UNIVERSIDAD SEÑOR DE SIPÁN

FACULTAD DE INGENIERÍA, ARQUITECTURA Y URBANISMO

ESCUELA ACADÉMICO PROFESIONAL DE INGENIERÍA DE SISTEMAS

OPTIMIZACIÓN DE PRUEBAS DE SOFTWARE ESTRUCTURALES USANDO ALGORITMOS GENÉTICOS: REVISIÓN SISTEMÁTICA

INFORME DE INVESTIGACIÓN:
PARA OPTAR EL GRADO ACADEMICO DE BACHILLER EN INGENIERÍA DE SISTEMAS

Autor(a):

Castro Alvites Anthony Giuseppe

Asesor(a):

Mg. Tuesta Monteza Víctor Alexci

Línea de Investigación:

Ingeniería, Infraestructura y Tecnología

Pimentel – Perú
2019

DEDICATORIA

A Dios.

Quien con su bendición llena siempre mi vida, también por ser el apoyo y fortaleza en aquellos momentos de dificultad y de debilidad.

A mi Familia

Por haber sido mi apoyo a lo largo de toda mi carrera universitaria y a lo largo de mi vida. A todas las personas especiales que me acompañaron en esta etapa, aportando a mi formación tanto profesional y como ser humano.

AGRADECIMIENTO

Me van a faltar palabras para agradecer a las personas que se han involucrado en la realización de este trabajo, sin embargo merecen reconocimiento especial mi Madre y mi Padre que con su esfuerzo y dedicación me ayudaron a culminar mi carrera universitaria y me dieron el apoyo suficiente para no decaer cuando todo parecía complicado e imposible.

TABLA DE CONTENIDOS

Índice de Figuras

Índice de Tablas

Resume

La ingeniería de software ha ido evolucionando con el paso de los años en cuanto a herramientas, métodos y técnicas para la implementación y desarrollo del software. Por lo tanto, las pruebas de software son una de las partes más importantes del desarrollo de software. Entre los diferentes tipos de pruebas de software, las pruebas estructurales se utilizan ampliamente. Es por ello que en el presente trabajo, se realizó un análisis investigativo sobre las pruebas estructurales, mediante un algoritmo heurístico. En este caso el algoritmo AG (algoritmo genético), dado que en la actualidad se han encontrado diversos estudios sobre pruebas de software utilizando GA como el algoritmo de elección para optimizar casos de prueba, dado que estos se aplican a diferentes tipos de problemas, principalmente problemas de búsqueda y optimización. Teniendo en cuenta lo antes mencionado el presente proyecto tiene como objetivo proporcionar un marco para llevar a cabo el desarrollo de la revisión bibliográfica de un tema específico de forma concisa, válida y justificable. Para ello se realizó un plan de revisión en las cuales se especifica las interrogantes de la investigación, en donde la interpretación de los resultados obtenidos se basa en el material bibliográfico de los artículos relacionados a la optimización de pruebas de software mediante el uso de técnicas y algoritmos heurísticos.

Palabras Claves: Pruebas de software, Pruebas de flujo de datos, Gráficos de flujo de control, Algoritmos genéticos, Pruebas de ruta.

Abstract

Software engineering has evolved over the years in terms of tools, methods and techniques for software implementation and development. Therefore, software testing is one of the most important parts of software development. Among the different types of software tests, structural tests are widely used. That is why in the present work, an investigative analysis on the structural tests was carried out, using a heuristic algorithm. In this case, the AG algorithm (genetic algorithm), given that several studies have been found on software tests using GA as the algorithm of choice to optimize test cases, since these apply to different types of problems, mainly Search and optimization problems. Taking into account the aforementioned this project aims to provide a framework to carry out the development of the bibliographic review of a specific topic in a concise, valid and justifiable way. For this, a review plan was made in which the research questions are specified, in which the interpretation of the results obtained is based on the bibliographic material of the articles related to the optimization of software tests through the use of techniques and heuristic algorithms.

Keywords: Software Testing, Data-Flow Testing, Control flow Graph, Genetic Algorithms, Path testing

1. Introducción

La importancia del Software para la sociedad, reside en como las personas lo utilicen, hablamos de optimizar tareas, aumentar ganancias, mejorar los ingresos, optimizar tiempos, hacer la vida más sencilla, podríamos usar la palabra fundamental o indispensable y no se habla solamente de los momentos libres, sino de toda la vida cotidiana, el trabajo, la comunicación, la educación, tanto que las personas no podrían vivir sin el uso del software, tal como lo conocemos [1].

Además, la calidad es uno de los factores principales que indica el éxito o el fracaso de un software, por no decir fundamental para el desarrollo del negocio de una empresa o institución. Los proyectos de desarrollo de software sufren principalmente problemas de calidad, Dado que la mayoría de los sistemas aumentan en tamaño y complejidad, garantizar una alta calidad del software es cada vez más desafiante y costoso. Una manera de incrementar la calidad en el software consiste en disminuir la mayoría de defectos [2].

Las pruebas de software son una de las etapas más importantes y más usadas. Porque con ellas se puede evitar y reducir los riesgos en el software, Los dos principales objetivos de las pruebas de software son asegurar que el sistema que se está desarrollando se ajuste a los requisitos del cliente y revelar errores [3].

Además, existen dos tipos principales de pruebas de software; las pruebas de Caja blanca (pruebas estructurales) y pruebas de Caja Negra (Pruebas Funcionales); las pruebas de caja negra es una técnica de pruebas de software en la cual la funcionalidad se verifica sin tomar en cuenta la estructura interna del código, se enfocan únicamente en las entradas y salidas del sistema [4]. En las pruebas de caja Blanca se pretende indagar sobre la estructura interna del código, su objetivo principal es probar la lógica del programa. Cabe decir que las pruebas de caja blanca son unos de los principales métodos con los que se obtienen como resultado la disminución en un gran porciento el número de errores existentes en el software, desarrollando casos de prueba que produzcan la ejecución de cada posible ruta del programa o módulo, por tanto, se genera una mayor calidad y confiabilidad en la codificación [5].

Dentro de las pruebas de caja blanca se encuentra la prueba de flujo de datos, que es un criterio que se enfoca en comprobar todas las rutas de acceso y encontrar la cobertura de ruta para el software haciendo uso del grafico de flujo de control (CFG) [6].

Por otro lado, las limitaciones que puede presentar las pruebas de flujo de datos, se deben al código, es decir cuando se hace más complejo y hay más variables que considerar. Esto provoca un proceso más lento al momento de identificar rutas, y diseñar casos de prueba. Otro problema

con las pruebas basadas en flujo de datos ocurre cuando no existen herramientas comerciales que provean soporte a este tipo de pruebas [7].

Así mismo, algunos estudios demostraron que las pruebas de flujo de datos son particularmente adecuadas para el código orientado a objetos, dado que los métodos orientados a objetos son generalmente más cortos que los procedimientos funcionales. Pero, a pesar de ello, las pruebas de flujo de datos rara vez se utilizan en la práctica, y esto se debe a 2 razones de suma importancia, primero, los evaluadores tienen poco apoyo para medir la cobertura de flujo de datos y segundo, los evaluadores deben poner mayor esfuerzo en escribir casos de prueba que satisfagan las pruebas de flujo de datos. Además, como se mencionó anteriormente, se debe enfatizar la importancia de las herramientas de generación de pruebas automatizadas. Sin embargo, la mayoría de las herramientas de generación de pruebas sistemáticas existentes se enfocan en la declaración o en la cobertura de sucursales [8].

Como se ha dicho anteriormente, el criterio de prueba de flujo de datos no se ha utilizado muy a menudo, debido, a que es difícil escribir casos de prueba que le satisfagan, a diferencia del criterio de prueba de ruta, que es un criterio más utilizado y entendido, ya que incluye, la cobertura de sucursales, que a su vez incluye, la cobertura de declaración. Sin embargo, recientemente, se han encontrado más estudios sobre el criterio de prueba de flujo de datos utilizando GA como el algoritmo de elección para optimizar casos de prueba, dado que estos se aplican a diferentes tipos de problemas, principalmente problemas de búsqueda y optimización, o usando técnicas basada en búsquedas, que generan una solución óptima, Otros algoritmos evolutivos altamente adaptativos, como PSO (Optimización Parcial de Enjambre) y ACO (Algoritmo de la colonia de hormigas), se están aplicando para la generación de datos de prueba debido a la simplicidad y la convergencia más rápida [9].

También se podría aplicar redes neuronales ya que se han utilizado para resolver una amplia variedad de problemas, o bien heurística, que se utiliza para resolver problemas de optimización que por su naturaleza son complejos, [10].

1.1 Antecedentes de estudio

Se propuso una técnica que utiliza el código fuente del programa, la transforma en el Gráfico de flujo de control (CFG), luego calcula el grado de salida y luego aplica el algoritmo genético sobre él para generar datos de prueba valiosos durante la fase de prueba. La ventaja de utilizar el concepto de outdegree en CFG es que simplifica la técnica de cálculo de la función de aptitud y reduce el tiempo total de prueba. La técnica propuesta no solo simplifica la tarea general de aplicar

operadores de algoritmo genético sobre los datos de prueba, sino que también reduce la complejidad y el tiempo de prueba, lo que aumenta la eficiencia de la técnica [11].

Las pruebas automáticas aseguran que todos los errores se identifiquen y todos los errores se eliminen con la ayuda de varias técnicas meta-heurísticas como los algoritmos genéticos con pruebas de mutación, el algoritmo de colonias de abejas artificiales y la optimización de colonias de hormigas. The Artificial Bee Colony trabaja en la sincronización inteligente de abejas, donde se ayudan unas a otras a encontrar nodos en el código del software con resultados prometedores. Este enfoque ha sido discutido en detalle. El enfoque propuesto es más escalable, requiere menos tiempo de cómputo y es fácil de entender e implementar [12].

Se presenta un enfoque basado en DE para generar datos de prueba óptimos de acuerdo con el criterio de adecuación de la prueba de cobertura de flujo de datos. La función de acondicionamiento físico está diseñada en base a los conceptos de relaciones de dominio y métrica de distancia de rama. Se recopilan y analizan medidas como el número promedio de generaciones y el porcentaje promedio de cobertura para evaluar el desempeño del enfoque propuesto y para la comparación con las técnicas de búsqueda aleatoria, GA y PSO en un conjunto de programas de referencia. Los resultados obtenidos han demostrado que el enfoque basado en DE propuesto es competente y tiene un mejor desempeño que la búsqueda aleatoria, GA y PSO con respecto a la generación óptima de datos de prueba de acuerdo con el criterio de adecuación de la prueba de cobertura de flujo de datos [13].

Proponen una técnica para optimizar la eficiencia de las pruebas estáticas mediante la identificación de clústeres de ruta crítica mediante algoritmos genéticos. La eficiencia de la prueba se optimiza aplicando el algoritmo genético en los datos de la prueba. Los escenarios de casos de prueba se derivan del código fuente. La métrica del flujo de información se adopta en este trabajo para calcular la complejidad del flujo de información asociada con cada nodo del gráfico de flujo de control generado a partir del código fuente. Este trabajo de investigación es una extensión de nuestro trabajo de investigación anterior [14].

Se presenta un nuevo método llamado EPDG-GA que utiliza el gráfico de dominación de particiones de borde (EPDG) y el algoritmo genético (GA) para las pruebas de cobertura de sucursales. Primero, se obtiene un conjunto de Ramas críticas (CB) al analizar el EPDG del programa probado, mientras que cubrir todos los CB implica cubrir todas las ramas del Gráfico de flujo de control (CFG). Luego, las funciones de acondicionamiento físico se instrumentan en la posición correcta mediante el análisis del árbol Pre-Dominator (PreDT), y se desarrollan dos métricas para priorizar los CB. La Tabla de cobertura se establece para registrar la información de los CB y realiza un seguimiento de si una rama se ejecuta o no. GA se utiliza para generar

datos de prueba para cubrir CBs para cubrir todas las ramas. Los resultados de la comparación muestran que este enfoque es más eficiente que las pruebas aleatorias [15].

En este artículo presentamos un enfoque novedoso para el cálculo de la aptitud física para la generación de datos de prueba mediante algoritmo genético. Cálculo de la aptitud es un proceso de dos pasos. En el primer paso, se determina una secuencia de nodo de destino y en el segundo paso, la ruta de ejecución real se compara con la secuencia de nodo de destino para calcular la aptitud. Cálculo de la aptitud utiliza tanto la información de la rama y la ruta. Los experimentos indican que la técnica de acondicionamiento físico descrita produce una mejora significativa en la búsqueda [16].

El uso de algoritmos evolutivos para la generación automática de pruebas ha sido un área de interés para muchos investigadores. Algoritmo genético (AG) es una de esas formas de algoritmos evolutivos. En este trabajo de investigación, presentamos una encuesta del enfoque de GA para abordar los diversos problemas encontrados durante las pruebas de software [17].

Este documento analiza la arquitectura y la implementación de una herramienta automatizada para pruebas de flujo de datos mediante la aplicación de un algoritmo genético para la generación automática de rutas de prueba para pruebas de flujo de datos basadas en criterios seleccionados para pruebas de flujo de datos. Esta herramienta genera una población inicial aleatoria de rutas de prueba y luego, en función de los criterios de prueba de flujo de datos seleccionados, se generan nuevas rutas aplicando un algoritmo genético. Una función de aptitud en la herramienta evalúa cada cromosoma (ruta) en función de los criterios de prueba de flujo de datos seleccionados y calcula su aptitud. Hemos aplicado operadores de mutación y mutación de un punto para la generación de nuevas rutas basadas en el valor de la condición física. La herramienta de investigación propuesta, llamada ETODF, es la continuación de nuestro trabajo de investigación anterior [6] sobre pruebas de flujo de datos utilizando enfoques evolutivos. La herramienta ETODF (pruebas evolutivas del flujo de datos) se ha implementado en Java. En experimentos con esta herramienta, nuestra herramienta implementada tiene resultados mucho mejores en comparación con las pruebas aleatorias [18].

La investigación de generación de datos de prueba de software se centra principalmente en el uso de gráficos de flujo de control para producir un conjunto óptimo de casos de prueba. Por lo tanto el desempeño del enfoque propuesto se evalúa y valida en una serie de programas de muestra de diferentes niveles de tamaño y complejidad. Los resultados experimentales asociados indican un desempeño exitoso en términos de cobertura de prueba, que es significativamente mejor en comparación con los métodos de generación de datos de prueba orientados al flujo de datos dinámicos existentes [19].

Este artículo presenta una herramienta experimental para pruebas de mutación basadas en búsquedas de programas Java. Es una herramienta totalmente automatizada que admite operadores de mutación estructurados y orientados a objetos. La herramienta implementada soporta cuatro técnicas de generación de casos de prueba. Una de ellas es la prueba aleatoria y el resto de las tres son variantes del algoritmo genético. Es un programa gratuito de código abierto que se puede descargar e instalar en cualquier plataforma compatible con Java. La herramienta es fácil de usar y permite al usuario familiarizarse y aprender la herramienta con facilidad. También hemos presentado resultados experimentales en este documento para mostrar la efectividad de la herramienta. Creemos firmemente que esta herramienta tiene el potencial de llamar la atención de los evaluadores para los estudios experimentales y también alentamos las propuestas para ampliar su base de funciones [20].

Recientemente, se han aplicado diferentes novedosas técnicas de optimización basadas en la búsqueda, como Ant Colony Optimization (ACO), Artificial Bee Colony (ABC), Artificial Inmune System (AIS), Particle Swarm Optimization (PSO), para generar una ruta óptima para completar la cobertura del software. En este artículo, se ha propuesto el algoritmo basado en la optimización de la colonia de hormigas (ACO) que generará un conjunto de 'rutas óptimas y priorizará las rutas. Además, el enfoque genera una secuencia de datos de prueba dentro del dominio para usar como entradas de las rutas generadas. El enfoque propuesto garantiza la cobertura total del software con una redundancia mínima. Este documento también demuestra el enfoque propuesto aplicándolo en un módulo de programa [21].

El objetivo principal de las pruebas es encontrar los errores durante la ejecución del programa. El error puede prevalecer en cualquier parte del programa, por lo que este estudio utiliza Control Flow Graph (CFG) para representar todas las rutas del programa durante su ejecución. Algunas rutas del programa se ejecutan con poca frecuencia, por lo que con la ayuda del probador automatizado de generación de datos de prueba puede ejecutar esas rutas porque la generación de datos de prueba para estas rutas no es posible manualmente. Este trabajo proporciona una revisión de algunos de los trabajos recientes que se han realizado en el área de generación de datos de prueba AOP. Basado en ese trabajo, este trabajo propone un enfoque para generar datos de prueba para AOP usando algoritmo genético (GA) [22]

La prueba de ruta de base o prueba estructurada es un método de caja blanca que analiza el gráfico de flujo de control para encontrar el conjunto de ruta lineal independiente. La prueba de ruta es un enfoque que garantiza que cada ruta independiente a través del programa se haya ejecutado al menos una vez. El programa de origen se convierte en un gráfico de flujo de control y la ruta de prueba se extrae de este gráfico. En este documento, proponemos una nueva función de aptitud para la generación de rutas de prueba de base utilizando algoritmo genético [23].

Aplicar el algoritmo genético de múltiples poblaciones (MPGA) a la generación de datos de prueba orientados a la ruta es un intento nuevo y significativo. Después de describir cómo transformar la generación de datos de prueba orientada a la ruta en un problema de optimización, se presentó el flujo de proceso básico de la generación de datos de prueba orientada a la ruta utilizando GA. Al utilizar un clasificador de triángulos como programa bajo prueba, los resultados experimentales muestran que el enfoque basado en MPGA puede generar datos de prueba orientados a la ruta de manera más efectiva y eficiente que el enfoque simple basado en GA [24].

En este documento, nos hemos centrado en resolver la optimización multiobjetivo de los datos de prueba basados en la cobertura al proponer el algoritmo de optimización multiobjetivo Ant Lion (MOALO). Además, hemos discutido cómo el algoritmo propuesto mejora la cobertura de la ruta con un número reducido de pruebas. Para validar el algoritmo propuesto, hemos comparado los resultados experimentales obtenidos con el reposo aleatorio y los datos del algoritmo genético convencional. Estos resultados muestran que el algoritmo propuesto supera los algoritmos existentes [25].

Este artículo presenta un modelo automático de generación de datos de prueba que combina el algoritmo genético (GA) con el análisis de flujo de datos. El modelo aplica el Gráfico de flujo de control (CFG), que se divide en bloques adecuados para ayudar al análisis del flujo de datos. El gráfico de tipo de datos (DTG) presentado en este documento facilita la construcción de un objeto para realizar pruebas automáticas. Debido a la función de búsqueda del algoritmo genético, mejora la eficiencia de la generación de datos de prueba. Los resultados experimentales demuestran que nuestro modelo es más efectivo en la generación de datos de prueba y en la cobertura del flujo de datos que el algoritmo aleatorio, lo que demuestra la importancia aplicable del modelo en pruebas de software tanto para sistemas de escritorio como para sistemas integrados [26].

Este articulo propone un marco de prueba de software dinámico, que puede analizar el código fuente de un programa, crear las estructuras de datos necesarias para las pruebas automáticas, como los gráficos de flujo de control, y generar un conjunto de casos de prueba casi óptimo con referencia a criterio de cobertura de la prueba. El marco consta de dos subsistemas: el primero es un sistema de análisis de programas que identifica el tipo de declaraciones y la complejidad de las condiciones, realiza análisis de variables, extrae rutas de código y crea el gráfico de flujo de control (CFG) del programa bajo prueba . El segundo es un sistema de prueba que utiliza el CFG para generar automáticamente datos de prueba basados en computación evolutiva. El último sistema utiliza un algoritmo genético especialmente diseñado para producir el conjunto de casos de prueba que satisfacen el criterio de cobertura seleccionado. La eficacia y el rendimiento del

enfoque de prueba propuesto se evalúa y valida utilizando una variedad de programas de muestra [27].

En este documento, presentamos una herramienta basada en ventanas para generar el CFG de un programa en C automáticamente. La prueba de flujo de datos, es decir, la prueba de flujo de control depende de todo el uso indebido de las variables. Por lo tanto, seleccionar los casos de prueba para un diagrama de flujo de datos en particular no es una tarea fácil. En este artículo, el algoritmo genético se ha utilizado para generar los casos de prueba automáticamente para las pruebas de flujo de datos [28].

1.2 Planteamiento del problema de Investigación.

¿Cuál es el estado del conocimiento respecto a la optimización de pruebas de software estructurales?

1.3 Objetivos

1.3.1. Objetivo general.

Realizar una revisión del material bibliográfico sobre la optimización de pruebas de software estructurales

1.3.2. Objetivos específicos.

- Elaborar el plan de investigación
- Desarrollar el procedimiento de investigación
- Crear la documentación de la investigación

1.4 Marco teórico conceptual.

1.4.1. Ingeniería de software

El desarrollo de software es una de las principales disciplinas que en la actualidad continúa creciendo a grandes pasos, puesto que el consumo de productos software por parte de la sociedad es cada vez mayor y la necesidad de brindar soluciones a problemas cotidianos con la tecnología se vuelve cada vez más indispensable. Queda entendido que en la sociedad no podemos vivir sin el software, debido a que nos soluciona la mayoría de nuestras tareas, optimiza tiempos y hace la vida más sencilla. Por lo tanto, muchos se pueden preguntar ¿cómo se hace el software?, es aquí en donde se menciona a la ingeniería de software, la cual comprende todos los aspectos de la producción del software [29].

Entonces, software es un conjunto programas aplicativos y sistemas operativos que posibilitan a una computadora realizar tareas inteligentes, manejando los componentes físicos o hardware con instrucciones y datos por medio de diferentes tipos de programas [30].

Así mismo, ingeniería es el "Conjunto de conocimientos orientados a la invención y utilización de técnicas para el aprovechamiento de los recursos naturales o para la actividad industrial" [31].

Partiendo de estas 2 definiciones la Ingeniería de Software es una disciplina que especifica los métodos y técnicas mediante los cual aborda el desarrollo y mantenimiento del software [32]. Por tanto, cabe decir que la ingeniería de software es muy importante, ya que con ella se puede analizar, diseñar, programar y aplicar un software de manera correcta y organizada, cumpliendo con todas las especificaciones del cliente y el usuario final.

1.4.2. Ciclo de vida del software

El proceso de desarrollo de un software se le conoce formalmente como ciclo de vida del software, lo cual está conformado por cuatro etapas: concepción(donde se determinan los objetivos y se desarrolla el modelo de negocio), elaboración(donde se detalla las características del plan del proyecto), construcción(desarrollo del producto) y transición(es la transferencia del producto final a los usuarios) [33].

Como bien sabemos la Ingeniería de Software es una disciplina que ofrece métodos y técnicas para desarrollar y mantener software de calidad que solucionen problemas de todo tipo.

Entonces, ¿Qué es la calidad de un producto software?

1.4.3. Calidad del software

Define la calidad como "la propiedad o conjunto de propiedades inherentes a algo que permite juzgarlo para determinar su valor" [34]. En otras palabras, se puede decir que si el software cumple con los requisitos funcionales y no funcionales posee calidad. El software puede mostrar una buena calidad a nivel general, pero a nivel de calidad interna es muy baja. Esto se debe que, aunque el código haya sido mal escrito en estilo, elección de patrones y uso de recursos, y aun así el software cumple su misión, entonces puede obtener una apariencia de calidad externa buena.

a. Calidad Externa

En cuanto a la calidad externa, se refiere a las características del software desde una perspectiva externa, las cuales se evalúan cuando se ejecuta el producto, es decir al realizar pruebas en un ambiente simulado utilizando datos de prueba [35].

b. Calidad Interna

Se entiende por calidad interna a las características del software internamente. Dichas características generalmente se encuentran a lo largo del ciclo de vida del software, aunque éste sea rediseñado y programado nuevamente [35].

Factores para determinar la Calidad Interna

Hay ciertos factores que nos facilitan ver si nuestro código tiene una buena calidad interna. Podemos decir que un código tiene calidad cuando: es legible (fácil de entender), es intuitivo (no necesita comentarios porque es auto explicativo), es conciso (lenguaje breve y preciso), es eficiente (emplea mejor los recursos) [34].

1.4.4. Testing de software

El Testing de Software es toda una disciplina en la ingeniería de software que permite tener procesos, métodos de trabajo y herramientas para identificar defectos en el software, alcanzando un proceso de estabilidad del mismo. El Testing no es una actividad que se piensa al final del desarrollo del software, va paralelo a este. Permite que lo que se está construyendo, se realice de manera correcta de acuerdo a lo que necesita un usuario final. De ahí radica su importancia, pues es una forma de prevenir o inclusive de corregir posibles desviaciones del software antes de que sea operable. Se tenía la equivocada idea que el testing se realizaba al final, cuando ya el software estaba codificado, pero actualmente el testing de software debe ir desde el inicio del proceso [36].

Objetivos del Testing de software

a. Calidad durante todo el proceso.
b. Disminución de Costos.
c. Reducción de Riesgos.
d. Optimizar Recursos.
e. Seguimiento de estándares.

Testing en distintos aspectos de un software

Un programa es correcto si verifica su especificación. Pero, ¿que describe la especificación de un software?, la especificación de un software, en términos más generales, incluye o debería incluir otros aspectos (o requerimientos no funcionales) más allá de la funcionalidad. Por ejemplo, suelen especificarse requisitos de seguridad, desempeño, tolerancia a fallas, usabilidad, etc. Por lo tanto, al decir que un programa es correcto si verifica su especificación en realidad estamos incluyendo todos esos aspectos además de la funcionalidad [37].

En consecuencia, al testear un programa deberíamos seleccionar casos de prueba con el objetivo de testear todos esos aspectos y no solo la funcionalidad. Es decir, no solo deberíamos intentar responder preguntas tales como "¿El sistema permite cargar una transacción solo en las condiciones especificadas?", sino también otras como "¿El sistema permite cargar hasta 10.000 transacciones por minuto?", "¿El sistema permite que solo los usuarios autorizados carguen transacciones?", etc. Claramente todo esto es posible si el equipo de desarrollo cuenta con las especificaciones correspondientes. El Testing de algunos de estos aspectos no funcionales se ha especializado en alguna medida dando lugar a áreas más específicas con sus propias metodologías, técnicas y herramientas.

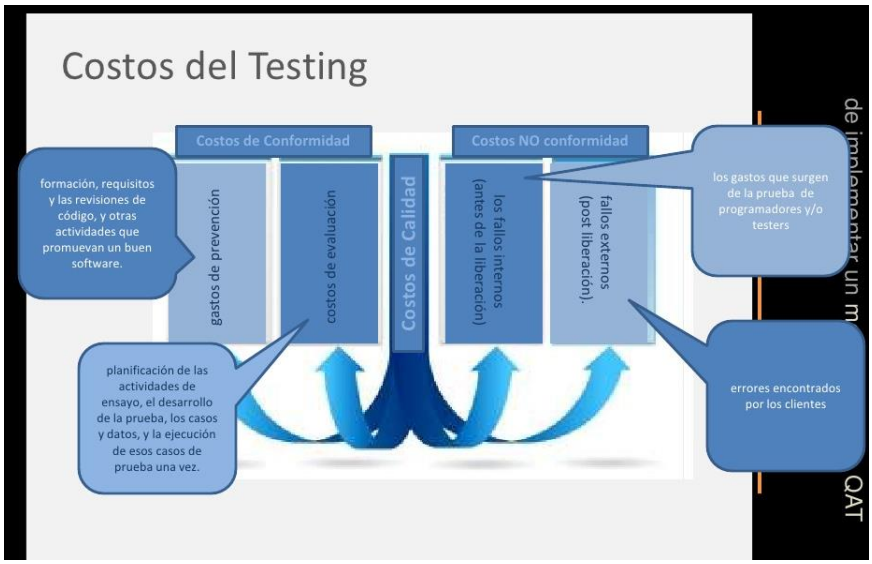

Fig. 1. Costos de Testing de software *[38]*

1.4.5. Casos de prueba

En relación con ISTQB (International Software Testing Qualifications Board), un caso de prueba es "Un conjunto de valores de entrada, precondiciones de ejecución, resultados esperados y postcondiciones de ejecución, desarrollados para un objetivo particular de condición de prueba, tal como para ejercer una ruta de un programa en particular o para verificar el cumplimiento de un requisito específico."

A su vez, los casos de prueba son sin duda uno de los elementos más importantes en cuanto a pruebas de software se refiere, dado que son un elemento que día a día le dan más valor al software, tanto así, que entre más casos y de mayor calidad se posean , el software va a adquirir más valor, adicionalmente cabe mencionar que un caso de prueba de ejecución manual bien elaborado es insumo clave en un proceso de automatización de pruebas, también se vuelven un valor agregado que apoya a los procesos de formación dentro de un área de calidad al ser fuente de información explicita del negocio. Los indicadores de resultados y la forma en que se puede

medir el desempeño de un equipo de pruebas de software se miden en base a casos de pruebas, por ejemplo, casos de pruebas ejecutados vs casos de prueba fallidos, casos de pruebas manuales vs casos de pruebas automatizados, casos de pruebas ejecutados vs esfuerzo en pruebas [39].

Cabe mencionar que hay muchos formatos para realizar casos de prueba con diferentes tipos de datos, pero en realidad no existe un formato único para diseñar casos de prueba, dado que dependiendo el negocio o escenario será necesario personalizar los datos para ajustarlo a lo que se requiere probar, sin embargo hay campos mínimos que se deben contemplar en todos los casos, sin embargo es de tener en cuenta ser muy agiles y tener presente que el caso debe cumplir una regla de oro que es que el caso de prueba sea tan claro y entendible que sea sencillo para otra persona lograr reproducirlo sin mucho esfuerzo [40].

Pasos para un buen caso de prueba

1. Crear casos de prueba simples y sencillos.
2. Que cumplan con todos los requerimientos.
3. No repetir casos de prueba.
4. Proporcione un nombre a la ID del caso de prueba para que pueda identificarse fácilmente.
5. Una vez que haya realizado los casos de prueba, los evaluará un colega, puesto que este puede encontrar errores que se pasaron por alto.

1.4.6. Técnicas de pruebas de software

Existen diversas técnicas para realizar pruebas de software, entre las más importantes tenemos la técnica de caja negra y técnica de caja blanca.

1.4.6.1. Técnicas Funcionales o de caja negra.

Las técnicas de caja negra también conocidas como Pruebas de Comportamiento, se basan en la especificación del programa o componente a ser probado para elaborar los casos de prueba. El componente sólo puede ser determinado estudiando sus entradas y las salidas obtenidas a partir de ellas. No obstante, como el estudio de todas las posibles entradas y salidas de un programa sería impracticable, para ellos se selecciona un conjunto de ellas sobre las que se realizan las pruebas. Para seleccionar el conjunto de entradas y salidas sobre las que trabajar, hay que tener en cuenta que en todo programa existe un conjunto de entradas que causan un comportamiento erróneo en nuestro sistema, y como consecuencia producen una serie de salidas que revelan la presencia de defectos. Entonces, dado que la prueba exhaustiva es imposible, el objetivo final es

pues, encontrar una serie de datos de entrada cuya probabilidad de pertenecer al conjunto de entradas que causan dicho comportamiento erróneo sea lo más alto posible [41].

Para confeccionar los casos de prueba de Caja Negra existen distintos criterios [41].

Teniendo en cuenta algunos de ellos son:

a. **Particiones de Equivalencia:** Divide el dominio de entrada de un programa en clases de datos de los que se pueden derivar casos de prueba. Así mismo, la partición equivalente se dirige a una definición de casos de prueba que descubran clases de errores, reduciendo así el número total de casos de prueba que hay que desarrollar.
b. **Análisis de Valores Límite:** Esta técnica prueba la habilidad del programa para manejar datos que se encuentren en los límites aceptables.
c. **Métodos Basados en Grafos:** Es una técnica que entiende los objetos que se van a modelar y las relaciones que conectan a estos.
d. **Pruebas de Comparación:** Consta en ejecutar versiones en paralelo para hacer una comparación en tiempo real de los resultados.
e. **Análisis Causa-Efecto:** Es una técnica de casos de prueba que proporciona una concisa representación de las condiciones lógicas y sus correspondientes acciones.

1.4.6.2. Técnicas Estructurales o de caja blanca.

La técnica de caja blanca es un método de prueba de software donde el probador entiende profundamente el funcionamiento interno del sistema o componente del sistema que se está probando. Es posible obtener una comprensión profunda del sistema o componente cuando el evaluador los comprende a nivel de programa o código. Por lo tanto, casi todo el tiempo, el evaluador debe entender o tener acceso al código fuente que conforma el sistema. Además, el probador podrá diseñar y ejecutar casos de prueba que cubran todos los escenarios posibles y las condiciones para las cuales el componente del sistema está diseñado [42].

Así mismo, se indica que la técnica de prueba de caja blanca es aplicable a los siguientes niveles de prueba de software [43].

1. **Prueba de unidad :** para probar rutas dentro de una unidad.
2. **Pruebas de integración :** Para probar rutas entre unidades.
3. **Pruebas del sistema :** para probar rutas entre subsistemas.

Sin embargo, se aplica principalmente a Pruebas Unitarias.

Ventajas de la prueba de caja blanca

- **a.** Obliga al desarrollador analizar cuidadosamente sobre la implementación.
- **b.** Revela errores en el código
- **c.** La prueba es más completa, con la posibilidad de cubrir la mayoría de los caminos.

Desventajas de la prueba de caja blanca:

- **a.** Dado que las pruebas pueden ser muy complejas, se requieren recursos altamente calificados, con un conocimiento profundo de la programación y la implementación.
- **b.** Dado que este método de prueba está estrechamente relacionado con la aplicación que se está probando, es posible que no haya herramientas disponibles para atender todo tipo de implementación.
- **c.** Cada posibilidad de que algunas líneas de código se pierdan accidentalmente.

El objetivo de la técnica es diseñar casos de prueba para que se ejecuten, al menos una vez, todas las sentencias del programa, y todas las condiciones tanto en su vertiente verdadera como falsa. Cabe decir, que puede ser impracticable realizar una prueba exhaustiva de todos los caminos de un programa. Por ello se han definido distintos criterios/técnicas de caja blanca [41].

Teniendo en cuenta dicha investigación [3], esos criterios son:

a. Pruebas de flujo de control.
b. Prueba de rama.
c. Prueba de ruta de base.
d. Prueba de bucle.
e. Prueba de flujo de datos.

Prueba de bucle

La Cobertura de bucle es un criterio de Caja Blanca, en donde el objeto es verificar los ciclos de un programa software, Estas técnicas se caracterizan por usar grafos para describir su funcionamiento. Estos grafos siempre se componen de: Aristas, nodos y regiones [44].

Existen 4 tipos de bucles que se debe tener en cuenta a la hora de analizarlos:

a. Bucle Simple.
b. Bucle Anidado.
c. Bucle concatenado.
d. Bucle no estructurado.

A. Bucle Simple

Estos ciclos son sencillos, generalmente tienen una condición

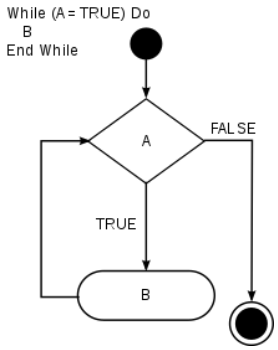

Fig. 2. Bucle Simple *[44]*

1. Pasar por alto el bucle.
2. Pasar una sola vez.
3. Pasar 2 veces por el bucle.
4. Pasar "m" veces por el bucle, siendo m<n.
5. Pasar n-1

B. Bucle Anidado

Estos ciclos son aquellos que tienen un ciclo en su interior, en este caso el bucle finalizará cuando se cumpla la condición de que N sea igual a cero

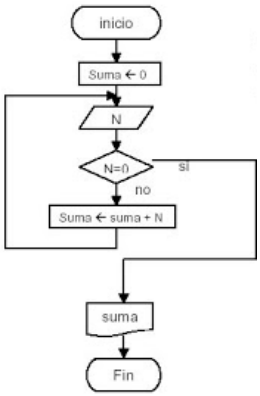

Fig. 3. Bucle Anidado *[44]*

Pasos:

a. Hacer pruebas con el bucle más interno y tratarlo como si fuera simple y el externo mantener el número mínimo de iteraciones.

b. Pruebas hacia fuera, para los internos mantener valores típicos y externos valores mínimos.

C. Bucle Concatenado

Estos ciclos son aquellos que tienen un ciclo en su interior, pero a diferencia del anterior vuelve no hasta el inicio del Ciclo externo, sino hasta sí mismo.

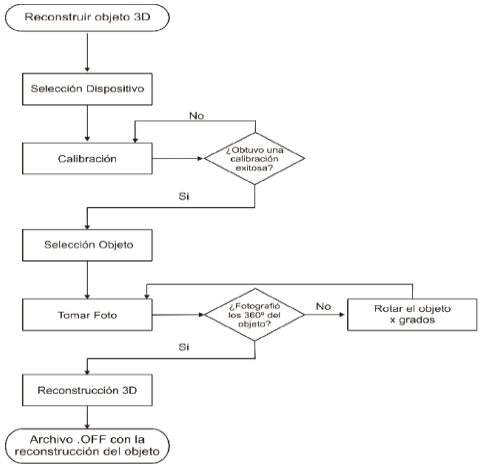

Fig. 4. Bucle Concatenado *[44]*

En estos se debe verificar que forma de concatenación tiene, si es concatenación independiente se prueba igual que los bucles simples, pero si es concatenación no independiente, se trata como bucles anidados.

D. Bucle no estructurado

Estos ciclos son aquellos que utilizan programación no Estructurada. Para este tipo de bucles se recomienda no hacer pruebas y replantearlos, pues son una muy mala práctica de programación y seria altamente riesgoso para el software. Por ejemplo, la siguiente imagen la cual se debe construir un grafo de flujo, correspondiente a la especificación del software en LDP.

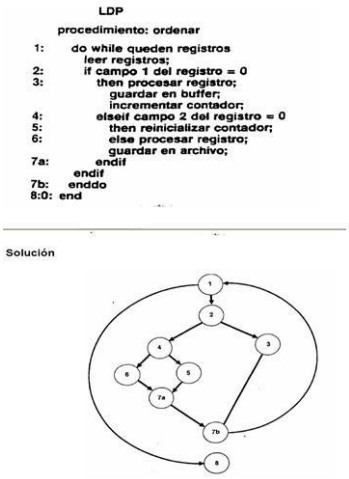

Fig. 5. Bucle no Estructurado *[44]*

Prueba de Rama

La prueba de rama también es conocido como prueba condicional o prueba de decisión, Es un criterio de prueba de caja blanca; se asegura de que cada resultado posible de la condición se prueba al menos una vez [3].

Una rama es el resultado de una decisión, por lo que la cobertura de la rama simplemente mide qué resultados de las decisiones se han probado [45].

Es decir, cada rama tomada en cada sentido, verdadera y falsa. Ayuda a validar todas las ramas en el código asegurándose de que ninguna rama conduce a un comportamiento anormal de la aplicación [46].

Formula:

Branch Testing = (Número de decisiones de los resultados probados / Número total de decisiones Resultados) x 100 %.

Ejemplo:

```
Read A
Read B
IF A+B > 10 THEN
  Print "A+B is Large"
ENDIF
If A > 5 THEN
  Print "A Large"
ENDIF
```

Fig. 6. Ejemplo de prueba de rama *[47]*

Representar mediante diagrama de flujo:

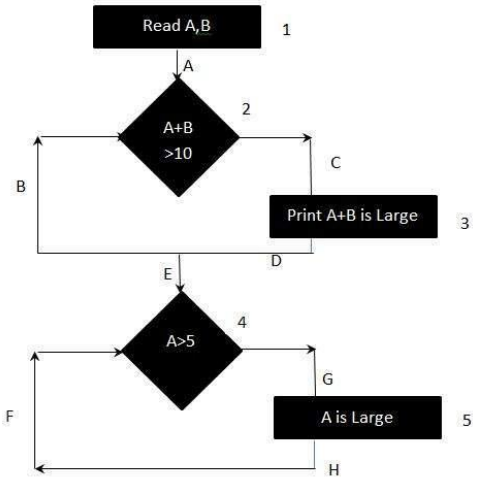

Fig. 7. Diagrama de flujo de prueba de rama *[47]*

Resultado

Se puede determinar según el ejemplo, que existen 2 posibles rutas,

- 1A-2C-3D-E-4G-5H
- 1A-2B-E-4F

Cabe mencionar además que para calcular la prueba de rama uno tiene que averiguar el número mínimo de rutas, Por lo tanto, la cobertura de la sucursal es 2 [46].

Prueba de flujo de control.

Una técnica de prueba de caja blanca para encontrar errores es la prueba de flujo de control. Las pruebas de flujo de control se aplican a casi todo el software y son efectivas para la mayoría de los programas. Es una estrategia de prueba estructural que utiliza el flujo de control del programa como modelo. La prueba de flujo de control favorece rutas más sencillas que rutas complicadas, pero menos [3].

Las investigaciones muestran que las pruebas de flujo de control detectan aproximadamente el 50% de todos los errores durante la prueba de unidad (la prueba de unidad está dominada por la prueba de flujo de control). Las pruebas de flujo de control son más efectivas para el código no estructurado que para el código estructurado. La mayoría de los errores pueden provocar errores en el flujo de control y, por lo tanto, un mal comportamiento que podría ser detectado por las

pruebas de flujo de control. Los errores de flujo de control no son tan comunes como solían ser, ya que se minimizan debido a los lenguajes de programación estructurados [3].

Algunas de las limitaciones de las pruebas de flujo de control son:

a. Las pruebas de flujo de control no pueden detectar todos los errores de inicialización.
b. Las pruebas de flujo de control no detectan los errores de especificación.
c. Es poco probable encontrar caminos y características faltantes si el programa y el modelo en el que se basan las pruebas son realizados por la misma persona.

La idoneidad de los casos de prueba medidos con una métrica llamada cobertura (la cobertura es una medida de la integridad del conjunto de casos de prueba).

los métodos de cobertura para pruebas de flujo de control de acuerdo a [44] son:

a. Cobertura de Decisión: Se escriben casos de prueba suficientes para que cada decisión dentro del programa se ejecute una vez con resultado verdadero y otra con el falso.
b. Cobertura de Condiciones: Se escriben casos de prueba suficientes para que cada condición simple en una decisión tenga un resultado verdadero y otro falso.
c. Cobertura Decisión/Condición: Se escriben casos de prueba suficientes para que cada condición en una decisión tome todas las posibles salidas, al menos una vez, y cada decisión tome todas las posibles salidas, al menos una vez.
d. Cobertura de Condición Múltiple: Se escriben casos de prueba suficientes para que todas las combinaciones posibles de resultados de cada condición se invoquen al menos una vez.

Prueba de ruta de base.

En las pruebas de software, hay muchas rutas entre la entrada y la salida de un programa de software. Por lo tanto, es difícil probar completamente todos los caminos de una unidad simple. Este es un reto cuando diseñamos casos de prueba. Necesitamos eliminar las pruebas redundantes proporcionando una cobertura de prueba adecuada para pruebas efectivas. Una de las formas de hacerlo, es mediante la aplicación de un método denominado prueba de ruta de base. La prueba de ruta de base o prueba estructurada es un método de prueba de caja blanca que se utiliza para diseñar casos de prueba destinados a examinar todas las rutas de ejecución posibles al menos una vez [48].

Pasos para la prueba de ruta base:

Primero, se debe escoger un pedazo de código y después realizar los siguientes pasos.

1. Dibuja un gráfico de flujo de control.
2. Determinar la complejidad ciclomática.
3. Encuentra un conjunto básico de caminos.
4. Generar casos de prueba para cada ruta.

Ejemplo.

Código en la prueba de ruta base, de acuerdo con [48].

```
Function fn_delete_element (int value, int array_size, int array [])
        {
                1 int i;
                location = array_size + 1;

                2 for i = 1 to array_size
                3 if (array[i] == value)
                4 location = i;
                end if;
                end for;

                5 for i = location to array_size
                6 array[i] = array[i+1];
                end for;
                7 array_size --;
        }
```

Pasos de prueba de ruta base.

1. **Gráfico de flujo de control.**

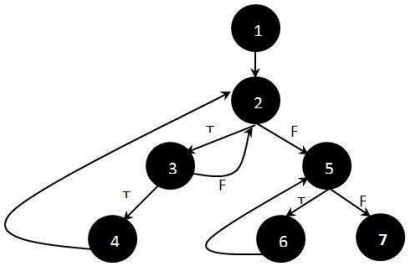

Fig. 8. Gráfico de flujo de prueba de ruta base *[48]*

2. **Determinar la complejidad Ciclomática.**

Una complejidad ciclomática es una métrica de software que proporciona una medida cuantitativa. Cuando se utiliza en el contexto del método de prueba de ruta de base esta define el número de rutas independientes (cualquier ruta a través del programa que introduce al

menos un nuevo conjunto de instrucciones de procesamiento o una nueva condición) en el conjunto de bases del programa y proporciona un límite superior para Número de pruebas requeridas para garantizar la cobertura de todas las declaraciones del programa. Podemos calcular la complejidad ciclomática para un gráfico de flujo en cualquiera de las formas dadas [3].

V(G) = Número de regiones

V(G) = Aristas - Nodos + 2

V(G) = Número de nodos predicado + 1

En este caso, para el ejemplo dado se usó la complejidad ciclomática "Aristas – Nodos + 2"

Lo cual seria "9 – 7 + 2 = 4"

3. **Numero de Caminos**

El número de caminos son 4

```
Path 1:  1 - 2 - 5 - 7
Path 2:  1 - 2 - 5 - 6 - 7
Path 3:  1 - 2 - 3 - 2 - 5 - 6 - 7
Path 4:  1 - 2 - 3 - 4 - 2 - 5 - 6 - 7
```

Fig. 9. Determinar número de caminos de prueba de ruta de base *[48]*

4. **Generar Casos de Prueba:** Por último, se generan casos de Pruebas para probar cada camino.

Prueba de Flujo de datos

Las pruebas de flujo de datos es una prueba basadas en la selección de rutas a través del flujo de control del programa para explorar secuencias de eventos relacionados con el estado de variables u objetos de datos. La prueba de flujo de datos se centra en los puntos en los que las variables reciben valores y los puntos en los que se utilizan estos valores [49].

¿Porque es importante?

Los diagramas de flujo de control son una piedra angular en la prueba de la estructura de los programas de software. Al examinar el flujo de control entre los diversos componentes, podemos

diseñar y seleccionar casos de prueba. La prueba de flujo de datos es una técnica de prueba de flujo de control que también examina el ciclo de vida de las variables de datos. El uso de pruebas de flujo de datos conduce a un conjunto de pruebas más rico que se concentra en el uso incorrecto de los datos debido a errores de codificación [50].

También proporciona una forma más intensiva de seleccionar los casos de prueba, que se encuentra entre las tareas más intensivas en mano de obra en su aplicación (esto también es válido para otros criterios de prueba estructurales) y tiene un fuerte impacto en su eficacia y eficiencia.

Autor:

Fue Herman quien introdujo la concepción original de prueba de flujo de datos (DTF) en 1976. Desde entonces, se han realizado varios estudios, tanto teóricos como empíricos, para analizar la complejidad y efectividad de la prueba de flujo de datos. En las últimas cuatro décadas, las pruebas de flujo de datos se han preocupado continuamente, y se proponen varios enfoques desde diferentes aspectos para realizar pruebas automáticas y eficientes de flujo de datos [51].

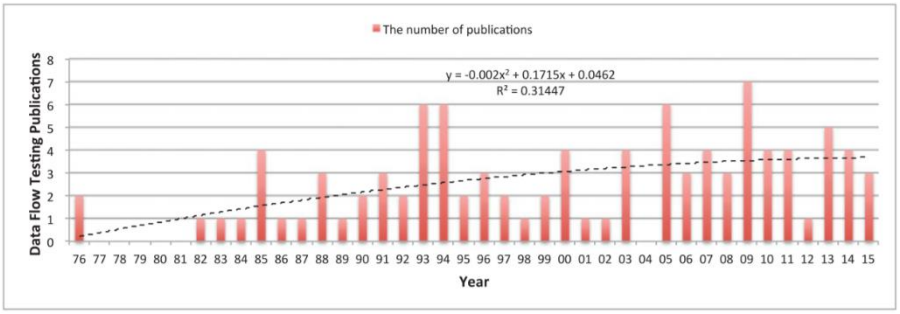

Fig. 10. Publicaciones de prueba de flujo de datos *[51]*

La idea básica detrás de esta forma de prueba, es revelar los errores y errores de codificación, lo que puede resultar en una implementación y uso incorrectos de las variables de datos o valores de datos en el código de programación, es decir, anomalías de datos [52], como:

a. Todas las variables de datos, presentes en el código de programación han sido inicializadas o no.
b. Las variables de datos que se ponen en uso, se han inicializado previamente o no.
c. Si las variables de datos inicializados, se han utilizado al menos una vez en el código de programación.

Generalmente, los objetos de datos que ocurren en el ciclo de vida de la programación pasan por 3 fases.

a. Definición: las variables de datos se definen, crean e inicializan, junto con la asignación de la memoria a ese objeto de datos en particular.

b. Uso: Las variables de datos declaradas se pueden usar en el código de programación, en dos formas: C en la forma computacional y P como parte del predicado.

c. Eliminación: la memoria asignada a las variables, se libera y se utiliza para otro uso.

TABLA I. ANOMALÍAS DE LA PRUEBA DE FLUJO DE DATOS *[53]*

Anomalía		Explicación
~d	Primer definir	Permitido
du	Definir - uso	Permitido - Caso normal
dk	Definir - eliminar	Error potencial - Los datos se eliminan sin uso después de la definición.
~u	Primer uso	Error potencial - Los datos se utilizan sin definición.
ud	Uso - definir	Permitido - Los datos se utilizan y luego se redefinen.
uk	Uso - eliminar	Permitido.
~k	Primer eliminar	Error potencial - Los datos se eliminan antes de la definición.
ku	Eliminar - uso	Defecto grave - Los datos se utilizan después de ser eliminados.
kd	Eliminar - definido	Permitido - Los datos se eliminan y luego se redefinen.
dd	Definir - definir	Error potencial - Doble definición
uu	Uso - uso	Permitido - Caso normal.
kk	Eliminar - eliminar	Error potencial.
d~	Definir	último error potencial.
u~	Uso	último permitido.
k~	Eliminar ultimo	Permitido - Caso normal.

Tipo de pruebas de flujo de datos

El proceso de prueba de flujo de datos puede llevarse a cabo a través de dos enfoques o metodologías diferentes [52].

a. Prueba de flujo estático:

En las pruebas estáticas, el estudio y el análisis del código se realizan sin realizar la ejecución real del código fuente, como el uso incorrecto de los archivos de encabezado o los archivos de la biblioteca o error de sintaxis.

b. Prueba de flujo Dinámico:

Implica la ejecución del código, para monitorear y observar los resultados intermedios. Básicamente, cuida la cobertura de las propiedades del flujo de datos.

Ejemplo en prueba de flujo estático

En el análisis estático, el código fuente se analiza sin ejecutarlo. Consideremos un ejemplo de una aplicación para calcular la factura de un cliente de servicio celular dependiendo de su uso. Lo siguiente calcula "Factura" según "Uso" con las siguientes reglas aplicables. Si 'Bill' es más de $ 100, se otorga un 10% de descuento [50].

```
public static double calculateBill (int Usage)
{
 double Bill = 0;
 if(Usage > 0)
 {
        Bill = 40;
 }
 if(Usage > 100)
 {
        if(Usage <= 200)
        {
        Bill = Bill + (Usage - 100) * 0.5;
        }
        else
        {
        Bill = Bill + 50 + (Usage - 200) * 0.1;
                if(Bill >= 100)
                {
                        Bill = Bill * 0.9;
                }
        }
 }
 return Bill;
}
```

Fig. 11. Ejemplo de prueba de flujo estático *[50]*

Para la variable 'Usage', los patrones definir-use-kill son:

a. ~ definir: caso normal
b. definir-uso: caso normal
c. uso-uso: caso normal
d. usar-matar: caso normal

Para la variable 'Bill', los patrones definir-use-kill son:

- ~ definir: caso normal
- definir-definir: sospechoso
- definir-uso: caso normal
- use-define: aceptable
- uso-uso: caso normal
- usar-matar: caso normal

Diagrama de flujo de control del ejemplo dado

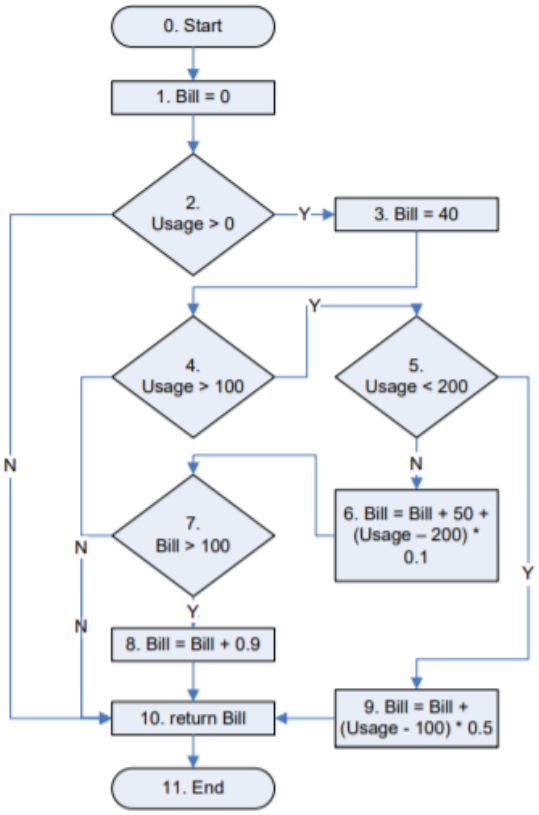

Fig. 12. Diagrama de flujo de control de prueba de flujo de datos *[50]*

Diagrama de flujo de control para “Bill”

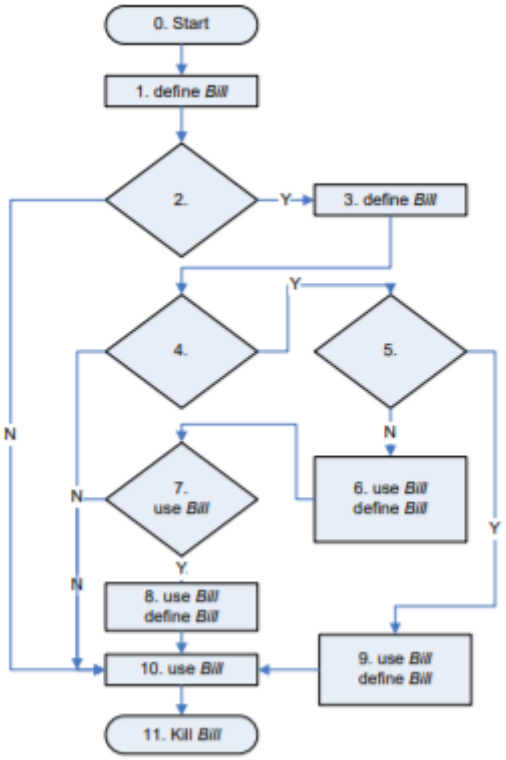

Fig. 13. Diagrama de flujo de control para el ejemplo "Bill" *[50]*

Análisis estático para la variable "Bill"

TABLA II. ANÁLISIS PARA LA VARIABLE BILL *[50]*

Anomaly		Explanation
~d	0-1	Allowed. Normal case
dd	0-1-2-3	Potential bug. Double definition.
du	3-4-5-6	Allowed. Normal case.
ud	6	Allowed. Data is used and then redefined.
uk	10-11	Allowed.
dd	1-2-3	Potential bug. Double definition.
uu	7-8	Allowed. Normal case.
k~	11	Allowed. Normal case.

Diagrama de flujo de control para "Usage"

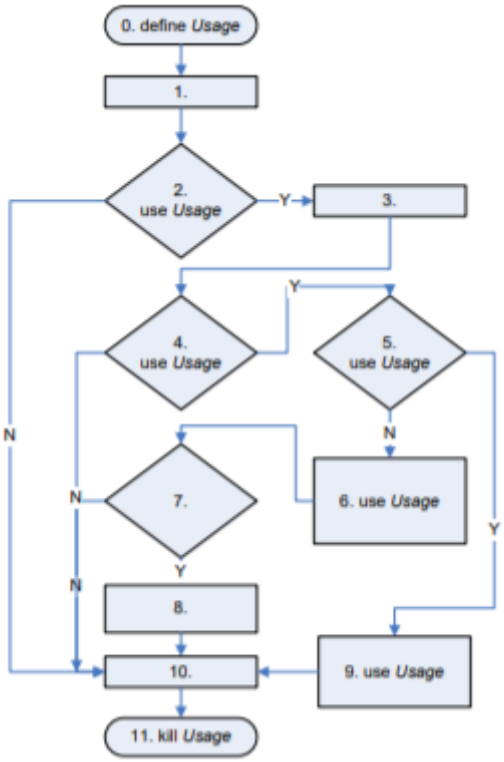

Fig. 14. Diagrama de flujo de control para el ejemplo "Usage" *[50]*

Análisis estático para la variable "Usage"

TABLA III. ANÁLISIS PARA LA VARIABLE "USAGE" *[50]*

Anomaly		Explanation
~d	0	Allowed.
du	0-1-2	Allowed. Normal case.
uk	9-10-11	Allowed.
uu	5-6	Allowed. Normal case.
k~	11	Allowed. Normal case.

1.4.7. Heurística

La Heurística "Es una técnica de búsqueda directa que utiliza reglas favorables para localizar soluciones mejoradas. Las técnicas Heurísticas se utilizan para resolver problemas de optimización que de por si son complejos de resolver con técnicas de optimización tradicionales" [54].

¿Cuándo se emplea Heurística?

a. Cuando el número de soluciones es enorme
b. El problema es tan complicado que se deben utilizar métodos complejos.
c. Las soluciones posibles están altamente restringidas, lo cual dificulta la generación de una solución factible

En las disciplinas científicas se suele emplear el método heurístico a fin de alcanzar el mejor resultado ante un problema en específico.

Algoritmos heurísticos

Algoritmo CS (Cuckoo search)

El algoritmo Cuckoo search (CS) fue desarrollado por Xin-she Yang and Suash Deb in 2009. Lo cual fue inspirado en la forma en que algunas especies de cuco depositan sus huevos en los nidos de otras aves hospedantes (de otras especies), para ser criados. Cada huevo en el nido representa una solución, y los huevos de cuco representan nuevas soluciones. El objetivo es utilizar las soluciones nuevas y potencialmente mejores (cucos) para reemplazar las soluciones menos adecuadas en los nidos. En cada iteración, un huevo de cuco se coloca al azar en un nido seleccionado; Los nidos con huevos de alta calidad se trasladarán a la generación de nidos; Luego, en los nidos menos adecuados restantes, se realiza una operación de descubrimiento por parte de las aves anfitrionas, recuperando aleatoriamente los huevos de cuco y descartándolos, por lo tanto, ignorándolos de otros cálculos. En la figura 15 se presenta un diagrama de flujo genérico [55].

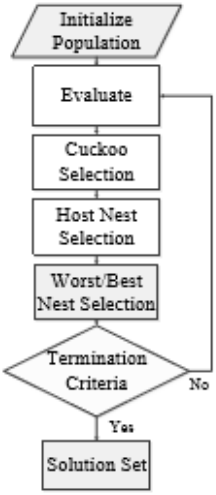

Fig. 15. Diagrama de flujo para el algoritmo CS *[55]*

Algoritmo ABC (Artificial Bee Colony)

Nos dicen que el algoritmo ABC fue propuesto por Karaboga en el 2005. Fue inspirado por la forma inteligente en que los enjambres de abejas ubican y aprovechan su comida. El espacio de solución candidato en ese caso está representado por lugares de fuentes potenciales de alimentos. La colonia de abejas, dividida en abejas empleadas, curiosas y exploradoras, se extiende a través de ella. las abejas empleadas apuntan y explotan la posición potencial de los alimentos e informan a los espectadores de más sitios potenciales de alimentos. Las abejas empleadas intentan determinar el potencial alimentario de esas posiciones, es decir, si son las buenas soluciones adecuadas. Si la posición de una abeja empleada no representa una buena solución, entonces la

abeja se convierte en exploradora y comienza a explorar el espacio de la solución. En la figura 16 se presenta un diagrama de flujo genérico [55].

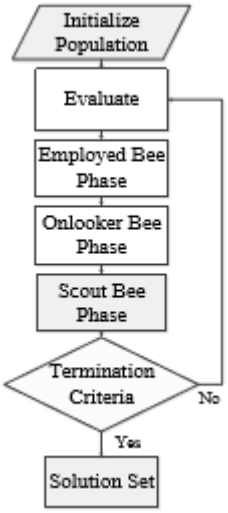

Fig. 16. Diagrama de flujo para el algoritmo ABC *[55]*

Algoritmo PSO (Particle Swarm Optimization)

Se les atribuye originalmente a los investigadores Kennedy y Eberhart en 1995. En el algoritmo PSO, una solución se representa como una partícula, y a la población de soluciones se le llama un enjambre de partículas. Cada partícula tiene dos propiedades principales: posición y velocidad. Y cada una de esta partícula se mueve a una nueva posición usando la velocidad. Una vez que se alcanza una nueva posición, la mejor posición de cada partícula y la mejor posición del enjambre se actualizan según sea necesario. La velocidad de cada partícula se ajusta según las experiencias de la partícula. El proceso se repite hasta que se cumple un criterio de parada [56].

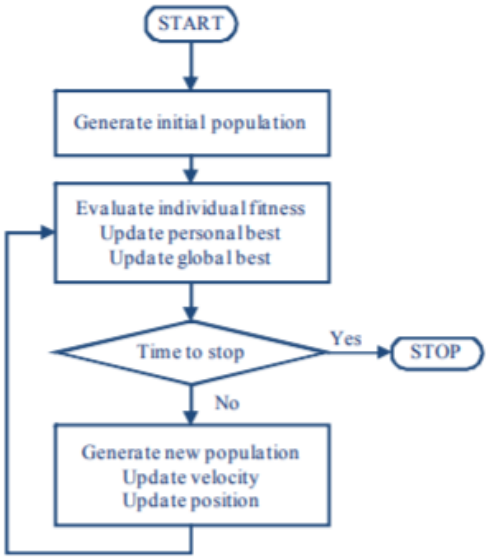

Fig. 17. Diagrama de flujo para Algoritmos PSO *[56]*

Algoritmos GA (Genetic algorithms)

Los algoritmos genéticos (GA) y sus conceptos básicos fueron desarrollados por John Holland en 1975. El algoritmo genético se aplica a diferentes tipos de problemas, principalmente problemas de búsqueda y optimización. Este algoritmo comprende los 3 procesos genéticos fundamentales, como selección, cruce y mutación. Como ejemplo estos autores aplicaron el esquema de selección de la ruleta, donde los padres se combinan aleatoriamente para criar descendientes que tienen combinaciones de sus cromosomas. Además, se lleva a cabo un proceso de mutación, en el que aleatoriamente se alteran varias partes de los cromosomas de la descendencia. Finalmente, los mejores entre ambos padres y descendientes se eligen para constituir la próxima generación de cromosomas, como se muestra a continuación [55].

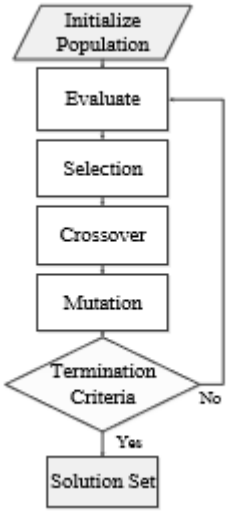

Fig. 18. Diagrama de flujos de Algoritmo Genético *[55]*

Algoritmo DE (Differential Evolution)

La evolución diferencial (DE), es una rama de la programación evolutiva, es un algoritmo de optimización global basado en la población, que fue desarrollado por Rainer Storn y Kenneth Price en 1995. DE utiliza un operador diferencial evolutivo para convertir las soluciones aleatorias iniciales en soluciones óptimas. A diferencia de los algoritmos genéticos (GA). Cabe decir, que DE es uno de los mejores algoritmos de tipo genético para resolver problemas con cromosomas de valor real. Además, la evolución diferencial utiliza la mutación como un mecanismo de búsqueda y la selección para dirigir la búsqueda [9]. La mayor diferencia entre los algoritmos genéticos y la evolución diferencial, es que los algoritmos genéticos se basan en el crossover, un mecanismo de intercambio de información probabilístico y útil entre soluciones para encontrar mejores soluciones, mientras que la evolución diferencial utiliza la mutación como mecanismo principal de búsqueda [57].

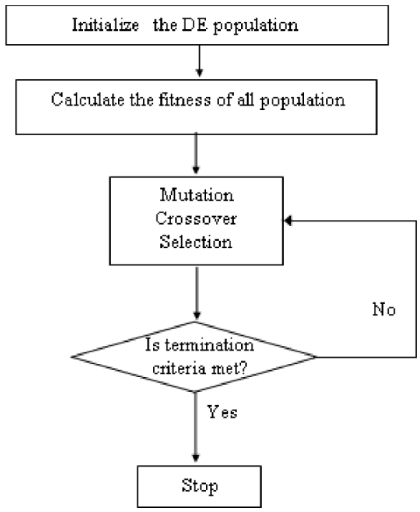

Fig. 19. Diagrama de flujo de la Evolución Diferencial *[58]*

Algoritmo NSGA-II (Elitist Non-Dominated Sorting Genetic Algorithm)

Con respecto al algoritmo NSGA-II fue propuesto por Deb y sus estudiantes en el año 2000. En este algoritmo inicialmente se crea una población (ya se aleatoria o mediante una técnica de inicialización). Esta población se ordena de acuerdo a los niveles de no dominancia y a cada solución se le asigna una función fitness de acuerdo a su nivel de no dominancia (el mejor nivel es 1) y se entiende que durante el proceso dicha función debe disminuir. la selección por torneo, cruzamiento y mutación son utilizados para crear la población de descendientes [59]

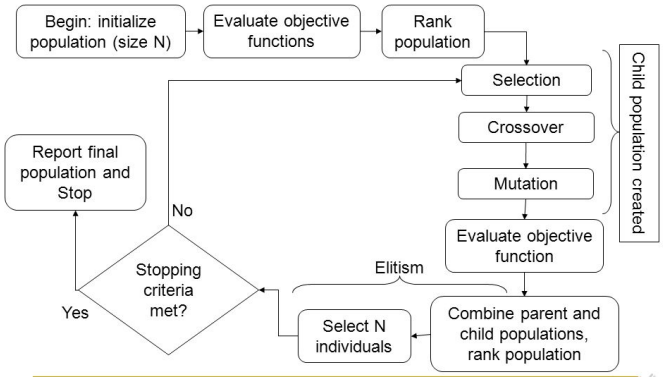

Fig. 20. Diagrama de flujo de NSGA-II *[60]*

Algoritmo MOALO (Multi-objective ant lion optimizer)

Acerca del algoritmo MOALO fue propuesto por Mirjalili. El algoritmo ALO es un nuevo algoritmo de búsqueda estocástica desarrollado, que imita el mecanismo de caza de los leones en la naturaleza. En este nuevo enfoque, se propone que las hormigas y los hormigueros, como agentes de búsqueda, encuentren soluciones a pasos de cazar presas, que incluyen el paseo aleatorio de las hormigas, la construcción de trampas, el atrapamiento de hormigas en las trampas, la captura de presas y la reconstrucción de las trampas. Además, en el algoritmo ALO, las actualizaciones de posición de las hormigas dependen de las caminatas aleatorias alrededor de la hormiga seleccionada por la ruleta y la élite, y la mejor partícula se conserva al establecer la élite

en el proceso de búsqueda. Esto hace que ALO tenga las ventajas de una velocidad de cálculo rápida, alta eficiencia y buena convergencia [61]**.**

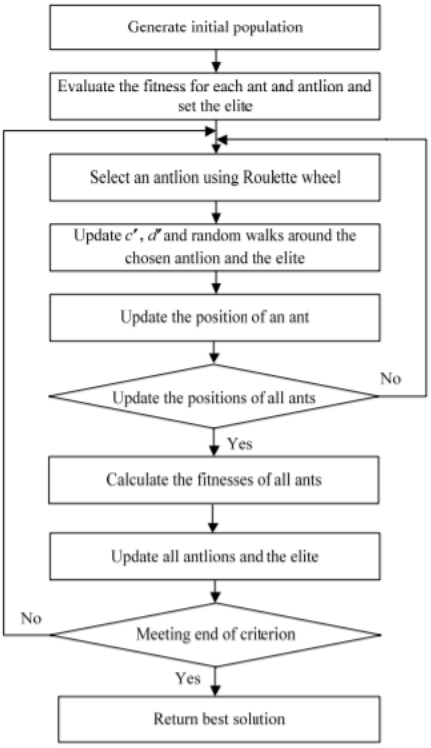

Fig. 21. Diagrama de flujo de MOALO *[61]*

Algoritmo ACO (Ant colony optimization)

El algoritmo ACO fue propuesto por primera vez por Dorigo en 1992, lo cual fue inspirado en el comportamiento de forrajeo de las colonias de hormigas en la naturaleza. Las hormigas cuando buscan su comida dejan un nivel químico llamado feromona en los caminos recorridos para coordinar con las otras hormigas. Al detectar el nivel de feromonas, las hormigas seleccionan el camino cuando llegan a la bifurcación en el camino. Al inicio, las hormigas seleccionan caminos al azar con dirección hacia los alimentos. Después de algunos recorridos, el nivel de feromonas del camino más corto llega a ser más alto que los otros porque en los caminos más largos la evaporación de feromonas es mayor. Para aplicar el algoritmo ACO, tenemos que construir los problemas mediante un gráfico ponderado. después, un conjunto de hormigas artificiales empezara a encontrar una solución en el gráfico al evaluar la feromona, que es un conjunto de parámetros relacionados con el gráfico. Las hormigas construyen de forma iterativa la solución en un proceso estocástico a partir de la preferencia de las feromonas y el conocimiento heurístico sobre el camino. Cada vez que una hormiga recorre el camino, se actualiza la feromona y el valor heurístico [62].

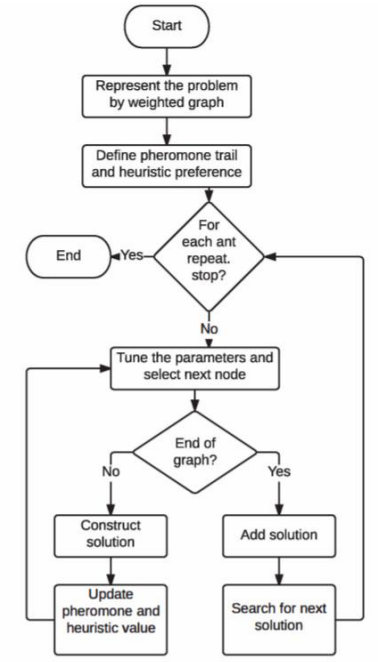

Fig. 22. Flujo de proceso genérico del algoritmo ACO *[62]*

2. Método de Investigación

El procedimiento de revisión del material bibliográfico del presente proyecto tiene como objetivo proveer un marco para llevar a cabo el desarrollo de la revisión bibliográfica de un tema específico de forma concisa, válida y justificable. Teniendo en cuenta lo antes mencionado, se desarrolla el proyecto centrándose en la optimización de pruebas de software estructurales. Se debe agregar que se ha optado por estudiar un rango de 16 años entre el 2002 y 2018, dado que son pruebas de software. A continuación se realiza una descripción del método mencionado en la presente figura.

Fig. 23 Método de investigación
Fuente: Elaboracion propia

2.1 Plan de la investigación

En esta etapa se realiza el procedimiento de revisión bibliográfica, en el cual, se describe en los siguientes 2 pasos:

- Interrogantes de investigación que el estudio responderá
- Protocolo de revisión que se seguirá para la búsqueda de la información

2.1.1 Interrogantes de la investigación

En este punto, se plantean las interrogantes con la finalidad de responder el estudio que se describe a continuación.

Pregunta 1. ¿Cuánto material bibliográfico relacionado a la optimización de pruebas de software estructurales se ha publicado entre el año 2002 y 2018?

Pregunta 2. ¿Cuáles son los algoritmos heurísticos que se utilizan para la optimización de pruebas de software?

Pregunta 3. ¿Qué tipo de problemas presenta el estado actual de la optimización de las pruebas de software?

Pregunta 4. ¿Se muestran etapas claras para realizar la optimización de pruebas de software?

2.1.2 Protocolo de revisión

En el desarrollo de la presente investigación se utilizó la plataforma Scimago Journal & Country Rank en donde se investigó el ranked de journals, en la opción de área se seleccionó Engineering y en la categoria de temas se seleccionó Control and System Engineering, dado que la presente revisión está enfocada a la ingeniería de software. Luego de realizar el proceso de filtración se seleccionó la base de datos IEEE como base de datos principal, debido que la mayoría de artículos relacionados con el tema se encuentra allí. La revisión del material encontrado en esta base de datos comprende desde el año 2002-2018.

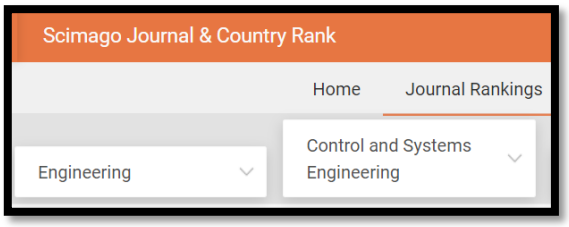

Fig. 24 Selección de campos en el ranked de journals
Fuente: Elaboracion propia

15	IEEE Transactions on Control Systems Technology	journal	1.811 Q1	136	430	644	6748	4262	639	5.96	15.69
16	Systems and Control Letters	journal	1.762 Q1	122	130	431	3315	1543	425	3.24	25.50
17	International Journal of Robust and Nonlinear Control	journal	1.716 Q1	91	368	695	13507	3295	686	4.47	36.70
18	IEEE/ASME Transactions on Mechatronics	journal	1.686 Q1	113	286	875	9689	5180	865	5.71	33.88
19	IEEE Transactions on Industrial Informatics	journal	1.678 Q1	100	729	727	18020	6348	715	8.48	24.72

Fig. 25 Selección de base de datos IEEE
Fuente: Elaboracion propia

2.2 Procedimiento de la investigación

2.2.1 Identificar las investigaciones relevantes

En este punto se buscan las propuesta más relevantes para responder las interrogantes planteadas. Cabe mencionar que, la propuesta de revisión se centra estrictamente en la optimización de pruebas de software estructurales es por ello que se ha diseñado una regla genérica de búsqueda utilizando "and" booleano para unir los principales términos "software testing, data flow testing, y genetic algorithm" como se muestra en la presente Tabla IV. Además, se ha revisado la base de datos IEEE, finalmente se determinó que revistas serán la fuente principal de esta revisión bibliográfica para garantizar la calidad de este estudio.

TABLA IV. BUSQUEDA DE DATOS Y RESULTADOS

Nº	Base de datos	Regla de Búsqueda	Resultados
1	IEEE Xplore Digital Library	('software testing AND data flow testing AND genetic algorithm')	55

Fuente: Elaboracion propia

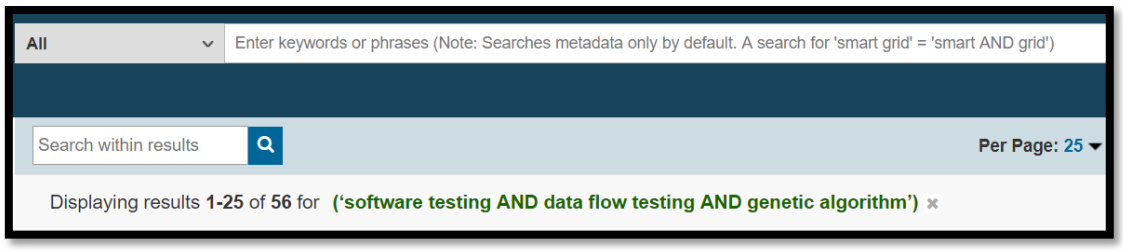

Fig. 26 Resultado de la regla de búsqueda en la base de datos IEEE
Fuente: Elaboracion propia

2.2.2 Sintetizar los datos

Se realizó un análisis del contenido de las palabras clave de cada artículo haciendo énfasis en el significado de cada palabra o concepto el cual va más allá de simplemente contar palabras para examinarlas. Se debe aclarar que para realizar este análisis se basa en la valoración de los términos controlados por INSPEC4 la cual es una importante base de datos de indexación de

literatura científica y técnica, publicada por el Instituto de Ingeniería y Tecnología. El resultado de esta evaluación se muestra en la siguiente Tabla V, la cual expone el título de la publicación, autor, año de la publicación, palabras clave del autor, términos controlados por IEEE, términos controlados y no controlados por INSPEC número de citas y referencias de cada publicación. Cabe mencionar que solo se encuentran los artículos enfocados a la optimización de pruebas de software.

TABLA V. VALORACION DE LAS PALBRAS CLAVES

Nº	Título de documento	Autores	Año	Palabras clave del autor	Términos de IEEE	Términos controlados por INSPEC	Términos no controlados por INSPEC	# de citas	# de Ref.
1	A concept of out degree in CFG for optimal test data using genetic algorithm [11]	Shadab Irfan ; Prabhat Ranjan	2012	Path Testing; Genetic Algorithm ; CFG; Out degree	Testing,Genetic algorithms,Flow graphs,Biological cells,Software,Information technology,Complexity theory	automatic test pattern generation; computational complexity; genetic algorithms; program testing	CFG out degree,optimal test data,genetic algorithm,testing phase,complex task,test data generation,program source code,control flow graph,CFG outdegree,fitness function,complexity reduction	1	11
2	A critical review of Artificial Bee Colony optimizing technique in software testing [12]	Disha Garg ; Abhishek Singhal	2016	software testing; genetic algorithm; software under test; flow diagram; artificial bee colony technique.	Software, Software testing,Optimization,Technological innovation,Computer security,Data mining	optimisation, program debugging,program testing,software engineering	artificial bee colony optimizing technique,software testing,software development life cycle,testing tools,manual testing,automatic testing,bug removal,meta-heuristic techniques,software code,SDLC	0	16

3	A differential evolution based approach to generate test data for data-flow coverage [9]	Sapna Varshney ; Monica Mehrotra	2016	Differential Evolution, Data Flow Testing, Dominator Tree, Search Based Software Testing	Genetic algorithms,Algorithm design and analysis,Software testing,Evolutionary computation,Measurement,Software algorithms	data flow analysis,evolutionary computation, program testing	differential evolution based approach,software testing efficacy,software development process,search-based evolutionary algorithm,automated test data generation,genetic algorithm comparison,highly-adaptive evolutionary algorithm,particle swarm optimization comparison, PSO ,data-flow coverage test adequacy criterion,dominance relations,branch distance metric,random search comparison	1	25
4	A genetic algorithm based approach for prioritization of test case scenarios in static testing [14]	Sangeeta Sabharwal ; Ritu Sibal ; Chayanika Sharma	2011	software testing,genetic algorithm, Information flow metric,CFG	Testing,Biological cells,Genetic algorithms,Complexity theory,Computers, Measurement,Communications technology	flow graphs,genetic algorithms,program testing,software metrics	genetic algorithm,test case scenario prioritization,white box testing,program code,code structure,internal design flow,source code,structural testing,static testing efficiency optimisation, information	3	18

							flow metric,information flow complexity,control flow graph		
5	A new method of test data generation for branch coverage in software testing based on EPDG and Genetic Algorithm [15]	Ciyong Chen ; Xiaofeng Xu ; Yan Chen ; Xiaochao Li ; Donghui Guo	2009	Test Data Generation,Edge Partitions, Genetic Algorithm ,Branch Coverage	Software testing,Genetic algorithms,Tree graphs,Instruments, Partitioning algorithms,Costs,Automatic testing,Physics,Information science,Flow graphs	flow graphs,genetic algorithms,graph theory,program testing	test data generation method,branch coverage software testing,edge partitions dominator graph,genetic algorithm,critical branch,control flow graph,fitness function,pre-dominator tree	1	11
6	A path and branch based approach to fitness computation for program test data generation using genetic algorithm [16]	Ankur Pachauri ; Gursaran ; Gaurav Mishra	2015	program test data generation,search-based software testing,genetic algorithm	Genetic algorithms,Sociology,Statistics,Market research, Knowledge management,Search problems, Flow graphs	genetic algorithms,program testing,search problems	path-and-branch based approach,fitness computation, program test data generation,genetic algorithm,target node sequence,actual execution path,branch information, path information,search performance, control flow graph	0	12
7	A Survey on Software Testing Techniques using Genetic Algorithm [17]	Chayanika Sharma, Sangeeta Sabharwal, Ritu Sibal	2013	Software testing, Genetic Algorithm	Software testing, Genetic Algorithm	Software testing, Genetic Algorithm	Software testing, Genetic Algorithm	1	7

8	A tool for data flow testing using evolutionary approaches (ETODF) [18]	Shaukat Ali Khan ; Aamer Nadeem	2014	Evolutionary Algorithm,Test Data,Data Flow Testing,Mutation,Crossover	Software testing,Genetic algorithms,Software,Semantics,Flow graphs,TV	genetic algorithms,program testing	data flow testing,evolutionary approach,software testing,software development lifecycle	1	25
9	An Automatic Software Test-Data Generation Scheme Based on Data Flow Criteria and Genetic Algorithms [63]	Andreas S. Andreou ; Kypros A. Economides ; Anastasis A. Sofokleous	2007	-	Automatic testing,Software testing,Genetic algorithms,Flow graphs,Data flow computing,System testing,Computer science,Information technology,Automatic generation control,Algorithm design and analysis	automatic programming,data flow graphs,genetic algorithms,program testing	automatic software test-data generation scheme,data flow coverage criteria,genetic algorithms	3	17
10	An Experimental Tool for Search-based Mutation Testing [20]	Muhammad Bilal Bashir ; Aamer Nadeem	2018	search-based mutation testing,object-oriented paradigm,equivalent mutant,control flow	Tools,Testing,Genetic algorithms,Java,Instruments,Automation,Generators	genetic algorithms,Java,program testing,search problems,software tools	automatic test case generation,software testing,random testing,genetic algorithm,Java programs	0	43
11	Applying Ant Colony Optimization in Software Testing to Generate	Sumon Biswas ; M. S. Kaiser ;	2015	Software testing,Ant Colony Optimization (ACO),Path	Testing	ant colony optimisation, artificial immune systems,prog	ant colony optimization, software testing,priorit	4	28

	Prioritized Optimal Path and Test Data [21]	S. A. Mamun		testing,Test data generation,Control Flow Graph (CFG)		ram testing,software engineering	ized optimal path,test data		
12	Approach for Automated Test Data Generation for Path Testing in Aspect-Oriented Programs using Genetic Algorithm [22]	Juhi Khandelwal ; Pradeep Tomar	2015	Genetic Algorithm,Automated Test Data Generation,Path Testing,Aspect-Oriented Programming	Genetic algorithms,Software,Java,Software testing,Automation, Flow graphs	aspect-oriented programming,automatic test software,flow graphs,genetic algorithms,program testing	path testing,aspect-oriented programming,AOP,programming paradigm,cross-cutting requirements	0	15
13	Automated Test Path Generation using Genetic Algorithm [23]	Niveth Vijay K	2017	Software testing, Test path generation, Control flow graph (CFG), Genetic algorithm	Software testing, Test path generation, Control flow graph (CFG), Genetic algorithm	Software testing, Test path generation, Control flow graph (CFG), Genetic algorithm	Software testing, Test path generation, Control flow graph (CFG), Genetic algorithm	1	11
14	Automatic Path-oriented Test Data Generation Using a Multi-population Genetic Algorithm [24]	Yong Chen ; Yong Zhong	2008	-	Automatic testing,Genetic algorithms,Software testing,Costs,Input variables	flow graphs,genetic algorithms,program testing	automatic path-oriented test data generation,multipopulation genetic algorithm,MPGA algorithm	8	25
15	Automatic test data generation based on multi-objective ant lion optimization algorithm [25]	Mayank Singh ; Viranjay M. Srivastava ; Kumar Gaurav ; P. K. Gupta	2018	Automated Software testing,Automated Test Data Generation,Control flow graph	Optimization,Software,Software algorithms,Software testing,Object	genetic algorithms,program testing,search problems,software engineering	novel automated test data generation method,complete path,structural testing	0	15

					oriented modeling				
16	Automatic Test Data Generation Model by Combining Dataflow Analysis with Genetic Algorithm [26]	Mingjie Deng ; Rong Chen ; Zhenjun Du	2009	automatic test,dataflow testing,CFG,genetic algorithm, data type graph	Automatic testing,Data analysis, Algorithm design and analysis, Genetic algorithms	data flow analysis,data flow graphs,embedded systems,genetic algorithms	dataflow analysis,genetic algorithm,object-oriented software,automatic test data generation model	4	6
17	Batch-Optimistic Test-Cases Generation Using Genetic Algorithms [27]	Anastasis A. Sofokleous ; Andreas S. Andreou	2008	-	Genetic algorithms,Automatic testing,System testing,Software testing,Flow graphs	flow graphs,genetic algorithms,program diagnostics,program testing,source coding	dynamic software testing framework,program source code,control flow graphs	1	22
18	Introduction to Data Flow Testing with Genetic Algorithm [28]	Rijwan Khan	2017	Data-Flow Testing, Control-Flow Graph, Genetic Algorithms, Software Testing, Automatic Test Cases	Data-Flow Testing, Control-Flow Graph, Genetic Algorithms, Software Testing, Automatic Test Cases	Data-Flow Testing, Control-Flow Graph, Genetic Algorithms, Software Testing, Automatic Test Cases	Data-Flow Testing, Control-Flow Graph, Genetic Algorithms, Software Testing, Automatic Test Cases	1	8
19	Evolutionary approach to generating test data for data flow test [64]	Shujuan Jiang ; Jieqiong Chen	2018	-	-	genetic algorithms,program testing	genetic algorithm,definition-use coverage,definition-use pairs,authors, test data generation	-	-
20	Generation of automatic test cases with mutation analysis and hybrid	Rijwan Khan ; Mohd Amjad ; Akhilesh Kumar Srivastava	2017	software testing,Hybrid Genetic Algorithm (HGA),Genetic Algorithm	Genetic algorithms,Software,Software testing,Hybrid power	automatic testing,data flow analysis,genetic algorithms,program testing	automatic test case generation,hybrid genetic algorithm,mutation analysis	0	17

	genetic algorithm [65]			,testing efficiency	systems,Conferences				
21	Identification of potentially infeasible program paths by monitoring the search for test data [66]	P.M.S. Bueno ; M. Jino	2002	-	Monitoring,Software testing,Electronic mail,Genetic algorithms,Automation,Automatic testing	program testing,genetic algorithms,search problems,data flow analysis,software tools	potentially infeasible program paths,test data search,software tool,test data generation	6	27
22	It Does Matter How You Normalise the Branch Distance in Search Based Software Testing [67]	Andrea Arcuri	2010	Branch Distance, Search Based Software Testing,Theory,Simulated Annealing,Genetic Algorithms	Software testing,Software algorithms,Flow graphs,Simulated annealing, Genetic algorithms	genetic algorithms,program testing,simulated annealing,tree searching	branch distance,search based software testing,search algorithm,test data generation	18	26
23	Optimizing test case generation in glass box testing using Bio-inspired Algorithms [68]	Sunny S. Bodiwala ; Devesh C. Jinwala	2016	Software engineering,test case,Software testing,Bio-inspired techniques,automated testing	Genetic algorithms,Optimization,Sociology,Statistics,Microorganisms,Complexity theory	genetic algorithms,graph theory,program debugging,program testing,software quality	test case generation optimization, glass box testing,bioinspired algorithm,software testing	0	22
24	Prioritization of test case scenarios derived from activity diagram using genetic algorithm [69]	Sangeeta Sabharwal ; Ritu Sibal ; Chayanika Sharma	2010	software testing,genetic algorithm, activity diagram, CFG	Biological cells,Gallium,Unified modeling language, Software testing,Software	genetic algorithms,program testing	software testing,program error,critical path cluster,test case scenario,activity diagram	2	17
25	Research on test data generation based on		2010	Software testing,Test	Gallium,Software algorithm	genetic algorithms,program	test data generation,simulated		

	Modified Genetic and Simulated Annealing Algorithm [70]	Li-Yun Yu ; Lu Lu		data,Genetic algorithm, Simulated Annealing Algorithm	s,Software testing,Convergence,Flow graphs,Simulated annealing	testing,simulated annealing	annealing algorithm,genetic algorithm,software testing procedure	0	10
26	Search-Based Testing Guidance Using Dominances vs. Control Dependencies [71]	Ahmed S. Ghiduk	2009	search-based testing,genetic algorithms,test-data generation,dominance,control dependecnies	Software engineering,Genetic algorithms,Switches,Software testing,Application software	optimisation, program testing,search problems,software engineering	search-based testing guidance,dominances,control dependencies	0	13
27	Tabu Search and Genetic Algorithm to Generate Test Data for BPEL Program [72]	Bo Yu ; Ye-mei Qin ; Ge Yao ; Chang Gong	2009	-	Genetic algorithms,Web services,Logic testing,Automatic testing,Service oriented architecture	business data processing,genetic algorithms,search problems,Web services	tabu search,genetic algorithm,business process execution language program,Web services	1	19
28	Test Data Generation from UML State Machine Diagrams using Gas [73]	Chartchai Doungsa-ard ; Keshav Dahal	2007	-	Unified modeling language, Software testing,Automatic testing,Genetic algorithms,System testing	finite state machines,genetic algorithms,program testing,Unified Modeling Language	automatic test data generation,UML state machine diagram,Unified Modeling Language	9	21
29	Using a Genetic Algorithm and Formal Concept Analysis to Generate Branch Coverage Test Data	S. Khor ; P. Grogono	2004	-	Genetic algorithms,Algorithm design and analysis, Automatic testing,Instruments	automatic test pattern generation,data flow analysis,formal specification, genetic algorithms	genetic algorithm,formal concept analysis,automatic branch coverage test data generation	4	17

	Automatically [74]								
30	Using Artificial Bee Colony for Code Coverage Based Test Suite Prioritization [75]	Patipat Konsaard ; Lachana Ramingwong	2016		Software algorithms,Optimization,Software testing,Standards,Software,Genetic algorithms	fault diagnosis,optimisation,program testing	artificial bee colony,code coverage based test suite prioritization ,fault detection,code coverage rate	3	31
31	Using Genetic Algorithms to Aid Test-Data Generation for Data-Flow Coverage [76]	Ahmed S. Ghiduk ; Mary Jean Harrold	2008		Genetic algorithms,Automatic testing,Software testing,Educational institutions,Automatic contro	automatic testing,data flow analysis,genetic algorithms,program testing	genetic algorithm,automatic test-data generation technique,data-flow coverage criteria	19	18
32	Using program data-state diversity in test data search [77]	M. Alshraideh ; L. Bottaci	2006		Automatic testing,Instruments, Cost function,Software testing,Computer science	data flow analysis,optimising compilers,program control structures,program testing	program data-state diversity,search-based automatic software test data generation	0	19

2.3 Documentación dela investigación

El objetivo de la fase 3 es documentar las respuestas a las preguntas de investigación presentadas en el paso 1, así como validar el informe.

2.3.1. Validar el informe

Los datos y el análisis de esta revisión integradora fueron validados en todo momento por los autores de este documento como parte del proceso de revisión

3. Resultados.

La interpretación de los resultados obtenidos se basa en la revisión bibliográfica de los artículos relacionados a la optimización de pruebas de software estructurales utilizando algoritmo genético.

Es importante señalar que los trabajos seleccionados otorgan datos puramente relacionados a la optimización de pruebas de software estructurales utilizando algoritmos heurísticos (es decir, si las propuestas realizan la optimización de las pruebas de software utilizando alguna técnica heurística o algún algoritmo heurístico relacionados a la optimización de los casos de prueba).

3.1 Análisis.

En esta sección se presentan los resultados de la análisis respondiendo a las preguntas de la investigación planteadas anteriormente en **interrogantes de la investigación**. Cabe decir que la interpretación de los resultados obtenidos se basa en el material bibliográfico de los artículos relacionados a la optimización de pruebas de software mediante el uso de técnicas y algoritmos heurísticos ubicados en la Tabla V.

Pregunta 1:

¿Cuánto material bibliográfico relacionado a la optimización de pruebas de software estructurales se ha publicado entre el año 2002 y 2018?

La cantidad de artículos científicos relacionados a la optimización de pruebas de software es muy variada. A continuación, se muestra en la Figura 27 la cantidad de estudios relacionados al tema antes mencionado ordenados de forma ascendente por año de publicación utilizando la regla de búsqueda ubicado en la Tabla IV. Pero antes de ello, primero se realiza una matriz de las cantidad de artículos relacionados a optimización de pruebas de software por año.

TABLA VI Matriz de artículos por año

Año	Cantidad
2002	1
2004	1
2006	1
2007	2
2008	3
2009	4
2010	3
2011	1
2012	1
2013	1
2014	1
2015	3
2016	4
2017	3
2018	3

Fuente: Elaboracion propia

A continuación se muestran los datos estadísticos de la matriz resultante en la figura 27.

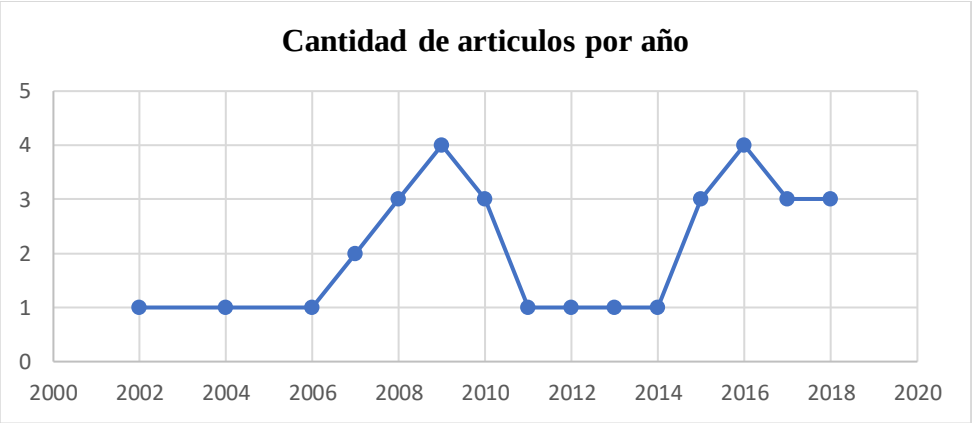

Fig. 27 Cantidad de artículos por año
Fuente: Elaboracion propia

Como se puede se puede observar hay un leve crecimiento entre los años 2006-2009 y 2014-2016, a excepción de los años 2002-2004 y 2010-2014 en los cuales muestran un desfice en el aporte de las publicaciones relacionadas. Así mismo, se demuestra un crecimiento considerable en los últimos años (2018), lo cual demuestra que la optimización de pruebas de software se está convirtiendo en un materia de investigación para diversos autores en la actualidad.

Pregunta 2.

¿Cuáles son los algoritmos heurísticos que se utilizan para la optimización de pruebas de software?

Para esta pregunta se realizó un análisis del material bibliográfico, donde los autores redactan que algoritmos heurísticos utilizan para la optimización de pruebas de software. Además, después de realizar dicho análisis, se genera una matriz para ver cuál es el algoritmo que más utilizan en la mayoría de artículos.

TABLA VII. MATRIZ DE ALGORITMO HEURISTICOS

Nº	Autor (s)	Técnicas o algoritmos Heurísticos
1	Irfan, Shadab	Algoritmo Genético
2	Garg, Disha	Algoritmo Genético, algoritmo de colonias de abejas artificiales y algoritmo de colonias de hormigas.
3	Varshney, Sapna; Mehrotra, Monica	Algoritmo Genético, Algoritmo Optimización de enjambre de partículas, Evolución diferencial
4	Sabharwal, Sangeeta; Sibal, RituSharma; Chayanika Sharma Department	Algoritmo Genético
5	Chen, Ciyong; Xu, Xiaofeng; Chen, Yan	Algoritmo Genético
6	Sharma, Chayanika; Sabharwal, Sangeeta; Sibal, Ritu	Algoritmo Genético
7	Khan, Shaukat Ali	Algoritmo Genético
8	Andreou, Andreas S	Algoritmo Genético
9	Bashir, Muhammad Bilal	Algoritmo Genético

10	Biswas, Sumon	Algoritmo de colonias de abejas artificiales, algoritmo de colonias de hormigas, Sistema inmune artificial, Optimización de Enjambre de partículas
11	Khandelwal, Juhi	Algoritmo Genético
12	Rauf, Abdul	Algoritmo Genético
13	K, Niveth Vijay	Algoritmo Genético
14	Chen, Yong; Zhong, Yong	Algoritmo genético multi-población, algoritmo genético
15	Singh, Mayank; Srivastava, Viranjay M; Gaurav, Kumar	Optimización Multi-Objetivo hormiga león
16	Deng, Mingjie; Chen, Rong; Du, Zhenjun	Algoritmo Genético
17	Sofokleous, Anastasis A; Andreou, Andreas S	Algoritmo Genético
18	Khan, Rijwan; Amjad, Mohd	Algoritmo Genético
19	Khan, Rijwan; Amjad, Mohd; Srivastava, Akhilesh Kumar	Algoritmo genético híbrido
20	Marcos, Paulo; Bueno, Siqueira; Jino, Mario	Algoritmo Genético
21	Bodiwala, Sunny S; Devesh, C	Algoritmo genético, algoritmo de luciérnaga y algoritmos de forrajeo bacteriano
22	Sabharwal, Sangeeta; Sibal, Ritu; Sharma, Chayanika	Algoritmo Genético
23	Yu, Li-yun; Lu, Lu	Algoritmo genético modificado y de recocido simulado
24	Bo, Yu; Yao, Qin Ye-mei; Gong, Ge	búsqueda tabú y algoritmo genético
25	Khor, Susan; Grogono, Peter	Algoritmo Genético
26	Konsaard, Patipat	Algoritmo de colonia de abeja artificial
27	Ghiduk, Ahmed S; Harrold, Mary Jean	Algoritmo Genético

Fuente: Elaboración propia

Como se puede observar en el análisis de la tabla anterior, demuestra que en muchos artículos los autores proponen diferentes algoritmo heurísticos o evolutivos, además de técnicas para la optimización de pruebas de software.

Posterior a ello se obtiene la siguiente matriz resultante.

TABLA VIII. MATRIZ RESULTANTE DE LOS ALGORITMO HEURISTICOS

Técnicas o algoritmos Heurísticos	Cantidad
Algoritmo Genético	22
algoritmo de colonias de abejas artificiales	3
algoritmo de colonias de hormigas	2
Algoritmo Optimización de enjambre de partículas	2
Evolución diferencial	1
Algoritmo genético multi-población	1

Optimización Multi-Objetivo hormiga león	1
Algoritmo genético híbrido	1
algoritmo de luciérnaga	1
algoritmos de forrajeo bacteriano	1
Algoritmo genético modificado	1
búsqueda tabú	1

Fuente: Elaboracion propia

Como se puede observar de la Tabla VIII, la mayoría de los autores en sus investigaciones de optimización de pruebas de software emplean el algoritmo genético, a diferencia de otros algoritmos heurísticos y técnicas, debido a que el algoritmo genético se aplican a diferentes tipos de problemas, principalmente problemas de búsqueda y optimización, esto se puede apreciar más a detalle en la figura 28.

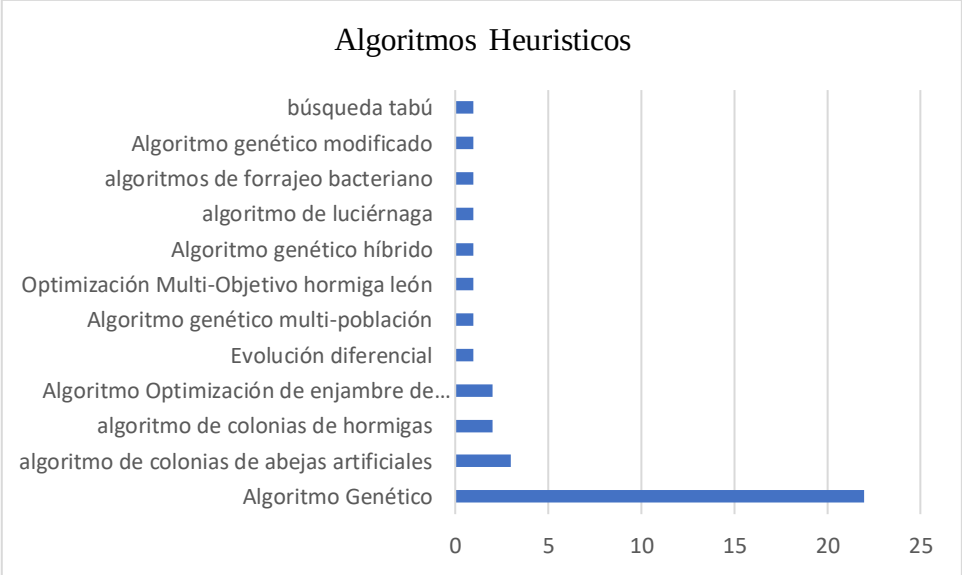

Fig. 28 Datos estadísticos de los algoritmos heurísticos
Fuente: Elaboracion propia

Pregunta 3.

¿Qué tipo de problemas presenta el estado actual de la optimización de las pruebas de software?

Se llevó a cabo un análisis en el material bibliográfico, en donde los autores describen cuales son los principales problemas o que problemas enfrentan al momento de optimizar pruebas de software. Algunos de ellos se detalla a continuación:

TABLA IX PROBLEMAS DE PRUEBAS DE SOFTWARE

Nº	Autor (s)	Problemas para optimizar pruebas de software
1	Irfan, Shadab	Se requiere mucho tiempo

2	Garg, Disha	Toman mucho tiempo para optimizar
3	Varshney, Sapna; Mehrotra, Monica	Se requiere mucho esfuerzo
4	Sharma, Chayanika; Sabharwal, Sangeeta; Sibal, Ritu	Exigen esfuerzo, tiempo y costo
5	Bashir, Muhammad Bilal	Es computacionalmente costosa
6	Khandelwal, Juhi	Son difíciles de tratar en muchas etapas
7	Rauf, Abdul	Se requiere un esfuerzo gigantesco
8	Chen, Yong; Zhong, Yong	Es un problema indecidible
9	Deng, Mingjie; Chen, Rong; Du, Zhenjun	no se presta atención a cómo generar datos de prueba automáticamente
10	Khan, Rijwan; Amjad, Mohd	Es un proceso que demanda tiempo y dificultad
11	Khan, Rijwan; Amjad, Mohd; Srivastava, Akhilesh Kumar	Es una de las fases más importantes y costosas (en términos de tiempo y costo)
12	Marcos, Paulo; Bueno, Siqueira; Jino, Mario	El software es muy impredecible en su comportamiento
13	Sabharwal, Sangeeta; Sibal, Ritu; Sharma, Chayanika	las pruebas de software rara vez son imposibles y requieren mucho tiempo

Fuente: Elaboracion propia

La mayoría de los autores tienen 1 o 2 problemas en comunes que siempre se presentan al momento de optimizar las pruebas de software. Lo cual se puede apreciar más a detalle en la siguiente tabla.

TABLA X PROBLEMAS EN COMÚN

Nº	Autor(s)	Problema en Común
1	Irfan, Shadab	Se requiere mucho tiempo para optimizar pruebas de software
2	Garg, Disha	
3	Sharma, Chayanika; Sabharwal, Sangeeta; Sibal, Ritu	
4	Khan, Rijwan; Amjad, Mohd; Srivastava, Akhilesh Kumar	
5	Sabharwal, Sangeeta; Sibal, Ritu; Sharma, Chayanika	
6	Varshney, Sapna; Mehrotra, Monica	Demanda un gran esfuerzo al momento de optimizar pruebas de software
7	Sharma, Chayanika; Sabharwal, Sangeeta; Sibal, Ritu	
8	Rauf, Abdul	
9	Sabharwal, Sangeeta; Sibal, Ritu; Sharma, Chayanika	

Fuente: Elaboracion propia

Como se puede apreciar los problemas que puede presentar al momento de optimizar las pruebas de software, es que en la mayoría de veces se exige mucho tiempo, como bien describen los autores en la Tabla X. Además de ejercer mucho tiempo, demanda esfuerzo, es por ello que la mayoría de testers no se centran en la optimización de pruebas de software, por otro lado cabe resaltar, que en la mayoría de veces no se encuentra una herramienta el cual ayude a en la optimización de pruebas de software, por tal motivo la mayoría de veces se presentan problemas en calidad con respecto al software.

Pregunta 4.

¿Se muestran etapas claras para realizar la optimización de pruebas de software?

De todo los artículos que se presentan en la **Tabla V** solo 17 de ellos presentan etapas claras y concisas con respecto a las pruebas de software para alcanzar una correcta optimización. Lo cual se evidencia en la siguiente Tabla

TABLA XI ARTÍCULOS CON PASOS Y ETAPAS PARA OPTIMIZAR PRUEBAS DE SOFTWARE

Nº	Autor (s)	Articulo
1	Irfan, Shadab	[11]
2	Chen, Ciyong; Xu, Xiaofeng; Chen, Yan	[15]
3	Sharma, Chayanika; Sabharwal, Sangeeta; Sibal, Ritu	[17]
4	Khan, Shaukat Ali	[18]
5	Andreou, Andreas S	[19]
6	Bashir, Muhammad Bilal	[20]
7	Biswas, Sumon	[21]
8	Khandelwal, Juhi	[22]
9	K, Niveth Vijay	[23]
10	Chen, Yong; Zhong, Yong	[24]
11	Singh, Mayank; Srivastava, Viranjay M; Gaurav, Kumar	[25]
12	Sofokleous, Anastasis A; Andreou, Andreas S	[27]
13	Khan, Rijwan; Amjad, Mohd	[28]
14	Marcos, Paulo; Bueno, Siqueira; Jino, Mario	[66]
15	Bodiwala, Sunny S; Devesh, C	[68]
16	Sabharwal, Sangeeta; Sibal, Ritu; Sharma, Chayanika	[69]
17	Bo, Yu; Yao, Qin Ye-mei; Gong, Ge	[72]

Fuente: Elaboración propia

Conclusiones

Las pruebas de software son esenciales para probar un gran número de rutas de código, especialmente con sistemas de gran complejidad. Del análisis del enfoque propuesto se puede concluir que el algoritmo heurístico (algoritmo genético) se puede utilizar en el ámbito de pruebas de software para producir resultados optimizados como se ha observado en todo el material bibliográfico. Así mismo se señala que se pueden incorporar diferentes técnicas meta-heurísticas o diferentes algoritmos heurísticos para obtener resultados óptimos.

BIBLIOGRAFIA

[1] Ok Hosting, "La importancia del Software en la Sociedad," 2018. [Online]. Available: https://okhosting.com/blog/la-importancia-del-software-en-la-sociedad/.

[2] G. Quintana, "Mapeo Sistemático y estudio de Caso de sobre técnicas de Generacion Automática de Pruebas," p. 8, 2014.

[3] M. E. Khan, "Diferentes enfoques de la técnica de prueba de caja blanca para encontrar errores," *International Journal of Software Engineering and its Applications,* pp. 1-2, 2011.

[4] G. Terrera, "Pruebas de Caja Negra y un enfoque práctico," 27 febrero 2017. [Online]. Available: https://testingbaires.com/pruebas-caja-negra-enfoque-practico/.

[5] E. S. Martínez, "Procedimiento para realizar pruebas de caja blanca," 2015. [Online]. Available: http://www.informatica-juridica.com/trabajos/procedimiento-realizar-pruebas-caja-blanca/.

[6] K. Rijwan and A. Mohd, "Introduccion a las pruebas de data flow testing con algoritmos geenticos," *International Journal of Computer Applications,* p. 2, 2017.

[7] E. R. Tello, "Estrategias y Tecnicas de pruebas del software," 24 octubre 2012. [Online]. Available: https://www.tamps.cinvestav.mx/~ertello/swe/sesion15.pdf.

[8] M. Vivanti, A. Mis, A. Gorla and G. Fraser, "Search-based Data-flow Test Generation," *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE),* pp. 370-379, 2013.

[9] S. Varshney and M. Mehrotra, "A Differential Evolution based Approach to generate Test Data for Data-Flow Coverage," *International Conference on Computing, Communication and Automation (ICCCA2016),* pp. 796-801, 2017.

[10] a. o. a. santos, "Algoritmo de enjambre de abejas," marzo 2013. [Online]. Available: https://es.calameo.com/books/00291084751027e8f6277.

[11] I. Shadab and R. Prabhat, "A concept of out degree in CFG for optimal test data using genetic algorithm," *Department of Computer Science & Engineering,* 2012.

[12] D. Garg, "A critical review of Artificial Bee Colony Optimizing Technique in Software Testing," *2016 1st International Conference on Innovation and Challenges in Cyber Security (ICICCS 2016),* 2016.

[13] "A Differential Evolution based Approach to generate Test Data for Data-Flow Coverage," 2016.

[14] S. Sabharwal, R. Sibal and C. Sharma, "A GENETIC ALGORITHM BASED APPROACH FOR PRIORITIZATION OF TEST CASE SCENARIOS IN STATIC TESTING," pp. 304-309, 2011.

[15] C. Chen, X. Xu, Y. Chen, X. Li and D. Guo, "A New Method of Test Data Generation for Branch Coverage in Software Testing Based on EPDG and Genetic Algorithm," pp. 1-4.

[16] A. Pachauri, "A Path and Branch Based Approach to Fitness Computation for Program Test Data Generation using Genetic Algorithm," pp. 49-55, 2015.

[17] C. Sharma, S. Sabharwal and R. Sibal, "A Survey on Software Testing Techniques using Genetic Algorithm," vol. 10, no. 1, pp. 381-393, 2013.

[18] S. A. Khan, "A Tool for Data Flow Testing using Evolutionary Approaches (ETODF)," 2013.

[19] A. S. Andreou, "criteria and genetic algorithms," pp. 867-872, 2007.

[20] M. B. Bashir, "An Experimental Tool for Search-based Mutation Testing," *2018 International Conference on Frontiers of Information Technology (FIT),* pp. 30-34, 2018.

[21] S. Biswas, "Applying Ant Colony Optimization in Software Testing to Generate Prioritized Optimal Path and Test Data," no. May, pp. 21-23, 2015.

[22] J. Khandelwal, "Approach for Automated Test Data Generation for Path Testing in Aspect-Oriented Programs using Genetic Algorithm," pp. 854-858, 2015.

[23] N. V. K, "Automated Test Path Generation using Genetic Algorithm".

[24] Y. Chen and Y. Zhong, "Automatic Path-oriented Test Data Generation Using a Multi-population Genetic Algorithm," pp. 566-570, 2008.

[25] M. Singh, V. M. Srivastava and K. Gaurav, "Automatic Test Data Generation Based On Multi-Objective Ant Lion Optimization Algorithm," pp. 168-174, 2017.

[26] M. Deng, R. Chen and Z. Du, "Automatic Test Data Generation Model by Combining Dataflow Analysis with Genetic Algorithm," no. 60775028, pp. 429-434, 2009.

[27] A. A. Sofokleous and A. S. Andreou, "Batch-Optimistic Test-Cases Generation Using Genetic Algorithms," pp. 157-164, 2007.

[2 R. Khan and M. Amjad, "Introduction to Data Flow Testing with Genetic
8] Algorithm," *International Journal of Computer Applications,* vol. 170, no. 5, pp. 39-45, 2017.

[2 medium, "¿Por qué es importante la Ingeniería de Software?," [Online]. Available:
9] https://medium.com/@FunktionellMx/por-qué-es-importante-la-ingeniería-de-software-a4000134f6e2.

[3 milenium, "software," 2018. [Online]. Available:
0] https://www.informaticamilenium.com.mx/es/temas/que-es-software.html.

[3 Drae, "definicion de Ingenieria," 2018. [Online]. Available:
1] http://dle.rae.es/srv/fetch?id=La5bCfD.

[3 udima, "Ingeniería del software," 2018. [Online]. Available:
2] https://www.udima.es/es/ingenieria-software.html.

[3 definicionabc, "Definición de Ingeniería de software," 2018. [Online]. Available:
3] https://www.definicionabc.com/tecnologia/ingenieria-de-software.php.

[3 M. Rubio, "Calidad Interna," 2018. [Online]. Available:
4] https://altenwald.org/2010/07/01/calidad-interna/.

[3 ISO/IEC 9126, "Calidad Interna y Externa," [Online]. Available:
5] https://diplomadogestioncalidadsoftware2015.wordpress.com/norma-iso-9126/calidad-interna-y-externa/.

[3 pacifitic, "QUÉ ES EL TESTING DE SOFTWARE Y POR QUÉ ES TAN
6] IMPORTANTE EN EL DESARROLLO DE SOFTWARE," 2014. [Online]. Available: https://pacifitic.org/que-es-el-testing-de-software-y-por-que-es-tan-importante-en-el-desarrollo-de-software/.

[3 M. Cristian, "Introduccio´n al Testing de Software," noviembre 2009. [Online].
7] Available: https://www.fceia.unr.edu.ar/ingsoft/testing-intro-a.pdf.

[3 odelorenzi, "Testing Software," 21 diciembre 2009. [Online]. Available:
8] https://es.slideshare.net/odelorenzi/testing-software.

[3 testingcolombia, "¡Casos de prueba, que son, como se hacen y para qué
9] sirven……!," 18 enero 2016. [Online]. Available: https://www.testingcolombia.com/casos-de-prueba-que-son-como-se-hacen-y-para-que-sirven/.

[4 testingcolombia, "¡Casos de prueba, que son, como se hacen y para qué
0] sirven……!," 18 enero 2016. [Online]. Available:

https://www.testingcolombia.com/casos-de-prueba-que-son-como-se-hacen-y-para-que-sirven/.

[41] Lenguajes y Sistemas informaticos, "TÉCNICAS DE EVALUACIÓN DINÁMICA," 15 noviembre 2018. [Online]. Available: http://www.lsi.us.es/docencia/get.php?id=361.

[42] U. Eriksson, "Este ejemplo paso a paso de prueba de caja blanca le enseña todo lo que necesita saber," 28 noviembre 2016. [Online]. Available: https://reqtest.com/testing-blog/white-box-testing-example/.

[43] softwaretestingfundamentals, "Prueba de caja blanca," 15 noviembre 2018. [Online]. Available: http://softwaretestingfundamentals.com/white-box-testing/.

[44] B. F. Gaviria, "TÉCNICAS DE CAJA BLANCA," 2017. [Online]. Available: https://campusvirtual.univalle.edu.co/moodle/pluginfile.php/1193779/mod_resource/content/1/2017A_CP_CajaBlanca.pdf.

[45] tryqa, "¿Qué es la cobertura de sucursal o la cobertura de decisión? Sus ventajas y desventajas," [Online]. Available: http://tryqa.com/what-is-decision-coverage-its-advantages-and-disadvantages/.

[46] tutorialpoint, "prueba de rama," 2018. [Online]. Available: https://www.tutorialspoint.com/software_testing_dictionary/branch_testing.htm.

[47] tutorialspoint, "Branch Testing," 2018. [Online]. Available: https://www.tutorialspoint.com/software_testing_dictionary/branch_testing.htm.

[48] tutorialspoint, "Prueba de ruta de base," 2018. [Online]. Available: https://www.tutorialspoint.com/software_testing_dictionary/basis_path_testing.htm.

[49] tutorialspoint, "Pruebas de flujo de datos," 2018. [Online]. Available: https://www.tutorialspoint.com/software_testing_dictionary/data_flow_testing.htm.

[50] B. Janvi, G. Rohit and J. Romit, *Department of Computer Science*, p. Introduccion a las pruebas de flujo de datos, 2006.

[51] S. Ting, W. Ke, M. Weikai, P. Geguang and H. Jifeng, "A Survey on Data-Flow Testing," *School of Computer Science and Software Engineering, East China Normal University*, 2017.

[52] professionalqa, "Prueba de flujo de datos," 2 septiembre 2016. [Online]. Available: http://www.professionalqa.com/data-flow-testing.

[53] J. Badlaney, R. Ghatol and R. Jadhwani, "Introduccion a las pruebas de flujo de datos," *Department of Computer Science North Carolina State University Raleigh, NC 27695, USA*, p. 02, 2006.

[54] S. Oriana, "Algoritmo de enjambre de abejas," 2018. [Online]. Available: https://es.calameo.com/books/00291084751027e8f6277.

[55] P. Gkaidatzis, D. I. Doukas, D. Labridis and A. S. Bouhouras, "Analisis Comparativo de Tecnicas Heuristicas aplicados a la Colocación óptima de la generación distribuida (ODGP)," *Department of Electrical Engineering Technological Education Institute of Western Macedonia Kozani, Greece ,* p. 02, 2017.

[56] V. Kachitvichyanukul, "Comparison of Three Evolutionary Algorithms : GA , PSO , and DE," *Industrial and Manufacturing Engineering, Asian Institute of Technology, Thailand*, pp. 03-04, 2014.

[57] V. Arunachalam, "Optimization Using Differential Evolution," *Facility for Intelligent Decision Support, Department of Civil and Environmental Engineering, The University Of Western Ontario, London, Ontario, Canada*, pp. 03-05, 2008.

[58] A. Aboshosha, "Research Gate," noviembre 2015. [Online]. Available: https://www.researchgate.net/publication/283714137_AIML-Volume7-Issue1-P1121546432.

[59] C. F. Carlos Adrián, B. Ricardo Andrés and M. C. Alexander, "ALGORITMO MULTIOBJETIVO NSGA-II APLICADO AL PROBLEMA DE LA MOCHILA," *Scientia et Technica Año XIV, No 39, Septiembre de 2008. Universidad Tecnológica de Pereira. ISSN 0122-1701*, pp. 03-06, 2008.

[60] J. Wright, "Elitist Non-dominated Sorting Genetic Algorithm: NSGA-II," 2015. [Online]. Available: https://slideplayer.com/slide/4273921/.

[61] C. L. Tian Tian, Y. Y. Qi Guo, W. Li and Q. Yan, "An Improved Ant Lion Optimization Algorithm and Its Application in Hydraulic Turbine Governing System Parameter Identificatio," *MDPI Open access journals*, pp. 02-04, 2018.

[62] S. Biswas, "Applying Ant Colony Optimization in Software Testing to Generate Prioritized Optimal Path and Test Data," *2015 Conferencia internacional sobre ingeniería eléctrica y tecnología de la información y la comunicación (ICEEICT)*, pp. 21-23, 2015.

[63] A. S. Andreou, "Andreou, Andreas S," *Seventh International Conference on Computer and Information Technology*, pp. 867-872, 2007.

[64] S. Jiang, J. Chen, Y. Zhang and J. Qian, "Evolutionary approach to generating test data for data flow test," 2018.

[65] R. Khan, M. Amjad and A. K. Srivastava, "Generation of Automatic Test Cases with Mutation Analysis and Hybrid Genetic Algorithm," pp. 1-4, 2017.

[66] P. Marcos, S. Bueno and M. Jino, "Identification of Potentially Infeasible Program Paths by Monitoring the Search for Test Data," 2000.

[67] A. Arcuri, "It Does Matter How You Normalise the Branch Distance in Search Based Software Testing," 2010.

[68] S. S. Bodiwala and C. Devesh, "Optimizing Test Case Generation In Glass Box Testing Using Bio-Inspired Algorithms," pp. 40-44, 2016.

[69] S. Sabharwal, R. Sibal and C. Sharma, "Prioritization Of Test Case Scenarios Derived From Activity Diagram Using Genetic Algorithm," pp. 481-485, 2010.

[70] L.-y. Yu and L. Lu, "Research on Test Data Generation Based on Modified Genetic and Simulated Annealing Algorithm," no. 2009, pp. 1-3.

[71] A. S. Ghiduk, "Search-Based Testing Guidance Using Dominances vs . Control Dependencies," 2009.

[72] Y. Bo, Q. Y.-m. Yao and G. Gong, 2009.

[73] C. Doungsa-ard, K. Dahal, A. Hossain and T. Suwannasart, "Test Data Generation from UML State Machine Diagrams using GAs," no. Icsea, 2007.

[74] S. Khor and P. Grogono, "Using a Genetic Algorithm and Formal Concept Analysis to Generate Branch Coverage Test Data Automatically," no. 0-3.

[75] P. Konsaard, "based Test Suite Prioritization," pp. 1-4, 2015.

[76] A. S. Ghiduk and M. J. Harrold, "Using Genetic Algorithms to Aid Test-Data Generation for Data-Flow Coverage," pp. 41-48, 2007.

[77] M. Alshraideh, L. Bottaci, M. Alshraideh and L. Bottaci, "Using program data-state diversity in test data search," 2006.