Agentes de Razonamiento para la Formulación de Problemas de Investigación Operativa

Lucas Argento
Facultad de Ingeniería - Universidad de Buenos Aires (UBA)
largento@fi.uba.ar

Octubre 2025

Resumen

Este trabajo analiza el desempeño de una arquitectura de agentes de inteligencia artificial, impulsada por grandes modelos del lenguaje (LLMs, por sus siglas en inglés) con capacidad de razonamiento, sobre ejemplos validados de los *datasets* de Investigación Operativa ***Text2Zinc*** [1] y ***NLP4LP*** [2]. Se consideraron tanto problemas de satisfacción como de optimización para las evaluaciones.

Se presenta la solución de alrededor de 120 problemas provenientes de los conjuntos de datos mencionados (con descripciones de los problemas, implementaciones de código y soluciones).

A partir de este trabajo, se apunta a contribuir con las comunidades de Investigación Operativa (IO) e Inteligencai Artificial (IA) mediante un analisis del desempeño en IO de los enfoques de IA de última generación, así como con la creación de nuevos puntos de datos de entrenamiento para futuros esfuerzos de *fine-tuning* [3] en IO.

1. Introducción

Muchos problemas de decisión del mundo real —desde logística y cadenas de suministro hasta finanzas y planificación— pueden ser formulados y optimizados mediante la aplicación de modelos y algoritmos de IO [4, 5, 6, 7, 8, 9, 10]. Esto implica que este campo de estudio ha demostrado tener el potencial de generar una enorme cantidad de valor y bienestar para nuestra sociedad, en caso de ser aplicado correctamente y a gran escala.

Sin embargo, la IO sigue estando alejada del día a día de muchas industrias, ya que traducir una descripción en lenguaje natural a un modelo matemático formal requiere habilidades especializadas y muy específicas (conocimiento de la industria, capacidades de programación y conocimientos matemáticos avanzados). Naturalmente, esta barrera de conocimiento limita el uso de la optimización: los no expertos tienen dificultades para interactuar con los solucionadores de IO, incluso si una solución óptima podría mejorar significativamente sus operaciones, desperdiciandose así un gran potencial de ahorro.

Los avances recientes en el campo de la IA, particularmente en los modelos de lenguaje de gran escala (LLMs), ofrecen una vía prometedora para cerrar esta brecha. Los LLMs preentrenados demuestran capacidades muy sólidas de comprensión y generación de lenguaje natural, y la proliferación de asistentes de codificación con IA ya ha demostrado que estos modelos pueden aplicarse con éxito a la tarea de generación de código. *El código es la forma en que los problemas de optimización modernos se formulan y resuelven.*

Aunque esfuerzos recientes han intentado crear conjuntos de datos de IO lo suficientemente grandes como para evaluar y potencialmente hacer fine-tuning de LLMs, la disponibilidad de problemas de IO del mundo real (con descripción del problema, código y resultados) sigue siendo muy limitada.

Dos de los conjuntos de datos de IO de código abierto más grandes y diversos disponibles hasta la fecha son el conjunto de datos ***Text2Zinc*** [1], que comprende alrededor de 110 problemas de IO validados por expertos (MILP, CP, Scheduling), y el ***NLP4LP*** [2], que se enfoca en Programación Lineal. Ambos conjuntos de datos incluyen, para cada problema, la descripción del problema, hojas de datos, formulación en código y resultados óptimos. El código para los modelos del conjunto de datos Text2Zinc está escrito en el lenguaje de optimización MiniZinc, mientras que el código del conjunto NLP4LP está escrito en Python.

En las próximas secciones presentamos un sistema agentico, impulsado por modelos fundacionales con capacidad de razonamiento, y los resultados obtenidos al evaluar este sistema sobre los conjuntos de datos mencionados. El lenguaje elegido para la generación de código fue Python, y la biblioteca seleccionada fue OR-Tools [1]. Como subproducto de la evaluación del agente de razonamiento sobre el conjunto de datos mencionado, se generaron soluciones en Python para cada uno de los ejemplos validados. Se publica junto con este trabajo el código correspondiente a cada uno de los puntos de datos del conjunto, con el objetivo de que los mismos sean

[1] https://developers.google.com/optimization?hl=es-419

útiles para futuros esfuerzos de *fine-tuning* o como parte de nuevos conjuntos de datos de IO.

2. Trabajos Relacionados

En esta sección se listaran algunos de los esfuerzos orientados a mejorar la intersección entre los campos de la IA y la IO que han surgido principalmente luego del lanzamiento de ChatGPT a fines de 2022. Entre éstos trabajos de investigación, podemos distinguir tres verticales principales relevantes en este espacio:

- **Creación de Conjuntos de Datos:** Creación de conjuntos de datos que de Lenguaje Natural a Formulación de Problemas de Optimización. La competencia NL4Opt (NeurIPS 2022) [11] introdujo un conjunto de datos de referencia para traducir lenguaje natural a formulaciones de Programación Lineal. Además, otros esfuerzos como Mamo: a Mathematical Modeling Benchmark with Solvers [14, 15], IndustryOR [16], y NLP4LP [2] también han contribuido al avance en la disponibilidad de conjuntos de datos abiertos de IO.

- **Enfoques Agénticos para la Formulación de IO:** Estudios muy recientes aprovechan el poder de los LLMs, algunos incluyendo arquitecturas y flujos de trabajo agénticos, para mejorar aún más el rendimiento del modelo. [2] presenta un agente basado en LLM que formula y resuelve problemas de PL enteros mixtos a partir de lenguaje natural. *OptLLM* [12] propone un agente LLM (ajustado con *fine-tuning*) con *solvers* externos y soporta diálogos multi-turno para la modelación de problemas de optimización. *OR-LLM-Agent* [17] propone un sistema agéntico que utiliza LLMs con capacidad de razonamiento para formular y resolver problemas de IO.

- **Fine-tuning de LLMs en tareas de Formulación de IO:** El fine-tuning es el proceso de (post)entrenamiento de un modelo fundacional (como Llama, Qwen, etc.), para mejorar su rendimiento en un dominio específico (como la formulación de problemas de IO a partir de descripciones de problemas).

  El equipo de *OptLLM*, ya mencionado, creó un conjunto de datos de aproximadamente 15.000 ejemplos, cuidadosamente elaborados por expertos humanos, y aplicó fine-tuning supervisado sobre modelos open source, mejorando el modelo base con conocimiento específico del dominio. *LLMOPT* [13] define un esquema universal para problemas de optimización y utiliza fine-tuning con múltiples instrucciones para generalizar entre dominios, utilizando un conjunto de datos creado por expertos humanos e inteligencia artificial.

  Dado que los conjuntos de datos utilizados en los trabajos mencionados siguen siendo propietarios, replicar experimentos similares continúa siendo muy difícil. Hasta donde sabemos, no se han realizado experimentos utilizando enfoques de Aprendizaje por Refuerzo (Reinforcement Learning) en el dominio de la IO. En la sección **Trabajos Futuros**, se elaborará una discusión más profunda sobre este paradigma en particular.

En comparación con los anteriores, este trabajo se enfocó específicamente en el uso de enfoques agénticos (como los propuestos en OptiMUS, OR-LLM y OptLLM) pero con la diferencia particular de que utiliza modelos LLMs con capacidad de razonamiento, como el modelo o3 de OpenAI, como LLM principal del sistema.

Un enfoque similar ha sido desarrollado en *OR-LLM-Agent*, pero sus evaluaciones fueron realizadas sobre un conjunto de datos que contiene únicamente ejemplos de programación lineal. Nuestro objetivo es producir enfoques similares y realizar evaluaciones tanto en problemas de programación lineal como en problemas de satisfacción.

La mayoría de los conjuntos de datos mencionados anteriormente contienen principalmente problemas de Programación Lineal y no cuentan con tríadas del tipo *[descripción del problema; formulación del problema (en matemáticas o código); solución óptima]*, con excepción de los conjuntos de datos *Text2Zinc* y *NLP4LP*.

Esto es importante porque la formulación en código y/o las soluciones óptimas pueden utilizarse junto con las descripciones de problemas para hacer fine-tuning de LLMs mediante métodos supervisados o de aprendizaje por refuerzo.

3. Agente de Razonamiento para IO: R.O.R.A

3.1. Agentes de IA

Los agentes de IA son entidades computacionales capaces de percibir su entorno [19], razonar sobre objetivos o restricciones, y actuar sobre el entorno para optimizar resultados—sin requerir intervención humana paso a paso. Los agentes utilizan *herramientas* para actuar sobre su entorno y LLMs como "cerebros" que guían sus acciones.

En el contexto de la Investigación Operativa, dichos agentes pueden diseñarse para entender y descomponer dinámicamente descripciones de problemas en lenguaje natural, para luego formular y —mediante el llamado a una *herramienta de solución*— resolver el problema de optimización.

3.2. Modelos de Razonamiento

Los LLMs con çapacidades de *razonamiento*"son modelos de lenguaje de gran escala diseñados o adaptados para realizar inferencias multi-paso dirigidas a objetivos durante el tiempo de inferencia (*test time compute*) [18], frecuentemente generando pasos intermedios o llamadas a herramientas para resolver tareas complejas.

A diferencia de los modelos "tradicionales", los LLMs de razonamiento asignan mayor capacidad de cómputo por consulta, permitiéndoles descomponer problemas, simular lógica o interactuar con herramientas externas para alcanzar salidas más confiables e interpretables.

Los avances recientes en modelos que aprovechan capacidades de razonamiento y *test time compute* han permitido a los LLMs alcanzar niveles de desempeño equivalentes al conocimiento de un Doctor en áreas como Matemática, Biología y Programación.

3.3. R.O.R.A

Se propuso un *Agente de Razonamiento para Investigación Operativa* (R.O.R.A, por sus siglas en inglés) que aprovecha los avances recientes en modelos *test time compute* y *arquitecturas agénticas* para formular y resolver problemas de IO. Se apuntó a ofrecer una línea base de *benchmark* para IA en IO, así como un esquema inicial para futuros investigadores y profesionales que deseen replicar sistemas similares para democratizar la IO.

El flujo lógico del agente consiste en cuatro pasos principales: formulación matemática, generación de código (y mejora iterativa del código mediante crítica experta), ejecución del código y evaluación de la solución.

Dentro de los cuatro pasos generales, se propone un proceso de **modelado** en dos niveles, donde primero se descompone el problema en su expresión matemática y luego se traduce a código.

Aprovechamos una adaptación del enfoque de los ***cinco elementos*** (five-elements) para la formulación matemática introducido en *LLM-Opt* [13], donde un **LLM experto en matemática** descompone el problema en lenguaje natural en definiciones concretas que describen el problema: *Conjuntos, Parámetros, Variables, Objetivo y Restricciones.*

Ésta descomposición permite al agente reducir la incertidumbre en su proceso de modelado y generar propuestas de código más precisas. Asimismo, se instruye al experto en matemática a *formular dos o más alternativas de modelado cuando detecte ambigüedades o inconsistencias* en la descripción del problema (por ejemplo, datos expresados en unidades diferentes a las indicadas, restricciones imprecisas o costos no especificados, etc).

Una vez generada la formulación matemática, un **LLM experto en código** recibe el modelo matemático como entrada para crear su expresión en código. Este experto genera código Python utilizando la biblioteca

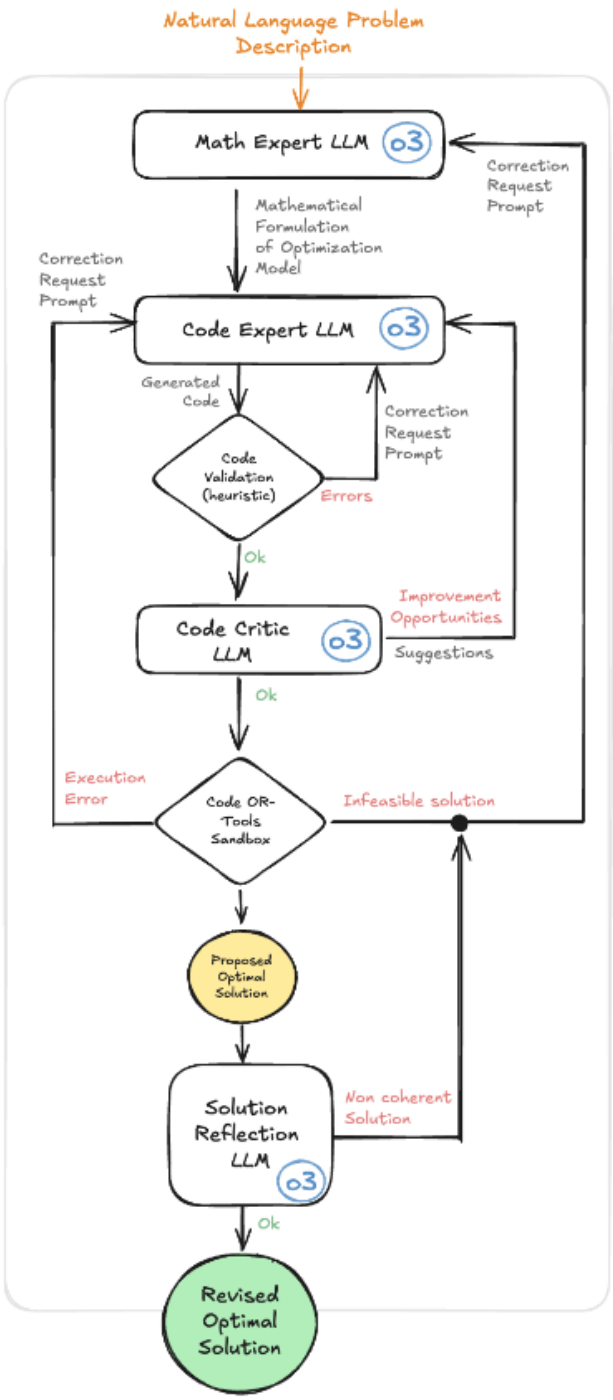

Figura 1: Flujo lógico de R.O.R.A. y actores involucrados.

OR-Tools para IO. Elegimos esta biblioteca por su flexibilidad para manejar no solo problemas LP, sino también otras técnicas de modelado matemático de forma eficiente.

Luego de que el **LLM experto en código** propone una implementación, el agente llama a una *herramienta de validación de código* que evalúa el código mediante heurísticas simples. Esta herramienta se usa para ase-

gurar que no existan errores de sintaxis y que todas las importaciones necesarias (como or-tools) estén correctamente definidas antes de pasar el código a la siguiente etapa.

Tras validar el código con herramientas heurísticas, se realiza una revisión más profunda enviando la propuesta a un **LLM crítico de código**. Este modelo es instruido para criticar el código generado buscando oportunidades de mejora, ineficiencias e incoherencias. Dado que los problemas de IO suelen ser abiertos y múltiples formulaciones pueden resolver un mismo problema, los modelos pueden generar soluciones ineficientes que tarden más en converger. Para evitar esto, se incorpora esta etapa de crítica en el flujo lógico del agente.

Finalmente, una vez que el código es aprobado por el crítico, se envía a un entorno de ejecución Python vía *tool-calling* para su posterior evaluación. Si la ejecución del código resulta en errores, los *logs* de error se envían junto con el código al experto en código para su corrección. Si en cambio la ejecución devuelve una solución infactible, los *logs* y el modelo (tanto código como matemática) se envían de vuelta al experto en matemática para reformular las restricciones.

Cabe destacar que en este escenario controlado podemos estar seguros de que este mecanismo tiene sentido, ya que todos los ejemplos validados de *Text2Zinc* y *NLP4LP* cuentan con solución óptima. Por lo tanto, si el solucionador devuelve "infactible" podemos estar seguros de que hay un error en la formulación. Esto no es lo común en escenarios reales, donde uno puede formular un problema de IO sin estar seguro de que exista una solución.

El paso final que realiza el agente es una *reflexión* sobre la solución generada. Este paso busca asegurar que la solución propuesta tenga coherencia dentro del contexto del problema. Por ejemplo: un pequeño negocio en una ciudad local que intenta optimizar su mix de productos no debería tener ganancias del orden de miles de millones. Si se encuentra que la solución no es coherente, se envía un prompt de corrección con feedback reflexivo al *LLM experto en matemática*.

Todos los pasos del agente que involucran LLMs utilizan el **modelo de razonamiento o3 de OpenAI**, aprovechando *test-time compute* para alcanzar mejores soluciones. Se utilizó **ingeniería de prompts** (*prompt-engineering* [20]) para generar los "perfiles" de cada agente (experto en matemática, programador, crítico) y acotar sus campos de acción.

Toda la estructura del agente fue construida con Langgraph [2] y sus modelos subyacentes pueden cambiarse fácilmente para adaptarse a modelos mejores en el futuro.

[2] `https://www.langchain.com/langgraph`

4. Text2Zinc & NLP4LP

El conjunto de datos ***Text2Zinc*** es un esfuerzo de código abierto para mejorar la disponibilidad de ejemplos de IO en línea.

Text2Zinc es un dataset unificado y multidominio que combina problemas tanto de optimización como de satisfacción expresados en lenguaje natural. Contiene una amplia variedad de tipos de problemas en múltiples dominios, cuidadosamente curados a partir de recursos existentes de IO. Sus ejemplos se enriquecen aún más mediante reformulación, adición de metadatos y curaduría para garantizar consistencia y calidad.

Contiene tanto problemas de **optimización** (problemas que buscan encontrar la mejor solución posible maximizando o minimizando una función objetivo bajo un conjunto de restricciones) como de **satisfacción** (problemas que buscan únicamente encontrar una solución factible que satisfaga un conjunto de restricciones sin optimizar una función objetivo específica).

El conjunto de datos incluye problemas provenientes de diversos dominios de aplicación de IO, lo que lo convierte en un recurso ideal para evaluar el desempeño de LLMs en escenarios del mundo real.

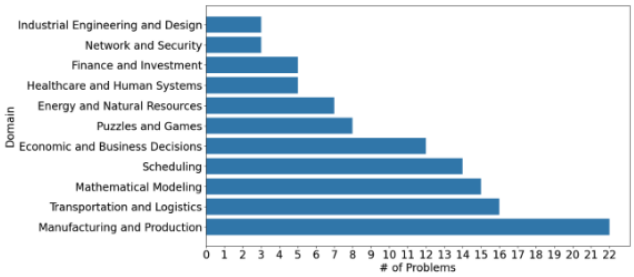

Figura 2: Distribución de los problemas de Text2Zinc por dominio. Fuente: *TEXT2ZINC: A Cross-Domain Dataset for Modeling Optimization and Satisfaction Problems in MINIZINC, Akash Singirikonda et al.*

El conjunto de datos ***NLP4LP*** fue introducido en 2023, presentando formulaciones realizadas por expertos humanos de 52 problemas de optimización LP y MILP, anotadas con sus soluciones y código para verificar la optimalidad.

Este conjunto fue posteriormente extendido hasta alcanzar 269 instancias de problemas de IO, de las cuales actualmente 249 cuentan con una solución óptima declarada en la version actual.

El conjunto fue liberado como recurso de investigación y está disponible en Hugging Face.

5. Protocolo de Evaluación

Para evaluar el desempeño del agente propuesto, se midieron cuántas de las instancias de problema intentadas resultaron en una ***coincidencia exacta*** con las

soluciones óptimas conocidas provistas en el conjunto. Dado que cada problema en el dataset incluye su valor óptimo de la función objetivo y (en algunos casos) la asignación óptima de variables, se ejecutó un *solver* sobre el código generado por el agente y se compararon los resultados obtenidos contra los esperados, listados en las triadas del dataset (*ground truth*, resultado correcto para cada ejemplo). Esta metodlogia permitió evaluar tanto la corrección de la formulación como la resolubilidad funcional del código generado.

Cabe destacar que este protocolo de evaluación no contempla la calidad ni eficiencia del código generado. Un agente puede producir implementaciones correctas pero mal estructuradas, incluyendo variables innecesarias, restricciones redundantes o prácticas no idiomáticas. Se deja pendiente una evaluación más detallada de calidad y legibilidad del código para trabajos futuros, pero ser proporcionan detalles sobre los tiempos de ejecución y la cantidad de caracteres de las formulaciones propuestas por el agente.

A su vez, se realizó un análisis de los casos fallidos para entender mejor oportunidades de mejora para futuras iteraciones del agente.

Métrica de Precisión. Se define la precisión como la proporción de instancias de problema correctamente resueltas (es decir, coincidencias exactas con los valores óptimos conocidos de la función objetivo). Formalmente:

$$\text{Precisión} = \frac{\sum_{i=1}^{N} \mathbb{1}[\hat{z}_i = z_i^*]}{N}$$

donde:

- N es el número total de instancias de problema evaluadas.
- z_i^* es el valor óptimo de la función objetivo del problema i (verdad de base), presentado en formatos JSON o txt.
- $\hat{z}_i$ es el valor de la función objetivo obtenido por el agente para el problema i.
- $\mathbb{1}[\cdot]$ es la función indicadora que devuelve 1 si la condición es verdadera y 0 en caso contrario.

La evaluación de las soluciones de los problemas fue realizada manualmente verificando si los resultados coincidían con la verdad de base para cada problema. Esto fue posible únicamente debido al tamaño relativamente pequeño del conjunto, pero sería mucho más difícil con un número mayor de ejemplos.

Estrategias de *AI-as-a-judge* podrían implementarse junto con coincidencia de strings o JSON para evaluar automáticamente el desempeño de agentes similares sobre etiquetas de *ground truth*.

Tiempo Promedio de Ejecución. Para medir el tiempo de ejecución, se registró la duración total (*wall-clock*) del pipeline completo del agente para cada instancia de problema. Esto incluye el tiempo necesario para generar la formulación matemática, producir y validar el código correspondiente, ejecutar el *solver* y realizar cualquier paso de reflexión o verificación. El temporizador comienza justo antes de que el agente empiece a procesar un problema y se detiene una vez obtenida la solución final. Esta métrica refleja la latencia de extremo a extremo del sistema, desde la entrada del problema hasta su salida final, y se reporta en segundos para cada instancia. No separa el tiempo del solucionador del tiempo de generación de prompts o del *overhead* de ejecución de código, ofreciendo en cambio una medida holística de usabilidad práctica del sistema.

6. Resultados

Resumen. R.O.R.A fue testeado sobre problemas provenientes de los conjuntos de datos *Text2Zinc* y *NLP4LP*, alcanzando los siguientes resultados:

Dataset	Examples considered	Accuracy [%]
Text2zinc	44	88,64
NLP4LP	241	81,82

Weighed Overall Accuracy [%]	82,87

Figura 3: R.O.R.A alcanza una **precisión general del 82,87 %**

Resultados en Text2Zinc. Luego de ejecutar R.O.R.A sobre 44 de los ejemplos validados del conjunto *Text2Zinc* que no pertenecen al dominio de Programación Lineal, se obtuvieron los siguientes resultados:

Problem Metadata			Matches expected output 100%?		
Problem Source	Domain	Objective	No	Yes	Total Sum
ComplexOR	Scheduling	minimization	1		1
	Transportation and Logistics	maximization		1	1
		minimization	1	4	5
Total ComplexOR			**2**	**5**	**7**
CSPLib	Industrial Engineering and Design	satisfaction		2	2
	Mathematical Modeling	minimization		2	2
	Puzzles and Games	satisfaction		3	3
	Scheduling	minimization		3	3
	Transportation and Logistics	minimization		1	1
Total CSPLib				**11**	**11**
Hakank	Economic and Business Decisions	maximization		1	1
		minimization		1	1
	Finance and Investment	maximization	1	1	2
	Healthcare and Human Systems	maximization		1	1
	Mathematical Modeling	minimization		2	2
	Network and Security	maximization		1	1
	Puzzles and Games	maximization		1	1
		satisfaction		4	4
	Scheduling	minimization		4	4
		satisfaction		1	1
	Transportation and Logistics	minimization		3	3
Total hakank			**1**	**20**	**21**
Others	Mathematical Modeling	maximization	1	3	4
		minimization	1		1
Total Others			**2**	**3**	**5**
Total Results			**5**	**39**	**44**
				Overall Accuracy	**88,64%**

Figura 4: Tabla de resultados para los problemas del conjunto *Text2Zinc* que no provienen de la fuente NLP4LP.

R.O.R.A alcanzó una **precisión del 88,64 % en los problemas de *Text2Zinc***, resolviendo —con una coin-

cidencia del 100 % frente a la solución de referencia— 39 de los 44 problemas propuestos.

De los 5 problemas que el agente no pudo resolver:

- Un problema —*Vehicle Routing Problem with Time Windows*— alcanzó el máximo de intentos de recursión debido a propuestas infactibles.
- Un problema —*Coin Plating Optimization*— obtuvo un resultado muy similar al esperado (con una diferencia de solo una unidad).
- Un problema —*Bakery Production Optimization*— devolvió una solución distinta a la esperada, donde la solución supuestamente correcta propone un escenario trivial en el que todas las variables del mix valen cero.
- Dos problemas —*Project Selection Optimization* y *Aircraft Landing Problem*— arrojaron resultados distintos por una representación matemática incorrecta de la descripción por parte del agente.

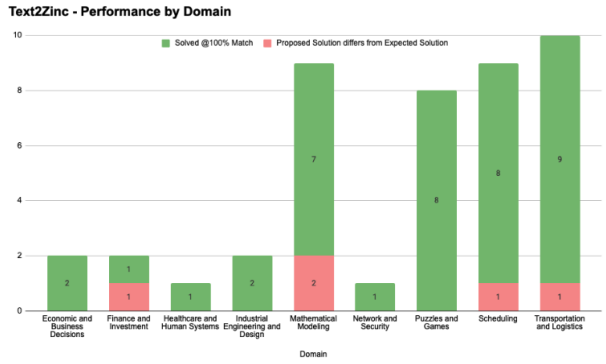

Figura 5: R.O.R.A @Text2Zinc - Desempeño por dominio

Resultados en NLP4LP. Luego de evaluar R.O.R.A en problemas distintos a los de Programación Lineal, el agente fue desafiado con 242 problemas de LP y MILP del conjunto *NLP4LP*, alcanzando una **precisión del 81,82 %** —con soluciones que coinciden completamente con las soluciones de referencia en 198 de los problemas del total evaluado—.

Es importante resaltar que en muchos casos R.O.R.A fue capaz de formular dos estrategias de modelado distintas para una misma instancia de problema (gracias a la estrategia de *prompting* utilizada en el agente experto en matemática) y resolver la tarea propuesta **en uno de los dos modelos matemáticos generados**.

Esto demuestra la importancia del modelado iterativo mediante *question-answering (QA)* —en el caso evaluado, el agente no puede hacer preguntas, pero hace suposiciones sobre restricciones poco claras— en espacios abiertos como la IO, donde preguntas aclaratorias sobre el contexto pueden ser muy útiles para un agente de IA (del mismo modo que lo serían para un experto humano) para mejorar la calidad de la solución generada.

Dentro del subconjunto de problemas que R.O.R.A no logró resolver correctamente, se encuentra que:

- **A.** Un problema alcanzó el máximo de intentos de recursión debido a propuestas infactibles.
- **B.** 30 problemas fueron formulados con estrategias matemáticas levemente distintas (por ejemplo: variables definidas como *floats* en lugar de enteras), lo cual derivó en resultados distintos. La mediana del error porcentual [*Diferencia/Óptimo Esperado*] en estos casos fue del 2,1 %.
- **C.** En 13 casos se encontró que la **solución esperada** proponía escenarios triviales o no coherentes (por ejemplo: mezclas de productos con todas las variables en cero, problemas de minimización de tiempo con tiempo óptimo igual a cero, etc.).

Tiempos de Ejecución. El tiempo promedio **end to end** (desde que la descripción del problema es ingresada al agente hasta que el agente entrega una solución factible) que el agente tomó para resolver los problemas propuestos fue de **40.0586 segundos (± 2.4665 segundos, IC=95 %)**.

Comparación contra modelos no-razonadores. Con el objetivo de analizar el desempeño de la arquitectura del agente al utilizar una familia distinta de modelos, se realizaron pruebas reemplazando el LLM central por GPT-4 (también de la empresa OpenAI). Sin embargo, no fue posible comparar directamente los resultados de este modelo con los generados por R.O.R.A + o3, ya que más de la mitad de los problemas que el agente con GPT-4 intentó resolver fallaron por *time-out de recursividad*. Es decir, el agente se contradijo a sí mismo durante los pasos internos de validación y crítica, entrando en un bucle de razonamiento que superó la recursividad máxima del grafo del agente (25 reintentos por defecto en LangGraph).

Si bien podría incrementarse este límite de recursividad interna para completar la comparación, las pruebas con GPT-4 consumieron aproximadamente la misma cantidad de tokens y costos que las realizadas con o3, debido a la alta recurrencia. Por este motivo, no se completó una pasada entera del dataset con este modelo. Se concluye que la capacidad de razonamiento de o3 permite obtener resultados apreciablemente superiores a los de sus contrapartes no razonadoras.

7. Conclusión

Se concluyó que los Agentes de Razonamiento basados en LLMs son suficientemente capaces como para comenzar a aportar valor en el complejo dominio de la Investigación Operativa. Si bien aún hay espacio para mejorar tanto en capacidades de los modelos como en ingeniería

de contexto y experiencia de usuario, el enfoque actual muestra un camino prometedor hacia un futuro en el que más personas no expertas puedan aprovechar el poder de la Investigación Operativa.

Como subproducto de la evaluación del agente, se generó un conjunto de formulaciones en código Python. El código generado está disponible públicamente en el repositorio del proyecto con el objetivo de mejorar la disponibilidad de conjuntos de datos de IO en internet. Se espera que futuros esfuerzos puedan apoyarse en el presente trabajo para realizar tareas de *fine-tuning* y evaluación de modelos.

8. Trabajos Futuros

8.1. Modelado Iterativo vía QA

El agente propuesto fue evaluado utilizando descripciones de problemas en modalidad *"single-shot"*. Esto significa que no hubo espacio en las evaluaciones (debido a que las descripciones de los problemas contienen toda la información necesaria y por restricciones del presupuesto de tokens) para realizar preguntas iterativas que el modelo podría encontrar útiles para reducir la ambigüedad del problema y mejorar sus capacidades de modelado.

Este tipo de escenarios también es muy probable que ocurra en entornos industriales, donde la resolución de problemas del mundo real requiere a menudo aclaraciones adicionales mediante preguntas, y donde un enfoque de *human in the loop* [21] podría tener sentido para una solución aumentada de problemas de IO.

Los sistemas con *humans in the loop* probablemente podrían generar una mejora significativa en el rendimiento, ya que el QA iterativo y experiencias de usuario bien diseñadas podrían ayudar tanto al humano como al agente de IA a colaborar en mejores condiciones.

Esperamos que este trabajo sirva como motivación para que futuras investigaciones exploren estos escenarios en los que el QA iterativo podría mejorar las capacidades de modelado del agente.

8.2. Aprendizaje por Refuerzo

Como se mencionó previamente, ya existe investigación en el campo del *fine-tuning* supervisado en IO.

Se plantea explorar la aplicación de técnicas de *fine-tuning* basadas en Aprendizaje por Refuerzo (RL), en lugar de —o en combinación con— el *Supervised Fine-Tuning* (SFT), dado que este último requiere ejemplos de código para el aprendizaje del modelo. El enfoque de RL, en cambio, podría aprovechar la solución óptima y la descripción del problema mediante el uso de Recompensas Verificables, como se propone en *DeepSeek-R1* (DeepSeek, 2025).

Otro aspecto relevante a investigar en el ámbito del *fine-tuning* es el Aprendizaje por Refuerzo con Recompensas Internas. Este enfoque, propuesto en *Learning to Reason without External Rewards* (Zhao et al., 2025), introduce una estrategia de entrenamiento que permite a los LLMs aprender a partir de su propia confianza en las respuestas generadas. Dicha metodología podría resultar especialmente útil en contextos donde los datos son escasos o costosos de obtener, como ocurre en el campo de la Investigación Operativa.

8.3. MiniZinc vs Python

Como se mencionó previamente, el conjunto de datos *Text2Zinc* incluye formulaciones de modelos en el lenguaje *MiniZinc*. *MiniZinc* es un lenguaje de modelado de restricciones de alto nivel que permite expresar y resolver problemas de optimización discreta de forma concisa y declarativa. En este trabajo se optó por utilizar *OR-Tools* y *Python* en lugar de otros enfoques, debido a la flexibilidad de la biblioteca y a la amplia presencia del código en *Python* dentro de los conjuntos de entrenamiento de LLMs. No obstante, dado que *MiniZinc* ofrece una interfaz más sencilla para la definición de problemas de Investigación Operativa, se considera de interés explorar enfoques similares a *R.O.R.A.* que empleen *MiniZinc* (u otros lenguajes de modelado de mayor nivel) como motor de generación de código, con el fin de analizar las diferencias de rendimiento y expresividad.

Apéndice

El código fuente, los *prompts* utilizados y los resultados generados por el agente están disponibles en el repositorio público de GitHub asociado a este trabajo:
`github.com/lucasargento/rora`

Referencias

[1] Singirikonda, A., Kadioglu, S., & Uppuluri, K. (2025). *Text2Zinc: A Cross-Domain Dataset for Modeling Optimization and Satisfaction Problems in MiniZinc*. arXiv preprint arXiv:2503.10642. https://doi.org/10.48550/arXiv.2503.10642

[2] AhmadiTeshnizi, M., et al. (2023). *OptiMUS: Optimization Modeling Using MIP Solvers and large language models*. Proceedings of the AAAI Conference on Artificial Intelligence, 2023.

[3] Howard, J., & Ruder, S. (2018). *Universal Language Model Fine-tuning for Text Classification*. Proceedings of ACL.

[4] Maloni, M. J., & Benton, W. C. (1997). *Supply chain partnerships: Opportunities for operations research.*

European Journal of Operational Research, 101(3), 419–429.

[5] Pochet, Y., & Wolsey, L. A. (2006). *Production Planning by Mixed Integer Programming*. Vol. 149. Springer.

[6] Beairsto, J., Tian, Y., Zheng, L., Zhao, Q., & Hong, J. (2021). *Identifying locations for new bike-sharing stations in Glasgow: An analysis of spatial equity and demand factors*. Annals of GIS, 0(0), 1–16. https://doi.org/10.1080/19475683.2021.1936172

[7] Tao, D. Q., Pleau, M., et al. (2020). *Analytics and Optimization Reduce Sewage Overflows to Protect Community Waterways in Kentucky*. Interfaces, 50(1), 7–20. https://doi.org/10.1287/inte.2019.1022

[8] Bitran, G. R., & Caldentey, R. A. (2016). *An overview of pricing models for revenue management*. IEEE Engineering Management Review, 44, 134–134.

[9] Singh, A. (2012). *An overview of the optimization modelling applications*. Journal of Hydrology, 466, 167–182.

[10] Antoniou, A., & Lu, W.-S. (2007). *Practical Optimization: Algorithms and Engineering Applications*. Vol. 19. Springer.

[11] Lubin, M., Huang, Z., et al. (2022). *NL4Opt: A Dataset for Natural Language Optimization Problem Solving*. NeurIPS Datasets and Benchmarks 2022.

[12] Zhang, X., Liu, Y., & Zhao, S. (2024). *OptLLM: Large Language Models for Optimization Problem Solving*. arXiv preprint arXiv:2404.12345.

[13] Jiang, W., et al. (2025). *LLMOPT: A Benchmark for Reasoning-based Optimization with Large Language Models*. arXiv preprint arXiv:2501.01234.

[14] Huang, X., Shen, Q., Hu, Y., Gao, A., & Wang, B. (2024). *Mamo: a Mathematical Modeling Benchmark with Solvers*. arXiv preprint arXiv:2405.13144v2. https://doi.org/10.48550/arXiv.2405.13144

[15] Jiang, C., Shu, X., Qian, H., Lu, X., Zhou, J., Zhou, A., & Yu, Y. (2024). *LLMOPT: Learning to Define and Solve General Optimization Problems from Scratch*. arXiv preprint arXiv:2410.13213v1. https://doi.org/10.48550/arXiv.2410.13213

[16] Huang, C., Tang, Z., Hu, S., Jiang, R., Zheng, X., Ge, D., Wang, B., & Wang, Z. (2024). *ORLM: A Customizable Framework in Training Large Models for Automated Optimization Modeling*. arXiv preprint arXiv:2405.17743. https://doi.org/10.48550/arXiv.2405.17743

[17] Zhang, B., Luo, P., Yang, G., Soong, B.-H., & Yuen, C. (2025). *OR-LLM-Agent: Automating Modeling and Solving of Operations Research Optimization Problems with Reasoning LLM*. arXiv preprint arXiv:2503.10009. https://doi.org/10.48550/arXiv.2503.10009

[18] OpenAI. (2024). *Learning to Reason with LLMs*. OpenAI (Blog / Research page). https://openai.com/index/learning-to-reason-with-llms/

[19] Huyen, Chip. (2025, 7 de enero). *Agents*. Blog de Chip Huyen. https://huyenchip.com/2025/01/07/agents.html

[20] Sahoo, P., Singh, A. K., Saha, S., Jain, V., Mondal, S., & Chadha, A. (2024). *A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications*. arXiv preprint arXiv:2402.07927. https://doi.org/10.48550/arXiv.2402.07927

[21] Google Cloud. (s. f.). *¿Qué es la interacción humana (HITL) en IA y AA?*. Google Cloud. https://cloud.google.com/discover/human-in-the-loop?hl=es$_4$19