

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



“GENERACIÓN AUTOMÁTICA DE PROBLEMAS DE PROGRAMACIÓN LINEAL”

IGNACIO ROJAS VEGA

**MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA**

Profesor Guía: Carlos Castro
Profesor Correferente: Sergio Campos

Diciembre - 2024

DEDICATORIA

Le dedico esta memoria a mi familia, especialmente a mi madre, quien me impulsó y animó constantemente en mis estudios para no rendirme. También a Claudia, el amor de mi vida, quien me motivó día a día en el escrito de esta memoria y a quien le debo el hecho de concluirla.

AGRADECIMIENTOS

Quiero agradecer de corazón a todas las personas que fueron parte de este proceso todos estos años. Primero agradecer a mi familia, a mi padre Henry por darme la oportunidad de poder estudiar una carrera universitaria; a mi madre Viviana por sostenerme día a día, por darme el apoyo emocional que necesitaba para sobrellevar el estrés universitario; a mi hermana Constanza, que con su sola presencia me sacaba una sonrisa hasta en el peor día que pude tener; a mi perrita Canela, que partió de este mundo antes de poder concluir este escrito y que me acompañó noches enteras de estudio.

Quiero agradecer a mis amigos de la vida, a Iván Toledo, Ángel Fernández, Adolfo Klug, Diego Tapia, Alejandro Henríquez y Viviana Henríquez, quienes fueron un soporte emocional en momentos de estrés, con quienes compartí momentos de alegría y tristeza durante esta etapa universitaria. También quiero agradecer a los amigos que hice durante la universidad, al grupo “Simios”, con quienes logré conformar un hermoso grupo de estudio, ayudándonos constantemente en todos los ramos que cursamos juntos y que incluso en plena pandemia logramos apoyarnos. Mención honrosa a Luis Blanco, compañero que se convirtió en amigo de la vida gracias a la universidad, con quien estuvimos estudiando y formando grupos juntos desde primer año hasta esta instancia.

Quiero agradecer a Claudia Hernández, el amor de mi vida, quien llegó en un momento importante de mi vida a hacerme feliz y con quién he logrado renacer, reencontrándome conmigo mismo, mejorando día a día en todo ámbito.

Quiero agradecer a mis compañeros de trabajo en *Cuprum* por bancarme y por enseñarme buenas prácticas en el ámbito laboral y por permitir tener un buen ambiente de trabajo todos los días. A Javier García y Glenis León, con quienes vengo trabajando en desarrollo de software por 3 años consecutivos pasando por distintos equipos juntos. A José Cruz, con quién he formado una linda amistad gracias al trabajo.

Quiero agradecer también a todas las personas que estuvieron antes de esta etapa, a los que se fueron sumando en el camino y a quienes también se fueron.

Agradecer también a mi profesor guía Carlos Castro por su apoyo, los documentos y herramientas otorgadas para poder llevar a cabo este documento.

Comencé esta etapa siendo un niño y la culmino ya como un adulto, gracias de corazón a todos quienes fueron testigos de este crecimiento y que hoy en día siguen a mi lado.

RESUMEN

Resumen— La presente memoria aborda el desarrollo de una herramienta automatizada para la generación de problemas de programación lineal (PL), diseñada específicamente para el ámbito educativo y de investigación en la Universidad Técnica Federico Santa María. La necesidad de esta herramienta surge del desafío que enfrentan tanto docentes como estudiantes en el curso de Optimización, donde la creación manual de problemas resulta un proceso laborioso y repetitivo. La solución propuesta emplea un algoritmo flexible capaz de generar problemas de PL con distintos niveles de complejidad y tipos de variables (continuas, enteras y binarias), permitiendo la creación de ejercicios personalizados. La herramienta, implementada en Python, utiliza la biblioteca OR-Tools para la optimización y se integra con *solvers* externos mediante formatos de exportación estándar como MPS, lp_solve y LINDO.

Palabras Clave— optimización, programación lineal, generación automática, algoritmo de optimización, enseñanza.

ABSTRACT

Abstract— This thesis presents the development of an automated tool for generating linear programming (LP) problems, designed specifically for educational and research purposes at Universidad Técnica Federico Santa María. The motivation behind this tool stems from the challenges faced by both instructors and students in the Optimization course, where manual problem creation is often time-consuming and repetitive. The proposed solution features a flexible algorithm capable of generating LP problems with varying complexity levels and variable types (continuous, integer, and binary), thus enabling personalized exercises. Implemented in Python, the tool uses the OR-Tools library for optimization and integrates with external solvers through standard export formats such as MPS, lp_solve, and LINDO.

Keywords— optimization, linear programming, automated generation, optimization algorithm, education.

GLOSARIO

DI: Departamento de Informática de la Universidad Técnica Federico Santa María.

PL: Programación Lineal.

PLE: Programación Lineal Entera.

PLB: Programación Lineal Binaria.

PLM: Programación Lineal Mixta.

UTFSM: Universidad Técnica Federico Santa María.

INDICE DE CONTENIDOS

RESUMEN.....	4
ABSTRACT	4
GLOSARIO	5
INDICE DE FIGURAS.....	9
INDICE DE TABLAS.....	11
INTRODUCCIÓN	12
CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA	13
1.1 Contexto	13
1.2 Situación actual	13
1.2.1 Estudiantes de educación superior en contexto de pandemia	13
1.2.2 Profesores en contexto de pandemia	17
1.2.3 Ramo Optimización en pandemia	19
1.3 Objetivos	19
1.3.1 Objetivo general	19
1.3.2 Objetivos específicos.....	19
CAPÍTULO 2: MARCO CONCEPTUAL	21
2.1 Programación Lineal.....	21
2.1.1 Definición de un modelo matemático	22
2.1.2 Formulación de un modelo matemático.....	22
2.2 Métodos de Resolución de Problemas de Programación Lineal	23
2.2.1 Método gráfico.....	24
2.2.2 Método Simplex	25

2.3 Técnicas Avanzadas en Programación Lineal	27
2.3.1 Programación Lineal Entera	27
2.3.2 Programación Lineal Mixta	29
2.3.3 Programación Lineal Binaria	30
CAPÍTULO 3: PROPUESTA DE SOLUCIÓN	32
3.1 Introducción a la Solución Propuesta	32
3.2 Metodología y Tecnologías Empleadas	33
3.2.1 Elección de Python y sus Bibliotecas	33
3.2.2 Descripción del Proceso Algorítmico	34
3.3 Algoritmo Propuesto	36
3.3.1 Explicación Detallada del Algoritmo Desarrollado	36
3.3.2 Pseudocódigo del Programa	44
CAPÍTULO 4: VALIDACIÓN DE LA APLICACIÓN	57
4.1 Introducción	57
4.2 Criterios de Validación	58
4.2.1 Exactitud de los Modelos Generados	58
4.2.2 Compatibilidad con Solvers Externos	59
4.2.3 Facilidad de Uso de la Interfaz	59
4.2.4 Eficiencia y Tiempos de Ejecución	60
4.3 Pruebas Realizadas	60
4.3.1 Pruebas Funcionales	61
4.3.2 Pruebas de Integración	67
4.3.3 Pruebas de Rendimiento	74
CAPÍTULO 5: ESTUDIO DE CARÁCTERÍSTICAS DE LAS SOLUCIONES	79

5.1 Introducción a las Características de las Soluciones	79
5.2 Condiciones Matemáticas	79
5.2.1 Solución Única	79
5.2.2 Solución Múltiple	79
5.2.3 Solución Ilimitada	80
5.2.4 Problema Sin Solución	80
5.3 Análisis de Resultados	80
5.3.1 Solución Única: Eficiencia en la Optimización	80
5.3.2 Solución Múltiple: Flexibilidad en la Toma de Decisiones	81
5.3.3 Solución Ilimitada: Problemas en la Modelización	81
5.3.4 Problema Sin Solución: Diagnóstico y Ajustes en el Modelo	81
CAPÍTULO 6: FACTIBILIDAD DE CONSTRUIR UNA APLICACIÓN PARA GENERAR MODELOS DE PL CON CARACTERÍSTICAS ESPECIALES	82
6.1 Introducción	82
6.2 Desarrollo y Experiencia	82
6.2.1 Descripción del proceso de desarrollo de la aplicación	82
6.2.2 Modelos Matemáticos Propuestos por Castillo	82
6.2.3. Tipos de Soluciones Propuestas	85
6.3 Desafíos y problemas encontrados durante el desarrollo	86
6.4 Soluciones implementadas y aprendizajes obtenidos	88
6.5 Evaluación de Factibilidad	88
CAPÍTULO 7: CONCLUSIONES	90
REFERENCIAS BIBLIOGRÁFICAS	93
ANEXOS	95

INDICE DE FIGURAS

Figura 1: Resultados sobre si los estudiantes habían tomado un curso online antes de la pandemia. Fuente: PulsoEstudiantil 2020, seguimosvirtual.com	14
Figura 2: Resultados sobre la satisfacción de los estudiantes con la respuesta de actores claves durante la pandemia. Fuente: PulsoEstudiantil 2020, seguimosvirtual.com	15
Figura 3: Resultados sobre apoyos de casas estudios hacia sus estudiantes. Fuente: PulsoEstudiantil 2020, seguimosvirtual.com.....	15
Figura 4: Resultado de estudiantes sobre la calidad de educación que éstos reciben. Fuente: PulsoEstudiantil 2020, seguimosvirtual.com.....	16
Figura 5: Resultados sobre si los estudiantes tomaran algunos de sus cursos en modalidad online post pandemia. Fuente: PulsoEstudiantil 2020, seguimosvirtual.com	16
Figura 6: Resultados sobre utilidad de innovación por parte de los docentes en pandemia. Fuente: PulsoEstudiantil 2020, seguimosvirtual.com	17
Figura 7: Resultados sobre la cantidad de interacción de estudiantes de educación superior en la pandemia. Fuente: PulsoEstudiantil 2020, seguimosvirtual.com	17
Figura 8: Árbol de causas y efectos del problema. Fuente: Elaboración propia.....	20
Figura 9: Función Objetivo genérica de programación lineal. Fuente: Apuntes de INF-292 Optimización.....	21
Figura 10: Pseudocódigo de método <i>init</i> . Fuente: Elaboración propia.....	45
Figura 11: Pseudocódigo de método <i>clear_fields</i> . Fuente: Elaboración propia.	45
Figura 12: Pseudocódigo de método <i>generate_problem</i> . Fuente: Elaboración propia.....	46
Figura 13: Pseudocódigo de método <i>convert_to_format</i> . Fuente: Elaboración propia.	46
Figura 14: Pseudocódigo de método <i>export_problem</i> . Fuente: Elaboración propia.	47
Figura 15: Pseudocódigo de método <i>convert_to_lp_solve</i> . Fuente: Elaboración propia. ...	47
Figura 16: Pseudocódigo de método <i>convert_to_mps</i> . Fuente: Elaboración propia.....	48
Figura 17: Pseudocódigo de método <i>convert_to_lindo</i> . Fuente: Elaboración propia.	48

Figura 18: Pseudocódigo de método <i>generar_PL</i> . Fuente: Elaboración propia.	49
Figura 19: Pseudocódigo de método <i>pl_solver</i> . Fuente: Elaboración propia.	50
Figura 20: Pseudocódigo de método <i>generar_PLB</i> . Fuente: Elaboración propia.	51
Figura 21: Pseudocódigo de método <i>plb_solver</i> . Fuente: Elaboración propia.	52
Figura 22: Pseudocódigo de método <i>generar_PLE</i> . Fuente: Elaboración propia.	53
Figura 23: Pseudocódigo de método <i>ple_solver</i> . Fuente: Elaboración propia.	54
Figura 24: Pseudocódigo de método <i>generar_PLM</i> . Fuente: Elaboración propia.	55
Figura 25: Pseudocódigo de método <i>plm_solver</i> . Fuente: Elaboración propia.	56

INDICE DE TABLAS

Tabla 1: Configuración de Pruebas. Fuente: Elaboración propia.	74
Tabla 2: Tiempos promedios de generación de problemas. Fuente: Elaboración propia. ..	75
Tabla 3: Tiempos promedio de resolución de problemas. Fuente: Elaboración propia.	76

INTRODUCCIÓN

La programación lineal es una herramienta fundamental en múltiples disciplinas, incluyendo la ingeniería, la economía y la investigación de operaciones, ya que permite modelar y resolver problemas de optimización en los que intervienen variables y restricciones lineales. Su uso es particularmente importante en el ámbito académico, donde los estudiantes de carreras relacionadas deben desarrollar una comprensión tanto teórica como práctica de esta técnica. Sin embargo, uno de los grandes desafíos en la enseñanza de programación lineal es la generación de problemas diversos y complejos que permitan a los estudiantes practicar y desarrollar sus habilidades en diferentes contextos.

En la asignatura de Optimización del departamento de informática de la Universidad Técnica Federico Santa María, esta necesidad es particularmente evidente, ya que la creación manual de problemas es un proceso tedioso y limitado en cuanto a variedad y dificultad. Además, la generación repetitiva de problemas idénticos puede llevar a que los estudiantes enfoquen su estudio en patrones específicos de solución, sin explorar ampliamente las aplicaciones prácticas de la programación lineal.

Frente a estos desafíos, el presente escrito propone el desarrollo de una herramienta que automatice la generación de problemas de programación lineal, permitiendo al cuerpo docente y estudiantil contar con una fuente de problemas variados y personalizados, ajustados a diferentes niveles de dificultad y tipos de variables (continuas, enteras y binarias). La solución se implementa en Python, aprovechando bibliotecas como OR-Tools para conectar el modelo con *solvers* de optimización, y brinda la posibilidad de exportar los problemas en formatos estándar como MPS, lp_solve y LINDO, lo cual amplía su aplicabilidad en diferentes plataformas.

El objetivo de esta herramienta es doble: por un lado, optimizar el tiempo y esfuerzo de los docentes en la preparación de ejercicios, y por otro, ofrecer a los estudiantes una oportunidad de aprendizaje adaptable y diversa, que fomente su capacidad de análisis y resolución de problemas en el contexto de la programación lineal.

CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA

1.1 Contexto

A comienzos del año 2020, la pandemia de *Sars-cov-2*, o popularmente llamado *Covid-19*, causó revuelo y problemas a nivel mundial. La cual, en distintas instituciones y/o empresas se vieron en la necesidad de innovar en sus tecnologías y en sus metodologías de trabajo. Esto con la finalidad, de mantener el distanciamiento social y evitar contagios, la solución más factible fue el teletrabajo. Debido a lo anterior, se tomó como medida, tanto en los colegios como en las instituciones de educación superior, impartir clases *online*, motivo por el cual las plataformas y herramientas *online* tomaron un rol protagónico a la hora de enseñar y evaluar a los estudiantes.

En el caso de la Universidad Técnico Federico Santa María (UTFSM), a través de la plataforma “Aula” se realizaron gran parte de las evaluaciones entre los años 2020 y 2021, por lo que la tarea de cambiar la metodología de enseñanza y de evaluación fue un gran desafío para los profesores, tanto del Departamento de Informática (DI) como de la UTFSM en sí. En primer lugar, el primer desafío fue, por el lado de los profesores, ya que hubo un cambio de tiempo requerido a la hora de crear evaluaciones, esto con la finalidad de evitar fraude académico de los alumnos, agregándole además el tener presente que se debía corroborar que los enunciados de las evaluaciones tuviesen distintas versiones para apoyar lo antes mencionado. Es por esto que, nace la necesidad de automatizar el proceso de creación de evaluaciones de forma aleatoria en la asignatura INF-292 Optimización.

Sumado a lo mencionado anteriormente, otro factor influyente con esta temática, fueron los alumnos de las instituciones académicas, quienes tuvieron que modificar su forma de relacionarse con los profesores y/o ayudantes, ya que se encontraban en el desafío de aprender desde sus hogares y de manera autodidacta el contenido académico, el cual no siempre fue fácil de entender y aplicar. Dado esto, que se crea la importancia de generar una herramienta dirigida hacia la asignatura de INF-292 Optimización. Lo cual conlleva, permitirles a los estudiantes, aplicar lo aprendido mediante ejercitación y que a través de dicha herramienta puedan comprender y aprender la materia que será evaluada.

1.2 Situación actual

1.2.1 Estudiantes de educación superior en contexto de pandemia

Desde la fecha, 15 de junio del año 2020 la plataforma de apoyo para la virtualización de las clases creada por ex alumnos de la Universidad de Chile, “Seguimos virtual”, entregó los resultados de la encuesta “Pulso Estudiantil”. Dicha encuesta fue un estudio nacional e independiente que pretendía entender cómo el *Covid-19* impactó en la vida de los estudiantes de educación superior y cuáles fueron los desafíos para las instituciones. Entre

otros aspectos, también analizó los hábitos de estudio, el bienestar y el aprendizaje de los estudiantes de educación superior en Chile [SeguimosVirtual20].

Es así como en este estudio participaron de forma voluntaria 2.650 estudiantes de más de 100 carreras en 15 regiones del país, representando así a 67 casas de estudio públicas, privadas, institutos profesionales y centros de formación técnica. Siendo así, que dentro de los hallazgos principales de esta encuesta, se encuentran los siguientes resultados:

En promedio, el 80% de los estudiantes de educación superior nunca había tomado un curso virtual antes de esta pandemia.

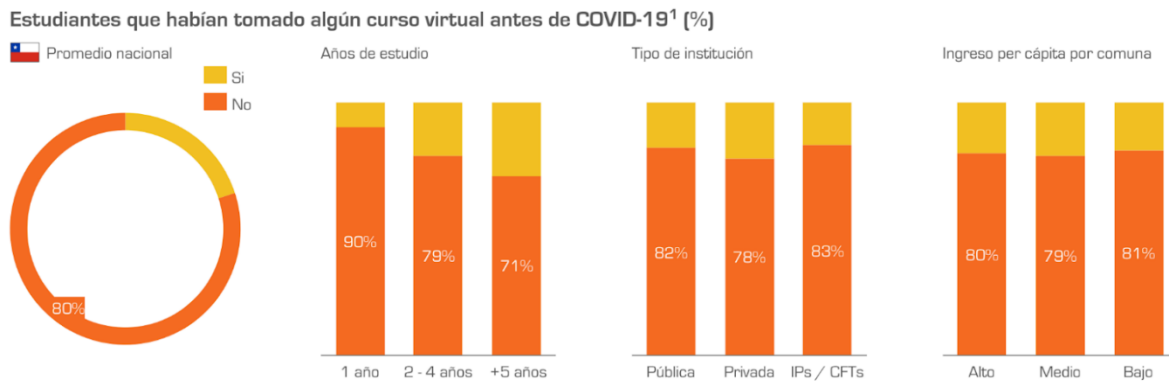


Figura 1: Resultados sobre si los estudiantes habían tomado un curso online antes de la pandemia. Fuente: PulsoEstudiantil 2020, seguimosvirtual.com

En cuanto a la relación con sus casas de estudio, el 60% de los estudiantes declara estar insatisfecho con la respuesta institucional frente a la crisis, mientras que un 47% dice que su casa de estudios se ha preocupado por asegurar que los estudiantes cuenten con los recursos mínimos. Además, el 54% de los estudiantes está satisfecho con el esfuerzo realizado por sus docentes.

Satisfacción con la respuesta de actores claves durante el COVID-19¹ (% estudiantes)

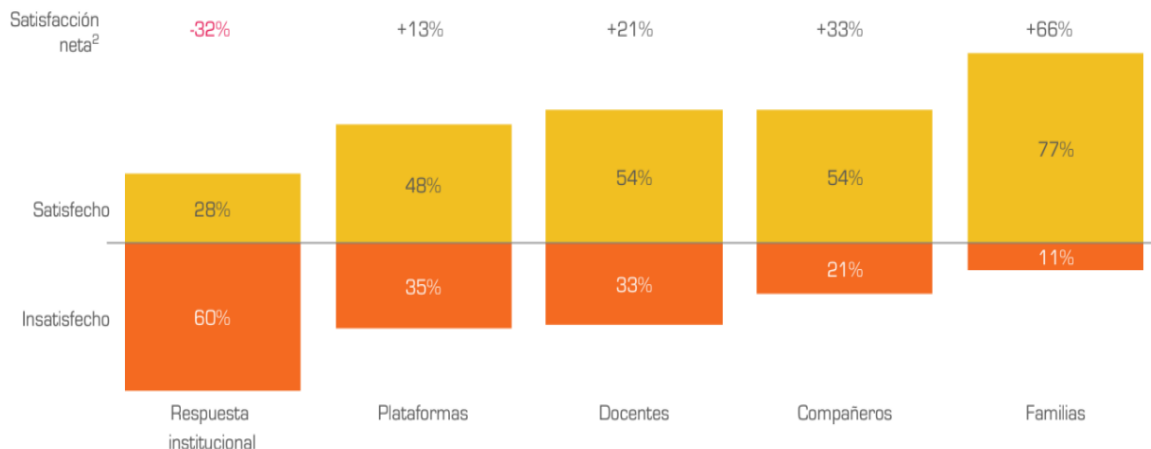
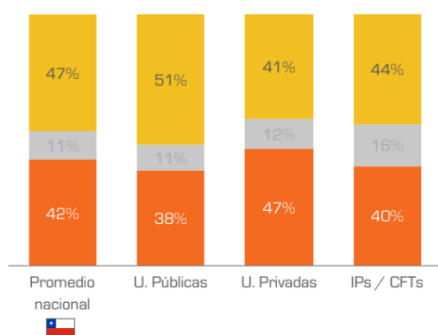


Figura 2: Resultados sobre la satisfacción de los estudiantes con la respuesta de actores claves durante la pandemia. Fuente: PulsoEstudiantil 2020, seguimosvirtual.com

Estudiantes que declaran que su casa de estudio se ha preocupado por asegurar recursos mínimos¹ (%)



Estudiantes que declaran que su casa de estudio los ha ayudado a ser más efectivos(as)¹ (%)

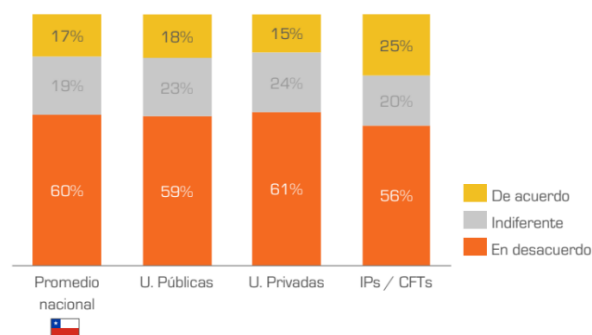


Figura 3: Resultados sobre apoyos de casas estudios hacia sus estudiantes. Fuente: PulsoEstudiantil 2020, seguimosvirtual.com

Cabe destacar que, el 81% de los estudiantes de educación superior chilena considera que están recibiendo una educación de peor calidad en comparación a la modalidad presencial.

Estudiantes que declaran recibir una educación de peor calidad¹ [%]

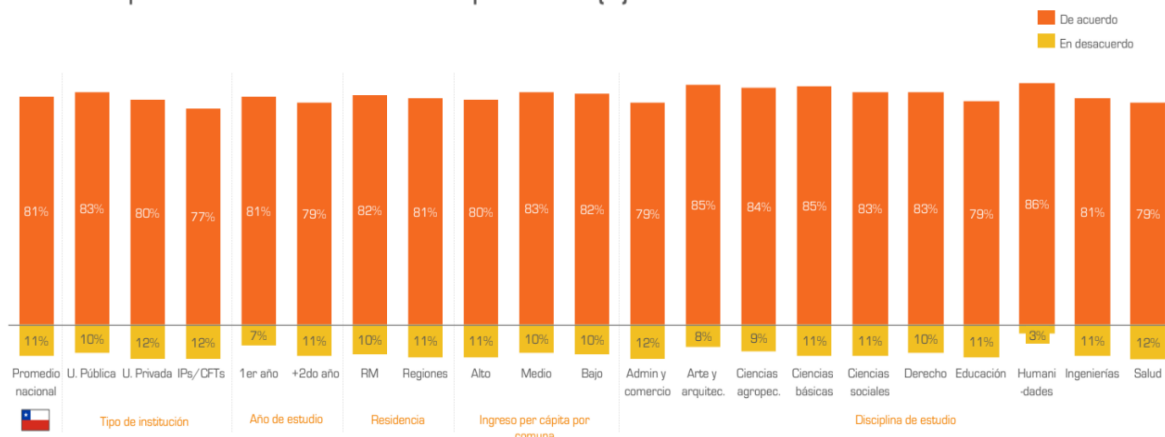


Figura 4: Resultado de estudiantes sobre la calidad de educación que éstos reciben.

Fuente: PulsoEstudiantil 2020, seguimosvirtual.com

A pesar de la insatisfacción con las medidas adoptadas por sus instituciones para adaptar sus procesos formativos durante la crisis, el 31% asegura que tomaría cursos virtuales el próximo semestre.

Intención de tomar cursos virtuales posterior al COVID-19 [%]

¿Tomarías alguno de tus cursos en modalidad virtual? ¹

Pasada la crisis, un 31% de las y los estudiantes tomaría cursos virtuales en su casa de estudio

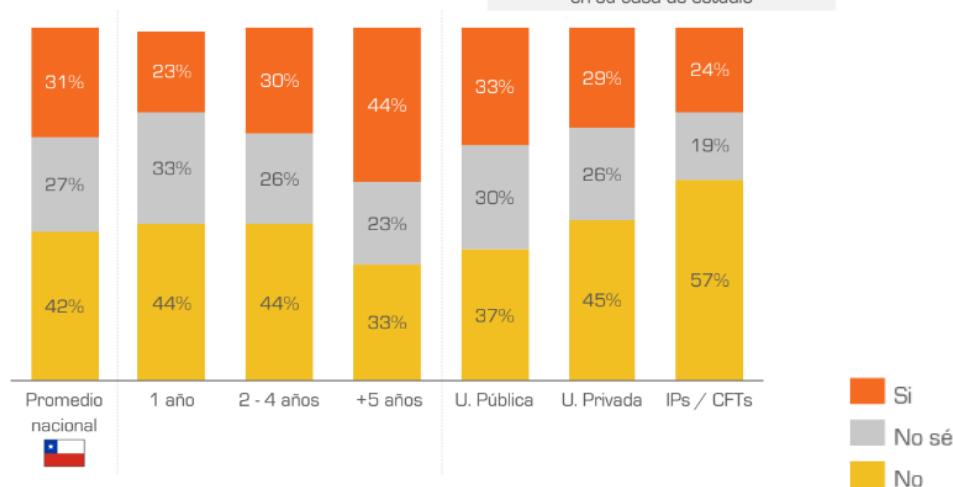


Figura 5: Resultados sobre si los estudiantes tomaran algunos de sus cursos en modalidad online post pandemia. Fuente: PulsoEstudiantil 2020, seguimosvirtual.com

Respecto de las innovaciones docentes, el 59% valora que suban los videos de las clases para revisarlos posteriormente.

Utilidad de innovaciones docentes durante el COVID-19¹ (% estudiantes)

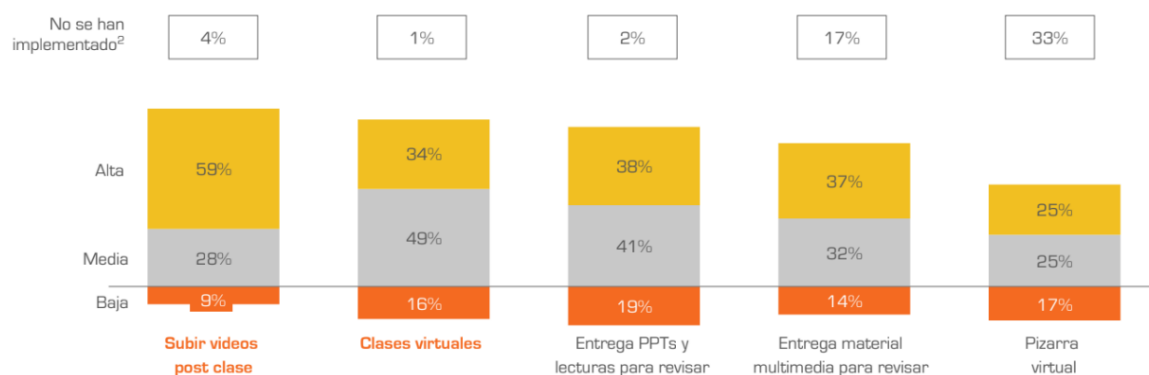


Figura 6: Resultados sobre utilidad de innovación por parte de los docentes en pandemia.

Fuente: PulsoEstudiantil 2020, seguimosvirtual.com

Un punto para considerar es que el 72% de los encuestados declara que existe una disminución en su relación con los académicos, lo cual es transversal según carrera, año de estudio, nivel socioeconómico, lugar de residencia y tipo de institución.

Cambio en la cantidad de interacciones producto del COVID-19¹ (%)

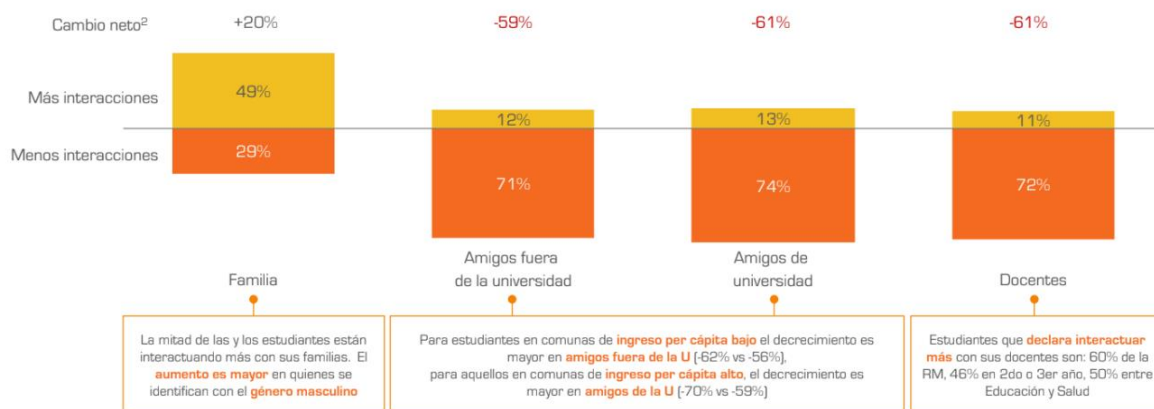


Figura 7: Resultados sobre la cantidad de interacción de estudiantes de educación superior en la pandemia. Fuente: PulsoEstudiantil 2020, seguimosvirtual.com

1.2.2 Profesores en contexto de pandemia

Con el propósito de conocer la realidad de profesores y comunidad educativa frente al desafío que trajo consigo la pandemia, la “Fundación Efecto Educativo” llevó a cabo un estudio, el cual fue publicado el mes de julio del año 2021, cuyo objetivo fue obtener un catastro de la visión y experiencia de profesores, docentes y directivos, sobre la actual situación de la educación en sus establecimientos. En relación a este análisis se centró en la

experimentación y el acceso a nuevas tecnologías y métodos tecnológicos como forma de entregar clases de cátedra [Álvarez21].

Por lo que, este escenario reveló que el 56% de los profesores está más cansado, tanto física como emocionalmente en una escala de 1 a 5 en comparación al mismo período antes de la pandemia. Por otro lado, la experimentación y el acceso a nuevos métodos tecnológicos son factores fundamentales a la hora de analizar si los profesores, docentes y/o directivos podrían enfrentar de manera exitosa esta nueva forma de enseñar. Siendo así que, el 49,3% declaró que no contaban con estos recursos mientras que el 50,7% dijo que sí.

Dentro de las conclusiones del estudio fue que, el 61,3% de los docentes encuestados considera que los alumnos, en ocasiones, eran apoyados por los adultos que viven con ellos para realizar las actividades evaluativas en casa.

Por otra parte, los encuestados creen que la mejor manera de fomentar la autonomía en los alumnos en la modalidad de clases “*online*” a través de trabajos reflexivos y grupales con producto en las clases, fomentando el responsable y correcto uso de los equipos, como computadores, *tablets*, celulares, etc. Asimismo, señalan la importancia para un buen aprendizaje las rutinas de trabajo, el análisis, la creatividad y la búsqueda de información a través de motores de navegación, investigación y segregación de información de las fuentes a las que acceden, potenciando a los estudiantes a estar más pendientes de sus tareas a través de las plataformas de cada establecimiento [Martínez21].

Una de las preguntas que fue realizada a los encuestados fue “¿Qué estrategias pedagógicas considera que le han dado mejor resultado en las clases *online*?”. En esta pregunta punto se obtuvieron las siguientes respuestas:

- El 78,7% declaró que la estrategia que mejor resultado les dio en metodologías *online* fue la incorporación de herramientas.
- El 41,3% dijo que fue el envío de materiales y guías
- El 32% mencionó que la mejor estrategia fue la aplicación de metodologías concretas para poder llevar a cabo de manera exitosa el aprendizaje de alumnos.

Finalmente, la realidad tecnológica vivida por los profesores y directivos, la manera en como la incorporaron a sus clases y así también, traspasarla a sus alumnos y familias, también fue abordada por la encuesta. En este sentido, el 80% de los encuestados considera que la forma de hacer clases evoluciona junto con las tecnologías. Sin embargo, el 44% de ellos declara no contar con las tecnologías propicias para realizar una óptima clase virtual. Asimismo, la mayoría de los entrevistados dice que una de las medidas que su establecimiento ha implementado para brindar apoyo con soluciones tecnológicas y digitales (que les permiten enfrentar las clases a distancia) es la entrega de computadores (70%), *tablets* (53,3%), *módems/routers* (48%), cámaras de video y sistemas de audio (40% cada uno) a los alumnos.

1.2.3 Ramo Optimización en pandemia

Debido a la metodología de educación a distancia tomada por la UTFSM, los profesores del ramo INF-292 Optimización del DI, liderados por el profesor Carlos Castro, se vieron en la necesidad de hacer uso de la plataforma “Aula” para realizar las evaluaciones del ramo. Sin embargo, esto trajo consigo un trabajo extra a la hora de realizar los enunciados de las evaluaciones (tal como se menciona en la sección 1.1). Debido a lo mencionado anteriormente, los profesores propusieron como tema de investigación, crear una herramienta que, a través del ingreso de parámetros, genere problemas de programación lineal de forma aleatoria. Además de ayudarles a elaborar enunciados de forma más automatizada, los profesores buscaban que esta herramienta en cuestión fuera de utilidad también para los estudiantes que estuviesen cursando el ramo, de tal manera que pudieran hacer uso de ella a la hora de estudiar y, así, aprender de forma didáctica la materia que posteriormente les sería evaluada.

1.3 Objetivos

1.3.1 Objetivo general

Desarrollar un *software* para colaborar en el proceso de aprendizaje de docentes y estudiantes del DI, el cual genera problemas de programación lineal en forma automática, a partir de parámetros, para la cátedra de la asignatura INF-292 Optimización.

1.3.2 Objetivos específicos

- Crear una aplicación flexible que permita la generación de modelos de PL.
- Integrar la aplicación con *solvers* externos.
- Estudiar y documentar las características de las soluciones generadas.
- Evaluar la factibilidad y documentar la experiencia del desarrollo de la aplicación.
- Evaluar posibilidad de establecer propiedades a los modelos generados.

A continuación, se adjunta el árbol que resume las causas y efectos del problema.

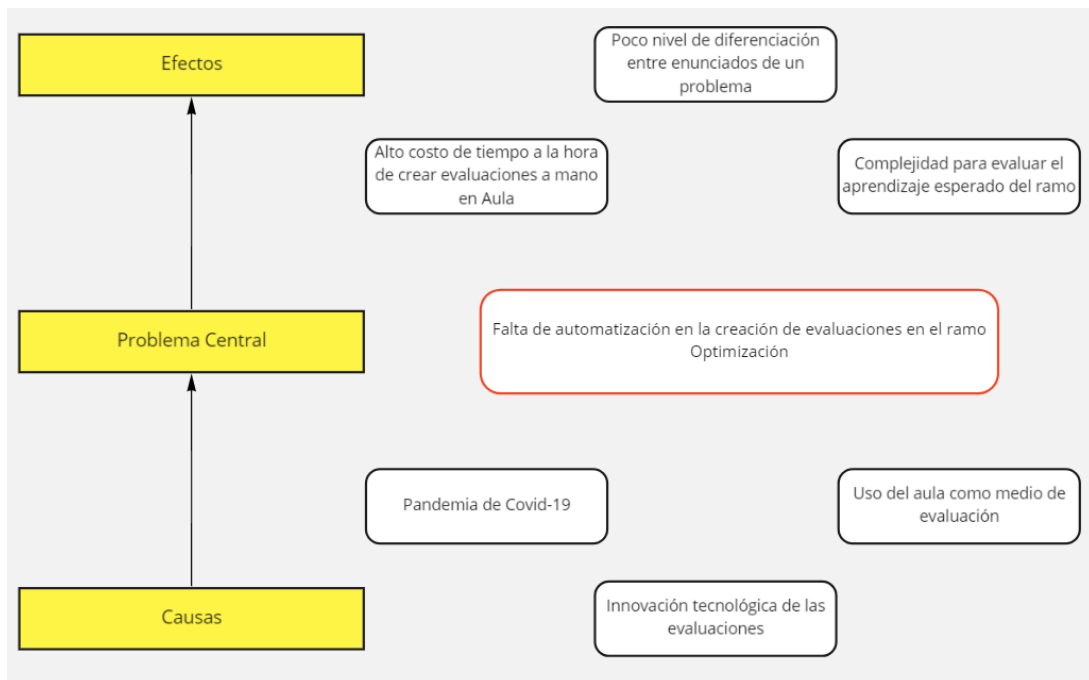


Figura 8: Árbol de causas y efectos del problema. Fuente: Elaboración propia.

CAPÍTULO 2: MARCO CONCEPTUAL

2.1 Programación Lineal

La programación lineal (PL) se utiliza en campos como la ingeniería, la economía, la gestión, y muchas otras áreas de la ciencia, la técnica y la industria [Castillo02]. En este tipo de problemas se cuenta con los siguientes elementos, los cuales permiten modelar matemáticamente el problema de PL:

- 1) **Función Objetivo:** Función lineal que, dependiendo del problema y del modelo, debe ser maximizada (ser incrementada lo más posible) o minimizada (ser reducida lo más posible), de tal manera que se encuentren valores que optimicen su solución.
- 2) **Variables de decisión:** Representan las incógnitas que deben determinarse.
- 3) **Restricciones:** Se describen en forma de un sistema de ecuaciones y/o inecuaciones lineales, las cuales limitan los valores de las variables de decisión.

El objetivo de la PL consiste en encontrar el óptimo, es decir, el máximo o mínimo, de una función lineal en un conjunto que puede expresarse como la intersección de un número finito de hiperplanos y semiespacios en R^n .

Uno de los puntos que es importante destacar es que la PL posee dos elementos fundamentales que la componen como tal:

- 1) Región factible
- 2) Restricciones estructurales y de no negatividad.

Por un lado, a las restricciones se les llama **restricciones de no negatividad** y, además, como condiciones del modelo matemático de PL se estipula que las variables de decisión del problema solo deben tener valores mayores o iguales a cero. Por otro lado, el conjunto de valores que satisfacen todas las restricciones es conocido como la **región factible**, la cual corresponde al espacio de solución de todos los puntos posibles de un problema de optimización que logra satisfacer las restricciones que el problema posee, incluyendo también las potencialidades, igualdades y restricciones enteras.

Un ejemplo genérico de un problema de programación lineal sería el siguiente:

$$\begin{array}{ll} \text{Función a optimizar :} & f(x, y) = Ax + By \\ & \left. \begin{array}{l} a_1x + b_1y > c_1 \\ a_2x + b_2y \leq c_2 \\ \vdots \end{array} \right\} \\ \text{Restricciones :} & \end{array}$$

Figura 9: Función Objetivo genérica de programación lineal. Fuente: Apuntes de INF-292 Optimización.

2.1.1 Definición de un modelo matemático

La investigación de operaciones reformula de manera conveniente un problema del mundo real y lo transforma en una expresión matemática para analizarlo y solucionarlo. Para lograr dicho objetivo, esto se hace mediante la construcción de un modelo matemático que represente la esencia del problema.

Por definición, los modelos matemáticos son modelos que utilizan fórmulas matemáticas para representar la relación entre distintas variables, parámetros y restricciones [Roldán19]. Es decir, un modelo matemático es una representación simplificada y abstraída de un sistema, proceso o fenómeno del mundo real utilizando conceptos y herramientas matemáticas. Se utiliza para comprender, analizar y predecir el comportamiento del sistema o resolver problemas relacionados. En general, un modelo matemático consiste en:

1. **Variables:** Representan las cantidades o características relevantes del sistema que se están modelando. Pueden ser variables numéricas, cualitativas, discretas, continuas, etc.
2. **Relaciones:** Establecen las interacciones y dependencias entre las variables del modelo. Estas relaciones pueden ser expresadas mediante ecuaciones, desigualdades, funciones, algoritmos u otras representaciones matemáticas.
3. **Parámetros:** Son valores constantes que influyen en el modelo y pueden ajustarse para reflejar diferentes escenarios o condiciones.
4. **Restricciones:** Limitaciones o condiciones que deben cumplirse en el modelo. Estas restricciones pueden ser de naturaleza física, económica, tecnológica u otras, y definen las limitaciones del sistema.
5. **Función objetivo:** Representa el objetivo o criterio que se busca optimizar en el modelo. Puede ser una medida de rendimiento, costo, utilidad, eficiencia, entre otros.

Por lo que, al combinar variables, relaciones, parámetros, restricciones y función objetivo, un modelo matemático proporciona una representación simplificada del sistema real que permite analizar su comportamiento, hacer predicciones, tomar decisiones o encontrar soluciones óptimas. Para esto, es importante considerar que un modelo matemático es una abstracción y simplificación del mundo real, lo que implica que puede haber suposiciones y aproximaciones. Ya que, la validez y utilidad de un modelo dependen de la adecuación de las simplificaciones realizadas y la correspondencia entre los resultados del modelo y la realidad.

2.1.2 Formulación de un modelo matemático

Definir un modelo matemático implica formular una representación matemática de un problema o situación real con el fin de analizarlo y tomar decisiones basadas en los

resultados obtenidos. Para ello, se cuenta con los siguientes pasos generales para definir un modelo matemático en investigación de operaciones:

1. **Identificar el problema:** Se debe comenzar por comprender y definir claramente el problema que se desea abordar en la investigación de operaciones. Se deben definir los objetivos, las restricciones y los parámetros relevantes del problema.
2. **Determinar las variables:** Tras identificar el problema, es necesario identificar las variables que intervienen en el problema. Estas pueden ser medidas, cantidades o características que afectan o se ven afectadas por las decisiones que se toman.
3. **Establecer las relaciones:** Se deben determinar las relaciones matemáticas entre las variables identificadas. Estas relaciones pueden ser lineales, no lineales, discretas, continuas, etc. Se puede hacer uso de ecuaciones, desigualdades, funciones, restricciones y otras representaciones matemáticas para definir las relaciones entre las variables.
4. **Formular la función objetivo:** Tras establecer las relaciones matemáticas entre las variables, se debe definir una función objetivo que represente el objetivo principal del problema. Puede ser una función que se deba maximizar o minimizar, dependiendo del objetivo específico del problema.
5. **Considerar las restricciones:** Identificar y formular las restricciones que limitan las soluciones factibles del problema. Estas restricciones pueden ser restricciones de capacidad, restricciones de tiempo, restricciones de recursos, restricciones tecnológicas, entre otras.
6. **Validar y ajustar el modelo:** Se verifica que el modelo matemático sea válido y represente adecuadamente el problema. Para ello se pueden realizar pruebas y validaciones para asegurarse de que el modelo capture todas las características y restricciones importantes.
7. **Implementar el modelo:** Una vez que el modelo matemático esté formulado y validado, se puede implementar en un software de optimización o utilizar técnicas de resolución de problemas adecuadas para analizar y obtener soluciones del modelo.

2.2 Métodos de Resolución de Problemas de Programación Lineal

Tras haber logrado definir previamente el problema de PL, se formula el modelo matemático que permite la comprensión de dicho problema para su posterior solución. Hoy en día existen diversos métodos que permiten encontrar la solución (o no solución) de cualquier problema de PL y, destacándose los siguientes métodos de solución:

- 1) Método gráfico.
- 2) Método Simplex.
- 3) Técnicas de Ramificación y Acotamiento para PL entera y PL entera mixta.

2.2.1 Método gráfico

El método gráfico se basa en la representación gráfica de las restricciones y la función objetivo del problema [Vanderbei13]. Las restricciones suelen estar representadas por un sistema de desigualdades lineales, mientras que la función objetivo está representada por una ecuación lineal. Para poder modelar un problema de programación lineal a través del método gráfico se hará una explicación paso a paso:

- 1) Formulación del problema:
 - Escribir la función objetivo en términos de las variables de decisión.
 - Escribir las restricciones como desigualdades lineales que involucran las variables de decisión.
- 2) Trazado de las restricciones:
 - Convertir cada restricción en una igualdad reemplazando el signo de desigualdad con un signo de igualdad.
 - Trazar líneas o curvas correspondientes a cada restricción en un sistema de coordenadas.
 - Determinar la región factible, que es la región donde todas las restricciones se satisfacen simultáneamente. Por lo general, es la intersección de las regiones de restricción individuales.
- 3) Identificación de la región factible:
 - Sombrear la región factible en el gráfico. Esta región representa todos los puntos que satisfacen todas las restricciones.
- 4) Determinación de la solución óptima:
 - Se determinan las coordenadas de los puntos de las esquinas (vértices) de la región factible.
 - Se evalúa la función objetivo en cada punto de esquina.
 - Identificar el punto de esquina que optimiza la función objetivo en función de si se trata de un problema de maximización o minimización.
 - Tras encontrar los vértices de la región factible, cada uno de ellos será evaluado en la función objetivo.

5) Interpretación de los resultados:

- La solución óptima es el punto de la esquina que arroja el valor más alto o bajo de la función objetivo, según el tipo de objetivo (maximización o minimización).
- Si la función objetivo tiene una solución óptima única, se ubicará en uno de los puntos de esquina.
- Si la función objetivo es paralela a una de las líneas de restricción, el problema tiene infinitas soluciones óptimas.

El método gráfico es particularmente útil para obtener información sobre la geometría del problema y comprender la relación entre la función objetivo y la región factible. Sin embargo, tiene limitaciones cuando se trata de problemas que involucran más de dos variables de decisión o restricciones complejas. Siendo así que, para problemas de dimensiones superiores, el método gráfico se vuelve poco práctico y se prefieren otras técnicas como el método *Simplex* o los métodos de puntos interiores.

2.2.2 Método Simplex

El Método Simplex es un algoritmo utilizado en la optimización matemática para resolver problemas de programación lineal. Este método fue desarrollado por George Dantzig en 1947 y es considerado uno de los algoritmos más eficientes y confiables en este campo [Dantzig51]. Se basa en el principio de que una solución óptima a un problema de programación lineal, si existe, está ubicada en uno de los vértices del polítopo (figura geométrica con un alto grado de simetría) formado por las restricciones. Es decir, si se encuentra una solución inicial que sea un vértice factible, se puede mover de vértice en vértice a lo largo del borde del polítopo, hasta llegar al vértice que proporciona el valor óptimo [Chvátal83].

2.2.2.1 Formulación General

$$\text{Max o Min } (z) = c_1x_1 + c_2x_2 + \cdots + c_nx_n$$

Sujeto a:

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2$$

⋮

$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m$$

$$x_1, x_2, \dots, x_n \geq 0$$

Donde:

- z es la función objetivo que se busca maximizar o minimizar.
- $c_1, c_2, \dots, c_n$ son los coeficientes de la función objetivo.
- a_{ij} son los coeficientes de las restricciones.
- $b_1, b_2, \dots, b_n$ son los términos independientes de las restricciones.
- $x_1, x_2, \dots, x_n$ son las variables de decisión.

2.2.2.2 Principio del Método Simplex

El Método Simplex trabaja de manera iterativa, explorando los vértices de la región factible del problema de programación lineal. Cada vértice representa una solución factible, y el algoritmo avanza de vértice en vértice en busca del que maximice (o minimice) la función objetivo. El proceso continúa hasta que no se puede mejorar más la función objetivo, lo que indica que se ha encontrado la solución óptima. En cada iteración, el algoritmo Simplex sigue los siguientes pasos:

1. **Identificar una solución básica inicial:** Simplex requiere una solución básica inicial que sea factible. En muchos casos, esta solución se encuentra utilizando variables de holgura (o artificiales), que transforman las desigualdades en igualdades.
2. **Verificar la optimalidad:** Se evalúa la fila de la función objetivo (fila z) en la tabla Simplex para determinar si la solución actual es óptima. Si todos los coeficientes de la fila z son no negativos (en problemas de maximización), la solución es óptima.
3. **Selección de la variable de entrada:** Si la solución actual no es óptima, se elige una variable de entrada que mejorará la función objetivo. Esta variable corresponde al coeficiente más negativo en la fila z (para problemas de maximización).
4. **Selección de la variable de salida:** Se determina qué variable sale de la base. Para esto, se calcula el cociente mínimo entre los valores de la columna de soluciones y los coeficientes de la variable de entrada. La fila con el cociente más pequeño indica qué variable deja la base.
5. **Realización del pivote:** Se realiza un pivoteo en la tabla Simplex, actualizando los valores de la tabla de acuerdo con la nueva variable de entrada y salida.
6. **Repetición del proceso:** El proceso se repite hasta que no se pueden hacer más mejoras a la función objetivo, lo que indica que se ha alcanzado la solución óptima.

2.2.2.4 Conclusión

El Método Simplex es un algoritmo poderoso y eficiente para resolver problemas de PL. A pesar de que su peor caso no es polinomial, en la práctica, se ha demostrado que el método es capaz de resolver problemas con miles de variables y restricciones en un tiempo razonable. Debido a su importancia en el ámbito de la optimización, sigue siendo una

herramienta clave tanto en la investigación operativa como en aplicaciones industriales y académicas.

2.3 Técnicas Avanzadas en Programación Lineal

En esta sección, se explorarán técnicas avanzadas que amplían la aplicación de la programación lineal estándar y permiten la resolución de problemas más desafiantes y diversas aplicaciones. Estas técnicas se destacan por su capacidad para modelar y resolver problemas que involucran variables enteras, mixtas y binarias.

2.3.1 Programación Lineal Entera

La **Programación Lineal Entera (PLE)** es una extensión de la PL en la que se restringe que las variables de decisión tomen únicamente valores enteros en lugar de continuos. En otras palabras, en la PLE, las soluciones sólo son consideradas válidas si todas las variables tienen valores enteros. Esto agrega una dimensión discreta a los problemas de optimización, lo que a menudo refleja mejor la naturaleza de muchas decisiones en la vida real.

2.3.1.1 Definición de un Problema PLE

Un problema de PLE se formula de manera similar a un problema de programación lineal, con la diferencia de que las variables de decisión deben ser enteras:

$$\text{Max o Min } (z) = c_1x_1 + c_2x_2 + \cdots + c_nx_n$$

Sujeto a

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2$$

$\vdots$

$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m$$

$$x_1, x_2, \dots, x_n \in \mathbb{Z}$$

Donde las variables $x_1, x_2, \dots, x_n$ deben tomar valores enteros.

2.3.1.2 Características de la Programación Lineal Entera

1. **Naturaleza Discreta:** En PLE, las soluciones solo pueden tomar valores enteros, lo que puede ser más realista en ciertos contextos. Por ejemplo, en la planificación de producción, no se pueden producir cantidades fraccionarias de un artículo.
2. **Complejidad Aumentada:** La restricción de que las variables deben ser enteras introduce una mayor complejidad en la resolución del problema. Mientras que la programación lineal continua se puede resolver eficientemente con el método simplex, la PLE puede requerir métodos más especializados.
3. **Aplicaciones Prácticas:** La PLE encuentra aplicaciones en una amplia gama de áreas, como la logística, la planificación de proyectos, la asignación de recursos, la asignación de personal y la ubicación de instalaciones. En estos casos, las soluciones enteras tienen un significado práctico.

2.3.1.3 Métodos de Resolución de PLE

1. **Método de Ramificación y Acotación (*Branch and Bound*):** Este método divide el problema en subproblemas más pequeños, explorando diferentes combinaciones de valores enteros para las variables en cada subproblema. Se utiliza para buscar la solución óptima en un espacio de búsqueda exponencial. Comienza con una relajación del problema y construye un árbol con soluciones enteras particionando el conjunto de soluciones factibles de modo de descartar soluciones fraccionarias [Goic08].
2. **Método de Corte de Planos (*Cutting Planes*):** Este método trata de la construcción aproximada del casco convexo de las soluciones factibles de un problema PLE [Testuri12]. Mediante un proceso iterado de inclusión de inecuaciones (planos de corte), se busca ajustar la formulación del problema de forma. Agrega restricciones adicionales al modelo de PLE para eliminar soluciones enteras inválidas. Estos cortes pueden ser obtenidos de diversas fuentes, como el relajamiento de restricciones o la aplicación de desigualdades válidas.

2.3.1.5 Limitaciones y Consideraciones

Dentro de las limitaciones y consideraciones, la PLE enfrenta desafíos de complejidad, ya que encontrar la solución óptima en problemas grandes puede ser computacionalmente costoso. Además, la PLE es un problema NP-completo, lo que significa que no se conocen algoritmos eficientes que puedan resolver todos los casos en un tiempo razonable.

No obstante, a pesar de estas limitaciones, la PLE sigue siendo una herramienta significativa para modelar situaciones en las que las decisiones deben tomarse en términos enteros. Dado que, la combinación de técnicas de resolución, heurísticas y métodos aproximados ha permitido abordar problemas prácticos de programación lineal entera de manera efectiva.

2.3.2 Programación Lineal Mixta

La programación lineal mixta (PLM) combina la flexibilidad de las variables continuas con la rigidez de las variables enteras, es decir, las variables de decisión pueden ser enteras y otras ser continuas. Siendo así que, la PLM encuentra aplicaciones en una variedad de áreas, desde la logística hasta la planificación de recursos. Sin embargo, a pesar de su versatilidad, la PLM también puede presentar retos computacionales debido a la naturaleza mixta de las soluciones.

2.3.2.1 Formulación de un Problema PLM

Un problema de PLM típicamente se formula de la siguiente manera:

$$\text{Max o Min } (z) = c_1x_1 + c_2x_2 + \cdots + c_nx_n$$

Sujeto a

$$a_{11}x_1 + a_{12}x_2 + \cdots a_{1n}x_n \leq b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots a_{2n}x_n \leq b_2$$

$\vdots$

$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m$$

$$x_j \in \mathbb{Z}, \quad \forall j \in J$$

$$x_k \geq 0, \quad \forall k \notin J$$

Donde J es el conjunto de índices de las variables enteras [Williams13].

2.3.2.2 Métodos de Solución de PLM

Dado que la resolución de problemas de PLM es computacionalmente más difícil que la de problemas de programación lineal, puesto que la naturaleza discreta de las variables enteras, métodos como el de ramificación y acotamiento (*Branch and Bound*) y cortes (*Cutting Planes*) son comúnmente utilizados para encontrar soluciones óptimas [Schrijver98].

2.3.2.3 Aplicaciones

Dentro de los campos utilizados de los PLM son ampliamente utilizados está el de producción, donde se pueden modelar problemas de programación de la producción y

diseño de redes de suministro. También es útil en la asignación óptima de recursos y la planificación financiera, donde se busca maximizar el retorno de la inversión sujeto a restricciones financieras y de riesgo [Wolsey98].

2.3.3 Programación Lineal Binaria

La programación lineal binaria (PLB) es una variante especializada de la PLM, donde las variables solo pueden tomar valores binarios, es decir, 0 o 1. Esta restricción en los valores de las variables encuentra aplicaciones en problemas de toma de decisiones y selección, como asignación de recursos y diseño de redes. La PLB puede simplificar la modelización y la resolución, pero a menudo introduce problemas de complejidad al explorar todas las combinaciones posibles.

2.3.3.1 Formulación de un Problema PLB

$$\text{Max o Min } (z) = c_1x_1 + c_2x_2 + \cdots + c_nx_n$$

Sujeto a

$$a_{11}x_1 + a_{12}x_2 + \cdots a_{1n}x_n \leq b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots a_{2n}x_n \leq b_2$$

$\vdots$

$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m$$

$$x_1, x_2, \dots, x_n \in \{0,1\}$$

2.3.3.2 Métodos de Solución de PLB

Los problemas PLB son NP-completos, lo que significa que pueden ser muy difíciles de resolver a medida que aumenta el número de variables. Algunos métodos comunes para resolver estos problemas incluyen:

- **Método de ramificación y acotamiento (*Branch and Bound*):** Se utiliza para explorar el espacio de soluciones binarias de manera sistemática.
- **Algoritmos Genéticos y Métodos Heurísticos:** Debido a la complejidad computacional de los PLB, los enfoques heurísticos y/o algoritmos genéticos son utilizados a menudo para encontrar soluciones aproximadas en un tiempo razonable [Papalambros00].

2.3.3.3 Aplicaciones Prácticas

Los problemas de PLB se aplican en la selección de proyectos, donde las decisiones deben ser de tipo sí/no. También se utilizan en problemas de asignación de recursos y en la optimización de redes donde, por ejemplo, se decide si conectar o no un nodo en una red de transporte o comunicación.

CAPÍTULO 3: PROPUESTA DE SOLUCIÓN

3.1 Introducción a la Solución Propuesta

En el contexto académico, la PL juega un papel crucial en diversas disciplinas, como la ingeniería, la economía y la investigación de operaciones. La enseñanza de esta materia requiere que los estudiantes no solo comprendan los fundamentos teóricos, sino que también adquieran habilidades prácticas para resolver problemas complejos. Sin embargo, uno de los mayores desafíos para los docentes es generar una variedad suficiente de problemas de PL que permitan a los estudiantes ejercitar sus capacidades en diferentes escenarios.

La generación manual de problemas puede resultar un proceso tedioso, repetitivo y limitado en cuanto a la diversidad de situaciones planteadas. Además, para garantizar que los estudiantes logren una comprensión sólida, es necesario exponerlos a diferentes niveles de dificultad, lo que demanda una gran cantidad de tiempo por parte del cuerpo docente. Por otro lado, los estudiantes, al practicar con los mismos problemas repetidamente, pueden desarrollar patrones de resolución que no representan un aprendizaje genuino ni los preparan para escenarios reales en los que se aplican estas técnicas.

Frente a estos desafíos, surge la necesidad de una herramienta que automatice el proceso de generación de problemas de programación lineal. Una solución de este tipo no solo alivia la carga de trabajo de los docentes, sino que también garantiza que los estudiantes tengan acceso a un conjunto más amplio de problemas, adecuados a diferentes niveles de habilidad y ajustados a una variedad de aplicaciones reales. Una herramienta automática de generación de problemas de PL permite la creación de ejercicios personalizados y adaptativos, que se ajustan al progreso y las necesidades de cada estudiante. Asimismo, la diversificación en la formulación de los problemas estimula el pensamiento crítico y la capacidad de adaptación, habilidades clave para quienes se enfrentan a la programación lineal en un contexto práctico.

El objetivo principal de la solución propuesta es desarrollar un generador de problemas de programación lineal que facilite tanto la enseñanza como el aprendizaje de esta disciplina. Esta herramienta permitirá la creación automática de problemas con distintos tipos de variables (binarias, enteras, continuas) y funciones objetivo, lo que genera un amplio espectro de ejercicios. Entre los beneficios más destacados de la solución propuesta se incluyen:

- **Diversificación del aprendizaje:** La herramienta generará problemas de PL únicos cada vez que se utiliza, lo que permite a los estudiantes enfrentarse a nuevos desafíos y evita la memorización mecánica.
- **Automatización:** Al automatizar el proceso de generación de problemas, se optimizará el tiempo dedicado por los docentes a la creación de ejercicios, permitiendo que se enfoquen en otras áreas pedagógicas.

- **Adaptabilidad:** La herramienta permitirá ajustar la complejidad de los problemas según el nivel de los estudiantes, adaptándose a cursos introductorios o avanzados de programación lineal.
- **Integración con *solvers* externos:** La posibilidad de exportar los problemas generados a formatos estándar (como MPS, lp_solve, LINDO) permitirá su resolución con *solvers* conocidos, ampliando así su aplicabilidad en entornos académicos y profesionales.

3.2 Metodología y Tecnologías Empleadas

En esta sección se expondrá en profundidad las decisiones tecnológicas adoptadas y la estructura algorítmica detrás de la herramienta, destacando cómo estas elecciones permitirán alcanzar soluciones óptimas de manera eficiente y escalable.

3.2.1 Elección de Python y sus Bibliotecas

Para el desarrollo del proyecto, se utilizará el lenguaje de programación Python con su versión 3.11.1. La elección de Python se fundamenta en una serie de características clave que lo convierten en la mejor opción para este proyecto. Python no solo es un lenguaje interpretado y de propósito general, sino que también ofrece una amplia gama de bibliotecas y *frameworks* que cubren las necesidades del proyecto en términos de manipulación matemática, construcción de interfaces gráficas y optimización [Python24]. Además, posee una curva de aprendizaje suave en comparación a otros lenguajes de programación. A continuación, se detallan las razones de la elección de cada una de las tecnologías y bibliotecas implementadas:

- **Tkinter (interfaz gráfica de usuario):** Python incluye nativamente la biblioteca Tkinter, una herramienta ligera y flexible que facilita la creación de interfaces gráficas [Tkinter24]. La elección de Tkinter no solo está motivada por su simplicidad, sino también por su capacidad para integrarse perfectamente con el flujo de trabajo de Python sin necesidad de instalar dependencias adicionales. En el presente proyecto será utilizada la versión 8.6 de esta librería.
- **NumPy (manipulación de datos y cálculos algebraicos):** NumPy es fundamental para el manejo eficiente de las matrices y los vectores involucrados en los problemas de optimización. En particular, se utiliza para la creación de matrices de coeficientes aleatorios de las restricciones y de la función objetivo, que son esenciales en la formulación de problemas de PL [Numpy22]. La eficiencia de NumPy en el manejo de grandes volúmenes de datos y su capacidad para realizar operaciones algebraicas vectorizadas asegura que el proceso de generación del modelo matemático sea rápido y eficiente, incluso cuando se trata de problemas de gran escala. La elección de NumPy también garantiza que la herramienta pueda escalar para manejar

problemas más complejos sin sacrificar rendimiento. En el presente proyecto será utilizada la versión 1.24.1 de esta librería.

- **OR-Tools (optimización):** Biblioteca desarrollada por Google que es capaz de manejar problemas de PL, PLE y otras variantes. OR-Tools destaca por su capacidad para conectarse con *solvers* avanzados de optimización, permitiendo resolver problemas de gran escala con tiempos de respuesta óptimos. Además, OR-Tools ofrece una interfaz flexible que se integra fácilmente con los modelos matemáticos generados, haciendo que la conexión entre el algoritmo de generación y el proceso de resolución sea directa y eficiente.

3.2.2 Descripción del Proceso Algorítmico

3.2.2.1 Interfaz de Usuario

La interfaz gráfica de usuario (GUI), desarrollada con **Tkinter**, permite al usuario interactuar con el sistema de manera sencilla, facilitando la personalización y generación de problemas de PL. Desde la interfaz, el usuario puede definir diversos parámetros que afectan directamente el modelo generado:

- **Número de variables de decisión:** El usuario seleccionará cuántas variables intervendrán en el problema. Este valor es fundamental ya que define la dimensionalidad del espacio de búsqueda en la optimización. Cuantas más variables tenga el problema, mayor será la complejidad de encontrar la solución óptima.
- **Número de restricciones:** El usuario escoge cuántas restricciones deben aplicarse al problema. El número y la naturaleza de las restricciones determinarán el espacio factible dentro del cual el *solver* debe encontrar la solución óptima. Además, la interfaz permitirá ajustar la relación entre restricciones y variables, lo que es crucial en problemas de optimización realistas.
- **Tipo de optimización:** El usuario elegirá si el problema es de **maximización** o **minimización**. Este parámetro es clave, ya que define la dirección en la que se optimiza la función objetivo.
- **Tipo de variables de decisión:** El sistema permitirá que las variables sean **continuas**, **enteras** o **binarias**, lo que da lugar a diferentes tipos de problemas (PL, PLE, PLB y PLM). Este aspecto es importante, ya que el tipo de variable afecta directamente la dificultad computacional del problema y el tipo de algoritmos que OR-Tools debe emplear para resolverlo.
- **Coefficientes de la función objetivo y restricciones:** La interfaz permitirá que el usuario defina los coeficientes de manera manual o deje que la herramienta los genere aleatoriamente. En casos de enseñanza o demostración, los coeficientes aleatorios son útiles para generar una gran variedad de problemas rápidamente,

mientras que los coeficientes manuales permiten un control más preciso sobre los problemas generados.

3.2.2.2 Validación de Entradas

Una parte crítica del flujo algorítmico es la **validación de las entradas** proporcionadas por el usuario. Este paso asegurará que los parámetros ingresados sean coherentes y correctos antes de proceder a la generación del modelo matemático. El sistema va a verificar lo siguiente:

- **Coherencia de los valores:** Se verificará que el número de variables y restricciones sea positivo, asegurando que el problema sea matemáticamente viable. También se validará que los coeficientes de las restricciones y de la función objetivo estén correctamente formateados, evitando errores que puedan comprometer la solución del problema.
- **Naturaleza de las variables:** Se va a comprobar que las variables seleccionadas por el usuario coincidan con el tipo de problema que desean resolver.

3.2.2.3 Generación del Modelo

Este proceso implicará la creación automática de la función objetivo y las restricciones del problema, utilizando los parámetros proporcionados por el usuario. Aquí es donde entra en juego **NumPy**, que se utilizará para manejar los coeficientes y operaciones algebraicas involucradas en la construcción del modelo. El modelo generado va a seguir el estándar de los problemas de optimización lineal y se estructurará de la siguiente manera:

- **Función objetivo:** Esta se construye como una combinación lineal de las variables de decisión, con los coeficientes que definen el impacto de cada variable en el resultado final. Dependiendo de la configuración del usuario, la función objetivo puede ser maximizada o minimizada.
- **Restricciones:** Se definen como ecuaciones o desigualdades lineales que limitan el espacio de búsqueda factible. Estas se construyen con base en los coeficientes proporcionados o generados aleatoriamente, y pueden ser de la forma $\leq$, $\geq$ o $=$.

3.2.2.4 Conexión con solvers externos

Una de las características más potentes de esta herramienta será su capacidad de conectarse con *solvers* externos, específicamente con los *solvers* de **OR-Tools**. Una vez que el modelo ha sido generado, el sistema pasa los datos al *solver*, que se encarga de encontrar

la solución óptima. OR-Tools ofrece una serie de algoritmos avanzados para la resolución de problemas de optimización, dependiendo de la naturaleza del problema.

Además, OR-Tools manejará situaciones en las que los problemas son **infactibles** (no existe solución que satisfaga todas las restricciones) o **no acotados** (la solución puede crecer indefinidamente), notificando al usuario de estas condiciones.

3.2.2.5 Visualización de resultados

Una vez resuelto el problema, los resultados serán presentados al usuario a través de la interfaz gráfica. El sistema va a mostrar:

- **Los valores óptimos de las variables de decisión:** Cada variable recibe un valor óptimo, ya sea continuo, entero o binario, dependiendo de la naturaleza del problema.
- **El valor de la función objetivo:** Este valor representa el óptimo alcanzado, ya sea un máximo o un mínimo, según lo solicitado por el usuario.

La herramienta permitirá que el usuario exporte el problema en formatos estándar como MPS, lp_solve y LINDO, lo que asegura que los problemas generados puedan ser utilizados en otros sistemas o *solvers*. Este proceso asegura que los usuarios no solo puedan visualizar los resultados de manera clara, sino que también puedan guardar y reutilizar los problemas generados para futuras referencias o análisis.

3.3 Algoritmo Propuesto

El algoritmo propuesto para la herramienta de generación automática y resolución de problemas de PL se diseñó con el objetivo de ofrecer una solución flexible, eficiente y accesible para usuarios que deseen generar problemas personalizados y obtener resultados óptimos mediante el uso de técnicas avanzadas de optimización. El flujo del algoritmo sigue un esquema bien definido, compuesto por diversas fases que van desde la captura de las entradas del usuario hasta la visualización y exportación de los resultados.

3.3.1 Explicación Detallada del Algoritmo Desarrollado

En concreto, el algoritmo diseñado se divide en cinco fases fundamentales:

1. Captura de Entradas.
2. Validación.
3. Generación del modelo matemático.
4. Resolución mediante *solver*.
5. Visualización/Exportación de resultados.

3.3.1.1 Fase 1: Entrada de Parámetros

La fase inicial del algoritmo consiste en capturar los parámetros proporcionados por el usuario a través de la interfaz gráfica de usuario (GUI), desarrollada a través de la herramienta Tkinter. Esta interfaz facilita al usuario personalizar el problema de PL que se desea generar, configurando diversos aspectos:

- **Número de variables de decisión:** Este parámetro, proporcionado por el usuario, define cuántas variables intervendrán en la función objetivo y en las restricciones. El número de variables es crucial, ya que influye directamente en la dimensionalidad del problema. Un mayor número de variables implica un espacio de búsqueda más amplio para el *solver*, lo que aumenta la complejidad del problema, pero al mismo tiempo permite modelar problemas más realistas.
- **Número de restricciones:** Las restricciones son las desigualdades y/o ecuaciones lineales que limitan el espacio de búsqueda en el cual el *solver* debe encontrar una solución óptima. El número de restricciones es otro factor de vital importancia en la complejidad del problema, ya que un conjunto de restricciones más amplio puede reducir el espacio de búsqueda factible, aumenta el tiempo de resolución del problema de PL.
- **Tipo de Optimización:** El usuario selecciona si desea maximizar o minimizar la función objetivo. Este es un parámetro clave, ya que define el tipo de problema de PL. Los problemas de maximización buscan encontrar el valor más alto posible de la función objetivo dentro del espacio de búsqueda definido por las restricciones, mientras que los problemas de PL de minimización buscan el valor más bajo.
- **Naturaleza de las variables:** Las variables de decisión pueden ser de naturaleza continua, entera o binaria. Las variables continuas permiten valores reales, mientras que las variables enteras están restringidas a valores discretos. Por otro lado, las variables binarias solo pueden tomar los valores 0 o 1. La naturaleza de las variables define el tipo de problema que se generará, influyendo en la técnica de resolución que utilizará el *solver*.
- **Coeficientes de la función objetivo y restricciones:** Los coeficientes pueden ser generados de forma aleatoria, dentro de un rango establecido por el usuario, a través de la librería NumPy de Python, lo que permite la creación de problemas únicos cada vez que se utiliza la herramienta.

Desafíos: Esta fase implica un balance entre flexibilidad y facilidad de uso. Los parámetros deben ser fáciles de ingresar para usuarios sin experiencia técnica avanzada, pero al mismo tiempo el programa debe ofrecer opciones suficientes para personalizar completamente el problema.

3.3.1.2 Fase 2: Validación de Entradas

Una vez que el usuario haya ingresado los parámetros solicitados por la GUI para la creación del problema de PL, el sistema procede a la validación de los datos para asegurarse que éstos sean correctos y que estén dentro de los límites adecuados. Esta fase es crítica, ya que permite garantizar que el problema generado sea matemáticamente viable y que no se presenten errores en las siguientes etapas:

- **Validación de tipo de datos:** En primer lugar, el programa se asegura de que los valores proporcionados por el usuario sean del tipo correcto. El número de variables y restricciones debe ser un valor entero positivo. Los coeficientes de la función objetivo y las restricciones deben ser numéricos (enteros o flotantes).
- **Validación de naturaleza de las variables:** Dependiendo de si el usuario ha seleccionado variables continuas, enteras o binarias, el sistema comprueba que la configuración sea coherente con los tipos de variables permitidos en el problema de PL.

Desafíos: Los sistemas de validación deben ser lo suficientemente robustos como para detectar errores comunes, pero también deben ser eficientes para no retrasar la ejecución del programa. El uso de mensajes de error claros y fáciles de entender es clave para mejorar la experiencia del usuario.

3.3.1.3 Fase 3: Generación del Modelo Matemático

La generación del modelo matemático es el corazón del algoritmo, y es en este punto en el que se construye el problema de PL basado en los parámetros ingresados. El modelo matemático consiste en:

- **Construcción de la función objetivo:** La función objetivo es una combinación lineal de las variables de decisión, donde los coeficientes indican el peso de cada variable en el valor total de la función. Si el usuario seleccionó maximización, el objetivo es encontrar el valor máximo de esta función. Por otro lado, si se seleccionó minimización, el objetivo es encontrar el valor mínimo.
- **Generación de las restricciones:** Las restricciones son desigualdades o ecuaciones lineales que limitan el conjunto de soluciones factibles. Las restricciones pueden ser del tipo $\leq$, $\geq$ o $=$, lo que permite modelar problemas complejos y realistas.
- **Definición de la naturaleza de las variables:** Dependiendo de la selección del usuario, el sistema establece si las variables son continuas, enteras o binarias. Este paso es importante, ya que afecta directamente el tipo de problema que será generado y al método de resolución del problema por parte del *solver* de **OR-Tools**. El tipo de problema generado se dará según las siguientes combinaciones:
 - **Variables Enteras + Variables Continuas + Variables Binarias:** PLM

- **Variables Enteras + Variables Continuas:** PLM
- **Variables Enteras + Variables Binarias:** PLM
- **Variables Continuas + Variables Binarias:** PLM
- **Variables Enteras:** PLE
- **Variables Continuas:** PL
- **Variables Binarias:** PLB

Este modelo matemático es representado internamente utilizando la librería **NumPy** para gestionar las operaciones algebraicas, asegurando que el sistema maneje eficientemente grandes cantidades de datos cuando sea necesario.

Desafíos: En esta etapa, el reto es manejar eficientemente la creación de matrices de coeficientes para la función objetivo y las restricciones, especialmente en problemas con muchas variables y restricciones. Aquí, la eficiencia en el manejo de matrices y operaciones algebraicas es crucial para no afectar el rendimiento general del sistema, lo cual es bastante importante considerando que se está utilizando un lenguaje interpretado como Python y no compilado.

3.3.1.4 Fase 4: Conexión con el Solver

Una vez generado el modelo matemático, el siguiente paso es conectarlo con un *solver* de optimización. Para este proyecto, se ha seleccionado la herramienta **OR-Tools** debido a su capacidad para manejar problemas de optimización de gran escala y su flexibilidad para trabajar con diferentes tipos de problemas (continuos, enteros, binarios). OR-Tools se encarga de resolver el problema utilizando diferentes algoritmos, dependiendo de la naturaleza de éste. Principalmente, en este proyecto se utilizan dos *solvers* principales según la naturaleza del problema: **GLOP**, un *solver* especializado en problemas de PL simples, y **SCIP**, un *solver* potente que maneja problemas de PLM, PLE y PLB. A continuación, se explicará en detalle cómo y cuándo se emplea cada uno de ellos.

Uso de GLOP para problemas de PL: Para los problemas de PL simples, donde las variables de decisión son **continuas** y no existen restricciones sobre el dominio de las variables (por ejemplo, variables enteras o binarias), el *solver* **GLOP** es el más adecuado. GLOP es un *solver* de optimización lineal, incluido en **OR-Tools**, que utiliza el método de punto interior y es altamente eficiente para resolver problemas lineales grandes y medianos. Este método es especialmente útil para encontrar soluciones en un tiempo relativamente corto cuando se trabaja en espacios continuos [GGLOP24]. El flujo de resolución con GLOP es el siguiente:

1. **Generación del modelo:** A partir de los parámetros proporcionados por el usuario, el modelo matemático lineal es construido utilizando la función objetivo y restricciones lineales con variables continuas.
2. **Envío al *solver* GLOP:** El modelo matemático generado se pasa a GLOP, el cual utiliza su método de punto interior optimizado para buscar soluciones factibles y óptimas

dentro del espacio de soluciones continuas. Este método permite encontrar soluciones rápidas y eficientes en problemas de gran tamaño donde las variables no están restringidas a valores discretos.

3. **Entrega del resultado:** GLOP retorna el valor óptimo de la función objetivo y los valores de las variables de decisión (en caso de que ésta tenga solución). Dado que todas las variables son continuas, el resultado es exacto dentro de la precisión definida por el *solver*.

El uso de GLOP es idóneo para problemas de PL debido a su capacidad para manejar problemas grandes de manera eficiente sin la complejidad adicional que implica la presencia de variables enteras o binarias.

Uso de SCIP para problemas de PLM, PLE y PLB: Cuando se trata de problemas más complejos como PLM, PLE y PLB, el *solver* **SCIP** se utiliza debido a su capacidad para manejar la mezcla de variables continuas y discretas (enteras o binarias). SCIP es un potente *solver* de código abierto diseñado específicamente para resolver problemas de optimización combinatoria, integrando métodos avanzados como **branch and bound**, **cortes de plano** y **heurísticas de búsqueda** [GSCIP24]. SCIP es adecuado para problemas que involucran restricciones más complejas sobre las variables, tales como la necesidad de que algunas variables tomen valores enteros o sean binarias (0 o 1). Este tipo de problemas tiende a ser más desafiante debido a la necesidad de explorar un espacio de soluciones discretas, lo que aumenta la complejidad de la resolución. El flujo de resolución con SCIP es el siguiente:

1. **Generación del modelo:** En problemas de PLM, PLE o PLB, el modelo matemático incluye variables de decisión mixtas, enteras y continuas. A partir de los parámetros ingresados por el usuario, el sistema construye la función objetivo y las restricciones, definiendo las variables como continuas, enteras o binarias según sea el caso.
2. **Envío al Solver SCIP:** El modelo se pasa al *solver* SCIP, que aplica técnicas avanzadas como:
 1. **Branch and Bound:** Una técnica que subdivide el problema original en subproblemas más pequeños (ramificación), y acota las posibles soluciones, descartando aquellas que no pueden conducir a una solución mejor. Esta técnica es fundamental para manejar problemas de PLE y PLB donde las variables son discretas.
 2. **Cortes de plano (Cutting Planes):** Una técnica que añade restricciones adicionales al modelo (cortes) para reducir el espacio de búsqueda y acelerar la convergencia hacia la solución óptima.
 3. **Heurísticas de búsqueda:** SCIP emplea heurísticas que permiten encontrar soluciones aproximadas en las primeras etapas del proceso de ramificación y acotamiento, mejorando así la eficiencia del algoritmo.
3. **Entrega del resultado:** SCIP devuelve la solución óptima o subóptima para problemas donde las variables son enteras o binarias. En muchos casos, el *solver*

también puede determinar que el problema es infactible (sin solución) o no acotado (cuando la función objetivo no tiene un límite superior o inferior definido).

El uso de SCIP es crucial en este tipo de problemas, ya que su capacidad para manejar variables discretas y sus técnicas de ramificación y cortes lo convierten en una opción ideal para resolver problemas complejos de optimización combinatoria.

Finalmente, ya sea con GLOP o SCIP, el *solver* devuelve diferentes resultados según el estado del problema:

- **Solución Óptima:** Si se encuentra una solución óptima, se devuelven los valores de las variables de decisión y el valor de la función objetivo.
- **Problema Infactible:** Si el problema no tiene solución, el sistema notifica que es infactible.
- **Problema No Acotado:** Si la solución puede crecer indefinidamente, se notifica que el problema no está acotado.

Desafíos: La correcta integración con OR-Tools requiere asegurarse de que el modelo matemático esté formulado correctamente antes de enviarlo al *solver*. Además, es fundamental manejar los diferentes resultados que puede devolver el *solver*, mostrando información clara al usuario sobre el estado del problema.

3.3.1.5 Fase 5: Visualización y exportación de resultados

La última fase del algoritmo no solo se encarga de mostrar los resultados obtenidos tras la resolución del problema de programación lineal, sino que también permite **exportar el modelo generado** a formatos compatibles con otros *solvers* de optimización. Esto amplía significativamente la aplicabilidad de la herramienta, ya que permite integrar los problemas generados en diferentes flujos de trabajo y en otras plataformas que utilicen *solvers* externos. La exportación incluye la conversión del problema a formatos como **MPS**, **lp_solve**, y **LINDO**, cada uno de los cuales es ampliamente utilizado en la industria y en entornos académicos.

Formatos de Exportación Soportados: El sistema permite exportar los problemas generados en varios formatos estándar que son reconocidos por *solvers* de optimización profesionales y académicos. A continuación, se describen los principales formatos soportados:

- **MPS (Mathematical Programming System):** Este es uno de los formatos más antiguos y ampliamente utilizados para representar problemas de optimización, en particular de programación lineal y entera. Desarrollado inicialmente en la década de 1960, el formato MPS sigue siendo compatible con muchos *solvers* modernos, como **CPLEX**, **Gurobi** y **GLPK**. El formato MPS es de texto plano, lo que facilita su

manipulación y edición. Para exportar el problema en este formato, el sistema convierte la función objetivo, las restricciones y las variables en el formato estructurado que define MPS, lo que incluye las secciones NAME, ROWS, COLUMNS, RHS y BOUNDS.

- **lp_solve:** **lp_solve** es un *solver* de código abierto que utiliza un formato propio para la entrada de problemas de programación lineal y entera. El formato de entrada de **lp_solve** es más sencillo y cercano a la notación matemática que otros formatos, lo que lo hace fácil de leer y modificar. La exportación del modelo al formato **lp_solve** implica convertir la función objetivo y las restricciones en un archivo de texto plano con sintaxis específica de **lp_solve**, donde las restricciones se describen de forma lineal, y la naturaleza de las variables se especifica al final del archivo.
- **LINDO:** **LINDO** es otro *solver* popular que utiliza un formato propio para la entrada de problemas de programación lineal y entera. La exportación al formato **LINDO** incluye la definición de la función objetivo (max o min), las restricciones (st) y la naturaleza de las variables (int para variables enteras o binarias y free para continuas).

Estos formatos son elegidos porque son ampliamente aceptados y utilizados en aplicaciones industriales y académicas. La flexibilidad para exportar el problema en cualquiera de estos formatos permite que los usuarios lo resuelvan utilizando otros *solvers*, o incluso en plataformas en las que el *solver* puede tener un mejor rendimiento o capacidades adicionales, como el manejo de problemas de gran escala o problemas no lineales.

Proceso de Exportación: El proceso de exportación del modelo generado es crucial para garantizar que los problemas puedan ser resueltos en *solvers* externos sin pérdida de información o errores en la conversión. A continuación, se detalla el flujo de este proceso:

1. **Conversión de la función objetivo:** El primer paso para exportar el problema es convertir la función objetivo en el formato adecuado para el *solver* de destino. Esto implica traducir la función lineal en la sintaxis correspondiente (por ejemplo, max o min en **lp_solve**, o la sección COLUMNS en MPS). Se debe tener especial cuidado en manejar los coeficientes de las variables correctamente, asegurando que mantengan su valor y ubicación.
 - En **MPS**, la función objetivo se representa como parte de la sección COLUMNS, donde se define el coeficiente de cada variable en la función objetivo.
 - En **lp_solve**, la función objetivo se define de manera explícita al inicio del archivo con la sintaxis min: o max:, seguida de la expresión lineal.
 - En **LINDO**, la función objetivo se define en la primera línea con la palabra clave max o min seguida de la expresión matemática.

2. **Conversión de las restricciones:** Las restricciones del problema también deben ser convertidas al formato adecuado. Cada restricción se traduce a una línea o bloque específico según el formato. Por ejemplo:
 - En **MPS**, las restricciones se definen en la sección ROWS, donde se indica el tipo de restricción ($\leq$, $\geq$, $=$), y en la sección COLUMNS se asocian los coeficientes de las variables con las restricciones.
 - En **lp_solve**, las restricciones se añaden en líneas separadas debajo de la función objetivo, utilizando una sintaxis lineal y separadas por punto y coma. Por ejemplo: $2x_1 + 4x_2 \leq 10$;
 - En **LINDO**, las restricciones se definen después de la palabra clave st, con cada restricción en una línea separada.
3. **Naturaleza de las variables:** El tercer paso es asegurar que la naturaleza de las variables (continuas, enteras, binarias) se traduzca correctamente en el formato de destino. Esta información es crucial, ya que determina cómo el *solver* manejará cada variable durante el proceso de optimización.
 - En **MPS**, las variables binarias y enteras se definen en la sección BOUNDS, donde se especifican los límites de las variables. Las variables continuas por defecto no requieren una especificación adicional.
 - En **lp_solve**, las variables enteras y binarias se definen al final del archivo mediante las palabras clave int (enteras) o bin (binarias), seguidas de la lista de variables.
 - En **LINDO**, las variables binarias y enteras se definen con las palabras clave int (enteras) o bin (binarias) en las últimas líneas del archivo.
4. **Exportación del archivo:** Una vez que todos los componentes (función objetivo, restricciones, variables) han sido convertidos, el sistema genera el archivo de salida en el formato seleccionado. El archivo resultante puede ser guardado local o directamente utilizado en un *solver* externo.
 - En el caso de **MPS**, el archivo exportado sigue una estructura jerárquica que incluye las secciones NAME, ROWS, COLUMNS, RHS, y BOUNDS.
 - En **lp_solve**, el archivo es un script que contiene toda la definición del modelo y puede ser ejecutado directamente en el entorno de **lp_solve**. Esto permite una integración directa y fácil con este *solver*.
 - En **LINDO**, el archivo contiene toda la información del modelo en un formato compacto y legible por el *solver* de LINDO.

Ventajas de la Exportación: La exportación de problemas a otros *solvers* proporciona múltiples beneficios:

- **Interoperabilidad:** Permite que los problemas generados en la herramienta sean resueltos en diferentes entornos, proporcionando flexibilidad en el uso de diferentes *solvers* que podrían tener ventajas particulares en rendimiento, escalabilidad o soporte para tipos específicos de problemas.
- **Uso en escenarios industriales:** Muchos entornos industriales prefieren usar *solvers* específicos debido a su optimización para tipos concretos de problemas. Al ofrecer la exportación a formatos compatibles con estos *solvers*, la herramienta puede ser utilizada en proyectos reales que requieren capacidades avanzadas o grandes volúmenes de datos.
- **Facilidad de análisis y edición:** Exportar en formatos estándar como MPS permite que los modelos generados sean analizados y editados manualmente por usuarios expertos, lo que facilita la validación y el refinamiento de los problemas.
- **Escalabilidad:** Algunos *solvers* externos están mejor optimizados para manejar problemas de gran escala o pueden ofrecer técnicas más avanzadas que las que ofrece el *solver* interno de OR-Tools. La exportación permite a los usuarios aprovechar estos *solvers* para mejorar el rendimiento en problemas complejos.

Desafíos Técnicos en la Exportación: El proceso de exportación no está exento de desafíos. Algunos de los problemas más comunes que deben abordarse incluyen:

- **Conversión precisa de las restricciones:** En algunos casos, las desigualdades o ecuaciones del problema original pueden no ser fácilmente convertibles a ciertos formatos debido a diferencias en la representación de los operadores o en los límites de las variables. Para solucionar este desafío, el sistema debe asegurarse de que los operadores sean correctamente traducidos y que no haya pérdida de información.
- **Compatibilidad con *solvers* específicos:** Algunos *solvers* tienen limitaciones en cuanto al número de restricciones o la naturaleza de las variables que pueden manejar. En estos casos, la exportación debe adaptarse para cumplir con estas restricciones, o proporcionar un mensaje de advertencia al usuario sobre posibles limitaciones.

3.3.2 Pseudocódigo del Programa

3.3.2.1 Pseudocódigo de main.py

En este archivo el algoritmo representa el flujo principal de la aplicación para generar y resolver problemas de PL, PLB, PLE y PLM, y exportar los resultados en diferentes formatos. La aplicación inicia una interfaz gráfica donde el usuario puede especificar el tipo de problema y los parámetros. Luego, dependiendo de la combinación de la naturaleza de las variables, se llama a la función de generación del tipo de problema correspondiente. Finalmente, los resultados se muestran al usuario y, si se desea, se exportan en formato

MPS, LINDO o lp_solve. En caso de exportar el problema, se genera un problema nuevo que contiene la representación de éste en el formato elegido. A continuación, se explicará el detalle del pseudocódigo del presente archivo.

Descripción General de la Clase ProblemGeneratorApp: Es la encargada de gestionar la interfaz gráfica de usuario de la aplicación para generar problemas de PL. Utiliza la librería Tkinter de Python para construir la ventana principal y los distintos componentes interactivos que permiten al usuario definir los parámetros del problema a generar.

Método __init__: Es el constructor de la clase y se ejecuta al instanciar un objeto de la clase **ProblemGeneratorApp**. Su función es inicializar la ventana principal de la aplicación y configurar los elementos visuales y variables que permiten la interacción del usuario.

Algorithm 1 __init__()

Crear ventana principal con título "Generador de Problemas de Programación Lineal"

Establecer tamaño de ventana

Crear un marco para contener todos los widgets y colocarlo en la ventana

Inicializar variables para:

- Tipo de optimización
- Cantidad de variables enteras, continuas y binarias
- Coeficientes de la función objetivo y restricciones
- Cantidad de restricciones ($\geq$, $\leq$, $=$)
- Coeficientes enteros y formato de exportación

Crear grupo de configuración general con etiquetas y entradas de texto para:

- Tipo de optimización
- Cantidad de variables enteras, continuas y binarias

Crear grupo de configuración de coeficientes con entradas para:

- Rango de coeficientes de la función objetivo y restricciones
- Rango de coeficientes del lado derecho de las restricciones

Crear grupo de configuración de restricciones con entradas para:

- Cantidad de restricciones ($\geq$, $\leq$, $=$)

Añadir botones para generar, exportar y limpiar problemas

Añadir un área de texto para mostrar el problema generado

Configurar la disposición de los elementos en el marco

Figura 10: Pseudocódigo de método *init*. Fuente: Elaboración propia.

Método clear_fields: Se encarga de restablecer los valores de las entradas de texto y los elementos visuales a su estado inicial. Esto permite que el usuario pueda ingresar nuevos parámetros sin que los valores anteriores interfieran.

Algorithm 2 clear_fields()

Limpiar todos los campos de entrada y área de texto

Restablecer los valores de las variables a sus valores predeterminados

Figura 11: Pseudocódigo de método *clear_fields*. Fuente: Elaboración propia.

Método `generate_problem`: Es el núcleo de la generación de problemas de PL. Recoge los valores ingresados por el usuario, realiza una validación básica de los mismos y, en función de estos datos, genera el problema correspondiente.

Algorithm 3: `generate_problem()`

Paso 1: Obtener los valores ingresados por el usuario
`tipo_optimizacion` ← valor de `self.optimization_type`
`num_enteras` ← valor de `self.num_enteras`
`num_continuas` ← valor de `self.num_continuas`
`num_binarias` ← valor de `self.num_binarias`
`obj_coef_min`, `obj_coef_max` ← valores de `self.obj_coef_min`, `self.obj_coef_max`
`restric_coef_min`, `restric_coef_max` ← valores de `self.restric_coef_min`, `self.restric_coef_max`
`right_coef_min`, `right_coef_max` ← valores de `self.right_coef_min`, `self.right_coef_max`
`num_less_equal` ← valor de `self.num_less_equal`
`num_greater_equal` ← valor de `self.num_greater_equal`
`num_equal` ← valor de `self.num_equal`

Paso 2: Validación de los datos
 if algún campo está vacío then
 Mostrar mensaje de error al usuario: “Complete todos los campos requeridos”
 return

Paso 3: Determinar el tipo de problema
 if `tipo_optimizacion` es “Maximizar” then
 `objetivo` ← Maximización
 else
 `objetivo` ← Minimización

Paso 4: Llamar al generador de problemas correspondiente
 if `num_binarias` > 0 and `num_enteras` > 0 and `num_continuas` > 0 then
 Llamar a `plm_generator` para generar un problema de PLM y resolverlo
 else if `num_binarias` > 0 and `num_enteras` > 0 and `num_continuas` = 0 then
 Llamar a `plm_generator` para generar un problema de PLM y resolverlo
 else if `num_binarias` > 0 and `num_enteras` = 0 and `num_continuas` > 0 then
 Llamar a `plm_generator` para generar un problema de PLM y resolverlo
 else if `num_binarias` = 0 and `num_enteras` > 0 and `num_continuas` > 0 then
 Llamar a `plm_generator` para generar un problema de PLM y resolverlo
 else if `num_binarias` = 0 and `num_enteras` > 0 and `num_continuas` = 0 then
 Llamar a `ple_generator` para generar un problema de PLE y resolverlo
 else if `num_binarias` = 0 and `num_enteras` = 0 and `num_continuas` > 0 then
 Llamar a `pl_generator` para generar un problema de PL simple y resolverlo
 else if `num_binarias` > 0 and `num_enteras` = 0 and `num_continuas` = 0 then
 Llamar a `plb_generator` para generar un problema de PLB y resolverlo
 else
 Mostrar mensaje de error: “Debe ingresar al menos una variable”

Paso 5: Visualizar el problema generado
 Limpiar el área de texto de `self.problem_text`
 Mostrar la representación textual de `self.problem` en el área de texto

Paso 6: Mostrar problema generado al usuario
 Mostrar problema generado y su solución.

Figura 12: Pseudocódigo de método `generate_problem`. Fuente: Elaboración propia.

Método `convert_to_format`: Toma el problema generado y lo transforma a un formato específico de acuerdo con la elección del usuario (lp_solve, MPS o LINDO).

Algorithm 5 `convert_to_format(problem, formato)`

Paso 1: Identificar el formato de exportación
 if `formato` es “lp_solve” then
 `resultado` ← `convert_to_lp_solve(problem)`
 else if `formato` es “mps” then
 `resultado` ← `convert_to_mps(problem)`
 else if `formato` es “lindo” then
 `resultado` ← `convert_to_lindo(problem)`
 else
 Mostrar mensaje de error: “Formato no compatible”
 return vacío
 return `resultado`

Figura 13: Pseudocódigo de método `convert_to_format`. Fuente: Elaboración propia.

Método `export_problem`: Se encarga de exportar el problema de PL generado a un archivo en el formato deseado por el usuario. Se basa en el método **`convert_to_format`** para transformar el problema al formato específico antes de guardar el archivo.

Algorithm 4: `export_problem()`

Paso 1: Obtener el formato de exportación
`formato` $\leftarrow$ valor de `self.export_format.get()`
if `formato` está vacío **then**
 Mostrar mensaje de error: "Seleccione un formato de exportación."
 return

Paso 2: Generar el problema sin mostrar en la interfaz
Llamar a `self.generate_problem(False)`

Paso 3: Obtener los datos del problema
`funcion_objetivo` $\leftarrow$ `self.funcion_objetivo`
`restricciones` $\leftarrow$ `self.restricciones`
`naturalidad` $\leftarrow$ `self.naturalidad`
if `funcion_objetivo` es `None` **or** `restricciones` es `None` **or** `naturalidad` es `None` **then**
 Lanzar error: "La generación del problema falló. No se pudieron obtener los datos necesarios."
Paso 4: Convertir el problema al formato seleccionado
`content` $\leftarrow$ `convert_to_format(funcion_objetivo, restricciones, naturalidad, formato)`

Paso 5: Determinar la extensión del archivo según el formato
if `formato` es "lp_solve" **then**
 `file_extension` $\leftarrow$ ".lp"
else if `formato` es "MPS" **then**
 `file_extension` $\leftarrow$ ".mps"
else if `formato` es "Lindo" **then**
 `file_extension` $\leftarrow$ ".ltx"
else
 `file_extension` $\leftarrow$ ".txt" {Por defecto, en caso de un formato no esperado}

Paso 6: Mostrar diálogo para guardar el archivo
`file_path` $\leftarrow$ `filedialog.asksaveasfilename(..., defaultextension=file_extension)`
if `file_path` no es `None` **then**
 Abrir archivo en `file_path` para escritura
 Escribir `content` en el archivo
 Cerrar archivo
 Mostrar mensaje de éxito: "Problema exportado como `formato`."
else
 Mostrar mensaje: "Exportación cancelada por el usuario."

Paso 7: Manejo de excepciones
if ocurre una excepción **then**
 Mostrar mensaje de error: "Error al exportar el problema: `e`"

Figura 14: Pseudocódigo de método `export_problem`. Fuente: Elaboración propia.

Método `convert_to_lp_solve`: Formatea el problema, dejándolo compatible con LP Solve.

Algorithm 6: `convert_to_lp_solve(funcion_objetivo, restricciones, naturalidad)`

Paso 1: Formatear la función objetivo
if "max" está en `funcion_objetivo` **then**
 `lp_solve.output` $\leftarrow$ "max: " + `funcion_objetivo` sin "max z =" + ";"
else
 `lp_solve.output` $\leftarrow$ "min: " + `funcion_objetivo` sin "min z =" + ";"

Paso 2: Formatear las restricciones
for cada `restriccion` en `restricciones` **do**
 Reemplazar "<=" por "<=" en `restriccion`
 Reemplazar ">" por ">=" en `restriccion`
 Añadir `restriccion` + ";" a `lp_solve.output`

Paso 3: Formatear la naturaleza de las variables
`int_vars` $\leftarrow$ lista vacía
`bin_vars` $\leftarrow$ lista vacía
for cada `var` en `naturalidad` **do**
 if " $\in \{0, 1\}$ " está en `var` **then**
 Añadir el nombre de la variable (antes de " $\in$ ") a `bin_vars`
 else if " $\in \mathbb{Z}$ " está en `var` **then**
 Añadir el nombre de la variable (antes de " $\in$ ") a `int_vars`

Paso 4: Agregar la naturaleza de las variables al output
if `int_vars` no está vacía **then**
 Añadir "int" + variables de `int_vars` separadas por comas + ";" a `lp_solve.output`
if `bin_vars` no está vacía **then**
 Añadir "bin" + variables de `bin_vars` separadas por comas + ";" a `lp_solve.output`

Paso 5: Devolver el contenido formateado
return `lp_solve.output`

Figura 15: Pseudocódigo de método `convert_to_lp_solve`. Fuente: Elaboración propia.

Método `convert_to mps`: Formatea el problema, dejándolo compatible con MPS.

Algorithm 7: `convert_to_mps(funcion_objetivo, restricciones, naturalidad)`

```

Paso 1: Iniciar la estructura del archivo MPS
lines ← lista vacía
Añadir "NAME PROBLEM" a lines
Paso 2: Formatear la sección ROWS
Añadir "ROWS" a lines
Añadir " N OBJ" a lines {N indica que es la función objetivo}
for cada restricción en restricciones do
  Determinar tipo de restricción:
  if restricción contiene "=" then
    tipo ← "E"
  else if restricción contiene "≥" then
    tipo ← "G"
  else if restricción contiene "≤" then
    tipo ← "L"
  else
    Lanzar error: "Tipo de restricción desconocido"
  Añadir tipo de restricción a lines {L para ≤, G para ≥, E para =}
Paso 3: Formatear la sección COLUMNS
Añadir "COLUMNS" a lines
Extraer coef_objetivo_matches de funcion_objetivo usando regex
for cada coef_obj en coef_objetivo_matches do
  var ← "xj+1"
  Añadir "var OBJ coef_obj" a lines
  for cada restricción en restricciones do
    Extraer coef_restriccion_matches de restricción usando regex
    coef_restriccion ← coef_restriccion_matches[j]
    Añadir "var Ri+1 coef_restriccion" a lines
Paso 4: Formatear la sección RHS (Right-Hand Side)
Añadir "RHS" a lines
for cada restricción en restricciones do
  rhs_value ← valor del lado derecho de restricción usando regex
  Añadir "RHS1 Ri + 1 rhs_value" a lines
Paso 5: Formatear la sección BOUNDS
if naturalidad no está vacía then
  Añadir "BOUNDS" a lines
  for cada var_nature en naturalidad do
    var_name ← "xj+1"
    if var_nature contiene "binaria" then
      Añadir "UP BND var_name 1" a lines
      Añadir "LO BND var_name 0" a lines
    else if var_nature contiene "entera" then
      {Las variables enteras pueden tener límites específicos si es necesario}
Paso 6: Formatear la sección ENDATA
Añadir "ENDATA" a lines
Paso 7: Combinar todas las líneas
mps_content ← combinar lines con saltos de línea
return mps_content

```

Figura 16: Pseudocódigo de método `convert_to_mps`. Fuente: Elaboración propia.

Método `convert_to lindo`: Formatea el problema, dejándolo compatible con LINDO.

Algorithm 8: `convert_to_lindo(funcion_objetivo, restricciones, naturalidad)`

```

Paso 1: Iniciar la estructura de salida de Lindo
lindo_output ← cadena vacía
Paso 2: Formatear la función objetivo
if "max" está en funcion_objetivo then
  lindo_output ← "max " + funcion_objetivo sin "max z =" + salto de línea
else
  lindo_output ← "min " + funcion_objetivo sin "min z =" + salto de línea
Paso 3: Formatear las restricciones
Añadir "st" + salto de línea a lindo_output
for cada restricción en restricciones do
  Reemplazar "<" por "<=" en restricción
  Reemplazar ">" por ">=" en restricción
  Añadir restricción + salto de línea a lindo_output
Paso 4: Finalizar la sección de restricciones
Añadir "end" + salto de línea a lindo_output
Paso 5: Definir la naturaleza de las variables
for cada var en naturalidad do
  var_name ← nombre de la variable antes de "e"
  if "e {0, 1}" está en var then
    Añadir "int var_name" + salto de línea a lindo_output
  else if "e Z" está en var then
    Añadir "gin var_name" + salto de línea a lindo_output {"gin" indica que la variable es entera general en Lindo}
  else
    Añadir "free var_name" + salto de línea a lindo_output {Las variables continuas se asumen como "free" por defecto}
Paso 6: Devolver el contenido formateado
return lindo_output

```

Figura 17: Pseudocódigo de método `convert_to_lindo`. Fuente: Elaboración propia.

3.3.2.2 Pseudocódigo de *pl_generator*

Este archivo genera problemas de PL configurando automáticamente una función objetivo y restricciones en base a parámetros del usuario como tipo de optimización (maximización o minimización), cantidad de variables y restricciones, y rangos de coeficientes. La generación de valores aleatorios permite crear problemas de diversa complejidad y asegura que todas las variables sean no negativas.

Una vez formulado el problema, el archivo permite resolverlo utilizando el *solver* **GLOP** de **OR-Tools**, que es específica para programación lineal. Se analizan los resultados y se determina si existe una solución óptima, si el problema es infactible (sin solución posible) o no acotado (solución indefinida). Este archivo facilita:

- **Simulación y Pruebas:** Genera problemas de PL variados para probar algoritmos y *solvers*.
- **Entornos Educativos:** Permite comprender y practicar con problemas de programación lineal.
- **Investigación y Desarrollo:** Proporciona un entorno para simular escenarios de optimización.

A continuación, se explicará el detalle del pseudocódigo del presente archivo.

Método *generar_PL*: Se encarga de crear un problema de PL aleatorio. Dado un tipo de optimización (maximización o minimización), número de variables y restricciones, y los rangos de valores para los coeficientes de la función objetivo y restricciones.

Algorithm 9: *generar_PL*(tipo.optimizacion, num.variables, num.restricciones, obj.coef.range, restric.coef.range, right.coef.range, num.less.equal, num.greater.equal, num.equal, integer.coeffs)

Paso 1: Generar la función objetivo con coeficientes
coef.objetivo ← generar valores aleatorios en el rango *obj.coef.range* con *num.variables* elementos
 if *integer.coeffs* es verdadero then
 Redondear *coef.objetivo* a números enteros
 else
 Redondear *coef.objetivo* a 2 decimales
 if *tipo.optimizacion* es "max" then
 funcion.objetivo ← "max z =" seguido de la suma de términos *coef.objetivo* y variables
 else
 funcion.objetivo ← "min z =" seguido de la suma de términos *coef.objetivo* y variables

Paso 2: Generar restricciones con coeficientes
restricciones ← lista vacía
tipos.restricciones ← lista de símbolos "<=", ">=", "=" según *num.less.equal*, *num.greater.equal* y *num.equal*
 Mezclar *tipos.restricciones* aleatoriamente
 for cada restricción de *num.restricciones* do
 coef.restriccion ← generar valores aleatorios en *restric.coef.range*
 if *integer.coeffs* es verdadero then
 Redondear *coef.restriccion* a números enteros
 else
 Redondear *coef.restriccion* a 2 decimales
 lado.derecho ← valor aleatorio en *right.coef.range*, redondeado si es necesario
 restriccion ← combinación de coeficientes, variables y el símbolo de *tipos.restricciones*
 Añadir *restriccion* a *restricciones*

Paso 3: Definir la naturaleza de las variables
naturalidad ← lista con " $x_i \geq 0$ " para cada variable
 return *funcion.objetivo*, *restricciones*, *naturalidad*

Figura 18: Pseudocódigo de método *generar_PL*. Fuente: Elaboración propia.

Método `pl_solver`: Utiliza **OR-Tools** para resolver el problema de PL generado.

Algorithm 10: `pl_solver(funcion_objetivo, restricciones, naturalidad)`

```

Paso 1: Crear el solver con OR-Tools
solver ← instancia de solver GLOP
if no se pudo crear solver then
    Mostrar mensaje: "Solver no disponible."
    return None, None
Paso 2: Obtener coeficientes de la función objetivo
coeficientes_obj ← lista de coeficientes extraídos de funcion_objetivo
num_variables ← longitud de coeficientes_obj
Paso 3: Crear variables continuas
variables ← lista de variables continuas en solver con límite inferior 0 e infinito superior
Paso 4: Definir la función objetivo
if funcion_objetivo empieza con "max" then
    Maximizar la suma de coeficientes y variables en solver
else
    Minimizar la suma de coeficientes y variables en solver
Paso 5: Agregar restricciones
for cada restriccion en restricciones do
    Extraer coeficientes, signo y lado_derecho de restriccion
    if signo es "≤" then
        Añadir restricción con ≤ en solver
    else if signo es "≥" then
        Añadir restricción con ≥ en solver
    else if signo es "=" then
        Añadir restricción con = en solver
Paso 6: Resolver el problema
status ← ejecutar solver
Paso 7: Interpretar el resultado
if status es OPTIMAL then
    resultado ← valores de las variables solución
    valor_funcion_objetivo ← valor óptimo de la función objetivo
    return valor_funcion_objetivo, resultado
else if status es INFEASIBLE then
    Mostrar mensaje: "El problema es infactible."
    return "Infactible", "Infactible"
else if status es UNBOUNDED then
    Mostrar mensaje: "El problema es no acotado."
    return "No acotado", "No acotado"
else
    Mostrar mensaje: "No se encontró una solución óptima."
    return "Sin solución", "Sin solución"

```

Figura 19: Pseudocódigo de método `pl_solver`. Fuente: Elaboración propia.

3.3.2.3 Pseudocódigo de `plb_generator`

Este archivo genera problemas de PLB con una función objetivo y un conjunto de restricciones, ambos configurados automáticamente con valores aleatorios en base a parámetros ingresados por el usuario. Se ajustan parámetros como el tipo de optimización (maximización o minimización), el número de variables y restricciones, y los rangos de valores posibles para los coeficientes

Una vez que el problema ha sido formulado, el archivo permite resolverlo utilizando el *solver* **SCIP** de la biblioteca **OR-Tools**, adecuada para problemas enteros y binarios, donde las variables tienen restricciones de tipo binario. El *solver* busca una solución óptima que maximice o minimice la función objetivo, aplicando las restricciones generadas. Tras resolver el problema, se analizan los resultados y se determina si existe una solución óptima, si el problema es infactible (sin solución posible) o no acotado (solución indefinida). Este archivo es particularmente útil para:

- **Simulación de Decisiones Binarias:** En problemas donde las decisiones son binarias, como selección o exclusión, se puede utilizar *plb_generator.py* para generar y resolver problemas de este tipo.
- **Estudio de Problemas Combinatorios:** Los problemas de PLB son comunes en problemas de optimización combinatoria, como en logística, planificación y diseño de redes, donde se necesitan soluciones de sí/no.
- **Prototipado y Pruebas:** Permite crear múltiples instancias de problemas binarios para probar diferentes configuraciones y evaluar el desempeño del *solver* SCIP en distintos escenarios.

Método generar_PLB: Crea un problema de PLB de manera aleatoria. Genera la función objetivo y las restricciones del problema en un formato legible, estableciendo que todas las variables sean binarias (valores de 0 o 1).

Algorithm 11: generar_PLB(tipo_optimizacion, num_variables, num_restricciones, obj_coef_range, restric_coef_range, right_coef_range, num_less_equal, num_greater_equal, num_equal, integer_coeffs)

Paso 1: Generar la función objetivo con coeficientes
 $coef_objetivo \leftarrow$ valores aleatorios en obj_coef_range con $num_variables$ elementos
if $integer_coeffs$ es verdadero **then**
 Redondear $coef_objetivo$ a números enteros
else
 Redondear $coef_objetivo$ a 2 decimales
if $tipo_optimizacion$ es "max" **then**
 $funcion_objetivo \leftarrow$ "max z =" seguido de la suma de términos $coef_objetivo$ y variables
else
 $funcion_objetivo \leftarrow$ "min z =" seguido de la suma de términos $coef_objetivo$ y variables
Paso 2: Generar restricciones con coeficientes
 $restricciones \leftarrow$ lista vacía
 $tipos_restricciones \leftarrow$ lista de símbolos " $\leq$ ", " $\geq$ ", " $=$ " basada en num_less_equal , $num_greater_equal$ y num_equal
Mezclar $tipos_restricciones$ aleatoriamente
for cada restricción de $num_restricciones$ **do**
 $coef_restriccion \leftarrow$ valores aleatorios en $restric_coef_range$ con $num_variables$ elementos
if $integer_coeffs$ es verdadero **then**
 Redondear $coef_restriccion$ a enteros
else
 Redondear $coef_restriccion$ a 2 decimales
 $lado_derecho \leftarrow$ valor aleatorio en $right_coef_range$, redondeado si es necesario
 $restriccion \leftarrow$ combinación de coeficientes, variables y tipo de $tipos_restricciones$
Añadir $restriccion$ a $restricciones$
Paso 3: Definir la naturaleza de las variables
 $naturalidad \leftarrow$ lista con " $x_i \in \{0,1\}$ " para cada variable
return $funcion_objetivo$, $restricciones$, $naturalidad$

Figura 20: Pseudocódigo de método *generar_PLB*. Fuente: Elaboración propia.

Método plb_solver: Toma un problema de PLE y utiliza el *solver* SCIP de **OR-Tools** para resolverlo, obteniendo los valores óptimos de las variables o informando sobre la factibilidad del problema.

Algorithm 12: *plb_solver*(funcion_objetivo, restricciones, naturalidad)

Paso 1: Crear el solver con OR-Tools
solver $\leftarrow$ instancia de solver SCIP
if no se pudo crear *solver* **then**
 Mostrar mensaje: "Solver no disponible."
 return None, None

Paso 2: Obtener coeficientes de la función objetivo
coeficientes_obj $\leftarrow$ lista de coeficientes extraídos de *funcion_objetivo*
num_variables $\leftarrow$ longitud de *coeficientes_obj*

Paso 3: Crear variables binarias
variables $\leftarrow$ lista de variables binarias en *solver* con valores 0 o 1

Paso 4: Definir la función objetivo
if *funcion_objetivo* empieza con "max" **then**
 Maximizar la suma de coeficientes y variables en *solver*
else
 Minimizar la suma de coeficientes y variables en *solver*

Paso 5: Agregar restricciones
for cada *restriccion* **en** *restricciones* **do**
 Extraer *coeficientes*, *signo* y *lado_derecho* de *restriccion*
 if *signo* es " $\leq$ " **then**
 Añadir restricción con $\leq$ en *solver*
 else if *signo* es " $\geq$ " **then**
 Añadir restricción con $\geq$ en *solver*
 else if *signo* es "=" **then**
 Añadir restricción con = en *solver*

Paso 6: Resolver el problema
status $\leftarrow$ ejecutar *solver*

Paso 7: Interpretar el resultado
if *status* es OPTIMAL **then**
 resultado $\leftarrow$ valores de las variables solución
 valor_funcion_objetivo $\leftarrow$ valor óptimo de la función objetivo
 return *valor_funcion_objetivo*, *resultado*
else if *status* es INFEASIBLE **then**
 Mostrar mensaje: "El problema es infactible."
 return "Infactible", "Infactible"
else if *status* es UNBOUNDED **then**
 Mostrar mensaje: "El problema es no acotado."
 return "No acotado", "No acotado"
else
 Mostrar mensaje: "No se encontró una solución óptima."
 return "Sin solución", "Sin solución"

Figura 21: Pseudocódigo de método *plb_solver*. Fuente: Elaboración propia.

3.3.2.4 Pseudocódigo de *ple_generator*

El archivo permite generar problemas de PLE con una función objetivo y un conjunto de restricciones, ambos configurados automáticamente con valores aleatorios en base a parámetros ingresados por el usuario. Se ajustan parámetros como el tipo de optimización (maximización o minimización), el número de variables y restricciones, y los rangos de valores posibles para los coeficientes

Una vez que el problema ha sido formulado, el archivo permite resolverlo utilizando el *solver* **SCIP** de la biblioteca **OR-Tools**, adecuada para problemas enteros y binarios. El *solver* busca una solución óptima que maximice o minimice la función objetivo, aplicando las restricciones generadas. Tras resolver el problema, se analizan los resultados y se determina si existe una solución óptima, si el problema es infactible (sin solución posible) o no acotado (solución indefinida). Este archivo es particularmente útil para:

- **Planificación y Logística:** Permite modelar problemas donde las decisiones deben ser enteras, como cantidades discretas de recursos o ubicaciones.
- **Asignación de Recursos:** Utilizable en problemas de optimización donde se necesita asignar recursos de forma entera (por ejemplo, número de empleados o unidades de producto).
- **Simulación de Escenarios Empresariales:** Facilita la creación de modelos de optimización para analizar opciones en contextos de negocio donde los resultados deben ser valores enteros.
- **Educación y Entrenamiento:** Herramienta didáctica para que estudiantes practiquen con problemas de PLE, mejorando su comprensión de la optimización discreta.

Método generar_PLE: Genera un problema de programación lineal entera a partir de valores y configuraciones aleatorias.

Algorithm 13: generar_PLE(tipo_optimizacion, num_variables, num_restricciones, obj_coef_range, restric_coef_range, right_coef_range, num_less_equal, num_greater_equal, num_equal, integer_coefs)

Paso 1: Generar la función objetivo con coeficientes
coef_objetivo ← valores aleatorios en *obj_coef_range* con *num_variables* elementos
if *integer_coefs* es verdadero **then**
 Redondear *coef_objetivo* a enteros
else
 Redondear *coef_objetivo* a 2 decimales
if *tipo_optimizacion* es "max" **then**
 funcion_objetivo ← "max z =" seguido de la suma de términos *coef_objetivo* y variables
else
 funcion_objetivo ← "min z =" seguido de la suma de términos *coef_objetivo* y variables

Paso 2: Generar restricciones con coeficientes
restricciones ← lista vacía
tipos_restricciones ← lista de símbolos "<=", ">=", "=" basada en *num_less_equal*, *num_greater_equal* y *num_equal*
Mezclar *tipos_restricciones* aleatoriamente
for cada restricción de *num_restricciones* **do**
 coef_restriccion ← valores aleatorios en *restric_coef_range* con *num_variables* elementos
 if *integer_coefs* es verdadero **then**
 Redondear *coef_restriccion* a enteros
 else
 Redondear *coef_restriccion* a 2 decimales
 lado_derecho ← valor aleatorio en *right_coef_range*, redondeado si es necesario
 restriccion ← combinación de coeficientes, variables y tipo de *tipos_restricciones*
 Añadir *restriccion* a *restricciones*

Paso 3: Definir la naturaleza de las variables
naturalidad_enteras ← lista con " $x_i \in \mathbb{Z}$ " para cada variable
return *funcion_objetivo*, *restricciones*, *naturalidad_enteras*

Figura 22: Pseudocódigo de método *generar_PLE*. Fuente: Elaboración propia.

Método ple_solver: Este método toma el problema generado y utiliza **OR-Tools** para resolverlo, obteniendo una solución óptima o determinando si el problema es infactible o no acotado.

Algorithm 14: *ple_solver*(funcion_objetivo, restricciones, naturalidad_enteras)

```

Paso 1: Crear el solver con OR-Tools
solver ← instancia de solver SCIP
if no se pudo crear solver then
    Mostrar mensaje: "Solver no disponible."
    return None, None
Paso 2: Obtener coeficientes de la función objetivo
coeficientes_obj ← lista de coeficientes extraídos de funcion_objetivo
num_variables ← longitud de coeficientes_obj
Paso 3: Crear variables enteras
variables ← lista de variables enteras en solver con valores no negativos
Paso 4: Definir la función objetivo
if funcion_objetivo empieza con "max" then
    Maximizar la suma de coeficientes y variables en solver
else
    Minimizar la suma de coeficientes y variables en solver
Paso 5: Agregar restricciones
for cada restriccion en restricciones do
    Extraer coeficientes, signo y lado_derecho de restriccion
    if signo es " $\leq$ " then
        Añadir restricción con  $\leq$  en solver
    else if signo es " $\geq$ " then
        Añadir restricción con  $\geq$  en solver
    else if signo es "=" then
        Añadir restricción con = en solver
Paso 6: Resolver el problema
status ← ejecutar solver
Paso 7: Interpretar el resultado
if status es OPTIMAL then
    resultado ← valores de las variables solución
    valor_funcion_objetivo ← valor óptimo de la función objetivo
    return valor_funcion_objetivo, resultado
else if status es INFEASIBLE then
    Mostrar mensaje: "El problema es infactible."
    return "Infactible", "Infactible"
else if status es UNBOUNDED then
    Mostrar mensaje: "El problema es no acotado."
    return "No acotado", "No acotado"
else
    Mostrar mensaje: "No se encontró una solución óptima."
    return "Sin solución", "Sin solución"

```

Figura 23: Pseudocódigo de método *ple_solver*. Fuente: Elaboración propia.

3.3.2.5 Pseudocódigo de *plm_generator*

El archivo permite generar problemas de PLM con una función objetivo y un conjunto de restricciones, ambos configurados automáticamente con valores aleatorios en base a parámetros ingresados por el usuario. Se ajustan parámetros como el tipo de optimización (maximización o minimización), el número de variables y restricciones, y los rangos de valores posibles para los coeficientes

Tras formular el problema, el archivo permite resolverlo utilizando el *solver* **SCIP** de la biblioteca **OR-Tools**, que maneja eficientemente problemas de optimización con tipos mixtos de variables. El *solver* busca una solución óptima que maximice o minimice la función objetivo, aplicando las restricciones generadas. Luego, se analizan los resultados y se determina si existe una solución óptima, si el problema es infactible (sin solución posible) o no acotado (solución indefinida). Este archivo es particularmente útil para:

- **Problemas Complejos en Investigación:** Genera problemas de optimización donde se combinan variables enteras y continuas, ideales para estudios en áreas como logística y operaciones.
- **Aplicaciones Industriales:** Facilita la simulación de escenarios de producción y planificación, donde la naturaleza mixta de las variables se ajusta a decisiones de asignación de recursos.
- **Desarrollo y Pruebas de Algoritmos:** Ofrece una base para evaluar algoritmos de optimización en problemas con estructuras mixtas, permitiendo analizar la eficacia de métodos para programación lineal mixta.
- **Educación y Capacitación:** Útil en entornos de aprendizaje para ilustrar la complejidad adicional de problemas mixtos en la optimización.

Método `generar_PLM`: Esta función genera un problema de PLM configurando aleatoriamente una función objetivo y restricciones.

Algorithm 15: `generar_PLM(tipo.optimizacion, num.variables, num.restricciones, num.enteras, num.continuas, num.binarias, obj.coef.range, restric.coef.range, right.coef.range, num.less.equal, num.greater.equal, num.equal, integer.coeffs)`

Paso 1: Generar la función objetivo con coeficientes
`coef_objetivo` ← valores aleatorios en `obj.coef.range` con `num.variables` elementos
 if `integer.coeffs` es verdadero then
 Redondear `coef_objetivo` a enteros
 else
 Redondear `coef_objetivo` a 2 decimales
 if `tipo.optimizacion` es "max" then
 `funcion_objetivo` ← "max z =" seguido de la suma de términos `coef_objetivo` y variables
 else
 `funcion_objetivo` ← "min z =" seguido de la suma de términos `coef_objetivo` y variables

Paso 2: Generar restricciones con coeficientes
`restricciones` ← lista vacía
`tipos_restricciones` ← lista de símbolos "<=", ">=", "=" basada en `num.less.equal`, `num.greater.equal` y `num.equal`
 Mezclar `tipos_restricciones` aleatoriamente
 for cada restricción de `num.restricciones` do
 `coef_restriccion` ← valores aleatorios en `restric.coef.range` con `num.variables` elementos
 if `integer.coeffs` es verdadero then
 Redondear `coef_restriccion` a enteros
 else
 Redondear `coef_restriccion` a 2 decimales
 `lado_derecho` ← valor aleatorio en `right.coef.range`, redondeado si es necesario
 `restriccion` ← combinación de coeficientes, variables y tipo de `tipos_restricciones`
 Añadir `restriccion` a `restricciones`

Paso 3: Definir tipos de variables
 Seleccionar aleatoriamente `indices.binarias`, `indices.enteras`, `indices.continuas`
`naturalidad` ← combinar descripciones de variables binarias, enteras y continuas
 return `funcion_objetivo`, `restricciones`, `naturalidad`

Figura 24: Pseudocódigo de método `generar_PLM`. Fuente: Elaboración propia.

Método `plm_solver`: Este método toma el problema generado y utiliza **OR-Tools** para resolverlo, obteniendo una solución óptima o determinando si el problema es infactible o no acotado.

Algorithm 16: `plm_solver(funcion_objetivo, restricciones, naturalidad)`

```

Paso 1: Crear el solver con OR-Tools
solver ← instancia de solver SCIP
if no se pudo crear solver then
    Mostrar mensaje: "Solver no disponible."
    return None, None
Paso 2: Obtener coeficientes de la función objetivo
coeficientes_obj ← lista de coeficientes extraídos de funcion_objetivo
num_variables ← longitud de coeficientes_obj
Paso 3: Crear variables por tipo
variables ← lista de variables (binarias, enteras, continuas) en solver
Paso 4: Definir la función objetivo
if funcion_objetivo empieza con "max" then
    Maximizar la suma de coeficientes y variables en solver
else
    Minimizar la suma de coeficientes y variables en solver
Paso 5: Agregar restricciones
for cada restriccion en restricciones do
    Extraer coeficientes, signo y lado_derecho de restriccion
    if signo es " $\leq$ " then
        Añadir restricción con  $\leq$  en solver
    else if signo es " $\geq$ " then
        Añadir restricción con  $\geq$  en solver
    else if signo es "=" then
        Añadir restricción con = en solver
Paso 6: Resolver el problema
status ← ejecutar solver
Paso 7: Interpretar el resultado
if status es OPTIMAL then
    resultado ← valores de las variables solución
    valor_funcion_objetivo ← valor óptimo de la función objetivo
    return valor_funcion_objetivo, resultado
else if status es INFEASIBLE then
    Mostrar mensaje: "El problema es infactible."
    return "Infactible", "Infactible"
else if status es UNBOUNDED then
    Mostrar mensaje: "El problema es no acotado."
    return "No acotado", "No acotado"
else
    Mostrar mensaje: "No se encontró una solución óptima."
    return "Sin solución", "Sin solución"

```

Figura 25: Pseudocódigo de método *plm_solver*. Fuente: Elaboración propia.

CAPÍTULO 4: VALIDACIÓN DE LA APLICACIÓN

4.1 Introducción

El propósito de este capítulo es presentar el proceso de validación de la aplicación desarrollada, asegurando que esta cumple con los objetivos y requisitos establecidos al inicio del proyecto. La validación no solo busca demostrar que la aplicación genera modelos de programación lineal correctamente estructurados, sino también que estos modelos cumplen con los estándares de calidad y utilidad esperados para apoyar la enseñanza en la asignatura **INF-292 Optimización**.

Dado el rol central que juega esta herramienta en facilitar la generación y resolución de problemas de PL, es esencial que se valide su capacidad para producir modelos matemáticos coherentes y funcionales en distintos tipos de problemas, tales como **programación lineal continua (PL)**, **programación lineal entera (PLE)**, **programación lineal mixta (PLM)** y **programación lineal binaria (PLB)**. Cada tipo de modelo representa una clase de problema específica y, por tanto, la herramienta debe garantizar que estos se generen y resuelvan adecuadamente.

La validación también se extiende a la **interacción con solvers externos** para la resolución de estos problemas. A través de una conexión sólida con *solvers* reconocidos (como MPS, lp_solve, LINDO), la aplicación asegura que los modelos generados puedan exportarse y procesarse de forma eficaz en diferentes entornos académicos y profesionales, ofreciendo flexibilidad y adaptabilidad a los usuarios.

Otro aspecto clave de la validación es la **usabilidad de la interfaz gráfica**. Dado que la herramienta está destinada tanto a estudiantes como a docentes, debe ser intuitiva y fácil de manejar, independientemente de la experiencia previa del usuario en optimización y programación lineal. Una interfaz accesible facilita que los usuarios configuren diversos parámetros, experimenten con diferentes configuraciones y obtengan resultados precisos sin enfrentar dificultades técnicas.

Finalmente, se considera la **eficiencia y robustez de la aplicación**. La eficiencia se mide en términos de los tiempos de respuesta para generar y resolver modelos, factor esencial en un contexto educativo donde el tiempo es limitado y la rapidez de generación y ejecución puede influir en la experiencia de aprendizaje. La robustez, por otro lado, se refiere a la capacidad de la aplicación para manejar entradas no válidas o casos extremos sin interrupciones graves en su funcionamiento. Esto incluye ofrecer mensajes de error claros y manejar adecuadamente los escenarios en los que el usuario proporciona datos atípicos o fuera de los rangos esperados.

En las subsecciones de este capítulo se detallan los **criterios de validación** definidos para evaluar la aplicación, las **pruebas realizadas** para comprobar su funcionalidad y rendimiento, y los **resultados obtenidos**, acompañados de una discusión sobre cómo estos resultados demuestran el cumplimiento de los objetivos. La sección final de este capítulo

presenta una **conclusión de la validación**, resumiendo los hallazgos clave y destacando las áreas en las que la aplicación cumple satisfactoriamente, así como aquellas en las que podría beneficiarse de mejoras futuras.

4.2 Criterios de Validación

La validación de una herramienta de software es un proceso crucial que garantiza que el sistema desarrollado cumpla con los requisitos funcionales y no funcionales establecidos, asegurando que sea confiable, eficiente y útil en su aplicación prevista. En el contexto de esta aplicación, los criterios de validación definidos abarcan tanto aspectos técnicos como de experiencia del usuario, lo que permite una evaluación integral del sistema. A continuación, se describen en detalle los criterios seleccionados y las métricas empleadas para evaluar cada uno de ellos.

4.2.1 Exactitud de los Modelos Generados

La exactitud es el fundamento de cualquier herramienta que genere modelos matemáticos, especialmente en un entorno académico donde los estudiantes y profesores dependen de la validez de estos modelos para practicar y evaluar conceptos de programación lineal. Este criterio se centra en garantizar que la aplicación genere modelos de programación lineal que sean matemáticamente correctos y cumplan con las expectativas de la asignatura INF-292.

- **Formulación de la Función Objetivo:** Se verifica que la función objetivo se construya correctamente en función de los parámetros definidos por el usuario. Esto incluye asegurar que los coeficientes y términos de las variables de decisión sean precisos y que la función se exprese correctamente como un problema de maximización o minimización, según la selección del usuario.
- **Construcción de Restricciones:** Las restricciones generadas deben representar correctamente las relaciones definidas por el usuario. La validación comprueba que cada restricción se traduzca en una ecuación o inecuación válida y que las relaciones ($\leq$, $\geq$, $=$) se mantengan conforme a las especificaciones ingresadas. Se evalúa la precisión en la asignación de coeficientes y términos independientes, garantizando que las restricciones sean consistentes y sin ambigüedades.
- **Configuración de las Variables de Decisión:** Se valida que las variables de decisión se definan correctamente según el tipo especificado (continuas, enteras o binarias). Esto implica asegurar que las variables enteras y binarias se restrinjan adecuadamente y que las variables continuas puedan asumir cualquier valor en el dominio definido por el problema. Además, se realiza una verificación cruzada para confirmar que los problemas de programación lineal mixta (PLM) se configuren correctamente al combinar variables continuas y discretas.

La exactitud de los modelos se valida a través de pruebas exhaustivas, generando múltiples instancias de problemas y comparando sus representaciones matemáticas con ejemplos de referencia y resultados esperados.

4.2.2 Compatibilidad con Solvers Externos

La capacidad de la aplicación para interactuar con *solvers* externos es crítica para su funcionalidad y versatilidad. Este criterio evalúa si los modelos generados pueden ser exportados y resueltos sin problemas por *solvers* reconocidos en la industria y el ámbito académico. La validación de compatibilidad abarca varios aspectos:

- **Exportación a Formatos Estándar:** Los modelos deben poder exportarse a formatos como **MPS**, **lp_solve** y **LINDO**. La estructura del archivo exportado se revisa para asegurar que cumpla con las especificaciones de cada formato, incluyendo la correcta codificación de la función objetivo, las restricciones y las definiciones de variables.
- **Resolución Exitosa por Solvers:** Se realiza una serie de pruebas en las que los modelos exportados se cargan en los *solvers* MPS, lp_solve y LINDO para confirmar que estos puedan procesar y resolver los problemas sin errores. Se verifica que los *solvers* devuelvan soluciones válidas y consistentes con las expectativas matemáticas del problema.
- **Consistencia de Resultados:** Se compara la solución obtenida por diferentes *solvers* para el mismo modelo, asegurando que los valores óptimos de la función objetivo y de las variables de decisión sean equivalentes, dentro de los márgenes de tolerancia aceptables. Cualquier discrepancia significativa se analiza para determinar si es un problema del modelo exportado o una limitación del *solver*.

La compatibilidad también implica evaluar la facilidad con la que los usuarios pueden integrar la aplicación en flujos de trabajo existentes, permitiendo un uso eficiente de los modelos generados en otros sistemas de optimización.

4.2.3 Facilidad de Uso de la Interfaz

La interfaz de usuario (GUI) desempeña un papel crucial en la experiencia del usuario, especialmente para una aplicación educativa destinada a estudiantes y profesores. La facilidad de uso se evalúa mediante los siguientes factores:

- **Intuitividad y Diseño de la Interfaz:** Se mide la facilidad con la que los usuarios pueden navegar por la aplicación, ajustar parámetros y generar modelos. La validación incluye pruebas de usabilidad con usuarios reales, recolectando retroalimentación sobre la claridad y simplicidad de los elementos de la interfaz, como menús, botones y cuadros de diálogo.

- **Configuración de Parámetros:** Se evalúa la facilidad con la que los usuarios pueden definir el número de variables, restricciones, y el tipo de optimización. El sistema debe permitir configuraciones rápidas y sin complicaciones, incluso para usuarios con poca experiencia en programación lineal. Se realiza una revisión de cómo la interfaz guía a los usuarios para minimizar errores de entrada.
- **Claridad en la Presentación de Resultados:** La forma en que los resultados se presentan al usuario es crítica para la comprensión y análisis. Se valida que los resultados se muestren de manera organizada y legible, con opciones claras para exportar o modificar el modelo. La interfaz debe permitir al usuario visualizar rápidamente la función objetivo, las restricciones y los valores óptimos de las variables.

Este criterio se mide a través de encuestas y pruebas de experiencia de usuario, evaluando aspectos como la curva de aprendizaje y la satisfacción general con el diseño y la funcionalidad de la interfaz.

4.2.4 Eficiencia y Tiempos de Ejecución

La eficiencia es un criterio técnico que se mide en términos de tiempo de respuesta y uso de recursos del sistema. Dado el entorno académico en el que se utilizará la aplicación, es importante que el tiempo de generación de modelos y la resolución de los mismos sean razonables y no afecten la experiencia del usuario.

- **Tiempo de Generación de Modelos:** Se mide cuánto tiempo tarda la aplicación en generar modelos de programación lineal en función de la complejidad del problema (es decir, el número de variables y restricciones). La aplicación debe poder manejar problemas de tamaño moderado sin demoras significativas, y se comparan los tiempos de generación con referencias de herramientas similares.
- **Desempeño en la Resolución con Solvers:** Se evalúa el tiempo que tardan los *solvers* en resolver los problemas generados, especialmente para modelos complejos o mixtos. Los tiempos de ejecución se registran y analizan para identificar posibles cuellos de botella en el proceso.

4.3 Pruebas Realizadas

Para asegurar que la aplicación desarrollada cumple con los criterios de validación descritos anteriormente, se llevaron a cabo diversas pruebas que abarcan aspectos funcionales, de integración, y de rendimiento, así como la capacidad del sistema para manejar entradas no válidas o casos extremos. Estas pruebas se diseñaron para evaluar la aplicación de manera exhaustiva y detectar cualquier posible fallo o área de mejora. A continuación, se describen las pruebas realizadas y la metodología empleada.

4.3.1 Pruebas Funcionales

Para validar que la aplicación genera correctamente los modelos de programación lineal en sus diferentes variantes (PL, PLE, PLB, PLM), se realizaron pruebas funcionales detalladas. Estas pruebas verifican que las configuraciones establecidas por el usuario en la interfaz gráfica se traduzcan correctamente en modelos matemáticos válidos y que las soluciones obtenidas sean consistentes y precisas. A continuación, se describen las pruebas y los resultados esperados. Todas las pruebas pueden encontrarse en la sección de Anexos.

Prueba 1: Generación de Problemas de Programación Lineal (PL)

- **Objetivo:** Verificar que la aplicación genere correctamente problemas de programación lineal continua, con coeficientes y restricciones definidos por el usuario.
- **Configuración de Prueba:**
 - Tipo de optimización: max
 - Número de variables enteras: 0
 - Número de variables continuas: 3
 - Número de variables binarias: 0
 - Rango de coeficientes de la función objetivo: -3 a 3
 - Rango de coeficientes de las restricciones: -4 a 4
 - Rango de coeficientes del lado derecho: -5 a 5
 - Cantidad de restricciones $\leq$: 1
 - Cantidad de restricciones $\geq$: 1
 - Cantidad de restricciones $=$: 1

Resultados Obtenidos Prueba 1

El problema generado fue un **Problema de Programación Lineal** con la siguiente formulación:

- **Función Objetivo:** $\max z = 0.0x_1 + -3.0x_2 + 0.0x_3$
- **Restricciones:**
 - $-3.0x_1 + 3.0x_2 + -2.0x_3 \geq 4.0$
 - $-2.0x_1 + -3.0x_2 + 1.0x_3 \leq 1.0$
 - $1.0x_1 + 2.0x_2 + -2.0x_3 = 4.0$

- **Naturaleza de las Variables:**
 - $x_1 \geq 0$
 - $x_2 \geq 0$
 - $x_3 \geq 0$
- **Resultados de la Solución:**
 - Valores de las variables: $\{x_1': 0.444, x_2': 1.778, x_3': 0.0\}$
 - Valor de la función objetivo: -5.333
- **Análisis**
 - **Validación de la Función Objetivo:** La función objetivo fue correctamente generada como una maximización con coeficientes dentro del rango especificado. Se puede observar que los coeficientes de x_1 , x_2 , y x_3 cumplen con los límites definidos de -3 a 3.
 - **Verificación de las Restricciones:** Las tres restricciones se generaron correctamente, cada una con coeficientes en el rango de -4 a 4. Los signos de las desigualdades ($\leq$, $\geq$, $=$) corresponden a la configuración especificada.
 - **Resultados del Solver:** El *solver* GLOP proporcionó una solución óptima con valores de las variables que cumplen con las restricciones y optimizan la función objetivo. El valor de la función objetivo y las variables de decisión fueron calculados y mostrados adecuadamente.

Conclusión: La prueba fue exitosa, demostrando que la aplicación puede generar correctamente problemas de programación lineal continua según los parámetros configurados por el usuario. Los coeficientes y las restricciones respetan los rangos definidos, y el *solver* devuelve una solución óptima cuando es posible. Esto confirma la funcionalidad básica del generador de problemas PL.

Prueba 2: Generación de Problemas de Programación Lineal Entera (PLE)

- **Objetivo:** Asegurar que la aplicación genere correctamente problemas de programación lineal en los que todas las variables son enteras, cumpliendo con las especificaciones del usuario.
- **Configuración de la Prueba:**
 - Tipo de optimización: max
 - Número de variables enteras: 3

- Número de variables continuas: 0
- Número de variables binarias: 0
- Rango de coeficientes de la función objetivo: -3 a 3
- Rango de coeficientes de las restricciones: -4 a 4
- Rango de coeficientes del lado derecho: -5 a 5
- Cantidad de restricciones $\leq$: 1
- Cantidad de restricciones $\geq$: 1
- Cantidad de restricciones $=$: 1

Resultados Obtenidos Prueba 2

El problema generado fue un **Problema de Programación Lineal Entera** con la siguiente formulación:

- **Función Objetivo:** $\max z = -1.0x_1 + -2.0x_2 + -3.0x_3$
- **Restricciones:**
 - $3.0x_1 + 3.0x_2 + -4.0x_3 \geq 2.0$
 - $3.0x_1 + 0.0x_2 + -1.0x_3 \leq 0.0$
 - $-3.0x_1 + -3.0x_2 + 2.0x_3 = -4.0$
- **Naturaleza de las Variables:**
 - $x_1 \in Z$
 - $x_2 \in Z$
 - $x_3 \in Z$
- **Resultados de la Solución:**
 - Valores de las variables: $\{x_1': 0.0, x_2': 2.0, x_3': 1.0\}$
 - Valor de la función objetivo: -7.0
- **Análisis**
 - **Validación de la Función Objetivo:** La función objetivo se generó correctamente como una maximización con coeficientes en el rango de -3 a 3, respetando la configuración de variables enteras.

- **Verificación de las Restricciones:** Las restricciones cumplen con los tipos especificados ($\leq$, $\geq$, $=$) y tienen coeficientes dentro del rango de -4 a 4. Las desigualdades se formularon correctamente, y el lado derecho de las restricciones está dentro del rango establecido.
- **Resultados del Solver:** El *solver* SCIP proporcionó una solución óptima. Los valores de las variables enteras son consistentes con las restricciones, y el valor de la función objetivo fue calculado correctamente.

Conclusión: La prueba fue exitosa. La aplicación demostró su capacidad para generar problemas de programación lineal con variables enteras y para resolverlos adecuadamente utilizando SCIP. La generación y la solución del problema fueron correctas, lo que valida la funcionalidad para este tipo de problemas.

Prueba 3: Generación de Problemas de Programación Lineal Binaria (PLB)

- **Objetivo:** Validar que la aplicación genere problemas de programación lineal donde todas las variables son binarias (0 o 1), y que estos problemas se formulen correctamente según las especificaciones del usuario.
- **Configuración de la Prueba:**
 - Tipo de optimización: max
 - Número de variables enteras: 0
 - Número de variables continuas: 0
 - Número de variables binarias: 3
 - Rango de coeficientes de la función objetivo: -3 a 3
 - Rango de coeficientes de las restricciones: -4 a 4
 - Rango de coeficientes del lado derecho: -5 a 5
 - Cantidad de restricciones $\leq$: 1
 - Cantidad de restricciones $\geq$: 1
 - Cantidad de restricciones $=$: 1

Resultados Obtenidos Prueba 3

El problema generado fue un **Problema de Programación Lineal Binaria** con la siguiente formulación:

- **Función Objetivo:** $\max z = -2.0x_1 + -1.0x_2 + 2.0x_3$

- **Restricciones:**
 - $-4.0x_1 + -3.0x_2 + 2.0x_3 \geq -4.0$
 - $-3.0x_1 + -1.0x_2 + -0.0x_3 \leq 4.0$
 - $-3.0x_1 + -1.0x_2 + 1.0x_3 = 1.0$
- **Naturaleza de las Variables:**
 - $x_1 \in \{0, 1\}$
 - $x_2 \in \{0, 1\}$
 - $x_3 \in \{0, 1\}$
- **Resultados de la Solución:**
 - Valores de las variables: $\{x_1': 0.0, x_2': 0.0, x_3': 1.0\}$
 - Valor de la función objetivo: 2.0
- **Análisis**
 - **Validación de la Función Objetivo:** La función objetivo fue correctamente formulada como una maximización, y los coeficientes están dentro del rango definido de -3 a 3.
 - **Verificación de las Restricciones:** Las restricciones se generaron correctamente, con signos de desigualdad ($\leq$, $\geq$, $=$) y coeficientes en el rango de -4 a 4. Las desigualdades y la ecuación son coherentes con los parámetros establecidos.
 - **Resultados del Solver:** El *solver* SCIP proporcionó una solución óptima, donde las variables binarias se encuentran en los valores 0 o 1, como era esperado. El valor de la función objetivo se calculó correctamente.

Conclusión: La prueba fue exitosa, demostrando que la aplicación puede generar y resolver problemas de programación lineal binaria de manera precisa y eficiente. La solución obtenida es consistente con el modelo matemático generado, confirmando la validez y funcionalidad de este módulo.

Prueba 4: Generación de Problemas de Programación Lineal Binaria (PLM)

- **Objetivo:** Verificar que la aplicación genere correctamente problemas de programación lineal mixta (PLM) con una combinación de variables enteras, continuas y binarias, y que estos problemas se resuelvan adecuadamente utilizando el *solver* SCIP.

- **Configuración de la Prueba:**
 - Tipo de optimización: max
 - Número de variables enteras: 1
 - Número de variables continuas: 1
 - Número de variables binarias: 1
 - Rango de coeficientes de la función objetivo: -3 a 3
 - Rango de coeficientes de las restricciones: -4 a 4
 - Rango de coeficientes del lado derecho: -5 a 5
 - Cantidad de restricciones $\leq$: 1
 - Cantidad de restricciones $\geq$: 1
 - Cantidad de restricciones $=$: 1

Resultados Obtenidos Prueba 4

El problema generado fue un **Problema de Programación Lineal Mixta** con la siguiente formulación:

- **Función Objetivo:** $\max z = -1.0x_1 + -2.0x_2 + -2.0x_3$
- **Restricciones:**
 - $-4.0x_1 + 3.0x_2 + -3.0x_3 \leq 1.0$
 - $0.0x_1 + 4.0x_2 + 2.0x_3 \geq -1.0$
 - $-3.0x_1 + -1.0x_2 + -3.0x_3 = -4.0$
- **Naturaleza de las Variables:**
 - $x_1 \in \mathbb{Z}$ (Variable entera)
 - $x_2 \in R$ (Variable continua)
 - $x_3 \in \{0, 1\}$ (Variable binaria)
- **Resultados de la Solución:**
 - Valores de las variables: $\{x_1': 1.0, x_2': 1.0, x_3': 0.0\}$
 - Valor de la función objetivo: -3.0

- **Análisis**

- **Validación de la Función Objetivo:** La función objetivo fue correctamente formulada como una maximización, y los coeficientes están dentro del rango -3 a 3, cumpliendo con las especificaciones.
- **Verificación de las Restricciones:** Las restricciones se generaron de manera correcta, con los signos de desigualdad y los coeficientes en el rango definido. Cada tipo de restricción ($\leq$, $\geq$, $=$) fue respetado.
- **Clasificación de las Variables:** Las variables fueron asignadas correctamente: x_1 como entera, x_2 como continua, y x_3 como binaria. Esta clasificación es fundamental para los problemas de programación lineal mixta.
- **Resultados del Solver:** El *solver* SCIP devolvió una solución óptima, con los valores de las variables y la función objetivo calculados correctamente. Las variables cumplen con las restricciones y la naturaleza especificada.

Conclusión: La prueba fue exitosa, validando que la aplicación puede generar y resolver problemas de programación lineal mixta con precisión y consistencia. El *solver* proporcionó una solución que cumple con las restricciones y la naturaleza de las variables, demostrando la funcionalidad y robustez de la herramienta para problemas mixtos.

4.3.2 Pruebas de Integración

Las pruebas de compatibilidad se dividieron en tres secciones, cada una enfocada en un formato de exportación específico: MPS, *lp_solve* y LINDO. Para cada formato, se evaluaron la exportación correcta, la resolución por parte de los *solvers* y la consistencia de los resultados.

4.3.2.1 Pruebas con Formato MPS

A. Exportación a MPS

- **Descripción del Modelo:** Se generó un problema de PL y se exportó en formato MPS. La imagen proporcionada muestra el archivo MPS generado, que contiene las secciones NAME, ROWS, COLUMNS, RHS, BOUNDS y ENDATA.
- **Interpretación del Modelo:**
 - **Función Objetivo:** $\min z = 1.0x_1 - 2.0x_2 + 0.0x_3$
 - **Restricciones:**

- $2.0x_1 + 3.0x_2 - 3.0x_3 = 1.0$ (Restricción R1)
- $-3.0x_1 - 3.0x_2 - 2.0x_3 \geq 1.0$ (Restricción R2)
- $3.0x_1 - 2.0x_2 + 1.0x_3 \leq 2.0$ (Restricción R3)
- **Naturaleza de las Variables:**
 - x_1 , x_2 , y x_3 son variables continuas sin límites específicos indicados en el archivo.
- **Estructura del Archivo:**
 - La sección NAME identifica el problema como “PROBLEM”.
 - ROWS define la función objetivo (OBJ) y las restricciones (R1, R2, R3) con sus respectivos tipo: igualdad (E), mayor o igual (G), y menor o igual (L), respectivamente.
 - COLUMNS muestra los coeficientes de las variables (x_1 , x_2 , x_3) en la función objetivo y en cada restricción.
 - RHS detalla los valores del lado derecho de las restricciones.
 - BOUNDS no contiene restricciones adicionales en este caso, y el archivo termina con ENDATA.
- **Revisión Manual:** Se revisó la estructura del archivo y se confirmó que cumplía con los requisitos de lp_solve, sin errores de sintaxis ni problemas de codificación.

B. Resolución con *Solver* LPSolve

- **Procedimiento:**
 - El archivo MPS generado fue cargado en el *solver* **LPSolve**.
 - Se ejecutó el problema utilizando los métodos DUAL simplex y PRIMAL simplex, con la estrategia de precios **Devex**.
 - Se observó si el *solver* procesaba el archivo sin errores y proporcionaba una solución válida.
- **Resultados:**
 - El *solver* **LPSolve** determinó que el modelo es **inviable**, lo que significa que no existe una solución que satisfaga todas las restricciones simultáneamente.
 - Se realizaron 2 iteraciones, y el *solver* no encontró soluciones viables.

- No se reportaron errores de sintaxis o estructura en el archivo MPS; sin embargo, el problema matemático en sí fue declarado como infactible.

C. Resultado Detallado del Solver

- **Detalles del Modelo:**
 - Nombre del modelo: 'PROBLEM'
 - Objetivo: Minimizar (OBJ)
 - Tamaño del modelo: 3 restricciones, 3 variables, 9 elementos no nulos.
- **Métodos Utilizados:**
 - Se usaron DUAL simplex para la fase 1 y PRIMAL simplex para la fase 2.
 - Estrategia de precios: **Devex**
- **Tiempo de Ejecución:**
 - Tiempo total de carga de datos: 0.002 segundos
 - Tiempo en el proceso de simplex: 0.007 segundos
 - Tiempo total: 0.011 segundos
- **Información Adicional:**
 - La matriz de restricciones tenía una norma infinita de 3, con un rango dinámico también de 3.
 - El *solver* utilizó la versión 5.5.2.11 de **LPSolve**, que es adecuada para este tipo de análisis.

D. Análisis del Resultado

El *solver* indicó que el modelo es infactible, lo cual sugiere que las restricciones especificadas no permiten encontrar una solución viable. Este resultado podría deberse a una formulación incorrecta o a configuraciones incompatibles de los coeficientes y las restricciones.

- **Posibles Causas:**
 - Restricciones conflictivas que no permiten la existencia de una solución factible.
 - Coeficientes y valores del lado derecho que generan inconsistencias.

- **Próximos Pasos:**

- Revisar y ajustar la formulación del modelo, asegurando que las restricciones sean coherentes y factibles.
- Realizar más pruebas con diferentes parámetros para identificar configuraciones que permitan una solución.

Conclusión de la Prueba con Formato MPS: La exportación en formato MPS fue correcta desde un punto de vista técnico, y el archivo fue procesado sin problemas por el *solver* **LPSolve**. Sin embargo, el modelo se declaró infactible, lo que destaca la necesidad de revisar y ajustar la formulación de las restricciones para lograr soluciones viables. La aplicación ha demostrado ser capaz de generar y exportar modelos MPS de manera precisa, validando su funcionalidad en este aspecto.

4.3.2.2 Pruebas con Formato lp_solve

A. Exportación a lp_solve

- **Descripción del Modelo:** Se generó un problema de programación lineal mixta y se exportó en formato lp_solve. La imagen proporcionada muestra el archivo exportado, que detalla la función objetivo, las restricciones y la naturaleza de las variables:
 - **Función Objetivo:** $\max z = -2.0x_1 - 2.0x_2 - 1.0x_3$
 - **Restricciones:**
 - $-1.0x_1 - 2.0x_2 + 2.0x_3 \geq -2.0$
 - $-2.0x_1 - 2.0x_2 - 1.0x_3 = -2.0$
 - $-1.0x_1 - 3.0x_2 + 2.0x_3 \leq -3.0$
 - **Naturaleza de las Variables:** Las definiciones de las variables se especifican al final, diferenciando entre enteras (*int* x_1 , *int* x_2 ;) y binarias (*bin* x_3 ;)
 - x_1 : Variable entera
 - x_2 : Variable entera
 - x_3 : Variable binaria (solo puede tomar valores 0 o 1)
- **Revisión Manual:** Se revisó la estructura del archivo y se confirmó que cumplía con los requisitos de lp_solve, sin errores de sintaxis ni problemas de codificación.

B. Resolución con *Solver* LPSolve

- **Procedimiento:**
 - El archivo generado fue cargado en el *solver* LPSolve, ejecutando la optimización.
 - Se utilizaron los métodos DUAL simplex para la fase 1 y PRIMAL simplex para la fase 2, con la estrategia de precios **Devex**.
 - El objetivo era observar si el *solver* procesaba el archivo correctamente y obtenía una solución sin errores.
- **Resultados:**
 - **Solución Relajada:** Se alcanzó después de 2 iteraciones utilizando la base B&B.
 - **Solución Factible:** Se encontró tras 2 iteraciones, sin necesidad de explorar nodos adicionales (gap del 0.0%).
 - **Solución Óptima:** Lograda tras 2 iteraciones, manteniendo una alta precisión numérica.
- **Solución del Problema:**
 - **Valor Óptimo de la Función Objetivo:** -2
 - **Valores de las Variables de Decisión:**
 - $x_1 = 0 \quad x_2 = 0 \quad x_3 = 1$

C. Detalles Técnicos y Precisión

- **Precisión Numérica:** El *solver* mostró una exactitud relativa de $||*|| = 2.22045 \times 10^{-16}$, lo que refleja una alta precisión en los cálculos.
- **Información del Solver:** **LPSolve** versión 5.5.2.11, adecuado para sistemas operativos de 32 bits, con variables de 64 bits en punto flotante.
- **Tiempo de Ejecución:**
 - Carga de datos: 0.002 segundos
 - Ejecución del Simplex: 0.007 segundos
 - Tiempo total: 0.011 segundos

D. Información Adicional

- No se realizaron refactorizaciones durante el proceso.

- La matriz de restricciones presentó una norma infinita de 3 y un rango dinámico también de 3.
- El *solver* alcanzó un nivel máximo de B&B de 1, con un orden MIP de 0.2x en la solución óptima.

E. Análisis del Resultado

El *solver* **LPSolve** resolvió el problema de manera óptima y eficiente, confirmando que el archivo `lp_solve` generado era estructuralmente correcto y compatible. No se presentaron errores de ejecución, y el *solver* proporcionó una solución precisa y consistente con las expectativas del modelo.

Conclusión de la Prueba con Formato `lp_solve`: La aplicación ha demostrado su capacidad para exportar modelos en formato `lp_solve` que son completamente compatibles con el *solver* LPSolve. La ejecución fue precisa y eficiente, validando la funcionalidad de la exportación y el procesamiento del modelo.

4.3.2.3 Pruebas con Formato LINDO

A. Exportación a LINDO

- **Descripción del Modelo:** Se generó un problema de programación lineal mixta y se exportó en formato LINDO. La imagen proporcionada muestra el archivo exportado, que detalla la función objetivo, las restricciones y la naturaleza de las variables.
- **Estructura del Archivo:**
 - La función objetivo comienza con el prefijo `max`, seguido de los coeficientes y variables: $\max 0.0x_1 - 2.0x_2 - 1.0x_3$
 - Las restricciones se enumeran bajo la etiqueta `st` (sujeto a), con los signos de desigualdad y el lado derecho correspondiente:
 - $1.0x_1 - 1.0x_2 + 2.0x_3 \leq 2.0$
 - $-2.0x_1 - 2.0x_2 - 1.0x_3 = -1.0$
 - $3.0x_1 + 1.0x_2 + 2.0x_3 \geq -1.0$
 - Las definiciones de las variables se especifican al final:
 - `free x2:` x_2 es una variable continua sin restricciones adicionales.
 - `gin x1:` x_1 es una variable entera general.

- int x3: x_3 es una variable entera específica.
- **Revisión Manual:** Se revisó la estructura del archivo y se confirmó que cumplía con los requisitos de LINDO, sin errores de sintaxis ni problemas de codificación.

B. Resolución con *Solver* Lingo 20.0

- **Procedimiento:**
 - El archivo generado fue cargado en el *solver* Lingo 20.0.
 - Se ejecutó la optimización para observar si el *solver* procesaba el archivo correctamente y proporcionaba una solución sin errores.
- **Resultados:**
 - **Estado del Solver:** El *solver* encontró una solución global óptima.
 - **Valor Óptimo de la Función Objetivo:** 1.0
 - **Valores de las Variables de Decisión:**
 - $x_1 = 1.0 \quad x_2 = -0.5 \quad x_3 = 0.0$
 - **Costes Reducidos:** Se observaron costes reducidos asociados a las variables, lo que proporciona información sobre cómo impactarían las restricciones adicionales.
 - **Superávit o Slack en las Restricciones:** Los valores de *slack* o superávit mostraron que las restricciones se cumplían de manera precisa.
 - **Tiempo de Ejecución:** El *solver* Lingo procesó el problema de manera eficiente en un tiempo de ejecución corto (0.04 segundos).

C. Detalles Técnicos y Precisión

- **Información del Solver:** Lingo 20.0 con licencia de uso evaluativa, versión adecuada para la optimización de problemas lineales mixtos (MILP).
- **Clase del Modelo:** MILP (Mixed Integer Linear Programming)
- **Variables y Restricciones:**
 - Total de variables: 3 (con 2 variables enteras)
 - Total de restricciones: 4, sin restricciones no lineales
 - Elementos no nulos: 11
- **Precisión:** No se encontraron infactibilidades, y el modelo fue resuelto con alta precisión.

D. Análisis del Resultado

- El *solver* Lingo 20.0 resolvió el problema de manera óptima y eficiente, confirmando que el archivo LINDO generado era estructuralmente correcto y compatible.
- **Consistencia de Resultados:** Los resultados obtenidos fueron coherentes y precisos, validando el correcto funcionamiento de la aplicación para exportar modelos en este formato.

Conclusión de la Prueba con Formato LINDO: La aplicación ha demostrado su capacidad para exportar modelos en formato LINDO de manera precisa y compatible con el *solver* Lingo 20.0. La ejecución fue eficiente, y la solución obtenida fue consistente con las especificaciones del problema, validando la funcionalidad de exportación y optimización de la aplicación.

4.3.3 Pruebas de Rendimiento

La eficiencia de la aplicación es fundamental para garantizar una experiencia de usuario satisfactoria en un entorno académico. Se evaluó el rendimiento de la aplicación en dos aspectos clave:

1. **Tiempo de Generación de Modelos:** Cuánto tarda la aplicación en generar modelos de programación lineal de diferentes tamaños y complejidades.
2. **Desempeño en la Resolución con Solvers:** Tiempo que toman los *solvers* (GLOP y SCIP) en resolver los modelos generados, especialmente en casos complejos o mixtos.

4.3.3.1. Tiempo de Generación de Modelos

Objetivo: Medir el tiempo que tarda la aplicación en generar modelos de programación lineal, variando el número de variables y restricciones, para evaluar su escalabilidad y eficiencia.

Procedimiento:

- **Configuración de Pruebas:** Se definieron cuatro tamaños de problemas.

Tabla 1: Configuración de Pruebas. Fuente: Elaboración propia.

Tamaño	Variables	Restricciones
Pequeño	3	2
Mediano	10	5

Grande	30	20
Muy Grande	50	30

- Para cada tamaño, se generaron modelos de los cuatro tipos: PL, PLE, PLB y PLM.

Medición de Tiempos:

- Se utilizó la función *time.time()* en Python para registrar el tiempo antes y después de la generación del modelo.
- Cada prueba se repitió 5 veces y se calculó el tiempo promedio de éstas.

Resultados:

- **Tiempos Promedio de Generación (en segundos):**

Tabla 2: Tiempos promedios de generación de problemas. Fuente: Elaboración propia.

Tamaño	PL	PLE	PLB	PLM
Pequeño	0,0	0,0	0,0	0,0
Mediano	0,0005	0,0005	0,0004	0.00008
Grande	0,00045	0,0005	0,0006	0,0005
Muy Grande	0,0012	0,0012	0,0011	0,0012

Análisis:

- **Escalabilidad:** La aplicación muestra una excelente escalabilidad en la generación de modelos, manteniendo tiempos de generación muy bajos incluso para problemas de tamaño "Muy Grande". Esto sugiere que el tiempo de generación crece de manera prácticamente insignificante en función de la complejidad del modelo.
- **Comparación entre Tipos de Modelos:** Los tiempos de generación son similares para todos los tipos de modelos (PL, PLE, PLB, PLM) en cada tamaño, lo cual indica que la complejidad del tipo de modelo no afecta significativamente el tiempo de generación. Esto es positivo, ya que implica que la aplicación es flexible y puede manejar diferentes tipos de problemas con tiempos consistentes.
- **Experiencia del Usuario:** Dado que los tiempos de generación son prácticamente instantáneos en todos los casos, la experiencia del usuario es óptima en este aspecto. Los usuarios pueden definir y generar modelos de cualquier tamaño sin

experimentar demoras, lo que contribuye a una interacción fluida y eficiente con la aplicación.

4.3.3.2. Desempeño en la Resolución con Solvers

Objetivo: Evaluar el tiempo que toman los *solvers* GLOP y SCIP en resolver los modelos generados, identificando posibles cuellos de botella y asegurando tiempos de respuesta razonables.

Procedimiento:

- **Asignación de Solvers:**
 - GLOP se utilizó para resolver modelos PL.
 - SCIP se empleó para los modelos PLE, PLB y PLM.
- **Medición de Tiempos:**
 - Se registró el tiempo antes y después de la llamada al *solver*.
 - Cada prueba se repitió 5 veces para obtener un promedio confiable.

Resultados

- **Tiempos Promedio de Resolución (en segundos):**

Tabla 3: Tiempos promedio de resolución de problemas. Fuente: Elaboración propia.

Tamaño	Solver	PL	PLE	PLB	PLM
Pequeño	GLOP	0,0046	-	-	-
Pequeño	SCIP	-	0.0072	0.0027	0.0046
Mediano	GLOP	0,0024	-	-	-
Mediano	SCIP	-	0,0083	0,0053	0,0040
Grande	GLOP	0,0024	-	-	-
Grande	SCIP	-	0,0113	0,0424	0.1077
Muy Grande	GLOP	0.0056	-	-	-
Muy Grande	SCIP	-	0,0126	0,4496	0,1359

Análisis:

- **Escalabilidad:** La escalabilidad de la aplicación es notable para problemas de programación lineal (PL) resueltos con GLOP, ya que mantiene tiempos de resolución bajos incluso para tamaños "Muy Grande" (0.0056 segundos). En cambio, los problemas PLE, PLB y PLM resueltos con SCIP presentan un aumento notable en los tiempos de resolución a medida que el tamaño del problema crece, especialmente en el caso de PLB "Muy Grande" (0.4496 segundos).
- **Comparación entre Tipos de Modelos:** Los problemas PL tienen los tiempos de resolución más bajos, lo que sugiere que son los más eficientes de resolver con la configuración actual. Los modelos PLE y PLM requieren más tiempo, especialmente para tamaños grandes y muy grandes, con PLB siendo el tipo de problema más demandante. Esto es un indicativo de que los modelos de programación lineal binaria y mixta, debido a su complejidad, exigen más recursos de procesamiento.
- **Experiencia del Usuario:** Para problemas pequeños y medianos, los tiempos de resolución son lo suficientemente bajos como para ofrecer una experiencia fluida al usuario. Sin embargo, en problemas grandes o muy grandes de PLE, PLB y PLM, los tiempos de resolución pueden llegar a impactar la experiencia, particularmente en PLB de tamaño "Muy Grande". Esto sugiere que para usuarios que trabajen con modelos muy complejos, podría ser necesario advertir sobre posibles demoras en el proceso de resolución.

4.3.3.3. Conclusiones

Eficiencia en la Generación: Los resultados de las pruebas muestran que la aplicación es altamente eficiente en la generación de modelos de programación lineal (PL, PLE, PLB, PLM), manteniendo tiempos prácticamente instantáneos incluso para los problemas de mayor tamaño. Esto demuestra que la estructura y optimización del código permiten manejar la generación de modelos sin comprometer el tiempo de respuesta, ofreciendo una experiencia ágil y efectiva para el usuario en esta fase.

Desempeño en la Resolución: El desempeño en la resolución varía dependiendo del tipo de modelo y del *solver* utilizado. Mientras que los problemas de PL son resueltos rápidamente con el *solver* GLOP incluso en tamaños grandes, los problemas de PLE, PLB y PLM resueltos con SCIP requieren más tiempo, especialmente en los tamaños grandes y muy grandes. Este resultado es esperado debido a la complejidad adicional de estos modelos. Sin embargo, para problemas pequeños y medianos, los tiempos de resolución se mantienen en un rango aceptable, permitiendo una interacción fluida en la mayoría de los casos.

Escalabilidad: La aplicación demuestra una escalabilidad adecuada en la generación de modelos, manteniendo tiempos de respuesta consistentes a medida que se incrementa el tamaño del problema. En la fase de resolución, sin embargo, se observan aumentos

significativos en los tiempos para modelos más complejos (PLE, PLB, PLM) al utilizar SCIP en problemas grandes y muy grandes. Estos resultados sugieren que la aplicación es capaz de manejar problemas de programación lineal a gran escala con eficiencia, pero podría experimentar un incremento en los tiempos de resolución para problemas complejos y de gran tamaño en *solvers* que requieren mayor capacidad computacional.

CAPÍTULO 5: ESTUDIO DE CARÁCTERÍSTICAS DE LAS SOLUCIONES

5.1 Introducción a las Características de las Soluciones

En la PL, las soluciones de un problema pueden adoptar diferentes formas según la estructura de la función objetivo y las restricciones impuestas. Es fundamental para la resolución y análisis de estos problemas comprender los tipos de soluciones que pueden surgir y sus implicancias. Los tipos principales de soluciones en programación lineal, según Hamdy A. Taha [Taha12] incluyen:

- **Solución única:** Ocurre cuando existe un único punto en la región factible que optimiza la función objetivo. Este tipo de solución se caracteriza por ser óptima y determinada sin ambigüedad.
- **Solución múltiple:** En algunos casos, un problema de PL puede tener múltiples soluciones óptimas, todas alcanzando el mismo valor de la función objetivo. Esto sucede cuando la función objetivo es paralela a una de las restricciones en la región factible, generando un conjunto de soluciones óptimas.
- **Solución ilimitada:** Ocurre cuando la función objetivo puede aumentar indefinidamente sin violar las restricciones del problema, lo cual indica una falta de límite superior o inferior en el espacio de soluciones.
- **Problema sin solución:** Se da cuando las restricciones definen una región factible vacía, es decir, no existe ningún conjunto de valores que satisfaga todas las restricciones simultáneamente.

5.2 Condiciones Matemáticas

5.2.1 Solución Única

Una solución única se presenta cuando la región factible se intersecta en un único vértice del politopo definido por las restricciones y cuando la función objetivo alcanza su valor máximo o mínimo en ese punto. Esto ocurre bajo las siguientes condiciones:

- La matriz de coeficientes en las restricciones es tal que el sistema tiene una única intersección factible.
- La función objetivo no es paralela a ninguna restricción en el punto de optimización.

5.2.2 Solución Múltiple

Para que existan soluciones múltiples, la función objetivo debe ser paralela a una restricción que limita la región factible. Esto implica:

- La dirección de la función objetivo coincide con la dirección de una de las restricciones en el espacio de soluciones.
- Esta configuración genera un conjunto de puntos en la frontera de la región factible que optimizan la función objetivo de igual forma.

5.2.3 Solución Ilimitada

La solución ilimitada ocurre cuando:

- La función objetivo no está acotada dentro de la región factible.
- La dirección de optimización se aleja de los límites impuestos por las restricciones, lo que permite que el valor de la función objetivo crezca o disminuya sin restricción.

5.2.4 Problema Sin Solución

Un problema sin solución se da cuando:

- No hay puntos comunes entre las restricciones, resultando en una región factible vacía.
- Esto ocurre frecuentemente cuando las restricciones se contradicen entre sí, imposibilitando que alguna solución cumpla todas las condiciones impuestas simultáneamente.

5.3 Análisis de Resultados

En esta sección, se examinan cómo las distintas características de las soluciones afectan el proceso de resolución y el análisis de problemas de programación lineal.

5.3.1 Solución Única: Eficiencia en la Optimización

Una solución única facilita el análisis y la interpretación de los resultados, ya que proporciona un valor óptimo inequívoco. Este tipo de solución es ideal para aplicaciones donde se requiere una respuesta clara y sin ambigüedades.

Ejemplo: En un problema de optimización de costos, una solución única ofrece el costo mínimo exacto, simplificando la toma de decisiones.

5.3.2 Solución Múltiple: Flexibilidad en la Toma de Decisiones

Las soluciones múltiples permiten al usuario elegir entre varias opciones óptimas. Esto puede ser útil en situaciones donde diferentes soluciones óptimas ofrecen distintos beneficios adicionales, como minimizar costos o maximizar el uso de ciertos recursos.

Ejemplo: En un modelo de producción, la existencia de soluciones múltiples puede permitir a una empresa seleccionar entre diferentes combinaciones de recursos que resultan igualmente óptimas, ofreciendo flexibilidad en la planificación.

5.3.3 Solución Ilimitada: Problemas en la Modelización

Una solución ilimitada generalmente indica que el modelo de PL carece de alguna restricción o que no está adecuadamente formulado. Este tipo de solución debe ser detectado rápidamente para corregir el modelo, pues una solución ilimitada suele ser impráctica en aplicaciones reales.

Ejemplo: Si un modelo de maximización de beneficios no tiene límites en la producción, puede resultar en una solución ilimitada que no es aplicable en un entorno real donde los recursos son finitos.

5.3.4 Problema Sin Solución: Diagnóstico y Ajustes en el Modelo

Cuando un problema de PL no tiene solución, suele ser indicativo de restricciones contradictorias o errores en la formulación. En estos casos, es fundamental revisar las restricciones y condiciones del modelo para identificar y resolver las inconsistencias.

Ejemplo: En una planificación de proyectos, restricciones contradictorias en los tiempos de entrega y disponibilidad de recursos pueden resultar en un modelo sin solución, obligando a revisar los supuestos iniciales.

CAPÍTULO 6: FACTIBILIDAD DE CONSTRUIR UNA APLICACIÓN PARA GENERAR MODELOS DE PL CON CARACTERÍSTICAS ESPECIALES

6.1 Introducción

En el ámbito de la enseñanza de la PL, surge una necesidad cada vez mayor de herramientas que faciliten la generación automatizada de problemas con características específicas. Estas herramientas no solo ayudan a los estudiantes a comprender los distintos tipos de soluciones posibles, sino que también permiten a los profesores personalizar los problemas para abordar conceptos avanzados en PL. En su investigación, Castillo [Castillo01] explora un enfoque matemático riguroso para la generación de problemas de PL, que permite crear modelos con soluciones acotadas, no acotadas, únicas o múltiples, mediante la manipulación estructural de la región factible de los problemas. Sin embargo, replicar este nivel de especificidad plantea varios desafíos, especialmente en un entorno de desarrollo enfocado en herramientas de programación modernas y orientado a fines educativos.

6.2 Desarrollo y Experiencia

6.2.1 Descripción del proceso de desarrollo de la aplicación

La aplicación desarrollada en esta memoria busca proporcionar una herramienta accesible para generar problemas de PL en un entorno educativo. A diferencia del enfoque complejo propuesto por Castillo, en el que la región factible se descompone en un espacio lineal, un cono y un politopo, esta aplicación adopta una metodología simplificada que permite crear problemas de PL, PLE, PLB y PLM con diferentes configuraciones de restricciones y objetivos.

El desarrollo de la aplicación se realizó utilizando Python y la biblioteca OR-Tools, enfocándose en la generación aleatoria de coeficientes para la función objetivo y las restricciones. Este enfoque permite una creación rápida y flexible de problemas, aunque no garantiza el control geométrico de la región factible, como sí lo permite el enfoque de Castillo.

6.2.2 Modelos Matemáticos Propuestos por Castillo

El *paper* de Castillo propone un modelo matemático para definir la región factible de los problemas de PL de una manera estructurada y adaptable. La metodología propuesta define la región factible S como una combinación de tres componentes geométricos:

1. **Espacio Lineal V :** El espacio lineal es un subespacio que permite desplazamientos infinitos en direcciones específicas sin violar las restricciones del sistema. En términos de PL, el espacio lineal permite que ciertas variables tomen valores en intervalos no acotados en ambas direcciones. Matemáticamente, V es un conjunto de vectores que definen direcciones de libertad en la solución, es decir, direcciones

en las que la función objetivo puede cambiar sin salir de la región factible. En términos prácticos, el espacio lineal se define como un conjunto de direcciones de variabilidad. Este componente es importante para generar problemas en los que la solución no esté completamente restringida, permitiendo que la función objetivo pueda cambiar en varias direcciones. Así, si un problema de PL contiene un espacio lineal amplio, es más probable que existan múltiples soluciones óptimas en esa dirección. Matemáticamente, el espacio lineal se define como:

$$V = \left\{ \sum_{i=1}^I n_i v_i \mid n_i \in R, v_i \in V \right\}$$

donde v_i corresponde a los vectores generadores y n_i a sus respectivos coeficientes con números reales. Esto significa que V es el conjunto de todas las combinaciones lineales posibles de los vectores v_i , y su presencia en la región factible indica la parte lineal del espacio donde las restricciones no limitan el crecimiento en una dirección específica.

2. **Cono W :** El cono es el conjunto de direcciones en las que los valores de la función objetivo pueden aumentar o disminuir indefinidamente. Este componente permite representar problemas de PL en los que la solución es no acotada. En programación lineal, el cono W representa todas las combinaciones lineales no negativas de un conjunto de vectores base que mantienen al sistema dentro de los límites de las restricciones. Matemáticamente, el cono se define como:

$$W = \left\{ \sum_{i=1}^k \lambda_i w_i \mid \lambda_i \geq 0, w_i \in W \right\}$$

donde cada w_i es un vector generador del cono, y los coeficientes λ_i son no negativos. Si el vector de costos de la función objetivo apunta en una dirección incluida en el cono, entonces el valor de la función objetivo no está acotado en esa dirección. Esto significa que el problema no tendrá una solución óptima única y acotada, sino que podrá crecer sin límites en esa dirección. La presencia del cono es clave en la generación de problemas no acotados.

3. **Politopo Q :** El politopo es la parte compacta y cerrada de la región factible. A diferencia del espacio lineal y el cono, el politopo contiene únicamente puntos que cumplen con las restricciones sin permitir variación infinita en ninguna dirección. Matemáticamente, el politopo se representa como la combinación convexa de un conjunto de puntos extremos (vértices) que delimitan la región factible. La construcción del politopo es esencial para definir problemas con soluciones acotadas y únicas. Los puntos extremos del politopo representan soluciones potenciales, y el valor de la función objetivo en estos puntos determina si el problema tendrá una solución única o múltiple. Si la función objetivo está alineada con una de las caras del politopo, se pueden obtener múltiples soluciones óptimas;

de lo contrario, la solución será única y estará ubicada en un vértice específico. Matemáticamente, el politopo se define como:

$$Q = \left\{ \sum_{i=1}^k \lambda_i q_i \mid \lambda_i \geq 0, \sum_{i=1}^k \lambda_i = 1, q_k \in Q \right\}$$

Matemáticamente, la región factible S se expresa como la suma de estas tres componentes:

$$S = V + W + Q$$

Además de estos tres componentes, el enfoque de Castillo introduce el **cono dual W^*** . El cono dual es una estructura geométrica en el espacio dual que representa la relación entre las restricciones del problema y la función objetivo. Su propósito es definir el conjunto de vectores de costo para los cuales la función objetivo permanece no decreciente en las direcciones de las restricciones activas. Matemáticamente el cono dual W^* se define como el conjunto de todos los vectores de costo c que cumplen con la siguiente condición:

$$c^T x \geq 0, \quad \forall x \in W$$

Esta definición implica que el vector de costos c está orientado de manera que no reduce el valor de la función objetivo en la dirección de las restricciones activas. El cono dual es crucial para determinar las características de la solución óptima en el espacio dual y para identificar si la función objetivo está orientada de tal manera que permita una solución acotada o no acotada.

En el contexto de la generación de problemas de programación lineal con características específicas, el trabajo de Castillo introduce la **función gamma** como una herramienta clave para construir conos duales. Matemáticamente, el dual de un cono C , representado como C^* se define como el conjunto de todos los vectores y tales que el producto interno con cualquier vector x en C es no positivo:

$$C^* = \{y \in R^n \mid x^T y \leq 0, \quad \forall x \in C\}$$

La función gamma permite una representación simplificada del cono dual, reduciendo los generadores necesarios para definir C^* y optimizando el proceso de generación de problemas.

La función gamma resulta fundamental por varias razones:

- **Optimización en la Representación Dual:** Permite reducir el número de vectores generadores en el dual, lo cual simplifica la representación y cálculo de C^* .
- **Facilitación en la Creación de Soluciones Específicas:** Al controlar los generadores en el cono dual, se pueden construir regiones factibles que se ajusten a problemas de programación lineal con soluciones deseadas, como soluciones únicas, múltiples, acotadas o no acotadas.

Cada componente define una característica específica de la región factible y ayuda a parametrizar problemas de PL que puedan representar tipos de soluciones variados, como acotadas, no acotadas, únicas o múltiples. Castillo también plantea condiciones para definir estos tipos de soluciones.

6.2.3. Tipos de Soluciones Propuestas

6.2.3.1 Modelo de Solución Acotada

Para asegurar que un problema de programación lineal tenga una solución acotada, se debe garantizar que la función objetivo $c^T x$ no pueda extenderse indefinidamente dentro de la región factible S . Esto ocurre si el espacio lineal V y el cono W están configurados de modo que:

$$c^T v = 0 \text{ y } c^T w \leq 0 \quad \forall v \in V \text{ y } w \in W$$

En otras palabras, si el vector de costos c es ortogonal al espacio lineal V y no tiene una componente positiva en el cono W , el valor de la función objetivo estará acotado dentro de la región factible. Esto asegura que exista un límite superior o inferior para $c^T x$, dependiendo de si el problema es de maximización o minimización, y que no se produzcan soluciones no acotadas.

6.2.3.2 Modelo de Solución No Acotada

Para configurar un problema de PL con una solución no acotada, se busca que la función objetivo tenga una dirección en la que pueda crecer indefinidamente dentro de la región factible. Esto se logra cuando existe al menos un vector w en el cono W tal que:

$$c^T w > 0$$

En este caso, el vector de costos c tiene una componente positiva en la dirección de w , lo cual permite que el valor de $c^T x$ aumente sin límite a medida que avanzamos en esa dirección. Este modelo de solución no acotada es común en problemas donde el espacio de búsqueda no tiene una restricción que limite el crecimiento de la función objetivo en ciertas direcciones.

6.2.3.3 Modelo de Solución Única

Para obtener una solución única en un problema de PL, la función objetivo debe alcanzar su valor óptimo en un único punto extremo del politopo Q . Esto ocurre cuando el vector de costos c de la función objetivo no es paralelo a ninguna cara de Q , es decir, solo intersecta un vértice.

Matemáticamente, una solución única se presenta si el valor de la función objetivo en un único punto x^* es óptimo y cumple con las restricciones, de modo que ningún otro punto en Q produce el mismo valor de la función objetivo. Este modelo asegura que el vector de costos c no se alinee con ninguna de las caras del politopo, lo cual restringe el óptimo a un solo vértice. El problema de PL para una solución única puede expresarse de la siguiente forma:

$$\text{Maximizar } c^T x \text{ sujeto a } x \in S$$

donde $S = V + W + Q$, y se configura c para que apunte hacia un vértice único en Q . Al ajustar el vector de costos c de esta manera, el problema de PL tendrá una única solución óptima en el vértice donde $c^T x$ alcanza su valor máximo o mínimo (dependiendo del tipo de optimización, maximización o minimización).

6.2.3.4 Modelo de Soluciones Múltiples

Para obtener múltiples soluciones, es necesario que la función objetivo $c^T x$ sea paralela a una de las caras del politopo Q . Esto permite que varios puntos sobre la misma cara del politopo cumplan con el valor óptimo de la función objetivo, generando así un conjunto de soluciones óptimas en lugar de un único punto.

Matemáticamente, el modelo de soluciones múltiples requiere que c se alinee de tal forma que todos los puntos en una cara del politopo Q resulten en el mismo valor de la función objetivo. Esta configuración es posible cuando el vector de costos está orientado en la misma dirección que una cara de Q . Así, al resolver el problema de PL, se obtendrá una serie de puntos óptimos en lugar de una solución única, ya que cualquier punto en esa cara será igualmente óptimo. Para representar este tipo de problema, Castillo et al. sugieren ajustar c de manera que:

$$c^T x = \text{constante} \quad \forall x \in \text{cara de } Q$$

Este enfoque asegura que múltiples puntos de la cara alineada serán soluciones óptimas, ya que el valor de la función objetivo se mantiene constante en toda esa región de la cara de Q .

6.3 Desafíos y problemas encontrados durante el desarrollo

En el proceso de desarrollo se presentaron varios desafíos al intentar implementar los conceptos matemáticos avanzados del *paper* de Castillo, los cuales incluyen:

1. **Complejidad Matemática en la Definición de la Región Factible:** Generar la región factible como la suma de un espacio lineal, un cono y un politopo es una construcción matemática avanzada que requiere un control preciso sobre las configuraciones geométricas de las restricciones. La implementación simplificada de

esta memoria no incorpora estas estructuras, optando en su lugar por una generación aleatoria de coeficientes que, si bien produce problemas válidos de optimización, no permite un control exhaustivo sobre el tipo de solución (como soluciones únicas, múltiples, acotadas o no acotadas) de la región factible.

2. **Limitaciones de las Herramientas de Optimización:** La biblioteca OR-Tools, empleada en esta memoria, permite la resolución de problemas de optimización, pero no ofrece herramientas para manejar la geometría de la región factible con la precisión requerida para ajustar y manipular estructuras como el cono W , el espacio lineal V , y el politopo Q . Estas herramientas no están diseñadas para controlar explícitamente los componentes geométricos, lo que limitó la capacidad de representar problemas no acotados y parametrizar soluciones únicas y múltiples. En el entorno de OR-Tools, no es factible construir y manipular directamente el cono W ni el cono dual W^* , lo cual obligó a optar por un enfoque simplificado en la generación de problemas.
3. **Parametrización para Tipos de Solución Específicos:** En el modelo de Castillo, la función objetivo y las restricciones se ajustan cuidadosamente para garantizar la alineación (o falta de alineación) del vector de costos c con caras específicas del politopo Q , permitiendo así generar soluciones únicas o múltiples. Implementar esta capacidad de alineación y parametrización en la aplicación habría requerido un control fino de los coeficientes y un ajuste dinámico en la construcción de las restricciones, lo cual excedió el alcance de esta implementación. Como resultado, la aplicación se limitó a la generación aleatoria de coeficientes y no pudo asegurar condiciones específicas en las soluciones generadas.
4. **Complejidad Teórica en la Geometría de Conos:** La función gamma se fundamenta en conceptos avanzados de geometría convexa y teoría de conos, especialmente en la representación de espacios duales y en la minimización de generadores. Estos conceptos requieren un alto nivel de entendimiento en álgebra lineal y teoría de la optimización, lo cual excede el alcance de la preparación matemática en algunos aspectos.
5. **Dominio de Métodos Avanzados de Álgebra Lineal:** El cálculo de un conjunto mínimo de generadores para el dual de un cono implica técnicas algebraicas complejas y abstractas, que no son comúnmente abordadas en los cursos estándar de PL. La falta de familiaridad con estos métodos dificultó la implementación de una versión funcional de la función gamma, ya que incluso entender el proceso de simplificación de generadores y aplicar la teoría era desafiante.
6. **Comprensión de la Interacción entre Espacios Lineales y Conos en el Dual:** La función gamma no solo se basa en cálculos algebraicos, sino también en un entendimiento geométrico profundo de cómo los conos y los espacios lineales se combinan para formar el dual. Esto requiere una visión geométrica intuitiva y

matemática avanzada, que resultó compleja de dominar para el alcance de este proyecto.

7. **Complejidad en la Definición del Cono Dual W^* :** Uno de los mayores desafíos fue la construcción automática del cono dual W^* . Este componente requiere identificar las direcciones de expansión en el espacio primal (definidas por el cono W) y luego traducirlas al espacio dual para determinar el conjunto de vectores de costos que mantendrán el valor de la función objetivo no decreciente. La creación del cono dual implica que el vector de costos c debe alinearse con las restricciones de tal manera que no permita una reducción en el valor de la función objetivo en las direcciones de W . Matemáticamente, esto significa encontrar el conjunto de c que satisfaga $c^T x \geq 0, \forall x \in W$, lo cual no es trivial. Este análisis de orientaciones entre el espacio primal y dual requiere una evaluación geométrica que excede las capacidades de OR-Tools, ya que esta herramienta no permite manipular explícitamente los componentes geométricos para evaluar la relación entre las restricciones y la función objetivo en el espacio dual.

6.4 Soluciones implementadas y aprendizajes obtenidos

Para superar estas dificultades, se optó por un enfoque de diseño que simplificara el proceso de generación. La aplicación genera problemas de PL mediante la creación aleatoria de coeficientes dentro de un rango especificado, tanto para la función objetivo como para las restricciones. Esto permite a los usuarios obtener problemas válidos de PL, PLE, PLB y PLM sin la necesidad de definir estructuras matemáticas complejas. Aunque esta solución no permite replicar completamente el enfoque detallado por Castillo, ofrece un generador práctico y funcional para propósitos educativos.

Los aprendizajes obtenidos de esta experiencia destacan la importancia de ajustar los objetivos de diseño al contexto del desarrollo y a las limitaciones de las herramientas empleadas. En lugar de intentar replicar un modelo altamente complejo, se priorizó la usabilidad y simplicidad de la aplicación, proporcionando una herramienta útil para los usuarios sin la complejidad de manejar estructuras matemáticas avanzadas.

6.5 Evaluación de Factibilidad

La investigación de Castillo demuestra que es posible desarrollar un generador de problemas de PL con características específicas, siempre que se disponga de algoritmos avanzados y herramientas adecuadas. En este trabajo, la aplicación desarrollada se concibió bajo un enfoque más accesible, adaptado al entorno educativo y a la simplificación en el diseño. La factibilidad de construir una aplicación como la desarrollada en esta memoria queda demostrada, aunque con ciertas limitaciones en la generación de tipos de soluciones específicas.

La aplicación permite a los usuarios generar problemas de PL básicos y ajustarlos en función de restricciones y variables. Aunque no ofrece la precisión y control que un generador

matemático más complejo podría proporcionar, cumple con el objetivo de facilitar el aprendizaje de PL mediante la exposición a problemas variados de forma práctica y rápida.

CAPÍTULO 7: CONCLUSIONES

El presente proyecto de memoria ha permitido el desarrollo de una herramienta automatizada para la **Generación de Problemas de Programación Lineal**, diseñada específicamente para apoyar la enseñanza y el aprendizaje en la asignatura de Optimización en la Universidad Técnica Federico Santa María. La creación de esta aplicación responde a la necesidad de disponer de ejercicios prácticos en el aula que permitan a los estudiantes experimentar con distintos tipos de problemas de optimización, proporcionando una alternativa a la generación manual, que es laboriosa y tiende a limitar la diversidad de problemas planteados. Mediante esta herramienta, se espera facilitar el aprendizaje práctico y fortalecer la comprensión de los fundamentos de la programación lineal, que resulta esencial en áreas como la ingeniería, la economía y la investigación operativa.

A lo largo del desarrollo, se logró implementar una aplicación flexible, capaz de generar problemas de distintos tipos: programación lineal (PL), programación lineal entera (PLE), programación lineal binaria (PLB) y programación lineal mixta (PLM). Estos tipos de problemas permiten cubrir una amplia gama de aplicaciones prácticas y académicas, brindando a los estudiantes la posibilidad de enfrentar desafíos adaptados a sus niveles de conocimiento y complejidad. La herramienta permite configurar parámetros fundamentales como el tipo de optimización (maximización o minimización), el número de variables y restricciones, y la naturaleza de las variables, ofreciendo una flexibilidad considerable. Para su desarrollo, se utilizó Python y bibliotecas como OR-Tools para la optimización, lo que habilita la integración con *solvers* reconocidos en la industria, como GLOP para problemas continuos y SCIP para problemas con restricciones enteras y binarias.

Uno de los aspectos clave del proyecto ha sido la capacidad de la herramienta para exportar los problemas generados en formatos estándar, como MPS, lp_solve y LINDO. Esta característica facilita que los problemas puedan ser utilizados en otros entornos de optimización, lo cual es fundamental tanto en el ámbito académico como profesional. La exportación asegura la interoperabilidad con plataformas de optimización ampliamente utilizadas, proporcionando un valor adicional que permite que la herramienta no se limite a un único entorno de uso, sino que se integre en flujos de trabajo y plataformas externas.

Durante el desarrollo y validación de la herramienta, se identificaron ciertos aspectos que limitan su funcionalidad actual. A pesar de haberse propuesto el testeo con estudiantes para evaluar la efectividad pedagógica de la aplicación, este objetivo no se logró cumplir. La ausencia de esta etapa de validación con usuarios finales impide contar con retroalimentación directa sobre la facilidad de uso de la herramienta y su impacto en el aprendizaje de los estudiantes. Este es un aspecto crítico para abordar en el futuro, ya que permitiría obtener información valiosa sobre cómo interactúan los estudiantes con la herramienta y cómo ésta puede optimizarse para mejorar la experiencia educativa. Las pruebas con estudiantes podrían ofrecer *insights* importantes para ajustar la interfaz, mejorar la claridad de los mensajes y asegurar que los ejercicios generados cumplan con los objetivos pedagógicos de la asignatura.

Adicionalmente, las pruebas internas mostraron que, si bien la generación de problemas es rápida y eficiente incluso para problemas de tamaño considerable, los tiempos de resolución pueden variar significativamente en función del *solver* y de la complejidad del problema. En problemas de programación lineal binaria de gran tamaño, por ejemplo, los tiempos de procesamiento aumentan considerablemente, lo que sugiere la necesidad de advertir a los usuarios sobre posibles demoras al trabajar con problemas de alta complejidad. Este hallazgo resalta la importancia de optimizar la herramienta o incorporar algoritmos que mejoren la eficiencia en problemas de gran escala.

La sección de factibilidad del proyecto también reveló que, si bien la herramienta satisface las necesidades educativas actuales de la asignatura de Optimización, existen limitaciones inherentes en cuanto a la posibilidad de generar modelos de extrema complejidad, lo cual podría ser necesario en aplicaciones avanzadas. Sin embargo, se priorizó un diseño simplificado que mantuviera la herramienta accesible para estudiantes sin experiencia técnica avanzada en programación o matemáticas complejas, asegurando así que cumpla con su propósito educativo de manera efectiva.

Con base en las experiencias y hallazgos obtenidos a lo largo del desarrollo de la herramienta, se identifican una serie de mejoras futuras que podrían enriquecer su funcionalidad y ampliar su aplicabilidad tanto en el ámbito académico como en el profesional:

1. **Visualización interactiva de la región factible:** Una mejora significativa sería la inclusión de una función gráfica interactiva que permita a los usuarios visualizar la región factible de problemas de programación lineal con dos variables. Esta funcionalidad ayudaría a los estudiantes a comprender mejor los conceptos geométricos de la optimización y a visualizar el impacto de las restricciones en la solución del problema, facilitando un aprendizaje más visual y profundo de los conceptos teóricos de la programación lineal.
2. **Incorporación de algoritmos adicionales de optimización:** Actualmente, la herramienta utiliza GLOP y SCIP como *solvers* principales. Ampliar la compatibilidad para incluir otros algoritmos, como los métodos de puntos interiores o técnicas heurísticas, podría mejorar la capacidad de resolución para problemas de alta complejidad y permitiría a los usuarios explorar métodos de optimización avanzados. Esto no solo enriquecería el aprendizaje, sino que también ampliaría el rango de problemas que pueden abordarse con la herramienta.
3. **Configuración de características específicas en los problemas generados:** Permitir que los usuarios ajusten propiedades avanzadas de los problemas generados, tales como la cantidad de soluciones factibles o la presencia de soluciones acotadas y no acotadas, enriquecería la experiencia educativa al permitir que los estudiantes exploren escenarios de optimización más diversos. Esto permitiría que los problemas generados reflejen situaciones reales de manera más precisa, aumentando el valor práctico de la herramienta.

4. **Mejoras en la interfaz de usuario y experiencia de uso:** Aunque la herramienta cuenta con una interfaz intuitiva, podrían implementarse mejoras adicionales, como una guía interactiva para los usuarios o recomendaciones en tiempo real sobre la configuración del tamaño del problema en función del rendimiento esperado. Estas mejoras contribuirían a que la herramienta fuera aún más accesible para usuarios con distintos niveles de habilidad, facilitando una experiencia de uso más clara y fluida.
5. **Incorporación de conceptos del *paper* de Castillo:** Para futuras implementaciones, se recomienda considerar los conceptos y definiciones presentados en el *paper* de Castillo, que ofrece un enfoque teórico robusto para el modelado de problemas de programación lineal. Integrar estos conceptos permitirá mejorar la precisión y profundidad de los modelos generados, lo cual podría ser beneficioso para aplicaciones más avanzadas y específicas, ampliando las capacidades de la herramienta en escenarios complejos.

En conclusión, el desarrollo de esta herramienta representa un aporte significativo para la enseñanza de la programación lineal en un contexto académico. La automatización en la generación de problemas proporciona a docentes y estudiantes un recurso innovador que facilita tanto la enseñanza como el aprendizaje de esta disciplina. La flexibilidad de configuración y la posibilidad de exportar los problemas a formatos reconocidos amplían las oportunidades de uso, tanto en el ámbito educativo como en aplicaciones prácticas de optimización. Aunque el proyecto ha cumplido con la mayoría de sus objetivos, incluyendo la validación técnica de la herramienta, la implementación de futuras mejoras y la realización de pruebas de usabilidad con estudiantes son pasos importantes para maximizar su potencial educativo y posicionarla como un recurso integral y robusto en la enseñanza de la optimización.

La incorporación de las mejoras propuestas y la validación directa en un entorno educativo permitirán que la herramienta evolucione hacia un sistema más avanzado y completo, alineado con los requerimientos actuales de la educación en ingeniería y optimización. Con ello, se espera que esta aplicación pueda contribuir al aprendizaje de programación lineal y a la formación de competencias esenciales en la toma de decisiones y la resolución de problemas en áreas clave de la industria y la investigación.

REFERENCIAS BIBLIOGRÁFICAS

[Álvarez21] Álvarez, M. (20 de agosto del 2021). Efecto pandemia: Un 62,7% de docentes reveló sentir agobio. Lugar de publicación: Diario Concepción. Recuperado de <https://www.diarioconcepcion.cl/ciudad/2021/08/20/efecto-de-la-pandemia-un-627-de-docentes-revelo-sentir-agobio.html>

[Castillo01] Castillo, E., Pruneda, R., Esquivel, M., (2001) Automatic generation of linear programming problems for computer aided instruction, International Journal of Mathematical Education in Science and Technology.

[Castillo02] Castillo, E., Conejo, A., Pedregal, P., García, R., Alguacil, N. (2022). Formulación y resolución de modelos de programación matemática en ingeniería y ciencia. Castilla-La Mancha, España. Universidad De Castilla-La Mancha.

[Chvátal83] Chvátal, V. (1983). Linear Programming. W. H. Freeman and Company.

[Dantzig51] Dantzig, G. B. (1951). Application of the Simplex Method to a Transportation Problem. En T. C. Koopmans (Ed.), Activity Analysis of Production and Allocation. Wiley.

[GGLOP24] Google Developers (2024). Resolviendo un problema de programación lineal (PL) usando programación matemática. Recuperado de https://developers.google.com/optimization/lp/lp_example?hl=es-419

[GSCIP24] Google Developers (2024). Resolviendo un problema de programación lineal entera mixta (PLEM). Recuperado de https://developers.google.com/optimization/mip/mip_example?hl=es-419

[Goic08] Goic, M. (2008). Análisis Post Optimal y Algoritmo de Ramificación y Acotamiento. Lugar de publicación: Universidad de Chile. Recuperado de https://www.u-cursos.cl/ingenieria/2008/2/IN34A/1/material_docente/bajar?id=198947

[Martínez21] Martínez, C. (27 de agosto del 2021). El 62,7% de los docentes declaró sentir agobio a la hora de enseñar. Lugar de publicación: Infogate. Recuperado de <https://www.infogate.cl/2021/08/27/el-627-de-los-docentes-declaro-sentir-agobio-a-la-hora-de-ensenar/>

[Numpy22] Numpy Developers (2022). Numpy documentation. Recuperado de <https://numpy.org/doc/1.24/>

[Roldán19] Roldán, P. N. (3 de enero del 2019). Modelo matemático. Lugar de publicación: Economipedia. Recuperado de <https://economipedia.com/definiciones/modelo-matematico.html>

[Papalambros00] Papalambros, P. Y., & Wilde, D. J. (2000). Principles of Optimal Design: Modeling and Computation. Cambridge University Press.

[Python24] Python Software Foundation. (2024). What is Python? Executive Summary. Recuperado de <https://www.python.org/doc/essays/blurb/>

[SeguimosVirtual20] Seguimos Virtual (15 de junio del 2020). Resultados investigación #PulsoEstudiantil. Lugar de publicación: Seguimos Virtual. Recuperado de <https://www.dii.uchile.cl/2020/06/15/resultados-investigacion-pulso-estudiantil/>

[Schrijver98] Schrijver, A. (1998). Theory of Linear and Integer Programming. John Wiley & Sons.

[Taha12] Taha, H. (2012). Investigación de Operaciones.

[Tkinter24] Python Software Foundation. (2024). tkinter — Python interface to Tcl/Tk. Recuperado de <https://docs.python.org/3/library/tk.html>

[Testuri12] Testuri, C., Islas, F. (2012) Fundamentos de Programación Entera. Lugar de publicación: Departamento de Investigación Operativa, Universidad de la República. Recuperado de https://eva.fing.edu.uy/pluginfile.php/76324/mod_resource/content/8/fpe-6-plano-corte.pdf

[Vanderbei13] Vanderbei, R. J. (2013). Linear Programming: Foundations and Extensions. Springer.

[Williams13] Williams, H. P. (2013). Model Building in Mathematical Programming. John Wiley & Sons.

[Wolsey98] Wolsey, L. A. (1998). Integer Programming. John Wiley & Sons.

ANEXOS

Prueba 1: Generación de PL

Configuración General			Rango de Coeficientes			Cantidad de Restricciones		
Tipo de Optimización	max		Rango Mínimo de Coeficientes (Objetivo)	-3		Cantidad de Restricciones ≤	1	
Número de Variables Enteras	0		Rango Máximo de Coeficientes (Objetivo)	3		Cantidad de Restricciones ≥	1	
Número de Variables Continuas	3		Rango Mínimo de Coeficientes (Restricciones)	-4		Cantidad de Restricciones =	1	
Número de Variables Binarias	0		Rango Máximo de Coeficientes (Restricciones)	4				
☒ Coeficientes Enteros			Rango Mínimo de Coeficientes (Lado Derecho)	-5				
Formato de Exportación			Rango Máximo de Coeficientes (Lado Derecho)	5				

Problema de Programación Lineal

Función Objetivo:
 $\max z = 0.0x_1 + -3.0x_2 + 0.0x_3$

Restricciones:
 $-3.0x_1 + 3.0x_2 + -2.0x_3 \geq 4.0$
 $-2.0x_1 + -3.0x_2 + 1.0x_3 \leq 1.0$
 $1.0x_1 + 2.0x_2 + -2.0x_3 = 4.0$

Naturaleza de las Variables:
 $x_1 \geq 0$
 $x_2 \geq 0$
 $x_3 \geq 0$

Valor variables: {'x1': 0.4444444444444445, 'x2': 1.7777777777777778, 'x3': 0.0}
 Valor función objetivo: -5.333333333333333

Prueba 2: Generación de PLE

Configuración General			Rango de Coeficientes			Cantidad de Restricciones		
Tipo de Optimización	max		Rango Mínimo de Coeficientes (Objetivo)	-3		Cantidad de Restricciones ≤	1	
Número de Variables Enteras	3		Rango Máximo de Coeficientes (Objetivo)	3		Cantidad de Restricciones ≥	1	
Número de Variables Continuas	0		Rango Mínimo de Coeficientes (Restricciones)	-4		Cantidad de Restricciones =	1	
Número de Variables Binarias	0		Rango Máximo de Coeficientes (Restricciones)	4				
☒ Coeficientes Enteros			Rango Mínimo de Coeficientes (Lado Derecho)	-5				
Formato de Exportación			Rango Máximo de Coeficientes (Lado Derecho)	5				

Problema de Programación Lineal Entera

Función Objetivo:
 $\max z = -1.0x_1 + -2.0x_2 + -3.0x_3$

Restricciones:
 $3.0x_1 + 3.0x_2 + -4.0x_3 \geq 2.0$
 $3.0x_1 + 0.0x_2 + -1.0x_3 \leq 0.0$
 $-3.0x_1 + -3.0x_2 + 2.0x_3 = -4.0$

Naturaleza de las Variables:
 $x_1 \in \mathbb{Z}$
 $x_2 \in \mathbb{Z}$
 $x_3 \in \mathbb{Z}$

Valor variables: {'x1': 0.0, 'x2': 2.0, 'x3': 1.0}
 Valor función objetivo: -7.0

Prueba 3: Generación de PLB

Configuración General			Rango de Coeficientes			Cantidad de Restricciones		
Tipo de Optimización	max		Rango Mínimo de Coeficientes (Objetivo)	-3		Cantidad de Restricciones ≤	1	
Número de Variables Enteras	0		Rango Máximo de Coeficientes (Objetivo)	3		Cantidad de Restricciones ≥	1	
Número de Variables Continuas	0		Rango Mínimo de Coeficientes (Restricciones)	-4		Cantidad de Restricciones =	1	
Número de Variables Binarias	3		Rango Máximo de Coeficientes (Restricciones)	4				
☒ Coeficientes Enteros			Rango Mínimo de Coeficientes (Lado Derecho)	-5				
Formato de Exportación			Rango Máximo de Coeficientes (Lado Derecho)	5				

Problema de Programación Lineal Binaria

Función Objetivo:
 $\max z = -2.0x_1 + -1.0x_2 + 2.0x_3$

Restricciones:
 $-4.0x_1 + -3.0x_2 + 2.0x_3 \geq -4.0$
 $-3.0x_1 + -1.0x_2 + -0.0x_3 \leq 4.0$
 $-3.0x_1 + -1.0x_2 + 1.0x_3 = 1.0$

Naturaleza de las Variables:
 $x_1 \in \{0, 1\}$
 $x_2 \in \{0, 1\}$
 $x_3 \in \{0, 1\}$

Valor variables: {'x1': 0.0, 'x2': 0.0, 'x3': 1.0}
 Valor función objetivo: 2.0

Prueba 4: Generación de PLM

Generador de Problemas de Programación Lineal

Configuración General

Tipo de Optimización:

Número de Variables Enteras:

Número de Variables Continuas:

Número de Variables Binarias:

☒ Coeficientes Enteros

Formato de Exportación:

Rango de Coeficientes

Rango Mínimo de Coeficientes (Objetivo):

Rango Máximo de Coeficientes (Objetivo):

Rango Mínimo de Coeficientes (Restricciones):

Rango Máximo de Coeficientes (Restricciones):

Rango Mínimo de Coeficientes (Lado Derecho):

Rango Máximo de Coeficientes (Lado Derecho):

Cantidad de Restricciones

Cantidad de Restricciones ≤:

Cantidad de Restricciones ≥:

Cantidad de Restricciones =:

Generar Problema

Exportar Problema

Limpiar

Problema de Programación Lineal Mixta

Función Objetivo:
 $\max z = -1.0x_1 + -2.0x_2 + -2.0x_3$

Restricciones:
 $-4.0x_1 + 3.0x_2 + -3.0x_3 \leq 1.0$
 $0.0x_1 + 4.0x_2 + 2.0x_3 \geq -1.0$
 $-3.0x_1 + -1.0x_2 + -3.0x_3 = -4.0$

Naturaleza de las Variables:
 $x_1 \in \mathbb{Z}$
 $x_2 \in \mathbb{R}$
 $x_3 \in \{0, 1\}$

Valor variables: ('x1': 1.0, 'x2': 1.0, 'x3': 0.0)
 Valor función objetivo: -3.0

Prueba de Exportación de MPS

File Edit Search Action View Options Help			
Source Matrix Options Result			
1	NAME	PROBLEM	
2	ROWS		
3	N OBJ		
4	E R1		
5	G R2		
6	L R3		
7	COLUMNS		
8	x1	OBJ	1.0000
9	x1	R1	2.0000
10	x1	R2	-3.0000
11	x1	R3	3.0000
12	x2	OBJ	-2.0000
13	x2	R1	3.0000
14	x2	R2	-3.0000
15	x2	R3	-2.0000
16	x3	OBJ	-0.0000
17	x3	R1	-3.0000
18	x3	R2	-2.0000
19	x3	R3	1.0000
20	RHS		
21	RHS1	R1	1.0000
22	RHS1	R2	1.0000
23	RHS1	R3	2.0000
24	BOUNDS		
25	ENDATA		

Resultado de Prueba de Exportación de MPS

```

C:\Program Files (x86)\LPSolv x + v
Model name: 'PROBLEM' - run #1
Objective: Minimize(OBJ)

SUBMITTED
Model size:      3 constraints,      3 variables,      9 non-zeros.
Sets:           0 GUB,              0 SOS.

Using DUAL simplex for phase 1 and PRIMAL simplex for phase 2.
The primal and dual simplex pricing strategy set to 'Devex'.

The model is INFEASIBLE
lp_solve unsuccessful after 2 iter and a last best value of 1e+030

MEMO: lp_solve version 5.5.2.11 for 32 bit OS, with 64 bit REAL variables.
In the total iteration count 2, 0 (0.0%) were bound flips.
There were 1 refactorizations, 0 triggered by time and 0 by density.
... on average 2.0 major pivots per refactorization.
The largest [LUSOL v2.2.1.0] fact(B) had 4 NZ entries, 1.0x largest basis.
The constraint matrix inf-norm is 3, with a dynamic range of 3.
Time to load data was 0.002 seconds, presolve used 0.002 seconds,
... 0.007 seconds in simplex solver, in total 0.011 seconds.
    
```

Prueba de Exportación de lp_solve

LPSolve IDE - 5.5.2.11 - C:\Users\naxo\Desktop\Ejemplos Solvers Memo

File Edit Search Action View Options Help

Source Matrix Options Result

```

1 max: -2.0x1 + -2.0x2 + -1.0x3;
2 -1.0x1 + -2.0x2 + 2.0x3 >= -2.0;
3 -2.0x1 + -2.0x2 + -1.0x3 = -2.0;
4 -1.0x1 + -3.0x2 + 2.0x3 <= -3.0;
5 int x1;
6 int x2;
7 bin x3;
8

```

Resultado de Prueba de Exportación de lp_solve

C:\Program Files (x86)\LPsolv x +

Model name: 'LPSolver' - run #1
Objective: Maximize(R0)

SUBMITTED
Model size: 3 constraints, 3 variables, 9 non-zeros.
Sets: 0 GUB, 0 SOS.

Using DUAL simplex for phase 1 and PRIMAL simplex for phase 2.
The primal and dual simplex pricing strategy set to 'Devex'.

Relaxed solution -2 after 2 iter is B&B base.
Feasible solution -2 after 2 iter, 0 nodes (gap 0.0%)
Optimal solution -2 after 2 iter, 0 nodes (gap 0.0%).

Relative numeric accuracy ||*|| = 2.22045e-016

MEMO: lp_solve version 5.5.2.11 for 32 bit OS, with 64 bit REAL variables.
In the total iteration count 2, 0 (0.0%) were bound flips.
There were 0 refactorizations, 0 triggered by time and 0 by density.
... on average 2.0 major pivots per refactorization.
The largest [LUSOL v2.2.1.0] fact(B) had 4 NZ entries, 1.0x largest basis.
The maximum B&B level was 1, 0.2x MIP order, 1 at the optimal solution.
The constraint matrix inf-norm is 3, with a dynamic range of 3.
Time to load data was 0.002 seconds, presolve used 0.002 seconds,
... 0.007 seconds in simplex solver, in total 0.011 seconds.

Prueba de Exportación de LINDO

Lingo 20.0 - Lingo Model - ejemplo_lindo_con_solución

File Edit Solver Window Help

Lingo Model - ejemplo_lindo_con_solución

```

max 0.0x1 + -2.0x2 + -1.0x3
st
1.0x1 + -1.0x2 + 2.0x3 <= 2.0
-2.0x1 + -2.0x2 + -1.0x3 = -1.0
3.0x1 + 1.0x2 + 2.0x3 >= -1.0
end
free x2
gin x1
int x3

```

Resultado de Prueba de Exportación de LINDO

Lingo 20.0 - [Solution Report - ejemplo_lindo_con_solución]

File Edit Solver Window Help

LINGO/WIN64 20.0.21 (23 Aug 2023), LINDO API 14.0.5099.238

Licensee info: Eval Use Only
License expires: 14 FEB 2025

Global optimal solution found.
Objective value: 1.000000
Objective bound: 1.000000
Infeasibilities: 0.000000
Extended solver steps: 0
Total solver iterations: 0
Elapsed runtime seconds: 0.04

Model Class: MILP

Total variables: 3
Nonlinear variables: 0
Integer variables: 2

Total constraints: 4
Nonlinear constraints: 0

Total nonzeros: 11
Nonlinear nonzeros: 0

Variable	Value	Reduced Cost
X2	-0.5000000	0.000000
X3	0.000000	0.000000
X1	1.000000	-2.000000

Row	Slack or Surplus	Dual Price
1	1.000000	1.000000
2	0.500000	0.000000
3	0.000000	1.000000
4	3.500000	0.000000

Lingo 20.0 Solver Status [ejemplo_lindo_con_solución]

Solver Status: MILP
Model Class: MILP
State: Global Opt
Objective: 1
Infeasibility: 0
Iterations: 0

Variables: Total: 3, Nonlinear: 0, Integers: 2

Constraints: Total: 4, Nonlinear: 0

Nonzeros: Total: 11, Nonlinear: 0

Generator Memory Used (K): 23

Elapsed Runtime (h:mm:ss): 00:00:00

Extended Solver Status: Solver Type: B-and-B
Best Obj: 1
Obj Bound: 1
Steps: 0
Active: 0

Update Interval: 2

Interrupt Solver Close

Tiempo de ejecución PL pequeño

```
PS C:\Users\naxo_OneDrive\Escritorio\Cosas de la U\T2\LP-Generator> & C:/Users/naxo/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/naxo_OneDrive/Escritorio/Cosas de la U\T2\LP-Generator/main.py"
Tiempo de generación del modelo: 0.0 segundos
Tiempo de resolución del modelo: 0.01347494125366211 segundos
```

Tiempo de ejecución PLE pequeño

```
PS C:\Users\naxo_OneDrive\Escritorio\Cosas de la U\T2\LP-Generator> & C:/Users/naxo/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/naxo_OneDrive/Escritorio/Cosas de la U\T2\LP-Generator/main.py"
Tiempo de generación del modelo: 0.0 segundos
Tiempo de resolución del modelo: 0.005003929138183594 segundos
```

Tiempo de ejecución PLB pequeño

```
PS C:\Users\naxo_OneDrive\Escritorio\Cosas de la U\T2\LP-Generator> & C:/Users/naxo/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/naxo_OneDrive/Escritorio/Cosas de la U\T2\LP-Generator/main.py"
Tiempo de generación del modelo: 0.0 segundos
Tiempo de resolución del modelo: 0.005002260208129883 segundos
```

Tiempo de ejecución PLM pequeño

```
PS C:\Users\naxo_OneDrive\Escritorio\Cosas de la U\T2\LP-Generator> & C:/Users/naxo/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/naxo_OneDrive/Escritorio/Cosas de la U\T2\LP-Generator/main.py"
Tiempo de generación del modelo: 0.0 segundos
Tiempo de resolución del modelo: 0.003515481948852539 segundos
```

Tiempo de ejecución PL mediano

```
PS C:\Users\naxo_OneDrive\Escritorio\Cosas de la U\T2\LP-Generator> & C:/Users/naxo/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/naxo_OneDrive/Escritorio/Cosas de la U\T2\LP-Generator/main.py"
Tiempo de generación del modelo: 0.00099945068359375 segundos
Tiempo de resolución del modelo: 0.0075452327728271484 segundos
```

Tiempo de ejecución PLE mediano

```
PS C:\Users\naxo_OneDrive\Escritorio\Cosas de la U\T2\LP-Generator> & C:/Users/naxo/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/naxo_OneDrive/Escritorio/Cosas de la U\T2\LP-Generator/main.py"
Tiempo de generación del modelo: 0.0 segundos
Tiempo de resolución del modelo: 0.010548830032348633 segundos
```

Tiempo de ejecución PLB mediano

```
PS C:\Users\naxo_OneDrive\Escritorio\Cosas de la U\T2\LP-Generator> & C:/Users/naxo/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/naxo_OneDrive/Escritorio/Cosas de la U\T2\LP-Generator/main.py"
Tiempo de generación del modelo: 0.0009996891021728516 segundos
Tiempo de resolución del modelo: 0.011161327362060547 segundos
```

Tiempo de ejecución PLM mediano

```
PS C:\Users\naxo_OneDrive\Escritorio\Cosas de la U\TT2\LP-Generator> & C:/Users/naxo_/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/naxo_/OneDrive/Escritorio/Cosas de la U/TT2/LP-Generator/main.py"
Tiempo de generación del modelo: 0.0 segundos
Tiempo de resolución del modelo: 0.01116037368774414 segundos
```

Tiempo de ejecución PL grande

```
PS C:\Users\naxo_OneDrive\Escritorio\Cosas de la U\TT2\LP-Generator> & C:/Users/naxo_/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/naxo_/OneDrive/Escritorio/Cosas de la U/TT2/LP-Generator/main.py"
Tiempo de generación del modelo: 0.0005044937133789062 segundos
Tiempo de resolución del modelo: 0.004004955291748047 segundos
```

Tiempo de ejecución PLE grande

```
PS C:\Users\naxo_OneDrive\Escritorio\Cosas de la U\TT2\LP-Generator> & C:/Users/naxo_/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/naxo_/OneDrive/Escritorio/Cosas de la U/TT2/LP-Generator/main.py"
Tiempo de generación del modelo: 0.0010042190551757812 segundos
Tiempo de resolución del modelo: 0.01454305648803711 segundos
```

Tiempo de ejecución PLB grande

```
PS C:\Users\naxo_OneDrive\Escritorio\Cosas de la U\TT2\LP-Generator> & C:/Users/naxo_/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/naxo_/OneDrive/Escritorio/Cosas de la U/TT2/LP-Generator/main.py"
Tiempo de generación del modelo: 0.000993967056274414 segundos
Tiempo de resolución del modelo: 0.09140777587890625 segundos
```

Tiempo de ejecución PLM grande

```
PS C:\Users\naxo_OneDrive\Escritorio\Cosas de la U\TT2\LP-Generator> & C:/Users/naxo_/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/naxo_/OneDrive/Escritorio/Cosas de la U/TT2/LP-Generator/main.py"
Tiempo de generación del modelo: 0.0 segundos
Tiempo de resolución del modelo: 0.008555173873901367 segundos
```

Tiempo de ejecución PL muy grande

```
PS C:\Users\naxo_OneDrive\Escritorio\Cosas de la U\TT2\LP-Generator> & C:/Users/naxo_/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/naxo_/OneDrive/Escritorio/Cosas de la U/TT2/LP-Generator/main.py"
Tiempo de generación del modelo: 0.0015254020690917969 segundos
Tiempo de resolución del modelo: 0.010126113891601562 segundos
```

Tiempo de ejecución PLE muy grande

```
PS C:\Users\naxo_OneDrive\Escritorio\Cosas de la U\TT2\LP-Generator> & C:/Users/naxo_/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/naxo_/OneDrive/Escritorio/Cosas de la U/TT2/LP-Generator/main.py"
Tiempo de generación del modelo: 0.0010058879852294922 segundos
Tiempo de resolución del modelo: 0.012511730194091797 segundos
```

Tiempo de ejecución PLB muy grande

```
PS C:\Users\naxo_OneDrive\Escritorio\Cosas de la U\TT2\LP-Generator> & C:/Users/naxo_/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/naxo_/OneDrive/Escritorio/Cosas de la U/TT2/LP-Generator/main.py"
Tiempo de generación del modelo: 0.00099945068359375 segundos
Tiempo de resolución del modelo: 1.657285451889038 segundos
```

Tiempo de ejecución PLM muy grande

```
PS C:\Users\naxo_OneDrive\Escritorio\Cosas de la U\TT2\LP-Generator> & C:/Users/naxo_/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/naxo_/OneDrive/Escritorio/Cosas de la U/TT2/LP-Generator/main.py"
Tiempo de generación del modelo: 0.001003265380859375 segundos
Tiempo de resolución del modelo: 0.02553272247314453 segundos
```