática central de esta propuesta, que son las pruebas de caja negra como una de las herramientas disponibles en el contexto de las pruebas de software. Más específicamente las pruebas de caja negra donde los casos de prueba se generan con un enfoque combinatorio basado en el uso de arreglos de cobertura mixtos, con un especial énfasis en el proceso de construcción y post optimización de estos arreglos.

### 2.1 PRUEBAS DE SOFTWARE

Las pruebas de software representan la agrupación de técnicas o metodologías en constante evolución e investigación que tienen como objetivo principal proporcionar a las partes interesadas, información objetiva sobre el estado o calidad del software. Realizar pruebas a los productos software es de vital importancia si se quiere garantizar la más alta calidad en dichos productos, privilegiando soluciones que conllevan el menor costo y tiempo de realización [4]. A nivel de investigación e innovación, es un área muy activa que busca continuamente, nuevas técnicas y metodologías que se adapten cada vez mejor a las necesidades del mercado.

Existen varios tipos de pruebas, varias de ellas pueden ser ejecutadas en cualquier etapa del desarrollo del proyecto, adaptándose eso sí a los diferentes modelos o metodologías existentes para la elaboración de los productos software. La existencia variada de modelos permite el uso de múltiples tipos de pruebas que conllevan a una mejora continua en la calidad del software [10].

La literatura organiza las pruebas en dos grandes tipos: 1. Las pruebas funcionales que incluyen las unitarias, de aceptación, de integración y de regresión, y 2. Las pruebas no funcionales que incluyen las de carga, de estrés, de escalabilidad y de portabilidad. En la actualidad existen varias herramientas para realizar estas pruebas, por ejemplo, JMeter[3] (pruebas de rendimiento), Selenium[4]

[3] https://jmeter.apache.org/
[4] https://www.selenium.dev/

(pruebas automatizadas), JUnit[5]- Nunit[6] (pruebas unitarias) y TestNG[7] (ejecutor de pruebas de Selenium).

En el marco de las pruebas unitarias, pruebas que trabajan con pequeñas unidades de código (métodos, funciones o procedimientos) para comprobar que funcionan correctamente, se destacan dos tipos dependiendo de la forma como se estructuran los casos de prueba: 1. Enfoque estructural o de caja blanca y 2. Enfoque funcional o de caja negra.

Las pruebas de caja blanca se enfocan en el código fuente y buscan verificar la estructura interna del componente software con independencia de la funcionalidad de este. Es decir, asegurar que los resultados sean correctos. A medida que se ejecutan los casos de prueba se determinan las instrucciones o bloques donde existen errores [10]. Estas pruebas son muy complejas ya que implican la ejecución del 100% de los caminos lógicos de las rutinas (métodos, procedimientos o funciones), haciéndolas muy costosas.

Las pruebas de caja negra son un método en el que el Tester no conoce la estructura, diseño o implementación interna del componente software que se está probando [11]. Es decir, en estas pruebas se proporcionan las entradas para el componente software bajo prueba, se realiza la ejecución del componente con estos valores de entrada y se determina si las salidas producidas son equivalentes a las esperadas. En las pruebas de caja negra deben estar muy bien definidas sus entradas y salidas, es decir, su interfaz. Por lo que no se precisa definir ni conocer los detalles internos del funcionamiento del componente software que se está probando.

En las pruebas de caja negra existen diferentes técnicas o métodos para la construcción de los casos de prueba, entre ellos: La partición de clases de equivalencia, los árboles de clasificación, el análisis de valores límites, las pruebas de sintaxis, las tablas de decisión, las transiciones de estados, las pruebas de escenarios (caso de uso, historias de usuario) y las combinatorias. Las técnicas combinatorias son el enfoque de la presente investigación y basándose en el estándar para las pruebas de software ISO/IEC/IEEE 29119[8] se puede encontrar cuatro tipos de pruebas combinatorias: las que hacen todas las combinaciones (all combinations testing), las que hacen todas las parejas (pair-wise testing), las que hacen la selección de la base más probable (base choice testing) y las que hacen la selección de cada opción (each choice testing).

La cobertura alcanzada es la medida clave para cualquiera de las técnicas combinatorias previamente mencionadas. Esta medida busca definir qué tanta

[5] https://junit.org/junit5/
[6] https://nunit.org/
[7] https://testng.org/doc/
[8] https://in2test.lsi.uniovi.es/gt26/?lang=es

cantidad de posibles errores se pueden detectar con las pruebas realizadas. En este sentido, los arreglos de cobertura, como herramienta de prueba combinatoria alternativa a la realización de todas las combinaciones (cobertura total con el mayor costo posible), y en especial los arreglos de cubrimiento mixto han mostrado obtener un alto valor de cobertura con el menor costo posible, representado, en un menor número de casos de prueba (ver experiencia del NIST[9]– National Institute of Standards and Technology, en relación con el uso de arreglos de cobertura para la detección de errores en diferentes tipos de software).

## 2.2 Herramientas de diseño de casos de prueba para caja negra

Actualmente existe una gran variedad de herramientas para realizar pruebas de software [12], muchas de ellas se encuentran orientadas a pruebas unitarias, pruebas de regresión, pruebas funcionales para aplicaciones web, entre otras. A continuación, se presentan los trabajos previos más relacionados con el enfoque de esta investigación. El trabajo en [13] presenta diversas herramientas comerciales que permiten realizar diferentes tipos de pruebas de caja negra, como lo es Testlink[10] y Testopia[11]. Estas herramientas están enfocadas a la gestión, mas no al diseño de los casos de prueba, como si es el enfoque del presente trabajo.

Testlink es una herramienta web que permite la gestión, organización de casos y planes de prueba. En esta herramienta se pueden ejecutar casos de prueba creados manualmente. Testlink cuenta con tres bases fundamentales para su funcionamiento; **Test Project** (incluye pruebas de especificación, requerimientos y palabras clave), **Test Plan** (incluye construcciones, hitos, asignación de usuarios y resultados de las pruebas) y **User** (cada usuario define las características que pueden ser usadas en la herramienta) [14].

Por otro lado, el administrador de casos de prueba Testopia, es una herramienta diseñada para hacer seguimiento a casos de prueba en dos modalidades disponibles, presencial o virtual, en donde su función se encuentra direccionada a realizar pruebas de software, encontrando defectos y los resultados de los casos de prueba [13].

Mas recientemente, como se menciona en [15] y [16] en el mercado global se pueden encontrar diferentes frameworks de trabajo para realizar testing como Serenty, Cypress, Robot Framework, TestProyect.io, Galen Framework y WebDriverIO, en los cuales se presta un entorno en el que el tester diseña pruebas automatizadas para la ejecución de los casos de pruebas mas no para el diseño de los casos en sí, es decir, se automatizan los flujos principales de los productos software, sin embargo, la generación de casos de prueba aún se limita

[9] https://csrc.nist.gov/projects/automated-combinatorial-testing-for-software

[10] http://testlink.org/

[11] https://developer.mozilla.org/es/docs/Mozilla/Bugzilla/Testopia

a la experiencia o intuición del tester, a pesar de que hay metodologías para este proceso, no garantizan la máxima cobertura posible de los casos de prueba.

## 2.3 Arreglos de Cobertura Mixtos

Para entender los MCA, primero se deben entender los arreglos de cobertura (CA) uniformes. Un CA uniforme es una estructura matemática utilizada en general para el diseño experimental porque logra la máxima cobertura de las pruebas con el menor costo, por lo que se usan en múltiples escenarios, entre ellos las pruebas de software y hardware. Un CA se define con cuatro parámetros (N, t, k, v) y corresponde a una matriz bidimensional de N filas por k columnas, donde N representa el número de casos de prueba en el contexto de las pruebas de caja negra, k representa el número de componentes (parámetros, columnas o atributos) que contiene cada prueba (en la prueba de un método corresponde al número de sus argumentos o parámetros) y t representa la fuerza con que se realiza las pruebas. En la prueba de un método, t indica el nivel de interacción que se espera evaluar entre los parámetros y v representa los símbolos, valores o alfabeto que cada componente (parámetro) puede tener, en el caso de un CA uniforme el valor de v es igual para todas las columnas. Un MCA es un CA con diferentes valores de alfabeto en por lo menos una columna, esto hace que la definición se establezca como MCA (N, t, k, $\{v_1, v_2, .., v_k\}$), donde cada $v_i$ define los valores (alfabeto) que componen cada columna [17], [18].

## 2.4 Post Optimización

Un CA o MCA se puede obtener con base en uno ya existente que tiene una configuración similar, pero con más columnas o que tiene un alfabeto mayor al solicitado en uno o más columnas. El proceso de tomar ese MCA mayor y reducirlo al MCA requerido se conoce como post optimización. En el proceso de pruebas de software esta técnica es muy útil para poder dar respuestas rápidas a los testers, ya que la construcción desde cero de un MCA requerido con un algoritmo metaheurístico o codicioso puede demorar más tiempo del que el tester estaría dispuesto a esperar. El MCA obtenido por post optimización cumple con los requerimientos de la prueba de caja negra que se va a realizar y generalmente tiene más filas que el MCA óptimo (teórico o conocido), pero el resultado se entrega en un tiempo apropiado para el tester.

El proceso de post optimización consiste básicamente en reducir al mínimo posible la cantidad de filas de un CA o MCA existente a un MCA con los k atributos, fuerza t y alfabeto requerido para cada columna [5]. Este problema de post optimización también es un problema NP-completo [19], pero aplicando operaciones determinísticas o estocásticas que siempre mantienen las características del MCA requerido, la matriz que se está operando puede reducir el número de filas del MCA original a un número aceptable para el desarrollo de las pruebas. La sección

2.6 presenta la propuesta más importante y completa de post optimización que se encontró en la revisión de la literatura.

## 2.5 Estado del arte en construcción de CA y MCA

Construir CA y MCA es una tarea compleja. Revisando en Google académico, IEEE, Scopus, ScienceDirect y SpringerLink se encontraron diferentes tipos de algoritmos, a saber: exactos, algebraicos, recursivos, codiciosos (greedy) y metaheurísticos [20]. Según [17], los algoritmos o técnicas metaheurísticas han producido los mejores resultados hasta el momento y hay una gran variedad de metaheurísticas que se han usado en la construcción de CA y MCA, tales como la búsqueda tabú [21], el recocido simulado (Simulated Annealing, SA) [22], los algoritmos genéticos [23], el algoritmo por colonia de hormigas [24], la optimización por enjambre de partículas [8], la búsqueda armónica [25], el enjambre de pájaros [26], algoritmos híbridos que combinan el algoritmo de colonia de abejas y la búsqueda armónica [27], la búsqueda cucú [27], la evolución diferencial [28], la búsqueda tabú como una propuesta hiper heurística [29], y una combinación de recocido simulado con un algoritmo codicioso [30], entre otros.

En estos trabajos previos se observa una tendencia a usar metaheurísticas que integran o se basan en recocido simulado con algoritmos codiciosos y vecindarios variables, ya que son los que reportan los mejores resultados. Además, se observa que los arreglos de cobertura óptimos no son compartidos por los investigadores que los encuentran, sino que se deben comprar o establecer convenios con los autores de los artículos para poder adquirirlos. Por otro lado, la mayoría de las propuestas que construyen CA o MCA de gran tamaño se ejecutan en supercomputadoras por largos periodos de tiempo para obtener el número óptimo (menor número teórico o conocido) de filas para cada arreglo construido.

A continuación, se resumen las principales y más recientes propuestas. En el año 2018 [31] se desarrolló un enfoque metaheurístico y codicioso de 3 etapas para la construcción de CA que cuenta con otros enfoques muy importantes como el complemento y la post optimización de CA. En la primera etapa para la construcción de CA, utilizan un algoritmo metaheurístico y crean Covering Perfect Hash Families (CPHF), facilitando la construcción de grandes matrices con poco esfuerzo de cómputo. La segunda etapa revisa si los CPHF son CA, sino es así se completan (se agregan filas) para asegurar que sean CA y finalmente en la tercera etapa se post optimizan (ver más detalles en el numeral 2.6 del presente documento), es decir se reducen filas buscando combinaciones de columnas y valores que son redundantes en el CA. En general, la propuesta obtuvo un resultado positivo al reducir y mejorar un total de 21,217 CA y el tiempo de ejecución del algoritmo en general es mucho menor al reportado en el estado del arte.

Este mismo año (2018) en [32] se define la construcción de CA utilizando un algoritmo de recocido simulado (SA) para crear CPHF, mencionando que uno de los algoritmos más exitosos para esta labor es el propuesto por Cohen [33]. En este trabajo, el arreglo de cobertura objetivo se divide en matrices más pequeñas o "componentes", después de eso, cada componente se construye mediante una técnica combinatoria o con SA, si previamente no se tiene dicha construcción. Según este artículo, los algoritmos metaheurísticos son una de las mejores soluciones para construir CA de tamaño medio y pequeño más exitosas, haciendo un énfasis en el recocido simulado como uno de los mejores y más utilizados algoritmos para esta tarea, debido a la relación entre el tiempo de ejecución y la calidad de la solución que se obtiene.

Para el 2019 se desarrolló una nueva estrategia para la construcción de CA basada en un algoritmo codicioso y teoría de grafos. Dicha investigación [34] se basa en una representación llamada Cobertura de Nodos (CN) y el algoritmo llamado Graph Based Greedy Algorithm (GBGA). Este trabajo utiliza una representación a través de cobertura de nodos, para dar solución a la construcción de CA en el dominio de los grafos (transformación de un problema NP completo en otro problema NP completo, se resuelve el segundo problema y se traslada la solución al primer problema), esto ha hecho que el algoritmo presentado sea competitivo en cuanto a la generación de CA con el menor números de filas. Es preciso señalar que este método mejoró los CA y MCA reportados en el estado del arte en 80 de 98 casos comparados.

Ya en el año 2020 [35] se mejoraron diversos CA a través de CPHF utilizando un algoritmo de recocido simulado. En este trabajo lograron mejorar los límites superiores de 19,669 CA, es decir redujeron la cantidad de parámetros que contenían estos CA. Se pudo lograr estas mejoras restringiendo las entradas en los grupos de filas de dichos arreglos, una mejora en la búsqueda local que consiste en reemplazar la peor columna de CPHF con una de mejor solución encontrada por el algoritmo. Este método evidencia sus mejoras en la creación de CA en comparación con [31].

Colbourn[12], uno de los autores más importantes en la temática, pone a disposición del público un repositorio con una amplia lista de configuraciones de CA y los valores óptimos (teóricos o conocidos) de sus filas, pero no deja disponibles los CA en sí mismos. En este repositorio se puede observar que para CA con k<= 6 los algoritmos metaheurísticos que mejores CA producen se basan en recocido simulado y tienen enfoques codiciosos.

Uno de los trabajos más actualizados encontrados en la literatura es [36] de 2022, donde se menciona nuevas alternativas en donde enfocar los esfuerzos para encontrar oportunidades de mejora o posibles aplicaciones de esta tecnología. En

[12] http://www.public.asu.edu/~ccolbou/src/tabby/catable.html

dicho trabajo se menciona los arreglos de cobertura, sin embargo, se tiene una particularidad y es que se resalta la importancia de la generación de Arreglos de Cobertura Restringidos (CCAG), los cuales permiten construir casos de prueba más específicos, implementando una etapa de restricción, la cual consiste en limitar las tuplas o combinaciones de pruebas que no son requeridas, restándoles importancia con un mecanismo de penalización adaptativa. Posteriormente se propone un novedoso algoritmo de generación de matriz de cobertura con restricciones paralelas (APPTS) que integra el mecanismo mencionado de penalización adaptativa y otro llamado paralelización en el algoritmo Tabu Search (TS).

## 2.6 Estado del arte en post optimización de CA o MCA

En 2017 [5], se hizo una investigación utilizando un algoritmo metaheurístico para la post optimización de los CA almacenados en el repositorio del NIST (National Institute of Standards and Technology de Estados Unidos). Para esto se usó un enfoque de Meta-Optimización Post heurística (MPO) compuesto de tres operaciones principales: a) un detector de redundancia (RD); un reductor de fila (RR); y un reductor de combinación faltante (MCR) en el marco de un algoritmo de recocido simulado (SA) [37] ya que SA había sido utilizado con éxito en problemas relacionados [37]–[40].

El Detector de redundancia busca las entradas redundantes de un CA. RD hace su trabajo haciendo tres escaneos, el primer escaneo establece las celdas de “símbolo fijo” (FS) que participan en las combinaciones, esto es, celdas que aportan una única combinación de valores por una única combinación de columnas. Las que participan en más de una, se agrupan en otro conjunto denominado PRC. El segundo escaneo trabaja con las celdas marcadas como PRC y decide cuales celdas se transforman al estado de FS, mientras se asegura de que se cumpla la propiedad de cobertura (todas las combinaciones deben estar cubiertas al menos una vez), este paso busca dejar el máximo número de celdas en PRC. El tercer escaneo elimina todas las filas en las que k-t+1 elementos de la fila tienen estado PRC. “Row Reducer” recibe como entrada un CA y de manera codiciosa busca una fila i de tal manera que su eliminación minimice las combinaciones faltantes. En el peor de los casos, RR prueba todas las filas del CA, pero cuando se encuentra una fila sin combinaciones que faltan, RR termina. Si este no es el caso, la fila eliminada es la que da la menor cantidad de combinaciones faltantes. El reductor de combinación faltante (MCR) se encarga de reducir a cero el número de t-tuplas faltantes del parámetro de entrada (una matriz con combinaciones faltantes). Los resultados del estudio demuestran una clara optimización de los CA como se puede apreciar en [5]. Los CA y MCA del NIST no son óptimos, ya que tienen en todos los casos, muchas más filas de las reportadas por Colbourn en su repositorio disponible en https://www.public.asu.edu/~ccolbou/src/tabby/catable.html.

Mas recientemente la atención de los investigadores se ha enfocado en adaptar las alternativas existentes para que se pueda obtener mejores resultados que aporten valor a la industria de software en sí, es el caso de [41], donde la investigación se centró en encontrar una opción viable para pruebas de carga con gran cantidad de parámetros de entrada, en donde ahorrar recursos computacionales y tiempo son la prioridad. Para este motivo los autores propusieron una operación matemática denominada “unión combinatoria basada en productos debilitados” que según se estipula construye un nuevo arreglo de cobertura a partir de dos arreglos ya existentes.

Aunque no se trate específicamente de una etapa de construcción si se puede considerar como tal, ya que a raíz de dos soluciones ya generadas busca ahorrar tiempo y recursos para la generación de un nuevo arreglo de cobertura más optimo. De acuerdo con sus resultados el ahorro en tiempo de optimización es apreciable, alrededor de un 33 %, lo cual contribuye a una solución aceptable en caso de una posible segunda etapa de optimización, sin embargo, el alcance del proyecto no permitió considerar una tercera etapa después de ya incluir dos, generación y Post-Optimización de MCA.

# CAPÍTULO 3

## 3 APLICACIÓN WEB

En este capítulo se presenta la arquitectura de software que se diseñó e implementó y una descripción del funcionamiento acoplado de los microservicios. En capítulos posteriores se hace una descripción detallada de cada uno de los componentes principales.

Con el fin de garantizar la accesibilidad a los usuarios, se construyó un aplicativo Web, que permite de forma intuitiva y sencilla llenar un formulario por medio del cual se genera un código fuente, código que los usuarios pueden copiar y realizar sus pruebas en su proyecto de desarrollo.

El aplicativo mencionado se denominó palabra Test del inglés prueba, y Code en relación con los códigos hechos por desarrolladores, el cual está centrado en el apoyo al desarrollo de pruebas unitarias.

Para el diseño y construcción del aplicativo se utilizaron diferentes herramientas las cuales se mencionan a continuación:

- Como IDE se usó el entorno de desarrollo Visual Studio 2021.
- Para el Backend se programó en C# sobre.Net Core 5.0. Además de una herramienta para implementar un bus de comunicación entre los microservicios llamado RabbitMQ versión 3.10.0.
- Para las bases de datos relacionales se utilizó Microsoft SQL Server.
- Para el Frontend se utilizó el framework Angular 12.
- Como servicios en la Nube se utilizó Microsoft Azure utilizando las App servicies y SQL Database que ofrece la compañía,
- Como herramienta de pruebas se utilizó Serenity 2.3.4

El **Anexo A** muestra el código fuente de la aplicación, el **Anexo B** presenta la documentación del código generada con DoxyGen y el **Anexo C** permite cargar el repositorio inicial de MCA.

### 3.1 ARQUITECTURA

De acuerdo con la revisión de la literatura realizada (basada en https://microservices.io./) y teniendo en cuenta la dimensión del proyecto, desde la definición del anteproyecto y los objetivos del mismo, se consideró que la mejor

opción para desarrollar la solución que se requería era implementar una arquitectura basada en microservicios, la cual a pesar de tener más complejidad en el desarrollo proporciona importantes beneficios a largo plazo como mayor disponibilidad, desacople, escalabilidad y mantenimiento. Lo anterior porque al ser una arquitectura desacoplada entre sus componentes permite intervenir una pequeña parte de la solución sin interrumpir el funcionamiento de la totalidad del sistema.

Concretamente en este caso las funcionalidades de Generación y Post optimización de arreglos de cobertura mixtos tienen un alto consumo de recursos computacionales lo cual impulso a considerar aún más este tipo de arquitectura, ya que permite balancear las cargas de trabajo entre todos los componentes del sistema para así garantizar siempre una alta disponibilidad.

Inicialmente la arquitectura de la solución se dividió en dos secciones principales, la parte orientada al usuario o Frontend y la parte que soporta la lógica del negocio y los servicios de base conocida como Backend.

El Backend se construyó con 4 microservicios principales, el primero para la gestión de los servicios u orquestador (Gateway API), el segundo para la generación de MCA (Generator API), el tercero dedicado a la Post Optimización (Optimizer API) de MCA y el cuarto encargado de la búsqueda especializada de los MCA (Seeker API). En cuanto al Frontend se diseñaron una interfaz que les permite a los usuarios testers interactuar con las herramientas para diseño de casos de prueba. La **Figura 1** presenta esquemáticamente como se planteó la arquitectura para abordar la solución que se construyó en esta investigación.

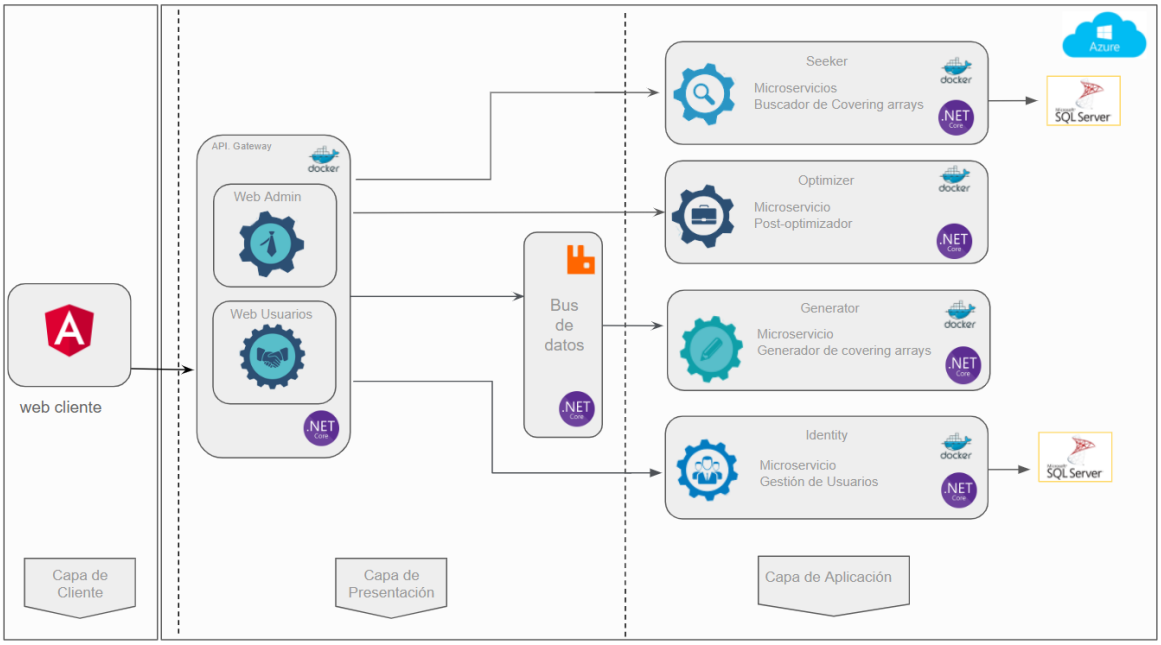

***Figura 1*** *Arquitectura de Microservicios de CodeTest*

En esta figura se puede apreciar la división entre la Capa de cliente, la Capa de presentación y la Capa de Aplicación, donde “Capa de Cliente” corresponde al Frontend y “Capa de presentación” junto a “Capa de Aplicación” corresponden al Backend.

El Backend se divide en dos partes conceptuales, en donde la lógica de negocio se organiza en la “Capa de aplicación” y consta de los microservicios Seeker, Generator y Optimizer los cuales se encargan principalmente de la gestión de MCA, por otro lado, se tiene la “Capa de Presentación”, en la cual se cumplen las funciones de interacción entre las capas de Cliente y Aplicación, mediante el funcionamiento del “API Gateway” que efectúa la labor de orquestador.

La “Capa de Aplicación” comprende los procesos necesarios para Buscar, Generar y Post optimizar MCA, en cada uno de estos microservicios se implementaron los algoritmos pertinentes a dichos procesos. En cuanto a la base de datos se implementó una para la gestión de MCA en la cual se alberga el repositorio de MCA asociada al microservicio Seeker.

En la “Capa de presentación” concretamente en el “API Gateway” se implementó una funcionalidad con el fin de secuenciar el orden de ejecución de las APIs, en el momento en que se realiza una petición al servicio Seeker, en paralelo y si es necesario se pone en funcionamiento el servicio Generador mediante el Bus de RabbitMQ, el cual se encarga de encolar las peticiones y dejar en ejecución el servicio de generación, al mismo tiempo que permite accionar el resto de las funcionalidades sin necesidad de interrumpir el flujo de ejecución. Esto se hace porque el proceso de Generar un MCA es una tarea que consumo más tiempo del que estaría dispuesto a esperar un tester, por eso se ejecuta en “segundo plano” y al tester se le entrega una respuesta basada en un MCA disponible en el repositorio o el resultado de la post optimización de un MCA ya existente pero que es más grande de lo requerido (sobrepasa los requisitos del tester), proceso que entrega una respuesta más rápida que la del generador, aunque en la mayoría de los casos, entregando un MCA con más número de filas que el que se podría obtener de la Generación.

Los usuarios sólo pueden usar las funcionalidades provistas a través de la interfaz Web, que con la ayuda de la capa de presentación interactúa con los diferentes servicios, y trae de la capa de aplicación los recursos necesarios para atender las necesidades del usuario.

## 3.2 Frontend

Teniendo en cuenta la necesidad de los usuarios (Testers) se diseñó una interfaz de usuario dinámica y minimalista, la cual en una sola pantalla (Single Page Application, SPA) le brinda al tester las herramientas necesarias para diseñar sus

casos de prueba enmarcados en proyectos de prueba. La aplicación se diseñó para mantener una comunicación clara y concisa con el usuario (es una aplicación que interactúa con algoritmos de una complejidad importante), brindándole la información justa y necesaria para la tarea que está desarrollando y evitando así la sobre carga de información. De esta manera se buscó la recepción (entrada de datos) de los valores requeridos con enunciados claros y sencillos; en esta sección se indica cómo funciona el Frontend orientado al usuario.

Inicialmente en la pantalla de creación de proyectos se reciben los datos concernientes a nombre de la clase, nombre del método, tipo de retorno, iteraciones, y los parámetros de la prueba, parámetros que están compuestos por nombre de la variable, el tipo de retorno y los valores para dicha variable (ver **Figura 2**). Es preciso comentar que en el **Anexo G** se puede ver un video con la funcionalidad de la aplicación web.

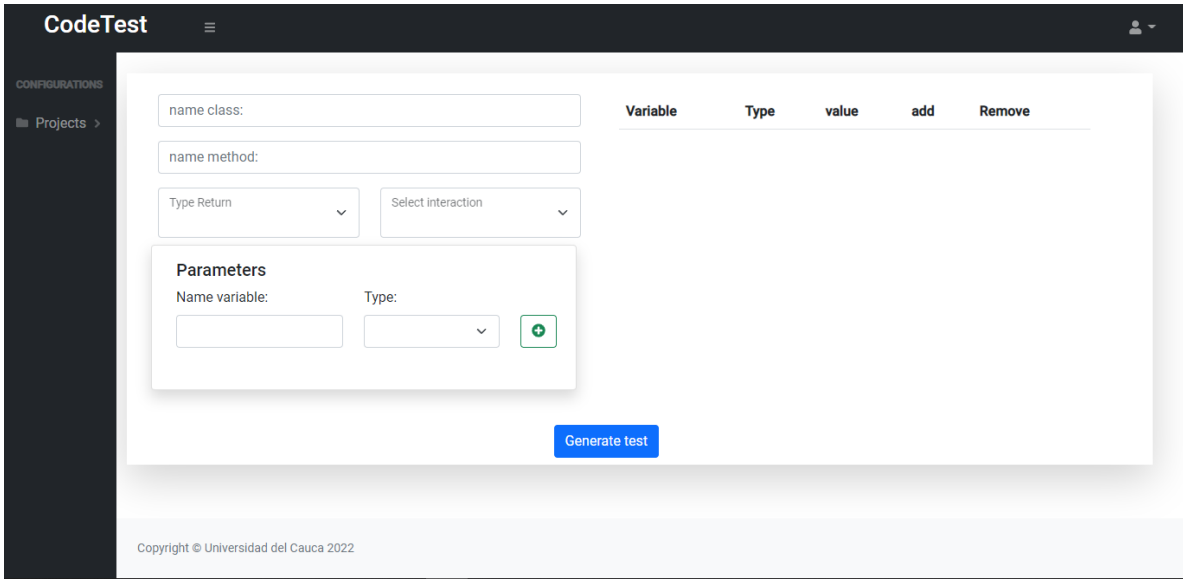


***Figura 2** pantalla Inicial Frontend*

A continuación, se muestran los pasos para que un tester defina como espera probar un método. En la **Figura 3** en el cuadro resaltado en rojo se recibe el nombre de la clase y método que se va a probar, estos datos se utilizan para crear el código de la prueba unitaria en NUnit o en Junit.

Luego de definir el método que se va a probar, se procede a llenar el tipo de retorno del método (int para entero) y la fuerza del MCA que se usará (Select interaction), esto es, el mayor grado de fuerza o interacción entre los parámetros que se espera evaluar (ver **Figura 4**).

En la interfaz presentada en la **Figura 5**, el usuario ingresa uno a uno los parámetros que el método recibe, por ejemplo, en este caso, se nombró la variable ***Procesador*** con el tipo de dato ***int*** (entero). Tan pronto se adiciona el parámetro

este se muestra al lado derecho de la interfaz. Esa lista de parámetros (variables) se incrementa cada vez que se adiciona un nuevo parámetro y permite además modificar un parámetro o borrarlo en caso de no requerirse (ver los iconos del lado derecho de la figura).

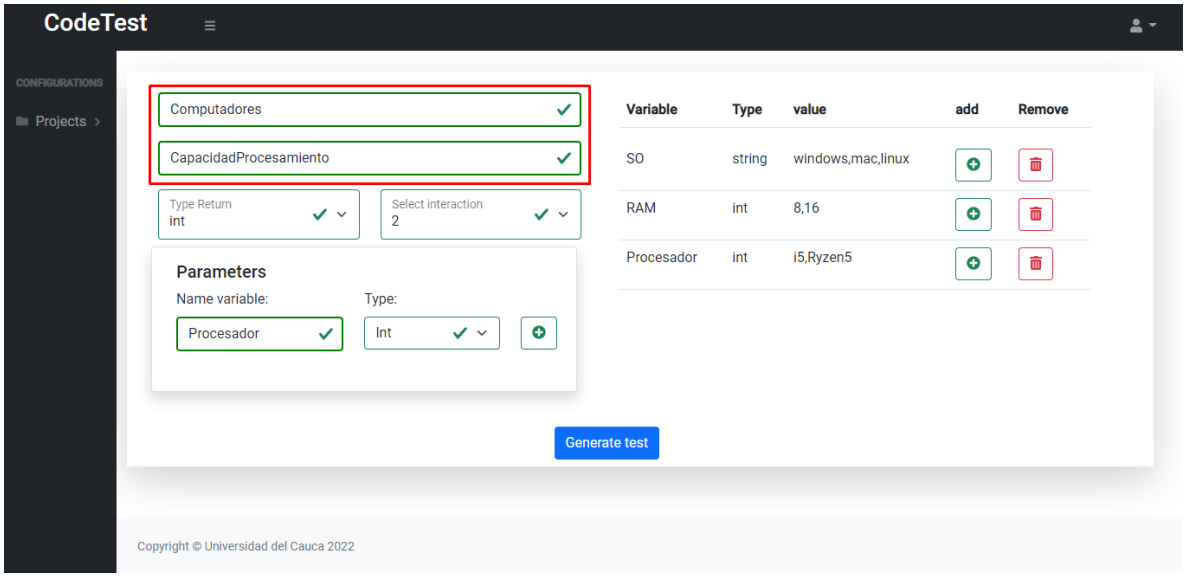


***Figura 3*** *Frontend: definir nombre de la clase y método a probar*

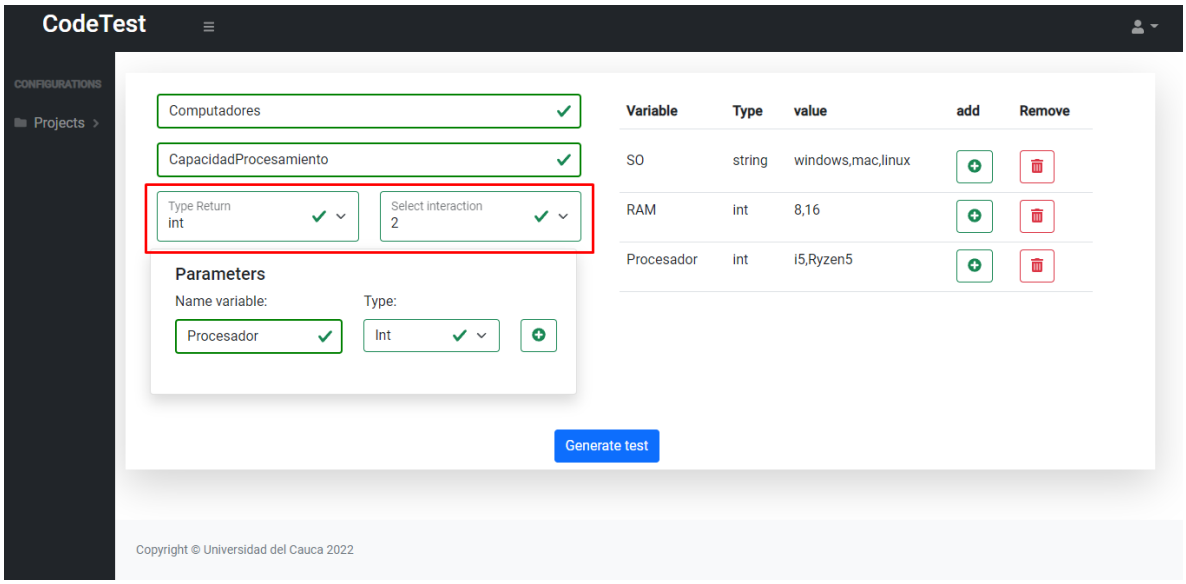


***Figura 4*** *Frontend: definir el tipo de retorno del método y la fuerza de la prueba*

Cada parámetro debe tener los valores esperados que se quieren evaluar, por ejemplo, para datos numéricos, un valor que este por encima del mayor valor permitido, un valor que este por debajo del menor valor permitido, un valor que este cerca del mayor valor permitido, un valor que este cerca del menor valor permitido y un valor normal o comúnmente usado. En este caso, estos valores se

incluyen presionando el botón con el símbolo de "+" en el lado derecho de la **Figura 5**. En este caso se tienen las variables (Windows, Mac, Linux) para el parámetro/variable denominada SO (sistema operativo), y de la misma forma para los otros dos parámetros. Después de determinar el tipo de prueba que se va a generar el usuario debe presionar el botón "Generate test" que se muestra en la parte inferior de la pantalla para continuar con la segunda parte de la definición de la prueba que corresponde a la tarea de establecer la respuesta a cada caso de prueba generado, tarea que debe realizar el tester.

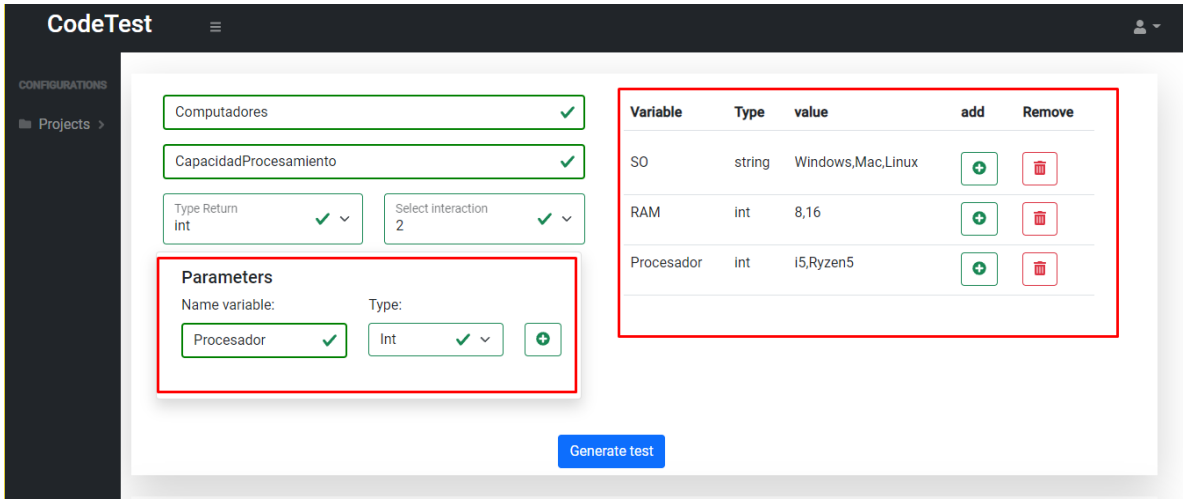


***Figura 5*** *Frontend: definir parámetros a probar (nombre de la variable y tipo)*

En la siguiente sección (ver **Figura 6**), el usuario establece el retorno o respuesta esperada (Expected value) que debería tener el método con las diferentes combinaciones de parámetros generada.


***Figura 6*** *Frontend: asignar resultado del método*

Por último, para que la prueba concluya el Usuario debe presionar el botón generar código el cual se mostrara en el recuadro negro ver (**Figura 7**), en donde el tester puede copiar el código y pegarlo en su proyecto para así ejecutar la prueba.

## 3.3 BACKEND

En esta sección se describe el funcionamiento y construcción de cada uno de los microservicios que compone la Capa de Aplicación y de Presentación, adicionalmente los criterios utilizados para su organización y diseño, así como su acople con los demás componentes. Esta sección contiene la descripción de los microservicios mencionados; Generador, Post Optimizador y Seeker, además del Bus de datos.

Los microservicios se organizaron (dividieron) en cuatro de acuerdo con la descomposición por subdominio, lo anterior gracias a que se logró identificar el desacoplamiento que logra tener cada uno de los objetivos informativos que se quieren gestionar en la aplicación, sin embargo, a pesar de que los microservicios para la gestión de MCA están muy relacionados, su carga de trabajo y consumo de recursos computacionales hizo que se considerara este tipo de división. A continuación, se realiza la descripción del funcionamiento de los microservicios de la Capa de Aplicación y como estos se integran con las demás partes de la solución.


***Figura 7*** *Frontend: código generado para las pruebas unitarias*

### 3.3.1 Microservicio Seeker

El sistema implementado fue diseñado para poder garantizar un tiempo de respuesta optimo para el usuario, y no solo eso, también se buscó proporcionar la mejor solución posible, en primer caso, porque este disponible en el repositorio, es por ello por lo que se identificó como una necesidad de alta prioridad crear un servicio capaz de proporcionar esta solución. En la API de Seeker se despliegan diferentes funcionalidades relacionadas con la búsqueda y organización de MCA dispuestas de la siguiente manera:

A. Inicialmente se reciben los datos del arreglo de cobertura que se requiere mediante un método POST, estos datos se digitan en el Frontend y se pasan a la API Gateway:
   - Columns (k): corresponde a la cantidad de variables o columnas dispuestas por el usuario.
   - Strength (t): corresponde a la fuerza o nivel de interacción entre las variables o parámetros.
   - Alphabet (v): corresponde a la distribución del alfabeto, es decir, a la cantidad de posibles valores asociados a cada una de las variables o parámetros.

B. Posteriormente este servicio realiza una ampliación de los parámetros de búsqueda (si se busca un MCA con el alfabeto 3,3,2,2 se genera una condición de búsqueda para cada columna que de la opción de alfabetos mayores, por ejemplo hasta 9 de la siguiente forma [9876543] [9876543] [98765432] [98765432] *, el asterisco al final permite tener en cuenta MCA con más columnas) y mediante una función SQL (ver **Figura 8**), busca el MCA que más se acerca a los requerimientos del usuario y que está disponible en el repositorio, de acuerdo con 4 criterios principales, a saber:
   1. El arreglo de cobertura debe tener un número igual o superior al número de columnas solicitado, ya que si tiene un número menor éste no podrá ser post optimizado.
   2. La fuerza debe ser estrictamente igual a la que se requiere, debido a que un número de interacciones menor no proporciona la cobertura adecuada, y un número mayor llega a soluciones con un número de filas relativamente grande cuando se realiza la post optimización.
   3. Con respecto al alfabeto se buscan todos los resultados iguales o superiores de acuerdo con el orden descendente establecido para este.
   4. Por último, los resultados se ordenan de forma ascendente con respecto a la fuerza, número de columnas y número de filas, esto con el fin de garantizar que la búsqueda retorne la mejor opción.

```
USE [BDMCA]
GO
/****** Object:  StoredProcedure [dbo].[listarCAs]
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
ALTER PROCEDURE [dbo].[listarCAs]
(
    @Condicion varchar(300),
    @Columns int,
    @Strength int
)
AS
    SELECT TOP (1) *
    FROM CaRepository
    WHERE Alphabet like @Condicion
    and Columns >= @Columns
    and Strength = @Strength
    ORDER BY Strength ASC, Columns ASC, Rows ASC
```

***Figura 8*** *Backend: Consulta SQL*

C. Por último, el servicio retorna la respuesta conformada por un JSON como el presentado en la **Figura 9**. En él se muestra la respuesta correspondiente a las características del arreglo de la siguiente manera: En “CA_notes” se entrega el MCA en formato string en “Columns”, “Strength” y “Alphabet” se entregan las características del arreglo de cobertura encontrado, estas pueden diferir a las condiciones de búsqueda de acuerdo a lo mencionado anteriormente, es decir, hay dos posibilidades, que correspondan exactamente a las del arreglo de cobertura requerido, el cual se entrega directamente al API Gatewey para posteriormente dar respuesta al usuario, o que dichas características sean superiores a las requeridas, en este caso el MCA se pasa al API Post_Optimizer para que el MCA sea optimizado y adecuado a las características requeridas por el usuario tester.

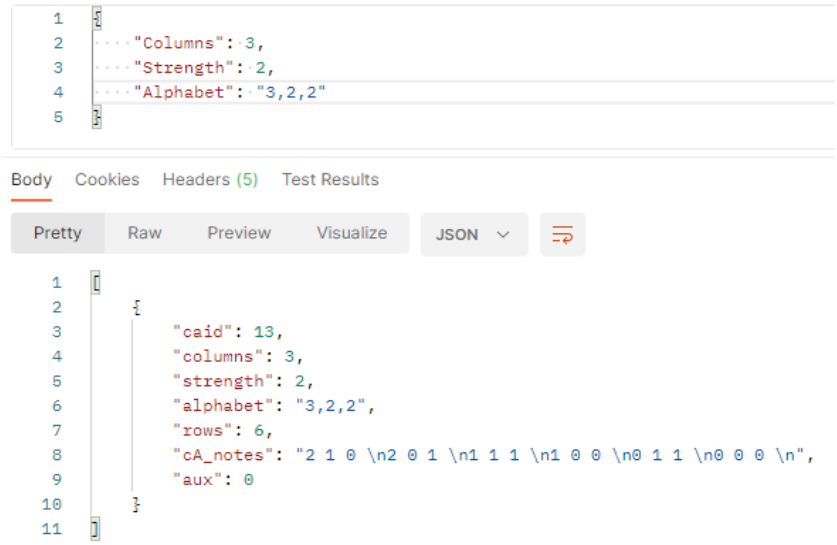


***Figura 9*** *Backend: respuesta del microservicio API Seeker*

Por ejemplo, en el caso de que el MCA mostrado en la **Figura 9** no existiera en el repositorio y se buscará un arreglo de cobertura con las características MCA (k=3, t=2, V (3,3,2)), la API Gateway podría responder con un MCA que lo subsuma (ver **Figura 10)**, es decir, un MCA que es un poco más grande que el requerido y que puede ser post optimizado.

Adicionalmente el microservicio (API Seeker) está relacionado directamente con el repositorio, por ello los demás servicios deben acceder a él para consultar o almacenar un MCA.

### 3.3.2 RabbitMQ

RabbitMQ es un broker de mensajería de código abierto, distribuido y escalable, que sirve como canal para la comunicación eficiente entre servicios. RabbitMQ implementa un protocolo de mensajería a nivel de la capa de aplicación AMQP (Advanced Message Queueing Protocol), el cual está enfocado en la comunicación de mensajes asíncronos con garantía de entrega.

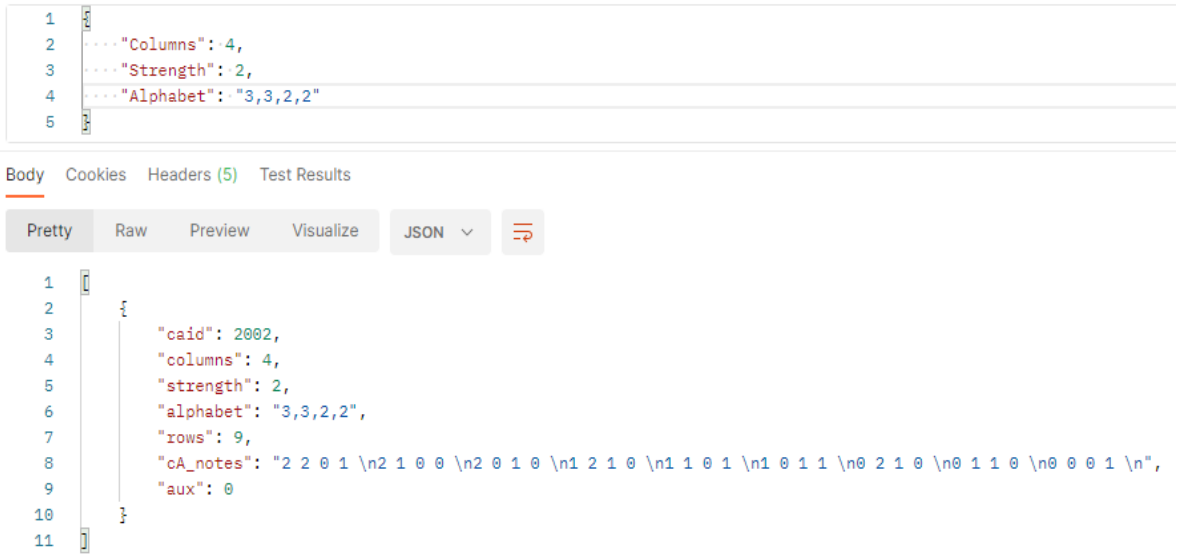


***Figura 10*** *Backend: respuesta del microservicio API Seeker*

La implementación del bus con RabbitMQ permitió utilizar su característica asíncrona para garantizar el encolamiento de las diferentes peticiones realizadas en este caso a la API Generator de la siguiente manera: después de que el servicio buscador identifica un MCA que no es exactamente el requerido, inmediatamente al hacer la petición al microservicio PostOptimizer, encola en el bus de RabbitMQ una petición, correspondiente con los valores del MCA solicitado por el usuario, la cual en su momento solicita una petición al microservicio Generator, donde se construirá el MCA y se almacenará en el repositorio y posteriormente este nuevo MCA podrá ser utilizado por un Tester en una nueva prueba.

### 3.3.3 Microservicio Generador

Para la generación de MCA se dispuso de un microservicio orientado estrictamente a esta labor, ya que puede requerir de gran cantidad de recursos computacionales (procesamiento y memoria RAM). A continuación, se describe como este servicio interactúa con el resto de la arquitectura sin hacer énfasis en su lógica interna, ya que esta se explica más adelante en el **Capítulo 4**.

De acuerdo con lo mencionado en la descripción de API Seeker, se estipuló que al momento de retornar el valor al API Gateway esta gestiona la respuesta y de acuerdo con la misma interactúa con las APIs Generador y Post-Optimizador de la siguiente manera:

1. Se hace una petición desde API Gateway a Seeker y este, de acuerdo con lo ya estipulado, responde con un MCA, en caso de que el arreglo no fuese encontrado en el repositorio, el API Gateway encola una petición en el bus de RabbitMQ antes de consumir el servicio Post-Optimizador.

2. En el bus de RabbitMQ se encola la petición de acuerdo con el orden de llegada y posteriormente consume una petición Post a la API Generador, la cual desencadena su lógica interna para construir un MCA con las condiciones establecidas.
3. Por último, el Generador guarda la solución generada para nutrir el repositorio y que más adelante en el tiempo, este nuevo MCA pueda ser usado por un tester.

El flujo de creación de MCA se estableció de esta manera ya que el generador realiza bastantes iteraciones o ciclos que consumen muchos recursos computacionales (procesamiento y memoria RAM) y tiempo que no satisfacen los lineamientos óptimos para dar respuesta al usuario.

### 3.3.4 Microservicio Post-Optimizador

Para el proceso de Post-Optimizacion también se vio necesario crear un microservicio capaz de responder rápidamente a las demandas del usuario, de acuerdo con los flujos establecido anteriormente, este servicio recibe una petición directamente desde el API Gateway, si el buscador (Seeker) no encuentra la respuesta exacta a la petición. En este sentido, recibe del Seeker dos cosas, el MCA disponible en el repositorio que subsume los requerimientos del tester y las características del MCA que se requiere (MCA objetivo), es decir, el solicitado por el usuario. A continuación, en la **Figura 11** se puede observar el tipo de petición que se le hace al endPont del servicio y la respuesta que se obtiene.


***Figura 11*** *Modulo Backend, respuesta servicio Post-Optimizador*

Se puede apreciar que en los campos Alphabet y TarjetAlphabet se tiene el alfabeto del MCA encontrado y el alfabeto del MCA objetivo respectivamente. Con estos datos el servicio puede adecuar dicho MCA encontrado en el repositorio hasta encontrar el MCA objetivo, el cual luego se retorna en el campo cA_notes del JSON de la respuesta.

# CAPÍTULO 4

## 4 GENERADOR DE MCA

En este capítulo se describe el diseño y funcionamiento de uno de los componentes principales de la aplicación, el algoritmo para la construcción de arreglos de cobertura mixtos.

Como ya se mencionó previamente en el capítulo relacionado con el estado del arte, existen diferentes formas de construir arreglos de cobertura, los cuales son adecuados para ciertas características de los MCA que se esperan generar. Por ejemplo, en [31] buscan mejorar los limites superiores de MCA con altos valores de fuerza, tema que no es de muy requerido en el desarrollo de software según los resultados arrojados por el NIST (**Figura 12**) en donde se pudo determinar que la mayoría de los errores o fallas en el software son causados por la interacción de uno, dos y tres componentes. Además, se tomaron en cuenta números de columnas hasta de 8 o 10, ya que, en la actualidad, un número alto de parámetros para un método se puede ver como un error de diseño [43].

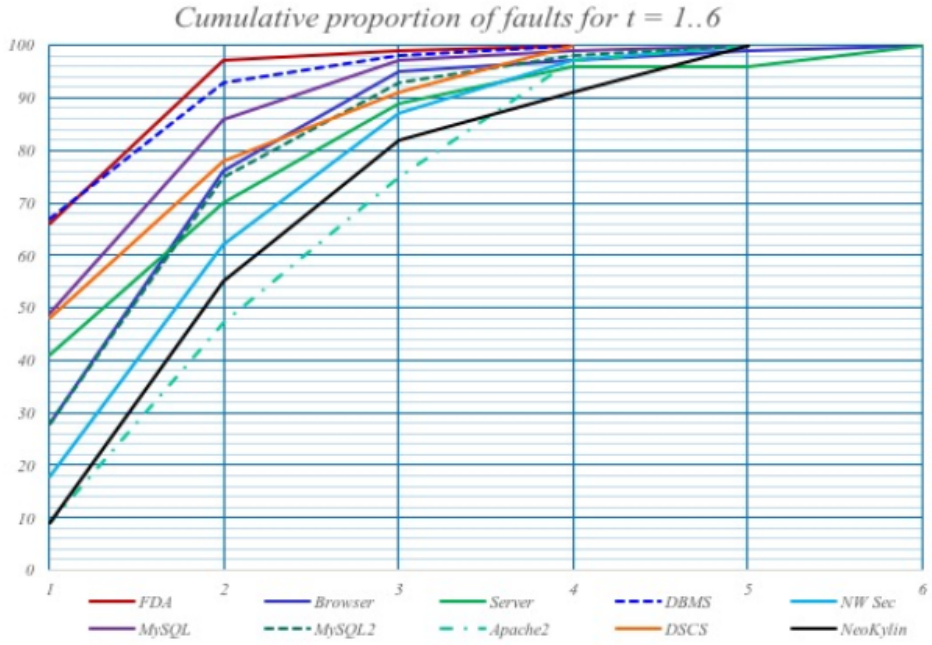


***Figura 12*** *De acuerdo con el estudio realizado por el NIST*

Por otro lado y basado en el estado del arte, se decidió tomar el **Recocido simulado** como la mejor opción para desarrollar el generador, ya que se encuentra con mayor frecuencia como herramienta generadora de los limites inferiores de arreglos de cobertura de dimensiones no superiores a la configuración MCA(t=6,k=6) según las tablas de filas óptimas presentadas por Charlie Colbourn (https://www.public.asu.edu/~ccolbou/src/tabby/catable.html).

A continuación, se describe en detalle el diseño del algoritmo que soporta el proceso de generación de MCA.

### 4.1 Perspectiva General del Algoritmo

Orden de ejecución de las funcionalidades del generador:

- Inicialmente se reciben los parámetros del MCA (t, k, v, n) que se espera generar, con ellos se crea una Matriz de dimensiones N*K donde se construirá posteriormente el MCA de manera aleatoria (en realidad no es un MCA, sino más bien una matriz que aspira a ser el MCA objetivo) y adicionalmente se genera la Matriz P herramienta con la que se calcula las combinaciones de columnas con sus combinaciones de alfabetos cubiertos hasta el momento por la solución.
  - Para crear las filas de esa matriz (MCA) se construye la primera fila de forma aleatoria y luego mediante un método Greedy y un algoritmo codicioso se construyen filas candidatas a posicionar en el MCA, se selecciona la mejor utilizando la distancia de haming con respecto a las filas ya creadas, de esta manera hasta que se complete el número de filas establecido.
  - Ya creado el MCA, se calcula el Fitness de la solución actual (MCA actual) mediante la Matriz P, que indica la cantidad de combinaciones de valores faltantes en las combinaciones de parámetros. Si el número de combinaciones con sus valores es igual a 0, se termina el proceso ya que se encontró el MCA requerido y se entrega la solución, si no, continua con el siguiente paso.

- Ya construido el MCA se realiza un proceso iterativo durante un número determinado de iteraciones o hasta que se encuentre el MCA objetivo. Dicho proceso iterativo busca optimizar o mejorar este MCA remplazando filas o celdas para llenar en la Matriz P combinaciones de valores faltantes en las posibles combinaciones de parámetros. Este proceso se realiza mediante la ejecución de uno de tres (3) posibles algoritmos:

  - **Algoritmo 1**: Selecciona aleatoriamente una fila del MCA actual y remplaza todos los posibles valores del alfabeto, columna por columna, hasta que se crea una nueva fila que pueda ser mejor, es decir, que disminuya el fitness.

- **Algoritmo 2**: De acuerdo con la Matriz P se selecciona una combinación faltante y se crea una fila, esta se prueba en todas las filas del MCA actual, y se establece en donde encuentra un menor Fitness.
- **Algoritmo 3**: De acuerdo con la Matriz P y las combinaciones faltantes, se construye una nueva fila, y se remplaza por la peor fila del MCA actual, la cual se selecciona basándose en las combinaciones sobrantes que aporta cada fila, es decir, la fila que más combinaciones repetidas tiene se reemplaza por la nueva.

Estos algoritmos se ejecutan de acuerdo con un número pseudoaleatorio que de acuerdo con una proporción establecida selecciona en cada iteración uno de ellos (**Tabla 1**), esta proporción se definió empíricamente buscando encontrar un balance adecuado de ejecución de los tres algoritmos. En este caso la proporción más alta se asignó al “algoritmo 1” el cual es más determinístico, sin embargo, el aporte del “algoritmo 2” y del “algoritmo 3” es crucial para no caer en óptimos locales y de esta forma se encuentren mejores soluciones de acuerdo con los conceptos del recocido simulado.

***Tabla 1*** *Porcentajes de ejecución de los algoritmos de optimización local*

| Algoritmo | % de ejecución por iteración |
|---|---|
| Algoritmo 1 | 20% |
| Algoritmo 2 | 30% |
| Algoritmo 3 | 50% |

- Por último, el algoritmo al encontrar un Fitness de 0 (el mínimo posible), disminuye la cantidad de filas del parámetro inicial en una unidad y repite el proceso con este nuevo número de filas. Esto se hace hasta que ya no sea posible encontrar un MCA con el número reducido de filas.

### 4.1.1 Inicialización Matriz MCA y Matriz P

Para poder manipular la información subministrada al algoritmo, se hizo necesario inicializar dos matrices con las cuales se trabajará la construcción y evaluación de los MCA, una matriz MCA y una matriz MP (Matriz P). A continuación, se hace una descripción de cada una de ellas, para esto se tomara un ejemplo de MCA (N=5, k=2, t=2, v=(3,2,2)) (ver **Figura 13**).

Como se puede apreciar en la **Figura 13**, se crea un espacio matricial en el sistema para que este pueda contener los valores correspondientes al MCA objetivo, de acuerdo con el número de columnas (K) correspondiente al número de variables designado por el usuario y N que corresponde al número de filas.

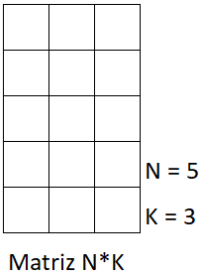


***Figura 13*** *Dimensiones matriz arreglo de cobertura (MCA)*

El método de inicialización de la Matriz P tiene algunos parámetros y submétricas adicionales que dependen del alfabeto (v de cada columna) y la fuerza (t) del MCA objetivo (ver **Figura 14**), estos son:

- Matriz J: corresponde a las interacciones entre las columnas, y se construye utilizando la fuerza y la cantidad de columnas, en este caso t = 2 y k = 3, es decir dos interacciones entre las columnas 0,1,2, a saber: (0,1), (0,2) y (1,2). Si se tuvieran 4 columnas y fuerza 3 sería: (0, 1, 2), (0, 1, 4), (0, 2, 3) y (1, 2, 3).
- Max J: Máximo número de interacciones entre columnas. Este valor se calcula mediante la función combinatoria de k en t, es decir, $(k\ t\ ) = \frac{k!}{t!(k-t)!}$.
- Vt: indica la cantidad de combinaciones posibles por fila teniendo en cuenta el alfabeto.
- Matriz P: donde se establece si la combinación de valores existe en determinada interacción entre columnas, si la combinación existe se inserta el número de veces que existe. La matriz se inicializa en 0.

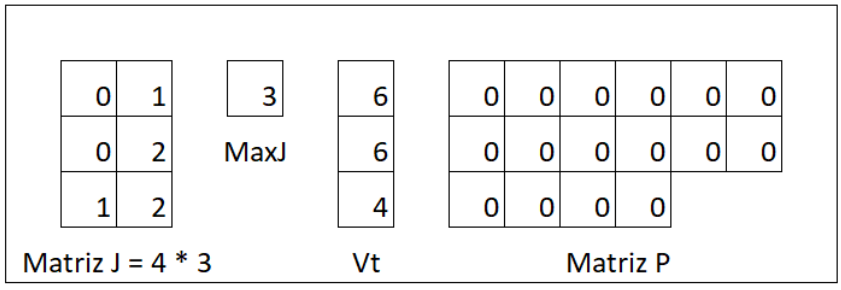


***Figura 14*** *Valores y matrices para crear la Matriz P*

En la **Figura 15** se puede apreciar un esquema más concreto de la matriz P en donde se muestra más claramente su construcción. Más adelante se explica cómo se calculan las interacciones y como se llena dicha matriz para determinar si se cumplen la totalidad de las interacciones.

| | | 0 | | 1 | | 2 | | 3 | | 4 | | 5 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 1 | 0 | 0 0 | 3 | 0 1 | 1 | 1 0 | 0 | 1 1 | 1 | 2 0 | 0 | 2 1 | 1 |
| | | | | | | | | | | | | | |
| 0 2 | 1 | 0 0 | 2 | 0 1 | 2 | 1 0 | 0 | 1 1 | 1 | 2 0 | 1 | 2 1 | 0 |
| | | | | | | | | | | | | | |
| 1 2 | 2 | 0 0 | 2 | 0 1 | 1 | 1 0 | 1 | 1 1 | 2 | | | | |

Coordenadas Matriz P
Combinaciones de columnas (Matriz J)
Cantidad de repeticiones
Combinaicones de alfabeto

***Figura 15*** *Matriz P*

### 4.1.2 Construcción Matriz MCA

Para el proceso de construcción del MCA en primera instancia se inserta la primera fila de forma completamente aleatoria y para las demás se dispuso un algoritmo basado en un método Greedy conforme se presenta en la **Figura 16**, en donde se busca asignar o llenar con filas diferentes a las ya insertadas en el MCA. Este proceso se realiza seleccionando de un grupo de renglones candidatos generados aleatoriamente el más diferente a los ya insertados teniendo en cuenta la distancia de Hamming, de acuerdo con esta distancia se identifica cual es el renglón que menos se asemeja y este se inserta.

Inicialmente en el procedimiento en la línea 1 se declara un vector filaElegida[MCA.k] de tamaño K (número de variables o columnas) en donde se almacenan todas las filas que serán insertadas en la matriz MCA hasta que se completa la totalidad de las iteraciones definas por la cantidad de filas N. Posteriormente en la línea 2 se declara una lista de vectores denominada listaFilasCandidatas en la cual se almacena un listado con los vectores candidatos a ser insertados en la matriz MCA, de acuerdo con la distancia Hamming.

Para el renglón inicial se dispuso de la línea 5 hasta la 7, en donde por cada posición j de la filaElegida se construye con un número aleatorio seleccionado entre los valores del alfabeto V adecuado para dicha columna o variable, posteriormente en la fila 16 se inserta la fila en cuestión.

En la condición else de la línea 8, que se refiere a las filas diferentes a la primera se implementó el algoritmo Greedy el cual garantiza que las filas siguientes se inserten lo más diferente posible a las previamente insertadas. En la línea 10 se envía la lista renglonesAuxiliares junto con la matriz del MCA al método calcularSimilitudes método en el cual se calcula que 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 Hamming del vectorRenglon y los renglones incluidos previamente en la matriz del MCA.

```
   Procedimiento construirMCA(Matriz MCA, entero Opcion de filas  )
 1 Vector filaElegida[MCA.K]
 2 Lista<Vector> listaFilasCandidatas
 3         for i ← 0 to MCA.N - 1 do
 4                 if i= o then
 5                         for j ← 0 to MCA.K -1 do
 6                                 filaElegida[j] ← Random(MCA.V[j])
 7                         end_for
 8                 else
 9                         for j ← 0 filasCandidatas -1 do
10                                 nuevaFila ← calcularSimilitudes(Matriz, renglonesAuxiliares)
11                                 listaCandidatos.Adicionar(nuevaFila)
12                         end_for
13                         ListaFilasCandidatas.Ordenar()
14                         filaElegida = listaFilasCandidatas.adicionar(renglonAuxiliar)
15                 end_if
16                 MCA.adicionarRenglon(RenglonElegido)
17         end_for
18 fin_procedimiento
```

***Figura 16*** *Algoritmo para la construcción del MCA inicial*

En la línea 14 se adiciona un renglonAuxiliar a la lista de vectores denominada ListaFilasCandidatos. Una vez se llene la lista con todos los renglones candidatos generales, se procede a ordenarla y se obtiene el primer elemento de la lista, que será el que menos parecido (mayor distancia de Hamming) tuvo con los renglones previamente incluidos en la Matriz MCA. Este renglón elegido entra a ser parte de la matriz del MCA en la línea 16. Este proceso se repite N veces donde N es el número de filas.

Por ejemplo, se parte de la solicitud de crear un MCA con las siguientes características MCA (k=3, t=2, v (3,2,2), N=6) en primer lugar se crea un vector aleatorio de tamaño k para inicializar la matriz y se inserta la primer fila de forma aleatoria (ver **Figura 17**).

| MCA | | |
|---|---|---|
| 0 | 0 | 0 |
| | | |
| | | |
| | | |
| | | |
| | | |

***Figura 17*** *Matriz MCA inicial con la primera fila creada aleatoriamente*

Después, siguiendo los pasos del algoritmo se crean posibles filas a insertar y se calcula la similitud de estas con las filas de la matriz MCA, que en este caso es sólo una fila (**Figura 18**).

| CANDIDATOS | | | SIMILITUD |
|---|---|---|---|
| 2 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 2 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |

***Figura 18*** *Lista de filas candidatas a ser insertadas en el MCA inicial*

La listaCandidatos se organiza de menor a mayor de acuerdo con la similitud y se toma la primera fila, que en este caso sería la fila (2,1,0) la cual se inserta en la matriz MCA (ver **Figura 19**). Es preciso ver que varias filas tienen la misma similitud, se toma según el resultado del algoritmo de ordenamiento.

| MCA | | |
|---|---|---|
| 0 | 0 | 0 |
| 2 | 1 | 0 |
| | | |
| | | |
| | | |
| | | |

***Figura 19*** *Matriz MCA inicial con dos filas insertadas*

Se repite e proceso de la misma manera hasta completar la construcción de la matriz MCA (ver **Figura 20**).

| MACs | | |
|---|---|---|
| 0 | 0 | 0 |
| 2 | 1 | 0 |
| 1 | 1 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 1 |
| 0 | 0 | 0 |

***Figura 20*** *MCA inicial completo para k=3, t=2, v (3,2,2) y N=6*

### 4.1.3 Cálculo de la Matriz P y el Fitness

El algoritmo Calcular Fitness le permite al sistema verificar que la matriz construida si corresponda a un MCA con las condiciones establecidas MCA (t, k, v, n). Partiendo de la estructura previamente presentada para la matriz P, en el siguiente algoritmo (ver **Figura 21**) como se llenan las casillas sumando un 1 a la posición en la matriz P donde se encuentra una interacción de columnas con unos

valores de alfabetos específicos. Después de calcular la Matriz P se verifica el Fitness el cual corresponde a las interacciones faltantes (cantidad de ceros en la matriz P).

```
   Procedimiento Calcular Fintess(MatrizP miP)
 1 int Faltantes
 2 boolean EsCA
 3 for filaJ←0 to miP.MaxJ do
 4         for filaMiCA←0 to miCA.N do
 5                 columnaEnP = DivisionSintetica (filaMiCA, combinacionenJ, alphabeto, fuerza)
 6                         if columnaEnP != -1
 7                                 incrementarCelda(filaJ, columnaEnP)
 8                                 calcularPeorFila(filaMiCA)
 9                         else
10                                 Exception
11                         end_if
12         end_for
13 end_for
14 EsCA = true
15 Faltantes = contarCerosEnP
16 if(Faltantes != 0) EsCA = false end_if
```

***Figura 21*** *Algoritmo para el cálculo de la matriz P*

En primer lugar, se inicializan dos variables en las líneas 1 y 2: Faltantes y EsCA, el primero es una variable entera donde se guardará la cantidad de ceros en la matriz P, que corresponde al número de t-adas (combinaciones de alfabetos basado en combinaciones de columnas) faltantes, y la segunda variable es booleana y resume si la matriz MCA es o no un verdadero MCA. En las filas siguientes desde la 3 hasta la 13 se recorre la Matriz MCA de forma vertical con el for de la línea 3, y de forma horizontal de acuerdo con el número de interacciones por fila definido por la matriz J con el for de la línea 4, de esta manera se evalúa cada posible interacción del renglón actual de la matriz MCA, en donde si dicha interacción existe se incrementa en 1 el índice de la Matriz P correspondiente a la posición de acuerdo con filaJ y columnaEnP en la línea 7.

En la línea 8 se castiga la fila en donde exista una combinación de variables que se encuentre más de una vez en la Matriz MCA (un valor mayor a 1 en la matriz P se considera una t-ada redundante). Este proceso se realiza para ser usado posteriormente en un paso de optimización que se explica en más detalle en una sección posterior.

Por último, en la línea 15 se ejecuta un método que recorre la matriz P en búsqueda de valores cero, con el fin de establecer la cantidad de combinaciones faltantes en la matriz MCA y asignarlas a la variable Faltantes (corresponde al Fitness del MCA actual), si el valor de Faltantes es mayor a cero, entonces no es una MCA, y el booleano EsCa pasa a estado falso. Continuando con el ejemplo

anterior MCA (k=3, t=2, v (3,2,2), N=6), se tiene la siguiente matriz P ver (***Figura 22***). Partiendo de la matriz MCA inicial, se calcula la Matriz P y el Fitness, inicialmente se verifican las combinaciones de acuerdo con J en este caso Columba 0 y 1 ya que es fuerza (t = 2) recorriendo todas las filas ver (**Figura 23**).

| | | 0 | | 1 | | 2 | | 3 | | 4 | | 5 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 1 | 0 | 0 0 | 0 | 0 1 | 0 | 1 0 | 0 | 1 1 | 0 | 2 0 | 0 | 2 1 | 0 |
| | | | | | | | | | | | | | |
| 0 2 | 1 | 0 0 | 0 | 0 1 | 0 | 1 0 | 0 | 1 1 | 0 | 2 0 | 0 | 2 1 | 0 |
| | | | | | | | | | | | | | |
| 1 2 | 2 | 0 0 | 0 | 0 1 | 0 | 1 0 | 0 | 1 1 | 0 | | | | |

***Figura 22*** *Matriz P inicial, sin calcular*

| K | 0 | 1 | 2 |
|---|---|---|---|
| N | MACs | | |
| 0 | 0 | 0 | 0 |
| 1 | 2 | 1 | 0 |
| 2 | 1 | 1 | 1 |
| 3 | 0 | 1 | 1 |
| 4 | 0 | 0 | 1 |
| 5 | 0 | 0 | 0 |

***Figura 23*** *Matriz MCA inicial, evaluación de las interacciones entre columna 0 y 1*

Las primeras dos filas muestran las combinaciones de valores 0,0 y 2,1 respectivamente con interacción entre las columnas 0 y 1. Entonces en la matriz P se incrementa un 1 justo para esas combinaciones en el primer renglón que muestra la combinación de columnas 0 y 1 (ver **Figura 24**).

| | | 0 | | 1 | | 2 | | 3 | | 4 | | 5 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 1 | 0 | 0 0 | 1 | 0 1 | 0 | 1 0 | 0 | 1 1 | 0 | 2 0 | 0 | 2 1 | 1 |
| | | | | | | | | | | | | | |
| 0 2 | 1 | 0 0 | 0 | 0 1 | 0 | 1 0 | 0 | 1 1 | 0 | 2 0 | 0 | 2 1 | 0 |
| | | | | | | | | | | | | | |
| 1 2 | 2 | 0 0 | 0 | 0 1 | 0 | 1 0 | 0 | 1 1 | 0 | | | | |

***Figura 24*** *Matriz P, incremento de valores según dos filas de la Matriz MCA*

Este proceso se repite hasta terminar todas las filas del MCA, después se continua con las combinaciones de columnas 0 y 2 y finalmente la combinación de las columnas 1 y 2, de esta manera se llena hasta recorrer toda la Matriz MCA dando como resultado la Matriz P presentada en la **Figura 25**.

| | | 0 | | 1 | | 2 | | 3 | | 4 | | 5 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 1 | 0 | 0 0 | 3 | 0 1 | 1 | 1 0 | 0 | 1 1 | 1 | 2 0 | 0 | 2 1 | 1 |
| | | | | | | | | | | | | | |
| 0 2 | 1 | 0 0 | 2 | 0 1 | 2 | 1 0 | 0 | 1 1 | 1 | 2 0 | 1 | 2 1 | 0 |
| | | | | | | | | | | | | | |
| 1 2 | 2 | 0 0 | 2 | 0 1 | 1 | 1 0 | 1 | 1 1 | 2 | | | | |

***Figura 25*** *Matriz P basada en el MCA inicial con Fitness = 4*

Como se puede apreciar en la **Figura 25** existen cuatro (4) ceros en la matriz P. Esto significa que no se están cubriendo todas las posibles combinaciones con la Matriz MCA actual y por consiguiente el Fitness de esta solución es igual a 4. Por lo que se menciono es necesario optimizar la respuesta generada inicialmente, este proceso se precisa en la siguiente sección.

## 4.2 Algoritmos Optimización local

Partiendo de la matriz MCA creada con el método Greedy es muy probable (cuando N es bajo para el MCA objetivo) que no se cumpla con el cubrimiento de todas las t-adas, lo que da pie a considerar el uso de un algoritmo de recocido simulado, que en este caso está construido con tres algoritmos de optimización previamente introducidos con el fin de buscar no caer en óptimos locales. A continuación, se hace una descripción de cada uno de estos tres algoritmos, haciendo especial énfasis en el algoritmo determinístico, el cual es la propuesta innovadora de este trabajo.

### 4.2.1 Algoritmo 1 – cambio de fila aleatoria según posibles alfabetos

En la **Figura 26** se describe el procedimiento con el cual el sistema crea una fila de forma aleatoria teniendo en cuenta el alfabeto correspondiente a cada columna, es decir, recorre un renglón elegido aleatoriamente y por cada columna prueba todas las posibles variables y calcula el Fitness por cada una de ellas, buscando un renglón que mejore el anteriormente construido, si es mejor entonces lo reemplaza, de esta forma continua con la siguiente columna, hasta construir toda la fila, si no es posible mejorar el Fitness, se utiliza la función Shake que modifica aleatoriamente el renglón seleccionado.

Inicialmente en las líneas 1 y 2 se declara un vector y una variable double, en el vector denominado nuevoRenglon se almacena el renglón a crear, y en la variable cambio se verificará si fue posible o no optimizar el renglón elegido aleatoriamente. En la línea 3 se recorre el renglón elegido columna por columna, y en la línea 4 se asigna a mejorValor, el valor actual de la fila elegida, en el ciclo de la línea 5 se recorre todos los posibles valores de esa columna de acuerdo al alfabeto, exceptuando el valor actual, esta discriminación se hace en el if de las líneas 7 a 9, en la línea 6 se adiciona el nuevo valor al renglón, para posteriormente en la línea 10 calcular el nuevo Fitness, si el nuevo fitness es

menor (línea 11), entonces se actualiza mejorValor con el valor encontrado (línea 12), así mismo el nuevoFitness y la variable cambio líneas 13 y 14, después de recorrer todos los posibles valores, y encontrar el que retorna un menor fitness, se asigna el valor encontrado al vector nuevoRenglon (línea 16) para posteriormente continuar con la siguiente columna, así hasta terminar con todas las columnas, lo cual al final retorna un nuevoRenglon optimizado, si no es así y no fue posible mejorar el Fitness en alguna iteración , se realiza el proceso de la función Shake (línea 19), en donde se toma el renglón original y se le cambia un valor en forma aleatoria.

```
   Procedimiento aleatorioConstruirFIla(miCA, menorFitness, out renglonElegido)
 1 Vector nuevoRenglon = CopiaDe(miCA.Matriz[seleccionado])
 2 double cambio = false
 3 for 0 ← col to col < MiSolucion.MICA.K to
 4         int mejorRenglon = nuevoRenglon[col]
 5         for 0 ← val  to val < miCA.V[col] do
 6                 nuevoRenglon[col] = val
 7                 if renglonOriginal[col] == nuevoRenglon[col]
 8                         continuar
 9                 end_if
10                 int nuevoFitness = QuitarPonerRenglon(renglonOriginal, nuevoRenglon, MiP, MiCA.V,MiCA.T, false)
11                 if  nuevoFitness < menorFitness then
12                         mejorValor = val
13                         menorFitness = nuevoFitness
14                         cambio = true
15                 end_if
16         nuevoRenglon[col] = mejorValor
17 end_for
18 if cambio = false
19         Shake(nuevoRenglon)
20 end_if
21 retorna nuevoRenglon
22 fin_procedimiento
```

***Figura 26*** *Algoritmo aleatorio de optimización local*

A continuación, se presenta un ejemplo de este método partiendo de la matriz MCA inicial (ver **Figura 20**) y la matriz P (ver **Figura 25**). Este método de optimización recibe como entrada un renglón del MCA inicial seleccionado de forma aleatoria en este caso el renglón 5 señalado en la **Figura 27**.

| MACs | | |
|---|---|---|
| 0 | 0 | 0 |
| 2 | 1 | 0 |
| 1 | 1 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 1 |
| 0 | 0 | 0 |

***Figura 27*** *Matriz MCA inicial, selección renglón aleatorio (4)*

El algoritmo toma dicho renglón e inicialmente la primera columna, en este caso, esta columna tiene un valor de 0 (ver **Figura 27**) y teniendo en cuenta el alfabeto que es 3, se tienen 3 posibles valores (0, 1, 2), entonces se reemplaza dicho valor por 1 y después por 2, y en cada iteración calcula el Fitness, si este Fitness mejora, se conserva el cambio y continua con la siguiente columna que en este caso tiene un valor de 0 y por último la columna con un valor de 1. Se tiene que en el primer ciclo el vector queda de la siguiente manera (ver **Figura 28**). Se calcula el nuevo fitness actualizando la Matriz P (**Figura 29**) y este mejoro en un 1 pasando a tener un Fitness = 3.

| | Columnas | | |
|---|---|---|---|
| Fila | 0 | 1 | 2 |
| 1 | 1 | 0 | 1 |

***Figura 28*** *renglón construido optimización local ciclo 1*

| | | 0 | | 1 | | 2 | | 3 | | 4 | | 5 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 1 | 0 | 0 0 | 3 | 0 1 | 1 | 1 0 | 1 | 1 1 | 1 | 2 0 | 1 | 2 1 | 1 |
| | | | | | | | | | | | | | |
| 0 2 | 1 | 0 0 | 2 | 0 1 | 2 | 1 0 | 0 | 1 1 | 1 | 2 0 | 1 | 2 1 | 0 |
| | | | | | | | | | | | | | |
| 1 2 | 2 | 0 0 | 2 | 0 1 | 1 | 1 0 | 0 | 1 1 | 2 | | | | |

***Figura 29*** *Nueva Matriz P con ciclo 1*

En el siguiente ciclo el algoritmo pone un 2 en la primera columna del vector quedando como se muestra en la **Figura 30**, se calcula el nuevo fitness y se encuentra que mejoro obteniendo un Fitness = 2 (ver **Figura 31**), por lo que el cambio queda registrado en esa columna.

| | Columnas | | |
|---|---|---|---|
| Fila | 0 | 1 | 2 |
| 1 | 2 | 0 | 1 |

***Figura 30*** *renglón construido optimización local ciclo 2*

| | | 0 | | 1 | | 2 | | 3 | | 4 | | 5 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 1 | 0 | 0 0 | 3 | 0 1 | 1 | 1 0 | 1 | 1 1 | 1 | 2 0 | 1 | 2 1 | 1 |
| | | | | | | | | | | | | | |
| 0 2 | 1 | 0 0 | 2 | 0 1 | 2 | 1 0 | 0 | 1 1 | 1 | 2 0 | 1 | 2 1 | 1 |
| | | | | | | | | | | | | | |
| 1 2 | 2 | 0 0 | 2 | 0 1 | 1 | 1 0 | 0 | 1 1 | 2 | | | | |

***Figura 31*** *Nueva Matriz P con ciclo 2*

Posteriormente continua con las demás columnas asignando los valores y verificando el fitness, al final la matriz obtenida se presenta en la **Figura 32**.

| MACs | | |
|---|---|---|
| 0 | 0 | 0 |
| 2 | 1 | 0 |
| 1 | 1 | 1 |
| 0 | 1 | 1 |
| 2 | 0 | 1 |
| 0 | 0 | 0 |

***Figura 32*** *Matriz obtenida después de optimización local*

### 4.2.2 Algoritmo 2 – cambio de fila aleatoria basado en t-adas faltantes

En esta sección se describe un algoritmo de Optimización que en su funcionamiento toma una tupla de valores faltantes de acuerdo con la Matriz P y lo reemplaza aleatoriamente fila por fila y verifica en cada iteración si el cambio que se realizó aporta a la solución de acuerdo con el Fitness encontrado, si reduce el Fitness entonces se reemplaza la fila optimizada. A continuación, La **Figura 33** presenta el algoritmo y luego se muestra un ejemplo para aclarar su funcionamiento.

```
   Procedimiento pseudoaleatorioFaltanenP(miCA, menorFitness, out renglonElegido)
 1 int  renglonElegido = Random(miCA.N)
 2 int totalCeros = MiP.ContarCerosEnP();
 3 Matriz lista[][] = FaltanEnP();
 4 int posCero = Random(TotalCeros)
 5 double cambio = false
 6 Vector mejorRenglon[miCA.K]
 7 Lista ListaPosiciones[miCA.N]
 8 while(ListaPosiciones.Count > 0)
 9         int seleccionado = Random(ListaPosiciones.Count)
10         ListaPosiciones.Remover(seleccionado)
11         Vector renglonOriginal = CopiaDe(miCA.Matriz[seleccionado])
12         Vector nuevoRenglon = CopiaDe(miCA.Matriz[seleccionado])
13         int filaP = lista[posCero][0];
14         int columnaP = Lista[posCero][1]
15         Vector ValoresenP[miCA.T] = multiplicacionSintetica()
16         for 0←columna to columna < MiCA.T do
17                 nuevoRenglon[MiP.J[filaP][columna]] = valoresP[columna]
18         end_for
19         int nuevoFitness = QuitarPonerRenglon(renglonOriginal, nuevoRenglon, MiP, MiCA.V,MiCA.T, false)
20         if  nuevoFitness < menorFitness then
21                 renglonElegido = seleccionado
22                 menorFitness = nuevoFitness
23                 mejorRenglon = copia(nuevoRenglon)
24                 cambio = true
25         end_if
26 end_while
27 if cambio == false then
28         mejorRenglon = copia(renglonElegido)
29         Sheke(mejorRenglon)
30 end_if
31 retorna mejorRenglon
32 fin_procedimiento
```

***Figura 33*** *Algoritmo Optimización local Pseudoaleatorio*

Desde la línea 1 hasta la línea 7 se inicializan variables pertinentes para la ejecución del algoritmo (ver **Figura 33**), iniciando por renglonElegido que se utiliza para retornar el renglón en donde se hará el cambio, en la línea 2 en TotalCeros, se almacena la cantidad de ceros que hay en la matriz P calculada de acuerdo con la Matriz MCA actual, en la línea 3 se trae todas las posiciones de la matriz P en donde hay ceros, en la línea 4 se selecciona aleatoriamente un cero de la matriz P, en la línea 5 se crea una variable para determinar si el cambio de fila es oportuno, en 6 se construye un vector para almacenar el mejorRenglon que al final retornará el método y en 7 se crea una lista con todas las filas posibles.

Desde la línea 8 hasta la línea 26 se repite un ciclo que recorre toda las t-adas faltantes, en la línea 9 se selecciona una de estas filas de la lista aleatoriamente, en 10 se resta esta fila para que en la siguiente iteración no se tenga en cuenta,

en las líneas 11 y 12 se hace una copia del renglón elegido, para que al hacer la modificación y si esta no sea satisfactoria no se pierda el renglón original, en las líneas 13 y 14 se trae las coordenadas del cero elegido en la Matriz P, en la línea 15 se trae los valores faltantes en la Matriz inicial mediante la operación multiplicación sintética.

Desde la línea 16 hasta 18 se recorre el renglón y se asignan los valores faltantes correspondientes al cero seleccionado aleatoriamente de la Matriz P en la línea 4, en la línea 19 se calcula el nuevo Fitness con el cambio realizado, si este es menor se guarda en renglonElegido la fila del MCA donde se hizo el cambio, además se guarda el nuevoFitnees en menorFitness, y se crea una copia del renglón modificado en el vector mejorRenglón, este ciclo se repite hasta que se terminen todas las filas del MCA, arrojando al final una fila óptima para reemplazar y el número de la fila que será remplazada.

Por último, desde la línea 27 hasta la 30, se verifica que, si en todo el proceso no fue posible optimizar al menos una fila, se retorna un renglón basado en el original solo que reorganizado de acuerdo con la función Shake (línea 29), que cambia un valor de la fila aleatoriamente.

Por ejemplo, se tiene el MCA inicial (ver **Figura 20**), y su respectiva matriz P (ver **Figura 25**), en donde de antemano se sabe que su Fitness es 4, cuando ingresa al algoritmo de Optimización local indicado, inicialmente se selecciona uno de los 0 aleatoriamente en este caso será el número 1 de la lista (ver **Figura 34**).

Posteriormente se selecciona un renglón de la Matriz MCA inicial, en este caso será el renglón o fila 1 (ver **Figura 35**).

De acuerdo con las coordenadas en la matriz P del cero seleccionado y mediante la multiplicación sintética se extrae los valores que hacen falta para cubrir en este caso seria 2 y 0 (ver **Figura 36**).

| Lista | Coordenadas Matriz P |
|---|---|
| 0 | 0 2 |
| 1 | 0 4 |
| 2 | 1 2 |
| 3 | 1 5 |

***Figura 34*** *Matriz lista de ceros en matriz P*

| MACs | | |
|---|---|---|
| 0 | 0 | 0 |
| 2 | 1 | 0 |
| 1 | 1 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 1 |
| 0 | 0 | 0 |

***Figura 35*** *Selección aleatoria de renglón de la matriz MCA inicial*

| | | 0 | | 1 | | 2 | | 3 | | 4 | | 5 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 1 | 0 | 0 0 | 3 | 0 1 | 1 | 1 0 | 0 | 1 1 | 1 | 2 0 | 0 | 2 1 | 1 |
| | | | | | | | | | | | | | |
| 0 2 | 1 | 0 0 | 2 | 0 1 | 2 | 1 0 | 0 | 1 1 | 1 | 2 0 | 1 | 2 1 | 0 |
| | | | | | | | | | | | | | |
| 1 2 | 2 | 0 0 | 2 | 0 1 | 1 | 1 0 | 1 | 1 1 | 2 | | | | |

***Figura 36*** *Valores faltantes de acuerdo con matriz P*

Al renglón elegido se le asignan los valores de acuerdo con la combinación pertinente, es decir, a la columna 0 el valor de 2 y a la columna 1 el valor de 0, el renglón queda como se presenta en la **Figura 37**.

| | Columnas | | |
|---|---|---|---|
| Fila | 0 | 1 | 2 |
| 1 | 2 | 0 | 0 |

***Figura 37*** *renglón 1 del MCA inicial con cambio de valores faltantes 2 y 0*

Posteriormente se recalcula el fitness, que en este caso no es mejor, incremento un cero y esto le da un Fitness = 5 (ver **Figura 38**)

| | | 0 | | 1 | | 2 | | 3 | | 4 | | 5 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 1 | 0 | 0 0 | 3 | 0 1 | 1 | 1 0 | 0 | 1 1 | 1 | 2 0 | 1 | 2 1 | 0 |
| | | | | | | | | | | | | | |
| 0 2 | 1 | 0 0 | 2 | 0 1 | 2 | 1 0 | 0 | 1 1 | 1 | 2 0 | 1 | 2 1 | 0 |
| | | | | | | | | | | | | | |
| 1 2 | 2 | 0 0 | 2 | 0 1 | 1 | 1 0 | 0 | 1 1 | 2 | | | | |

***Figura 38*** *Calculo de Matriz P con mayor Fitness*

Por lo que el algoritmo descarta esa fila y continúa con una nueva de forma aleatoria, por ejemplo, la fila 4 se hace el cambio y queda el siguiente renglón **Figura 39**.

| | Columnas | | |
|---|---|---|---|
| Fila | 0 | 1 | 2 |
| 1 | 2 | 0 | 1 |

***Figura 39*** *renglón 4 del MCA inicial con cambio de valores faltantes 2 y 0*

Como resultado después de recorrer todos los posibles renglones del MCA actual, se construyó la siguiente Matriz MCA (ver **Figura 41**) que da como resultado la Matriz P de la **Figura 42**.

| MACs | | |
|---|---|---|
| 0 | 0 | 0 |
| 2 | 1 | 0 |
| 1 | 1 | 1 |
| 0 | 1 | 1 |
| 2 | 0 | 1 |
| 0 | 0 | 0 |

***Figura 40*** *Matriz MCA optimizada localmente*

| | | 0 | | 1 | | 2 | | 3 | | 4 | | 5 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 1 | 0 | 0 0 | 3 | 0 1 | 1 | 1 0 | 0 | 1 1 | 1 | 2 0 | 1 | 2 1 | 1 |
| | | | | | | | | | | | | | |
| 0 2 | 1 | 0 0 | 2 | 0 1 | 2 | 1 0 | 1 | 1 1 | 1 | 2 0 | 1 | 2 1 | 1 |
| | | | | | | | | | | | | | |
| 1 2 | 2 | 0 0 | 2 | 0 1 | 1 | 1 0 | 0 | 1 1 | 2 | | | | |

***Figura 41*** *Nueva Matriz P después de optimización local*

### 4.2.3 Algoritmo 3 – cambio peor fila por mejor según t-adas faltantes

De acuerdo con el estado del arte se encontró un algoritmo en donde se partía de una matriz inicial para la construcción del MCA y se aplicaban dos algoritmos para optimizar la matriz inicial y encontrar el arreglo de cobertura objetivo, en uno de ellos se incrementaban las filas de acuerdo con las combinaciones faltantes. Teniendo en cuenta este algoritmo se diseñó una propuesta que se integra con los anteriores algoritmos. En esta sección se describe la implementación de un método para la optimización de la matriz MCA buscando optimizar su calidad y los tiempos de respuesta.

Esta propuesta de algoritmo cuenta con tres pasos principales:

1. Calcula la fila que menos aporta a la matriz MCA.
2. Calcula una matriz U la cual permite identificar las t-adas faltantes y construir una fila que aporte más a la solución.
3. Por último, recalcula el Fitness con el cambio establecido y si esta mejora la fila actual la reemplaza.

En el paso 1 se tiene como insumo la matriz MCA actual y la peor fila que se calcula junto con proceso de llenar la matriz P, la peor fila se establece de acuerdo con un contador de castigo por fila, el cual considera el número de repeticiones extra que contiene dicha fila, es decir, una fila que tiene muchas combinaciones repetidas no aporta al objetivo del arreglo de cobertura, estas filas serán

reemplazada por una ya optimizada calculada de acuerdo con el algoritmo que se propone.

Para el paso 2 se crea una Matriz U de acuerdo con la cantidad de combinaciones faltantes determinada por la Matriz P. La **Figura 42** describe el algoritmo para inicializar y llenar la matriz U, con la que después se calcula la fila optimizada.

```
   Procedimiento calcularrenglonOptimizado(MiP,MiCA)
 1 int columnas = MiCA.K
 2 int NumCombinaciones = MiCA.T
 3 Vector nuevoRenglon[columnas]
 4 Vector cont[columnas]
 5 Matriz lista[][] = FaltanEnP();
 6 Matiiz U[columnas][tamañodeLista]
 7 for i←0 to tamañolista do
 8         for j←0 to NumCombinaciones do
 9                 int columnaMCA = MiP[lista[j][0][j]]
10                 Vector coordenadasenP = lista[i]
11                 Vector valoresFaltantes = valoresFaltantes(coordenadasenP)
12                 U[columnaMCA][cont[columnaMCA]]= valoresFaltantes[j]
13                 cont[columnaMCA]+1
14         end_for
15 end_for
16 for k←0 to tamañodeU  do
17         nuevoRenglon[k] = calcularModa(U[k], cont[k])
18 end_for
19 retorna nuevoRenglon
20 fin_procedimiento
```

***Figura 42*** *Algoritmo determinístico para optimización local*

Desde la línea 1 hasta la 6 se inicializan las variables necesarias para el algoritmo, en la línea 1 se establece la variable columnas la cual viene directamente del MCA y será utilizada para establecer el resto de variables y definir siclos, posteriormente en la línea 2 se llama la fuerza del MCA, en la línea 3 se crea un vector con la dimensión del MCA, en el cual se almacenara la fila a ser optimizada, en la línea 4 se crea un vector de la misma dimensión el cual llevara un contador que incrementara cada vez que se encuentre una variable faltante por columna, por último se tienen dos matrices, lista y U, en lista se guardan las opciones de la matriz P en las cuales existen combinaciones faltantes y U en donde se almacena por columna que variables faltan por combinación.

Inicialmente en la línea 7 se recorre la matriz lista por filas, ya que en dicha matriz se encuentran las combinaciones faltantes representadas por un cero en la matriz P, seguidamente se recorre la Matriz J por el número de combinaciones o fuerza,

esto permite verificar con respecto a las combinaciones y posición en la matriz P en cero los valores faltantes por cada columna.

En la línea 9 se trae en el entero columnaMCA, la columna del MCA en donde hace falta una variable para la combinación dada, en el vector coordenadasenP de la línea 10 se trae de la matriz lista las coordenadas de donde se encuentra el cero en matrizP, con este dato es posible en la línea 11 encontrar los valores faltantes del MCA, con la ayuda del método traer valoresFaltantes en el cual de acuerdo a las coordenadas subministradas trae los valores relacionados con la posición del cero en la matriz P y las combinaciones faltantes en la matriz MCA, teniendo las combinaciones faltantes en la línea 12 es posible almacenar el valor faltante por columna en la matriz U, es decir, cada que exista un valor faltante para esa columna se asociara directamente en la matriz U y por último en la línea 13 se incrementas el valor del contador para así poder en la siguiente iteración asociada a esa columna posicionar el valor faltante correspondiente.

Ya construida la matriz U, se calcula la moda por columna para así establecer cuál es el valor que más falta por columna y de esta forma crear una fila para el MCA lo más adecuada posible, ya que fue construida con las combinaciones que más faltan en dicha matriz MCA.

Continuando con el ejemplo de las anteriores secciones se tienen el MCA de la **Figura 43**, adicionalmente la columna castigo que se incrementa cada vez que el algoritmo que calcula la Matriz P encuentra una combinación repetida.

| MACs | | | Castigo |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 2 |
| 0 | 0 | 0 | 3 |

***Figura 43*** *MCA inicial, columna de castigo por fila*

Como se puede apreciar en la columna castigo se le sumo a la fila que más repeticiones aporta en la Matriz MCA (valores mayores a 1) de acuerdo con la matriz P, (ver **Figura 44**) el Fitness es 4.

| | | 0 | | 1 | | 2 | | 3 | | 4 | | 5 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 1 | 0 | 0 0 | 3 | 0 1 | 1 | 1 0 | 0 | 1 1 | 1 | 2 0 | 0 | 2 1 | 1 |
| | | | | | | | | | | | | | |
| 0 2 | 1 | 0 0 | 2 | 0 1 | 2 | 1 0 | 0 | 1 1 | 1 | 2 0 | 1 | 2 1 | 0 |
| | | | | | | | | | | | | | |
| 1 2 | 2 | 0 0 | 2 | 0 1 | 1 | 1 0 | 1 | 1 1 | 2 | | | | |

***Figura 44*** *Matriz P correspondiente al MCA inicial*

Teniendo en cuenta lo anterior y como ya se estableció en el paso 1, de acuerdo con el mayor castigo se verifica cual es la fila para reemplazar (peor fila), en este caso es la fila 6 (0, 0, 0). Ahora partiendo de la siguiente información se construye la matriz U siguiendo el paso del algoritmo (ver **Figura 45**).

| Matriz Lista | | Matriz Valores |
|---|---|---|
| Coordenadas Matriz P | Combinacion de columnas | Valores Faltantes |
| 0 2 | 0 1 | 1 0 |
| 0 4 | 0 1 | 2 0 |
| 1 2 | 0 2 | 1 0 |
| 1 5 | 0 2 | 2 1 |

***Figura 45*** *Datos sacados de Matriz P*

La Matriz U está constituida por las columnas de la Matriz MCA actual a las cuales les hace falta combinaciones, y por el valor que les hace falta (ver **Figura 46**).

| Columna de MCA | Valores Faltantes | | | | Moda |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 1 | 2 | 2 |
| 1 | 0 | 0 | | | 0 |
| 2 | 0 | 1 | | | 1 |

***Figura 46*** *Matriz U, valores que más faltan por cubrir por columna de MCA*

De acuerdo con la tabla anterior se sabe que para la columna 1 (índice de numeración de columnas arranca en 0) los valores faltantes en su mayoría son 0 por lo que su moda es 0, para las columnas 0 y 2 el algoritmo aproxima la moda al valor cercano más grande, en este caso 2 y 1 respectivamente.

Por último, se ejecuta el tercer paso, que corresponde en asignar la fila construida en la peor fila encontrada y recalcular la matriz P (ver **Figura 47**).

| K | 0 | 1 | 2 |
|---|---|---|---|
| N | MACs | | |
| 0 | 0 | 0 | 0 |
| 1 | 2 | 1 | 0 |
| 2 | 1 | 1 | 1 |
| 3 | 0 | 1 | 1 |
| 4 | 0 | 0 | 1 |
| 5 | 0 | 0 | 0 |

| K | 0 | 1 | 2 |
|---|---|---|---|
| N | MACs | | |
| 0 | 0 | 0 | 0 |
| 1 | 2 | 1 | 0 |
| 2 | 1 | 1 | 1 |
| 3 | 0 | 1 | 1 |
| 4 | 0 | 0 | 1 |
| 5 | 2 | 0 | 1 |

***Figura 47*** *Cambio de fila en MCA inicial por fila optimizada*

Construyendo la fila optimizada con este algoritmo se garantiza que se cubran combinaciones que en la matriz MCA actual no se estaban considerando. La

**Figura 48** muestra la nueva Matriz P con un Fitness = 2, es decir, se cubren 2 t-adas que estaban faltando.

| | | 0 | | 1 | | 2 | | 3 | | 4 | | 5 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 1 | 0 | 0 0 | 3 | 0 1 | 1 | 1 0 | 0 | 1 1 | 1 | 2 0 | 1 | 2 1 | 1 |
| | | | | | | | | | | | | | |
| 0 2 | 1 | 0 0 | 2 | 0 1 | 2 | 1 0 | 0 | 1 1 | 1 | 2 0 | 1 | 2 1 | 1 |
| | | | | | | | | | | | | | |
| 1 2 | 2 | 0 0 | 2 | 0 1 | 1 | 1 0 | 1 | 1 1 | 2 | | | | |

***Figura 48*** *Matriz P calculada con el nuevo MCA*

# CAPÍTULO 5

## 5 POST-OPTIMIZADOR

En esta sección se presenta el segundo modulo principal del sistema, el cual se ha diseñado con el fin de dar respuesta en un tiempo optimo a los clientes que soliciten MCA y que estos no estén almacenados en el repositorio. Se utiliza para dar una respuesta rápida ya que la opción de generarlos no es una opción por la demora en tiempo de ejecución de este algoritmo.

Este algoritmo parte de un MCA ya construido y encontrado en el repositorio; este MCA subsume (cubre) al MCA requerido por el usuario y por esto es susceptible de ser optimizado. Las operaciones que realiza el post optimizador aseguran que el resultado siempre sigue siendo un MCA con las características requeridas por el tester y están encaminadas a reducir el número de filas del MCA mayor del que se partió. A continuación, se describe paso a paso el algoritmo que permite Post-Optimizar MCA con un enfoque Top-down, de lo general a lo particular.

### 5.1 VISTA GENERAL DEL ALGORITMO

El proceso de este módulo tiene como objetivo extraer primero un MCA del repositorio de mayor dimensión que contenga al MCA requerido, para que esto sea posible se requiere que el valor de fuerza (t) del MCA encontrado sea igual al t objetivo y el alfabeto del MCA encontrado sea mayor o igual en todas las columnas del MCA objetivo. A continuación, se usará la siguiente nomenclatura:

$$Vi: Alfaeto\ inicial; Vo:\ alfabeto\ objetivo;\ t: fuerza;$$

$$MCA: matriz;\ N:\ numero\ de\ filas; k: numero\ de\ columnas$$

Los grandes pasos que se desarrollan son:

- Organizar los datos de entrada: Como primer paso se recibe los parámetros del MCA encontrado $(t,\ k, Vi, N)$ y la matriz MCA correspondiente, adicional a esto se recibe el alfabeto objetivo $Vo$.
- Adecuar el alfabeto del MCA al objetivo: si el alfabeto original contiene uno o más valores adicionales se debe eliminar las columnas sobrantes
- Asignación de comodines: Se verifica todas las posiciones en el MCA encontrado y se asigna un comodín (-1) a todos los valores que en su

respectiva columna sean mayores o iguales al alfabeto objetivo correspondiente.

- Eliminar y ordenar filas: Como tercer paso se procede a la eliminación de las filas que no cumplan con combinaciones en fuerza $t$, para lo cual se establece la siguiente condición: eliminar fila si $comodines > k - t$, es decir, eliminar todas las filas que su cantidad de comodines sea mayor a la diferencia entre las columnas y la fuerza, a su vez en el momento de reconstruir la matriz MCA, se ordena con base a la cantidad de comodines por cada fila, y se establece de forma descendente, donde la primera fila contiene la menor cantidad de comodines, dejando las ultimas filas de la matriz con la mayor cantidad de comodines.
- Unir filas: En este paso se busca cómo combinar filas, esto consiste en comparar una fila con las otras y buscar una combinación que unifique a las dos, si es posible el caso, se elimina una de ella dejando una sola como la representante que cubre las combinaciones necesarias, este proceso se repite la cantidad de veces que sea necesario hasta que ya no se logre combinar más filas.
- Optimizar según algoritmo 3: Para finalizar se optimiza la matriz a través de unas combinaciones faltantes, teniendo en cuenta la matriz P, calculando e identificando una nueva fila optima que remplace la peor de ellas, este es un proceso que se realiza en la etapa del generador de MCA presentado en el **Capítulo 4**.
- Garantizar validez del MCA: como paso final se debe garantizar la validez de la nueva matriz resultante a entregar, recalculando la Matriz P y asegurándose de que no haya quedado ninguna t-ada faltante (por seguridad).

### 5.1.1 Organizar los datos de entrada

Inicialmente para manejar la información recibida en formato Json, se mapea en su respectivo objeto, para lo cual se hace necesario transformar la información a vectores y/o matrices para poder manipularla adecuadamente. La


```
{
    "Rows" :9,
    "Columns": 3,
    "Strength": 2,
    "Alphabet": "3,3,2",
    "TargetAlphabet": "3,2,2",
    "CA_notes": "2 2 1 \n2 1 0 \n2 0 1 \n1 2 1 \n1 1 1 \n1 0 0 \n0 2 0 \n0 1 1 \n0 0 0 \n"
}
```


***Figura 49*** muestra una llamada al microservicio de Post-Optimizar, en donde se tienen las características del MCA y su contenido y se tiene la configuración del MCA objetivo (TargetAlphabet).

```
{
    "Rows" :9,
    "Columns": 3,
    "Strength": 2,
    "Alphabet": "3,3,2",
    "TargetAlphabet": "3,2,2",
    "CA_notes": "2 2 1 \n2 1 0 \n2 0 1 \n1 2 1 \n1 1 1 \n1 0 0 \n0 2 0 \n0 1 1 \n0 0 0 \n"
}
```


***Figura 49*** *Modelo Json a enviar al microservicio Post-optimizador*

### 5.1.2 Adecuar el MCA original al MCA objetivo

En el proceso de Adecuación de la matriz MCA, como primera medida se debe tener claro que, si existen más columnas de las requeridas, se deben seleccionar y eliminar aquellas columnas que sobran, sabiendo que, por las propiedades de los MCA, esta operación mantiene al MCA objetivo como valido.

El procedimiento para esta adecuación de manera general es bastante corto (ver **Figura 50**), inicialmente se recorre la matriz MCA de tamaño $NxK$, si al comparar cada valor del MCA con el vector alfabeto, se cumple que:

$$MCA\left[fila_x\right]\left[columna_y\right] \geq Vo\left[columna_y\right]$$

Entonces en la posición $MCA\left[fila_x\right]\left[columna_y\right]$ se asigna un comodín (-1), el cual representa un valor que a futuro se podrá manipular sin afectar la integridad (sigue cumpliendo con las características del MCA objetivo) del MCA.

```
   Procedimiento adaptacion.convertirMCAs(filas, columnas, alfabeto, MCAs)
1  matrizMCAs
2   for  posicion_Fila  ← 0 to tamaño_filas do
3              for  posicion_Columnas ← 0 to tamaño_columnas do
4                        if posicion_matriz[posicion_Fila][ posicion_Columnas]z > = alfabeto then
5                                 matriz[posicion_Fila][ posicion_Columnas] = comodin
6                        end_if
7               end_for
8   end_for
```

***Figura 50*** *Algoritmo adaptación MCA*

Por ejemplo, para un MCA con N = 10, K=3 y alfabeto $Vo$ (5, 2, 2) que se quiere llevar a un alfabeto objetivo (3, 2, 2), la **Figura 51** en su tabla A, marca las casillas cuyos valores son igual o superiores al alfabeto de destino en cada columna. con base en esto, se pueden asignar en dichos campos un valor comodín conforme se muestra en la tabla B y C respectivamente.

### 5.1.3 Eliminar y ordenar filas

El algoritmo eliminar y ordenar filas nos permite reducir $n$, $\forall n \{1..R\}$ filas de la matriz MCA, por cada vez que se encuentre una fila con la cantidad de comodines mayores a $K - T$ es eliminada, es decir, cada fila tiene la posibilidad de superar o no el umbral de comodines donde $comodines > (K - T)$, por tanto, si no supera dicho umbral no será eliminada.

Inicialmente se declara un variable comodines (ver **Figura 52**) definida por la diferencia entre las columnas $k$ y la fuerza $t$, se inicia una lista de arreglos para guardar en ella las filas que tienen muchos comodines, en la línea 4 se obtiene cada fila correspondiente a la matriz MCA, y se verifica en línea 5 que dicha fila no se encuentre dentro de la lista de guardados, si dicha fila se encuentra en la lista no se tendrá en cuenta y se continua con la siguiente fila, en caso contrario si la fila a evaluar no está guardada, el paso siguiente será contar la cantidad de comodines que se encuentran en ella (línea 9), si al momento de comparar la cantidad de comodines de la fila con el umbral de comodines, esta es menor, dicha fila será almacenada en la lista, este proceso se realiza hasta que se haya recorrido todas las filas de la matriz, este método descarta las filas sobrantes para el MCA objetivo.

| | | | |
|---|---|---|---|
| | Columnas | | |
| 1 | 4 | 1 | 0 |
| 2 | 4 | 0 | 1 |
| 3 | 3 | 1 | 0 |
| 4 | 3 | 0 | 1 |
| 5 | 2 | 1 | 1 |
| 6 | 2 | 0 | 0 |
| 7 | 1 | 1 | 0 |
| 8 | 1 | 0 | 1 |
| 9 | 0 | 1 | 1 |
| 10 | 0 | 0 | 0 |

**Tabla A**

| Vo | 3 | 2 | 2 |
|---|---|---|---|
| | Columnas | | |
| 1 | 4 | 1 | 0 |
| 2 | 4 | 0 | 1 |
| 3 | 3 | 1 | 0 |
| 4 | 3 | 0 | 1 |
| 5 | 2 | 1 | 1 |
| 6 | 2 | 0 | 0 |
| 7 | 1 | 1 | 0 |
| 8 | 1 | 0 | 1 |
| 9 | 0 | 1 | 1 |
| 10 | 0 | 0 | 0 |

**Tabla B**

| Vo | 3 | 2 | 2 |
|---|---|---|---|
| | Columnas | | |
| 1 | -1 | 1 | 0 |
| 2 | -1 | 0 | 1 |
| 3 | -1 | 1 | 0 |
| 4 | -1 | 0 | 1 |
| 5 | 2 | 1 | 1 |
| 6 | 2 | 0 | 0 |
| 7 | 1 | 1 | 0 |
| 8 | 1 | 0 | 1 |
| 9 | 0 | 1 | 1 |
| 10 | 0 | 0 | 0 |

**Tabla C**

***Figura 51*** *adecuación de matriz original al alfabeto objetivo*

```
   Procedimiento borrarFilas_KenT
 1 comodines = columnas - fuerza
 2 listaArray <filas>
 3 for posicion_Fila ← 0 to  filas do
 4                 filas ← traerfila(MCA)
 5                 if buscarFila(listArray)
 6                       for posicion_Columnas ← 0 to columnas do
 7                          if  filaenj ← comodin then
 8                             contarComodinesEnFila
 9                          end_if
10                       end_for
11                       if contarComodinesEnFila  < comodinesKenT
12                          listaArray agregar fila
13                       end_if
14            end_for
15 end_for
16 reconstruirMCA(listArray)
```

***Figura 52*** *Algoritmo adaptación MCA*

Una vez se culmine todos los pasos, se reconstruye la matriz principal con base en la nueva lista línea 16, creando un nuevo MCA con menor cantidad de filas a partir de la matriz inicial $Ni < Nf$; donde $Ni$: $Numero\ filas\ iniciales$; $Nf$: $Numero\ de\ filas\ finales$.

Es importante reconstruir la matriz principal MCA en orden descendente basándose en la cantidad de comodines que contenga cada fila, siendo así, la primera fila estará conformada por la menor cantidad de comodines de la lista y a su vez la última fila tendrá la mayor cantidad de comodines posibles, esta transformación favorece en los próximos procesos, en donde se pueda conseguir la eliminación de las peores filas buscando llegar a un MCA con el menor número posible de filas.

En este ejemplo no se eliminan filas por tener muchos comodines ya que ninguna tiene más de uno ($comodines > (3 - 2) = 1$). Se procede a observar si hay filas duplicadas para eliminarlas. En la **Figura 53**, la fila 1 con la fila 3 y la fila 2 con la fila 4 de la tabla D (misma tabla C de la figura anterior) son iguales y por eso se debe eliminar una de cada una de las repetidas (ver tabla E). Finalmente se ordena la matriz de forma descendente con base en la cantidad de comodines, como se mencionó anteriormente (ver tabla F de la mencionada figura).

| Vo | 3 | 2 | 2 |
|---|---|---|---|
| | columnas | | |
| 1 | -1 | 1 | 0 |
| 2 | -1 | 0 | 1 |
| 3 | -1 | 1 | 0 |
| 4 | -1 | 0 | 1 |
| 5 | 2 | 1 | 1 |
| 6 | 2 | 0 | 0 |
| 7 | 1 | 1 | 0 |
| 8 | 1 | 0 | 1 |
| 9 | 0 | 1 | 1 |
| 10 | 0 | 0 | 0 |

**Tabla D**

| Vo | 3 | 2 | 2 |
|---|---|---|---|
| | columnas | | |
| 1 | -1 | 1 | 0 |
| 2 | -1 | 0 | 1 |
| 3 | 2 | 1 | 1 |
| 4 | 2 | 0 | 0 |
| 5 | 1 | 1 | 0 |
| 6 | 1 | 0 | 1 |
| 7 | 0 | 1 | 1 |
| 8 | 0 | 0 | 0 |

**Tabla E**

| Vo | 3 | 2 | 2 |
|---|---|---|---|
| | columnas | | |
| 1 | 2 | 1 | 1 |
| 2 | 2 | 0 | 0 |
| 3 | 1 | 1 | 0 |
| 4 | 1 | 0 | 1 |
| 5 | 0 | 1 | 1 |
| 6 | 0 | 0 | 0 |
| 7 | -1 | 1 | 0 |
| 8 | -1 | 0 | 1 |

**Tabla F**

***Figura 53*** *prueba de eliminación y reordenar un MCA.*

### 5.1.4 Unir filas

Este paso busca combinar filas similares si es posible. El objetivo es convertir dos filas en una sola, que pueda representar y cubrir las combinaciones suficientes dentro la matriz objetivo, para entender mejor esto se ilustra un ejemplo del procedimiento.

En primera medida se sabe que un comodín $(-1)$ es un valor que se puede asignar a cualquier valor válido del alfabeto de la columna donde se encuentra sin afectar la fila o el MCA, siendo este el caso, de la lista de MCA se toman parejas de filas que en cada columna sus valores a excepción de los comodines sean similares, como se muestra las columnas resaltadas en color verde en la **Figura 54**, para esto se presentan 4 casos.

a. Cuando los valores en la misma posición son iguales, por tanto, al combinar su resultado será el mismo valor.
b. Cuando en la misma posición existen dos comodines su resultado será un comodín.
c. Cuando hay un valor y un comodín en la misma posición su resultado es cambiar el comodín por el valor.
d. Cuando hay dos valores distintos en la misma posición, en este caso la unión de filas no se puede realizar.

La **Figura 55** presenta en resumen los tres casos que al cumplir en su totalidad dos filas, estas se pueden unir. Es decir, este proceso se realiza para todas las columnas de las filas seleccionadas de la matriz MCA, al final se tiene un resultado que es la combinación de dichas filas, y a su vez se elimina la fila con

mayor índice de numeración, el cual se sigue con la regla de eliminar en orden ascendente.

| | Filas a combinar | | |
|---|---|---|---|
| | **caso b** | **caso a** | **caso c** |
| Fila 1 | -1 | 1 | 0 |
| Fila 2 | -1 | 1 | -1 |
| Resultado | **-1** | **1** | **0** |

***Figura 54*** *tabla combinación filas para un caso dado*

La **Figura 55** muestra un ejemplo donde no es posible realiza la combinación de dos filas del MCA.

| | Filas a combinar | | |
|---|---|---|---|
| | **caso b** | **caso d** | **caso c** |
| Fila 1 | -1 | 1 | 0 |
| Fila 2 | -1 | 0 | -1 |
| Resultado | **-1** | **1** | **0** |

***Figura 55*** *tabla combinación filas para un caso fallido*

Esta etapa se realiza hasta lograr cubrir la totalidad de las filas de la matriz, con esto se logra mejorar el MCA reduciendo la cantidad de filas gracias a la combinación de ellas, sin embargo, existe la posibilidad de que esa nueva matriz reducida aun presente nuevas combinaciones entre filas, es por ello por lo que este procedimiento se somete a un ciclo anidado, hasta que este ya no pueda reducir la cantidad de filas ver **Figura 56**.

```
   Procedimiento AlgoritmoPostOptmizar.combinarFilas
1  filaAnterior = filaActual
2  mientras ( filaAnterior ¡= filaActual)
3                combinarFilas(MCA,filas,columnas)
4                filaAnterior = filaActual
5                filaActual = combinarFila(filas)
6   end_mientras
```

***Figura 56*** *Algoritmo repetitivo para combinación de filas*

### 5.1.5 Optimizar según algoritmo 3

Como ultimo procedimiento a esta etapa de post optimización se plantea el uso del algoritmo 3 presentado en la **sección 4.2.3**. de la siguiente forma.

Dado que el resultado del paso anterior puede retornar una matriz que aun contiene comodines, se implementó el algoritmo 3 de optimización para poder reemplazar estos por los valores óptimos sacados de acuerdo con la matriz P. Al momento de calcular la matriz P con respecto a la matriz generada por el último paso se encuentra que existen combinaciones faltantes, teniendo en cuenta estas se construye la matriz U, con la que se genera los valores óptimos a reemplazar, ya que se tiene un vector con los valores óptimos por columna, se procede a reemplazarse por los comodines en la matriz objetivo ver figura 57.

| Vo | 3 | 2 | 2 |
|---|---|---|---|
| | columnas | | |
| 1 | 2 | 1 | 1 |
| 2 | 2 | 0 | 0 |
| 3 | 1 | 1 | 0 |
| 4 | 1 | 0 | 1 |
| 5 | 0 | -1 | 1 |
| 6 | 0 | -1 | 0 |

tabla G

| Vo | 3 | 2 | 2 |
|---|---|---|---|
| | columnas | | |
| 1 | 2 | 1 | 1 |
| 2 | 2 | 0 | 0 |
| 3 | 1 | 1 | 0 |
| 4 | 1 | 0 | 1 |
| 5 | 0 | 1 | 1 |
| 6 | 0 | 0 | 0 |

tabla H

***Figura 57*** *Remplazo de comodines*

### 5.1.6 Garantizar validez del MCA

Finalizado el proceso, opcionalmente se puede validar que el MCA cumpla con las características solicitadas. Además, si todavía hay comodines, estos se pueden convertir a un valor del alfabeto de la columna donde se encuentra. Para asignar los comodines se busca nivelar los alfabetos de cada columna por lo cual se define la cantidad de veces que aparece cada valor en una columna y se asigna el valor con menos frecuencia hasta que se acaben los comodines de la columna, luego se continua con la siguiente columna hasta recorrer todas las columnas. El resultado de este microservicio de Post-Optimización se muestra a nivel conceptual en la **Figura 58** y como JSON en la **Figura 599**.

| Vo | 3 | 2 | 2 |
|---|---|---|---|
| | columnas | | |
| 1 | 2 | 1 | 1 |
| 2 | 2 | 0 | 0 |
| 3 | 1 | 1 | 0 |
| 4 | 1 | 0 | 1 |
| 5 | 0 | 1 | 1 |
| 6 | 0 | 0 | 0 |

tabla H

***Figura 58*** *Resultado Final Matriz MCA Post-Optimizada*


***Figura 59*** *Resultado prueba de post- optimización*

# CAPÍTULO 6

## 6 RESULTADOS Y ANÁLISIS DE ENCUESTA DE SATISFACCIÓN

### 6.1 ANÁLISIS DE RESULTADOS

En este capítulo se presentan los resultados de las pruebas realizadas a los algoritmos de Generación y Post-Optimización, teniendo en cuenta los criterios de tiempo de respuesta y cantidad de filas generadas por arreglo, aclarando que el algoritmo generador se puede poner a correr tiempo muy grandes si es necesario, pero por el escenario de uso esto no se considera necesario. Para la realización de las pruebas se implementaron tanto pruebas manuales como automatizadas utilizado la herramienta Junit y Serenitybdd para generar los reportes. En el **Anexo D** se presentan los reportes en diferentes folders cada una con un archivo index.html en el cual se puede verificar lo pertinente a cada prueba, la cantidad de pruebas satisfactorias, la respuesta del servicio con el modelo JSON de la petición POST y el tiempo de respuesta.

Se hizo necesaria la implementación de una herramienta de automatización para la realización de las pruebas ya que se quiso probar una gran cantidad de datos, de esta manera se facilita el llamado de las APIs y la generación de los reportes.

#### 6.1.1 Algoritmo Generador

Para el plan de pruebas del algoritmo generador se diseñó una automatización que ejecuta la petición al servicio de 142 MCA con las condiciones iniciales MCA (K=6, T=2, N=50) y el alfabeto va desde v (3, 2, 2, 2, 2, 2) hasta v(6, 6, 3, 3, 2, 2) pasando por diversas combinaciones en ese intervalo. La configuración anterior se utilizó para cada una de las pruebas listadas a continuación.

##### 6.1.1.1 Prueba de configuración de optimización local

Para establecer los porcentajes de ejecución adecuados para el funcionamiento de los algoritmos de optimización local, se ejecutó el algoritmo de generación modificando los porcentajes de ejecución de acuerdo con la **Tabla 2**.

***Tabla 2*** *Set de pruebas con diferentes porcentajes de ejecución.*

| Prueba | Algoritmo 1 | Algoritmo 2 | Algoritmo 3 | Promedio de Filas | Promedio de respuesta (s) |
|---|---|---|---|---|---|

| | | | | | |
|---|---|---|---|---|---|
| 1 | 30% | 60% | 10% | 27,6 | 34,79066667 |
| 2 | 20% | 30% | 50% | 27,8 | 14,19533333 |
| 3 | 10% | 20% | 70% | 28,2 | 15,05133333 |
| 4 | 70% | 20% | 10% | 27,4 | 22,96333333 |
| 5 | 30% | 70% | 0% | 27,8 | 31,95333333 |
| 6 | 33% | 33% | 33% | 28,06 | 16,54866667 |

Los resultados de cada una de las pruebas se muestran en la última columna de la **Tabla 2**. En el **Anexo E** se encuentran las tablas de resultados completas y adicionalmente los reportes generados por la herramienta Serenity. Para la realización de las pruebas subsecuentes se usaron los parámetros de la configuración 2 donde se obtiene el menor tiempo de ejecución y el promedio de filas es muy cercano al mejor obtenido en la configuración 4.

#### 6.1.1.2 Prueba con y sin algoritmo 3 de optimización

En esta sección se presentan las tablas correspondientes a las pruebas realizadas al algoritmo Generador con y sin el uso del algoritmo 3 de optimización local descrito en la **sección 4.2.3**. El reporte de Serenity concerniente a la prueba se encuentra en el archivo (Prueba 2 Serenity_deterministico) y (Prueba 5 Serenity_sindeterministico) del **Anexo D**, es preciso aclarar que en este documento solo se incluye una parte de las tablas obtenidas ya que el set de pruebas es demasiado extenso, las tablas completas quedan adjuntas en el **Anexo E.**

De acuerdo con los resultados obtenidos en las **Tabla 3** se puede evidenciar una gran mejora con respecto al tiempo de respuesta del algoritmo cuando usa el algoritmo 3 de optimización ya que el tiempo promedio de respuesta se redujo en un 42,91% si se tiene en cuenta solo los datos de la tabla presentadas; si se considera todo el set de pruebas se tiene una mejora que reduce en un 48,69% el tiempo de respuesta, esto permite confirmar la mejora que se pudo obtener implementando el algoritmo de la **sección 4.2.3**.

Por otro lado, en relación con el número de filas promedio de los arreglos obtenidos se observa una desmejora de un 1,21% equivalente a aproximadamente media fila sobre los datos de la **Tabla 3**. Este costo es muy bajo en relación con la reducción de tiempo y la posibilidad de modificar el porcentaje de veces que se usa este algoritmo permite reducir un poco esta brecha.

#### 6.1.1.3 Comparación con los resultados Charlie Colbourn

En esta sección se comparan los resultados obtenidos en cuanto a calidad (cantidad de filas) de los MCA generados con el algoritmo propuesto y los valores de referencia (valores óptimos conocidos) publicados en el repositorio de Charlie

Colbourn (ver **Tabla 4**). Adicionalmente en el **Anexo D** se encuentra el reporte de Serenity correspondiente a esta prueba.

***Tabla 3*** *Resultados con y sin el algoritmo 3*

| Columns | Strength | Alphabet | Rows_sin | Time_sin | Rows_con | Time_con | Δ Rows | Δ Time |
|---|---|---|---|---|---|---|---|---|
| 6 | 2 | 3,3,2,2,2,2 | 9 | 7,9 | 9 | 3,98 | 0 | -3,92 |
| 6 | 2 | 3,3,3,2,2,2 | 9 | 6,29 | 9 | 4,11 | 0 | -2,18 |
| 6 | 2 | 3,3,3,3,2,2 | 10 | 7,26 | 10 | 4,5 | 0 | -2,76 |
| 6 | 2 | 3,3,3,3,3,2 | 11 | 9,97 | 11 | 4,65 | 0 | -5,32 |
| 6 | 2 | 3,3,3,3,3,3 | 12 | 9,75 | 12 | 6,4 | 0 | -3,35 |
| 6 | 2 | 4,2,2,2,2,2 | 8 | 5,55 | 8 | 3,31 | 0 | -2,24 |
| 6 | 2 | 4,3,2,2,2,2 | 12 | 8,39 | 12 | 4,29 | 0 | -4,1 |
| 6 | 2 | 4,3,3,2,2,2 | 12 | 9,02 | 12 | 4,86 | 0 | -4,16 |
| 6 | 2 | 5,5,5,4,3,2 | 25 | 21,32 | 25 | 11,77 | 0 | -9,55 |
| 6 | 2 | 5,5,5,4,3,3 | 25 | 23,06 | 25 | 11,21 | 0 | -11,85 |
| 6 | 2 | 5,5,5,4,4,2 | 25 | 35,42 | 26 | 11,88 | +1 | -23,54 |
| 6 | 2 | 5,5,5,4,4,3 | 25 | 27,07 | 26 | 12,95 | +1 | -14,12 |
| 6 | 2 | 5,5,5,4,4,4 | 25 | 35,22 | 27 | 13,18 | +2 | -22,04 |
| 6 | 2 | 5,5,5,5,2,2 | 26 | 38,38 | 26 | 13,11 | 0 | -25,27 |
| 6 | 2 | 5,5,5,5,3,2 | 26 | 26,4 | 26 | 15,14 | 0 | -11,26 |
| 6 | 2 | 5,5,5,5,3,3 | 28 | 24,83 | 27 | 15,89 | -1 | -8,94 |
| 6 | 2 | 5,5,5,5,4,2 | 27 | 40,15 | 28 | 12,05 | +1 | -28,1 |
| 6 | 2 | 5,5,5,5,4,3 | 28 | 32,23 | 29 | 13,98 | +1 | -18,25 |
| 6 | 2 | 5,5,5,5,4,4 | 28 | 30,99 | 30 | 13,41 | +2 | -17,58 |
| 6 | 2 | 5,5,5,5,5,2 | 30 | 40,57 | 29 | 15,14 | -1 | -25,43 |
| 6 | 2 | 5,5,5,5,5,3 | 30 | 30,91 | 30 | 16,47 | 0 | -14,44 |
| 6 | 2 | 6,5,5,5,4,2 | 31 | 33,45 | 31 | 18,56 | 0 | -14,89 |
| 6 | 2 | 6,5,5,5,4,3 | 31 | 32,65 | 32 | 18,38 | +1 | -14,27 |
| 6 | 2 | 6,5,5,5,4,4 | 31 | 29,93 | 33 | 21,28 | +2 | -8,65 |
| 6 | 2 | 6,5,5,5,5,2 | 32 | 29,6 | 33 | 16,41 | +1 | -13,19 |
| 6 | 2 | 6,5,5,5,5,3 | 32 | 36,03 | 34 | 17,73 | +2 | -18,3 |
| 6 | 2 | 6,5,5,5,5,4 | 32 | 42,98 | 36 | 22,4 | +4 | -20,58 |
| 6 | 2 | 6,5,5,5,5,5 | 36 | 34,5 | 36 | 10,08 | 0 | -24,42 |
| 6 | 2 | 6,6,2,2,2,2 | 36 | 20,06 | 36 | 10,72 | 0 | -9,34 |
| 6 | 2 | 6,6,3,2,2,2 | 36 | 20,84 | 36 | 11,34 | 0 | -9,5 |
| 6 | 2 | 6,6,3,3,2,2 | 36 | 22,61 | 36 | 11,34 | 0 | -11,27 |
| Porcentajes | | | 24,65 | 24,95 | 25,16 | 11,95 | +0,52 | -12,99 |

Como se puede apreciar en la **Tabla 4**, existe una diferencia entre la cantidad de filas del repositorio de Colbourn con las que fue posible generar con el algoritmo propuesto en este trabajo, que en promedio se incrementó en 4 filas. No obstante, esta diferencia sólo se hace significativa cuando se tienen más de 6 columnas cada una con un alfabeto de 4 a 6 donde el número de filas adicionales llega a ser de 12 lo que implica un esfuerzo adicional de pruebas hasta de un 25% en relación con los óptimos reportados. Adicionalmente es preciso aclarar que los datos de los arreglos proporcionados por el repositorio de Colborn arrojan los resultados óptimos posibles y cada uno de ellos está construido con algoritmos diferentes, especializados en encontrar la mejor solución posible para ese arreglo en específico y sin reportar el tiempo requerido para su generación, mientras que en esta investigación se propone un algoritmo que puede crear diversos MCA y estos se generaron con poco tiempo de ejecución del algoritmo, menos de 1 minuto en todos lo casos probados.

***Tabla 4*** *Comparación con valores de referencia del repositorio de Charlie Colbourn*

| Columns | Strength | Alphabet | Rows | Rows(Colbourn) | Δ Rows |
|---|---|---|---|---|---|
| 4 | 2 | 3,3,3,3 | 9 | 9 | 0 |
| 5 | 2 | 3,3,3,3,3 | 11 | 11 | 0 |
| 7 | 2 | 3,3,3,3,3,3,3 | 13 | 12 | +1 |
| 9 | 2 | 3,3,3,3,3,3,3,3,3 | 15 | 13 | +2 |
| 10 | 2 | 3,3,3,3,3,3,3,3,3,3 | 15 | 14 | +1 |
| 5 | 2 | 4,4,4,4,4 | 16 | 16 | 0 |
| 6 | 2 | 4,4,4,4,4,4 | 19 | 19 | 0 |
| 7 | 2 | 4,4,4,4,4,4,4 | 23 | 21 | +2 |
| 9 | 2 | 4,4,4,4,4,4,4,4,4 | 25 | 22 | +3 |
| 6 | 2 | 5,5,5,5,5,5 | 33 | 25 | +8 |
| 7 | 2 | 5,5,5,5,5,5,5 | 35 | 29 | +6 |
| 8 | 2 | 5,5,5,5,5,5,5,5 | 38 | 33 | +5 |
| 9 | 2 | 5,5,5,5,5,5,5,5,5 | 39 | 35 | +4 |
| 10 | 2 | 5,5,5,5,5,5,5,5,5,5 | 41 | 36 | +5 |
| 3 | 2 | 6,6,6 | 36 | 36 | 0 |
| 4 | 2 | 6,6,6,6 | 38 | 37 | +1 |
| 5 | 2 | 6,6,6,6,6 | 43 | 39 | +4 |
| 6 | 2 | 6,6,6,6,6,6 | 48 | 41 | +7 |
| 8 | 2 | 6,6,6,6,6,6,6,6 | 54 | 42 | +12 |
| 9 | 2 | 6,6,6,6,6,6,6,6,6 | 56 | 46 | +10 |
| 10 | 2 | 6,6,6,6,6,6,6,6,6,6 | 60 | 48 | +12 |

| | | | |
|---|---|---|---|
| Porcentaje | 31,761905 | 27,80952381 | 3,952380952 |

### 6.1.2 Algoritmo Post_Optimizador

En este apartado se exponen los resultados concernientes a las pruebas realizadas al algoritmo Post_Optimizador, teniendo en cuenta los parámetros de tiempo de respuesta y la cantidad de filas, los resultados del set de pruebas se comparan con los resultados obtenidos por el algoritmo generador. Los reportes y tablas completas se encuentran en el **Anexo D y E** respectivamente.

***Tabla 5*** *Resultados algoritmo Post_Optimizador vs Generador*

| Columns | Strength | Alphabet | Rows_Opt | Time_Opt | Rows_Gen | Time_Gen | Δ Rows | Δ Time |
|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 2,2,2 | 4 | 6,5 | 4 | 7,46 | 0 | -0,96 |
| 3 | 2 | 3,3,3 | 10 | 0,11 | 9 | 3,61 | +1 | -3,5 |
| 3 | 2 | 4,4,4 | 19 | 0,11 | 16 | 5,17 | +3 | -5,06 |
| 3 | 2 | 5,5,2 | 25 | 0,08 | 25 | 8,94 | 0 | -8,86 |
| 3 | 2 | 3,2,2 | 6 | 0,09 | 6 | 2,6 | 0 | -2,51 |
| 2 | 2 | 4,2 | 8 | 0,08 | 8 | 1,84 | 0 | -1,76 |
| 4 | 2 | 2,2,2,2 | 6 | 0,12 | 5 | 3,65 | +1 | -3,53 |
| 5 | 2 | 3,3,3,3,2 | 16 | 0,09 | 10 | 6,96 | +6 | -6,87 |
| 5 | 2 | 4,4,4,4,2 | 23 | 0,08 | 16 | 13,28 | +7 | -13,2 |
| 5 | 2 | 5,5,5,5,2 | 34 | 0,08 | 27 | 20,46 | +7 | -20,38 |
| 5 | 2 | 5,5,5,5,2 | 34 | 0,06 | 27 | 20,49 | +7 | -20,43 |
| 8 | 2 | 5,3,3,3,3,3,3,3 | 17 | 0,09 | 16 | 24,84 | +1 | -24,75 |
| 8 | 2 | 5,3,3,3,3,2,2,2 | 17 | 0,09 | 15 | 23,01 | +2 | -22,92 |
| 8 | 2 | 4,3,3,3,2,2,2,2 | 17 | 0,06 | 12 | 15,84 | +5 | -15,78 |
| 6 | 2 | 4,3,3,2,2,2 | 15 | 0,07 | 12 | 9,76 | +3 | -9,69 |
| 6 | 2 | 4,3,3,2,2,2 | 15 | 0,1 | 12 | 8,95 | +3 | -8,85 |
| 6 | 2 | 4,3,3,2,2,2 | 15 | 0,06 | 12 | 9,3 | +3 | -9,24 |
| 6 | 2 | 4,3,3,2,2,2 | 15 | 0,21 | 12 | 9,69 | +3 | -9,48 |
| 8 | 2 | 2,2,2,2,2,2,2,2 | 7 | 0,05 | 6 | 9,82 | +1 | -9,77 |
| 7 | 2 | 2,2,2,2,2,2,2 | 10 | 0,06 | 6 | 8,16 | +4 | -8,1 |
| 5 | 2 | 7,6,2,2,2 | 42 | 0,05 | 42 | 19,49 | 0 | -19,44 |
| 5 | 2 | 6,5,2,2,2 | 32 | 0,13 | 30 | 13,54 | +2 | -13,41 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 2 | 7,5,2,2,2 | 38 | 0,14 | 35 | 16,71 | +3 | -16,57 |
| Porcentajes | | | 18,48 | 0,37 | 15,78 | 11,46 | 2,70 | -11,09 |

De acuerdo con los resultados presentados en la **Tabla 5**, se puede determinar que el proceso de Post_Optmizacion garantiza un tiempo de respuesta optimo (0.37 segundos), disminuyendo en promedio 11.09 segundos, lo cual se considera un aporte importante al funcionamiento general del sistema, sin embargo el número de filas en promedio incremento en 2,7 lo cual no se considera una diferencia apreciable y en general esperada, sin embargo quedan hasta 0.5 segundos que se pueden usar en el post optimizador para incluir nuevas etapas de mejora que reduzcan un poco la brecha con el generador.

## 6.2 Encuesta de satisfacción

Para cumplir con el último objetivo específico planteado en el anteproyecto se realizó una encuesta a Testers y desarrolladores de las siguientes empresas: Sophos solutions, Stefanini Group, Tata, y Tuya del grupo empresarial Éxito, los cuales interactuaron con la aplicación Web y compartieron sus opiniones y respuestas. Con estos resultados se realizaron modificaciones en la interface y se establecieron trabajos futuros relacionados con funcionalidades que los testers creen importante incluir en la aplicación desarrollada.

Para poder realizar la prueba con los testers se hizo un acompañamiento previo con el fin de explicar el modo de uso del aplicativo y como se realizaban las configuraciones para construir las pruebas, para que ellos al momento de realizar la evaluación pudieran ser objetivos con los comentarios y observaciones, además se contó con un espacio virtual en el cual se iba atendiendo a cualquier inquietud con respecto al uso de la aplicación.

La prueba se realizó de forma virtual con cada uno de los voluntarios para realizar la evaluación los cuales cuentan con una experiencia mayor a tres años en el área de calidad, es decir, están enfocados netamente al área de pruebas y testeo de los aplicativos softwares, sin embargo, también fue realizada por desarrolladores Backend, la **Tabla 6** muestra el listado de los voluntarios que realizaron la evaluación.

***Tabla 6*** *Lista de voluntarios prueba CodeTest*

| Nombre | Correo | Cargo | Empresa |
|---|---|---|---|
| Daniel Gonzales Torres | daniel.g.t@hotmail.com | Analista de Calidad | Sophos Solutions |
| Diana Marcela Chávez | dmarcela.chavez@outlook.com | Analista de Calidad | TATA |
| Maribell Muñoz Zuleta | mary198mx@gmail.com | Analista de Calidad | TATA |

| | | | |
|---|---|---|---|
| Oscar López Cobo | Oacobo@stefanini.com | Desarrollador Back | Stefanini |
| Dilan Joel Bergaño Caicedo | djcaicedo@latam.stefanini.com | Desarrollador Back | Stefanini |
| Cristian David Riaño | cristiandavid0512@gmail.com | Desarrollador Back | Stefanini |
| Cristian Camilo Belalcázar | kmilodorado@gmail.com | Desarrollador Back | Tuya |

Para la encuesta se utilizó la plantilla de “encuestafacil” denominada satisfacción del Cliente (Producto) y se adecuó para calcular el índice CSAT (Customer SATisfaction) numérico para la característica de satisfacción definida en la norma ISO 9001:2015. Este rango numérico se encuentra entre 1 y 5, donde el número 5 es el mayor y el número 1 es el menor. En este caso las preguntas se construyeron para evaluar cada una de las características del aplicativo CodeTest .En primer lugar se buscó evaluar la satisfacción en general correspondiente a toda el área de trabajo, seguidamente se solicita información acerca de la configuración y solicitud de datos, ya que esta es una parte muy importante tanto para la experiencia del usuario así como para la creación de los casos de prueba, posteriormente se solicitó al evaluador dar su opinión sobre la forma en que se entregan los resultados. Las preguntas siguientes (a partir del número 3) se crearon con el fin de definir aspectos generales sobre el aplicativo. La **Tabla 7** muestra los resultados de la encuesta.

***Tabla 7*** *Tabla de satisfacción*

| No | Pregunta | | Número de participantes | | | | |
|---|---|---|---|---|---|---|---|
| | | | **5** | **4** | **3** | **2** | **1** |
| 1 | ¿Cuál es su grado de satisfacción con el área de trabajo en general? | Fluidez de la interfaz | 1 | 5 | - | - | - |
| | | Facilidad de aprendizaje para usar esta funcionalidad | 3 | 3 | | - | - |
| | | Interfaz amigable al usuario | - | 4 | 2 | - | - |
| | | La funcionalidad resulta útil para el usuario | - | 6 | | - | - |
| 2 | ¿En términos generales, que tan adecuada es la forma de pedir los datos en la interfaz?, | Fluidez de la interfaz | - | 4 | 2 | - | |
| | | Facilidad de aprendizaje para usar esta funcionalidad | - | 3 | 3 | - | - |
| | | Interfaz amigable al usuario | - | 1 | 3 | 2 | - |
| | | La funcionalidad resulta útil para el usuario | - | 4 | 2 | - | - |
| 3 | ¿Qué tan satisfecho esta con la forma de entregar el resultado de la prueba? | Fluidez de la interfaz | 2 | 4 | - | - | - |
| | | Facilidad de aprendizaje para usar esta funcionalidad | 3 | 3 | | - | - |
| | | Interfaz amigable al usuario | 2 | 4 | | - | - |
| | | La funcionalidad resulta útil para el usuario | 5 | 1 | | - | - |
| 4 | ¿Es clara la funcionalidad de la plataforma? | | 1 | 4 | 1 | - | - |
| 5 | ¿Qué tan probable es que recomiende CodeTest? | | | 5 | 1 | - | - |
| 6 | ¿En comparación con otras alternativas de CodeTest, CodeTest es ...? | | 1 | 3 | 2 | - | - |

| | | | | | | |
|---|---|---|---|---|---|---|
| 7 | ¿Le gustaría utilizar CodeTest nuevamente? | 3 | 3 | - | - | - |
| 8 | ¿No he tenido ningún inconveniente a la hora de usar CodeTest? | - | 1 | 4 | 1 | - |
| 9 | ¿CodeTest satisface las necesidades al construir pruebas unitarias? | 1 | 5 | - | - | - |
| 10 | ¿CodeTest es fácil de usar? | - | 4 | 2 | - | - |

Específicamente las preguntas iniciales se crearon para poder tener claro el nivel de satisfacción de las funcionalidades principales de la aplicación, las cuales son, el área de trabajo donde se listan los proyectos, el ingreso de los datos para realizar los casos de prueba, y la entrega del resultado del código. Con el objetivo de limitar bien el grado de satisfacción de acuerdo con ítems más específicos se crearon apartados adicionales relacionados con la pregunta: fluidez de la interfaz, facilidad de aprendizaje para usar la funcionalidad, amigabilidad de la interfaz y utilidad de la funcionalidad para el usuario.

En cuanto a los resultados obtenidos para la primera pregunta (Área de trabajo) en general se obtuvo respuestas en satisfactorio y regular, para la segunda pregunta relacionada con la forma en que se solicitan los datos, se obtuvo resultados entre satisfecho y no satisfecho, con tendencia a satisfecho, por último con respecto a la forma en que se entregan los resultados se obtuvo resultados entre satisfecho y muy satisfecho, esto en cuanto a las 3 preguntas iniciales relacionadas con las funcionalidad principales de la plataforma, con respecto a los grados de satisfacción regulares y no satisfecho se establecieron mejoras en coordinación con los comentarios de los testers para mejorar según el alcance del proyecto las sugerencias planteadas.

La pregunta 4 en adelante indican un grado de satisfacción positivo, sin embargo, hay aspectos a resaltar importantes, como el nivel de recomendación que tiene la aplicación, como parte positiva y con respecto a la parte negativa los inconvenientes que se tubo al momento de interactuar con la plataforma.

En la segunda parte de la prueba se abrió un espacio con los usuarios basado en las siguientes preguntas abiertas.

1. ¿Cuál considera que es el aporte principal de la aplicación en su trabajo diario?
2. ¿Cuáles son sus sugerencias en general de la aplicación CodeTest?
3. ¿Qué aspecto mejoraría de la plataforma que considera el menos eficiente?

De acuerdo con las respuestas obtenidas se tuvo comentarios positivos en cuanto a la usabilidad y las necesidades que cubre el aplicativo de forma práctica en el trabajo habitual de los usuarios encuestados. Entre los comentarios a resaltar se encuentra la facilidad que brinda el aplicativo al generar el código adecuado para ejecutar las pruebas unitarias, los usuarios manifestaron que la construcción de este tipo de pruebas se ha convertido en un trabajo repetitivo y tedioso que tiene un costo en tiempo considerable. Adicionalmente se comentó que siempre existe

la posibilidad del error humano al construir las pruebas netamente con la experiencia que se tenga con respecto al cubrimiento de las pruebas.

Se sugirió mejoras en cuanto a la forma en que se ingresan los datos ya que en ocasiones según los encuestados no se tenía claro como configurar la prueba, sin embargo, también se comentó que después de realizar el proceso por primera vez las pruebas siguientes se hicieron mucho más sencillas. Una sugerencia que se comentó como trabajo futuro, es que se implemente la posibilidad de ingresar el archivo del método a probar y que el sistema automáticamente extraiga los parámetros pertinentes para la prueba y que el usuario simplemente configure los valores de prueba.

# CAPÍTULO 7

## 7 CONCLUSIONES Y TRABAJO FUTURO

En este trabajo se propuso una solución basada en las necesidades crecientes del mercado de software, que corresponden al área de calidad, ya que garantizar la calidad del software actualmente es la prioridad de toda compañía desarrolladora de software, teniendo en cuenta la cantidad de propuestas y productos software que existen actualmente en el mercado, un error en producción podría ser el causante de grandes pérdidas no solo monetarias si no de posición y prestigio de cara al cliente. Si se busca tener una empresa competitiva, mitigar los posibles errores en el desarrollo de los productos es indispensable. Esta propuesta se construyó buscando reducir el costo de tiempo y dinero a las compañías desarrolladoras de software, ya que con una cobertura de pruebas superior a la brindada por la experiencia de los testers (que varía de proyecto a proyecto y de empresa a empresa), se garantiza encontrar en gran medida la mayor cantidad de fallas y con su arreglo se disminuyen los errores en producción. Teniendo en cuenta lo anterior, abordar este enfoque brinda un gran aporte de valor no solo en el ámbito investigativo si no también en el industrial y comercial.

En este proyecto se diseñó e implementó una plataforma Web en la cual se buscó que el usuario tester pueda generar y gestionar sus pruebas de tal forma que la solución propuesta se convierte en una excelente herramienta aliada a la hora de garantizar la calidad de un producto software. De acuerdo con la investigación realizada las herramientas para generar los casos de pruebas son completamente construidas por la intuición del tester y están relacionadas fuertemente con la experiencia del mismo, a pesar de la experiencia o conocimiento que se tenga es más probable que se cometa un error, por el contrario construir las pruebas con una herramienta matemática que garantice la cobertura optima, minimiza la probabilidad de error en gran medida, es por ello que los arreglos de cobertura se posicionaron como una excelente opción a la hora de generar las posibles interacciones de prueba en un contexto de pruebas de Caja Negra para garantizar la calidad de los productos.

El sistema generador de casos de prueba que se construyó se basó en algoritmos precisos que garantizan la cobertura requerida de la prueba requerida con el menor número de recursos necesitados, orquestar estos algoritmos para proporcionar una respuesta rápida y optima al cliente se convirtió en uno de los

objetivos centrales en la construcción de la arquitectura de la plataforma Web, ya que los procesos incluidos en la aplicación conllevan diferentes tiempos de respuesta, unos más altos que otros, esto de acuerdo a las características y cantidad de subprocesos pertinentes al algoritmo. Es por esto por lo que organizar los flujos de ejecución de los algoritmos, contar con un repositorio de MCA y adicionar servicios en paralelo, permitió tener la posibilidad de que la solución responda oportunamente a los requerimientos del usuario. Por otra parte, la implementación del tipo de arquitectura de microservicios propuesta facilito la orquestación y organización de los flujos para cumplir con el objetivo planteado.

Posterior a la evaluación de la solución, se identificaron ciertos aspectos a resaltar de la plataforma, el tiempo que le ahorra el desarrollador al crear su plan de pruebas, construir los casos de prueba, la rapidez con la que se generaba el código como insumo para el Tester y el dinamismo de la interfaz, sin embargo, se recalcó también otros aspectos a mejorar, por ejemplo, la implementación de tooltips o cuadros de recomendación, agrupación de los proyectos de acuerdo a la empresa y tener la posibilidad de insertar una clase (código, ddl o jar) y facilitar con eso la configuración de las pruebas. Adicionalmente se pueda generar otro tipo de pruebas orientadas a las pruebas manuales e incluir tablas donde agrupar los casos de prueba por Historia de Usuario (HU) de acuerdo con las practicas agiles. Estas sugerencias se esperan tener en cuenta como trabajo futuro.

Durante el desarrollo de la plataforma surgieron diferentes alternativas para mejorar los algoritmos de Generación y Post-Optimización que no se pudieron incluir ya que se desbordaba el alcance del proyecto, sin embargo se pueden tener en cuenta para un trabajo futuro en el cual se busque el objetivo de mejorar de maneras alternativas los algoritmos, para el módulo de Generador, se sugiere implementar una alternativa nueva en la parte de optimización local, en donde se puede implementar adicional al algoritmo 3 una sección en donde se seleccionen varias alternativas de filas a ser reemplazadas y ver en qué posición quedaría mejor la fila optimizada. En la etapa de Post-Optimización se resalta los tiempos bajos de respuesta en comparación a la etapa de Generación, sin embargo, la cantidad de filas no fue optima, por lo que se sugiere implementar un algoritmo que en una parte intermedia del proceso evalué si se pueden adicionar más comodines, haciendo uso de la Matriz P, en donde cada que se encuentre una fila con combinaciones redundantes adicione nuevos comodines, lo cual incrementaría eventualmente la probabilidad de reducir aún mas las filas del arreglo.

Teniendo en cuenta las herramientas de Generación y Post-Optimización fue claro apreciar que el potencial de la plataforma es muy grande, debido a que no solo se puede utilizar para hacer pruebas unitarias con los frameworks de NUnit y JUnit, sino también para crear features en Cucumber, probar APIs como si de métodos se tratara con las herramientas de Serenity, es decir, la plataforma podría brindar

una gama de alternativas al Tester que le proporcionan las herramientas necesarias para mejorar significativamente la calidad de sus pruebas y por lo tanto también de sus productos software.

Desarrollando lo anterior y teniendo como base unos buenos algoritmos de Generación y Post-Optimización de arreglos de cobertura que garanticen una óptima cobertura de las pruebas, el potencial de la plataforma implementando Frameworks adicionales se incrementa, incrementando de igual manera su impacto en el sector de calidad de software como en el mercado.

Esta página ha sido dejada intencionalmente en blanco.

# CAPÍTULO 8

## 8 BIBLIOGRAFÍA

[1] A. Kovačević, “The Top 4 Trends That Will Change Software Development in the 2020s.” https://www.computer.org/publications/tech-news/trends/the-top-4-trends-that-will-change-software-development-in-the-2020s (accessed May 23, 2020).

[2] A. Tripathi, K. K. Mishra, S. Tiwari, and N. Kumar, “Improved software cost estimation models: A new perspective based on evolution in Dynamic Environment,” *Journal of Intelligent and Fuzzy Systems*, vol. 35, no. 2, IOS Press, pp. 1707–1720, Jan. 01, 2018. doi: 10.3233/JIFS-169707.

[3] H. Wu, C. Nie, F. C. Kuo, H. Leung, and C. J. Colbourn, “A Discrete Particle Swarm Optimization for Covering Array Generation,” *IEEE Trans. Evol. Comput.*, vol. 19, no. 4, pp. 575–591, Aug. 2015, doi: 10.1109/TEVC.2014.2362532.

[4] T. Sibgatullina, “Classification and calculation of different types of costs of software quality testing,” in *Annals of DAAAM and Proceedings of the International DAAAM Symposium*, 2011, pp. 1565–1566.

[5] J. Torres-Jimenez and A. Rodriguez-Cristerna, “Metaheuristic post-optimization of the NIST repository of covering arrays,” *CAAI Transactions on Intelligence Technology*, vol. 2, no. 1, Elsevier BV, pp. 31–38, Mar. 01, 2017. doi: 10.1016/j.trit.2016.12.006.

[6] C. J. Colbourn, “Covering arrays from cyclotomy,” *Designs, Codes, and Cryptography*, vol. 55, no. 2–3, Springer, pp. 201–219, May 11, 2010. doi: 10.1007/s10623-009-9333-8.

[7] S. Anand *et al.*, “The Journal of Systems and Software An orchestrated survey of methodologies for automated software test case generation,” *J. Syst. Softw.*, vol. 86, pp. 1978–2001, 2013, doi: 10.1016/j.jss.2013.02.061.

[8] 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, no. 22, Elsevier Ltd, pp. 8753–8765, Dec. 01, 2015. doi: 10.1016/j.eswa.2015.07.029.

[9] J. Adriana Timaná-Peña, C. Alberto Cobos-Lozada, and J. Torres-Jimenez, “Metaheuristic algorithms for building Covering Arrays A review,” *Revista Facultad de Ingeniería*, vol. 25, no. 43, Tunja-Boyacá, pp. 31–45, 2016. doi: 10.19053/01211129.v25.n43.2016.5295.

[10] S. Nidhra and J. Dondeti, “Black box and white box testing techniques-A literature review,” *International Journal of Embedded Systems and Applications (IJESA)*, vol. 2, no. 2, 2012. doi: 10.5121/ijesa.2012.2204.

[11] A. Andrews, A. Alhaddad, and S. Boukhris, “Black-Box model-Based regression testing of fail-Safe behavior in web applications,” *Journal of Systems and Software*, vol. 149, Elsevier Inc., pp. 318–339, Mar. 01, 2019. doi: 10.1016/j.jss.2018.11.020.

[12] M. Estrada and R. Rincón, “Framework para el proceso de testing,” Medellin: Universidad EAFIT, 2010, pp. 89–87. Accessed: May 22, 2020. [Online]. Available: http://hdl.handle.net/10784/2772

[13] S. De Telecomunicación, U. Politécnica, D. E. Madrid, J. Manuel, and S. Peño, “Pruebas de Software. Fundamentos y Técnicas,” Universidad Politécnica de Madrid, 2015. Accessed: May 22, 2020. [Online]. Available: http://oa.upm.es/40012/1/PFC_JOSE_MANUEL_SANCHEZ_PENO_3.pdf

[14] T. San Francisco, “Manual de TestLink.” Accessed: Dec. 17, 2020. [Online]. Available: https://bijao.uninorte.edu.co/AFS/ServiceDesk/Services/87/Manual_usuario_testlink.pdf

[15] “AUTOMATIC TESTING FRAMEWORK BASED ON SERENITY AND JENKINS AUTOMATED BUILD | Sa’adah | JUTI: Jurnal Ilmiah Teknologi Informasi.” http://juti.if.its.ac.id/index.php/juti/article/view/1017 (accessed Apr. 03, 2022).

[16] “6 Frameworks abiertos para testing automation - OpenExpo Europe 2022.” https://openexpoeurope.com/es/automatizacion-de-frameworks-de-codigo-abierto/ (accessed Apr. 04, 2022).

[17] J. Timaná, C. Cobos, and J. Torres-Jimenez, “Memetic algorithm for constructing covering arrays of variable strength based on global-best harmony search and simulated annealing,” in *Mexican International Conference on Artificial Intelligence*, Oct. 2018, vol. 11288 LNAI, pp. 18–32. doi: 10.1007/978-3-030-04491-6_2.

[18] S. Sabharwal, P. Bansal, and N. Mittal, “Construction of t-way covering arrays using genetic algorithm,” *Int. J. Syst. Assur. Eng. Manag.*, vol. 8, no. 2, pp. 264–274, Jun. 2017, doi: 10.1007/s13198-016-0430-6.

[19] A. Riscos Núñez, “Programación celular: resolución eficiente de problemas numéricos NP-completos,” p. 175, 2004, Accessed: May 16, 2020. [Online]. Available: https://dialnet.unirioja.es/servlet/tesis?codigo=43020

[20] Y. Xin-She, “Nature-inspired Metaheuristic Algorithms,” United kingdom: Luniver press, 2010, pp. 4–9. Accessed: Aug. 17, 2020. [Online]. Available: https://books.google.com.co/books?hl=es&lr=&id=iVB_ETlh4ogC&oi=fnd&pg=PR5&dq=Y.+Xin-She,+“Nature-inspired+Metaheuristic+Algorithms,”+Luniver+press,+2010.&ots=DwjAskIFqa&sig=D87E-YKgHeU538-8lArT6HVnNnI#v=onepage&q=Y. Xin-She%2C “Nature-inspired Metaheurist

[21] R. A. Walker and C. J. Colbourn, “Tabu search for covering arrays using permutation vectors,” *Journal of Statistical Planning and Inference*, vol. 139, no. 1, North-Holland, pp. 69–80, Jan. 01, 2009. doi: 10.1016/j.jspi.2008.05.020.

[22] S. Esfandyari and V. Rafe, “A tuned version of genetic algorithm for efficient test suite generation in interactive t-way testing strategy,” *Inf. Softw. Technol.*, vol. 94, pp. 165–185, Feb. 2018, doi: 10.1016/j.infsof.2017.10.007.

[23] G. Demiroz and C. Yilmaz, “Using simulated annealing for computing cost-aware covering arrays,” *Applied Soft Computing Journal*, vol. 49, Elsevier Ltd, pp. 1129–1144, Dec. 01, 2016. doi: 10.1016/j.asoc.2016.08.022.

[24] X. Chen, Q. Gu, A. Li, and D. Chen, “Variable strength interaction testing with an ant colony system approach,” in *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 2009, pp. 160–167. doi: 10.1109/APSEC.2009.18.

[25] X. Bao, S. Liu, N. Zhang, and M. Dong, “Combinatorial Test Generation Using Improved Harmony Search Algorithm,” *Int. J. Hybrid Inf. Technol.*, vol. 8, no. 9, pp. 121–130, 2015, doi: 10.14257/ijhit.2015.8.9.13.

[26] Y. Zhang, L. Cai, and W. Ji, “Combinatorial testing data generation based on bird swarm algorithm,” in *2017 2nd International Conference on System Reliability and Safety, ICSRS 2017*, Jan. 2018, vol. 2018-January, pp. 491–499. doi: 10.1109/ICSRS.2017.8272871.

[27] B. S. Ahmed, T. S. Abdulsamad, and M. Y. Potrus, “Achievement of minimized combinatorial test suite for configuration-aware software functional testing using the Cuckoo Search algorithm,” *Information and Software Technology*, vol. 66, Elsevier, pp. 13–29, Oct. 01, 2015. doi: 10.1016/j.infsof.2015.05.005.

[28] Y. Wang, M. Zhou, X. Song, M. Gu, and J. Sun, “Constructing Cost-Aware Functional Test-Suites Using Nested Differential Evolution Algorithm,” *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 3, Institute of Electrical and Electronics Engineers Inc., pp. 334–346, Jun. 01, 2018. doi: 10.1109/TEVC.2017.2747638.

[29] K. Z. Zamli, B. Y. Alkazemi, and G. Kendall, “A Tabu Search hyper-heuristic strategy for t-way test suite generation,” *Applied Soft Computing Journal*, vol. 44, Elsevier Ltd, pp. 57–74, Jul. 01, 2016. doi: 10.1016/j.asoc.2016.03.021.

[30] J. Torres-Jimenez, H. Avila-George, and I. Izquierdo-Marquez, “A two-stage algorithm for combinatorial testing,” *Optimization Letters*, vol. 11, no. 3, Springer Verlag, pp. 457–469, Mar. 01, 2017. doi: 10.1007/s11590-016-1012-x.

[31] I. Izquierdo-Marquez, J. Torres-Jimenez, B. Acevedo-Juárez, and H. Avila-George, “A greedy-metaheuristic 3-stage approach to construct covering arrays,” *Inf. Sci. (Ny).*, vol. 460–461, pp. 172–189, Sep. 2018, doi: 10.1016/j.ins.2018.05.047.

[32] J. Torres-Jimenez and I. Izquierdo-Marquez, “A Simulated Annealing Algorithm to Construct Covering Perfect Hash Families,” *Math. Probl. Eng.*, vol. 2018, 2018, doi: 10.1155/2018/1860673.

[33] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling, “Constructing strength three covering arrays with augmented annealing,” *Discrete Math.*, vol. 308, no. 13, pp. 2709–2722, Jul. 2008, doi: 10.1016/j.disc.2006.06.036.

[34] J. Torres-Jimenez and J. C. Perez-Torres, “A greedy algorithm to construct

covering arrays using a graph representation," *Inf. Sci. (Ny)*., vol. 477, pp. 234–245, Mar. 2019, doi: 10.1016/j.ins.2018.10.048.

[35] J. Torres-Jimenez and I. Izquierdo-Marquez, "Improved covering arrays using covering perfect hash families with groups of restricted entries," *Appl. Math. Comput.*, vol. 369, p. 124826, Mar. 2020, doi: 10.1016/j.amc.2019.124826.

[36] Y. Wang, H. Wu, X. Niu, C. Nie, and J. Xu, "An Adaptive Penalty based Parallel Tabu Search for Constrained Covering Array Generation," *Inf. Softw. Technol.*, vol. 143, p. 106768, Mar. 2022, doi: 10.1016/J.INFSOF.2021.106768.

[37] J. Torres-Jimenez and E. Rodriguez-Tello, "New bounds for binary covering arrays using simulated annealing," *Information Sciences*, vol. 185, no. 1, Elsevier, pp. 137–152, Feb. 15, 2012. doi: 10.1016/j.ins.2011.09.020.

[38] H. Avila-George, J. Torres-Jimenez, V. Hernández, and L. Gonzalez-Hernandez, "Simulated annealing for constructing mixed covering arrays," in *Advances in Intelligent and Soft Computing*, 2012, vol. 151 AISC, pp. 657–664. doi: 10.1007/978-3-642-28765-7_79.

[39] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling, "Augmenting simulated annealing to build interaction test suites," in *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, 2003, vol. 2003-January, pp. 394–405. doi: 10.1109/ISSRE.2003.1251061.

[40] 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, North-Holland, pp. 249–256, Dec. 01, 2012. doi: 10.1016/j.endm.2012.10.033.

[41] H. Ukai, X. Qu, H. Washizaki, and Y. Fukazawa, "Accelerating covering array generation by combinatorial join for industry scale software testing," *PeerJ Comput. Sci.*, vol. 8, p. e720, Feb. 2022, doi: 10.7717/PEERJ-CS.720/SUPP-2.

[42] "Decompose by subdomain."

[43] "SOLID: los 5 principios que te ayudarán a desarrollar software de calidad." https://profile.es/blog/principios-solid-desarrollo-software-calidad/ (accessed Apr. 20, 2022).