Pontificia Universidad JAVERIANA Cali

Facultad de Ingeniería y Ciencias

Acta de Correcciones al Proyecto de Grado
Ingeniería de Sistemas y Computación

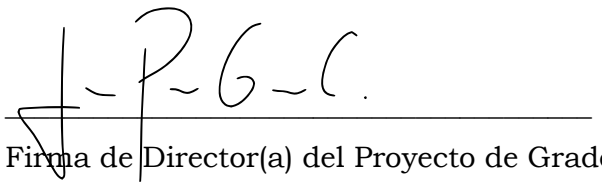
Fecha: 22 de agosto del 2023

Autores: Alejandro Beltrán Zuluaga & Geiler Orlando Hipia Mejía

Nombre del Proyecto de Grado: Implementación de una aplicación web que facilite el uso de técnicas de reducción de casos de prueba del software

Director: Juan Pablo García Cifuentes

Como indica el artículo 2.27 de las Directrices de Trabajo de Grado, he verificado que los estudiantes indicados arriba han implementado todas las correcciones que los Jurados del Proyecto de Grado definieron que se efectuaran, como consta en el Acta de Calificación correspondiente.



Firma de Director(a) del Proyecto de Grado

Nota de Aceptación

Aprobado por el Comité de Trabajo de Grado en cumplimiento de los requisitos exigidos por la Pontificia Universidad Javeriana para optar el título de Ingeniero de Sistemas y Computación.

Camilo Rocha

Dr. CAMILO ROCHA
Decano de la Facultad de Ingeniería

ING. GERARDO MAURICIO SARRIA
Director Carrera Ingeniería Sistemas y Computación.

ING. JUAN PABLO GARCÍA CIFUENTES
Director(a) Trabajo

ING. MARIA CONSTANZA PABÓN
Jurado 1

Carlos Ramirez

ING. CARLOS ALBERTO RAMIREZ
Jurado 2

Pontificia Universidad Javeriana Cali
Facultad de Ingeniería Y Ciencias.
Ingeniería de Sistemas y Computación.
Proyecto de Grado.

Implementación de una aplicación web que facilite el uso de técnicas de reducción de casos de prueba del software

Alejandro Beltrán Zuluaga
Geiler Orlando Hipia Mejía

Director: Mr. Juan Pablo García

21 de Julio del 2023

Santiago de Cali, 21 de Julio del 2023.

Señores
Pontificia Universidad Javeriana Cali.
Dr. Gerardo Mauricio Sarria
Director Carrera de Ingeniería de Sistemas y Computación.
Cali.

Cordial Saludo.

Por medio de la presente me permito informarle que los estudiantes de Ingeniería de Sistemas y Computación Alejandro Beltrán Zuluaga (cod: 8948297) y Geiler Orlando Hipia Mejía(cod: 8948106) trabajan bajo mi dirección en el proyecto de grado titulado "Implementación de una aplicación web que facilite el uso de técnicas de reducción de casos de prueba del software".

Atentamente,

Mr. Juan Pablo García

Santiago de Cali, 21 de Julio del 2023.

Señores
Pontificia Universidad Javeriana Cali.
Dr. Gerardo Mauricio Sarria
Director Carrera de Ingeniería de Sistemas y Computación.
Cali.

Cordial Saludo.

Nos permitimos presentar a su consideración el anteproyecto de grado titulado "Implementación de una aplicación web que facilite el uso de técnicas de reducción de casos de prueba del software" con el fin de cumplir con los requisitos exigidos por la Universidad para llevar a cabo el proyecto de grado y posteriormente optar al título de Ingeniero de Sistemas y Computación.

Al firmar aquí, damos fe que entendemos y conocemos las directrices para la presentación de trabajos de grado de la Facultad de Ingeniería aprobadas el 26 de Noviembre de 2009, donde se establecen los plazos y normas para el desarrollo del anteproyecto y del trabajo de grado.

Atentamente,

Alejandro Beltrán Zuluaga
Código: 8948297

Geiler Orlando Hipia Mejía
Código: 8948106

Resumen

Este proyecto se enfoca en la conceptualización, diseño e implementación de una plataforma web que tiene como objetivo principal facilitar el uso de técnicas de reducción de casos de prueba de caja negra, haciendo que sean más accesibles y fáciles de utilizar.

En el ciclo de vida del desarrollo de software, las pruebas desempeñan un papel crítico en la detección de fallas y en el aseguramiento de la calidad del producto final. Sin embargo, el proceso de pruebas puede ser costoso y demandar mucho tiempo, debido a que si el software que se está probando es complejo, con múltiples características y escenarios, se requerirá una cantidad considerable de casos de prueba para lograr una cobertura adecuada. Esto implica más tiempo y recursos para diseñar, ejecutar y mantener esas pruebas.

El principal desafío al que nos enfrentamos es la falta de soluciones prácticas y accesibles para la reducción de casos de prueba, así como la curva de aprendizaje y aplicación de estas técnicas, que puede representar una barrera para muchos profesionales y equipos de aseguramiento de calidad. Por lo tanto, este proyecto propone una solución innovadora: una plataforma web dotada de una interfaz intuitiva y amigable, que permita a los usuarios ingresar sus casos de prueba y aplicar técnicas de reducción de manera eficiente y rápida.

La plataforma se centrará en técnicas de caja negra, como la partición de equivalencias, el análisis de valores límite y los arreglos ortogonales de Taguchi. Además, se proporcionará una documentación detallada y explicaciones sobre las técnicas implementadas, así como la presentación de resultados claros y relevantes para asegurar un proceso de pruebas exitoso. Como valor añadido, la plataforma contará con una API que permitirá la integración con otras herramientas y sistemas.

La expectativa es que esta herramienta optimice tanto el tiempo como los costos asociados al desarrollo de software, potenciando la eficiencia del proceso de pruebas y garantizando la entrega de aplicaciones robustas y seguras.

Palabras Clave: Caja negra, Técnicas de reducción, Json, XLSX, Lambda, Unittest, Coverage, APIs, QA.

Abstract

This project focuses on the conceptualization, design and implementation of a web platform that aims to facilitate the use of test case reduction techniques. that aims to facilitate the use of black box test case reduction techniques, making them more accessible and easier to use. and making them more accessible and easy to use.

In the software development life cycle, testing plays a critical role in detecting failures and ensuring the quality of the final product. However, the testing process can be costly and time-consuming, because if the software being tested is complex, with multiple features and scenarios, a considerable number of test cases will be required to achieve adequate coverage. This implies more time and resources to design, execute and maintain those tests.

The main challenge we face is the lack of practical and accessible solutions for test case reduction, as well as the learning and implementation curve for these techniques, which can represent a barrier for many QA professionals and teams. Therefore, this project proposes an innovative solution: a web-based platform with an intuitive and user-friendly interface that allows users to enter their test cases and apply reduction techniques efficiently and quickly.

The platform will focus on black box techniques such as equivalence partitioning, boundary value analysis and Taguchi orthogonal arrays. In addition, detailed documentation and explanations of the implemented techniques will be provided, as well as the presentation of clear and relevant results to ensure a successful testing process. As an added value, the platform will have an API that will allow integration with other tools and systems.

The expectation is that this tool will optimize both the time and costs associated with software development, enhancing the efficiency of the testing process and ensuring the delivery of robust and secure applications.

Keywords: Black Box, Reduction Techniques, Json, XLSX, Lambda, Unittest, Coverage, APIs, QA.

Índice general

Capítulo 1

Introducción

En el desarrollo de software el ciclo de vida del producto final está dado por unas etapas que normalmente son: la planeación, el análisis de requerimientos, el diseño, la implementación, pruebas y mantenimiento. En este proceso complejo y desafiante del desarrollo de software, la calidad del producto es un aspecto fundamental que influye directamente en la satisfacción del usuario y en la confianza que se deposita en la aplicación. Para asegurar dicha calidad, las pruebas de software juegan un papel crucial, y dentro de ellas, las pruebas de caja negra son una de las técnicas más utilizadas y efectivas [Bei95].

El presente proyecto tiene como objetivo abordar el desafío de proveer una herramienta útil en la cual se pueda hacer uso de las técnicas de reducción de casos de prueba de caja negra en el contexto de desarrollo de aplicaciones o software. A lo largo del tiempo las aplicaciones han crecido exponencialmente en tamaño, lo que genera un aumento significativo en la cantidad de casos de prueba necesarios para garantizar la calidad del software y la cobertura adecuada del mismo. En el área de pruebas de un proyecto de software la creación y ejecución de un elevado número de casos de prueba han sido una tarea costosa y trabajosa, ya que requiere de un personal capacitado para realizarlas, tiempo y esfuerzo. Es aquí donde surge la necesidad de reducir los casos de prueba necesarios de forma fácil y sin comprometer la calidad del producto final. Para la solución de este problema se han presentado diversas propuestas donde la gran mayoría son heurísticos y también hay algunas aplicaciones que solo presentan una técnica y son limitados en cuanto a funcionalidades. Estas propuestas son de gran valor cuando se busca acerca de la reducción del número de casos de prueba ya que son base fundamental para el desarrollo de la aplicación.

En este proyecto se detallará el diseño de la plataforma, el proceso de desarrollo y los resultados obtenidos. En el Capítulo 2 se describe el problema planteado, formulando y sistematizando los objetivos del proyecto. El Capítulo 3 aborda el marco de referencia, incluyendo las áreas temáticas, el marco teórico y los trabajos relacionados. En el Capítulo 4 se identifican y seleccionan las técnicas de reducción de casos de prueba más adecuadas, centrándose en la partición de equivalencias, el análisis de valores límite y los arreglos ortogonales. El Capítulo 5 se dedica a la implementación de estas técnicas, tanto en el contexto de pruebas de caja negra como en la estructuración de clases. El Capítulo 6 se enfoca en el desarrollo de la aplicación web, describiendo las tecnologías utilizadas, el diseño de la interfaz de usuario y la presentación de resultados. Finalmente, en el Capítulo 7 se evalúa la aplicación web a través de un plan de pruebas que incluye casos de prueba, cobertura, procedimiento de ejecución, herramientas utilizadas y un ejemplo cuantitativo. Y en el capítulo 8 se incluye el trabajo adicional que se realizó a partir de la aplicación culminada. El documento

concluye con las principales conclusiones del proyecto y recomendaciones para trabajos futuros.

Capítulo 2

Descripción del Problema

2.1. Planteamiento del Problema

En la actualidad, existen técnicas para reducir el número de casos de prueba, como las pruebas de caja blanca, las pruebas de caja negra y las pruebas de caja gris [rt22]. Sin embargo, implementar y adaptar estas técnicas a entradas específicas resulta un proceso lento y costoso. Esto se debe a que estas técnicas requieren un profundo conocimiento del código y el funcionamiento del software. Además, las pruebas de software deben realizarse en una variedad de condiciones, lo que puede aumentar el tiempo y el costo del proceso.

La mayoría de los profesionales en informática coinciden en que probar es una tarea ardua y vital dentro del desarrollo de software. Esto se debe a que las pruebas de software son esenciales para garantizar que el software sea de alta calidad y cumpla con los requisitos del cliente. Las pruebas pueden ayudar a identificar errores y defectos en el software, lo que puede evitar que el software se lance con errores.

Aprender las técnicas de programación es difícil, y las técnicas de prueba y su automatización resultan aún más complejas. Esto se debe a que las pruebas de software requieren un conocimiento profundo del código, el funcionamiento del software y los requisitos del cliente. Además, las pruebas de software deben realizarse en una variedad de condiciones, lo que puede aumentar la complejidad del proceso. Esto se puede evidenciar en el texto "Reducing Cost Complexity of Testing : ML-based Continuous Verification" [Dom20], donde se menciona que las pruebas de software son esencialmente difíciles debido a la necesidad de conocer la funcionalidad del software, los requisitos del cliente y las complejidades técnicas de la disciplina de pruebas.

Algunas de las técnicas de caja negra se basan en algoritmos genéticos y análisis estadístico, lo cual requiere un amplio conocimiento para implementarlos. Estos conocimientos son menos comunes y más abstractos en el mundo de la programación. Aunque existen algoritmos que utilizan estas técnicas para disminuir los casos de prueba de software, resulta difícil encontrar implementaciones específicas en Internet. En su mayoría, solo se encuentran pseudocódigos y las pocas implementaciones disponibles son difíciles de entender y propensas a errores. Por lo tanto, se descarta esta opción, lo que implica la necesidad de comprender el pseudocódigo para luego llevarlo a un lenguaje de programación.

El proceso de búsqueda, comprensión, implementación y adaptación de técnicas de prueba pue-

de ser muy lento y costoso. Esto puede llevar a retrasos en el desarrollo de software y a un aumento de los costos. Además, a veces hay una gran falta de conocimiento en el área de pruebas, especialmente en las técnicas utilizadas en esta disciplina. Esta falta de conocimiento puede llevar a la creación de aplicaciones que no cumplen con los requisitos del usuario o que presentan errores significativos. Estos errores pueden generar pérdidas importantes, dependiendo del contexto y el ámbito en el que se desarrolle el software. Un ejemplo de esto es una aplicación que presenta errores significativos puede ser inestable o puede conducir a la pérdida de datos. Esto puede conducir a una pérdida de tiempo y dinero, así como a una pérdida de confianza en el usuario.

Las pruebas de software son una parte esencial del ciclo de vida de cualquier proyecto de software. Las pruebas ayudan a garantizar que el software sea de alta calidad y cumpla con los requisitos del usuario. Un mal control de calidad por parte del área de testing en un proyecto puede llevar a retrasos en las entregas o los fallas del software pueden dañar la reputación de una marca, lo que provoca la pérdida de clientes [IBM21]. Hay muchas herramientas y formas de conocimiento disponibles para ayudar a los profesionales de las pruebas a realizar sus trabajos de manera efectiva. Estas herramientas y formas de conocimiento pueden ayudar a los profesionales de las pruebas a comprender mejor el campo de las pruebas, realizar pruebas de calidad y garantizar la entrega de aplicaciones confiables y seguras.

Las técnicas de reducción de casos de prueba de caja negra pueden ser una herramienta valiosa para los equipos de desarrollo que buscan reducir los costos, el tiempo y mejorar la calidad de su software. Por ejemplo, un equipo de desarrollo que estaba trabajando en una aplicación web con un millón de líneas de código pudo reducir el número de casos de prueba de un millón a 10,000 utilizando técnicas de reducción de caja negra. Esto les ahorró tiempo y dinero, y permitió al equipo entregar la aplicación a tiempo y dentro del presupuesto. La aplicación también era más confiable y libre de errores como resultado de utilizar técnicas de reducción de caja negra.

2.1.1. Formulación

¿Cómo diseñar e implementar una aplicación web que facilite el uso de técnicas de reducción de casos de prueba del software?

2.1.2. Sistematización

- ¿Cuáles son las técnicas más utilizadas en el desarrollo de pruebas de software?
- ¿Cómo implementar las técnicas de reducción de casos de prueba?
- ¿Cómo desarrollar una aplicación web que haga uso de estas técnicas de prueba?
- ¿Cómo presentar los resultados generados por la aplicación web?
- ¿Cómo evaluar en términos de funcionalidad la aplicación web?

2.2. Objetivos

2.2.1. Objetivo General

Diseñar e implementar una aplicación web que facilite el uso de técnicas de reducción de casos de prueba del software.

2.2.2. Objetivos Específicos

- Identificar y seleccionar las técnicas de caja negra a implementar.
- Implementar las técnicas de caja negra.
- Desarrollar la aplicación web que haga uso de las técnicas.
- Presentar los resultados generados por las técnicas.
- Evaluar la aplicación web en términos de adecuación funcional mediante pruebas.

2.3. Justificación

En la fase de pruebas del software pueden surgir muchos errores como pueden ser errores de funcionamiento del sistema, rendimiento, paso de datos erróneos, entre otros; algunos de los cuales pueden ser fatales, especialmente en industrias o sectores críticos. Por ejemplo, en el desarrollo de software para el sector salud, es crucial que sea robusto, preciso y seguro, ya que incluso un pequeño error puede poner en peligro la vida de una persona. Por ejemplo un sistema de guía de rayos X llamado Therac-25 causó la muerte de seis pacientes y hirió a otros tres. El sistema usaba rayos X para guiar a los médicos durante la radioterapia, pero tenía un error de software que hacía que los rayos X fueran demasiado potentes, lo que causaba quemaduras graves y muerte [Lyn17]. Por lo tanto, realizar pruebas exhaustivas en el software es fundamental para minimizar los errores ya que se cubren todas las posibilidades que pueden cambiar el comportamiento del sistema. Sin embargo, hacer todas las pruebas de manera exhaustiva es prácticamente imposible, por lo que es vital aplicar técnicas de reducción que permitan cubrir las mejores combinaciones de casos de prueba, garantizando la cobertura de todas las funcionalidades del software y emulando situaciones reales a las que estará expuesto.

Estas pruebas deben abarcar desde los casos más comunes hasta los más extremos, con el objetivo de identificar cualquier problema que pueda surgir en situaciones poco comunes.

Para los equipos de trabajo encargados de realizar pruebas en las funcionalidades del software, contar con una aplicación web que reduce los casos de prueba de algún software resulta de gran utilidad ya que la misma es accesible por la mayoría de personas por medio de su navegador, además de disponer de interfaces que son mas comunes para el usuario. Esta herramienta les permite generar casos de prueba de manera fácil y rápida, adaptando conjuntos de entradas de usuario a

las funcionalidades y aplicando las técnicas de reducción correspondientes. Esto brinda seguridad al usuario final, al asegurar una cobertura óptima de las funcionalidades y algoritmos utilizados en el software, y permite optimizar el tiempo en la planificación, diseño y ejecución de las pruebas. Los probadores que deben adaptar sus algoritmos a conjuntos de entradas también encontrarán mucho más fácil utilizar una aplicación web que brinde estas técnicas, lo que les permitirá optimizar el tiempo en la planificación y ejecución de las pruebas al reducir el número de casos de prueba a ejecutar en el software.

Por otro lado, el mundo de las aplicaciones web está experimentando un crecimiento exponencial. El acceso a una página web es fácil y rápido ya que las personas están acostumbradas a ingresar a la mismas, lo que facilita el acceso del usuario final a la aplicación web que proporciona las técnicas de reducción. Además, la aplicación web se diseñará con la intención de ser intuitiva para la mayoría de los usuarios, brindándoles ayuda para aprovechar al máximo sus capacidades. El desarrollo de una aplicación web para proporcionar estas herramientas no requiere una gran inversión de tiempo y recursos, gracias a las herramientas y frameworks disponibles en la actualidad.

2.4. Delimitaciones y Alcances

- La aplicación web facilitará las siguientes técnicas: partición de equivalencias, análisis de valores al límite y arreglos ortogonales.
- La aplicación web contará con una interfaz que permitará al usuario ingresar los datos relacionados con el caso de prueba en la técnica seleccionada.
- La aplicación web permitirá generar la salida sólo en formato XLSX y Json.
- La aplicación web proporcionará las restricciones correspondientes a cada técnica.

2.5. Entregables

- Aplicación web.
- Documento escrito del proyecto.

Capítulo 3

Desarrollo del Proyecto

3.1. Marco de Referencia

3.1.1. Áreas temáticas

Las áreas temáticas abordadas en este trabajo de investigación son diversas. En primer lugar, se explorará el concepto de aplicación web, se analizará la naturaleza de los casos de prueba y se profundizará en el concepto de reducción de casos de prueba y su importancia. En segundo lugar, se investigarán las técnicas de prueba de caja negra, centrándose en sus características y ventajas durante la etapa de pruebas de software. En tercer lugar, se enfocará en tres tipos específicos de técnicas de prueba de caja negra: particiones de equivalencia, análisis de valores al límite y arreglos ortogonales. Estas técnicas serán implementadas en el aplicativo web con el objetivo de reducir de manera efectiva el número de casos de prueba en el software.

3.1.2. Marco Teórico

Durante el desarrollo de nuestro proyecto, han surgido conceptos clave. Estos incluyen la aplicación web, que sirve como plataforma accesible a través de un navegador web para que los usuarios interactúen con las ténicas. Además, los casos de prueba desempeñan un papel fundamental como la base sobre la cual se aplican las técnicas de reducción. La reducción de casos de prueba se convierte en el objetivo central de nuestro proyecto, ya que implica disminuir el conjunto de pruebas a un subconjunto más pequeño y sin comprometer la calidad de la validación del software.

1. **Aplicación web:** Según Lujan Mora (2002), las aplicaciones web son herramientas a las que los usuarios pueden acceder a través de un servidor web utilizando un navegador determinado [IST18] (Página 32). En nuestro proyecto, la aplicación web juega un papel fundamental, ya que es la plataforma principal a través de la cual los usuarios interactuarán con las funcionalidades de reducción de casos de prueba. Proporcionar una aplicación web como solución facilita el acceso y uso de la herramienta desde cualquier lugar con conexión a internet. Además, al utilizar un navegador web como interfaz, no se requiere instalar software adicional en los dispositivos de los usuarios, lo que simplifica la distribución y adopción de la herramienta.

2. **Casos de pruebas:**

   Según IBM, los casos de prueba son desarrollados para validar aspectos específicos y garantizar la calidad y correcto funcionamiento de un sistema [IBM21]. En nuestro proyecto, los casos de prueba son el punto de partida para aplicar las técnicas de reducción. Los usuarios ingresarán

sus conjuntos de casos de prueba en la aplicación web, definiendo los escenarios que desean validar en su software. Estos casos de prueba representan los requisitos funcionales y no funcionales que deben ser probados. La gestión y organización eficiente de los casos de prueba es esencial para el éxito del proceso de reducción y para garantizar que los aspectos críticos del software sean cubiertos adecuadamente.

3. **Reducción de casos de prueba:**

   La reducción de casos de prueba implica la disminución del conjunto de pruebas a un subconjunto más pequeño y manejable, sin comprometer la calidad de la validación del software. En nuestro proyecto, la reducción de casos de prueba es el objetivo principal de la herramienta que estamos desarrollando. Los usuarios podrán aplicar diferentes técnicas y estrategias de reducción en la aplicación web para optimizar el proceso de prueba. Al reducir la cantidad de casos de prueba necesarios, se obtienen beneficios significativos, como una mayor eficiencia en el proceso de prueba, ahorro de recursos y tiempo.

4. **Pruebas exhaustivas:**

   No es posible probar todo (todas las combinaciones de entradas y precondiciones) excepto en casos triviales. En lugar de intentar realizar pruebas exhaustivas se deberían utilizar el análisis de riesgos, las técnicas de prueba y las prioridades para centrar los esfuerzos de prueba [IST18] (Página 24). Las pruebas exhaustivas en el contexto de nuestro proyecto es el punto de partida en términos de numero de casos de prueba máximos para probar todas las entradas de usuario posibles de una funcionalidad, esto nos sirve para tener un punto de referencia de el número mas grande de casos de prueba que se pueden tomar para que un software este bien probado antes de aplicar la reducción de las técnicas.

5. **Técnicas de caja negra:**

   - **¿Qué son las ténicas de caja negra?**
     Según José Sánchez (2015), las técnicas de diseño de caja negra, también llamadas pruebas de comportamiento son las que utilizan el análisis de la especificación, tanto funcional como no funcional, sin tener en cuenta la estructura interna del programa para diseñar los casos de prueba y, a diferencia de las pruebas de caja blanca, estas pruebas se suelen realizar durante las óltimas etapas de la prueba [SÍ5] (Pagina 39, 2.3.2.2).

     El objetivo de las técnicas de caja negra es encontrar errores, funciones incorrectas, errores de inicialización y finalización, errores en la interfaz y módulos del sistema, mediante un conjunto de datos o pruebas, ver figura 3.1, aquí podemos ver que primeramente se ingresan unos datos, se realiza la manipulación de estos datos internamente y luego obtenemos una salida.

     El uso que van a tener las técnicas de reducción de casos de prueba de caja negra en el desarrollo de software es el de ayudar a garantizar que el software sea de alta calidad y

cumpla con los requisitos de los usuarios. Al identificar y probar los casos de prueba más importantes, las técnicas de reducción de casos de prueba de caja negra pueden ayudar a ahorrar tiempo y dinero, y a mejorar la calidad del software.a

Figura 3.1: Caja Negra

- **Partición de equivalencias**

  En las pruebas de partición de equivalencia, los valores de entrada del programa o del sistema se dividen en grupos que vayan a tener un comportamiento similar, de manera que puedan ser procesados de la misma forma. Las particiones de equivalencia o clases son aplicables a datos válidos y datos no válidos. También pueden aplicarse a los valores de salida, valores internos, valores relativos al tiempo o a los parámetros de interfaz [IST18] (Página 73).

- **Análisis de valores al límite**

  Esta técnica de diseño de casos de prueba complementa la técnica de partición de equivalencia u otras pruebas de caja negra. Los valores máximos y mínimos de una partición son sus valores límites [IST18] (Página 73). Las pruebas pueden diseñarse para cubrir tanto los valores límites válidos, como no válidos.

- **Arreglos ortogonales**

  Un arreglo ortogonal es un conjunto de columnas que cumplen con una propiedad particular: el producto interno entre cada par de columnas es cero. Esta técnica es especialmente útil en el área de las pruebas de software, ya que permite comprender las relaciones entre diferentes variables.

  Al utilizar esta técnica, es posible obtener resultados eficientes al combinar diversas variables en cada prueba, maximizando así la cantidad de valores y variables que se pueden evaluar en un solo caso de prueba. El objetivo de esta técnica es representar todas las posibles combinaciones entre variables y sus valores, lo que acelera el proceso de planificación de las pruebas y permite detectar errores y defectos en menos tiempo y con menos recursos [Rie23].

3.1.3. Trabajos Relacionados

1. **Analysis of black box software testing techniques: A case study [KS11]**

   En la investigación se muestra la implementación de algunas de las técnicas de raducción de casos de pruebas para demostrar la validez de una herramienta la cual se encarga de generar la ecuación de la línea para varias lineas dentro de un plano, para esto se consideraron generar varios casos de prueba, los cuales fueron validados y se llegó a la conclusión que la técnica de valores al límite era la mejor en estos casos, esta investigación nos ayuda en nuestro proyecto, ya que es un acercamiento a implementación y comparación de las pruebas en algoritmos para los cuales no fueron pensados.

2. **Comparing method equivalence class partitioning and boundary value analysis with study case [Per20]**

   En esta investigación se utiliza la ténica de partición de clases de equivalencia (ECP)y la búsqueda de números los cuales aseguran el valor del límite inferior, el valor del límite medio y el valor del límite superior del número de cadenas como entradas de un módulo del sistema. El método de este proceso se denomina análisis del valor límite (BVA). El objetivo de este estudio es comparar los métodos ECP y BVA. El resultado esperado de este estudio es averiguar el mejor método entre el ECP o el BVA. Y la comparación de los cálculos mediante pruebas matriciales estándar. Según este estudio, el método ECP es superior al BVA con 5 parámetros utilizados como puntos de referencia.

3. **Black Box testing Strategies used in Financial Industry for Functional testing [Usm16]**

   En este estudio, se profundiza en las técnicas de prueba de caja negra y se resalta su importancia en la funcionalidad del sistema sin necesidad de conocer los detalles internos. Estas técnicas garantizan un comportamiento preciso del sistema. Las estrategias de prueba son fundamentales para generar datos y casos de prueba de calidad, considerando dependencias lógicas y de datos, lo cual aumenta el potencial de identificar defectos. Las estrategias de prueba de caja negra desempeñan un papel fundamental en la detección de posibles defectos en el sistema y contribuyen al éxito del proceso de prueba. Además, se analiza la aplicación de estas técnicas en compañías financieras, evidenciando su importancia en el ciclo de pruebas.

Los trabajos relacionados anteriormente presentados proporcionan una información valiosa la cual podemos utilizar para el estudio de las técnicas de reducción de casos de prueba de caja negra, ya que estos trabajos nos muestran algunas implementaciones de estas técnicas y cómo estas funcionaron de manera correcta para que proyectos reales pudieran seguir adelante. También se nos muestra una comparativa entre algunas técnicas, lo cual nos sirve para la selección de las mejores

técnicas para nuestra aplicación y además se muestra información sobre los beneficios que trae utilizar estas técnicas en un proyecto complejo de software.

Capítulo 4

Identificación de técnicas

4.1. Reducción de casos de prueba

La reducción de pruebas de software es un tema de creciente relevancia en el ámbito del desarrollo de software. En entornos altamente competitivos, alcanzar una mayor eficiencia es fundamental para destacar y lograr el éxito en cualquier industria. Sin embargo, realizar pruebas exhaustivas de una aplicación compleja puede ser un proceso largo y costoso, especialmente cuando se trata de sistemas robustos y complejos. Según Glenford J. Myers (2004), " las pruebas completas de una aplicación compleja llevarían demasiado tiempo y requerirían demasiados recursos para ser económicamente viable " [Mye04]. Las pruebas exhaustivas implican evaluar todas las combinaciones posibles de valores de entrada, escenarios y variables, lo cual se vuelve impracticable para sistemas y aplicaciones complejas.

En este contexto, cobra una importancia fundamental la reducción de la cantidad de casos de prueba y la variedad de escenarios a los que se somete el sistema. La adopción de técnicas eficaces de reducción de pruebas de software se vuelve esencial. Estas técnicas permiten optimizar el proceso de prueba al enfocarse en los aspectos críticos y prioritarios del sistema, minimizando la cantidad de pruebas necesarias sin comprometer la calidad y la efectividad de la validación del software.

Al utilizar estrategias de reducción de casos de prueba, como la selección de casos representativos y el análisis de riesgos, es posible obtener un subconjunto más reducido pero efectivo de pruebas. Esto contribuye a maximizar los recursos disponibles, superar las limitaciones de tiempo y recursos, y lograr una mayor eficiencia en el proceso de prueba.

Entre las técnicas más utilizadas para reducir casos de prueba se encuentran las técnicas de caja negra. Estas técnicas se centran en las entradas y salidas de un objeto o funcionalidad, sin tener en cuenta la estructura interna del mismo. Al utilizar estas técnicas, el probador no necesita conocimiento detallado del código fuente y se enfoca en verificar el correcto funcionamiento del software a través de su interfaz de usuario, analizando y verificando las entradas y salidas.

Algunas de las técnicas de caja negra más utilizadas según artíículos de Software testing web[th23a] y QA lead [lea23] incluyen los valores al límite, la partición de equivalencias, transición de estados, la combinación de variables y tablas de desición, entre otras. Estas técnicas permiten a los desarrolladores probar escenarios representativos, críticos o complejos, seleccionando de manera óptima los valores de entrada y variables necesarios en el proceso de creación de casos de prueba. Gracias

a estas técnicas, los probadores no necesitan realizar pruebas exhaustivas de todos los escenarios posibles, ya que logran reducir significativamente la cantidad de casos de prueba necesarios debido a su enfoque selectivo y eficiente.

Las técnicas de caja negra reducen el número de casos de prueba al centrarse en los escenarios más representativos, críticos o complejos del software, en lugar de comprobar todas las combinaciones posibles. Aunque el porcentaje exacto de reducción de casos puede variar según varios factores, como la complejidad de la aplicación, el número de valores de entrada del usuario y el enfoque de las pruebas, estas técnicas suelen generar una reducción significativa en el número de casos de prueba necesarios para garantizar la calidad del software.

Es importante destacar que, si bien la reducción de casos de prueba puede ser beneficiosa en términos de tiempo y costos para una empresa, es esencial asegurarse de que los casos de prueba seleccionados proporcionen una cobertura suficiente del sistema y validen adecuadamente su comportamiento. Es necesario encontrar un equilibrio en el que los casos de prueba se reduzcan sin comprometer la calidad del software.

Las técnicas de reducción de casos de prueba ofrecen numerosos beneficios para cualquier software. Ya que, proporcionan un enfoque sistemático para seleccionar casos de prueba que cubran una amplia gama de escenarios, evitando pruebas repetitivas o redundantes y ayudando a evaluar los posibles escenarios a los que puede estar expuesto el software en un entorno real.

4.1.1. Selección técnicas de reducción de casos de prueba

Para la identificación y elección de las técnicas de caja negra que se van a utilizar en la herramienta buscamos las técnicas de reducción de casos de prueba que son más conocidas dentro de la disciplina de testing y las comparamos con respecto a diferentes criterios de evaluación los cuales nos van a ayudar a identificar cuales son los más adecuados para implementar dentro de la aplicación web.

Para la selección de las técnicas utilizamos criterios de evaluación los cuales se basan en estudios previos, proporcionando una base sólida en investigaciones similares. estos criterios se derivarán y adaptarán para satisfacer las necesidades específicas de nuestro contexto y del proyecto en cuestión. A través de este proceso nos aseguraremos que los criterios de evaluación sean apropiados y pertinentes para este estudio. Para el primer criterio que es cobertura de casos de prueba nos basamos en el artículo de el sitio web Ranorex[Ran22] que nos habla acerca de la importancia de la cobertura de casos de prueba para un producto de software. Para los criterios de eficiencia y eficacia de indentificación de casos de prueba nos basamos en el texto “Test Case Reduction: A Framework, Benchmark, and Comparative Study” [KKP21], en el cual nos hablan de como la eficiencia y la eficacia fueron claves para evaluar los reductores de casos de prueba y los llevaron a la conclusión de cual es mejor; y en cuanto a la adaptabilidad pensamos que para que la aplicación llegue a la

mayor cantidad de personas posibles, debemos procurar que el sistema del usuario final pueda ser probado por todas las técnicas disponibles en la aplicación. Con esto en cuenta los siguientes son los criterios de evaluación de las técnicas de reducción de casos de prueba:

- **Cobertura de casos de prueba:**

  En un principio vamos a evaluar la cobertura de casos de prueba de cada técnica analizando qué tan bien cubren los diferentes escenarios y situaciones de prueba. Esto implica revisar si las técnicas seleccionadas logran abarcar los casos más críticos, límites y combinaciones clave de valores. Esta "ayuda a controlar la calidad de las pruebas y ayuda a los probadores a crear pruebas que cubran las áreas que faltan o no están validadas" Según Software testing help en su artículo acerca del test coverage [th23b] Lo que nos da a entender que es un buen indicador al momento de elegir las técnicas ya que evidencia la calidad y la robustez de la elección de casos de prueba por parte de cada técnica

- **Eficiencia en la reducción de casos de prueba:**

  La eficiencia en la reducción de casos de prueba se puede evaluar comparando la cantidad total de casos de prueba requeridos antes y después de aplicar cada técnica. Se busca determinar cuáles de las técnicas logran una reducción significativa y efectiva en el número de casos de prueba sin comprometer la cobertura necesaria. Además de la capacidad que tiene la técnica de omitir los casos redundantes para así solo utilizar los mínimos casos de prueba que se requieran para tener una seguridad considerable del software que se prueba.

- **Identificación de errores:**

  Para evaluar la capacidad de cada técnica para identificar errores, se pueden realizar pruebas reales utilizando las técnicas seleccionadas. Se analiza la detección de errores y se evalúa si los casos de prueba generados logran descubrir problemas y fallas en el sistema de manera efectiva.

- **Adaptabilidad:**

  La adaptabilidad de cada técnica se puede evaluar considerando la flexibilidad y aplicabilidad de la técnica en diferentes sistemas, dominios y requisitos de prueba. Se analiza si las técnicas pueden adaptarse y proporcionar resultados efectivos en diversos contextos.

Estos criterios de evaluación son características importantes que cualquier técnica debería tener para ser utilizada con regularidad al resolver problemas de testing en el mundo del software, estas van a ser medidas en cuanto a qué tanto se acomoda la técnica a cada criterio de evaluación en un rango de alto, moderado y bajo donde las técnicas que mejor se adapten a los criterios son aquellas que van a ser seleccionadas para ser implementadas dentro de la aplicación web. También para la selección que vamos a realizar se tiene en cuenta un factor el cual es diferencial a la hora de elegir una técnica sobre otra y es su facilidad en cuanto a la implementación ya que hay muchas técnicas las cuales se basan en grafos y generación de casos de prueba que son más visuales que practicas, es por esto que también se tiene en cuenta su complejidad en la implementación en relación con el

beneficio que nos aportaría implementar dicha técnica.

Las técnicas de reducción de casos de prueba de caja negra que se van a comparar dentro de este estudio son: Particiones de equivalencia, Análisis de valores al límite, Selección basada en riesgo, Arreglos ortogonales, Análisis de causa-efecto y Prueba de transición de estados; las cuales cada una tiene una manera distinta de abordar aspectos particulares de la reducción de casos de prueba, pero al compararlas y evaluarlas en cuanto a los criterios de evaluación nombrados anteriormente nos va a proporcionar una visión clara de las fortalezas y limitaciones de cada una lo cual nos va a llevar a seleccionar la técnica más adecuada para implementar dentro de la aplicación web.

Cuadro 4.1: Tabla de evaluación de cada técnica de reducción de casos de prueba

Técnica De Reducción	**Cobertura**	**Eficiencia**	**Identificación Errores**	**Adaptabilidad**
Particiones De Equivalencia	Alta	Alta	Alta	Alta
Análisis De Valores Al Limite	Alta	Moderada	Alta	Alta
Arreglos Ortogonales	Moderada	Alta	Moderada	Moderada
Selección Basada En Riesgo	Moderada	Alta	Moderada	Alta
Análisis De Causa-Efecto	Moderada	Alta	Moderada	Baja
Prueba De Transición De Estados	Alta	Baja	Moderada	Moderada

Particiones de equivalencia: La técnica de particiones de equivalencia se basa en que las variables de entrada se dividen en clases distintas llamadas particiones de equivalencia en las cuales el sistema se comporta de la misma manera para todos los miembros de la clase, cada clase tiene su valor representativo y es con el cual se trabaja para realizarlos casos de prueba. Teniendo esto en cuenta, a continuación se muestran las valoraciones de la técnica de particiones de equivalencia con respecto a los diferentes criterios de evaluación:

- **Cobertura de casos de prueba:** Alta. Esta técnica tiene como objetivo cubrir los valores de entrada del sistema por medio de particiones de equivalencia por lo cual se cubre la mayor parte de las funcionalidades del sistema. Lo anterior se puede apreciar en el artćulo de A Permanasari llamado “ Comparing method equivalence class partitioning and boundary value analysis with study case ” [Per20] donde el estudio encontró que la técnica de partición de equivalencia fue más efectiva para cubrir los casos de prueba que la técnica de análisis de valores límite.

- **Eficiencia en la reducción de casos de prueba:** Alta. Esta técnica reduce la cantidad de valores a probar por medio de los representantes de cada partición de equivalencia. Esto se puede evidenciar en el artículo de “ Comparing the Effectiveness of Equivalence Partitioning, Branch Testing and Code Reading by Stepwise Abstraction Applied by Subjects ”[NJ18] en el

cual se comparan las particiones de equivalencia con otras técnicas de reducción de casos de prueba y resultó que las particiones de fueron significativamente mejores que las otras técnicas.

- **Identificación de errores:** Alta. Al seleccionar de manera correcta las particiones de equivalencia inválidas se tiene un alto potencial de identificar los errores dentro del sistema
- **Adaptabilidad:** Alta. La técnica de particiones de equivalencia es ampliamente aplicable y adaptable a una variedad de sistemas y dominios.

Análisis de valores al límite: se basa en una extensión de las particiones de equivalencias en el cual las clases válidas e inválidas se definen en relación a los bordes del rango que se esté tratando. El comportamiento en el borde de la partición de equivalencia tiene más probabilidades de ser incorrecto que el comportamiento dentro de la partición, por lo que los límites son un área en la que es probable que las pruebas produzcan defectos. por lo cual se abarcan más casos en los cuales puede fallar el sistema.

- **Cobertura de casos de prueba:** Alta. Esta técnica tiene como objetivo cubrir los valores límite y escenarios críticos, lo que permite una cobertura exhaustiva en esos aspectos.
- **Eficiencia en la reducción de casos de prueba:** Moderada. Si bien esta técnica reduce la cantidad total de casos de prueba necesarios al enfocarse en los valores límite, puede requerir una selección adicional de casos de prueba para abordar otros escenarios.
- **Identificación de errores:** Alta. Al cubrir los valores límite, esta técnica tiene un alto potencial para detectar errores relacionados con los límites del sistema.
- **Adaptabilidad:** Alta. El particionamiento en base a valores límite es una técnica ampliamente aplicable y adaptable a una variedad de sistemas y dominios.

  En el libro "The Art of Agile Development" [JS07] se proporciona una visión general del desarrollo ágil, incluyendo cómo utilizar el análisis de valores límite en un entorno ágil. En el cual se afirma que la técnica de valores al límite puede ser una herramienta eficiente a la hora de crear casos de prueba que se adapten a los cambios de los requisitos de los usuarios, además de estos ser muy precisos a la hora de encontrar errores y cubrir la mayor cantidad de funcionalidades posibles

Arreglos ortogonales: Esta técnica se basa en el concepto de probar todas las posibles combinaciones de variables o valores de entrada relevantes para el sistema; el arreglo se muestra como una matriz la cual se adapta a los valores de entrada de manera que las combinaciones son escogidas de manera estratégica para cubrir todos los casos de prueba de manera eficiente.

- **Cobertura de casos de prueba:** Moderada. La técnica de arreglos ortogonales permite cubrir diferentes combinaciones posibles, pero puede ser difícil lograr una cobertura exhaustiva

de todas las combinaciones. El artículo " Orthogonal Array Testing: A Review" [AMG94] encontró que la técnica de arreglos ortogonales fue capaz de cubrir una amplia gama de combinaciones posibles, pero que puede ser complejo probar todas las combinaciones posibles, pero si las más óptimas.

- **Eficiencia en la reducción de casos de prueba:** Alta. Esta técnica reduce el número total de casos de prueba al seleccionar combinaciones clave en lugar de probar todas las posibilidades.
- **Identificación de errores:** Moderada. Al evaluar las interacciones entre diferentes variables, esta técnica puede ayudar a descubrir errores relacionados con esas interacciones. Pero al no ser exhaustiva pueden haber combinaciones las cuales tengan errores y estos casos no se contemplen.
- **Adaptabilidad:** Moderada. La aplicabilidad de esta técnica puede depender del número y la complejidad de las variables involucradas. Ya que hay algunas condiciones que se deben cumplir para que esta pueda ser aplicada correctamente.

Selección basada en riesgos: "Las pruebas basadas en el riesgo utilizan el riesgo para priorizar y enfatizar las pruebas apropiadas durante la ejecución de las pruebas" [IST18] Esto quiere decir que los esfuerzos y el enfoque de las pruebas van a estar basados en los riesgos identificados y en cómo mitigar los mismos, esta se hace en fases tempranas del diseño de pruebas y la especificación, planeación y ejecución de las pruebas son basadas en cuanto a la mitigación de estos riesgos identificados.

- **Cobertura de casos de prueba:** Moderada. Esta técnica se enfoca en cubrir los riesgos más críticos, lo que puede implicar una cobertura selectiva de ciertos escenarios.
- **Eficiencia en la reducción de casos de prueba:** Alta. Al priorizar los casos de prueba según su importancia y riesgo, se puede reducir la cantidad total de casos de prueba necesarios.
- **Identificación de errores:** Moderada. Al enfocarse en los riesgos más críticos, esta técnica tiene un alto potencial para descubrir errores con alto impacto. pero se pueden dejar de lado los de bajo impacto
- **Adaptabilidad:** Alta. La selección basada en riesgos se puede aplicar a diferentes sistemas y dominios, adaptándose a los riesgos específicos identificados.

  En el artículo "Art of software testing" [Mye04] se proporciona una discusión sobre cómo reducir las pruebas de software sin comprometer la calidad del software. Se afirma que las pruebas basadas en riesgos pueden ser una herramienta eficaz para reducir el tiempo y los recursos necesarios para probar el software, esto por centrarse en los problemas que puede

tener el software lo que hace que se diseñen casos más centrados a los errores pero que dejan otras posibilidades abiertas.

Análisis de causa-efecto: Esta técnica utiliza el diagrama de espina de pescado o también conocido como diagrama de Ishikawa el cual es una herramienta que sirve para comprender las posibles causas de un futuro problema o defecto y su relación con los efectos esperados [Rod23]. En este diagrama la espina representa el problema observado y las ramas representan los grupos con las diferentes causas las cuales necesitan ser probadas para que no surjan los problemas dentro del software.

- **Cobertura de casos de prueba:** Moderada. Esta técnica se enfoca en cubrir diferentes combinaciones de factores, lo que puede proporcionar una cobertura adecuada, pero puede ser difícil lograr una exhaustividad completa ya que solo se enfoca en las problemáticas.
- **Eficiencia en la reducción de casos de prueba:** Alta. Al seleccionar casos de prueba que representan combinaciones clave de factores sobre problemáticas, esta técnica puede reducir el número total de casos de prueba necesarios.
- **Identificación de errores:** Moderada. Al analizar las relaciones causa-efecto, esta técnica puede ayudar a identificar errores relacionados con los factores y su impacto en los resultados.
- **Adaptabilidad:** Baja. La aplicabilidad de esta técnica puede depender de la naturaleza de los factores, las relaciones causa-efecto en el sistema bajo prueba y la obtención de la matriz de espina de pescado es complicada.

Prueba de transición de estados: Esta técnica se basa en la idea de los diferentes estados en los que puede estar un sistema y como se puede pasar de un estado a otro por medio de transiciones como resultado de alguna acción o evento específico.. Estas pruebas se centran en diseñar casos de prueba que cubran todas las posibles transiciones y evaluar que el sistema si responde de manera correcta a estas.

- **Cobertura de casos de prueba:** Alta. Esta técnica se enfoca en cubrir las transiciones de estados, lo que permite una cobertura exhaustiva en esa dimensión.
- **Eficiencia en la reducción de casos de prueba:** Bajo. En general, esta técnica puede requerir un número significativo de casos de prueba para cubrir todas las transiciones de estados posibles.
- **Identificación de errores:** Moderada. Al probar las transiciones entre estados, esta técnica puede ayudar a identificar errores relacionados con el cambio de estados del sistema.

- **Adaptabilidad:** Moderada. Esta técnica es especialmente aplicable a sistemas que tienen estados o transiciones de estados bien definidos.

Después de comparar las 6 técnicas de reducción de casos de prueba de caja negra se obtuvieron los resultados que evidencian las fortalezas y las limitaciones de cada una de las técnicas. los resultados de la comparación de las técnicas con respecto a los criterios de evaluación previamente mencionados presentaron los siguientes resultados:

- En cuanto a la cobertura de casos se prueba se logró observar que las técnicas que mejor cubren la mayoría de escenarios y que pueden encontrar de mejor manera los casos de prueba límites o críticos que pueden llevar al sistema a un posible fallo son la técnica de particiones de equivalencia, análisis de valores al límite y la transición de estados. Todas estas permiten explorar diferentes escenarios lo que deriva en una mejor cobertura para los posibles casos de prueba.

- En términos de la eficiencia en la reducción de casos de prueba las técnicas que sobresalieron por encima de las demás fueron las de Particiones de equivalencia, arreglos ortogonales y análisis de causa efecto estas son especialmente efectivas para reducir el número de casos de prueba sin comprometer la cobertura de las mismas.

- En cuanto a la identificación de errores las que destacan por encima del resto son las técnicas de arreglos ortogonales, selección basada en riesgo y análisis de causa efecto, esto se puede deber a que la mayoría de las técnicas aquí nombradas se basan en el diseño de pruebas basados en la identificación de problemáticas y errores que pueden ocurrir en el sistema. Estas técnicas además nos permiten una mejor visualización de las problemáticas raíz de nuestro sistema, lo cual nos facilita su fácil resolución y por ende la calidad del sistema.

- En términos de adaptabilidad las que están por encima de todas las demás son las de particiones de equivalencia y análisis de valores límite, ya que son las que mejor se adaptan a los diferentes contextos y requisitos del sistema, lo que los hace flexibles y aplicables a la mayoría de proyectos de testing de software.

Como se puede observar las técnicas de reducción de casos de prueba que decidimos comparar tienen sus limitaciones y sus fortalezas las cuales en el contexto en el que estamos el cual es la creación de una aplicación web que las utilice pueden valer más técnicas que tienen una alta cobertura y eficiencia, por lo cual podemos escoger las técnicas de particiones de equivalencia, análisis de valores

al límite, análisis de causa efecto y arreglos ortogonales; estas 4 opciones son las que mejor se adaptan a lo que estábamos buscando en cuanto a criterios de evaluación, pero en cuanto al criterio de la facilidad en la implementación nos quedamos con 3 técnicas las cuales mejor se adaptan a este último requisito. Es por esto que las técnicas las cuales vamos a implementar dentro de la aplicación web son:

- Particiones de equivalencia
- Análisis de valores límite
- Arreglos ortogonales

Estas técnicas son elegidas ya que son capaces de generar casos de prueba que cubren una alta gama de funcionalidades y escenarios en los cuales puede estar sometido el software, además de generar las combinaciones entre variables que mejor ayuden a tener una cobertura alta y con pocos casos de prueba, también estas cumplen la función de evitar las pruebas repetitivas o redundantes. Las técnicas de reducción se basan en el hecho de que el usuario que va a reducir los casos de prueba tiene un conocimiento moderado de la aplicación que se va a probar y por ende es capaz de reconocer los valores de entrada, valores de salida, variables y funcionamiento de la misma para verificar que las pruebas que son generadas si se acomoden a las necesidades del equipo de pruebas

Tomando esto en cuenta, vamos a revisar cada una de las técnicas de reducción de casos de prueba de caja negra en profundidad para ver sobre qué tratan, cada una de las características, como estas se pueden aplicar y cómo reducen los casos de prueba en un proyecto de software.

4.2. Técnicas de reducción de casos de prueba

4.2.1. Particiones de equivalencia

La primera técnica que vamos a implementar dentro de la aplicación es la de particiones de equivalencias la cual es una de las más importantes cuando de técnicas de caja negra se habla y una de las más fáciles de entender e implementar. Según el manual ISTQB esta técnica se basa en dividir el dominio de entrada de un componente de software en un número finito de conjuntos, llamados clases de equivalencia, de tal manera que se espera que todos los miembros de la misma clase de equivalencia se comporten de la misma manera con respecto al componente bajo prueba. [IST18] .

La técnica de particiones de equivalencia permite identificar grupos de datos de entrada o situaciones de prueba que son equivalentes en términos de comportamiento esperado del sistema.Esto significa que en lugar de probar todas las combinaciones de datos de entrada, se escogen representantes de cada partición para probar; esto reduce el número total de casos de prueba necesarios para lograr una cobertura de los requisitos adecuada, lo que ahorra tiempo y dinero en esta fase del ciclo de desarrollo de software.

Para aplicar esta técnica se debe identificar las particiones de equivalencia relevantes para cada valor de entrada y además cada partición debe tener un comportamiento distintivo dentro del sistema y este debe ser equivalente desde la perspectiva del sistema sin importar si es una partición válida o una partición inválida. Cuando se tienen las particiones ya definidas se busca que cada una tenga una definición que describa de la mejor manera que se quiere probar con esa partición o qué comportamiento va a tener esta dentro del sistema; estas se pueden definir en función de los límites de rangos que tenga el sistema o condiciones específicas que tenga que cumplir. Después de definir las particiones pasamos a definir los representantes de cada partición los cuales son un valor de prueba el cual debe cumplir con las características descritas por la partición de equivalencia sea esta una clase inválida o valida. Generalmente las particiones de equivalencia válidas representan valores los cuales pertenecen al flujo normal del sistema y no generan ningún error dentro del mismo, en cambio, las particiones inválidas representan valores los cuales no están contemplados en un flujo correcto del sistema y generalmente conllevan a que el sistema no se comporte de manera deseada o generen algún error dentro del mismo. Para evidenciar mejor cómo crear las particiones de equivalencia, los representantes y como se definen las particiones válidas e inválidas podemos recurrir a un ejemplo.

Tenemos un software que se encarga de gestionar las compras dentro de un comercio electrónico. En este software, se requiere que el usuario ingrese el valor a pagar, el cual no puede ser mayor a $ 1000 para que el pago sea aprobado. Además, el comercio electrónico cuenta con dos tipos de usuarios: general y preferencial. Dependiendo del tipo de usuario que realice el pago, se aplicará un descuento adicional a la compra o no. Con esta información, podemos definir el siguiente caso de prueba.

Funcionalidad a probar	Partición de equivalencia	Representante	Válida o inválida
Valor a pagar	Precio menor a $ 1000	700	Válida
	Precio mayor a $ 1000	1100	Inválida
Tipo de usuario	Usuario válido (Preferencial - general)	Prefrencial	Válida
	Usuario inválido	Premium	Inválida

Cuadro 4.2: Ejemplo particiones de equivalencia

Como se puede ver en el ejemplo se definieron las particiones, sus representantes y también si eran particiones válidas o inválidas, después de este punto lo que se hace es generar los caso de prueba combinando los representantes de cada una de las particiones, como en este caso son 4 tenemos 3 casos de prueba, uno donde probamos los representantes válidos que es cuando el valor a pagar es menor a $ 1000 el cual su representante es 700 y cuando el tipo de usuario está entre los usuarios válidos el cual su representante es Prefrencial; y en los otros dos casos de prueba probamos las particiones inválidas los cuales son cuando probamos que el valor a pagar es mayor a $ 1000 donde su representante es 1100 y otro caso de prueba diferente cuando probamos el usuario inválido el cual es Premium. Esto nos deja con los casos de prueba listos para probar esta funcionalidad del sistema, pero el objetivo de la implementación de esta técnica en la aplicación es que se puedan realizar casos mucho más complejos y con una cantidad de funcionalidades y variables más grande.

4.2.2. Análisis de valores límite

El manual ISTQB define el análisis de valores límite como "una extensión de la partición de equivalencia, pero solo se puede utilizar cuando la partición está ordenada, y consiste en datos numéricos o secuenciales. los valores mínimo y máximo (o valores inicial y final) de una partición son sus valores límite" [IST18]. Lo que nos quiere dar a entender el manual ISTQB acerca de la técnica de valores al límite es que debemos de tratar esta técnica como la de particiones de equivalencia, pero las particiones son identificadas como los valores extremos o límite dentro de un rango de valores de entrada de un sistema . El objetivo principal de esta técnica es probar los valores límite y cercanos a los límites de un rango ya que estos son los más propensos a causar errores y comportamientos inesperados dentro del software; estas pruebas en los límites de los rangos hacen que sean más fáciles de identificar los problemas que tienen que ver con validación, manejo de errores y comportamiento anómalo del sistema. Para aplicar esta técnica de manera adecuada deben haber valores límite los cuales son los valores más pequeños y más grandes dentro de cada rango de valores y para hallar la mayoría de casos de prueba se utilizan estos valores límite, además de un valor solo 1 unidad por encima del límite que debe ser un valor invalido y un valor 1 unidad por debajo del límite que debe ser otro valor invalido. Esta técnica es comúnmente aplicada a rangos de números, fechas y longitudes de cadenas.

Para evidenciar mejor cómo aplicar esta técnica y cómo se generan los valores al límite vamos a hacer uso de un ejemplo práctico para ver cómo utilizar la técnica en un entorno realista.

Tenemos el software del e-commerce el cual realiza transacciones de compras por medio de las cuentas de los usuarios, cuando un usuario intenta realizar una compra debe ingresar el valor a pagar, este campo del software solo admite transacciones desde $ 1 hasta $ 1000 dólares, Además, el comercio electrónico cuenta con dos tipos de usuarios: general y preferencial. Dependiendo del tipo de usuario que realice el pago, se aplicará un descuento adicional a la compra o no. Nuestro deber es probar el software para ver que si se valide esta restricción del valor a pagar con valores al límite y los usuarios con particiones de equivalencia.

Funcionalidad	**Partición de equivalencia**	**Representante**	**Válido o inválido**
Valor a pagar	Menor a $1	$0.99	Inválido
	Limite $1	$1	Válido
	Limite $1000	$1000	Válido
	Mayor a $1000	$1000.01	Inválido
Tipo de usuario	Usuario válido (Preferencial - general)	Prefrencial	Válido
	Usuario inválido	Premium	Inválido

Cuadro 4.3: Ejemplo Valores al límite

Se debe tener en cuenta que para definir los valores límite que están fuera del rango se necesita saber la diferencia que hay de un valor a otro para tener la referencia de cual es el valor que le sigue a los límites. En este ejemplo como estamos trabajando con dinero podemos ver como el salto entre un valor y el siguiente esta dado en centavos por lo cual es necesario definir esto para sacar el máximo provecho a la técnica. Como podemos ver en el cuadro 4.3 se generaron hasta 4 casos solo para esa funcionalidad del valor a pagar, esto se debe a que la técnica de valores al límite se pueden definir los puntos que tenga, en este caso se definieron 4 puntos que fueron: el límite superior, el límite inferior, uno más del límite superior y uno menos del límite inferior, lo cual es suficiente para hacer una validación rápida de la funcionalidad. si ya queremos una mayor robustez y seguridad se deben agregar más puntos para que los casos de prueba sean más precisos. Para esta funcionalidad se generan 5 casos de pruebas donde se busca probar al menos una vez cada caso inválido y probar las combinaciones de todos los casos válidos.

4.2.3. Arreglos ortogonales

Esta técnica es aquella que utiliza la estadística cuyo objetivo es optimizar la cantidad de pruebas a realizar un software por medio de la utilización de diseños involucra un número considerable de variables y valores de las mismas, las cuales hacen del modelo más complejo y de las pruebas aún más difíciles de hacer. Taguchi propone que a la hora de diseñar un producto sea de software o no tenemos múltiples variables, que en este caso los llamamos factores, que se tienen en cuenta en la etapa de planeación del mismo y que cada uno de estos factores toma valores los cuales los llamamos niveles y dependiendo de los factores y los niveles de los cuales se componga la funcionalidad la cual vamos a tratar se utiliza un arreglo ortogonal más fácil si se tienen pocos factores y niveles o se considera un arreglo más grande si se tienen muchos factores y niveles. También se debe aclarar que cuando estos son muy grandes, es más sencillo aplicar técnicas distintas antes que arreglos ortogonales.

Para utilizar la técnica de arreglos ortogonales se deben tener en cuenta en un principio los factores y los niveles y como estos se relacionan, los factores o variables pueden tener diferentes valores o niveles y estas al pertenecer a una funcionalidad, tiene su forma de operar y su forma de ser útiles, es por esto que debemos identificar cuales son los factores y niveles que deseamos testear con esta técnica. Una vez estén identificados debemos pasar a la selección del arreglo ortogonal que se ajuste más a la naturaleza de nuestro problema; para esto vamos a identificar el número de factores los cuales posee la funcionalidad que queremos probar y dependiendo de estos vamos a elegir el arreglo ortogonal que mejor se adapta a la funcionalidad:

- Si el número de factores está entre 1 y 3 se utiliza el arreglo ortogonal de L4
- Si el número de factores está entre 4 y 7 se utiliza el arreglo ortogonal de L8
- Si el número de factores está entre 8 y 11 se utiliza el arreglo ortogonal de L12
- Si el número de factores está entre 12 y 15 se utiliza el arreglo ortogonal de L16

Cabe aclarar que la técnica solo se puede aplicar si tenemos 2 o menos niveles dentro de la funcionalidad. Por ejemplo si tenemos 3 variables en nuestro sistema: Tipo cliente que puede tomar 3 valores, Plazo de pago que puede tomar 3 valores y Envío del pedido que puede tomar 2 valores; esto se puede ver representado de la siguiente manera:

$(3^2 \cdot 2^1) = L8(3^2 \cdot 2^1)$

Donde el (3^2) significa que hay 2 variables que pueden tomar 3 valores distintos cada una donde la base representa los valores que pueden tomar y el exponente el número de variables que cumplen con esta condición. En este ejemplo se pueden utilizar arreglos ortogonales ya que solo poseemos de dos niveles el (3^2) y el (2^1), si agregaramos otra variable que sea tipo de tarjeta que puede tomar 4 valores, quedaría de esta forma:

$(4^1 \cdot 3^2 \cdot 2^1)$

Y en ese caso cuando tenemos 3 niveles, no podemos aplicar los arreglos ortogonales, entonces es un factor muy importante a tener en cuenta cuando se realiza el uso de esta técnica dentro del software.

Los arreglos ortogonales que se van a utilizar en el desarrollo de la aplicación son L4, L8, L9, L12, L16, L25 y estos son arreglos los cuales al adaptarse a las entradas de las variables pueden lanzar las combinaciones óptimas para generar casos de prueba.

Para evidenciar mejor cómo podemos hacer uso de esta técnica y como la misma puede reducir los casos de prueba, vamos a realizar un ejemplo práctico en el cual podremos ver más fondo como funcionan las variables, los valores y los niveles para así pasar a elegir un arreglo ortogonal el cual nos de las mejores combinaciones con las cuales puedo probar los valores de entrada o variables del sistema.

Vamos a utilizar el ejemplo del e-commerce que hemos estado utilizando. Dentro del software, se incluyen campos en los que podemos seleccionar el tipo de cliente que somos y el valor a pagar en el comercio electrónico. En cuanto al tipo de cliente, podemos ser preferenciales o generales. En cuanto al valor a pagar, existen dos opciones: puede ser superior a $1000 o igual o inferior a $1000. con lo cual podemos observar que tenemos 2 variables las cuales ambas pueden tomar 2 valores diferentes cada una con lo cual nos dejaría con una funcionalidad con 2 factores y 2 niveles lo cual sería (2^2) con lo cual necesitaríamos un arreglo L4(2^2) para sacar todas las combinaciones y por consiguiente los casos de prueba para esta funcionalidad

Número de experimento	**Columna**		
	1	2	3
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

Cuadro 4.4: Arreglo ortogonal L4

Caso de prueba	**Cliente**	**Valor**
1	General	$700
2	General	$1000
3	Preferencial	$700
4	Preferencial	$1000

Cuadro 4.5: Ejemplo arreglos ortogonales

Vemos como con el uso de la técnica de arreglos ortogonales nos lanzó las combinaciones entre las dos variables que representan los casos de prueba para estas dos variables. en este caso al tener dos factores y utilizar el arreglo ortogonal de L4 nos lanza todas las combinaciones entre estas dos variables lo que hace que las pruebas que se realizan sean exhaustivas, pero mientras el modelo se vaya volviendo más complejo se va a poder apreciar mejor como la técnica reduce significativamente los casos de prueba por medio de las combinaciones entre las variables y sus valores.

Capítulo 5

Implementación de las técnicas

El capítulo 4 se centra en la implementación de técnicas de reducción de casos de prueba de caja negra, específicamente las técnicas de particiones de equivalencias, análisis de valores límite y arreglos ortogonales de Taguchi. Se describe en detalle cómo se solicitan los parámetros, se generan los casos de prueba y se abordan los desafíos durante la implementación. El capítulo proporciona una visión general de cómo se aplican estas técnicas para facilitar el proceso de pruebas y mejorar la eficiencia en la detección de fallas y aseguramiento de la calidad del software.

5.1. Ténicas de caja negra

5.1.1. Particiones de equivalencias

La implementación de la técnica de particiones de equivalencia en la aplicación tiene como objetivo brindar al usuario un mayor control sobre los valores de entrada necesarios para generar los casos de prueba. Al utilizar esta técnica, se solicita al usuario que proporcione el parámetro a probar, que puede ser la entrada de un formulario.

El parámetro se clasifica en diferentes clases de equivalencia, que representan categorías o conjuntos de valores con características similares. Estas clases se dividen en dos categorías principales: válidas e inválidas. Las clases válidas contienen valores de entrada que no generan errores en el sistema y se espera que sean manejados correctamente por el componente de software. Para determinar las clases de equivalencia veamos el siguiente ejemplo: si el parámetro a probar es la edad y solo se admiten números positivos, se pueden identificar dos clases de equivalencia: una clase válida que incluya valores positivos dentro del rango aceptado (por ejemplo, de 1 a 100), y una clase inválida que contenga valores negativos. Estas clases representan diferentes escenarios de prueba con características distintas.

Una vez identificadas todas las clases válidas e inválidas, se generan uno o más casos de prueba para cada clase de equivalencia. Esto implica combinar diferentes valores de parámetros para obtener una cobertura exhaustiva de la funcionalidad que se está probando. Al hacer todas las combinaciones entre los parámetros, se logra una mayor cobertura de la funcionalidad y se reduce significativamente la cantidad total de casos de prueba necesarios.

5.1.2. Análisis de valores límite

Para implementar esta técnica en nuestra aplicación, es necesario establecer algunas condiciones y limitaciones para su correcto funcionamiento. En primer lugar, se deben definir los valores límite

de entrada, los cuales deben ser números reales. Estos valores límite deben cumplir la condición de que el primer número no puede ser mayor que el segundo, para que formen un intervalo de números reales que pueda seguir una secuencia y permita realizar operaciones de suma y resta entre ellos.

Otra condición importante es el salto o diferencia entre un número límite y su siguiente número en la secuencia. Este salto también es una entrada proporcionada por el usuario y debe ser un número real que denote la diferencia entre un número y su consecuente en la secuencia. Por ejemplo, si tenemos una secuencia 2, 2.1, 2.2, ..., el salto entre cada número es de 0.1. Este valor del salto o diferencia es definido por el usuario antes de generar los casos de prueba.

Además de los valores límite y el salto, se pueden establecer condiciones adicionales para la secuencia que ayuden al usuario a ser más específico en cuanto a la solución que busca. Estas condiciones actúan como modificadores de la secuencia inicial y permiten al usuario analizar solo los valores límite que cumplan con ciertas características. Por ejemplo, si tenemos la secuencia 1, 2, 3, 4, ..., se puede establecer una condición que solo permita analizar los valores límite que sean números pares. En este caso, la aplicación mostrará los casos de prueba válidos e inválidos que cumplan con las condiciones establecidas por el usuario.

Es importante tener en cuenta algunas restricciones en el algoritmo de implementación para garantizar su funcionamiento correcto. Una de estas restricciones es que solo se permiten números enteros y reales, evitando así problemas de secuencialidad y precisión en los cálculos internos del algoritmo. Además, no se permiten valores alfanuméricos ni clases de equivalencia que no sigan una secuencia, ya que esto haría que la técnica fuera inaplicable (puede considerarse inaplicable en casos donde los valores de entrada no siguen una secuencia lógica, existen restricciones complejas o se requiere una precisión demasiado alta) y no cumpliría su propósito principal.

Para asegurar la robustez de la aplicación y prevenir posibles fallos, es necesario establecer limitaciones en los valores permitidos. Los límites superiores e inferiores se definen en $\pm 1{,}0 \times 10^9$, lo cual es suficiente para que los usuarios puedan poner a prueba su software sin exceder los límites permitidos por el lenguaje de programación `Python` en el que se implementará el algoritmo.

5.1.3. Arreglos ortogonales

Para la implementación de la técnica de arreglos ortogonales de Taguchi en nuestra herramienta, se ha optado por permitir que el usuario ingrese las variables y sus respectivos valores. Estos valores son luego ordenados dentro de los arreglos ortogonales predefinidos en la herramienta, lo que aumenta la eficiencia, dado que dichos arreglos son constantes en el código y se utilizan para generar los casos de prueba óptimos

- L4 de 3 factores y 2 niveles
- L8 de 7 factores y 2 niveles

- L9 de 4 factores y 2 niveles
- L12 de 11 factores y 2 niveles
- L16 de 15 factores y 2 niveles
- L16b de 5 factores de 4 niveles
- L25 de 6 factores con 5 niveles

Cuando hablamos de factores y niveles podemos hacer referencia a esta explicación: "Es muy frecuente que a la hora de diseñar un producto tengamos múltiples variables (FACTORES) a tener en cuenta. Cada uno de estos factores toma distintos valores (NIVELES)" [Sau12] con lo cual podemos tener en cuenta que los niveles funciona como las variables a probar por parte del usuario y los niveles la cantidad de valores que pueden tomar estas variables.

Los arreglos predefinidos se pueden ver de esta manera:

```
__L8 = {
    'array' : [
        [1, 1, 1, 1, 1, 1, 1],
        [1, 1, 1, 2, 2, 2, 2],
        [1, 2, 2, 1, 1, 2, 2],
        [1, 2, 2, 2, 2, 1, 1],
        [2, 1, 2, 1, 2, 1, 2],
        [2, 1, 2, 2, 1, 2, 1],
        [2, 2, 1, 1, 2, 2, 1],
        [2, 2, 1, 2, 1, 1, 2]
    ],
    'levels' : 2,
    'factors' : 7
}
```

5.2. Clases en Python

En la siguiente sección, se presentarán las clases que se utilizarán para la implementación de las técnicas mencionadas en la sección 4.1. A continuación, se proporciona una descripción de cada una de estas clases y sus métodos correspondientes.

5.2.1. Particiones de equivalencias

```
class EquivalencePartition:
    def __init__(self, parameters : dict):
        # Constructor de la clase EquivalencePartition
```

```
    def __valid_parameters(self):
        # Valida los parametros almacenados en la instancia de la clase

    def __generate_valid_test_cases(self):
        # Genera una lista de los casos de pruebas validos

    def __generate_invalid_test_cases(self):
        # Genera una lista de los casos de pruebas invalidos

    def build_test_cases(self):
        # Genera una lista de los casos de prueba validos e invalidos.
```

En general, los parámetros utilizados en la técnica de particiones de equivalencias se pueden representar de la siguiente manera:

```
{
    "parametro_1": {
        "clase_equivalencia_1": { "valido": boolean, "representante": any
            ↪ },
        ...
        "clase_equivalenci_n": { "valido": boolean, "representante": any }
            ↪ ,
    },
    ...,
    "parametro_n": {
        "clase_equivalencia_1": { "valido": boolean, "representante": any
            ↪ },
        ...
        "clase_equivalencia_n": { "valido": boolean, "representante": any
            ↪ },
    }
}
```

Donde:

- **parametro_n**: Representa el nombre del parámetro que se está evaluando.
- **clase_equisvalencia_n**: Es el nombre de cada clase de equivalencia que se ha identificado para el parámetro.
- **valido**: indica si la clase de equivalencia es válida (true) o inválida (false).
- **representante** es un valor representativo dentro de la clase de equivalencia, que ayuda a describir sus características.

Cada parámetro puede tener múltiples clases de equivalencia, cada una con su estado de validez (válida o inválida) y un valor representativo dentro de esa clase.

Es importante tener en cuenta que estos son ejemplos generales y los nombres de los parámetros y clases de equivalencia pueden variar según el contexto y los requisitos específicos del sistema o la aplicación en desarrollo.

Ejemplo de entrada:

```
"parametros" :  {
    "salario" : {
        "mayor_igual_1200000": {"valido": true, "representante": 1200000},
        "menor_igual_2800000": {"valido": true, "representante": 2800000},
        "valor_negativo": {"valido": false, "representante": -2800000},
        "cadena": {"valido": false, "representante": "2800000"}
    },
    "horas_trabajadas" : {
        "mayor_igual_1" : {"valido" : true, "representante" : 3},
        "menor_igual_45" : {"valido" : true, "representante" : 27},
        "valor_negativo" : {"valido" : false, "representante" : -45},
        "cadena" : {"valido" : false, "representante" : "1"}
    },
    "experiencia" : {
        "mayor_igual_1" : {"valido" : true, "representante" : 8},
        "menor_igual_20" : {"valido" : true, "representante" : 19.5},
        "valor_negativo" : {"valido" : false, "representante" : -20},
        "cadena" : {"valido" : false, "representante" : "-20.5"}
    }
}
```

Ejemplo de salida:

```
"casos_pruebas": {
    "casos_validos": [
        {
            "salario": {"clase_equivalencia": "mayor_igual_1200000",
               ↪ "representante": 1200000},
            "horas_trabajadas": {"clase_equivalencia": "mayor_igual_1",
               ↪ "representante": 3},
            "experiencia": {"clase_equivalencia": "mayor_igual_1",
               ↪ "representante": 8}
        },
        {
            "salario": {"clase_equivalencia": "mayor_igual_1200000",
               ↪ "representante": 1200000},
            "horas_trabajadas": {"clase_equivalencia": "mayor_igual_1",
               ↪ "representante": 3},
            "experiencia": {"clase_equivalencia": "menor_igual_20",
               ↪ "representante": 19.5}
```

```
        },
        ...
    ],
    "casos_invalidos": [
        {
            "salario": {"clase_equivalencia": "valor_negativo",
               ↪ "representante": -2800000},
            "horas_trabajadas": {"clase_equivalencia": "mayor_igual_1",
               ↪ "representante": 3},
            "experiencia": {"clase_equivalencia": "mayor_igual_1",
               ↪ "representante": 8}
        },
        {
            "salario": {"clase_equivalencia": "cadena", "representante":
               ↪ "2800000"},
            "horas_trabajadas": {"clase_equivalencia": "mayor_igual_1",
               ↪ "representante": 3},
            "experiencia": {"clase_equivalencia": "mayor_igual_1",
               ↪ "representante": 8}
        },
        ...
    ]
}
```

La salida proporcionada es una muestra limitada de casos de prueba válidos e inválidos generados utilizando la técnica de particiones de equivalencias. Los tres puntos suspensivos (...) indican que hay más casos de prueba disponibles además de los que se muestran en la salida.

En total, se generan 8 casos de prueba válidos, los cuales representan diferentes combinaciones de valores representativos dentro de las clases de equivalencia definidas para cada parámetro.

Además, se generan 6 casos de prueba inválidos, que representan situaciones en las que al menos uno de los parámetros no cumple con las condiciones establecidas. Estos casos de prueba ayudan a identificar escenarios en los que el sistema o la aplicación pueden comportarse de manera incorrecta o inesperada.

En general, la salida proporcionada contiene una lista de casos de prueba, tanto válidos como inválidos, generados a partir de las particiones de equivalencias. Cada elemento de la lista representa un caso de prueba independiente que se puede utilizar para probar el software. Aquí se explica cómo interpretar y utilizar esta salida:

- **casos_validos**: Esta lista contiene casos de prueba válidos. Cada caso de prueba válido se representa como un objeto y contiene los siguientes parámetros y sus respectivas clases de equivalencia y representantes. Estos casos de prueba representan situaciones en las que se espera que el software funcione correctamente.

Para cada caso de prueba válido, el usuario debe probar el software utilizando los valores especificados en ese caso. Estos valores corresponden a los representantes de las clases de equivalencia y son una muestra significativa de dicha clase. Se espera que el software funcione correctamente y produzca los resultados esperados cuando se utilicen estos valores.

- **casos_invalidos**: Esta lista contiene casos de prueba inválidos. Cada caso de prueba inválido se representa de la misma manera que los casos de prueba válidos, con los parámetros, clases de equivalencia y representantes correspondientes. Estos casos de prueba representan situaciones en las que el software no debería funcionar correctamente debido a valores incorrectos o incompatibles.

  Para cada caso de prueba inválido, el usuario también debe probar el software utilizando los valores especificados en ese caso. Sin embargo, se espera que el software presente algún tipo de error, excepción o comportamiento incorrecto cuando se utilicen estos valores inválidos. El propósito de estos casos de prueba es verificar la capacidad del software para manejar adecuadamente situaciones inesperadas o entradas incorrectas.

Si consideramos un escenario con 10 parámetros de entrada, cada uno con 4 posibles valores, realizar una prueba exhaustiva requeriría generar 4^{10} casos de prueba, lo que resultaría en un número muy grande de casos (más de un millón). Además, si asignamos un minuto para cada caso de prueba, tomaría mucho tiempo completar todas las pruebas.

Aquí es donde la técnica de particiones de equivalencia se vuelve muy útil. En lugar de probar cada combinación posible, la técnica nos permite identificar clases de equivalencia y seleccionar representantes significativos de cada clase. Esto reduce significativamente el número total de casos de prueba necesarios, al tiempo que proporciona una cobertura adecuada.

En el ejemplo anterior, la técnica de particiones de equivalencia permitió reducir el número de casos de prueba de 4^3 a solo 14 casos. Del mismo modo, en este nuevo escenario con 10 parámetros de entrada, podríamos aplicar la técnica para identificar clases de equivalencia y seleccionar representantes. Esto nos permitiría reducir drásticamente el número de casos de prueba necesarios sin comprometer la cobertura y la capacidad de detectar errores.

5.2.2. Análisis de valores límite

```
class LimitValueAnalysis:

    def __init__(self, parameters : dict):
        # Constructor de la clase EquivalencePartition

    def __valid_parameters(self):
```

```
        # Valida los parametros almacenados en la instancia de la clase

    def __has_lambda(self, value : dict):
        # Verifica si un parametro tiene un lambda

    def __get_min_max(self, lambda_str : str):
        # Determina el valor minimo y maximo de lambda.

    def __get_values_aux(self, Fn: Callable, min_value: Union[int, float],
                         max_value: Union[int, float], delta: Union[int,
                            float]):
        # Obtiene los valores limite para un lambda.

    def build_limits(self):
        # Genera los limites de cada parametro que tiene un lambda

    def build_test_cases(self):
        # Genera los casos de prueba con los limites usando la tecnica
           particion de equivalencias.
```

En general, los parámetros utilizados en la técnica de particiones de equivalencias se pueden representar de la siguiente manera:

```
"parametros": {
    "parametro_1": {
        "lambda": string | lambda,
        "delta": int | float
    },
    ..,
    "parametro_n": {
        "lambda": string | lambda,
        "delta": int | float
    },
}
```

Donde:

- **parametro_n**: Representa el nombre del parámetro que se está evaluando.
- **lambda** Es una expresión o condición que define los límites del parámetro.
- **delta**: Es el valor de diferencia o salto entre los límites del parámetro.

Ejemplo de entrada:

```
"parametros": {
    "salario": {
        "lambda": "x>=1200000␣and␣x<=2800000",
        "delta": 100000
```

```
    }
}
```

Para comprender cómo se generan los casos de prueba utilizando valores al límite observemos la Figura 5.1. En la figura, los valores azules representan los valores definidos por el lambda (máximo y mínimo), mientras que los valores verdes son aquellos que cumplen la condición establecida por el lambda. Por otro lado, el color rojo indica los valores inválidos o aquellos que no cumplen con la condición lambda. Para cada lambda, siempre se generan estos valores.

La técnica de valores límite extiende la técnica de partición de equivalencias. Después de calcular los valores límite utilizando los parámetros lambda, se aplica la técnica de partición de equivalencias.

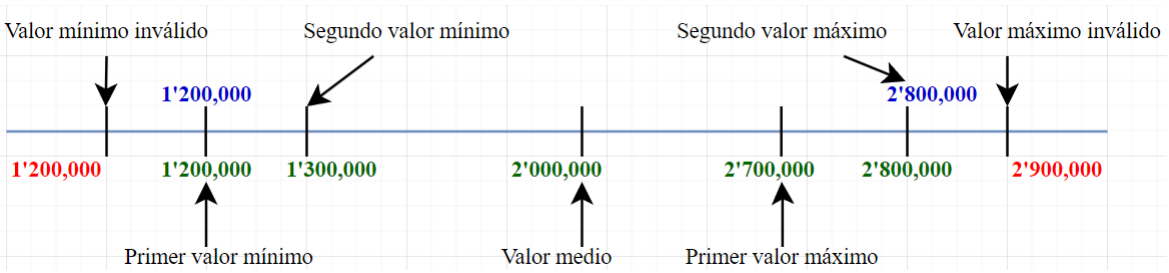


Figura 5.1: Valores límite

La salida correspondiente sería:

```
"casos-pruebas": {
    "casos_validos": [
        {
            "salario": {"clase_equivalencia": "primer_valor_minimo",
               ↪ "representante": 1200000}
        },
        {
            "salario": {"clase_equivalencia": "segundo_valor_minimo",
               ↪ "representante": 1200001}
        },
        {
            "salario": {"clase_equivalencia":
               ↪ "valor_medio","representante": 2000000}
        },
        {
            "salario": {"clase_equivalencia": "primer_valor_maximo",
               ↪ "representante": 2799999}
        },
        {
            "salario": {"clase_equivalencia": "segundo_valor_maximo",
               ↪ "representante": 2800000}
        }
```

```
    ],
    "casos_invalidos": [
        {
            "salario": {"clase_equivalencia": "valor_minimo_invalido",
               ↪ "representante": 1199999}
        },
        {
            "salario": { "clase_equivalencia": "valor_maximo_invalido",
               ↪ "representante": 2800001}
        }
    ]
}
```

Al igual que en la partición de equivalencias, cada elemento de la lista de casos válidos e inválidos representa un caso de prueba específico. Estos casos de prueba se generan con el objetivo de probar diferentes escenarios y asegurarse de que el software funcione correctamente.

5.2.3. Arreglos ortogonales

```
class OrthogonalArray:
    def __init__(self, parameters : dict):
        # Constructor de la clase OrthogonalArray.

    def __valide_parameteres(self):
        # Valida los parametros almacenados en la instancia de la clase.

    def __err_get_L(self):
        # Calcula la propiedad de longitud (L) para un conjunto de par\'
           ametros dados.

    def __get_orthogonal_array(self):
        # Obtiene el arreglo ortogonal adecuado para los parametros
           almacenados en la instancias de la clase.

    def build_test_cases(self):
        # Genera una lista de los casos de pruebas.
```

En general, los parámetros utilizados en la técnica arreglos ortogonales se pueden representar de la siguiente manera:

```
{
  "parametros": {
    "parametro_1": [v1, v2, ... ,vn],
    ...,
    "parametro_n": [v1, v2, ... ,vn]
  }
```

```
}
```

Donde:

- **parametro_1**:Representa el nombre del parámetro que se está evaluando.
- **[v1, v2, ... ,vn]**: Representa los posibles valores del parámetro, el cual puede ser de cualquier tipo.

Cada parámetro puede tener diferentes valores en su lista, y estos valores representan las opciones que se utilizarán para generar casos de prueba. Al combinar los diferentes valores de cada parámetro, se generan casos de prueba que cubren diferentes combinaciones posibles de los valores de entrada.

Ejemeplo de entrada:

```
"parametros" :  {
    "salario" : [1199999, 1200000, 120000,  2800000, 2800001],
    "horas_trabajadas" : [0, 1,  45, 46],
    "experiencia" : [0.5, 1.0, 25.0, 25.5]
}
```

Ejemplo de salida:

```
    "casos-pruebas": {
        "L": "L25",
        "keys": ["salario", "horas_trabajadas", "experiencia"],
        "array": [
            [1199999, 0, 0.5],
            [1199999, 1, 1.0],
            [1199999, 45, 25.0],
            [1199999, 46, 25.5],
            [1199999, 0, 0.5],
            [1200000, 0, 1.0],
            [1200000, 1, 25.0],
            [1200000, 45, 25.5],
            [1200000, 46, 1.0],
            [1200000, 1, 0.5],
            [120000, 0, 25.0],
            [120000, 1, 25.5],
            [120000, 45, 25.0],
            [120000, 46, 0.5],
            [120000, 45, 1.0],
            [2800000, 0, 25.5],
            [2800000, 1, 25.5],
            [2800000, 45, 0.5],
            [2800000, 46, 1.0],
            [2800000, 46, 25.0],
            [2800001, 0, 0.5],
```

```
            [2800001, 1, 0.5],
            [2800001, 45, 1.0],
            [2800001, 46, 25.0],
            [2800001, 0, 25.5]
        ]
    }
```

La salida representa un conjunto de casos de prueba generados utilizando la técnica de arreglos ortogonales.

- **L: "L25"**: Este elemento indica la designación del arreglo ortogonal utilizado para generar los casos de prueba. En este caso, se utiliza el arreglo L25.
- **keys: ["salario", "horas_trabajadas", "experiencia"]**: Esta lista enumera los nombres de los parámetros utilizados en los casos de prueba. Los parámetros incluidos son "salario", "horas trabajadas" y "experiencia".
- **array**: Esta es una matriz que contiene todas las combinaciones de valores de los parámetros para generar los casos de prueba. Cada fila de la matriz representa un caso de prueba y contiene los valores correspondientes a cada parámetro en el mismo orden que se definió en la lista "keys".

Por ejemplo, la primera fila de la matriz $[1199999, 0, 0{,}5]$ representa un caso de prueba con los valores de salario = 1199999, horas_trabajadas = 0 y experiencia = 0.5. Cada fila de la matriz representa un caso de prueba único generado combinando los diferentes valores de los parámetros.

5.2.4. Desafíos durante la Implementación de las técnicas

Durante el desarrollo e implementación de las técnicas partición de equivalencias, análisis de valores límite y arreglos ortogonales de Taguchi, nos enfrentamos a varios desafíos.Primero, comprender en profundidad cómo funcionaban estas técnicas teóricamente y segundo analizar cómo podríamos generalizar su aplicación, lo cual resultó ser difícil debido a la naturaleza particular de cada escenario.

Para la **partición de equivalencias**, el concepto teórico era combinar los casos de prueba válidos de un parámetro con los válidos de otro parámetro, sin repetir, y para los inválidos, combinar uno inválido con los válidos de los otros parámetros. Utilizamos una enfoque recursivo para resolver esto, ya que no conocemos de antemano el número de parámetros ni de clases de equivalencia por parámetro. Afortunadamente, logramos implementar esta técnica de manera efectiva una vez que comprendimos cómo abordarla.

En cuanto al **análisis de valores límite**, enfrentamos el desafío de definir un mecanismo para

la representación numérica de un parámetro, que nos permitiera encontrar los límites. Esta parte resultó ser difícil y requirió un enfoque creativo. Ideamos una solución utilizando un enfoque basado en el concepto del lambda (Una expresión o función que establece una condición lógica), lo cual nos ayudó a determinar los límites necesarios para nuestros casos de prueba. Fue un proceso complejo, pero finalmente logramos establecer una metodología sólida para abordar este aspecto.

Por último, los **arreglos ortogonales de Taguchi** presentaron un obstáculo adicional, ya que había escasa información disponible sobre los arreglos ortogonales L, es decir, las definiciones de los arrays. Tuvimos que realizar una búsqueda para encontrar suficientes ejemplos de estos arreglos que abarcaran una amplia gama de entradas. Aunque este proceso fue desafiante debido a la falta de información, logramos recopilar los arreglos necesarios para aplicar eficazmente esta técnica.

En resumen, el desarrollo e implementación de las técnicas de reducción de casos de prueba de caja negra nos presentaron retos significativos. Tuvimos que aprender teóricamente cómo funcionaban estas técnicas y luego adaptarlas a nuestras necesidades específicas. A través de una combinación de investigación, indagación y resolución creativa de problemas, logramos implementar de manera efectiva las técnicas de partición de equivalencias, análisis de valores límite y arreglos ortogonales de Taguchi, superando los obstáculos que surgieron en el camino.

5.3. Funcionamiento de las técncias

5.3.1. Partición de equivalencias

La técnica de partición de equivalencias incluye las siguientes funciones:

1. **Función** `__init__(self, parameters : dict)`: Es el constructor de la clase `EquivalencePartition`. Se utiliza para inicializar una instancia de la clase con los parámetros proporcionados en forma de un diccionario.

   Flujo de la función:

   a) Recibe un diccionario llamado `parameters` que contiene los parámetros y sus clases de equivalencia. Este diccionario se asigna al atributo privado `self.__parameters` y debe tener el siguiente formato:

```
{
    "nombre_de_parametro_1": {
        "equivalencia_1": {"valido": bool, "representante":
            ↪ valor},
        "equivalencia_2": {"valido": bool, "representante":
            ↪ valor},
        ...
```

```
            "equivalencia_n": {"valido": bool, "representante":
                ↪ valor}
        },
        "nombre_de_parametro_2": {
            "equivalencia_1": {"valido": bool, "representante":
                ↪ valor},
            "equivalencia_2": {"valido": bool, "representante":
                ↪ valor},
            ...
            "equivalencia_n": {"valido": bool, "representante":
                ↪ valor}
        },
        ...
        "nombre_de_parametro_n": {
            "equivalencia_1": {"valido": bool, "representante":
                ↪ valor},
            "equivalencia_2": {"valido": bool, "representante":
                ↪ valor},
            ...
            "equivalencia_n": {"valido": bool, "representante":
                ↪ valor}
        }
    }
```

b) Se obtienen los nombres de los atributos (claves del diccionario `parameters`) y se guardan en el atributo privado `self.__attribute_names`. Esto se hace utilizando la función`keys()` para obtener las claves y luego convirtiéndolas en una lista.

c) Se guarda la cantidad de parámetros en el atributo privado `self.__n`, que se obtiene mediante la función `len(parameters)`.

d) Se llama al método privado `self.__valid_parameters()` para validar los parámetros proporcionados. Este método se encarga de verificar que los parámetros y las clases de equivalencia tengan los valores adecuados.

```
    def __init__(self, parameters : dict):
        self.__parameters = parameters
        self.__attribute_names = list(self.__parameters.keys())
        self.__n = len(parameters)
        self.__valid_parameters()
```

2. **Función** `__valid_parameters(self)`: Es un método interno de la clase EquivalencePartition que se encarga de validar los parámetros almacenados en la instancia actual de la clase. Su objetivo es verificar que los parámetros y las clases de equivalencia tengan los valores adecuados.
   Flujo de la función:

a) Recorre cada variable (parámetro) en el diccionario de parámetros `self.__parameters` usando un bucle for var in `self.__parameters`.

b) Dentro del bucle, realiza una serie de verificaciones utilizando declaraciones `assert` para asegurarse de que los parámetros y las clases de equivalencia sean válidos. Si alguna de las verificaciones falla, se lanza una excepción `AssertionError` con un mensaje de error específico.

c) Las verificaciones realizadas son las siguientes:

- Verifica que cada parámetro tenga al menos una clase de equivalencia. Para hacerlo, comprueba si el número de claves (clases de equivalencia) en el diccionario correspondiente a la variable `var` es mayor que cero. Si no lo es, se lanza un `AssertionError` con el mensaje de error *'No hay clases de equivalencia.'*.
- Para cada clase de equivalencia en la variable `var`, realiza las siguientes comprobaciones:
  - Verifica que el valor de la clave `'valido'` en la clase de equivalencia no sea `None`. Si es `None`, se lanza un `AssertionError` con el mensaje de error.
  - Verifica que el tipo del valor de la clave `'valido'` sea `bool`. Si no es un booleano, se lanza un `AssertionError` con el mensaje de error.
  - Verifica que el valor de la clave `'representante'` en la clase de equivalencia no sea `None`. Si es `None`, se lanza un `AssertionError` con el mensaje de error.
  - Verifica que el valor de la clave `'representante'` en la clase de equivalencia no sea una cadena vacía. Si es una cadena vacía, se lanza un `AssertionError` con el mensaje de error.

d) Si ocurre algún error durante las verificaciones, se lanza una excepción `AssertionError` con un mensaje específico que indica el parámetro (`var`) donde se produjo el error y la descripción del error (`e`).

```
def __valid_parameters(self):
for var in self.__parameters:
    try:
        assert len(self.__parameters[var].keys()) > 0, 'No hay
            clases de equivalencia.'
        for equiv_class_name in self.__parameters[var]:
            equiv_class = self.__parameters[var][equiv_class_name]
            assert equiv_class.get('valido') != None, f''El valor
                "es valido" de {equiv_class_name}' es None.'
            assert type(equiv_class['valido']) == bool, f'El tipo
                "es valido: de '{equiv_class_name}' no es un
                boolean.'
            assert equiv_class.get('representante') != None, f'El
                valor de '{equiv_class_name}' es None.'
            assert equiv_class['representante'] != "", f'El valor
                de '{equiv_class_name}' es vacio.'
```

```
        except AssertionError as e:
            raise AssertionError(F'Error en `{var}`: {e}')
```

3. **Función** `build_test_cases()`: Este método genera una lista de casos de prueba válidos e inválidos para la combinación de parámetros especificados. Devuelve un diccionario que contiene dos claves: `'casos_validos'` y `'casos_invalidos'`. Cada clave contiene una lista de diccionarios que representan casos de prueba válidos e inválidos, respectivamente, para la combinación de parámetros.

```
    def build_test_cases(self):
        valid_test_cases = self.__generate_valid_test_cases()
        invalid_test_cases = self.__generate_invalid_test_cases()
        tests = {'casos_validos' : valid_test_cases, 'casos_invalidos'
            : invalid_test_cases}

        return tests
```

4. **Función** `__generate_valid_test_cases(self)`: Este método interno se encarga de generar una lista de casos de prueba válidos para la combinación de parámetros especificados en el diccionario de parámetros de la instancia actual.

   Flujo de la función:

   a) Se inicializa una lista vacía llamada `test_cases` que se utilizará para almacenar los casos de prueba válidos generados.

   b) Se define una función interna llamada `generate_combinations(remaining_attributes, current_combination)`. Es una función recursiva que se utiliza para generar todas las combinaciones posibles de las clases de equivalencia válidas para los parámetros.

   c) Si no quedan atributos (parámetros) pendientes de procesar (es decir, `remaining_attributes` está vacío), se ha generado una combinación completa de clases de equivalencia válidas para todos los parámetros. En ese caso, se agrega la combinación actual (`current_combination`) a la lista de casos de prueba válidos (`test_cases`), siempre y cuando no esté duplicada.

   d) Si aún quedan atributos pendientes de procesar, se selecciona el primer atributo de la lista de atributos pendientes (`remaining_attributes[0]`).

   e) Se filtran las clases de equivalencia de ese atributo seleccionado para obtener solo aquellas cuyo valor de `'valido'` es `True`. Esto se hace utilizando la función `filter` y una expresión `lambda` que verifica si el valor de `'valido'` es `True`.

   f) Para cada clase de equivalencia válida obtenida, se crea una nueva combinación (`new_combination`) copiando la combinación actual (`current_combination`) y agregando el atributo seleccionado con su clase de equivalencia y representante correspondientes.

g) Se realiza una llamada recursiva a `generate_combinations` con los atributos pendientes actualizados (`remaining_attributes[1:]`) y la nueva combinación generada (`new_combination`).

h) Al final (de hecho es el primero llamado), se llama a la función `generate_combinations` inicialmente con los atributos de la instancia actual (`self.__attribute_names`) y una combinación inicial vacía (`{}`) para comenzar el proceso de generación de casos de prueba válidos.

i) Se devuelve la lista de casos de prueba válidos generados (`test_cases`).

```
    def __generate_valid_test_cases(self):
        test_cases = []
        def generate_combinations(remaining_attributes,
            current_combination):
            if not remaining_attributes:
                if current_combination not in test_cases:
                    test_cases.append(current_combination)
            else:
                current_attribute = remaining_attributes[0]
                items = list(filter(lambda x: x[1]['valido'] == True,
                    self.__parameters[current_attribute].items()))
                for class_equivalent, class_value in items:
                    new_combination = current_combination.copy()
                    new_combination[current_attribute] = {'
                        clase_equivalencia' : class_equivalent, '
                        representante' : class_value['representante']}
                    generate_combinations(remaining_attributes[1:],
                        new_combination)

        generate_combinations(self.__attribute_names, {})
    return test_cases
```

5. **Función** `__generate_invalid_test_cases(self)`: Este método interno se utiliza para generar una lista de casos de prueba inválidos para la combinación de parámetros especificados en el diccionario de parámetros de la instancia actual.

   Flujo de la función:

   a) Se inicializa una lista vacía llamada `test_cases` que se utilizará para almacenar los casos de prueba inválidos generados.

   b) Se define una función interna llamada `generate_combinations(remaining_attributes, current_combination, valid)`. Esta función es una función recursiva que se utiliza para generar todas las combinaciones posibles de las clases de equivalencia inválidas para los parámetros, combinadas con clases de equivalencia válidas para los otros parámetros.

   c) Si no quedan atributos (parámetros) pendientes de procesar (es decir, `remaining_attributes` está vacío), se ha generado una combinación completa de clases de equivalencia inválidas

para todos los parámetros. En ese caso, se agrega la combinación actual (`current_combination`) a la lista de casos de prueba inválidos (`test_cases`), siempre y cuando no esté duplicada.

d) Si aún quedan atributos pendientes de procesar, se selecciona el primer atributo de la lista de atributos pendientes (`remaining_attributes[0]`).

e) Se filtran las clases de equivalencia de ese atributo seleccionado para obtener solo aquellas cuyo valor de `'valido'` coincida con el parámetro `valid`. Si `valid` es `False`, se obtienen las clases de equivalencia inválidas; si `valid` es True, se obtiene la primera clase de equivalencia válida.

f) Para cada clase de equivalencia obtenida, se crea una nueva combinación (`new_combination`) copiando la combinación actual (**current_combination**) y agregando el atributo seleccionado con su clase de equivalencia y representante correspondientes.

g) Se realiza una llamada recursiva a `generate_combinations` con los atributos pendientes actualizados (`remaining_attributes[1:]`), la nueva combinación generada (`new_combination`), y `valid=True` para continuar generando combinaciones con clases de equivalencia válidas para los otros atributos.

h) En el bucle principal, se realiza un bucle for que se ejecuta `self.__n` veces. Esto se hace para generar casos de prueba inválidos combinando las clases de equivalencia inválidas de cada atributo con las clases de equivalencia válidas de los otros atributos. En cada iteración, se llama a `generate_combinations` inicialmente con los atributos de la instancia actual (`self.__attribute_names`), una combinación inicial vacía ({}) y `valid=False` para generar combinaciones con clases de equivalencia inválidas.

i) Al final del bucle principal, se actualiza el orden de los atributos en `self.__attribute_names` para asegurarse de que cada atributo tenga la oportunidad de estar en la primera posición y combinar sus clases de equivalencia inválidas con las clases de equivalencia válidas de los otros atributos.

j) Finalmente, se devuelve la lista de casos de prueba inválidos generados (`test_cases`).

```
    def __generate_invalid_test_cases(self):
        test_cases = []
        def generate_combinations(remaining_attributes,
            current_combination, valid):
            if not remaining_attributes:
                if current_combination not in test_cases:
                    test_cases.append(current_combination)
            else:
                current_attribute = remaining_attributes[0]
                items = list(filter(lambda x: x[1]['valido'] == valid,
                    self.__parameters[current_attribute].items()))
                items = items if not valid else [items[0]]
                for class_equiv, class_value in items:
                    new_combination = current_combination.copy()
```

```
                    new_combination[current_attribute] = { '
                        clase_equivalencia' : class_equiv, '
                        representante' : class_value['representante'] }
                    generate_combinations(remaining_attributes[1:],
                        new_combination, valid=True)
        for _ in range(self.__n):
            generate_combinations(self.__attribute_names, {}, valid=
                False)
            self.__attribute_names = self.__attribute_names[1:] + [
                self.__attribute_names[0]]
        return test_cases
```

5.3.2. Análisis de valores límite

1. **Función __init__(self, parameters: dict)**: Es el constructor de la clase `LimitValueAnalysis`. Se utiliza para inicializar un objeto de prueba con los parámetros especificados.

   Flujo de la función:

   a) Recibe un diccionario llamado `parameters` que define los parámetros de prueba para la ténica. Cada clave del diccionario es el nombre del parámetro y su valor es otro diccionario que contiene las siguientes claves:

   - `"lambda"`: Es una función anónima que es utilizada para expresar operaciones o cálculos. En este caso, se usa para limitar el dominio de una función lambda a un intervalo cerrado:

     $$[x_{\min}, x_{\max}]$$

     Donde $x_{\min}$ es el valor mínimo y $x_{\max}$ es el valor máximo del dominio.

     $$\lambda x : x \geq x_{\texttt{min}} \ \texttt{y} \ x \leq x_{\texttt{max}}$$

     Ejemplo:

     $$\lambda x : x \geq 0 \ \texttt{y} \ x \leq 100$$

     . En esta expresión, se indica que `x` debe ser mayor o igual que 0 y, al mismo tiempo, ser menor o igual que 100. Esto establece un rango cerrado para `x` con un valor mínimo de 0 y un valor máximo de 100.
   - `"delta"`: El intervalo entre cada valor límite para el parámetro de prueba.

```
    {
        "parametro_1": {
            "lambda": string | lambda,
            "delta": int | float
        },
```

```
        ...,
        "parametro_n": {
            "lambda": string | lambda,
            "delta": int | float
        },
    }
```

Nota: Esta técnica acepta parámetros de partición de equivalencias, pero solo se evalúan aquellos que tienen un tipo lambda.

b) Se asigna el diccionario `parameters` al atributo privado `self.__parameters` de la instancia actual.

c) Se llama al método privado `self.__valid_parameters()` para verificar que los parámetros proporcionados sean válidos. Este método realiza una serie de validaciones en los parámetros, como verificar que cada parámetro tenga una cadena lambda válida y un valor delta válido. Si se encuentra algún problema durante la validación, se lanzará una excepción. (Sólo aplica para parámetros de tipo `lambda`).

2. **Función** `__valid_parameters(self)` : Es un método interno de la clase `LimitValueAnalysis` que se utiliza para verificar que los parámetros de la instancia sean válidos.

   Flujo de la función:

   a) Se itera a través de los elementos del diccionario `self.__parameters` utilizando un bucle `for` con el formato `for key, value in self.__parameters.items():`.

   b) Para cada parámetro, se verifica si tiene una cadena `lambda` válida utilizando el método privado `self.__has_lambda(value)`. Este método verifica si el parámetro tiene las claves `"lambda"` y `"delta"`. Si el parámetro no tiene una cadena `lambda` válida, se omite el procesamiento de ese parámetro (No se finaliza porque puede que sea un parámetro de la clase `EquivalencePartition`).

   c) Si el parámetro tiene una cadena `lambda` válida, se extrae la cadena `lambda` y el valor `delta` del diccionario.

   d) Se realizan una serie de verificaciones en la cadena `lambda` y el valor `delta`:

   - Se verifica que la cadena `lambda (lambda_str)` sea del tipo `str`. Si no es así, se lanza una excepción de tipo `TypeError` indicando que el tipo es incorrecto.
   - Se verifica que la longitud de la cadena lambda sea al menos `LimitValueAnalysis.MIN_OPT_SIZE`. Si no cumple con esta longitud mínima, se lanza una excepción de tipo `ValueError` indicando que la longitud es incorrecta.
   - Se intenta analizar sintácticamente la cadena `lambda` utilizando el módulo `ast`. Si se produce un error de sintaxis, se lanza una excepción de tipo `SyntaxError` indicando que la sintaxis es inválida.

- Se reemplaza el carácter `^` en la cadena lambda con `**` y se evalúa la cadena utilizando la función `eval()` en un contexto donde `lambda x` es una función anónima. Esto se hace para verificar que la cadena lambda sea evaluable correctamente sin errores de tiempo de ejecución.

e) Se verifica que el valor `delta` sea del tipo `float` o `int`. Si no es así, se lanza una excepción de tipo `TypeError` indicando que el tipo es incorrecto.

```
    def __valid_parameters(self):
     for key, value in self.__parameters.items():
         if self.__has_lambda(value):
             lambda_str = value['lambda']
             delta = value['delta']
             if type(lambda_str) != str:
                 raise TypeError(f'Tipo incorrecto {key}:{
                     lambda_str}')
             if len(lambda_str) < LimitValueAnalysis.MIN_OPT_SIZE:
                 raise ValueError(f'El lambda de {key} debe ser
                     almenos de {LimitValueAnalysis.MIN_OPT_SIZE}.')
             try:
                 ast.parse(lambda_str)
                 _ = lambda x : eval(lambda_str.replace('^', '**'))
             except SyntaxError:
                 raise SyntaxError(f'El lambda de {key} tiene una
                     sintaxis inv\'alida')

             if type(delta) != float and type(delta) != int:
                 raise TypeError(f'El tipo del delta/paso es
                     incorrecto ({key}:{delta})')
```

3. **Función** `__has_lambda(self, value: dict)`: Es un método interno que se utiliza para verificar si un diccionario tiene las claves `"lambda"` y `"delta"`, que representan la cadena lambda y el valor delta respectivamente.

   Flujo de la función:

   a) Se realiza un intento (try) de ejecutar las siguientes líneas de código:
   - Se compara el conjunto de claves del diccionario `value` con el conjunto `['lambda', 'delta']` utilizando el operador de igualdad `(==)`.
   - Si los conjuntos son iguales, se devuelve `True`. De lo contrario, se devuelve `False`.

   b) Si ocurre alguna excepción durante el intento, se captura `(except)` y se lanza una excepción de tipo `ValueError`. Esta excepción indica que el objeto `value` debe tener las claves `"lambda"` y `"delta"` para que pueda ser considerado válido en el contexto de la función.

```
    def __has_lambda(self, value : dict):
    try:
        return set(value.keys()) == set(['lambda', 'delta'])
    except Exception as e:
        raise ValueError(f'El objeto {value} debe tener la clave delta
            y lambda.')
```

4. **Función** `build_limits(self)`: Es un método que se utiliza para construir una lista de objetos que representan los casos de prueba para cada parámetro en los valores proporcionados.

   Flujo de la función:

   a) Se inicializa un diccionario vacío llamado `limits` que almacenará los límites de prueba para cada parámetro.

   b) Se itera sobre cada par `clave-valor` en el diccionario `self.__parameters` (los parámetros especificados). Se verifica si el parámetro tiene una cadena lambda y un valor delta utilizando el método `__has_lambda(value)`.

   c) Si el parámetro tiene una cadena lambda y un valor delta, se procede a construir los límites de prueba.

   d) Se obtiene la cadena `lambda` y el valor `delta` del parámetro actual.

   e) Se crea una función `Fn` que evalúa la cadena lambda utilizando `eval()` y reemplazando el operador ^ con ** para representar la potenciación.

   f) Se llama al método `__get_min_max(lambda_str)` para obtener el valor mínimo y el valor máximo de la cadena lambda. Este método analiza la cadena lambda y extrae los valores numéricos para determinar los límites.

   g) Se llama al método `__get_values_aux(Fn, min_value, max_value, delta)` para obtener los valores límite para el parámetro. Este método utiliza la función `Fn` para evaluar las restricciones, el valor mínimo y máximo, y el valor delta para calcular los límites de prueba.

   h) Los límites de prueba obtenidos se asignan al diccionario `limits` utilizando el nombre del parámetro como clave.

   i) Una vez que se han procesado todos los parámetros, se devuelve el diccionario `limits` que contiene los casos de prueba para cada parámetro.

```
    def build_limits(self):
        limits = {}
        for key, value in self.__parameters.items():
            if self.__has_lambda(value):
                lambda_str = value['lambda']
                delta = value['delta']
```

```
                Fn = lambda x : eval(lambda_str.replace('^', '**'))
                min_value, max_value = self.__get_min_max(lambda_str)
                limit_values =  self.__get_values_aux(Fn, min_value,
                    max_value, delta)
                limits[key] = limit_values
        return limits
```

5. **Función** `__get_min_max(self, lambda_str: str)`: Es un método que se utiliza para determinar el valor mínimo y máximo de una expresión lambda dada en formato de cadena (sólo el valor númerico del lambda).

   Flujo de la función:

   a) Se especifica que la función toma un parámetro `lambda_str` que representa la expresión matemática en formato de cadena.

   b) Se intenta ejecutar el código dentro de un bloque `try-except` para manejar posibles errores.

   c) Se define un patrón de expresión regular utilizando la función `re.compile()` para encontrar las coincidencias de números en la cadena lambda. El patrón busca números enteros o flotantes que pueden estar seguidos de operadores matemáticos `(+, -, *, /, **, ^)` y otros números.

   d) Se utiliza la función `re.findall()` para encontrar todas las coincidencias del patrón en la cadena `lambda`. Esto devuelve una lista de cadenas que representan los números encontrados.

   e) Se itera sobre la lista de cadenas encontradas y se utiliza `eval()` para evaluar cada cadena como una expresión matemática. El operador `^` se reemplaza por `**` ya que `**` se utiliza en `Python` para representar la potenciación.

   f) Los valores obtenidos se almacenan en una lista llamada `values`.

   g) Se utiliza la función `min()` para encontrar el valor mínimo en la lista `values` y se almacena en la variable `min_value`.

   h) Se utiliza la función `max()` para encontrar el valor máximo en la lista `values` y se almacena en la variable `max_value`.

   i) Se verifica si el valor mínimo es igual al valor máximo. Si son iguales, se lanza una excepción `ValueError` con un mensaje indicando que el valor mínimo debe ser mayor que el valor máximo.

   j) Se devuelve una `tupla` que contiene el valor mínimo y el valor máximo de la expresión matemática.

   k) Si se produce algún error durante la ejecución del bloque `try`, se captura la excepción y se lanza una excepción `Exception` con un mensaje indicando el error ocurrido al intentar determinar el valor mínimo y máximo de la expresión.

```
    def __get_min_max(self, lambda_str : str):
        try:
            pattern = re.compile(r'-?\d+(?:\.\d+)?(?:\s*[+\-*/\*\*\^]\
                s*-?\d+(?:\.\d+)?)*|-?\d+(?:\.\d+)?')
            matches = re.findall(pattern, lambda_str)
            values = [eval(match.replace('^', '**')) for match in
                matches]
            min_value = min(values)
            max_value = max(values)
            if min_value == max_value:
                raise ValueError(f'Error, el valor m\'inimo debe ser
                    mayor que el valor m\'aximo. ({lambda_str})')
            return min_value, max_value
        except Exception as e:
            raise  Exception(f'Error al intentar determinar el valor m
                \'inimo y m\'aximo del lambda. ({lambda_str})')
```

6. **Función** `build_test_cases`: Es un método que se utiliza para construir los casos de prueba a partir de los límites generados en la función `build_limits` y los parámetros especificados en la instancia de la clase.

   Flujo de la función:

   a) Se llama a la función `build_limits` para obtener los límites de prueba para cada parámetro. Los límites se almacenan en la variable `limits`.

   b) Se itera sobre cada par de `clave-valor` en el diccionario `limits`. Cada clave representa el nombre de un parámetro, y cada valor es un diccionario que contiene los límites de prueba asociados a ese parámetro.

   c) Para cada parámetro, se crea un diccionario vacío en `self.__parameters` utilizando el nombre del parámetro como clave. Esto se hace para almacenar los límites de prueba generados.

   d) Se itera sobre cada `clave-valor` en el diccionario de límites del parámetro actual. Cada clave representa un límite específico, como `ïnvalid_min", "first_min", "second_min", etc.`, y cada valor es el valor correspondiente a ese límite.

   e) Se crea un diccionario con dos claves: `"valido"` y `representante"`. La clave `"valido"` tiene el valor `True` si la cadena `ïnvalid"` no está presente en la clave del límite, lo que indica que es un límite válido. La clave `representante"` tiene el valor del límite.

   f) El diccionario recién creado se asigna al diccionario `self.__parameters` bajo la clave correspondiente al nombre del parámetro y la clave del límite. Esto se hace para almacenar los detalles de cada límite de prueba.

g) Se crea una instancia de la clase `EquivalencePartition` pasando `self.__parameters` como parámetro. Esto se hace para utilizar la técnica de partición de equivalencias y generar casos de prueba.

h) Se llama al método `build_test_cases` en la instancia de `EquivalencePartition` para generar los casos de prueba. Los casos de prueba se devuelven como resultado de la función `build_test_cases`.

```
    def build_test_cases(self):
        limits = self.build_limits()
        for key, value in limits.items():
            self.__parameters[key] = {}
            for value_key, value_value in value.items():
                self.__parameters[key][value_key] = {'valido' : not '
                   invalid' in value_key, 'representante' :
                   value_value}

        equivalence = EquivalencePartition(parameters=self.
           __parameters)
        return equivalence.build_test_cases()
```

5.3.3. Arreglos Ortogonales

La ténica arreglos ortogonales incluye las siguientes funciones:

1. **Función** `__init__(self, parameters : dict)` : Es el método constructor de la clase `OrthogonalArray`.Se utiliza para inicializar un objeto de prueba con los parámetros especificados

   Flujo de la función:

   a) ecibe un diccionario llamado parameters que define los parámetros de prueba para la ténica. Donde cada clave es el nombre del parámetro y su valor es una lista con los posibles valores que puede tomar el parámetro.

   ```
   {
     "parametros": {
       "parametro_1": [v1, v2, ... ,vn],
       ...,
       "parametro_n": [v1, v2, ... ,vn]
     }
   }
   ```

   b) Se asigna el diccionario `parameters` al atributo `__parameters` del objeto. Este diccionario contiene los parámetros de prueba para el objeto.

c) Luego se asigna el diccionario `orthogonal_arrays` al atributo `self.__orthogonal_arrays` del objeto. Este diccionario contiene diferentes arreglos ortogonales predefinidos con sus propias características (número de factores y niveles).

El diccionario de los arreglos ortogonales tiene el siguiente formato, donde cada `__L` es un array único.

```
orthogonal_arrays = {
    'L4' : __L4, 'L8' : __L8, 'L9' : __L9,
    'L12' : __L12, 'L16' : __L16, 'L16B' : __L16B,
    'L18' : __L18, 'L25' : __L25, 'L27' : __L27,
    'L32' : __L32
}
```

d) Se llama al método privado `self.__valide_parameteres()` para verificar que los parámetros proporcionados sean válidos. Este método realiza una serie de validaciones en los parámetros, como verificar que cada parámetro tenga un nombre válido, que el valor asociado sea una lista no vacía y que no contenga elementos nulos. Si se encuentra algún problema durante la validación, se lanzará una excepción.

e) Se calcula el número de factores tomando la longitud del conjunto de claves del diccionario de parámetros .

f) Finalemente, se calcula el máximo nivel tomando la longitud del valor más largo del diccionario de parámetros. Esto corresponde al número máximo de posibles valores que puede tomar un parámetro en la lista de valores proporcionada.

```
def __init__(self, parameters : dict):

    self.__parameters = parameters
    self.__orthogonal_arrays = orthogonal_arrays

    self.__valide_parameteres()

    self.__num_factors = len(self.__parameters.keys())
    self.__max_level = len(max(self.__parameters.values(), key=len
        ))
```

2. **Función** `__valide_parameteres(self)`: Este método se utiliza para validar los parámetros proporcionados al objeto y asegurarse de que cumplan ciertos criterios antes de utilizarlos en otras partes de la clase.

Flujo de la función:

a) Se recorre el diccionario de parámetros `__parameters` utilizando un bucle `for`. Cada iteración del bucle obtiene una clave `(key)` y un valor `(value)` del diccionario.

Para cada parámetro, se realizan las siguientes validaciones:

- Se verifica si la clave `(key)` es `None`. Si la clave es `None`, se lanza una excepción `ValueError` con un mensaje que indica que se proporcionó una clave inválida.
- Se verifica si el valor `(value)` asociado a la clave no es una lista. Si el valor no es una lista, se lanza una excepción `ValueError` con un mensaje que indica que el valor de la clave no es una lista.
- Se verifica si la longitud del valor `(value)` de la clave es igual a cero (es decir, si la lista está vacía). Si la longitud es cero, se lanza una excepción `ValueError` con un mensaje que indica que la longitud del valor debe ser al menos 1.
- Se verifica si hay algún valor `None` en la lista. Si se encuentra algún valor `None`, se lanza una excepción `ValueError` con un mensaje que indica que los valores de la lista deben ser diferentes de `None`.

b) Si todas las validaciones son exitosas y no se encuentra ningún problema con los parámetros, la función termina sin devolver ningún valor.

```
    def __valide_parameteres(self):
        for key, value in self.__parameters.items():
            if key is None:
                raise ValueError(f'Clave inv\'alida {key}')
            if type(value) != list:
                raise ValueError(f'El valor de {key} no es una
                    lista.')
            if len(value) == 0:
                raise ValueError(f'La longitud del valor de {key}
                    debe ser al menos 1.')
            if None in value:
                raise ValueError(f'Los valores de {key} deben ser
                    diferentes de None.')
```

3. **Función** `__get_orthogonal_array(self)`: Este método busca y devuelve un arreglo ortogonal adecuado que cumpla ciertas condiciones para el conjunto de parámetros dados.

   Flujo de la función:

   a) Se recorre el diccionario `__orthogonal_arrays` utilizando un bucle `for`. Cada iteración del bucle obtiene una clave `(key)` y un valor `(value)` del diccionario.

   b) Para cada arreglo ortogonal en el diccionario, se verifica si su cantidad de niveles `(levels)` es igual al máximo nivel `(__max_level)` encontrado en los valores de los parámetros y si su cantidad de factores `(factors)` es mayor o igual a la cantidad de factores en los parámetros `(__num_factors)`. Los factores se refieren a la cantidad de parámetros diferentes en el conjunto de prueba.

c) Si se encuentra un arreglo ortogonal que cumpla con las condiciones, se devuelve la clave y el arreglo asociado, utilizando `return key, value['array']`. El arreglo ortogonal contiene las combinaciones óptimas de los valores de los parámetros para construir el conjunto de prueba.

d) Si no se encuentra ningún arreglo ortogonal que cumpla con las condiciones, se lanza una excepción `ValueError` con un mensaje que indica que no se encontró un arreglo ortogonal adecuado para el conjunto de parámetros. La función `__err_get_L()` se utiliza para obtener la descripción de la propiedad de longitud `(L)` del conjunto de parámetros, que se incluye en el mensaje de la excepción para proporcionar más información sobre los valores de los parámetros.

```
    def __get_orthogonal_array(self):
        for key, value in self.__orthogonal_arrays.items():
            if value['levels'] == self.__max_level and self.
                __num_factors <= value['factors']:
                 return key, value['array']

        raise ValueError(f'No hay arreglo ortogonal para {self.
            __err_get_L()}')
```

4. **Función** `__err_get_L(self)`: Este método calcula la propiedad de longitud `(L)` para un conjunto de parámetros dados. Esta propiedad `L` se refiere a la cantidad de valores que pueden tomar cada parámetro en el conjunto de prueba.

   Flujo de la función:

   a) Se inicializa un diccionario vacío llamado `length_counts`, que se utilizará para almacenar el conteo de la longitud de los valores de los parámetros.

   b) Se recorre el diccionario de parámetros `__parameters` utilizando un bucle `for`. Cada iteración del bucle obtiene el valor de un parámetro, pero no utiliza la clave (nombre del parámetro). La variable `length` se establece en la longitud de los valores del parámetro actual.

   c) Se actualiza el diccionario `length_counts` contando la cantidad de veces que se encuentra cada longitud `(length)` en los valores de los parámetros. Si la longitud ya existe en el diccionario, se incrementa su conteo en 1; de lo contrario, se agrega una nueva entrada al diccionario con una cuenta inicial de 1.

   d) Se inicializa una cadena `L` con el valor `'L( '`. Esta cadena se utilizará para construir la descripción de la propiedad de longitud `(L)`.

   e) Se recorre el diccionario `length_counts` utilizando otro bucle `for`, esta vez para obtener las longitudes `(length)` y los conteos correspondientes. En cada iteración, se agrega una parte de la cadena `L` que representa la longitud y el conteo en el formato `'Length ^Count'`.

f) Finalmente, se devuelve la cadena `L` completa, que representa la descripción de la propiedad de longitud `(L)` en términos de las longitudes y conteos de los valores de los parámetros. La cadena tiene el formato `'L( Length1 ^Count1 Length2 ^Count2 ... )'`, donde `'LengthX'` representa la longitud de los valores que puede tomar un parámetro y `'CountX'` es la cantidad de veces que se encuentra esa longitud en los valores de los parámetros.

```
def __err_get_L(self):
    length_counts = {}
    for _, value in self.__parameters.items():
        length = len(value)
        length_counts[length] = length_counts.get(length, 0) + 1

    L = 'L( '
    for length, count in length_counts.items():
        L = L + f'{length}^{count} '
    return L + ')'
```

5. **Función** `build_test_cases(self)`: Es el método principal de la clase `OrthogonalArray`. Su objetivo es construir un conjunto de casos de prueba utilizando un conjunto ortogonal previamente seleccionado.

   Flujo de la función:

   a) Se llama al método privado `__get_orthogonal_array()` para obtener el conjunto ortogonal adecuado para los parámetros proporcionados. Este método devuelve dos valores: `L` (la propiedad de longitud) y `array` (el conjunto ortogonal).

   b) Se crea un nuevo diccionario `sorted_parameters` que contiene los mismos elementos que `__parameters` (los parámetros proporcionados), pero ordenados por la longitud de los valores en orden descendente. Esto se hace utilizando la función `sorted()` y especificando la función `lambda key=lambda x: len(x[1])` que indica que se debe usar la longitud de los valores como criterio de ordenamiento.

   c) Se crea una lista `keys` que contiene las claves de `sorted_parameters` en el orden en que fueron ordenadas.

   d) Se obtienen los tamaños de las dimensiones del conjunto ortogonal `array`. La variable `N` representa el número de filas y `M` representa el número de columnas.

   e) Se crea una lista bidimensional `test_cases` inicializada con ceros. Esta lista tendrá la misma forma que el conjunto ortogonal, donde cada elemento representa un caso de prueba.

   f) Se realiza un bucle `for` sobre las columnas del conjunto ortogonal `(array)`. Para cada columna, se obtiene la clave del parámetro correspondiente desde `keys`.

g) Se obtienen los valores asociados a esa clave (`key`) en el diccionario `sorted_parameters`.

h) Se crea una cola (`queue`) copiando los valores de la clave, que se utilizará para obtener valores de parámetros en caso de que los índices en el conjunto ortogonal sean mayores que la cantidad de valores disponibles.

i) Se realiza un bucle `for` sobre las filas del conjunto ortogonal. Para cada fila, se obtiene el índice correspondiente desde `array` y se le resta 1 para obtener el índice base 0.

j) Si el índice es mayor o igual a 0 y menor que la cantidad de valores en la lista de la clave (`values`), se asigna el valor correspondiente a `test_cases[row][column]`.

k) Si el índice es mayor o igual a la cantidad de valores en la lista de la clave (`values`), se obtiene el primer valor de la cola (`queue`) y se asigna a `test_cases[row][column]`. Luego, se mueve ese valor al final de la cola para que esté disponible para futuras asignaciones.

l) Al final del proceso, se devuelve un diccionario que contiene la propiedad de longitud (`L`), las claves de los parámetros (`keys`), y el conjunto de casos de prueba (test_cases).

```
    def build_test_cases(self):
        L, array =  self.__get_orthogonal_array()

        sorted_parameters = dict(sorted(self.__parameters.items(), key
            =lambda x: len(x[1]), reverse=True))

        keys = list(sorted_parameters.keys())
        N, M = len(array), len(keys)
        test_cases = [[0 for _ in range(M)] for _ in range(N)]

        for column, key in enumerate(keys):
            values = sorted_parameters[key]
            queue = values.copy()
            for row in range(N):
                idx = array[row][column] - 1
                value = None
                if -1 < idx < len(values): value = values[idx]
                else: value = queue[0]; queue = queue[1:] + [queue[0]]
                test_cases[row][column] =  value

        return {
            'L' : L,
            'keys' : keys,
            'array' : test_cases
        }
```

Capítulo 6

Desarrollo de la aplicación web

Este capítulo se centra en desarrollo de la aplicación web, abarcando tanto el desarrollo del backend como del frontend. Se describen las tecnologías utilizadas en cada parte, como Python y FastAPI en el backend, y Next.js de React en el frontend. También se mencionan las ventajas de estas tecnologías, como su alta velocidad de ejecución, facilidad de uso y documentación completa.

Además, se hace énfasis en el diseño de la interfaz de usuario de la aplicación. Se destaca la importancia de crear una interfaz intuitiva y estética para captar la atención de los usuarios. Se mencionan herramientas utilizadas, como Bootstrap, que facilitan la creación de interfaces atractivas y responsivas.

6.1. Tecnologías

6.1.1. Backend

Para el desarrollo del backend, se utilizó `Python 3.8` [Fou23a] y la biblioteca `FastAPI` [Ram23] para crear la API. Elegimos `FastAPI` debido a sus numerosas ventajas. En primer lugar, FastAPI es conocido por su alta velocidad de ejecución, lo que lo convierte en una opción ideal para aplicaciones de alto rendimiento. Además, `FastAPI` ofrece una sintaxis sencilla y declarativa, lo que facilita la creación de endpoints y la definición de los modelos de datos. También cuenta con una documentación clara y extensa, lo que facilita el desarrollo y la colaboración en equipo.

Además, para almacenar las claves API, se utilizó `PostgreSQL` [Gro23], una base de datos relacional. `PostgreSQL` debido a su confiabilidad, capacidad de escalabilidad y fácil uso. Proporciona un rendimiento sólido y una amplia gama de características avanzadas que son útiles para gestionar los datos de manera eficiente y segura.

6.1.2. Frontend

En el desarrollo del frontend, se utilizó el framework `Next.js` [Rau23] de React. `Next.js` ofrece varias ventajas significativas, como la posibilidad de generar páginas estáticas o dinámicas, el enrutamiento fácil de usar y una configuración simplificada. `Next.js` también proporciona un enfoque eficiente para el manejo del estado y una experiencia de desarrollo más rápida mediante la recarga en caliente *(hot reloading)*, lo que permite ver los cambios en tiempo real durante el desarrollo sin necesidad de recargar la página. Además, se utilizó Bootstrap [MO], un framework de diseño `CSS`

ampliamente adoptado, que proporciona un conjunto completo de estilos y componentes predefinidos, facilitando la creación de la interfaces atractivas y responsivas, reduciendo así el tiempo de desarrollo y mejorando la consistencia visual del proyecto.

6.2. Diseño de la interfaz de usuario

En la fase de diseño de la interfaz de usuario de la aplicación se busca que los usuarios que lleguen a la aplicación no solo observen el buen trabajo que se hizo con la implementación de las técnicas y pueda darles uso, si no que también darles una interfaz de usuario con la que se sientan motivados a aprender más acerca de como funcionan las pruebas de software y esto se logra con un buen diseño y una aplicación intuitiva y estética. Ya que se ha demostrado en un estudio que evalúa el impacto de una buena interfaz de usuario y un buen diseño de la experiencia de usuario que esta influye en la atención y satisfacción del usuario dentro de la aplicación. Este articulo nos advierte que " El incumplimiento de los aspectos de calidad de la interfaz de usuario puede dar lugar a insatisfacción, malentendidos o uso inadecuado, lo que se traduce en críticas y un bajo nivel de aceptación en las plataformas de aprendizaje" [VH20], y esto es justo lo que no queremos que pase dentro de la aplicación, que el usuario se sienta frustrado o perdido al no saber como utilizarla y es por eso que nuestro objetivo va a ser brindar al usuario una interfaz intuitiva con la que se sienta cómodo.

Para la parte del diseño de la interfaz de usuario de la herramienta de reducción de casos de prueba la intención principal es que esta sea fácil de utilizar para cualquier usuario sin importar su nivel de experiencia tecnológica. Nuestro enfoque se basa en garantizar que la interfaz de la herramienta sea intuitiva, amigable y accesible con cualquier usuario brindándole a este una experiencia fluida y satisfactoria. Buscamos que los usuarios puedan comprender fácilmente el funcionamiento de cada componente solo a primera vista o con ver una vez el manual de usuario; esto lo lograremos utilizando elementos visuales reconocibles e iconos representativos, además de elementos que son familiares para cualquier usuario que ya se usen dentro de otras herramientas web y estas estando organizadas dentro de la herramienta de una forma lógica para facilitar la interacción y navegación por parte del usuario.

6.2.1. Página principal

En la figura 6.1 se observa la página principal de nuestra herramienta en la cual buscamos un diseño minimalista que tenga las funcionalidades principales y mantener la atención en lo importante, además de tener las opciones diferenciales en un plano principal en la aplicación junto con textos los cuales sirven de guía y ayudan al usuario para que sepa el objetivo de la herramienta y sus funcionalidades principales. Con este diseño nos aseguramos de que el usuario no se abrume con mucho texto o muchos botones y lo mantenemos en pocas opcionalidades que son claras y concisas lo cual guía al usuario a tomar la decisión deseada.

Figura 6.1: Página principal

6.2.1.1. Tarjetas de contenido

En la figura 6.2 podemos ver como se muestran cada una de las técnicas que maneja la herramienta, en las cuales se utilizaron tarjetas de contenido para así lograr un diseño de interfaz más organizado y estructurado, además de la familiaridad de los usuarios con estas; cada tarjeta contiene una imagen que identifica cada técnica además de una breve descripción de que realiza cada técnica de reducción de casos de prueba. El uso de estas nos proporciona un diseño estético además de proveer un diseño responsivo para que en un futuro se puedan usar las herramientas en diferentes dispositivos.

Figura 6.2: Tarjetas de contenido

6.2.2. Partición de equivalencia

Para la interfaz de particiones de equivalencia que se puede observar en la figura 6.3 nos basamos en el estilo minimalista y sencillo que manejamos en la página de inicio y adaptamos esta vista a los requisitos necesarios para mostrar la técnica de la mejor manera posible para el usuario sin que hayan confusiones. En la interfaz se puede ver los botones que nos llevan al inicio y a la vista de "acerca de" para posteriormente mostrar el titulo de la técnica que estamos utilizando y una breve descripción de la misma para que el usuario tenga un mayor conocimiento sobre como este va a funcionar. Para la parte de la funcionalidad se tienen 3 campos los cuales representan la información mínima que debe tener para que funcione el código que implementa la técnica, es por esto que tenemos los campos de Parámetro donde se ingresa la variable a probar, después se define la partición de equivalencia y posteriormente se define el representante de esta clase de equivalencia para finalmente definir si la partición es vaálida o inválida.

Figura 6.3: Interfaz partición de equivalencias

6.2.2.1. Agregar parámetro

En la figura 6.4 se puede observar la opción que permite agregar parámetros a la tabla de clases de equivalencia. Cada parámetro añadido representa una característica relevante y por defecto crea una nueva clase de equivalencia. Es importante asignar a cada parámetro un nombre claro y descriptivo que represente adecuadamente la característica que se está evaluando.

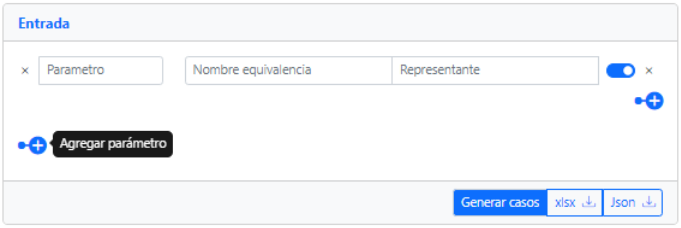


Figura 6.4: Agregar parámetro partición de equivalencias

6.2.2.2. Eliminar parámetro

En la figura 6.5 se puede identificar la opción que permite eliminar un parámetro específico de la tabla de clases de equivalencia. Al eliminar un parámetro, todas sus clases de equivalencia asociadas también son eliminadas.

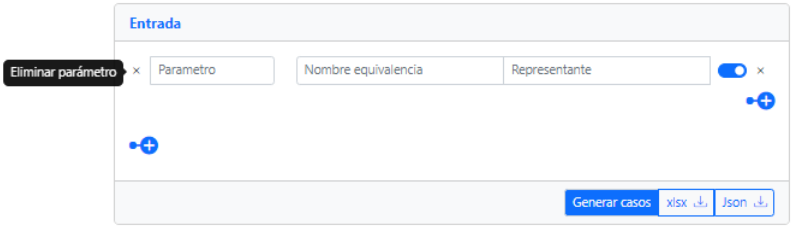


Figura 6.5: Eliminar parámetro partición de equivalencias

6.2.2.3. Agregar clase de equivalencia

En la figura 6.6 se puede identificar la opción de agregar clase de equivalencia la cual está disponible para cada parámetro de la tabla y permite agregar nuevas clases de equivalencia al parámetro en cuestión. Cada clase de equivalencia se define por un nombre descriptivo y un representante que ejemplifica las características comunes de dicha clase. Por ejemplo, supongamos que tenemos un parámetro llamado “ monto”. Utilizando la opción de agregar clase de equivalencia, podríamos crear una clase llamada “ Entero mayor o igual a 1000” y establecer 1000 como su representante.

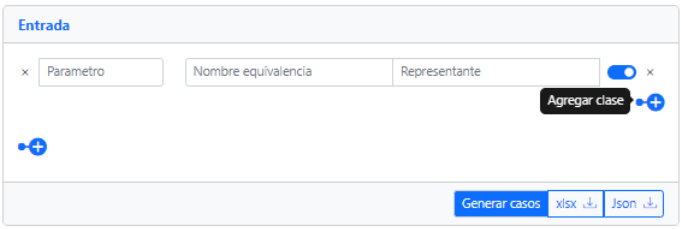


Figura 6.6: Agregar clase de equivalencia

6.2.2.4. Eliminar clase de equivalencia

La opción que se observa en la figura 6.7 permite eliminar una clase de equivalencia específica asociada a un parámetro en la tabla. Si la clase de equivalencia que se elimina es la única asociada a un parámetro, el parámetro completo también se elimina. Antes de utilizar esta opción, es importante considerar las implicaciones y asegurarse de que la eliminación de la clase de equivalencia no afecte negativamente la cobertura y representatividad de las entradas de prueba.

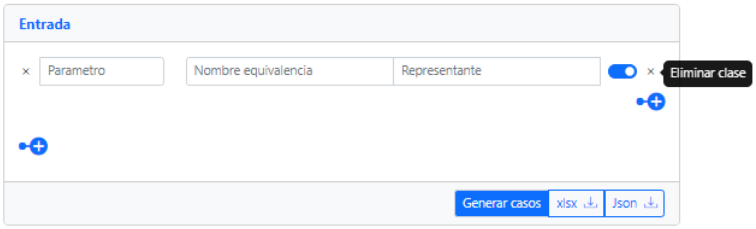


Figura 6.7: Eliminar clase de equivalencia

6.2.2.5. Establecer clase

En la figura 6.8 se puede observar la opción que permite marcar una clase de equivalencia como válida o inválida dentro de un parámetro en la tabla. Proporciona flexibilidad al definir restricciones y condiciones específicas que afectan la validez de cada clase de equivalencia en la prueba de caja negra. Por ejemplo, en el parámetro "monto", se puede establecer una clase de equivalencia llamada "Valor negativo" con un representante de -1000 y marcarla como inválida. Esto permite precisión al evaluar las entradas de prueba y sus resultados asociados.

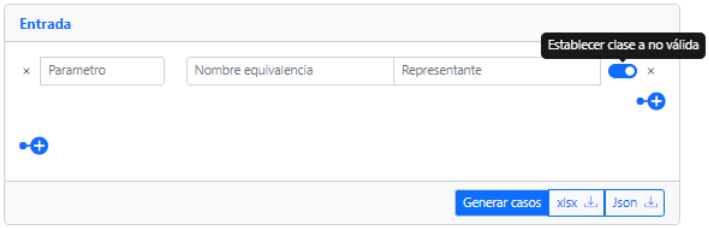


Figura 6.8: Establecer clase de equivalencias

6.2.3. Análisis de valores al límite

Para la interfaz de la técnica de análisis de valores al límite se optó por no perder la estética de las demás páginas de la aplicación y se adaptó esta a los requerimientos del código que implementa esta técnica como se puede observar en la figura 6.9. Así que en en interfaz tenemos el nombre de la técnica acompañado de una pequeña definición y en la parte funcional tenemos la variable la cual se va a probar, la función lambda que es la que nos da el rango y las restricciones a los cuales vamos a aplicar la técnica, además de un campo para definir el paso; todos estos campos son requeridos por la aplicación para aplicar la técnica y lanzar el resultado esperado al usuario.

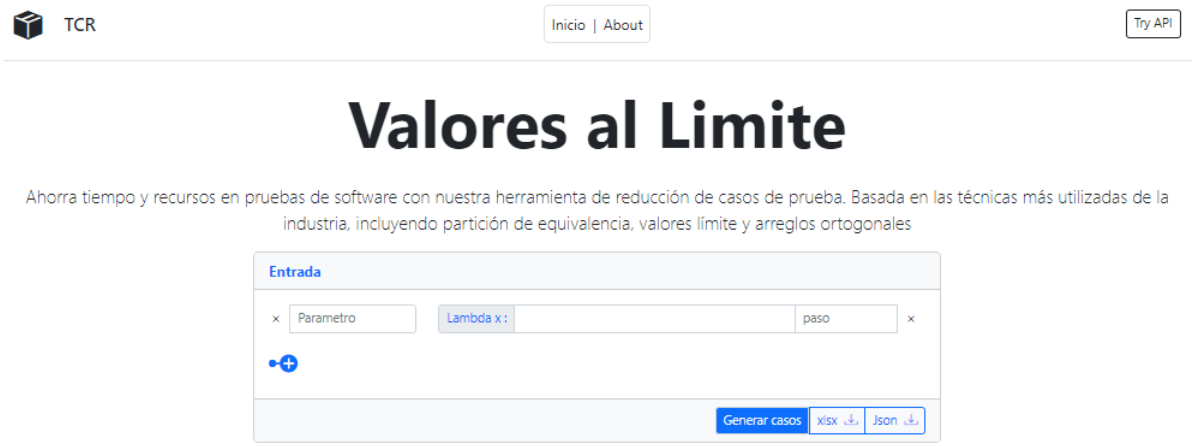


Figura 6.9: Interfaz Valores al límite

6.2.3.1. Agregar parámetro

La opción que se observa en la figura 6.10 permite agregar un nuevo parámetro a la tabla. Se recomienda que se asigne un nombre claro y descriptivo al nuevo parámetro para facilitar su comprensión y uso. Por defecto, se agrega un valor "lambda" como representante del parámetro recién creado.

Figura 6.10: Agregar parámetro valores al límite

6.2.3.2. Eliminar parámetro

En la figura 6.11 se puede observar la opción que permite eliminar un parámetro específico de la tabla, junto con su valor lambda o las clases de equivalencia asociadas a dicho parámetro.

Figura 6.11: Eliminar parámetro valores al límite

6.2.3.3. Editar incremento

En la figura 6.12 se ve la opción que permite establecer el incremento o paso de x en la generación de valores de prueba. Al definir un paso específico, como 10, se generarán valores de prueba que sigan ese incremento dentro de un rango predefinido, como $x \geq 10$ y $x \leq 100$. Esto te brinda el control para seleccionar valores clave o puntos de interés dentro del rango y ajustar la cobertura de prueba según tus necesidades. El incremento de x es opcional y te permite obtener una cobertura óptima de los valores de prueba al ajustar el paso según los requisitos de prueba y la resolución necesaria.

Figura 6.12: Editar el incremento en valores al límite

6.2.3.4. Eliminar lambda

En la figura 6.13 se puede observar la opción que permite eliminar el valor “lambda” por defecto asociado a cada parámetro en la tabla. Al eliminar el lambda, se habilitan dos botones: “Agregar lambda” y “Agregar clases de equivalencia”. Estas opciones te permiten establecer un nuevo valor lambda o crear clases de equivalencia específicas en lugar del valor lambda predeterminado.

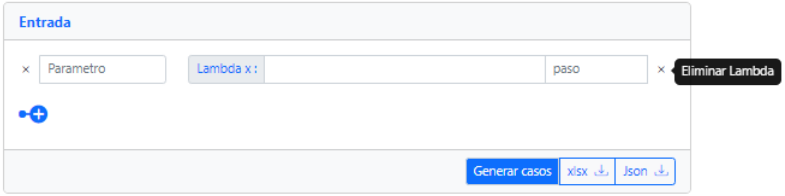


Figura 6.13: Eliminar lambda en valores al límite

6.2.3.5. Agregar lambda

La opción que se observa en la figura 6.14 permite agregar un valor lambda a un parámetro específico en la tabla. Puedes definir una función lambda válida sintácticamente que esté dentro de un rango y que incluya restricciones adicionales. Esto te permite establecer condiciones más complejas para la generación de valores de prueba. Por ejemplo, podrías agregar un lambda complejo como x >= 1 and x <= 250 and x % 2 == 0 and x not in [20, 50]. Este lambda establece múltiples condiciones, como que x debe ser mayor o igual a 1, menor o igual a 250, un número par y no puede ser igual a 20 o 50.

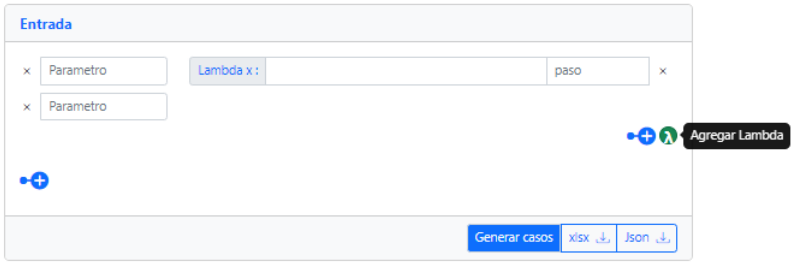


Figura 6.14: Agregar lambda en valores al límite

6.2.4. Arreglos ortogonales

En la figura 6.15 se puede observar la interfaz de la técnica de arreglos ortogonales, en la cual se ha tomado la decisión de mantener la estética de las demás interfaces de la aplicación y adaptarla a los requisitos del código que implementa esta técnica. En esta interfaz, se muestra el nombre de la técnica junto con una breve definición. En la parte funcional, se presentan los parámetros necesarios para ejecutar la técnica de arreglos ortogonales, los cuales incluyen el nombre de las diferentes variables y los diferentes valores que puede tomar cada una de estas variables. Además de poder editar el número de variables y de valores que pueden tomar dichos parámetros dentro de la aplicación.Todos estos campos son requeridos por la aplicación para aplicar la técnica y proporcionar el resultado esperado al usuario.

Figura 6.15: Interfaz Arreglos ortogonales

6.2.4.1. Agregar parámetro

En la figura 6.16 se puede obervar la opción permite agregar un nuevo parámetro a la tabla. Al seleccionar esta opción, se crean automáticamente filas específicas para ese nuevo parámetro, basadas en el número de filas existentes en la tabla en ese momento.

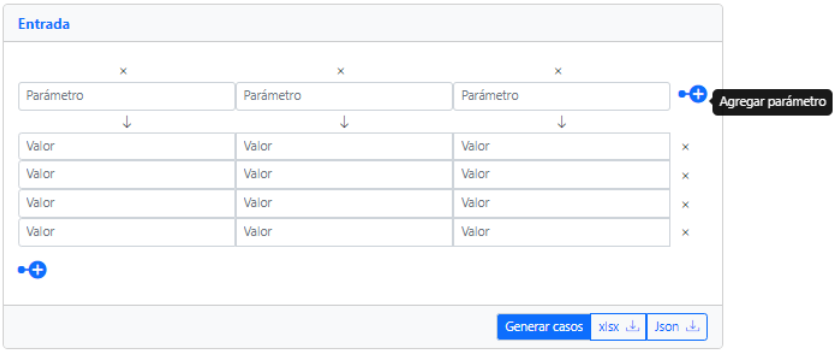


Figura 6.16: Agregar parámetro arreglos ortogonales

6.2.4.2. Eliminar parámetro

En la figura 6.17 se puede observar la opción que permite eliminar un parámetro específico de la tabla. Al seleccionar esta opción, se eliminará el parámetro junto con todas las columnas asociadas a él.

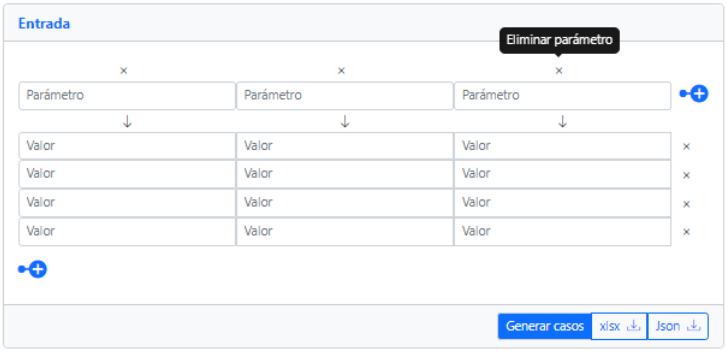


Figura 6.17: Eliminar parámetro arreglos ortogonales

6.2.4.3. Agregar fila de Valores

La opción que se ve en la figura 6.18 permite agregar una fila de valores para cada uno de los parámetros actuales en la tabla. Al seleccionar esta opción, puedes ingresar un conjunto de valores que representen una combinación específica de niveles para los factores de prueba.

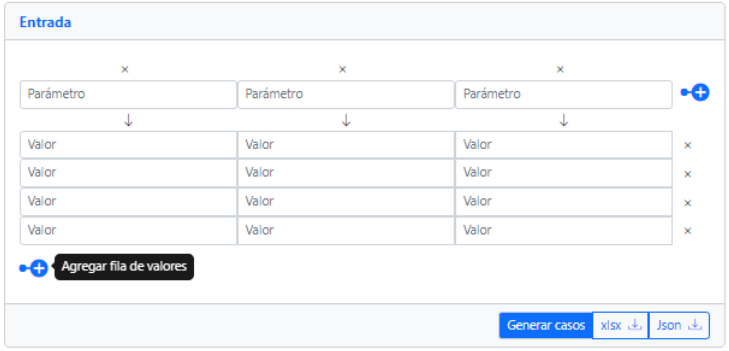


Figura 6.18: Agregar valor arreglos ortogonales

6.2.4.4. Eliminar fila de valores

En la figura 6.19 se puede ver la opción que permite eliminar una fila específica de la tabla, lo cual implica la eliminación de los valores asociados a cada uno de los parámetros en esa fila.

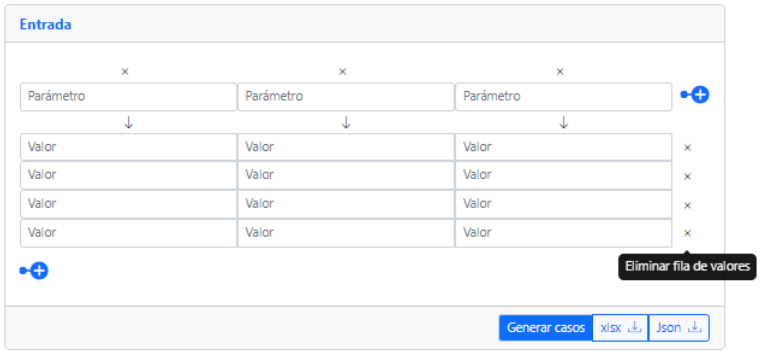


Figura 6.19: Eliminar valor arreglos ortogonales

6.3. Presentación de resultados

Cuando el usuario utiliza la aplicación de reducción de casos de prueba, este con el previo conocimiento que tiene sobre el software que se va a probar rellena todos los campos requeridos por la aplicación para generar los casos de prueba óptimos para posteriormente probar su software, pero ¿cómo se muestran estos datos al usuario? si no se tiene en la interfaz un lugar donde aparecen los caos de prueba ¿Donde puede ver el usuario los casos de prueba generados por la aplicación?. Es ahí donde en la aplicación se implementa el módulo de presentación de los resultados que es donde se recopilan los casos de prueba que genera cada técnica desde la lógica de la aplicación y se muestran al usuario final para que pueda hacer uso de estos en la fase de pruebas de su software; basados en esta problemática se investigaron los diferentes formatos los cuales podrían ser presentados los datos y estos son:

- Texto plano
- xlsx
- JSON
- HTML

Los formatos que se eligieron para presentar los datos fueron el Json y el Excel. Estos son los formatos de presentación más conocidos y que se utilizan más dentro del mundo de testing de software ya que además de ser prácticos son útiles en cuanto a visualización y entendimiento de los mismos, ademas de ser formatos con los que se complementa el dearrollo de la aplicaición.

6.3.1. Presentación de datos por medio de XLSX

El formato xlsx almacena datos en una estructura tabular, donde las celdas se organizan en filas y columnas. Cada celda puede contener diferentes tipos de datos, como números, texto, fechas o fórmulas. es un tipo de archivo utilizado para almacenar y manipular datos en hojas de cálculo, siendo específicamente asociado con el software Microsoft Excel y es muy útil ya que comparado con los demás formatos esta tiene el factor de que las personas esta mas familiarizadas al utilizar excel y sus funciones, además de que la mayoría de casos de prueba manuales se planean en este mismo formato. Entonces para obtener los casos de prueba que lanza la aplicación se deben llenar los campos con valores válidos y se da clic en generar casos y xlsx.

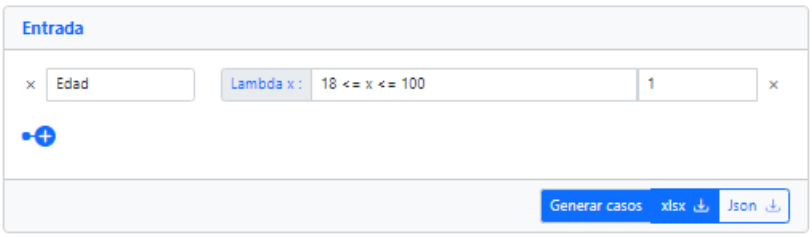


Figura 6.20: Generar resultados xlsx

Cuando generamos los resultados con la aplicación, estos se descargan de forma automática a nuestro computador como una archivo xlsx con los casos de prueba óptimos y listos para probar el software. Lo que se genera dentro de este archivo xlsx son dos hojas, la primera que se puede ver en el cuadro 6.6 tiene los casos de prueba válidos y la segunda que se puede ver en el cuadro 6.7 con los casos de prueba inválidos.

Edad	
clase_equivalencia	representante
primer_valor_minimo	18
segundo_valor_minimo	19
valor_medio	59
primer_valor_maximo	99
segundo_valor_maximo	100

Cuadro 6.6: Generar resultados válidos en el xlsx

Edad	
clase_equivalencia	representante
valor_minimo_invalido	17
valor_maximo_invalido	101

Cuadro 6.7: Generar resultados inválidos en el xlsx

6.3.2. Presentación de datos por medio de JSON

El formato JSON o JavaScript Object Notation [jso23] es un formato que se utiliza comúnmente para transmitir y almacenar información estructurada que en este caso se va a utilizar parra guardar casos de prueba; algunas ventajas que nos hicieron escoger este formato por encima de los otros es que esta es una estructura simple y ligera lo que la hace eficiente para transmitir y mostrar datos; además de que en el paso de la generación de casos de prueba es mas eficiente que otros formatos por su estructura de datos simple basada en pares clave-valor. Entonces para obtener los casos de prueba que lanza la aplicación se deben llenar los campos con valores validos y se da clic en generar casos y JSON.

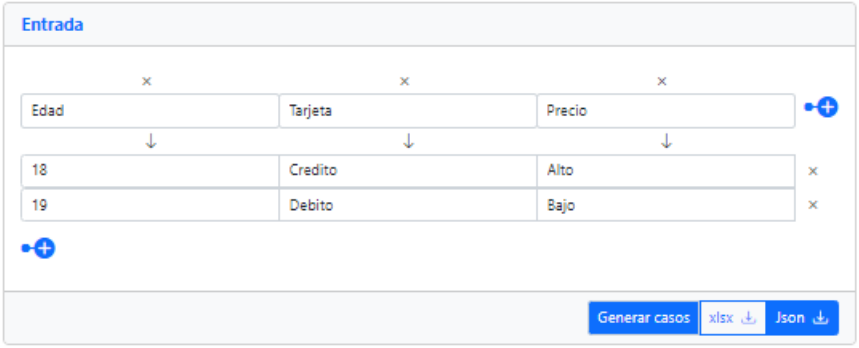


Figura 6.21: Generar resultados en JSON


```
{
    "error": false,
    "tecnica": "AO",
    "casos-pruebas": {
        "L": "L4",
        "keys": ["Edad", "Tarjeta", "Precio"],
        "array": [
            ["18", "Credito", "Alto"],
            ["18", "Debito", "Bajo"],
            ["19", "Credito", "Bajo"],
            ["19", "Debito", "Alto"]
        ]
    },
    "tiempo-transcurrido": "0.00005"
}
```


Figura 6.22: Generar resultados válidos e inválidos en JSON

Como podemos ver en la figura 6.22 se generan los casos de prueba en formato JSON con las variables y valores que se pueden ver el la figura 6.21. Además de esto nos da información adicional acerca de la ejecución del algoritmo como pueden ser que técnica se utilizó, en este caso que arreglo se utilizó y tiempo de ejecución, todo esto esta en la generación de resultados junto con los casos de prueba.

6.4. Link página web

A continuación, se proporciona el enlace a la página web correspondiente:
Test case reducer.

Capítulo 7

Evaluación de la aplicación web

En este capítulo se presenta la evaluación de la aplicación web desarrollada. Se detalla el plan de pruebas, que tiene como objetivo garantizar la precisión de los resultados, verificar la cobertura de sentencia de los algoritmos y asegurar el manejo adecuado de entradas invélidas. Se describen los casos de prueba definidos para cada técnica implementada, incluyendo pruebas con parémetros vélidos e invélidos, valores None y entradas vacías.

Ademés, se aborda la cobertura de sentencia, que es una métrica utilizada para evaluar qué porcentaje de las sentencias de código ha sido ejecutado durante las pruebas. Se utiliza la herramienta Coverage para calcular la cobertura de cada técnica, buscando alcanzar un 100 % de cobertura de sentencia.

7.1. Plan de pruebas

7.1.1. Objetivo y Alcance

El objetivo de la evaluación es garantizar la precisión de los resultados, verificar la cobertura de sentencia de los algoritmos y asegurar el manejo adecuado de entradas inválidas. Durante la evaluación, se pondrán a prueba las funcionalidades técnicas de la aplicación web para garantizar su correcto funcionamiento.

7.1.2. Casos de prueba

Se han definido diversos casos de prueba (funciones con diferentes parámetros) para verificar el funcionamiento de la API y las técnicas implementadas. Estos casos incluyen pruebas con parámetros válidos e inválidos, valores None, entradas vacías, entre otros, con el objetivo de cubrir la mayor cantidad de escenarios posibles. A continuación, se presentan en los cuadros 7.5, 7.6 y 7.7 las funciones correspondientes a cada caso de prueba, junto con su descripción y resultado.

Función	Descripción	Resultado
test parámetros válidos	Se prueba con entradas válidas	OK
test parámetros inválidos	Se prueba cuando los parámetros son inválidos	OK
test parámetro inválido válido es none	Se prueba cuando el valor de ”válido” es ”None”.	OK
test parámetro inválido representate es none	Se prueba cuando el valor ”representante” es ”None”.	OK

test parámetro inválido clave none	Se prueba cuando la clave de una clase de equivalencia es ”None”.	OK
test generar casos de pruebas salida esperada 1	Se prueba con 2 parámetros, cada uno con 3 clases de equivalencia, todas las cuales son válidas. El resultado esperado son 9 casos de pruebas válidos y 0 inválidos.	OK
test generar casos de pruebas salida esperada 2	Se prueba con 2 parámetros, el primer parámetro tiene 2 clases de equivalencias validas y una invalida, el segundo parámetro tiene 2 clases de equivalencias invalidas y una valida. El resultado esperado son 2 pruebas validas y 3 pruebas invalidas.	OK
test generar casos de pruebas salida esperada 3	Se prueba con 3 parámetros, cada uno con 3 clases de equivalencia, el primer parámetro tiene 2 validas y 1 invalida, el segundo 2 invalidas y una valida, el tercero dos validas y 1 invalida. El resultado esperado son 4 pruebas validas y 4 invalidas.	OK
test generar casos de pruebas salida esperada 4	Se prueba con 1 solo parámetro, dicho parámetro tiene 1 clase de equivalencia valida y una invalida. El resultado esperado son 1 caso válido y 1 caso inválido.	OK
test generar casos de prueba cantidad esperada 1	Se prueba con 3 parámetros, cada uno con 3 clases de equivalencias validas. El resultado esperado son 27 casos válidos y 0 inválidos.	OK
test generar casos de prueba cantidad esperada 2	Se prueba con 3 parámetros, cada uno con tres clases de equivalencia. El primer parámetro con dos clases validas y una invalida, el segundo con 1 valida y 2 invalidas, el tercero con dos validas y 1 invalida. El resultado esperado son 4 casos válidos y 4 casos inválidos.	OK
test generar casos de prueba cantidad esperada 3	Se prueba con 3 parámetros, cada uno con tres clases de equivalencia. El primer parámetro con dos clases invalidas y una valida, el segundo con 1 valida y dos invalidas, el tercero con 2 invalidas y 1 invalida. El resultado esperado son 4 casos válidos y 4 casos inválidos.	OK

Cuadro 7.8: Casos de pruebas para la técnica partición de equivalencia (unittest)

Función	Descripción	Resultado

test parametros validos	Se prueba con parámetros válidos. El resultado esperado son los límites de cada parámetro.	OK
test generar limites 1	Se prueba con 1 parámetro válido, es decir, un lambda y un delta bien definidos. El resultado esperado es un caso de prueba con 7 posibles valores para dicho lambda.	OK
test generar limites 3	Se prueba con 3 parámetros válidos, cada uno bien definido. El resultado esperado son 7 posibles valores para cada parámetro.	OK
test generar limites con particion de equivalencias	Se prueba con 3 parámetros válidos, cada uno bien definido. El resultado esperado son 7 posibles valores para cada parámetro. Además, se utiliza la técnica de partición de equivalencias para generar los casos de prueba.	OK
test generar limites timelimit	Se prueba un parámetro en el cual se tiene bien definido el lambda, sin embargo, tomará mucho tiempo en calcular los valores válidos e inválidos.	OK
test parametro invalido delta none	Se prueba con un parámetro que tiene bien definido el lambda, pero el delta es None. El resultado esperado es una excepción.	OK
test parametro invalido lambda incorrecto	Se prueba con un parámetro que tiene bien mal definido el lambda, pero el delta esta bien definido. El resultado esperado es una excepción.	OK
test parametro invalido lambda size	Se prueba con un parámetro el cual tiene un lambda muy pequeño, por ejemplo: x >0, el cual es inválido ya que debe estar acotado. El resultado esperado es una excepción.	OK
test parametro invalido lambda syntaxError	Se prueba con un parámetro que tiene un error de sintaxis en el lambda. El resultado esperado es una excepción.	OK
test parametro invalido lambda type	Se prueba con un parámetro que tiene un lambda de tipo diferente a string. El resultado esperado es una excepción.	OK
test parametros invalidos 1	Se prueba con parámetros inválidos, donde la entrada está mal definida (solo lambda). El resultado esperado es una excepción.	OK
test parametros invalidos 2	Se prueba con parámetros inválidos, donde la entrada está mal definida (solo delta). El resultado esperado es una excepción.	OK

Cuadro 7.9: Casos de pruebas para la tćenica análisis de valores limite (unittest)

Función	Descripción	Resultado
test array L11 1	Se prueba con 8 factores y cada uno de ellos con 2 niveles. El resultado esperado es un arreglo L11.	OK
test array L4 1	Se prueba con 3 factores o parámetros, con un máximo de 2 posibles valores por factor para dos parámetros. El resultado esperado es un arreglo L4.	OK
test array L4 2	Se prueba con 3 factores o parámetros, con un máximo de 2 posibles valores por factor para cada uno. El resultado esperado es un arreglo L4.	OK
test array L8 1	Se prueba con 5 factores y cada uno de ellos con 2 niveles. El resultado esperado es un arreglo L8.	OK
test array L8 2	Se prueba con 7 factores y cada uno de ellos con 2 niveles. El resultado esperado es un arreglo L8.	OK
test array L9 1	Se prueba con 4 factores y cada uno de ellos con 3 niveles. El resultado esperado es un arreglo L9.	OK
test arreglo ortogonal fuera de rango	Se prueba con una serie de factores y niveles que no tiene un L definido.	OK
test keys none	Se prueba con 2 factores bien definidos y uno mal definido, como None. El resultado esperado es una excepción.	OK
test valores vacios	Se prueba con 3 factores o parámetros y una lista de posibles valores vacía. El resultado esperado es una excepción.	OK
test values none 1	Se prueba con 2 factores bien definidos y uno mal definido, como None. El resultado esperado es una excepción.	OK
test values none 2	Se prueba con 3 factores pero con sus posibles valores en None. El resultado esperado es una excepción.	OK

Cuadro 7.10: Casos de pruebas para la técnica arreglos ortogonales (unittest)

7.1.3. Cobertura

La cobertura de sentencia (statement coverage) es una métrica utilizada en las pruebas de software para evaluar qué porcentaje de las sentencias o líneas de código de un programa ha sido

ejecutado durante las pruebas. El objetivo de la cobertura de sentencia es asegurar que todas las líneas de código se hayan ejecutado al menos una vez, lo que indica que todas las instrucciones han sido visitadas durante las pruebas.

En el presente proyecto se utilizó la herramienta Coverage para llevar a cabo la cobertura de las técnicas. El objetivo establecido fue lograr una cobertura del 100 % de sentencia para cada una de ellas. Esto significa que se buscó ejecutar todas las líneas de código de las técnicas durante las pruebas, sin dejar ninguna sin probar.

Ejemplo:
Supongamos que tenemos una función llamada *divide_ entre_ cinco* que toma un número como argumento y devuelve True si el número es divisible entre 5, y False en caso contrario:

```
def divide_entre_cinco(numero):
    if numero % 5 == 0:
        return True
    else:
        return False
```

Ahora, podemos escribir pruebas para esta función y calcular su cobertura de sentencia utilizando la biblioteca de pruebas unittest de Python:

```
import unittest

class PruebaDivideEntreCinco(unittest.TestCase):

    def test_divisible_entre_cinco(self):
        self.assertTrue(divide_entre_cinco(10))

    def test_no_divisible_entre_cinco(self):
        self.assertFalse(divide_entre_cinco(7))

if __name__ == '__main__':
    unittest.main()
```

Si usamos estas dos pruebas, la cobertura de sentencia sería del 100 %. Sin embargo, si omitimos una de ellas, la cobertura sería del 50 %, ya que no habría suficientes casos de prueba para evaluar todo el código.

Dicho esto, en los cuadros 7.11, 7.12 y 7.13 proporcionan detalles adicionales sobre la cobertura obtenida para cada técnica específica. Estos cuadros brindan información sobre qué porcentaje de las sentencias de código de cada técnica se han ejecutado durante las pruebas.

Name	Stmts	Miss	Cover
.../techniques/EquivalencePartition/__init__.py	2	0	100 %
.../techniques/EquivalencePartition/__version__.py	6	0	100 %
.../techniques/EquivalencePartition/equivalencePartition.py	56	0	100 %
testEquivalencePartition.py	91	0	100 %
TOTAL	155	0	100 %

Cuadro 7.11: Cobertura para la ténica partición de equivalencia (coverage)

Interpretación:

- El archivo .../techniques/EquivalencePartition/__init__.py tiene 2 declaraciones en total, ninguna falta (miss) y tiene una cobertura del 100 %.
- El archivo .../techniques/EquivalencePartition/__version__.py tiene 6 declaraciones en total, ninguna falta (miss) y también tiene una cobertura del 100 %.
- El archivo .../techniques/EquivalencePartition/equivalencePartition.py tiene 56 declaraciones en total, ninguna falta (miss) y también tiene una cobertura del 100 %.
- El archivo testEquivalencePartition.py tiene 91 declaraciones en total, ninguna falta (miss) y también tiene una cobertura del 100 %.
- El total de todas las declaraciones en la técnica es de 155, sin ninguna falta (miss) y tiene una cobertura total del 100 %.

Name	Stmts	Miss	Cover
.../techniques/EquivalencePartition/__init__.py	2	0	100 %
.../techniques/EquivalencePartition/__version__.py	6	0	100 %
.../techniques/EquivalencePartition/equivalencePartition.py	56	2	96 %
.../techniques/LimitValueAnalysis/__init__.py	2	0	100 %
.../techniques/LimitValueAnalysis/__version__.py	6	0	100 %
.../techniques/LimitValueAnalysis/limitValueAnalysis.py	106	0	100 %
testLimitValueAnalysis.py	69	1	99 %
TOTAL	247	3	99 %

Cuadro 7.12: Cobertura para la ténica análisis de valores limite (coverage)

Interpretación:

- El archivo ".../techniques/EquivalencePartition/equivalencePartition.py"tiene 56 declaraciones en total, 2 faltas (miss) y tiene una cobertura del 96 %.

- El archivo ".../techniques/LimitValueAnalysis/__init__.py"tiene 2 declaraciones en total, ninguna falta (miss) y tiene una cobertura del 100 %.
- El archivo ".../techniques/LimitValueAnalysis/__version__.py"tiene 6 declaraciones en total, ninguna falta (miss) y tiene una cobertura del 100 %.
- El archivo ".../techniques/LimitValueAnalysis/limitValueAnalysis.py"tiene 106 declaraciones en total, ninguna falta (miss) y tiene una cobertura del 100 %.
- El archivo "testLimitValueAnalysis.py"tiene 69 declaraciones en total, 1 falta (miss) y tiene una cobertura del 99 %. No se obtiene una cobertura total del 100 % ya que, la ténica partición de equivalencia tiene una cobertura del 96 %, afectando el resultado final, sin embargo, esto no es importante, ya que el objetivo es la ténica de valores limites, quien tiene una cobertura del 100 %.
- El total de todas las declaraciones en la técnica es de 257, con una falta (miss) de 3 y tiene una cobertura total del 99 %.

Name	**Stmts**	**Miss**	**Cover**
.../techniques/OrthogonalArray/__init__.py	2	0	100 %
.../techniques/OrthogonalArray/__version__.py	6	0	100 %
.../techniques/OrthogonalArray/arrays.py	11	0	100 %
.../techniques/OrthogonalArray/orthogonalArray.py	48	0	100 %
testOrthogonalArray.py	109	0	100 %
TOTAL	176	0	100 %

Cuadro 7.13: Cobertura para la ténica arreglos ortogonales (coverage)

Interpretación:

- El archivo ".../techniques/OrthogonalArray/__init__.py"tiene 2 declaraciones en total, ninguna falta (miss) y tiene una cobertura del 100 %.
- El archivo ".../techniques/OrthogonalArray/__version__.py"tiene 6 declaraciones en total, ninguna falta (miss) y tiene una cobertura del 100 %.
- El archivo ".../techniques/OrthogonalArray/arrays.py"tiene 11 declaraciones en total, ninguna falta (miss) y tiene una cobertura del 100 %.
- El archivo ".../techniques/OrthogonalArray/orthogonalArray.py"tiene 48 declaraciones en total, ninguna falta (miss) y tiene una cobertura del 100 %.
- El archivo "testOrthogonalArray.py"tiene 109 declaraciones en total, ninguna falta (miss) y tiene una cobertura del 100 %.

- El total de todas las declaraciones en la técnica es de 176, sin una falta (miss) y tiene una cobertura total del 100 %.

-

7.1.4. Procedimiento de ejecución de pruebas

1. Configuración del entorno de pruebas

   Para la configuración del entorno de pruebas de la aplicación web, se requirió lo siguiente:

   - Un sistema operativo compatible con Python 3 y las bibliotecas necesarias.
   - Python 3 en el sistema.
   - Las dependencias requeridas, como FastAPI.

2. Ejecución de casos de prueba

   Para la ejecución de los casos de prueba, se siguen los siguientes pasos:

   - Se ejecuta el entorno de desarrollo de la aplicación web.
   - Se utiliza el framework de pruebas unittest para definir y ejecutar los casos de prueba.
   - Se usa el comando `python -m unittest discover` para ejecutar automáticamente todos los casos de prueba.
   - Se usa el comando `coverage run -m unittest` para ejecutar automáticamente todos los casos de prueba y por ender generar un reporte de la cobertura, usando el comando `coverage report`.

7.1.5. Herramientas de prueba

- **Unittest** [Fou23b]

  El framework de pruebas unittest proporciona una solución integrada para el diseño y la ejecución de casos de prueba en Python. Permite crear conjuntos de pruebas, definir casos de prueba individuales y realizar aserciones para verificar los resultados.

- **Coverage** [Bat23]

  La herramienta de análisis de cobertura coverage es ampliamente utilizada para medir la cobertura del código durante la ejecución de las pruebas en Python. Permite identificar las partes del código que han sido ejecutadas y las que no, proporcionando información valiosa sobre la efectividad de las pruebas.

7.1.6. Criterios de aceptación

- Los resultados generados por las técnicas implementadas deben ser correctos y coincidir con los resultados obtenidos manualmente.
- Se debe lograr una cobertura del 100 % en las pruebas, asegurando que todas las funcionalidades y casos de uso sean probados.
- Todos los casos de prueba deben ser exitosos y no deben producir errores o fallos en la aplicación web.

7.2. Usuario Final

El objetivo principal de la evaluación de la aplicación web es obtener información precisa y relevante sobre su usabilidad, identificar problemas y áreas de mejora, y recopilar retroalimentación de los usuarios finales para mejorar su experiencia de uso.

7.2.1. Metodología de evaluación

La evaluación se llevó a cabo proporcionando a los usuarios finales la URL de la página y solicitándoles que navegaran en ella sin previo conocimiento. Los evaluadores eran ingenieros de QA con una sólida experiencia en pruebas de software. Se les pidió que exploraran la aplicación web, probaran las diferentes funcionalidades y compartieran sus comentarios y observaciones.

7.2.2. Resultados de la evaluación

Durante la evaluación, los usuarios notificaron que la página era muy minimalista y fácil de navegar. También expresaron que las técnicas implementadas eran claras y sencillas de entender debido a la existencia de una documentación con ejemplos. Los evaluadores apreciaron la simplicidad y la claridad de la interfaz de usuario, lo que facilitaba la interacción con la aplicación.

7.2.3. Sugerencias y recomendaciones

A partir de la evaluación, se recopilaron las siguientes sugerencias y recomendaciones:

- Los usuarios notificaron que, a pesar de la simplicidad de la página, sería importante incluir un modo dark, ya que pasan mucho tiempo frente a la pantalla y el modo light puede afectar la visión. Recomendaron considerar esta funcionalidad para mejorar la experiencia de uso.
- Los evaluadores sugirieron mejorar la API de la aplicación. Específicamente, mencionaron que cada técnica debería tener ejemplos de entrada y salida claros, lo que ayudaría a los usuarios a comprender mejor cómo utilizar la API y qué resultados esperar.

7.3. Ejemplo cuantitativo

En esta sección llevaremos a cabo un ejemplo cuantitativo para poner a prueba la aplicación de reducción de casos de prueba utilizando técnicas de caja negra. El objetivo de este estudio es demostrar cómo estas técnicas pueden ayudar a optimizar la cantidad de casos de prueba necesarios para evaluar la funcionalidad de un sistema o software, sin sacrificar la calidad de las pruebas.

La reducción de casos de prueba es un enfoque eficiente que busca minimizar la cantidad de escenarios de prueba, manteniendo una cobertura adecuada para asegurar que los principales casos de uso se evalúen. Al aplicar las técnicas de caja negra escogidas, nos centramos en probar el sistema desde una perspectiva externa, sin conocer su estructura interna, centrándonos únicamente en las entradas y salidas del usuario.

Para este ejemplo cuantitativo, utilizaremos una interfaz creada por nosotros que simula un sistema de envío internacional de paquetes. El sistema tiene un formulario donde los usuarios pueden especificar los detalles del envío y la información del paquete. Algunos de los campos del formulario deben cumplir con condiciones especiales. Estas condiciones especiales cambian el número de casos de prueba que deben ejecutarse para probar el sistema de envío. En este ejemplo, solo probaremos la interfaz del formulario.

En la figura 7.1 se puede observar la interfaz diseñada y los diferentes campos que posee, la interfaz esta diseñada para que el usuario ingrese detalles relacionados con el envió del paquete como pueden ser peso, precio y dimensiones del paquete, pero además contiene información personal del usuario que envía el paquete e información de el país y ciudad destino del paquete. Cada uno de los campos del formulario que esta en la interfaz posee una serie de restricciones las cuales hacen que hayan más o menos casos de prueba dependiendo de la condición. Estas son las condiciones que se deben de tener en cuenta por cada uno de los campos:

- Para los campos de nombre y apellido no se pueden tener caracteres numéricos, solo letras y no puede dejarse vacío.
- Para el campo de teléfono solo se pueden ingresar caracteres numéricos y especiales como (.,-) y no puede dejarse vació.
- Para el campo del email el cual es opcional se tiene que ingresar en el formato el cual se presenta en el campo.
- Para los campos de país origen, país destino, ciudad origen y ciudad destino solo se pueden escoger entre los 80 países que están disponibles los envíos y para cada país solo hay 10 ciudades en la que los envíos se pueden realizar con éxito.
- Para los campos de peso, precio del paquete, alto, largo y ancho son rangos de datos numéricos por lo cual se debe tener en cuenta el número máximo y mínimo de estos campos.

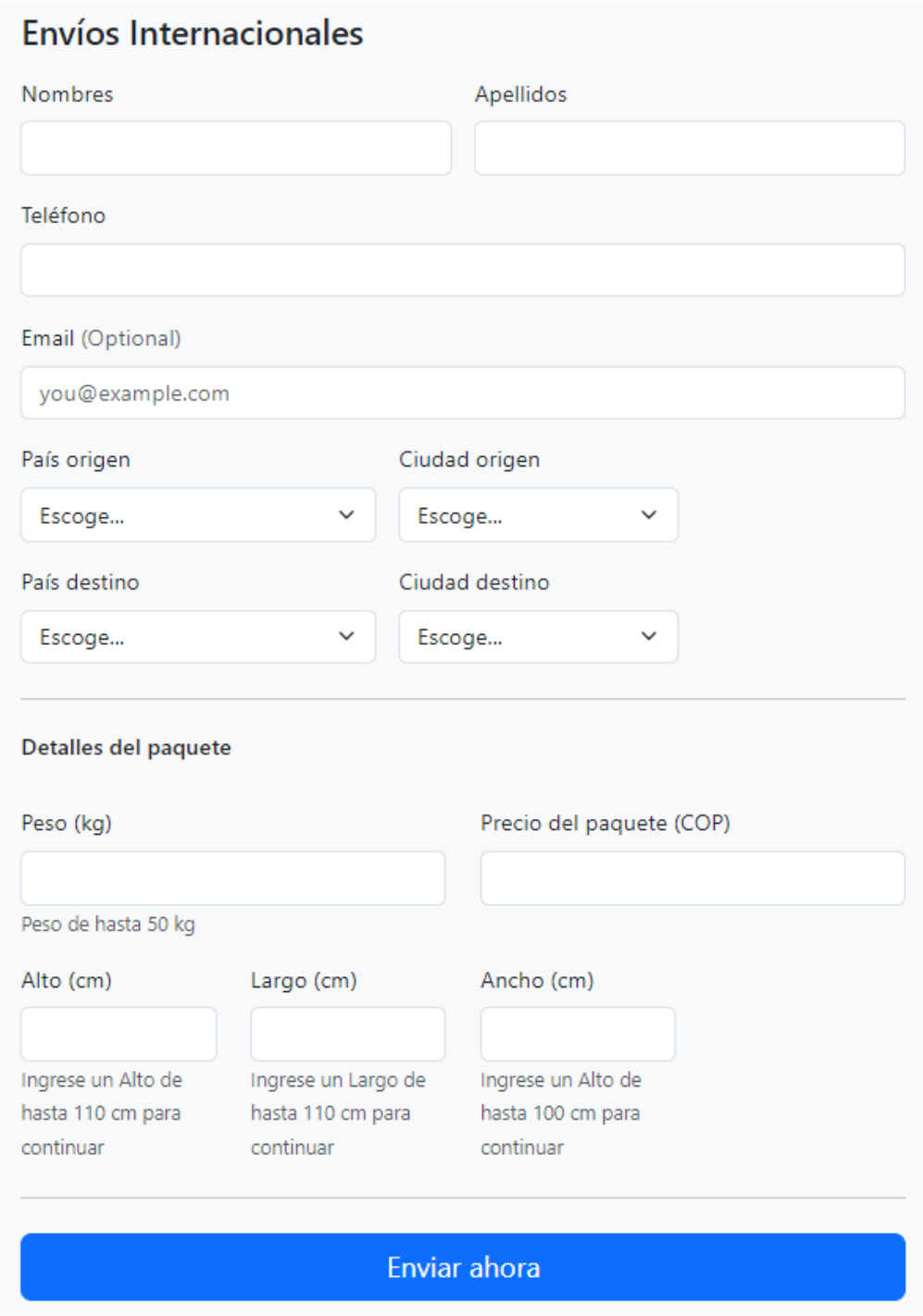


Figura 7.1: Interfaz ejemplo cuantitativo

Teniendo en cuenta todas las restricciones que poseen los campos dentro del formulario, se procede a crear los casos de prueba de forma exhaustiva para estar seguros de que ninguna funcionalidad se quede sin probar de la mejor manera. Cabe aclarar que en este momento no se aplica ninguna técnica de reducción de casos de prueba ya que esas se van a aplicar posteriormente para compararlo con el número de casos de prueba exhaustivos.

Por esto vamos a dividir la cuenta de los casos de prueba por campo dentro del formulario y al final se multiplicaran estos casos de prueba ya que cada uno cuenta como una entrada diferente por parte del usuario y una combinación a tomar en cuenta para así sacar el total de casos de prueba que se necesitan para probar de forma exhaustiva solo esta interfaz dentro del sistema.

Campo	**Tipo de casos de prueba**	**Número casos**
Nombre	Nombre valido, Nombre con número, Nombre vacío	3
Apellido	Apellido válido, Apellido con número, Apellido vacío	3
Teléfono	Teléfono válido, Teléfono válido con signos	4
Email	Email válido, Email inválido	2
País origen	80 Paises válidos, País inválido	81
Ciudad origen	10 Ciudades Válidas, ciudad inválida	11
País destino	80 Paises válidos, País inválido	81
Ciudad destino	10 Ciudades Válidas, ciudad inválida	11
Peso	50 pesos dentro del rango, Peso menor a 0, Peso mayor a 50kg	52
Precio	Precio en formato válido, precio en formato inválido	2
Alto	Alto de 0 a 110 cm, alto menor a 0cm, altos mayor a 110cm	112
Largo	Largo de 0 a 110 cm, largo menor a 0cm, altos mayor a 110cm	112
Ancho	Ancho de 0 a 100 cm, ancho menor a 0cm, ancho mayor a 100cm	102

Cuadro 7.14: Casos de prueba por campo

Como se puede ver en el cuadro 7.14 tenemos campos en los cuales se necesita menos casos de prueba que en otros, esto se debe a la cantidad de restricciones que tenga el campo, como en este caso estos campos tienen pocas restricciones, es mas difícil realizar los casos de prueba exhaustivos para campos como el nombre, apellido, teléfono, Email y precio ya que hay mas libertad del usuario para introducir mas opciones sin afectar de mayor manera el funcionamiento de esta parte del sistema. Por otra parte hay campos los cuales requieren muchos mas casos de prueba ya que poseen más restricciones como son los campos de ciudad y país, los cuales solo se puede escoger dentro de los países y ciudades que están listados y por lo cual cada opción se debe contar para realizar un análisis exhaustivo; esto mismo pasa con los campos de peso, alto, largo y ancho los cuales al estar acotados por un intervalo se prestan a probar exhaustivamente todos los valores que existen dentro de ese par de números para estar seguros que esta parte del sistema funciona con cualquier entrada

de usuario y no presente ningún error por parte de los datos que ingresa el usuario en estos campos con altas restricciones.

Para tener el número total de casos de prueba que hay que diseñar y ejecutar de manera exhaustiva para esta parte del sistema debemos multiplicar cada uno de los números de casos de prueba que tenemos por campo ya que estas son las diferentes combinaciones de datos de entrada por parte del usuario que pueden ser tanto válidas como inválidas y que pueden llevar al software a comportarse de manera diferente dependiendo de la elección por parte del usuario de los valores que va a ingresar dentro de los campos del formulario. Si hacemos la multiplicación del número de casos de prueba por campo, este número tendría mas de 14 dígitos, por lo cual se vuelve inviable realizar el diseño y la ejecución de estos casos de prueba de manera exhaustiva, ya que si un probador se demorara 1 minuto realizando el diseño y ejecución de cada caso de prueba, estaríamos hablando de que demoraríamos cientos de años solo probando esta parte del sistema.

Después de observar que realizar los casos de prueba de manera exhaustiva es imposible en términos de tiempo, entonces se va a proceder a utilizar la aplicación que utiliza técnicas de reducción de casos de prueba para generar los casos de prueba para esta parte del sistema.

7.3.1. Particiones de equivalencia

Para aplicar la técnica de particiones de equivalencia para reducir los casos de prueba en la aplicación, se deben tener en cuneta una serie de requisitos para poder ejecutar el algoritmo y que retorne los casos de prueba reducidos, si queremos aplicar el ejemplo del formulario debemos ajustar los elementos de la funcionalidad a las entradas del algoritmo; para ejecutar el algoritmo necesitamos en primera instancia un parámetro el cual debe ser el campo de la interfaz y a cada uno de esos parámetros se les deben de sacar las particiones de equivalencia las cuales van a ser las restricciones que se le tenían a cada campo; cada una de estas particiones deben tener un nombre diferenciador para así tener una mayor facilidad para encontrar el representante de cada partición de equivalencia y por ultimo indicar si la partición es válida o inválida lo que define el comportamiento de esa entrada con respecto al sistema.

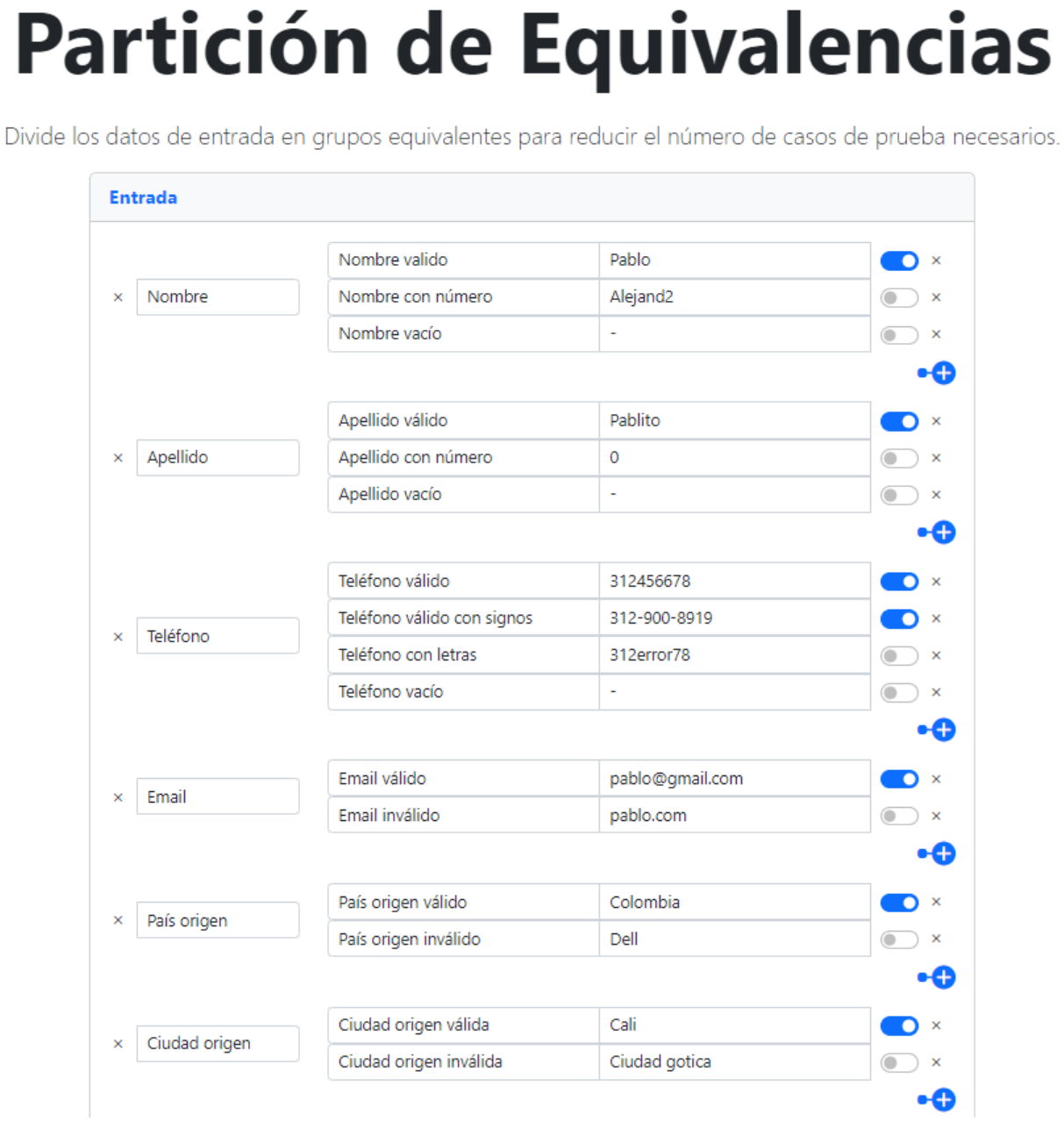


Figura 7.2: Ejemplo particiones de equivalencia 1

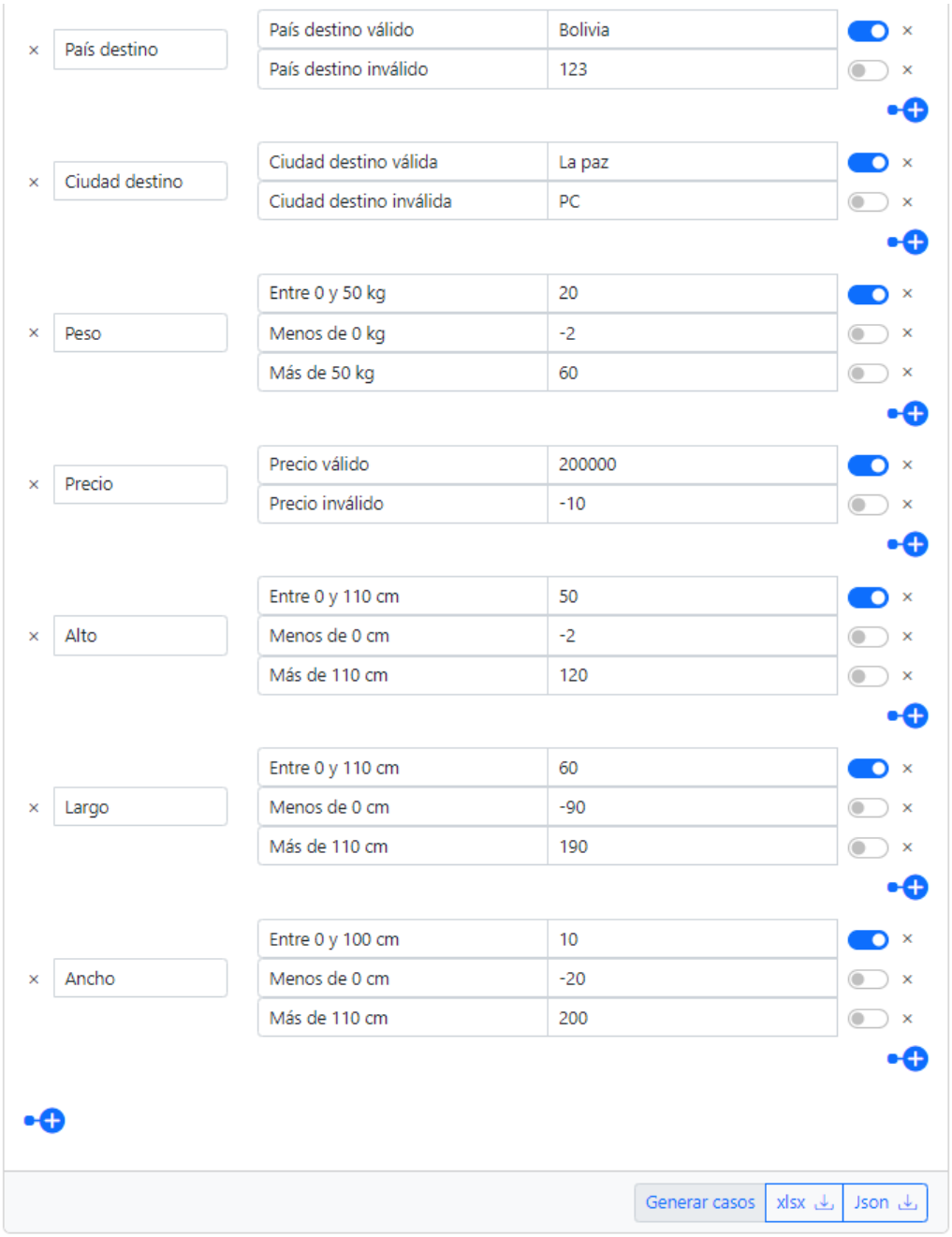


Figura 7.3: Ejemplo particiones de equivalencia 2

Como se puede observar en la figura 7.2 y en la figura 7.3 se puede observar como el ejemplo del formulario está en la aplicación y como los campos del ejemplo como son el nombre, apellido, peso, etc. Están representados como parámetros dentro de la aplicación y a cada uno de estos en

base a sus restricciones se definieron las particiones de equivalencia correspondientes y con respecto a esas particiones se generaron los representantes y se indicó si la partición es válida o inválida con respecto a las restricciones.

Nombre	Apellido	Teléfono	Email	País origen	Ciudad origen	País destino	Ciudad destino	Peso	Precio	Alto	Largo	Ancho
Pablo	Pablito	312456678	pablo@gmail.c	Colombia	Cali	Bolivia	La paz	20	200000	50	60	10
Pablo	Pablito	312-900-8919	pablo@gmail.c	Colombia	Cali	Bolivia	La paz	20	200000	50	60	10

Figura 7.4: Casos de prueba válidos ejemplo

Nombre	Apellido	Teléfono	Email	País origen	Ciudad origen	País destino	Ciudad destino	Peso	Precio	Alto	Largo	Ancho
Alejand2	Pablito	312456678	pablo@gmail.c	Colombia	Cali	Bolivia	La paz	20	200000	50	60	10
-	Pablito	312456678	pablo@gmail.c	Colombia	Cali	Bolivia	La paz	20	200000	50	60	10
Pablo	0	312456678	pablo@gmail.c	Colombia	Cali	Bolivia	La paz	20	200000	50	60	10
Pablo	-	312456678	pablo@gmail.c	Colombia	Cali	Bolivia	La paz	20	200000	50	60	10
Pablo	Pablito	312error78	pablo@gmail.c	Colombia	Cali	Bolivia	La paz	20	200000	50	60	10
Pablo	Pablito	-	pablo@gmail.c	Colombia	Cali	Bolivia	La paz	20	200000	50	60	10
Pablo	Pablito	312456678	pablo.com	Colombia	Cali	Bolivia	La paz	20	200000	50	60	10
Pablo	Pablito	312456678	pablo@gmail.c	Dell	Cali	Bolivia	La paz	20	200000	50	60	10
Pablo	Pablito	312456678	pablo@gmail.c	Colombia	Ciudad gotica	Bolivia	La paz	20	200000	50	60	10
Pablo	Pablito	312456678	pablo@gmail.c	Colombia	Cali	123	La paz	20	200000	50	60	10
Pablo	Pablito	312456678	pablo@gmail.c	Colombia	Cali	Bolivia	PC	20	200000	50	60	10
Pablo	Pablito	312456678	pablo@gmail.c	Colombia	Cali	Bolivia	La paz	-2	200000	50	60	10
Pablo	Pablito	312456678	pablo@gmail.c	Colombia	Cali	Bolivia	La paz	60	200000	50	60	10
Pablo	Pablito	312456678	pablo@gmail.c	Colombia	Cali	Bolivia	La paz	20	-10	50	60	10
Pablo	Pablito	312456678	pablo@gmail.c	Colombia	Cali	Bolivia	La paz	20	200000	-2	60	10
Pablo	Pablito	312456678	pablo@gmail.c	Colombia	Cali	Bolivia	La paz	20	200000	120	60	10
Pablo	Pablito	312456678	pablo@gmail.c	Colombia	Cali	Bolivia	La paz	20	200000	50	-90	10
Pablo	Pablito	312456678	pablo@gmail.c	Colombia	Cali	Bolivia	La paz	20	200000	50	190	10
Pablo	Pablito	312456678	pablo@gmail.c	Colombia	Cali	Bolivia	La paz	20	200000	50	60	-20
Pablo	Pablito	312456678	pablo@gmail.c	Colombia	Cali	Bolivia	La paz	20	200000	50	60	200

Figura 7.5: Casos de prueba inválidos ejemplo

En la figura 7.4 se pueden observar los casos de prueba válidos los cuales lanza la ejecución del algoritmo de particiones de equivalencia con el ejemplo del formulario en la aplicación, como vemos son solo dos casos de prueba, ya que todas las particiones de equivalencia válidas se pueden probar por medio de esos 2 casos de prueba, se espera que en la ejecución de los casos de prueba válidos el sistema no lance errores ni haga cosas que no se esperan del mismo. por otro lado en la figura 7.5 se pueden ver los casos de prueba inválidos, que en este caso son 20 y esto se debe a la cantidad de particiones de equivalencia inválidas y que cada una de estas debe ser probada al menos una vez.

Con estos resultados podemos concluir que la aplicación utilizando la técnica de particiones de equivalencia si redujo de manera considerable los casos de prueba con respecto a la prueba exhaustiva, ya que pasaron de ser mas de 14 dígitos a solo ser 22 casos de prueba lo que lo hace mas

manejable tanto para el proyecto como para los probadores del mismo.

7.3.2. Valores al límite

Para la técnica de valores al límite solo nos vamos a enfocar en los campos numéricos y que representen un intervalo como son el peso, el alto, largo y ancho; esto se debe a que la técnica de valores al límite busca entre el límite inferior y el límite superior para encontrar donde puede estar fallando un sistema.

En un principio tenemos que el campo de peso tiene 52 pruebas exhaustivas, el alto tiene 112, el largo tiene 112 y el ancho 102; Eso resultaría en un total de 66,533,376 casos de prueba solo con esos 4 campos, lo que sigue siendo un número muy elevado cuando pasamos al diseño y la ejecución de las pruebas, es por esto que por medio de la aplicación y la técnica de valores al límite vamos a buscar reducir ese número de casos de prueba sin comprometer la cobertura de las pruebas.

Para ejecutar esta técnica necesitamos que los parámetros estén definidos de igual manera como los campos están definidos en el formulario y que los intervalos estén dados en funciones lambda, además de definir cual es el incremento con el cual crece el parámetro en cuestión.

Figura 7.6: Ejemplo valores al límite

Como se observa en la figura 7.6 ya se tienen los parámetros correspondientes a los campos del

ejemplo del formulario dentro de la interfaz de la técnica de valores al límite, junto con su función lambda la cual muestra el rango de el intervalo en cuestión y el paso el cual para el peso es 0.1 ya que esta en kilogramos y si se quieren medir gramos ese es el paso indicado y por parte del alto, largo y ancho es de 1 ya que esta medido en centímetros entonces si puede tener esta diferencia para funcionar correctamente.

Peso	Alto	Largo	Ancho
0	1	1	1
50.1	1	1	1
0.1	0	1	1
0.1	111	1	1
0.1	1	0	1
0.1	1	111	1
0.1	1	1	0
0.1	1	1	101

Figura 7.7: Resultado inválidos valores al límite

En la figura 7.7 se pueden visualizar los resultados obtenidos por la técnica de análisis de valores al límite en el apartado de los casos de prueba inválidos, podemos ver que ha generado 8 casos de prueba los cuales evalúan los valores que están por fuera de los límites establecidos por parte del sistema. En cuanto a los casos de prueba válidos se generaron 625 casos de prueba, esto se debe a la combinación de 5 casos de prueba generados por cada una de los campos, entonces por recomendación para evaluar los casos de prueba válidos de esta técnica es mas sencillo hacerlos de uno en uno, pero si se necesitan las combinaciones entre las mismas se pueden poner varias al tiempo pero se tiene que tener en cuenta que genera 5 casos de prueba válidos y 2 inválidos por cada parámetro; entonces es más sencillo diferenciar casos de prueba si se ejecuta la técnica un parámetro a la vez. En cuanto a la comparación con los casos de prueba exhaustivos, teníamos que eran 66,533,376 solo para los 4 parámetros evaluados; y comparado a los 633 casos de prueba generados por la herramienta, sigue siendo mas eficiente generar los casos de prueba con la aplicación que de forma exhaustiva.

7.3.3. Arreglos Ortogonales

Como última técnica que vamos a utilizar para comparar los casos de prueba obtenidos es la de arreglos ortogonales, en esta técnica podemos utilizar todos los campos de la interfaz para incluirlos en el ejemplo y no solo los numéricos como pasaba con la anterior técnica; pero esta técnica también tiene su peculiaridad y es que debemos tener los representantes de cada uno de los parámetros y tener cuidado con el número de diferentes valores que puede tener cada parámetro, ya que los arreglos ortogonales de Taguchi están diseñados para que al menos 2 parámetros puedan tener diferente numero de valores que pueden tomar, por ejemplo, si tenemos el parámetro peso que solo puede tomar 2 valores y tenemos el parámetro alto que puede tomar 3 valores, hasta ahí todo bien,

pero la técnica no se puede aplicar si agregamos otro parámetro que tenga 4 valores, si podríamos agregar otro que tome 3 o 2 valores.; Es por esto que hay que tener cuidado cuando se utiliza esta técnica. Para nuestro ejemplo vamos a utilizar los campos utilizados en el ejemplo de particiones de equivalencias y sus representantes para poder hacer uso de esta técnica.

Figura 7.8: Ejemplo arreglos ortogonales

Como se puede ver en la figura 7.8 en este ejemplo utilizamos los mismos parámetros y representantes del ejemplo de particiones de equivalencia y además se tuvo cuidado con la cantidad de valores que podía tomar cada parámetro asegurándonos de que ninguno pase de los 2 o 3 valores que pueden tomar.

Nombre	Apellido	Telefono	Peso	Alto	Largo	Ancho	Email	País origen	Ciudad	País destino	Ciudad desti	Precio
Pablo	Pablito	3124536782	20	50	60	10	pablo@gmail	Colombia	Cali	Bolivia	La paz	20000
Pablo	Pablito	3124536782	20	50	60	10	Pablo.com	Dell	Gotica	123	PC	-2
Pablo	Pablito	3124536782	-2	-2	-90	-20	pablo@gmail	Colombia	Cali	Bolivia	PC	-2
Pablo	Pablito	3124536782	-2	-2	-90	-20	Pablo.com	Dell	Gotica	123	La paz	20000
Pablo	0	126172tytt12	20	50	-90	-20	pablo@gmail	Colombia	Gotica	123	La paz	20000
Pablo	0	126172tytt12	20	50	-90	-20	Pablo.com	Dell	Cali	Bolivia	PC	-2
Pablo	0	126172tytt12	-2	-2	60	10	pablo@gmail	Colombia	Gotica	123	PC	-2
Pablo	0	126172tytt12	-2	-2	60	10	Pablo.com	Dell	Cali	Bolivia	La paz	20000
Alejan2	Pablito	126172tytt12	20	-2	60	-20	pablo@gmail	Dell	Cali	123	La paz	-2
Alejan2	Pablito	126172tytt12	20	-2	60	-20	Pablo.com	Colombia	Gotica	Bolivia	PC	20000
Alejan2	Pablito	126172tytt12	-2	50	-90	10	pablo@gmail	Dell	Cali	123	PC	20000
Alejan2	Pablito	126172tytt12	-2	50	-90	10	Pablo.com	Colombia	Gotica	Bolivia	La paz	-2
Alejan2	0	3124536782	20	-2	-90	10	pablo@gmail	Dell	Gotica	Bolivia	La paz	-2
Alejan2	0	3124536782	20	-2	-90	10	Pablo.com	Colombia	Cali	123	PC	20000
Alejan2	0	3124536782	-2	50	60	-20	pablo@gmail	Dell	Gotica	Bolivia	PC	20000
Alejan2	0	3124536782	-2	50	60	-20	Pablo.com	Colombia	Cali	123	La paz	-2

Figura 7.9: Resultado arreglos orotogonales

Los resultados de esta técnica son mas que satisfactorios ya que solo resultaron 16 casos de prueba en los cuales se distribuyen bien la cantidad de datos válidos e inválidos y nos deja los valores de entrada bien organizados para empezar la fase de pruebas del sistema. Vemos también que esta técnica es la que menos casos de prueba dejó comparado con los 22 de partición de equivalencias y con los 633 de valores al límite, pero esto se puede entender ya que los arreglos ortogonales tiene menor cobertura de casos de prueba que las otras dos técnicas por lo cual sacrifican un poco de esa cobertura por ser mas eficientes en cuanto al número de casos de prueba que entregan.

En general se puede decir que todas las técnicas de reducción de casos de prueba que se implementaron en la aplicación web cumplen con la tarea de reducir los casos de prueba de manera significativa con respecto a los casos de prueba que se encuentran de manera exhaustiva; técnicas como partición de equivalencia y análisis de valores al limite tienen mejor cobertura que arreglos ortogonales; pero también se puede afirmar que todas son buenas cuando se busca la eficiencia en la reducción de casos de prueba.

7.4. Ejemplo cuantitativo reducido

En esta sección llevaremos a cabo un ejemplo cuantitativo reducido para poner a prueba la aplicación de reducción de casos de prueba utilizando técnicas de caja negra en un entorno más controlado en el cual se puedan identificar mejor los casos de prueba necesarios para asegurar la calidad del software. El objetivo de este estudio es demostrar cómo estas técnicas pueden ayudar a optimizar la cantidad de casos de prueba necesarios para evaluar la funcionalidad de un sistema o software con pocos parámetros.

Para este ejemplo cuantitativo reducido, utilizaremos una interfaz creada por nosotros que simula un sistema de reserva de vuelos nacionales. este sistema al igual que el ejemplo anterior va a poseer un formulario en el cual los pasajeros pueden indicar la ciudad desde donde quieren viajar, la ciudad a donde quieren viajar y la cantidad de adultos y niños para los cuales se va a realizar la reserva. Algunos de estos campos deben cumplir ciertas condiciones especificas las cuales nos representan mas casos de prueba en el software que se deben diseñar y ejecutar.

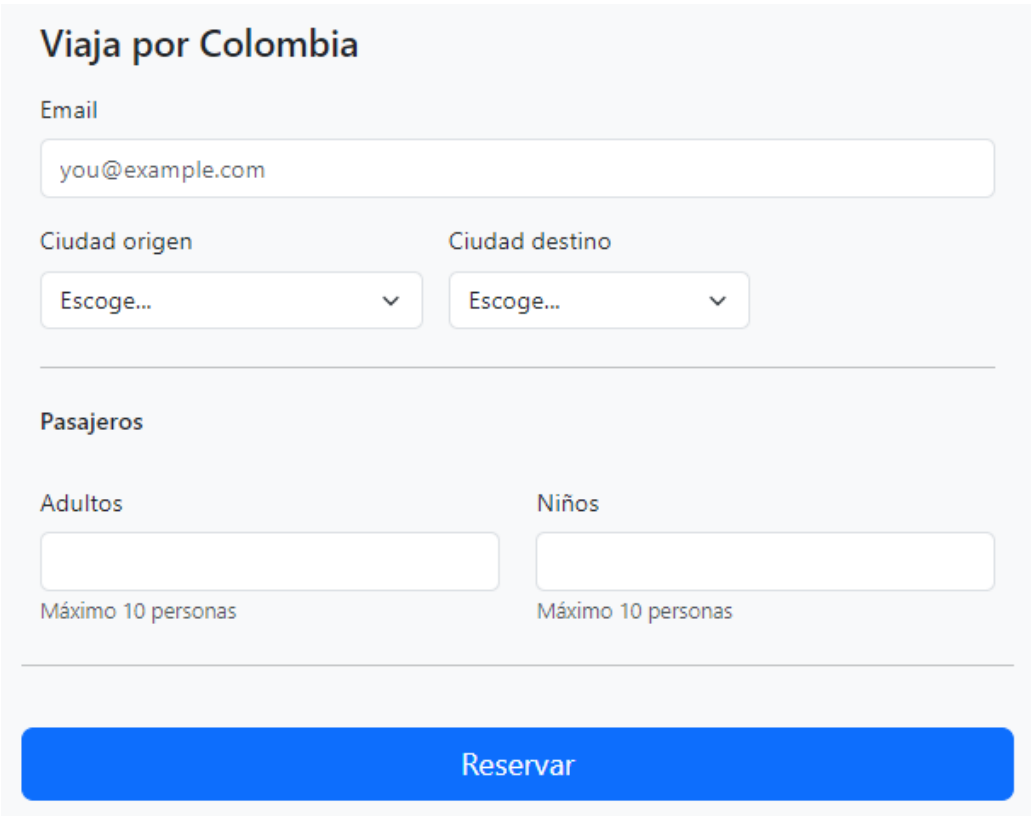


Figura 7.10: Interfaz ejemplo reducido

Como se puede ver en la figura 7.10 podemos observar la interfaz la cual consta de un formulario donde el usuario puede ingresar el e-mail de la reserva, además de la ciudad de origen y la ciudad de destino del viaje y también el número de pasajeros adultos y niños que van a viajar. Cada uno de los campos del formulario que esta en la interfaz posee una serie de restricciones las cuales hacen que hayan más o menos casos de prueba dependiendo de la condición. Estas son las condiciones que se deben de tener en cuenta por cada uno de los campos:

- Para el campo del email el cual es opcional se tiene que ingresar en el formato el cual se presenta en el campo.
- Para los campos de la ciudad de origen y ciudad destino se pueden escoger solo las 32 ciudades capitales de Colombia para elegir desde donde y hacía donde viajar.
- Para los campos del número de adultos y niños solo se pueden ingresar valores numéricos el cual es un rango del 0 al 10.

Teniendo en cuenta todas las restricciones que poseen los campos dentro del formulario, se procede a crear los casos de prueba de forma exhaustiva para estar seguros de que ninguna funcionalidad se quede sin probar de la mejor manera. Cabe aclarar que en este momento no se aplica ninguna técnica de reducción de casos de prueba ya que esas se van a aplicar posteriormente para compararlo con el número de casos de prueba exhaustivos.

Por esto vamos a dividir la cuenta de los casos de prueba por campo dentro del formulario y al final se multiplicaran estos casos de prueba ya que cada uno cuenta como una entrada diferente por parte del usuario y una combinación a tomar en cuenta para así sacar el total de casos de prueba que se necesitan para probar de forma exhaustiva solo esta interfaz dentro del sistema.

Campo	**Tipo de casos de prueba**	**Número de casos**
Email	Email formato válido, Email formato inválido, vácio	3
Ciudad Origen	32 Ciudades Válidas, ciudad inválida	33
Ciudad Destino	32 Ciudades Válidas, ciudad inválida	33
Adultos	De 0 a 10 persononas, menos de 0 y más de 10	11
Niños	De 0 a 10 persononas, menos de 0 y más de 10	11

Cuadro 7.15: Casos de prueba por campo

Como se puede ver en el cuadro 7.15 tenemos campos que requieren de menos casos de prueba exhaustivos que otros, esto se debe a la cantidad de restricciones que posea cada campo, como en el caso del email hay mas libertad por parte del usuario para introducir su correo siempre y cuando cumpla el formato establecido, mientras que los campos de ciudad origen, ciudad destino, número de niños y adultos son campos los cuales requieren más casos de prueba ya que poseen mas restricciones como puede ser el rango en el número de adultos y de niños.

Para obtener el número total de casos de prueba que hay que diseñar y ejecutar de manera exhaustiva para esta parte del sistema debemos multiplicar cada uno de los número de casos de prueba que tenemos y estas representan todas las combinaciones de datos de entrada y acciones por parte del

usuario. Si hacemos la multiplicación de todos los casos de prueba serían 395.303 casos de prueba que se necesitarían diseñar y ejecutar para probar esta interfaz, lo cuál se vuelve inviable, ya que si un probador se demorara 1 minuto realizando el diseño y ejecución de estos casos de prueba estaríamos hablando de que se demora años solo probando esta parte del software.

Después de observar que realizar los casos de prueba de manera exhaustiva es imposible en términos de tiempo, aunque sea un ejemplo mas reducido, entonces se va a proceder a utilizar la aplicación que utiliza técnicas de reducción de casos de prueba para generar los casos de prueba para esta parte del sistema.

7.4.1. Particiones de equivalencia

Para aplicar la técnica de reducción de casos de prueba de particiones de equivalencia con el ejemplo reducido se debe tener en cuenta ciertos requisitos que se necesitan para el buen funcionamiento del algoritmo; para ejecutar el algoritmo necesitamos en primera instancia un parámetro el cual debe ser el campo de la interfaz y a cada uno de esos parámetros se les deben de sacar las particiones de equivalencia las cuales van a ser las restricciones que se le tenían a cada campo; cada una de estas particiones deben tener un nombre diferenciador para así tener una mayor facilidad para encontrar el representante de cada partición de equivalencia y por ultimo indicar si la partición es válida o inválida lo que define el comportamiento de esa entrada con respecto al sistema

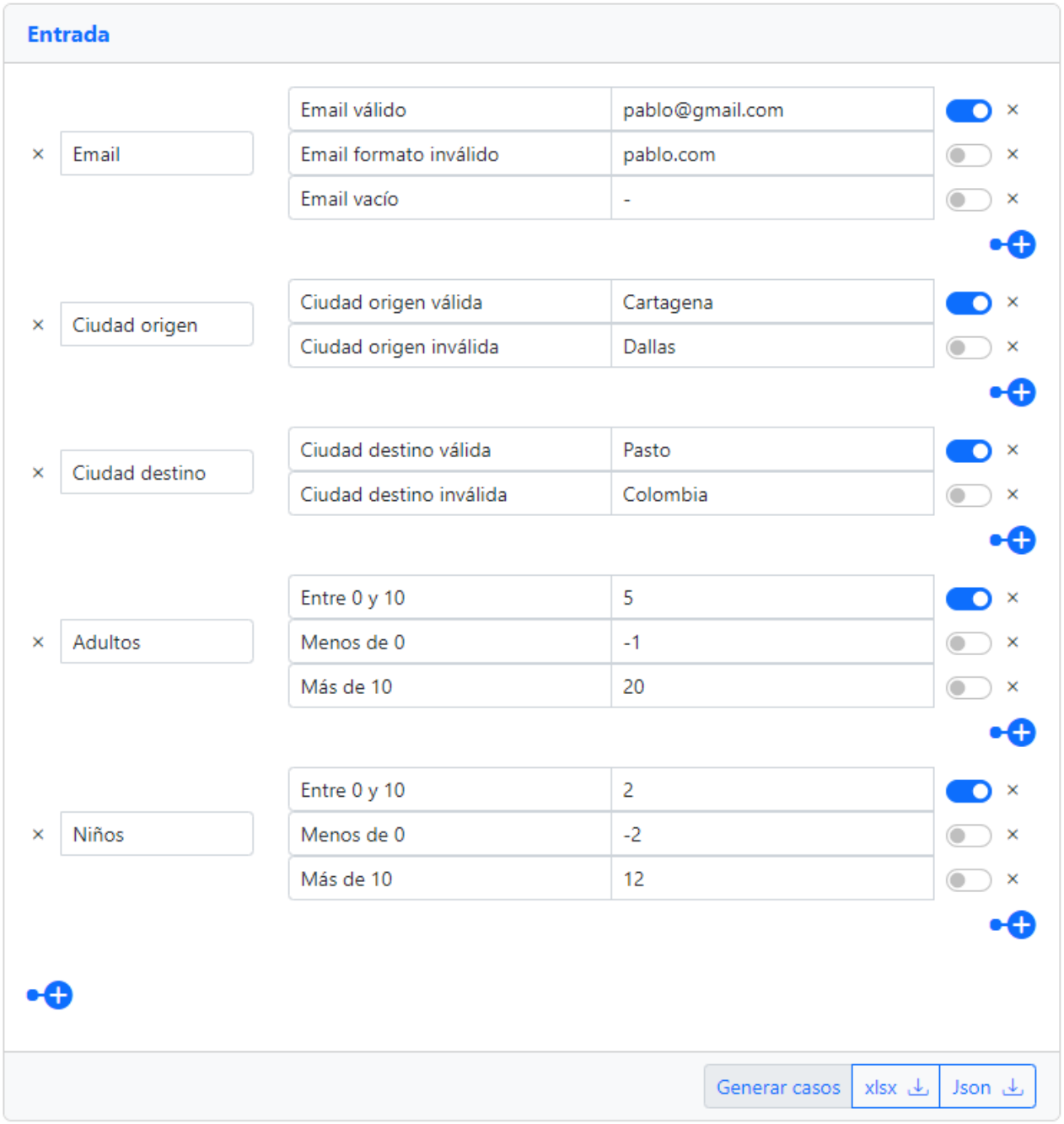


Figura 7.11: Ejemplo reducido particiones de equivalencia

Como se puede ver en la figura 7.11 vemos como cada uno de los campos de la interfaz están representados como parámetros dentro de la aplicación y cada uno de estos en base a las restriccio-

nes que tiene se definieron las particiones de equivalencia correspondientes y con respecto a estas particiones sus respectivos representantes con el indicativo de si la partición es válida o inválida.

Email	Ciudad origen	Ciudad destino	Adultos	Niños
pablo@gmail.com	Cartagena	Pasto	5	2

Figura 7.12: Resultado válidos ejemplo reducido particiones de equivalencia

Email	Ciudad origen	Ciudad destino	Adultos	Niños
pablo.com	Cartagena	Pasto	5	2
-	Cartagena	Pasto	5	2
pablo@gmail.com	Dallas	Pasto	5	2
pablo@gmail.com	Cartagena	Colombia	5	2
pablo@gmail.com	Cartagena	Pasto	-1	2
pablo@gmail.com	Cartagena	Pasto	20	2
pablo@gmail.com	Cartagena	Pasto	5	-2
pablo@gmail.com	Cartagena	Pasto	5	12

Figura 7.13: Resultado inválidos ejemplo reducido particiones de equivalencia

En la figura 7.12 se puede observar el caso de prueba válido el cuál lanza la ejecución del ejemplo reducido utilizando la técnica de particiones de equivalencia, en este caso de prueba se pueden observar todos los representantes de las particiones válidas. por otro lado en la figura 7.13 se pueden observar los casos de prueba inválidos que en este caso son 8 y esto se debe a que cada una de las particiones inválidas debe ser probada la menos una vez.

Con estos resultados podemos concluir que la aplicación utilizando la técnica de particiones de equivalencia si redujo de manera considerable los casos de prueba con respecto a la prueba exhaustiva, ya que pasaron de ser más de 300.000 a solo ser 9 casos de prueba lo que lo hace mas manejable para los probadores de esta funcionalidad

7.4.2. Valores al límite

Para la técnica de valores al límite solo nos vamos a enfocar en los campos numéricos y que representen un intervalo como son el número de adultos y de niños; esto se debe a que la técnica de valores al límite busca entre el límite inferior y el límite superior para encontrar donde puede estar fallando un sistema. Es por esto que tenemos que el número de adultos posee 11 pruebas exhaustivas, al igual que el número de niños; esto resultaría en un total de 121 casos de prueba solo para estos dos sencillos campos, es por esto que por medio de la aplicación y la técnica de valores al límite vamos a buscar reducir ese número de casos de prueba sin comprometer la cobertura de

las pruebas.

Para ejecutar esta técnica necesitamos que los parámetros estén definidos de igual manera como los campos están definidos en el formulario y que los intervalos estén dados en funciones lambda, además de definir cual es el incremento con el cual crece el parámetro en cuestión.

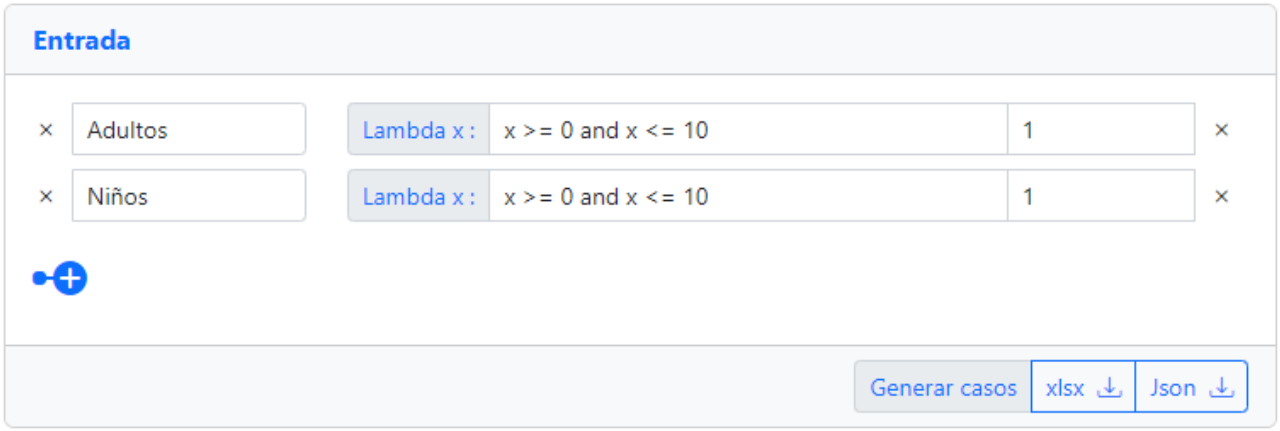


Figura 7.14: Ejemplo reducido valores al límite

En la figura 7.14 se puede observar como en la aplicación ya se tiene los parámetros correspondientes a los campos del ejemplo reducido, junto con sus funciones lambda la cual muestra el rango del intervalo en cuestión y el paso es 1 ya que estamos hablando de personas, con estos datos vamos a ejecutar la técnica de valores al límite.

Adultos	Niños
0	0
0	1
0	5
0	9
0	10
1	0
1	1
1	5
1	9
1	10
5	0
5	1
5	5
5	9
5	10
9	0
9	1
9	5
9	9
9	10
10	0
10	1
10	5
10	9
10	10

Figura 7.15: Resultado válidos ejemplo reducido valores al límite

Adultos	Niños
-1	0
11	0
0	-1
0	11

Figura 7.16: Resultado inválidos ejemplo reducido valores al límite

En la figura 7.15 se pueden visualizar los resultados obtenidos por la técnica de valores al límite en el apartado de los casos de prueba válidos podemos ver que se han generado 25 casos de prueba los cuales evalúan los valores que están dentro de los límites establecidos por parte del sistema. En cuanto a los casos de prueba inválidos se generaron 4 casos de prueba los cuales evalúan los valores los cuales están por fuera de los límites que permite el sistema. en cuanto a la comparación con

los casos de prueba exhaustivos, teníamos 121 casos de prueba sin aplicar las técnicas y después de aplicar los valores al límite resultaron 29 casos de prueba, con lo cual tenemos casi 100 casos de prueba de diferencia entre probar exhaustivamente y con la técnica de valores al límite, con lo cual se puede decir que se redujeron los casos de prueba de forma significativa.

7.4.3. Arreglos ortogonales

Como última técnica que vamos a utilizar para comparar los casos de prueba obtenidos es la de arreglos ortogonales, en esta técnica podemos utilizar todos los campos de la interfaz para incluirlos en el ejemplo y no solo los numéricos como pasaba con la anterior técnica.Para nuestro ejemplo vamos a utilizar los campos utilizados en el ejemplo de particiones de equivalencias y sus representantes para poder hacer uso de esta técnica.

Figura 7.17: Ejemplo reducido arreglos ortogonales

Como se puede ver en la figura 7.17 en este ejemplo utilizamos los mismos parámetros y representantes del ejemplo de particiones de equivalencia y además se tuvo cuidado con la cantidad de valores que podía tomar cada parámetro asegurándonos de que ninguno pase de los 2 o 3 valores que pueden tomar.

Email	Adultos	Niños	Ciudad origen	Ciudad destino
pablo@gmail.com	5	2	Cartagena	Pasto
pablo@gmail.com	5	2	Cartagena	Pasto
pablo@gmail.com	5	2	Dallas	Colombia
pablo@gmail.com	5	2	Dallas	Colombia
pablo@gmail.com	-1	-2	Cartagena	Pasto
pablo@gmail.com	-1	-2	Cartagena	Pasto
pablo@gmail.com	-1	-2	Dallas	Colombia
pablo@gmail.com	-1	-2	Dallas	Colombia
Pablo.com	5	-2	Cartagena	Colombia
Pablo.com	5	-2	Cartagena	Colombia
Pablo.com	5	-2	Dallas	Pasto
Pablo.com	5	-2	Dallas	Pasto
Pablo.com	-1	2	Cartagena	Colombia
Pablo.com	-1	2	Cartagena	Colombia
Pablo.com	-1	2	Dallas	Pasto
Pablo.com	-1	2	Dallas	Pasto

Figura 7.18: Resultado ejemplo reducido arreglos ortogonales

Los resultados de esta técnica son mas que satisfactorios ya que se redujeron los casos de prueba de 395.303 a solo 16 casos de prueba en los cuales se distribuyen de forma correcta tanto representantes válidos como inválidos y lo que hacen que el proceso de pruebas se agilice por medio de saber las combinaciones óptimas a probar.

7.4.4. Comparación

Técnica	Número de casos de prueba
Prueba exhaustiva	395.303
Particiones de equivalencia	9
Valores al límite	29
Arreglos ortogonales	16

Cuadro 7.16: Resultado número de casos de prueba por técnica

En el cuadro 7.16 podemos ver la comparativa en cuanto a número de casos de prueba de cada una de las técnicas, en la cual se puede observar como todas las técnicas de reducción de casos de prueba que se encuentran en la aplicación cumplieron su propósito y lograron disminuir considerablemente el conjunto de casos de prueba a realizar dentro de la interfaz propuesta, vimos como cada técnica logró reducir de cientos de miles casos de prueba a solo unas pocas decenas y de como estas pueden ayudar a ahorrar tiempo y dinero en el proceso de pruebas de software.

En general se puede decir que todas las técnicas de reducción de casos de prueba que se implementaron en la aplicación web cumplen con la tarea de reducir los casos de prueba de manera significativa con respecto a los casos de prueba que se encuentran de manera exhaustiva; técnicas como partición de equivalencia y análisis de valores al limite tienen mejor cobertura que arreglos ortogonales; pero también se puede afirmar que todas son buenas cuando se busca la eficiencia en la reducción de casos de prueba.

Capítulo 8

Trabajo Adicional

En este capítulo, se aborda la incorporación de una API como una solución para mejorar la eficiencia y efectividad en la reducción de casos de prueba. Se comienza introduciendo la importancia y las ventajas de utilizar una API, como la simplificación del acceso a los algoritmos de reducción y la automatización de tareas relacionadas.

Se describe en detalle la API utilizada, destacando sus funcionalidades clave diseñadas específicamente para facilitar los algoritmos de reducción de casos de prueba. Estas funcionalidades incluyen la generación de casos de prueba en formato JSON, con información sobre su tiempo de ejecución y mensajes de error. La API cuenta con documentación disponible en `Apiary` [Api23], lo que facilita su uso al proporcionar ejemplos claros y detallados de las entradas y salidas de la API. Se enfatiza la importancia de aprovechar esta documentación para acelerar la implementación y garantizar un uso correcto de la API.

8.1. Incorporación de API

8.1.1. Introducción

La incorporación de una API es una solución para facilitar el uso de los algoritmos de reducción de casos de prueba, ya que, ofrece numerosas ventajas, como la simplificación del acceso a los algoritmos de reducción de casos de prueba y la automatización de tareas relacionadas. El objetivo principal es presentar cómo se puede utilizar una API para mejorar la eficiencia y efectividad en la reducción de casos de prueba.

8.1.2. Descripción de la API

La API ofrece una amplia gama de funcionalidades diseñadas específicamente para facilitar los algoritmos de reducción de casos de prueba. Algunas de las funcionalidades clave incluyen generar los casos de prueba en formato Json, con su tiempo de ejecución y un mensaje si hubo algún error o no. Estas funcionalidades son altamente relevantes para nuestro trabajo, ya que nos facilitan las pruebas.

Es importante destacar que la API cuenta con documentación disponible en `Apiary`, lo cual facilita significativamente su uso. La documentación en `Apiary` proporciona ejemplos claros y detallados de cada una de las entradas y salidas de la API. Esto permite comprender rápidamente cómo utilizar

los diferentes endpoints y cómo estructurar las solicitudes para obtener los resultados deseados. Aprovechar esta documentación es altamente recomendado, ya que ayuda a acelerar la implementación y asegura un uso correcto de la API en el contexto de los algoritmos de reducción de casos de prueba.

8.1.3. Generación de la API Key

Para poder utilizar la API, es necesario generar una API key asociada a una dirección de correo electrónico válida. El proceso de generación de la API key implica ingresar tu dirección de correo y generar la clave API.Una vez generada la API key, se debe mantener en un lugar seguro y seguir las mejores prácticas de seguridad para protegerla de accesos no autorizados.

8.1.4. Ejemplo de Uso

Para ilustrar cómo usar API, podemos utilizar una herramienta como `Postman` [Pos23], que facilita la interacción con servicios web. A continuación, se muestra un ejemplo de cómo utilizar la API a través de Postman:

- Abre Postman y crea una nueva solicitud.
- Configura la URL de la solicitud para apuntar al endpoint relevante de la API.
- Agrega los parámetros necesarios en el cuerpo de la solicitud, siguiendo las especificaciones proporcionadas en la documentación de la API.
- Incluye la API key en el cuerpo de la solicitud para autenticarte correctamente.
- Envía la solicitud.

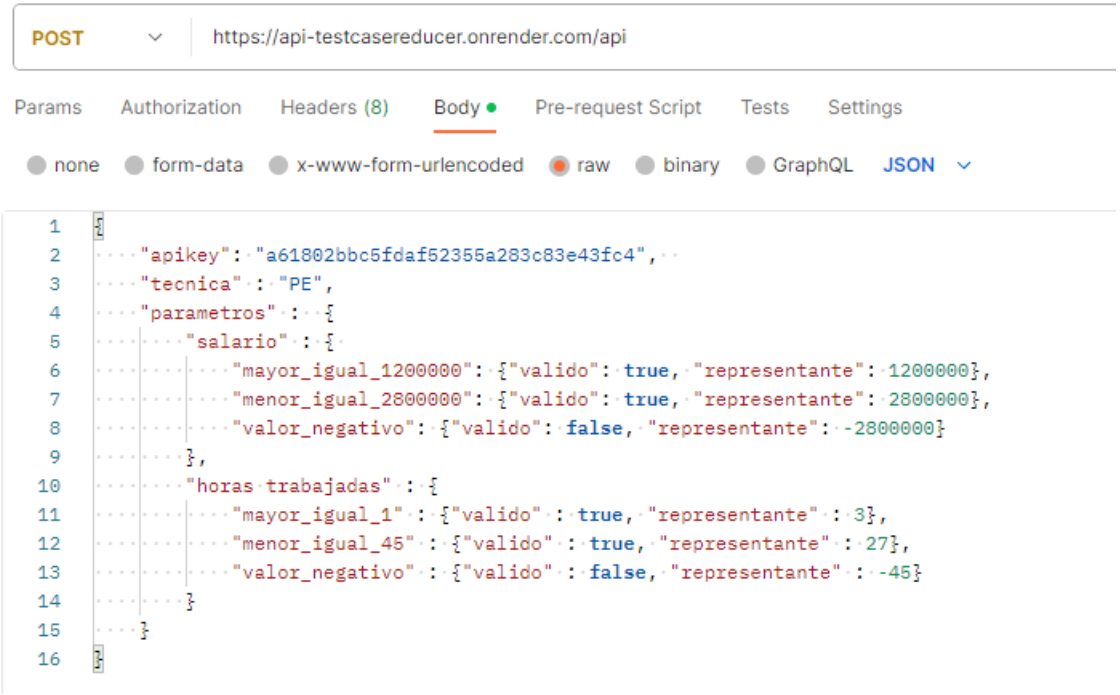


Figura 8.1: Solicitud partición de equivalencias.

En la figura 8.1 se presenta un escenario con 2 parámetros: `''salario''` con 3 clases de equivalencia (2 válidas y 1 inválida), `''horas trabajadas''` con 3 clases (2 válidas y 1 inválida). Cada parámetro se define como clave:valor, donde la clave es el nombre del parámetro y el valor es un diccionario que contiene un conjunto de objetos. Cada objeto representa una clase de equivalencia y debe tener el siguiente formato: `clave: {''valido'' : true | false, ''representante'' : any}`. Donde clave es el nombre de la clase de equivalencia.

```
{
    "error": false,
    "tecnica": "PE",
    "casos-pruebas": {
        "casos_validos": [
            {
                "salario": {
                    "clase_equivalencia": "mayor_igual_1200000",
                    "representante": 1200000
                },
                "horas trabajadas": {
                    "clase_equivalencia": "mayor_igual_1",
                    "representante": 3
                }
            },
            {
```


Figura 8.2: Respuesta partición de equivalencias

```
{
    "error": false,
    "tecnica": "PE",
    "casos-pruebas": {
        "casos_validos": [···
        ],
        "casos_invalidos": [
            {
                "salario": {
                    "clase_equivalencia": "valor_negativo",
                    "representante": -2800000
                },
                "horas trabajadas": {
                    "clase_equivalencia": "mayor_igual_1",
                    "representante": 3
                }
            },
```


Figura 8.3: Respues partición de equivalencias (continuación)

La respuesta generada por la API es la siguiente:

- ”error”: Indica si se produjo un error durante la ejecución. En este caso, el valor es false, lo que significa que no se produjo ningún error.
- ”tecnica”: Indica la técnica utilizada. En este ejemplo, la técnica utilizada es ”PE”.
- ”casos-pruebas”: Contiene dos listas de casos de prueba, ”casos_validos” y ”casos_invalidos”.
    - ”casos_validos”: Contiene una lista de casos de prueba válidos. Cada caso de prueba válido incluye información sobre el salario y las horas trabajadas.

”salario”: Información sobre el salario, que incluye la clase de equivalencia y el valor representativo.
”horas trabajadas”: Información sobre las horas trabajadas, que incluye la clase de equivalencia y el valor representativo.

- ”casos_invalidos”: Contiene una lista de casos de prueba inválidos. Cada caso de prueba inválido también incluye información sobre el salario y las horas trabajadas

- ”tiempo-transcurrido”: Indica el tiempo transcurrido durante la ejecución.

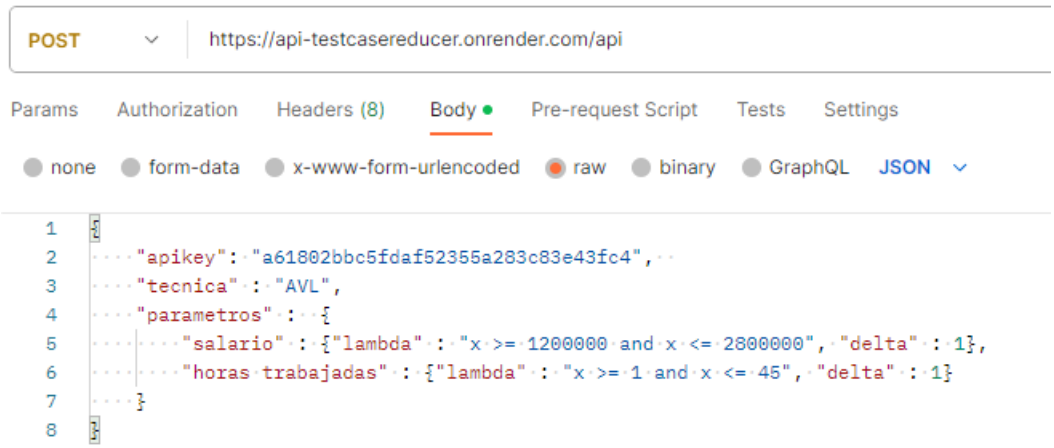
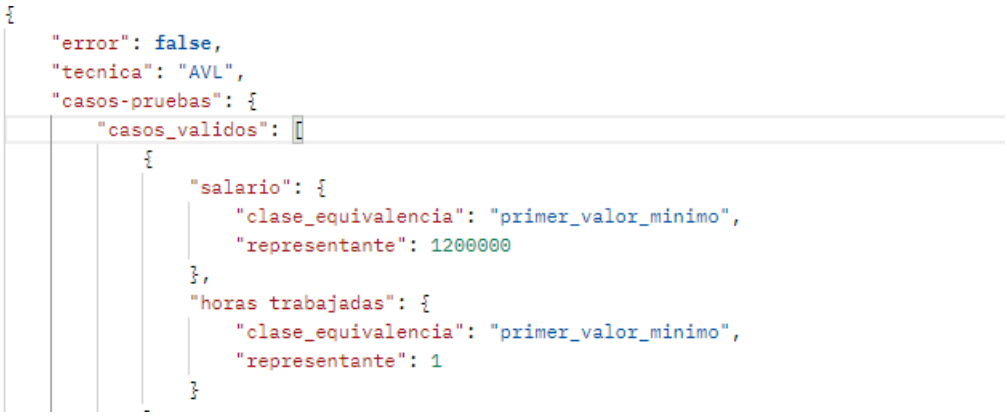


Figura 8.4: Solicitud valores límites

En la figura 8.4 consideramos dos parámetros de tipo lambda: salario y horas trabajadas. Cada parámetro está acotado en un rango específico y con un paso especifico. Cada parámetro genera 2 valores inválidos y 5 valores válidos mediante la función lambda correspondiente, finalmente, se usa la técnica de partición de equivalencias para generar los casos de prueba.



```
{
    "error": false,
    "tecnica": "AVL",
    "casos-pruebas": {
        "casos_validos": [
            {
                "salario": {
                    "clase_equivalencia": "primer_valor_minimo",
                    "representante": 1200000
                },
                "horas trabajadas": {
                    "clase_equivalencia": "primer_valor_minimo",
                    "representante": 1
                }
```


Figura 8.5: Respuesta valores límites

La respuesta de los valores límites sigue el mismo formato que las particiones de equivalencia. Cada parámetro lambda se define mediante las siguientes clases de equivalencia (claves): 'valor_minimo_invalido', 'primer_valor_minimo', 'segundo_valor_minimo', 'valor_medio', 'primer_valor_maximo', 'segundo_valor_maximo', 'valor_maximo_invalido'.

Estas claves se utilizan para representar las diferentes clases de equivalencia en el conjunto de parámetros lambda. Cada clase de equivalencia tiene un propósito específico para abarcar diferentes escenarios y limitaciones. Es importante considerar estas claves al realizar pruebas y validar los valores límites de los parámetros.

```
{
    "error": false,
    "tecnica": "AVL",
    "casos-pruebas": {
        "casos_validos": [ ···
        ],
        "casos_invalidos": [
            {
                "salario": {
                    "clase_equivalencia": "valor_minimo_invalido",
                    "representante": 1199999
                },
                "horas trabajadas": {
                    "clase_equivalencia": "primer_valor_minimo",
                    "representante": 1
                }
            },
            {
```


Figura 8.6: Respuesta valores límites (continuación)

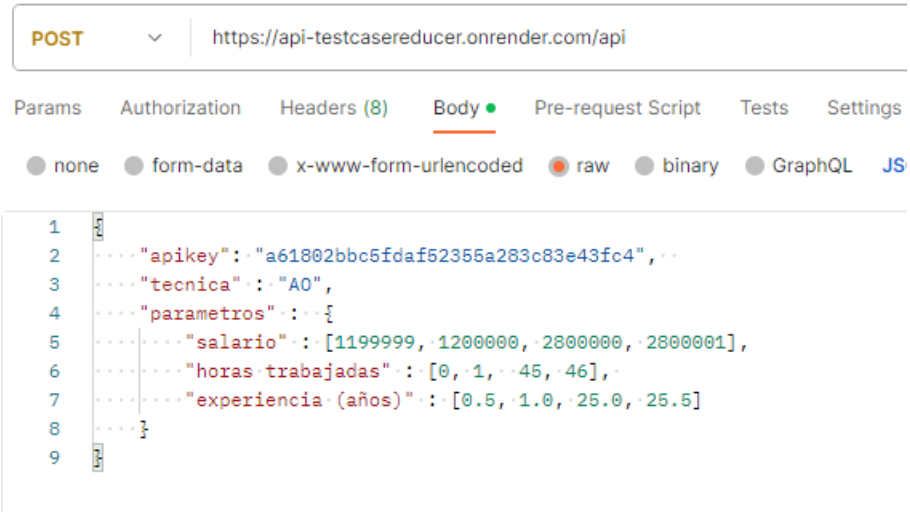


Figura 8.7: Solicitud arreglos ortogonales

En la figura 8.7 consideramos 4 parámetros: ”salario” con 4 opciones, ”horas trabajadas” con 4 opciones, ”experiencia (años)” con 4 opciones.

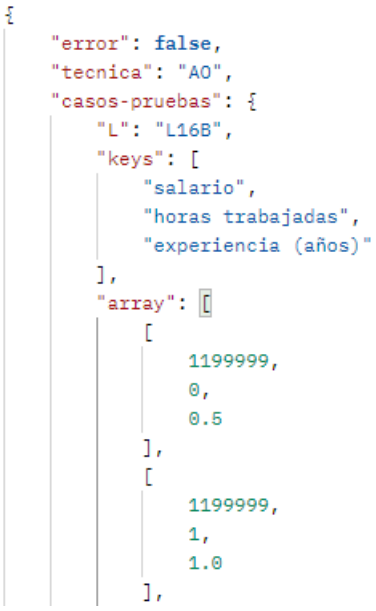


Figura 8.8: Respuesta arreglos ortogonales

La respuesta generada por la API es la siguiente (ver Figura 8.8):

- ”L”: Indica el arreglo ortogonal usado. En este caso, L es ”L16B”.

- ”keys”: Es una lista de las claves de los casos de prueba. En este caso, las claves son ”salario”, ”horas trabajadas” y ”experiencia (años)”. Cada clave es el nombre de la columna en el array.
- ”array”: Es una matriz que representa los valores de los casos de prueba. Cada fila de la matriz corresponde a un caso de prueba y contiene los valores para las claves mencionadas anteriormente.

Capítulo 9

Conclusiones

9.1. Conclusión

9.1.1. Conclusión general

En este proyecto, llevamos a cabo el desarrollo e implementación de una aplicación web que utiliza técnicas de reducción de casos de prueba de caja negra. Identificamos y seleccionamos las técnicas más usadas en el contexto de pruebas. Para cada técnica, evaluamos sus ventajas y desventajas, y con base a esto seleccionamos aquellas que brindarían mayor valor al usuario.

Además de la implementación de las técnicas de reducción de casos de prueba, diseñamos una interfaz de usuario intuitiva y fácil de usar para nuestra aplicación web. Siguiendo los principios de diseño centrados en el usuario y la usabilidad, nos aseguramos de proporcionar una experiencia agradable para el usuario. También integramos la capacidad de presentar los resultados generados por la aplicación en formatos Json y XLSX.

Durante la implementación de la aplicación, utilizamos las tecnologías como FastAPI y Next.js. Estas tecnologías nos permitieron desarrollar de manera eficiente y rápida, asegurando un rendimiento óptimo y una experiencia fluida para el usuario. Como valor adicional, nuestra aplicación web cuenta con una API bien diseñada y documentada. Esto permite a los desarrolladores interactuar con la plataforma de manera programática y aprovechar sus funcionalidades en sus propias aplicaciones. La API amplía las posibilidades de nuestra aplicación y la hace más versátil y adaptable a diferentes entornos y necesidades.

Se evidenció de manera a través de ejemplos concretos, que la aplicación de las técnicas de reducción de casos de prueba resultó en una disminución significativa en la cantidad total de casos de prueba necesarios. Esto demuestra la eficacia de dichas técnicas para optimizar y simplificar el proceso de pruebas en la aplicación web desarrollada.

En resumen, este proyecto involucró la investigación, implementación y evaluación de técnicas de reducción de casos de prueba en una aplicación web. Además, la interfaz ha sido diseñada con el objetivo de ser intuitiva y fácil de usar. Sin embargo, no podemos confirmar de manera definitiva si se ha logrado ese objetivo, logramos exportar los casos de prueba en un formato fácil de usar y también proporcionamos una API que permite una integración programática con la plataforma. Los resultados obtenidos demostraron el buen funcionamiento de las técnicas de reducción de casos de

prueba, lo que proporciona una base sólida para futuras investigaciones y ofrece un enfoque efectivo para mejorar el proceso de pruebas en las aplicaciones web.

9.1.2. Objetivos

En conclusión, en este proyecto de grado se lograron completar todos los objetivos definidos desde un principio en la propuesta. Se desarrolló una aplicación web que utiliza las técnicas de reducción de casos de prueba de caja negra, evidenciando el uso de las técnicas que mejor se adaptaban a las necesidades de los usuarios en general.

Además, se realizó el análisis e implementación de estas técnicas en algoritmos que se pueden usar desde una interfaz gráfica intuitiva, lo que facilita su uso de manera sencilla y comprensible. El manual de usuario y la interfaz clara y concisa contribuyen a una experiencia favorable para los usuarios.

Asimismo, se logró presentar los resultados en los formatos que mejor se ajustaban a las necesidades de los usuarios y de la aplicación en general. También se incorporó una API que proporciona una forma más ténica de interactuar con la aplicación, brindando mayor flexibilidad y posibilidades de integración.

En general, la aplicación web cumple con todos los objetivos planteados inicialmente y se entregó un aplicación web funcional. Este proyecto proporciona una base sólida para futuras investigaciones relacionadas a la implementación de técinas de reducción de casos de prueba de caja negra, y ofrece un enfoque efectivo para mejorar el proceso de pruebas en las aplicaciones web.

9.1.3. Trabajo futuro y consideraciones

Después de revisar los resultados obtenidos y completar todos los objetivos propuestos inicialmente, identificamos algunas áreas de mejora durante el desarrollo del proyecto. En primer lugar, consideramos que sería interesante implementar métricas que permitan al usuario observar de manera más clara cómo la aplicación reduce los casos de prueba y el porcentaje de reducción logrado por la técnica seleccionada. Esto proporcionaría una mejor comprensión de los beneficios que ofrece la aplicación en términos de optimización de las pruebas.

Además, en futuras versiones de la aplicación, sería valioso explorar técnicas de reducción de pruebas más complejas, como aquellas basadas en inteligencia artificial y algoritmos genéticos. Aunque estas técnicas pueden ser más precisas, también son más desafiantes de implementar. Sin embargo, ofrecen la posibilidad de optimizar aún más la selección de casos de prueba y brindar un mayor beneficio al usuario final.

Por último, buscamos que la aplicación tenga la capacidad de expandirse a otros entornos y sea reconocida por un público más amplio, tanto dentro como fuera del ámbito de la disciplina de pruebas de software. Creemos que esto sería de gran ayuda para fomentar el interés en esta disciplina y motivar a más personas a explorar el campo de las pruebas de software, dado su relevancia para

asegurar la calidad en los proyectos.

En resumen, para futuras mejoras consideramos implementar métricas visuales, explorar técnicas más avanzadas de reducción de pruebas y trabajar en la expansión y divulgación de la aplicación en diferentes ámbitos. Estas acciones contribuirán a aumentar su valor y su impacto en la comunidad interesada en las pruebas de software.

Bibliografía

[AMG94] M. C. Paul A. M. Goel. Orthogonal array testing: A review. *Software Testing, Verification and Reliability*, 1994.

[Api23] Apiary. Apiary - design, develop, and document apis, 2023.

[Bat23] Ned Batchelder. Coverage.py - code coverage measurement for python. Recuperado el 2023, desde https://coverage.readthedocs.io/en/7.2.7/, 2023.

[Bei95] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. 1995.

[Dom20] Gopal Dommety. Reducing cost complexity of testing : Ml-based continuous verification. *OpsMX*, 2020.

[Fou23a] Python Software Foundation. Python 3.8 documentation. Recuperado el 2023, desde https://docs.python.org/3.8/, 2023.

[Fou23b] Python Software Foundation. unittest - unit testing framework. Recuperado el 2023, desde https://docs.python.org/3/library/unittest.html, 2023.

[Gro23] PostgreSQL Global Development Group. Postgresql documentation. Recuperado el 2023, desde https://www.postgresql.org/docs/, 2023.

[IBM21] IBM. *Visión general de casos de prueba y suites de pruebas*. Marzo 2021.

[IST18] ISTQB. *International Software Testing Qualifications Board*. Spanish Software Testing Qualifications Board con el apoyo del Hispanic America Software Testing Qualifications Board, Certified Tester - Foundation Level, Version 2018.

[JS07] Shane Warden James Shore. *The Art of Agile Development*. 2007.

[jso23] Json.org, 2023.

[KKP21] Patrick Kreutzer, Tom Kunze, and Michael Philippsen. *Test Case Reduction: A Framework, Benchmark, and Comparative Study*. 2021.

[KS11] Mumtaz Ahmad Khan and Mohd. Sadiq. *Analysis of black box software testing techniques: A case study*. 2011.

[lea23] QA lead. 5 types of black box testing techniques + examples. 2023.

[Lyn17] Jamie Lynch. Therac-25 causes radiation overdoses. *Bugsnag*, 2017.

[MO] Jacob Thornton Mark Otto. Bootstrap documentation. Recuperado el 2023, desde https://getbootstrap.com/.

[Mye04] Glenford J. Myers. *Art of software testing.* capítulo 2, 2004.

[NJ18] M. Solari S. Abraham 1. Ramos N. Juristo, S. Vegas. Comparing the effectiveness of equivalence partitioning, branch testing and code reading by stepwise abstraction applied by subjects. 2018.

[Per20] A E Permanasari. *Comparing method equivalence class partitioning and boundary value analysis with study case.* 2020.

[Pos23] Postman. Postman api development environment, 2023.

[Ram23] Sebastián Ramírez. Fastapi. Recuperado el 2023, desde https://fastapi.tiangolo.com/, 2023.

[Ran22] Ranorex. Testing coverage techniques for the testing process. 2022.

[Rau23] Guillermo Rauch. Next js. Recuperado el 2023, desde https://nextjs.org/, 2023.

[Rie23] Montalbán Riera. *Optimiza tus pruebas de software con arreglos ortogonales.* Abril 2023.

[Rod23] Johanna Rodriguez. Qué es el diagrama de ishikawa, para qué sirve, cómo crearlo y ejemplos. 2023.

[rt22] The redscan team. Types of pen testing: Black box, white box grey box. 2022.

[SÍ5] José Sánchez. *Pruebas de Software. Fundamentos y Técnicas.* Junio 2015.

[Sau12] Ángel Eduardo Pentón Saucedo. Aplicación de la tabla ortogonal en diseño de los casos de prueba de software. 2012.

[th23a] Software testing help. Black box testing: An in-depth tutorial with examples and techniques. 2023.

[th23b] Software testing help. Test coverage in software testing (tips to maximize testing coverage). 2023.

[Usm16] Muhammad Usman. *Black Box testing Strategies used in Financial Industry for Functional testing.* Department of Computer Science, COMSATS University Islamabad, 2016.

[VH20] Dede Rosyada Velia Handayani. Gamified learning platform analysis for designing a gamification-based ui / ux of e-learning applications. 2020.