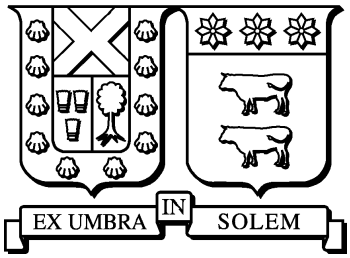


UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

DEPARTAMENTO DE INFORMÁTICA

AUTOMATIZANDO LA COMBINACIÓN DE LÓGICAS SMT EN MODELOS MATEMÁTICOS PARA LA RESOLUCIÓN DE PROBLEMAS DE OPTIMIZACIÓN

Ignacio Alejandro Jorquera Navarro

MAGÍSTER EN CIENCIAS DE LA INGENIERÍA INFORMÁTICA

ABRIL 2026

CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☐ Memoria o trabajo de título; ☒ Tesis de Postgrado;

Título del trabajo: Automatizando la combinación de lógicas SMT en modelos matemáticos para la resolución de problemas de optimización

Nombre del candidato(a): Ignacio Alejandro Jorquera Navarro

Carrera / Grado: Magíster en Ciencias de la Ingeniería Informática

Campus: Casa Central Valparaiso **;** **Departamento**: Departamento de Informática

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Carlos Castro, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL

☒ El trabajo **NO contiene información que amerite confidencialidad** y puede ser publicado de inmediato en repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (embargo) por:

☐ 6 meses; ☐ 12 meses; ☐ 2 años; ☐ 3 años; ☐ 5 años; ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

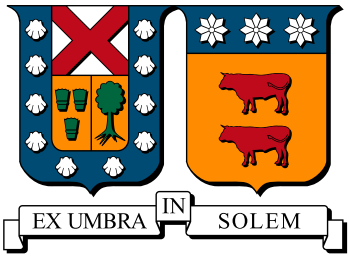
Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 14.4.2026 ; **Firma:** 

Estudiante o Candidato(a):

Fecha: 27/03/2026 ; **Firma:** 

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.



UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA

AUTOMATIZANDO LA COMBINACIÓN DE LÓGICAS SMT EN MODELOS MATEMÁTICOS PARA LA RESOLUCIÓN DE PROBLEMAS DE OPTIMIZACIÓN

Tesis de Grado presentada por
IGNACIO ALEJANDRO JORQUERA NAVARRO

como requisito parcial para optar al grado de
MAGÍSTER EN CIENCIAS DE LA INGENIERÍA INFORMÁTICA

PROFESOR GUÍA
CARLOS CASTRO

PROFESOR CO-DIRECTOR
NICOLÁS GÁLVEZ RAMÍREZ

ABRIL 2026

TÍTULO DE LA TESIS:

AUTOMATIZANDO LA COMBINACIÓN DE LÓGICAS SMT EN MODELOS MATEMÁTICOS PARA LA RESOLUCIÓN DE PROBLEMAS DE OPTIMIZACIÓN

AUTOR:

IGNACIO ALEJANDRO JORQUERA NAVARRO

Trabajo de Tesis, presentado en cumplimiento parcial de los requisitos para el Grado de **Magíster en Ciencias de la Ingeniería Informática** de la Universidad Técnica Federico Santa María.

Carlos Castro	Profesor Guía
Nicolás Gálvez Ramírez	Profesor Co-Director
Elizabeth Montero	Profesor Correferente Interno
Óscar Vásquez	Profesor Correferente Externo

Abril 2026.

Santiago, Chile.

El presente trabajo de tesis ha sido posible gracias al apoyo y financiamiento otorgado por el proyecto Fondecyt de Iniciación Nº11250273, dirigido por el profesor Nicolás Gálvez Ramírez.

Agradecimientos

Quiero agradecer al profesor Nicolás Sebastián Gálvez Ramírez, investigador responsable del proyecto que enmarca esta tesis y además mi profesor guía. Su respaldo académico y su constante orientación fueron fundamentales para el desarrollo de esta investigación, tanto en los aspectos conceptuales como metodológicos. Agradezco profundamente la confianza depositada en mí, el acompañamiento durante cada etapa del proyecto y cada etapa de mi carrera, la paciencia y las valiosas discusiones que orientaron, enriquecieron y consolidaron esta tesis. El financiamiento proporcionado por el proyecto Fondecyt permitió dedicar tiempo, recursos computacionales y recibir apoyo institucional que resultaron indispensables para la realización íntegra de este trabajo.

Extiendo también mi gratitud al equipo del proyecto, Javiera y Matías, cuyo apoyo constante ha sido fundamental desde el primer día en que se oficializó el Fondecyt. Agradezco especialmente su dedicación en las tareas administrativas asociadas al desarrollo de esta investigación, así como su gestión en la habilitación, mantención y administración de los servidores y equipos computacionales necesarios para la ejecución de los experimentos.

Al profesor Carlos Castro, por ser mi tutor del programa de Magíster y por acompañarme en este proceso que será íntegro para mi carrera profesional.

A mi familia, cuyo apoyo, paciencia y comprensión ha sido imprescindible en mi desarrollo personal y sumamente necesaria para poder llegar hasta el final de este trabajo.

Y a quienes, desde el ámbito académico y personal, contribuyeron directa o indirectamente a la culminación de esta investigación.

Resumen

La optimización ha sido un componente fundamental en la resolución de problemas complejos en ciencia, ingeniería y múltiples aplicaciones prácticas. En este trabajo se estudian problemas clásicos de satisfacción y optimización de restricciones (*CSP/CSOP*) mediante técnicas de *Optimization Modulo Theories (OMT)*, poniendo especial énfasis en cómo la elección de la lógica de primer orden utilizada para modelar las restricciones afecta significativamente el desempeño del solver. Para ello, se desarrolló un flujo de trabajo para transformar modelos originalmente formulados en aritmética entera lineal (`LIA`) hacia otras lógicas: `LRA`, `QF_BV` y `QF_ALIA` mediante reglas de traducción formales y mecanismos de consistencia entre lógicas heterogéneas. Sobre esta base se incorporó un proceso de búsqueda guiado por *Hill Climbing* para seleccionar, entre combinaciones homogéneas y heterogéneas, aquellas combinaciones de lógicas capaces de mejorar el desempeño del solver. El enfoque fue evaluado en cinco problemas representativos: *N-Queens*, *Unbounded Knapsack Problem*, *Nurse Scheduling Problem*, *Travelling Salesman Problem* y *Balanced Academic Curriculum Problem*. Los resultados muestran que, en cuatro de los cinco problemas, existen combinaciones heterogéneas y homogéneas que superan consistentemente al modelo inicial basado únicamente en `LIA`, logrando reducciones de tiempo de hasta dos órdenes de magnitud, incrementando la cantidad de instancias resueltas y disminuyendo el tiempo requerido y el uso de memoria. Estos hallazgos confirman la hipótesis de que la mezcla automática de lógicas, junto con la transformación dinámica de restricciones a partir de un modelo inicial, constituye un mecanismo eficaz para potenciar el rendimiento de solvers *OMT*. Además se demuestra que sería eficaz construir un sistema general de modelamiento que ajuste automáticamente la lógica usada en cada restricción según la estructura del problema, ampliando la eficiencia, aplicabilidad y capacidad adaptativa de los métodos de optimización.

Abstract

Optimization is fundamental in solving complex problems across science, engineering, and numerous applications. This work investigates Constraint Satisfaction and Optimization Problems (CSP/CSOP) using Optimization Modulo Theories (OMT) techniques. We emphasize how the choice of SMT logic used to model constraints significantly impacts solver performance. Building on this, we developed a novel workflow that systematically transforms models initially formulated in Linear Integer Arithmetic (`LIA`) into `LRA`, `QF_BV`, and `QF_ALIA`. This transformation uses formal encoding rules for consistency between these heterogeneous logics. We incorporated a Hill Climbing search to dynamically select logic combinations, homogeneous and heterogeneous, capable of improving solver performance. To assess this methodology, the approach was evaluated on five representative problems: N-Queens, Unbounded Knapsack, Nurse Scheduling, Traveling Salesman, and Balanced Academic Curriculum. The results consistently demonstrate that, across four of the five problems, the dynamically selected heterogeneous and homogeneous logic combinations significantly outperform the initial single-logic `LIA` model. These combinations achieve time reductions of up to two orders of magnitude, resulting in an increased number of solved instances while simultaneously reducing the required time and memory usage. These findings confirm the hypothesis that automatic logic mixing, coupled with dynamic constraint transformation from an initial model, is an effective mechanism for enhancing the performance of OMT solvers. Furthermore, they demonstrate the viability of building a general modeling system that automatically adjusts the logic for each constraint based on the problem's inherent structure, thereby expanding the efficiency, applicability, and adaptive capacity of optimization methods.

Glosario

ALIA: *ArraysEx over LIA* (Arreglos con Extensibilidad sobre Aritmética Lineal de números Enteros).
AX: *ArraysEx* (Arreglos con Extensibilidad).
BACP: *Balanced Academic Curriculum Problem* (Problema de Balanceo de Mallas Curriculares).
BV: *FixedSizeBitVectors* (Vectores de Bits).
CP: *Constraint Programming* (Programación con Restricciones).
CSP: *Constraint Satisfaction Problem* (Problema de Satisfacción de Restricciones).
CSOP: *Constraint Satisfaction and Optimization Problem* (Problema de Satisfacción de Restricciones y Optimización).
LIA: *Linear Integer Arithmetic* (Arítmetica Lineal de números Enteros).
LRA: *Linear Real Arithmetic* (Aritmética Lineal de números Reales).
NSP: *Nurse Scheduling Problem* (Problema de Asignación de Enfermeros/as).
OMT: *Optimization Modulo Theories* (Teorías de Optimización de Módulos).
SMT: *Satisfiability Modulo Theories* (Teorías de Satisfacibilidad de Módulos).
TSP: *Travelling Salesman Problem* (Problema del Vendedor Viajero).
UKP: *Unbounded Knapsack Problem* (Problema de la Mochila sin Límites).
Z3: *SMT solver* de *Microsoft Research*.

Índice de Contenidos

Lista de Tablas

Lista de Algoritmos

Lista de Figuras

Capítulo 1

Introducción

La optimización y la resolución de problemas de satisfacción de restricciones han acompañado a la humanidad desde sus orígenes [1], primero en la vida cotidiana [2] y, más recientemente, en múltiples áreas de la ciencia, la ingeniería y la industria [3]. Estos problemas aparecen en contextos tan diversos como la planificación de recursos [4, 5], la logística y el diseño de rutas [6], la gestión energética [7, 8], la inteligencia artificial [9], biología computacional [10, 11], la economía [12], la verificación de software y hardware [13], y en cualquier otro problema que contenga múltiples soluciones posibles [14, 15]. En todos estos ámbitos, contar con herramientas eficientes para modelar y resolverlos se ha vuelto fundamental [16, 17, 18, 19].

Una de las aproximaciones más utilizadas para representar y resolver este tipo de problemas es la Programación Lineal [20], que permite modelar tanto restricciones como funciones objetivo en un marco matemático bien definido. Sin embargo, existen problemas cuya naturaleza requiere herramientas más expresivas. En estos casos, *Satisfiability Modulo Theories* (SMT) [21] se presenta como una alternativa poderosa, al extender la verificación proposicional (SAT) [22] hacia múltiples dominios lógicos como números enteros, reales, arreglos, bit-vectors o strings.

A su vez, la generalización *Optimization Modulo Theories* (OMT) [23] combina SMT con

problemas de optimización, lo que habilita el modelado y resolución de *Constraint Satisfaction and Optimization Problems* (CSOP) en entornos complejos a partir de distintas teorías, vale decir, lógicas de primer-orden y sus combinaciones.

En [24] se expone el impacto que tiene la elección de una lógica SMT específica —como *Linear Integer Arithmetic* (`LIA`), *Linear Real Arithmetic* (`LRA`), *Quantifier-Free Bit-Vectors* (`QF_BV`) y *Quantifier-Free Arrays with Integer Arithmetic* (`QF_ALIA`)— sobre el desempeño de un mismo problema. Los resultados mostraron diferencias significativas: en algunos casos el solver alcanzaba la solución óptima en menos del uno porciento del tiempo que requería otra lógica. En ese trabajo, la comparación entre lógicas se abordó considerando cada problema en el marco de una única lógica, modelando todas sus restricciones de manera uniforme. No obstante, un problema real puede involucrar restricciones de distinta naturaleza, lo que sugiere que diferentes dominios lógicos podrían resultar más adecuados para representar distintas partes del modelo. Esta perspectiva abre la posibilidad de combinar múltiples lógicas en una misma instancia, ampliando el espacio de búsqueda y ofreciendo un marco más flexible para la resolución de problemas complejos.

La adopción de este enfoque plantea diversos desafíos conceptuales y técnicos. Entre ellos se encuentran la definición de criterios para determinar qué lógica resulta más apropiada en cada segmento del modelo, la posibilidad de traducir representaciones entre diferentes lógicas de forma automatizada, y la necesidad de evaluar rigurosamente el impacto que dicha combinación tiene en el desempeño de los solvers. Estos aspectos constituyen un eje central en la discusión sobre cómo aprovechar de manera más efectiva las capacidades heterogéneas de los distintos formalismos lógicos.

1.1. Hipótesis y Objetivos

Para el presente trabajo se plantean las siguientes hipótesis:

- La resolución de problemas de satisfacción de restricciones y optimización modelados con más de una lógica *SMT* podría presentar mejores resultados en comparación a resolver ese mismo problema usando solo una lógica.

- La resolución de problemas de satisfacción de restricciones y optimización modelados con lógicas *SMT*, en general, podría verse mejorada con un algoritmo o framework que permita encontrar la mejor lógica o combinación de lógicas *SMT* para un determinado problema, para luego transformar automáticamente el modelo matemático original del problema a la o las lógicas encontradas.

1.1.1. Objetivo general

Para la comprobación de las hipótesis planteadas, se define el siguiente objetivo general:

Mejorar el desempeño de solvers *OMT* mediante la selección y transformación automática de lógicas *SMT* aplicadas sobre las **restricciones y funciones objetivo** de los modelos, cuando se intente resolver problemas de satisfacción de restricciones y optimización formulados en una lógica *SMT*.

1.1.2. Objetivos específicos

Para cumplir con el objetivo general se definen los siguientes objetivos específicos:

- Modelar, resolver y analizar resultados usando *OMT* sobre un conjunto de problemas de satisfacción de restricciones y optimización conocidos en la literatura, explorando diferentes combinaciones de lógicas *SMT* aplicadas a las restricciones de cada modelo.
- Automatizar el proceso de búsqueda y selección de lógicas con algoritmos de búsqueda trayectorial para encontrar la combinación de lógicas *SMT* sobre las restricciones de cada modelo que permita obtener las mejores soluciones en el menor tiempo al resolver *CSOP* usando *OMT*.
- Automatizar el proceso de *encoding* para convertir las restricciones y expresiones del modelo desde una lógica *SMT* hacia otra, garantizando la equivalencia semántica y la factibilidad de las soluciones en el modelo original.

1.2. Contribuciones

El presente trabajo aporta un conjunto de avances conceptuales, metodológicos y experimentales en el estudio del desempeño de solvers de *Optimization Modulo Theories (OMT)* bajo diferentes elecciones de lógicas y combinaciones híbridas. Las principales contribuciones son las siguientes:

1. **Un framework general para la transformación automática de modelos entre lógicas *SMT***: Se propone un sistema capaz de transformar modelos inicialmente formulados en `LIA` hacia otras teorías de la *SMT-LIB*: `LRA`, `QF_BV` y `QF_ALIA`, mediante reglas de traducción formales que preservan la semántica del modelo. El framework propuesto automatiza la conversión del dominio de las variables, genera la estructura necesaria para garantizar la consistencia entre múltiples lógicas y permite construir modelos homogéneos e híbridos a partir de un único modelo base.

   Esto reduce la dependencia del modelador y permite experimentar rápidamente con distintas representaciones del mismo problema.

2. **Un mecanismo para seleccionar combinaciones de lógicas mediante *Hill Climbing***: Se propone un proceso de búsqueda que identifica automáticamente combinaciones de lógicas capaces de mejorar el rendimiento del solver. Esta contribución incluye un vecindario diseñado para explorar transformaciones parciales del modelo, una estrategia de evaluación basada en métricas de éxito, tiempo y memoria, y por último, evidencia empírica de que el enfoque encuentra consistentemente combinaciones híbridas más eficientes que las configuraciones homogéneas en determinados problemas.

   Este mecanismo constituye una primera aproximación práctica a la selección automática de teorías en *OMT*.

3. **Una evaluación experimental exhaustiva en cinco problemas representativos**: Se construyó un banco de experimentos sobre los problemas *N-Reinas*, *UKP*, *NSP*, *TSP* y *BACP*, analizando más de un centenar de combinaciones lógicas. Esto permitió identificar cuáles teorías son naturalmente más adecuadas para cada tipo de estructura y

demostrar el potencial de las combinaciones heterogéneas para superar a las lógicas puras

Esta evaluación constituye uno de los estudios más amplios de comparación empírica entre combinaciones de lógicas de primer orden aplicadas a *OMT*.

4. **Evidencia empírica de que las combinaciones heterogéneas pueden mejorar drásticamente el desempeño**: Los resultados muestran que, en cuatro de los cinco problemas evaluados, las combinaciones heterogéneas superan al modelo original en `LIA`, se logran reducciones de tiempo de hasta dos órdenes de magnitud en problemas complejos, se incrementa la cantidad de instancias resueltas y se reduce el uso de memoria.

   Estos hallazgos consolidan la hipótesis central del trabajo y abren la puerta a sistemas de modelamiento más adaptativos.

1.3. Estructura de la tesis

Esta tesis esta estructurada de la siguiente forma:

- **Capítulo 2 - Modelamiento y Resolución de Problemas de Optimización**: Se revisitan los conceptos necesarios que fundamentan este trabajo, y se analiza el estado del arte para, a partir de este, definir el problema a abordar.
- **Capítulo 3 - En camino hacia la codificación automática**: Descripción del framework propuesto para la transformación de restricciones entre distintas lógicas *SMT* a partir de un modelo basado en `LIA`, y del algoritmo utilizado para encontrar combinaciones de lógicas mejores que los modelos uniformes.
- **Capítulo 4 - Resultados**: Se presenta la configuración experimental y un análisis crítico de los resultados obtenidos a partir de la propuesta de solución.
- **Capítulo 5 - Conclusiones y Trabajo Futuro**: Se encuentran las principales conclusiones derivadas del trabajo realizado y los resultados experimentales. Se discute en torno al alcance de los objetivos y se presentan las limitaciones de la propuesta.

También se proponen futuras líneas de investigación que surgen a partir de esta investigación.

Capítulo 2

Modelamiento y Resolución de Problemas de Optimización

El presente capítulo aborda las principales aproximaciones formales utilizadas en el modelamiento y resolución de problemas de optimización y satisfacción de restricciones. Su propósito es establecer un hilo conceptual que permita comprender la evolución desde los métodos clásicos, como la Programación Lineal, hasta los enfoques modernos basados en lógicas de primer orden, como *Satisfiability Modulo Theories* (SMT) y *Optimization Modulo Theories* (OMT).

En particular, se revisan las bases matemáticas y computacionales que sustentan la formulación de problemas de optimización, destacando cómo distintas representaciones, continuas, discretas o lógicas, influyen en la expresividad del modelo y en la eficiencia de los procedimientos de resolución. Esta revisión servirá de fundamento para formalizar, posteriormente, el problema central abordado en este trabajo, el cual busca explorar mecanismos automáticos para la traducción y combinación de lógicas dentro del marco de *OMT*.

Nota de lectura. Las Secciones 2.1, 2.2 y 2.3 presentan una revisión de fundamentos ampliamente conocidos por la comunidad (*LP*, *SAT* y metaheurísticas), incluida aquí con el propósito de establecer la notación y el hilo conceptual que conecta estos enfoques con el marco *SMT*/*OMT*. Los lectores familiarizados con estos temas pueden dirigirse directamente

a las Secciones 2.2 y 2.4, donde se introduce el paradigma central de este trabajo.

2.1. Programación lineal

La Programación Lineal (*LP*, del inglés *Linear Programming*), constituye una de las metodologías fundamentales para la resolución de problemas de optimización. Su objetivo es determinar los valores de un conjunto de variables que optimicen (maximicen o minimicen) una función objetivo lineal, sujeta a un conjunto de restricciones lineales que delimitan el espacio factible de soluciones [20].

En términos generales, un problema de Programación Lineal puede expresarse como:

$$\begin{aligned} \text{Minimizar} \quad & \mathbf{z}(x) = Cx \\ \text{Sujeto a:} \quad & Ax \leq B \\ & x \in \mathbb{R}^n \end{aligned}$$

donde $x \in \mathbb{R}^n$ representa el vector de variables de decisión, $A \in \mathbb{R}^{m \times n}$ representa la matriz de coeficientes de las restricciones, $B \in \mathbb{R}^m$ representa el vector de términos independientes y $C \in \mathbb{R}^n$ representa los coeficientes de la función objetivo.

Los principales componentes de un modelo de Programación Lineal son los siguientes:

- **Parámetros:** Valores constantes que definen las condiciones del problema y que provienen de su contexto de aplicación.
- **Variables:** Elementos que representan las decisiones bajo control del modelo, cuyo valor se determina al resolver el problema.
- **Función objetivo:** Representa la magnitud a optimizar, típicamente asociada a costos, beneficios, tiempos o recursos.
- **Restricciones:** Relaciones lineales entre las variables que limitan el espacio de búsqueda de soluciones factibles.

- **Dominios:** Conjuntos que definen el tipo de valores que pueden tomar las variables.

En este tipo de modelos, las variables suelen definirse en el dominio de los números reales ($\mathbb{R}$). Sin embargo, muchas aplicaciones prácticas requieren que algunas o todas las variables adopten únicamente valores enteros o binarios, lo que amplía el marco de la programación lineal hacia formulaciones más específicas. Estas extensiones dan origen a la **Programación Lineal Entera** (ILP, del inglés *Integer Linear Programming*), a su versión binaria *0–1 ILP* [25], y a la **Programación Lineal Entera Mixta** (MILP, del inglés *Mixed-Integer Linear Programming*), ampliamente utilizadas en problemas de asignación, planificación y optimización combinatoria [26, 27, 28].

2.1.1. Programación Lineal Entera y Binaria

En la **Programación Lineal Entera** (ILP, del inglés *Integer Linear Programming*), las variables de decisión se restringen a valores del conjunto de los números enteros ($\mathbb{Z}$), lo que permite modelar fenómenos discretos, como cantidades indivisibles, asignaciones o decisiones lógicas [20]. Su formulación general se expresa como:

$$\begin{aligned} \text{Minimizar:} \quad & \mathbf{z}(x) = Cx \\ \text{Sujeto a:} \quad & Ax \leq B \\ & x \in \mathbb{Z}^n \end{aligned}$$

A modo de ejemplo, consideremos el problema 054 de la *CSPLib*: *N-Queens* o *N-Reinas* [29]: ¿Pueden $\boldsymbol{N}$ reinas del mismo color ser posicionadas en un tablero de ajedrez de tamaño $\boldsymbol{N} \times \boldsymbol{N}$, tal que ninguna de las reinas pueda atacar o ser atacada por las otras?

Para este problema se considera que una reina será atacada si hay otra reina en la misma fila, columna o diagonal.

En [30] se propone un método de trabajo usando Programación Lineal Entera para modelar y resolver el problema de las *N-Reinas* en múltiples dimensiones, generando la solución mostrada en la Figura 2.1 para el caso $N = 6$. Primero comienzan desde el modelo del

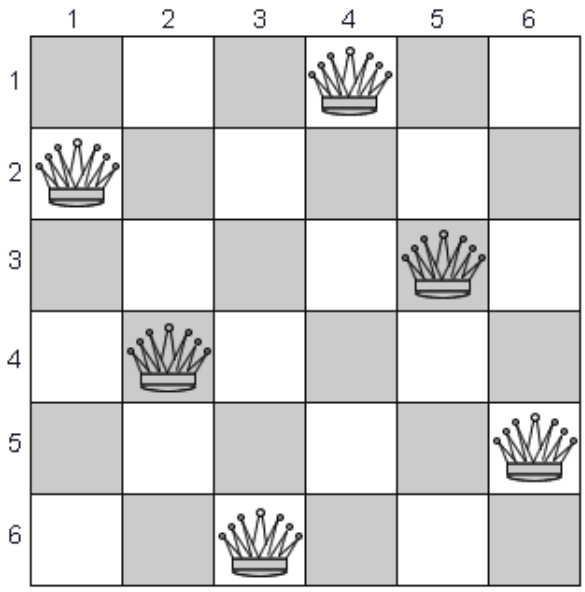


Figura 2.1: Tablero mostrando una de las posibles soluciones al problema las *N-Reinas* cuando $N = 6$

problema original una configuración $(N, 2)$, donde tienen N reinas en un tablero de tamaño 2×2, y luego lo van extendiendo para eventualmente poder resolver instancias de tamaño (N, d) tal que $N \geq 4$ y $d \in \{2, 3, 4, 5\}$.

A continuación se presenta el modelo de *ILP* resuelto en el trabajo mencionado, para la configuración $(N, 2)$ del problema original, el cual evita que hayan reinas atacándose unas a otras por medio de restricciones, y a la vez intenta colocar el mayor número de reinas posibles en el tablero mediante la función objetivo:

Parámetros:

- N : Número de reinas y número de filas y columnas del tablero, $N \in \mathbb{Z}^+$

Variables:

- $x_{ij} = \begin{cases} 1 & \text{Si hay una reina en la casilla } (i, j) \\ 0 & \text{en caso contrario} \end{cases}$; $\forall i, j \in \{1, ..., N\} : N \geq 4$

Basándonos en el ejemplo de la Figura 2.1, la reina posicionada en la fila con índice 1 y en la columna con índice 4 es representada por la variable $X_{1,4} = 1$, mientras que el resto de las casillas de la misma fila se representan como $X_{1,j} = 0 : j \in \{1, 2, 3, 5, 6\}$.

Función objetivo:

- Maximizar la cantidad de reinas posicionadas en el tablero:

$$\text{Maximizar:} \quad \mathbf{z}(x) = \sum_{i=1,j=1}^{N} x_{ij}$$

Restricciones:

1. Debe haber exactamente una reina por fila:

$$\sum_{i=1}^{N} x_{ij} = 1; \quad \forall j \in \{1, ..., N\}$$

2. Debe haber exactamente una reina por columna:

$$\sum_{j=1}^{N} x_{ij} = 1; \quad \forall i \in \{1, ..., N\}$$

3. Debe haber a lo más una reina por diagonal negativa:

$$\sum_{i+j=k}^{N} x_{ij} \leq 1; \quad \forall i, j \in \{1, ..., N\} \quad : \quad 2 \leq k \leq 2 \times N$$

4. Debe haber a lo más una reina por diagonal positiva:

$$\sum_{i-j=k}^{N} \sum_{j=1}^{N} x_{ij} \leq 1; \quad \forall i, j \in \{1, ..., N\} \quad : \quad -N + 1 \leq k \leq N - 1$$

5. Dominio de la variable de decisión:

$$0 \leq x_{ij} \leq 1; \quad \forall i, j \in \{1, ..., N\}$$

Los problemas que siguen la estructura del modelo anterior se denominan Problemas de Satisfacción de Restricciones y Optimización (*CSOP*, del inglés *Constraint Satisfaction Optimization Problem*), y como su nombre indican, buscan optimizar una función objetivo mientras se satisface un conjunto de restricciones.

Así mismo, existen también los Problemas de Satisfacción de Restricciones (*CSP*, del inglés *Constraint Satisfaction Problem*), los cuales solo buscan satisfacer un conjunto de restricciones sin tener ninguna función objetivo que optimizar. En [31] se aborda el problema de las N-Reinas como un *CSP* con la herramienta *GNU Linear Programming Kit* (*GLPK*) [32]. Con ello logran resolver satisfactoriamente una instancia de 16 reinas en donde, solo por medio de restricciones, se logró posicionar exactamente 16 reinas de tal manera que ninguna resultara ser atacada por las demás.

Una versión particular de los problemas de **Programación Lineal Entera** es la **Programación Lineal Binaria (PLB)**, en la cual las variables están restringidas al dominio $\{0, 1\}$. Esta formulación es ampliamente utilizada para representar problemas de decisión ("sí o no"), como la selección de proyectos, rutas o asignaciones.

A diferencia de la **Programación Lineal** continua, los problemas enteros y binarios presentan una complejidad computacional considerablemente mayor, dado que el espacio de búsqueda es discreto y, en general, no convexo. Por esta razón, las técnicas de resolución exactas suelen apoyarse en métodos de ramificación y acotación (*B&B*) [33, 34, 35] o en procedimientos híbridos que combinan relajaciones continuas y estrategias heurísticas [36, 37, 38].

La **Programación Lineal Entera**, además de su relevancia práctica, constituye un punto de conexión natural entre los métodos de optimización matemática clásicos y los enfoques lógicos modernos, ya que muchas de sus formulaciones pueden expresarse equivalentemente mediante representaciones proposicionales o de primer orden, lo cual conduce al marco de los problemas de **Satisfacción Booleana (SAT)**.

2.2. Problemas de Satisfacción Booleana (*SAT*)

Los **problemas de Satisfacción Booleana** (o *SAT*, del inglés *Satisfiability*) [22] constituyen una de las clases más fundamentales dentro de la teoría de la complejidad computacional. Un problema *SAT* busca determinar si existe una asignación de valores de verdad a un conjunto de variables booleanas que satisfaga un conjunto de fórmulas proposicionales dadas.

Simplificadamente, dada un conjunto de fórmulas proposicionales ϕ construido a partir de variables booleanas $x_1, x_2, ..., x_n \in \{\text{True}, \text{False}\}$ y operadores lógicos ($\neg, \wedge, \vee$), el problema consiste en determinar si existe una asignación tal que $\phi = \text{True}$. Si tal asignación existe, la fórmula se considera satisfacible (*satisfiable* o *sat*); de lo contrario, insatisfacible (*unsatisfiable* o *unsat*).

El problema de decisión *SAT* fue demostrado NP-completo [22], y su estudio ha dado origen a una extensa línea de investigación orientada al desarrollo de solvers altamente eficientes, como MiniSAT [39], Glucose [40] o Lingeling [41].

Más allá de su interés teórico, los problemas *SAT* tienen una amplia aplicabilidad práctica. Se emplean como motor base en campos como la verificación formal de hardware y software [42], la planificación automática (*AI planning*) [43], el testing de circuitos [44], la verificación de modelos (*model checking*) [45] y, en general, en cualquier problema que pueda formularse como la búsqueda de una asignación booleana consistente.

El fuerte del problema de Satisfacibilidad Booleana (*SAT*) radica en su versatilidad para representar problemas combinatorios complejos mediante **Fórmulas Conjuntivas Normales (*CNF*), del inglés *Conjunctive Normal Form***. En este formato estándar, una fórmula lógica φ se define como una conjunción ($\wedge$) de cláusulas, donde cada cláusula es, a su vez, una disyunción ($\vee$) de literales. Un literal corresponde simplemente a una variable booleana x o a su negación $\neg x$.

Formalmente, una fórmula en *CNF* se expresa como:

$$\varphi = \bigwedge_{i=1}^{m} C_i = \bigwedge_{i=1}^{m} \left(\bigvee_{j \in J_i} l_{i,j} \right)$$

Donde C_i representa la i-ésima cláusula y $l_{i,j}$ es un literal perteneciente a ella.

Por ejemplo, considere una fórmula φ sobre las variables $\{x_1, x_2, x_3\}$:

$$\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_3) \tag{2.1}$$

Para que φ sea satisfacible, el solver debe encontrar una asignación de valores de verdad (*True*/*False*) tal que todas las cláusulas se evalúen como verdaderas simultáneamente. Esta estructura regular y uniforme facilita enormemente el diseño de estructuras de datos eficientes y procedimientos de deducción.

Dichos procedimientos han sido progresivamente optimizados mediante algoritmos completos como ***Davis-Putnam-Logemann-Loveland (DPLL)*** [46], y su evolución moderna, ***Conflict-Driven Clause Learning (CDCL)*** [47], los cuales se potencian con heurísticas adaptativas de selección de variables como VSIDS [48, 49, 50].

La sintaxis formal para los problemas de **Satisfacción Booleana** se construye de manera recursiva. Sea $\mathcal{V}$ un conjunto finito de variables proposicionales. Una **Fórmula Bien Formada** ($\mathcal{F}$) se define mediante las siguientes reglas:

1. **Base (Átomo):** Toda variable proposicional $x \in \mathcal{V}$ es una fórmula (denominada átomo).
2. **Negación:** Si $\mathcal{F}$ es una fórmula, entonces su negación $\neg\mathcal{F}$ también lo es.
   - Si $\mathcal{F}$ es un átomo x, entonces $\neg\mathcal{F}$ corresponde al literal negativo $\bar{x}$.
   - El conjunto de variables que componen una fórmula se denota como $var(\mathcal{F})$.
3. **Operadores Binarios:** Si $\mathcal{F}$ y $\mathcal{G}$ son fórmulas, entonces su disyunción ($\mathcal{F} \vee \mathcal{G}$) y su conjunción ($\mathcal{F} \wedge \mathcal{G}$) son fórmulas válidas.
4. **Operadores Derivados:** A partir de los operadores fundamentales anteriores, se definen construcciones lógicas avanzadas para representar relaciones de causalidad y equivalencia:
   - **Implicación Lógica ($\Rightarrow$):** La expresión $\mathcal{F} \Rightarrow \mathcal{G}$ (si $\mathcal{F}$ entonces $\mathcal{G}$) es sintácticamente equivalente a $\neg\mathcal{F} \vee \mathcal{G}$.

- **Doble Implicación (⇔):** La expresión $\mathcal{F} \Leftrightarrow \mathcal{G}$ ($\mathcal{F}$ si y solo si $\mathcal{G}$) es equivalente a $(\mathcal{F} \Rightarrow \mathcal{G}) \wedge (\mathcal{G} \Rightarrow \mathcal{F})$.

Esta capacidad de derivación es fundamental para el modelado. Por ejemplo, una restricción de implicación como "Si el motor se enciende (x_1), la válvula debe abrirse (x_2)" se modela inicialmente como $x_1 \Rightarrow x_2$. Para que un solver *SAT* estándar pueda procesarla, esta se transforma a su forma clausal equivalente:

$$(x_1 \Rightarrow x_2) \equiv (\neg x_1 \vee x_2)$$

De manera similar, una equivalencia estricta entre dos componentes, $x_a \Leftrightarrow x_b$, se descompone en dos cláusulas conjuntivas: $(\neg x_a \vee x_b) \wedge (\neg x_b \vee x_a)$.

Usando *SAT* es posible revisitar el problema de las ***N-Reinas*** visto en la Sección 2.1.1, y modelarlo utilizando proposiciones lógicas [51]:

Parámetros:

- N : Número de reinas y número de filas y columnas del tablero, $N \in \mathbb{Z}^+$

Variables:

- $x_{ij} = \begin{cases} True & \text{si hay una reina en la casilla } (i, j) \\ False & \text{en caso contrario} \end{cases}$; $\quad \forall i, j \in \{1, ..., N\}$

Restricciones:

1. Debe haber ***N*** reinas en el tablero:

$$\bigvee_{j=1}^{N} x_{ij}; \quad \forall i \in \{1, ..., N\}$$

2. No puede haber más de una reina por columna:

$$\bigwedge_{j=1}^{N} \neg \left(\bigvee_{1 \leq i < k \leq N}^{N} \left(x_{ij} \wedge x_{kj} \right) \right)$$

3. No puede haber más de una reina por fila:

$$\bigwedge_{i=1}^{N} \neg \left(\bigvee_{1 \leq j < k \leq N} \left(x_{ij} \wedge x_{ik} \right) \right)$$

4. No puede haber más de una reina por diagonal:

$$\bigwedge_{\substack{1 \leq i < k \leq N \\ 1 \leq j, \ell \leq N \\ i - j = k - \ell}}^{N} (\neg x_{ij} \vee \neg x_{k\ell}) \quad \wedge \quad \bigwedge_{\substack{1 \leq i < k \leq N \\ 1 \leq j, \ell \leq N \\ i + j = k + \ell}}^{N} (\neg x_{ij} \vee \neg x_{k\ell})$$

De manera similar es posible modelar otros *CSP*/*CSOP* como problemas *SAT* para luego resolverlos por medio de *solvers* especializados en operaciones booleanas, como por ejemplo, *MiniSAT* [39], *Glucose* [49], *CaDiCaL* [52], *Kissat* [53], *OptiMathSAT* [54], max-SAT y max-SMT [55], *Z3* [56] o *CVC* [57].

2.3. Técnicas de Resolución para Problemas de Optimización y Satisfacción de Restricciones

Existen diversas estrategias para abordar los problemas de optimización y satisfacción de restricciones, las cuales se pueden agrupar en dos grandes categorías según su naturaleza de exploración del espacio de soluciones: los métodos de resolución completa o exacta y los métodos de resolución incompleta o aproximada. Los primeros garantizan la obtención de una solución óptima global, pero su costo computacional puede crecer exponencialmente con el tamaño del problema. Los segundos, en cambio, buscan soluciones de alta calidad en tiempos razonables, sacrificando la garantía de optimalidad a cambio de eficiencia y escalabilidad [58, 59].

Dentro de los métodos exactos, la Programación Lineal (LP) y la Programación Lineal Entera (ILP) —junto con su versión mixta, MILP— constituyen la base formal más extendida para modelar y resolver problemas de optimización combinatoria. Su desarrollo ha dado origen a una serie de algoritmos especializados, como los métodos Simplex [60], Branch and Bound [61] y Cutting Planes [62]. Estos métodos se emplean de manera intensiva en los principales

solvers comerciales de optimización, que combinan varias de estas técnicas en su núcleo de solución [63].

No obstante, la aplicabilidad de los enfoques exactos se ve severamente limitada por la complejidad computacional. A medida que incrementa el número de variables y restricciones, estos métodos se vuelven computacionalmente intratables, particularmente en espacios de búsqueda discretos, no convexos o altamente multimodales [64, 65]. Ante este escenario, surgen como alternativa los métodos metaheurísticos, los cuales ofrecen mecanismos flexibles de búsqueda guiada que, mediante principios estocásticos y evolutivos, permiten obtener soluciones de alta calidad en tiempos razonables [66]. Estos métodos se dividen, de manera general, en metaheurísticas trayectoriales, que operan sobre una única solución en evolución, como *Hill Climbing* [64], *Simulated Annealing* [67] o *Iterated Local Search* [66], y metaheurísticas poblacionales, que trabajan con conjuntos de soluciones que evolucionan colectivamente [68], como *Evolutionary Programming* [69], *Evolutionary Strategies* [70], *Genetic Algorithms* [71] y *Genetic Programming* [72].

La base teórica que sostiene a estos métodos para resolver problemas de optimización de manera eficiente reside en el Teorema Fundamental de la Programación Lineal [73]. Este teorema establece una equivalencia crucial entre la geometría del problema y su álgebra. Formalmente, sostiene que si un problema de Programación Lineal en forma estándar tiene una solución óptima finita, entonces existe al menos una Solución Básica Factible (*SBF*) que es óptima [74].

Geométricamente, esto implica que no es necesario buscar la solución en los infinitos puntos interiores del poliedro factible; basta con examinar sus vértices (puntos extremos). Dado que el número de vértices es finito (aunque potencialmente grande), el problema continuo se reduce a una búsqueda combinatoria finita [75].

2.3.1. Métodos de Resolución Completa

Los métodos de resolución completa o exacta se fundamentan en la exploración sistemática y exhaustiva del espacio de soluciones, garantizando la obtención de un óptimo global siempre

que el proceso logre completarse [76, 77]. Dentro de este grupo, algunas de las técnicas más influyentes en la resolución de modelos de *LP*, *ILP* y/o *MILP* son el método *Simplex* [60], *Branch and Bound* [61] y los métodos de *Cutting Planes* [62].

Simplex

El método *Simplex*, introducido por George B. Dantzig en 1947 [78, 60], constituye uno de los pilares fundamentales en la historia de la optimización matemática. Diseñado para resolver problemas de Programación Lineal (*LP*, del inglés *Linear Programming*). Este algoritmo destaca por su enfoque geométrico: explora sistemáticamente los vértices (puntos extremos) del poliedro factible definido por las restricciones del problema. Dado que, por el Teorema Fundamental de la Programación Lineal, si existe una solución óptima finita, esta se encuentra en uno de los vértices, *Simplex* garantiza encontrar el óptimo global mediante un proceso iterativo de mejora de la función objetivo.

A nivel general, el propósito de *Simplex* es trasladarse de un vértice a otro adyacente que ofrezca un mejor (o igual) valor de la función objetivo, hasta que no sea posible realizar más mejoras. Matemáticamente, esto se traduce en operaciones de álgebra lineal sobre un sistema de ecuaciones.

Para describir su funcionamiento paso a paso, consideremos un problema *LP* en su forma estándar. El objetivo es minimizar una función lineal sujeta a restricciones de igualdad y no negatividad:

$$\begin{aligned} \text{Minimizar} \quad & z = \mathbf{c}^T \mathbf{x} \\ \text{Sujeto a:} \quad & A\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned}$$

Donde $\mathbf{x} \in \mathbb{R}^n$ es el vector de variables, $A \in \mathbb{R}^{m \times n}$ es la matriz de coeficientes (con $m < n$ y rango completo de filas), $\mathbf{c} \in \mathbb{R}^n$ es el vector de costos y $\mathbf{b} \in \mathbb{R}^m$ es el vector de recursos (se asume $\mathbf{b} \geq 0$).

El algoritmo opera mediante la partición de la matriz A y el vector $\mathbf{x}$ basándose en el concepto de **base**. Una base B es una submatriz cuadrada $m \times m$ formada por columnas linealmente

independientes de A. Las columnas restantes forman la matriz no básica N. De esta forma, las variables se dividen en básicas ($\mathbf{x}_B$) y no básicas ($\mathbf{x}_N$), permitiendo reescribir la restricción $A\mathbf{x} = \mathbf{b}$ como:

$$B\mathbf{x}_B + N\mathbf{x}_N = \mathbf{b}$$

El procedimiento iterativo de *Simplex* se desarrolla en los siguientes pasos:

1. **Determinación de la Solución Básica Factible (SBF):** En cada iteración, se fija $\mathbf{x}_N = 0$. Dado que B es invertible, las variables básicas se determinan como $\mathbf{x}_B = B^{-1}\mathbf{b}$. Si $\mathbf{x}_B \geq 0$, la solución es factible y corresponde a un vértice del poliedro.

2. **Cálculo de Costos Reducidos (Criterio de Optimalidad):** Para decidir si es posible mejorar la solución actual, se analiza el impacto de introducir una variable no básica en la base. Esto se realiza mediante el vector de costos reducidos $\bar{\mathbf{c}}_N$, derivado de la función objetivo expresada en términos de las variables no básicas:

   $$z = \mathbf{c}_B^T B^{-1}\mathbf{b} + (\mathbf{c}_N^T - \mathbf{c}_B^T B^{-1} N)\mathbf{x}_N$$

   El término entre paréntesis corresponde a los costos reducidos: $\bar{\mathbf{c}}_N^T = \mathbf{c}_N^T - \mathbf{c}_B^T B^{-1} N$.

   - Si todos los componentes de $\bar{\mathbf{c}}_N$ son no negativos ($\bar{c}_j \geq 0, \forall j$), no es posible reducir más el costo z incrementando ninguna variable x_j (dado que $x_j \geq 0$). Por lo tanto, la solución actual es **óptima**.
   - Si existe algún $\bar{c}_j < 0$, la variable correspondiente x_j puede entrar a la base para reducir el valor de la función objetivo.

3. **Selección de la Variable de Entrada:** Se selecciona una variable no básica x_j con costo reducido negativo (comúnmente la más negativa, siguiendo la regla de Dantzig) para entrar a la base. Esta variable aumentará su valor desde 0.

4. **Test de la Razón (Variable de Salida):** Al aumentar el valor de la variable de entrada x_j, las variables básicas actuales deben ajustarse para mantener la igualdad $A\mathbf{x} = \mathbf{b}$. Esto podría violar la no negatividad ($\mathbf{x}_B \geq 0$). Para evitarlo, se calcula cuánto puede

aumentar x_j antes de que una variable básica se vuelva cero. Se define la dirección de cambio $\mathbf{d} = B^{-1}A_j$ y se aplica el criterio de la razón mínima:

$$\theta^* = \min_i \left\{ \frac{(\mathbf{x}_B)_i}{d_i} : d_i > 0 \right\}$$

La variable básica que alcanza este mínimo debe salir de la base (volverse 0) para mantener la factibilidad. Si todos los $d_i \leq 0$, el problema es **no acotado**.

5. **Actualización (Pivoteo):** Se actualizan las matrices B y N intercambiando la columna de la variable entrante con la de la variable saliente. Se recalculan B^{-1} (o se actualiza la factorización) y se repite el proceso desde el paso 2.

Aunque en el peor de los casos la complejidad de *Simplex* puede ser exponencial, en la práctica ha demostrado ser extremadamente eficiente para una gran variedad de problemas reales, sirviendo como base para algoritmos más complejos en optimización entera y mixta.

Branch and Bound

El paradigma *Branch and Bound* (Ramificación y Acotamiento), formalizado por Ailsa H. Land y Alison G. Doig en 1960 [79], surgió como una respuesta necesaria a las limitaciones del método *Simplex* para manejar variables discretas. Mientras que la Programación Lineal asume divisibilidad continua, muchos problemas reales exigen decisiones binarias o enteras. Este método aborda la naturaleza combinatoria de la Programación Entera (*IP*) o Mixta (*MIP*) mediante una estrategia de "divide y vencerás", garantizando la obtención del óptimo global sin necesidad de enumerar explícitamente todas las soluciones posibles.

El propósito general del algoritmo es resolver problemas de la forma:

$$\begin{aligned} z^* = \min \quad & \mathbf{c}^T\mathbf{x} \\ \text{Sujeto a:} \quad & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq 0 \\ & x_j \in \mathbb{Z}, \quad \forall j \in I \end{aligned}$$

Donde I es el subconjunto de índices de variables que deben tomar valores enteros.

El funcionamiento de *Branch and Bound* se basa en la construcción implícita de un árbol de búsqueda, donde cada nodo representa un subproblema con restricciones adicionales. El proceso se articula en tres componentes fundamentales: relajación, ramificación y poda.

1. **Relajación Lineal y Acotamiento (*Bounding*):** El primer paso en cualquier nodo del árbol es resolver la *Relajación Lineal* del problema. Esto consiste en ignorar las restricciones de integralidad ($x_j \in \mathbb{Z}$) y resolver el problema como si fuera un *LP* continuo (usando *Simplex*).

   Sea z_{LP} el valor de la función objetivo de la relajación en un nodo específico y z_{best} el valor de la mejor solución entera encontrada hasta el momento (conocida como *incumbente*). En un problema de minimización:

   - z_{LP} proporciona una **Cota Inferior** (*Lower Bound*) para ese nodo y todos sus descendientes. No es posible encontrar una solución entera mejor que la solución óptima continua en esa región.
   - z_{best} actúa como una **Cota Superior Global**.

2. **Criterios de Poda (*Fathoming*):** Antes de seguir dividiendo el problema, se analiza si vale la pena explorar la rama actual. Un nodo se "poda" (se descarta) si se cumple alguna de estas condiciones:

   - **Infactibilidad:** La relajación lineal no tiene solución factible (la región geométrica es vacía).
   - **Optimalidad por Integralidad:** La solución de la relajación lineal cumple casualmente con todas las restricciones de integralidad. Si su valor es mejor que z_{best}, se actualiza la incumbente.
   - **Poda por Cota:** El valor de la relajación z_{LP} es peor (mayor, en minimización) o igual que la mejor solución entera conocida ($z_{LP} \geq z_{best}$). Esto implica que ninguna solución entera derivada de este nodo podrá superar a la que ya se tiene.

3. **Ramificación (*Branching*):** Si el nodo no se poda, significa que la solución de la relajación es fraccionaria para alguna variable x_j que debería ser entera (con valor

$x_j^* \notin \mathbb{Z}$). Para resolver esto, el problema se divide en dos subproblemas mutuamente excluyentes, imponiendo nuevas restricciones sobre x_j:

$$S_1 = S \cap \{x_j \leq \lfloor x_j^* \rfloor\}, \quad S_2 = S \cap \{x_j \geq \lceil x_j^* \rceil\}$$

Por ejemplo, si en la relajación obtenemos $x_j = 3.7$, se crean dos ramas: una con la restricción $x_j \leq 3$ y otra con $x_j \geq 4$. Esto elimina la región fraccionaria inválida sin perder ninguna solución entera potencial.

El algoritmo itera seleccionando nodos abiertos y aplicando estos pasos hasta que no queden nodos por explorar. En ese punto, la solución *incumbente* se certifica como el óptimo global del problema original.

Cutting Planes

Por su parte, los métodos de *Cutting Planes*, introducidos formalmente por Ralph E. Gomory a finales de la década de 1950 [62], se basan en el fortalecimiento iterativo de las relajaciones lineales de un problema entero. A diferencia de *Branch and Bound*, que divide el espacio de búsqueda, este enfoque refina la aproximación del poliedro factible agregando desigualdades lineales válidas, denominadas planos de corte o *cuts*, que violan la solución fraccionaria actual pero satisfacen todas las soluciones enteras factibles.

El propósito general es aproximar progresivamente la envoltura convexa (*Convex Hull*) de los puntos enteros, permitiendo que el algoritmo *Simplex* converja hacia una solución entera de manera natural.

El procedimiento general para utilizar *Cutting Planes* es el siguiente:

1. **Resolver Relajación:** Se resuelve la relajación lineal del problema ($\mathbf{x} \in \mathbb{R}^n$). Sea $\mathbf{x}^*$ la solución óptima obtenida.
2. **Verificación de Integralidad:** Si $\mathbf{x}^*$ tiene componentes enteras para todas las variables requeridas, entonces $\mathbf{x}^*$ es óptimo para el problema original y el algoritmo termina.

3. **Problema de Separación:** Si $\mathbf{x}^*$ es fraccionario, se debe identificar una desigualdad lineal (corte) $\pi^T \mathbf{x} \leq \pi_0$ tal que:
    - Sea válida para todos los puntos enteros factibles.
    - **Corte** o separe la solución actual: $\pi^T \mathbf{x}^* > \pi_0$.
4. **Actualización:** Se agrega este corte a la matriz de restricciones y se vuelve al paso 1 (generalmente utilizando el método Simplex Dual para aprovechar la solución previa).

Para ejemplificar la generación matemática de un corte, consideremos el método de los **Cortes Fraccionales de Gomory**. Supongamos que, tras resolver la relajación con *Simplex*, la ecuación asociada a una variable básica x_i en la tabla óptima es:

$$x_i + \sum_{j \in N} \bar{a}_{ij} x_j = \bar{b}_i$$

Donde $\bar{b}_i$ es fraccionario (lo que impide que x_i sea entero). Cada coeficiente y el lado derecho se pueden descomponer en una parte entera y una parte fraccionaria no negativa (ej. $3.7 = 3 + 0.7$), tal que $\bar{a}_{ij} = \lfloor \bar{a}_{ij} \rfloor + f_{ij}$ y $\bar{b}_i = \lfloor \bar{b}_i \rfloor + f_i$.

Reescribiendo la ecuación y agrupando los términos enteros a la izquierda y los fraccionarios a la derecha, Gomory demostró que se puede derivar la siguiente desigualdad válida para cualquier solución entera:

$$\sum_{j \in N} f_{ij} x_j \geq f_i$$

Esta inecuación es el *corte*. Dado que las variables no básicas x_j son cero en la solución actual y $f_i > 0$, la solución fraccionaria actual viola esta restricción, forzando al solver a buscar una nueva solución más cercana a los enteros en la siguiente iteración.

La combinación de este enfoque con *Branch and Bound* dio origen al método *Branch and Cut* [80], el estándar actual en la industria. Estos algoritmos se implementan de forma integrada en los principales solvers comerciales de optimización, tales como CPLEX [81], Gurobi [82] o SCIP [83]. Estas herramientas combinan dinámicamente técnicas de ramificación, generación automática de cortes (como cortes de Gomory, MIR, o *Knapsack covers*) y heurísticas de primalidad para acelerar la convergencia en problemas de gran escala.

2.3.2. Métodos de Resolución Incompleta

A medida que aumenta la dimensionalidad y la complejidad de las restricciones en los problemas de optimización, los métodos exactos descritos anteriormente suelen volverse computacionalmente intratables. Esto es particularmente evidente en problemas clasificados como *NP-Hard* [84], donde el tiempo requerido para garantizar la optimalidad global crece exponencialmente con el tamaño de la entrada. Frente a esta barrera, los métodos de resolución incompleta emergen como una alternativa pragmática, priorizando la obtención de soluciones de buena calidad ("cuasi-óptimas") en tiempos de ejecución acotados, renunciando a la prueba formal de optimalidad [85].

Dentro de esta categoría, es fundamental distinguir entre las *heurísticas* y las *metaheurísticas*. Mientras que las heurísticas suelen ser algoritmos diseñados *ad-hoc* para explotar la estructura específica de un problema particular, las metaheurísticas se definen como estrategias maestras o marcos algorítmicos generales aplicables a una amplia variedad de problemas de optimización [86]. Estas últimas orquestan la interacción entre procedimientos de mejora local y estrategias de alto nivel para escapar de óptimos locales.

El éxito de estos métodos radica en su capacidad para gestionar el equilibrio (*trade-off*) entre dos fuerzas contrapuestas:

- **Intensificación (Explotación):** La capacidad de investigar en profundidad el vecindario de una buena solución para encontrar mejoras locales.
- **Diversificación (Exploración):** La capacidad de investigar regiones no visitadas del espacio de búsqueda para evitar quedar atrapado en óptimos locales y descubrir nuevas zonas prometedoras.

Tal como señalan Talbi [87] y Glover [88], las metaheurísticas modernas se clasifican comúnmente en dos grandes familias según su mecanismo de búsqueda: métodos basados en trayectoria (*Single-solution based*), que evolucionan una única solución a través del tiempo, y métodos poblacionales (*Population-based*), que utilizan un conjunto de soluciones que interactúan y evolucionan simultáneamente.

Metaheurísticas Trayectoriales

Las metaheurísticas trayectoriales (o basadas en solución única) fundamentan su búsqueda en la evolución de una sola solución candidata s a lo largo del tiempo. El proceso genera una secuencia o "trayectoria" de soluciones $s_0, s_1, \ldots, s_{final}$ en el espacio de búsqueda, transitando desde el estado actual hacia uno nuevo seleccionado dentro de su vecindario $N(s)$ [87].

El concepto central en estos métodos es la definición del operador de vecindad, el cual determina qué movimientos son permitidos y cómo se transforma una solución en otra. A diferencia de los métodos de búsqueda local simple, que aceptan exclusivamente movimientos que mejoran la función objetivo (enfoque *greedy*), las metaheurísticas trayectoriales avanzadas incorporan mecanismos para aceptar, bajo ciertos criterios, soluciones de menor calidad. Esta capacidad es crítica para evitar la *convergencia prematura* en óptimos locales y permitir la exploración de regiones más amplias del espacio de soluciones [89, 88].

A continuación, se describen algoritmos representativos de esta categoría, comenzando por el enfoque básico de búsqueda local hasta técnicas más sofisticadas de escape y diversificación: *Hill Climbing*, *Simulated Annealing*, *Tabu Search* e *Iterated Local Search*.

Hill Climbing (HC)

El algoritmo *Hill Climbing* (o Ascenso de Colina) representa la estrategia más intuitiva y básica dentro de las metaheurísticas de trayectoria. Conceptualizado a menudo como la analogía de un escalador que intenta alcanzar la cima de una montaña en medio de una niebla densa, este método toma decisiones basadas únicamente en la información local inmediata, sin tener perspectiva del panorama global [90].

Su funcionamiento es estrictamente *greedy* (codicioso): a partir de una solución inicial, el algoritmo explora iterativamente su vecindario y acepta un movimiento si y solo si este mejora el valor de la función objetivo. Existen dos variantes principales en la estrategia de selección:

- **Steepest Ascent (Mejor Mejora):** Evalúa *todos* los vecinos posibles y selecciona aquel que ofrece la mayor reducción de costo (en minimización).
- **First Ascent (Primera Mejora):** Selecciona el *primer* vecino encontrado que sea mejor que la solución actual, lo que suele reducir el tiempo de cómputo por iteración a costa de una convergencia potencialmente menos directa.

Matemáticamente, para un problema de minimización con función objetivo $f(s)$, el procedimiento paso a paso es el siguiente:

1. **Inicialización:** Se genera una solución inicial $s \in S$ (generalmente de forma aleatoria).
2. **Evaluación del Vecindario:** Se genera un conjunto de soluciones candidatas $s' \in N(s)$, donde $N(s)$ representa el vecindario de s.
3. **Selección y Movimiento:**
   - Si existe un $s' \in N(s)$ tal que $f(s') < f(s)$, se actualiza la solución actual: $s \leftarrow s'$.
   - Si $f(s') \geq f(s)$ para todo $s' \in N(s)$, entonces s es un **óptimo local** y el algoritmo se detiene.
4. **Iteración:** Se repiten los pasos 2 y 3 hasta cumplir el criterio de parada (no encontrar mejoras).

A pesar de su simplicidad y bajo costo computacional, *Hill Climbing* presenta una limitación crítica: su incapacidad para aceptar movimientos de deterioro (donde $f(s') > f(s)$). Esto provoca que el algoritmo quede atrapado fácilmente en **óptimos locales** (soluciones mejores que sus vecinas pero peores que el óptimo global) o en **mesetas** (regiones planas donde el gradiente de mejora es nulo), impidiendo la exploración hacia zonas más prometedoras del espacio de búsqueda [91]. Esta debilidad estructural motivó el desarrollo de técnicas como *Simulated Annealing*, que incorporan mecanismos estocásticos para escapar de estas trampas.

Simulated Annealing (SA)

El algoritmo de *Simulated Annealing* (Recocido Simulado), propuesto independientemente por Kirkpatrick, Gelatt y Vecchi en 1983 [67] y por Černý en 1985 [92], representa una de las primeras adaptaciones exitosas de la física estadística a la optimización combinatoria. Su diseño está inspirado en el proceso de recocido en metalurgia, técnica que consiste en calentar un material a altas temperaturas y enfriarlo lentamente para permitir que sus átomos se reorganicen en una estructura cristalina de mínima energía, libre de defectos.

En la analogía con la optimización:

- Los estados físicos de los átomos corresponden a las **soluciones** del problema.
- La energía interna del sistema equivale al valor de la **función objetivo** (costo).
- La temperatura es un **parámetro de control** T que regula la probabilidad de aceptar cambios desfavorables.

El propósito fundamental de *SA* es superar la limitación de *Hill Climbing* permitiendo, de manera controlada, movimientos que empeoren la función objetivo. Esto otorga al algoritmo la capacidad de escapar de óptimos locales. La decisión de aceptar una solución peor se rige por el criterio de Metropolis [93].

El procedimiento paso a paso para un problema de minimización es el siguiente:

1. **Inicialización:** Se selecciona una solución inicial s, una temperatura inicial elevada T_0 y un esquema de enfriamiento.
2. **Generación de Vecino:** Se selecciona aleatoriamente una solución candidata $s' \in N(s)$.
3. **Evaluación y Criterio de Aceptación:** Se calcula la variación de costo $\Delta = f(s') - f(s)$.
    - Si $\Delta < 0$ (la nueva solución es mejor), se acepta automáticamente: $s \leftarrow s'$.

- Si $\Delta \geq 0$ (la nueva solución es peor), se acepta con una probabilidad P dada por la distribución de Boltzmann:

$$P(\text{aceptar}) = e^{-\frac{\Delta}{T}}$$

Para implementar esto, se genera un número aleatorio $r \sim U[0, 1]$. Si $r < P$, se acepta el movimiento de deterioro; de lo contrario, se rechaza.

4. **Enfriamiento:** Tras un número determinado de iteraciones a temperatura constante, se reduce T siguiendo un esquema de enfriamiento (por ejemplo, geométrico: $T_{k+1} = \alpha T_k$, con $0.8 < \alpha < 0.99$).

5. **Terminación:** El proceso se detiene cuando la temperatura es cercana a cero (el sistema se "congela") y no se aceptan más cambios, convergiendo a una solución estable.

La dinámica del algoritmo cambia con el tiempo: a altas temperaturas, $e^{-\Delta/T}$ es cercano a 1, comportándose casi como una búsqueda aleatoria (*Random Walk*) que explora ampliamente el espacio. A medida que T disminuye, la probabilidad de aceptar deterioros cae drásticamente, y el algoritmo se comporta progresivamente como un *Hill Climbing*, intensificando la búsqueda en la región prometedora encontrada.

Tabu Search (TS)

Propuesto formalmente por Fred Glover en 1986 [94] y detallado en su obra seminal de 1989 [95], *Tabu Search* (Búsqueda Tabú) representa un cambio de paradigma respecto a los métodos estocásticos como *Simulated Annealing*. En lugar de confiar en el azar para escapar de óptimos locales, *TS* emplea una estrategia determinista basada en el uso explícito de **memoria adaptativa**.

El propósito central del algoritmo es imitar la inteligencia humana: recordar el pasado inmediato para no cometer los mismos errores (o ciclos) y utilizar el conocimiento acumulado para diversificar la búsqueda. Para lograr esto, el método permite movimientos hacia soluciones que empeoran la función objetivo, pero restringe volver a visitar soluciones recientes mediante una estructura de memoria a corto plazo denominada **Lista Tabú**.

El procedimiento general para un problema de minimización es el siguiente:

1. **Inicialización:** Se genera una solución inicial s y se inicializa la Lista Tabú L como vacía. Se define un tamaño máximo para la lista (tenencia tabú).
2. **Generación de Vecindario y Filtrado:** Se genera el conjunto de vecinos $N(s)$. De este conjunto, se identifican los vecinos "admisibles". Un vecino $s' \in N(s)$ es admisible si:
   - El movimiento que genera s' **no** está en la Lista Tabú.
   - O bien, el movimiento es tabú pero cumple con el **Criterio de Aspiración**.
3. **Criterio de Aspiración:** Este mecanismo permite anular el estado tabú de un movimiento si la solución resultante es de calidad excepcional. La regla más común es permitir el movimiento tabú si la solución s' es mejor que la mejor solución encontrada en toda la búsqueda histórica ($f(s') < f(s_{best})$).
4. **Selección (Estrategia *Best Improvement*):** Se evalúan todos los vecinos admisibles y se selecciona el mejor de ellos (s_{next}), **incluso si su valor es peor** que el de la solución actual ($f(s_{next}) > f(s)$).
5. **Actualización:**
   - Se mueve a la nueva solución: $s \leftarrow s_{next}$.
   - Se actualiza la memoria: El "movimiento" inverso que llevaría de regreso a la solución anterior se agrega a la Lista Tabú para prevenir el ciclado. Si la lista está llena, se elimina el elemento más antiguo (política FIFO).
6. **Iteración:** Se repiten los pasos 2 a 5 hasta cumplir un criterio de parada (número de iteraciones o tiempo límite).

La eficacia de *Tabu Search* radica en que la Lista Tabú fuerza al algoritmo a explorar nuevas direcciones cuando se encuentra atrapado en un valle (óptimo local), ya que le prohíbe retroceder. Versiones más avanzadas incorporan **memoria a largo plazo** para estrategias de *intensificación* (volver a regiones prometedoras históricas) y *diversificación* (visitar regiones inexploradas basadas en la frecuencia de movimientos) [96].

Iterated Local Search (ILS)

La metaheurística *Iterated Local Search* (Búsqueda Local Iterada), formalizada en el marco moderno por Lourenço, Martin y Stützle en 2003 [97], propone un enfoque jerárquico para la optimización. Su premisa fundamental es que las soluciones de alta calidad suelen encontrarse agrupadas en regiones específicas del espacio de búsqueda o comparten componentes estructurales comunes.

A diferencia de los métodos anteriores que operan directamente sobre el espacio de soluciones original, *ILS* restringe su búsqueda al **espacio reducido de los óptimos locales**. La estrategia consiste en encadenar ejecuciones de un algoritmo de búsqueda local, utilizando una perturbación entre ellas para saltar de una cuenca de atracción a otra, evitando así el estancamiento y preservando al mismo tiempo la "buena estructura" encontrada en iteraciones previas.

El algoritmo se compone de cuatro módulos básicos:

1. **Generación de Solución Inicial:** Se crea una solución inicial s_0 (generalmente aleatoria).
2. **Búsqueda Local (Subsidiaria):** Se aplica un método de descenso (como *Hill Climbing*) a s_0 hasta alcanzar un óptimo local s^*.
3. **Perturbación (*Perturbation*):** Este es el paso crítico que distingue a *ILS* de un reinicio aleatorio (*Random Restart*). Se aplica una modificación no determinista a la solución óptima actual (s^*) para generar una solución intermedia s'.
    - La perturbación debe ser lo suficientemente fuerte para sacar al sistema de la cuenca de atracción actual, pero lo suficientemente débil para no destruir la información estructural valiosa obtenida en la búsqueda anterior.
4. **Búsqueda Local Post-Perturbación:** Se aplica nuevamente el algoritmo de búsqueda local partiendo de s' hasta encontrar un nuevo óptimo local $s^{*'}$.
5. **Criterio de Aceptación:** Se decide si el nuevo óptimo local $s^{*'}$ reemplaza al anterior

s^* como base para la siguiente iteración.

$$s^* \leftarrow \text{Aceptación}(s^*, s^{*'})$$

Este criterio puede ser estricto (solo aceptar si mejora, $f(s^{*'}) < f(s^*)$) o probabilístico (similar a *Simulated Annealing*) para favorecer la diversificación.

El ciclo de Perturbación → Búsqueda Local → Aceptación se repite iterativamente. La potencia de *ILS* radica en su simplicidad modular: permite reutilizar algoritmos de búsqueda local rápidos y eficientes, delegando la complejidad de la exploración global al diseño de una buena estrategia de perturbación.

Algoritmos Evolutivos

Los Algoritmos Evolutivos (*EA*, del inglés *Evolutionary Algorithms*) constituyen el subconjunto más prominente de las metaheurísticas poblacionales. A diferencia de los métodos de trayectoria, que refinan una única solución, los *EA* operan sobre un conjunto de soluciones candidatas denominado *población*, las cuales evolucionan simultáneamente a través de un proceso iterativo de "generaciones" [98].

El funcionamiento de estas técnicas se inspira en los principios de la evolución biológica neodarwiniana: la selección natural y la herencia genética. Conceptualmente, los *EA* distinguen entre el espacio de soluciones (fenotipo) y el espacio de representación (genotipo). El proceso de búsqueda se guía por la premisa de la "supervivencia del más apto", donde los individuos con mejor desempeño en la función objetivo (*fitness*) tienen mayor probabilidad de sobrevivir y transmitir su información genética a la siguiente generación a través de operadores de variación estocástica [99]:

- **Recombinación (Cruce):** Mezcla la información de dos o más padres para explotar las mejores características de cada uno.
- **Mutación:** Introduce cambios aleatorios en los individuos para mantener la diversidad genética y explorar nuevas regiones del espacio de búsqueda.

- **Selección:** Filtra la población para determinar qué individuos pasarán a la siguiente iteración.

Una ventaja crítica de este enfoque es el *paralelismo implícito*: al evaluar múltiples soluciones simultáneamente, el algoritmo muestrea el hiperplano del espacio de búsqueda de manera mucho más eficiente, reduciendo drásticamente el riesgo de estancamiento en óptimos locales [71].

Históricamente, la Computación Evolutiva se ha dividido en cuatro corrientes principales que, aunque hoy convergen, tuvieron orígenes independientes: Algoritmos Genéticos (*Genetic Algorithms*), Programación Evolutiva (*Evolutionary Programming*), Estrategias Evolutivas (*Evolution Strategies*) y Programación Genética (*Genetic Programming*).

Genetic Algorithms (GA)

Formalizados por John Holland en la Universidad de Michigan en 1975 [71], y popularizados posteriormente por David Goldberg [100], los Algoritmos Genéticos (GA) representan la rama más conocida de la computación evolutiva. A diferencia de las Estrategias Evolutivas, que se enfocan en la mutación de parámetros reales, los GA clásicos enfatizan el uso de operadores de **cruce** (*crossover*) sobre representaciones discretas (originalmente cadenas binarias) para recombinar "bloques de construcción" de soluciones de alta calidad.

El fundamento teórico de los *GA* descansa en el *Teorema de los Esquemas* (Schema Theorem), el cual postula que esquemas (patrones de bits) cortos, de bajo orden y con un *fitness* superior al promedio, aumentan exponencialmente su presencia en la población a través de las generaciones.

El ciclo de un Algoritmo Genético Canónico se estructura de la siguiente manera:

1. **Codificación (Genotipo):** El problema se traduce a una representación lineal (cromosoma). Por ejemplo, en un problema de la mochila (*Knapsack Problem*), un vector binario donde el bit *i*-ésimo indica si el objeto se incluye (1) o no (0).

2. **Evaluación (Fenotipo):** Se decodifica el cromosoma y se evalúa su función de aptitud (*fitness*).

3. **Selección Proporcional:** Se seleccionan individuos para la reproducción. El método clásico es la *Ruleta* (Roulette Wheel Selection), donde la probabilidad de elegir a un individuo i es proporcional a su *fitness* relativo $p_i = f_i / \sum f_j$. Esto favorece a los mejores, pero da oportunidad a los peores para mantener diversidad.

4. **Cruce (Crossover):** Es el operador principal de búsqueda en los GA. Con una probabilidad P_c (típicamente alta, $0.6 - 0.9$), se toman dos padres y se intercambian segmentos de su código genético (cruce de un punto, dos puntos o uniforme) para crear descendencia que comparta características de ambos.

5. **Mutación:** Es un operador secundario diseñado para recuperar material genético perdido. Con una probabilidad muy baja P_m (ej. 0.01 por bit), se invierte aleatoriamente el valor de un gen. Esto previene la convergencia prematura en un óptimo local.

6. **Reemplazo:** La nueva población sustituye a la anterior (generacional) o se mezcla con ella (estado estacionario).

La versatilidad de este esquema ha dado lugar a una familia de métodos híbridos avanzados:

- **Algoritmos Meméticos (MA):** Acuñados por Moscato [101], combinan la evolución global del GA con un procedimiento de búsqueda local (como *Hill Climbing*) aplicado a cada individuo en cada generación ("aprendizaje de vida"). Esto mejora drásticamente la capacidad de explotación.

- **Algoritmos Genéticos Adaptativos:** Ajustan dinámicamente P_c y P_m durante la ejecución. Si la población pierde diversidad, la tasa de mutación aumenta automáticamente; si converge, disminuye [102].

- **GA Multiobjetivo (MOEA):** Como el NSGA-II [103], que utilizan frentes de Pareto para optimizar múltiples funciones objetivo en conflicto simultáneamente.

Evolutionary Programming (EP)

Introducida por Lawrence J. Fogel, Alvin J. Owens y Michael J. Walsh en 1966 [69], *Evolutionary Programming* (*EP*) fue concebida originalmente no como un optimizador de funciones, sino como un enfoque de Inteligencia Artificial para crear máquinas capaces de predecir su entorno. Su hipótesis central es que el comportamiento inteligente es una propiedad emergente de la evolución de una especie, no del diseño de un individuo.

A diferencia de los *Genetic Algorithms*, que simulan la evolución a nivel genético (manipulando cromosomas mediante cruce), *EP* simula la evolución a nivel **fenotípico**. Esto implica dos diferencias técnicas fundamentales:

1. **Ausencia de Cruce:** Dado que se evoluciona el comportamiento observable de la especie, no existe el operador de recombinación sexual. La evolución es estrictamente asexual, impulsada exclusivamente por la **mutación**.
2. **Representación Flexible:** No requiere codificación binaria. Las soluciones se representan de la forma más natural para el problema (originalmente Autómatas Finitos, hoy comúnmente vectores de números reales para optimización continua).

En su formulación original, las soluciones eran representaciones de autómatas finitos, y la evolución ocurría mediante mutaciones aleatorias sobre sus parámetros, sin utilizar operadores de cruce. Cada generación se evaluaba según un criterio de desempeño definido para la tarea, y los mejores individuos sobrevivían para producir descendencia.

EP se caracteriza por su simplicidad conceptual, su independencia de la estructura del problema y su orientación hacia la optimización continua o de parámetros más que hacia problemas combinatorios.

El procedimiento canónico para un problema de optimización continua (mín $f(\mathbf{x}), \mathbf{x} \in \mathbb{R}^n$) es el siguiente:

1. **Inicialización:** Se genera una población de μ individuos padres. Cada individuo se define como un par $(\mathbf{x}, \boldsymbol{\sigma})$, donde $\mathbf{x}$ es el vector de variables de decisión y $\boldsymbol{\sigma}$ es un

vector de parámetros de estrategia (desviaciones estándar para la mutación).

2. **Mutación (Perturbación Gaussiana):** Cada padre genera un descendiente mediante mutación. Lo distintivo de *EP* es la **auto-adaptación**: los parámetros de mutación también evolucionan. Las nuevas variables $\mathbf{x}'$ se calculan como:

$$x_i' = x_i + \sigma_i \cdot N_i(0, 1)$$

   Donde $N_i(0, 1)$ es una variable aleatoria con distribución normal estándar. Esto permite que el algoritmo "aprenda" qué tan grandes deben ser los pasos de búsqueda en diferentes etapas.

3. **Evaluación:** Se calcula el *fitness* $f(\mathbf{x}')$ de cada descendiente.

4. **Selección (Torneo Estocástico):** A diferencia de las Estrategias Evolutivas que usan selección determinista, *EP* emplea un torneo probabilístico. Cada individuo (padres e hijos) compite contra q oponentes elegidos al azar de la población. Se asigna una puntuación basada en cuántas "victorias" (menor costo) obtuvo.

5. **Supervivencia:** Los μ individuos con mayor puntuación de victorias pasan a formar la siguiente generación.

Aunque inicialmente se diseñó para la evolución de lógica de control, *EP* ha demostrado ser altamente eficaz en optimización paramétrica numérica debido a su capacidad de adaptar la "forma" de la búsqueda mediante la evolución de las varianzas $\boldsymbol{\sigma}$ [104].

Evolution Strategies (ES)

Desarrolladas inicialmente en la Universidad Técnica de Berlín en 1964 por Ingo Rechenberg y Hans-Paul Schwefel [70, 105], las Estrategias Evolutivas nacieron con un enfoque pragmático para resolver problemas de ingeniería experimental (como la optimización de la forma de placas articuladas para minimizar el arrastre en un túnel de viento).

A diferencia de los *Genetic Algorithms* que enfatizan la recombinación, o de la *Evolutionary Programming* que utiliza selección estocástica, las *ES* se distinguen por utilizar una **selección**

determinista basada estrictamente en el rango de *fitness* y por operar de manera nativa con vectores de números reales.

La notación estándar en *ES* describe el algoritmo mediante los parámetros μ (tamaño de la población de padres) y λ (número de descendientes generados en cada generación), definiendo dos estrategias de selección canónicas [106]:

- **Estrategia** (μ, λ)**:** Los μ padres generan λ hijos (con $\lambda > \mu$). La selección de la nueva generación se realiza eligiendo a los μ mejores individuos *exclusivamente* del conjunto de hijos. Los padres se descartan, lo que evita el estancamiento en óptimos locales antiguos pero arriesga perder la mejor solución.
- **Estrategia** $(\mu + \lambda)$**:** La selección elige a los μ mejores individuos del conjunto unido de padres e hijos. Esto garantiza elitismo (la mejor solución nunca se pierde), pero puede provocar una convergencia prematura.

El ciclo algorítmico típico es:

1. **Recombinación:** A diferencia de *EP*, las *ES* modernas utilizan recombinación. Se generan λ descendientes combinando los parámetros de ρ padres (usualmente mediante promedio o intercambio discreto de componentes).
2. **Mutación y Auto-adaptación:** Se aplica una perturbación gaussiana al descendiente. La contribución clave de Rechenberg fue la **Regla del 1/5** (ajustar la varianza para mantener una tasa de aceptación del 20%) y posteriormente la auto-adaptación de la matriz de covarianza. Esto permite que la "nube" de mutación rote y se escale para alinearse con la topología de la función objetivo.
3. **Evaluación y Selección:** Se evalúa el *fitness* y se seleccionan determinísticamente los μ mejores sobrevivientes para la siguiente iteración.

Hoy en día, la variante más avanzada de esta familia es el algoritmo **CMA-ES** (*Covariance Matrix Adaptation Evolution Strategy*) [107]. CMA-ES adapta una matriz de covarianza completa que representa las dependencias entre variables, convirtiéndolo en el estándar de

oro para la optimización continua de caja negra en problemas no separables y mal condicionados.

Genetic Programming (GP)

Popularizado por John R. Koza con su obra fundamental en 1992 [108], *Genetic Programming* (*GP*) representa una extensión natural de los Algoritmos Genéticos hacia el dominio de la generación automática de código. Mientras que los *GA* optimizan un conjunto de parámetros para un modelo predefinido (longitud fija), *GP* tiene como objetivo descubrir tanto la estructura del modelo como sus parámetros simultáneamente (longitud y forma variable).

En *GP*, los individuos son programas informáticos ejecutables, representados tradicionalmente como **árboles de análisis sintáctico** (*parse trees*). Para construir estos árboles, se definen dos conjuntos primitivos:

- **Conjunto de Terminales (T):** Las hojas del árbol, que corresponden a variables del problema (ej. x, y) o constantes numéricas.
- **Conjunto de Funciones (F):** Los nodos internos, que operan sobre los terminales. Pueden ser aritméticos ($+, -, *, /$), lógicos (AND, OR), o condicionales (IF-THEN-ELSE).

Una condición necesaria en *GP* es la propiedad de **clausura** (*closure*): cada función en F debe ser capaz de procesar cualquier valor devuelto por cualquier otra función en F o cualquier terminal en T, garantizando que cualquier árbol generado aleatoriamente sea sintácticamente válido y ejecutable.

El ciclo evolutivo de *GP* sigue estos pasos:

1. **Inicialización (Ramped Half-and-Half):** Se genera una población aleatoria de árboles combinando terminales y funciones. Se suele utilizar el método "Ramped Half-and-Half" para asegurar variedad en la profundidad y forma de los árboles iniciales.
2. **Evaluación:** Cada programa (árbol) se ejecuta contra un conjunto de casos de prueba.

El *fitness* mide qué tan bien el programa resuelve el problema (por ejemplo, la suma de errores al cuadrado en un problema de regresión).

3. **Selección:** Se utilizan métodos estándar como el torneo para elegir los padres.
4. **Cruce de Subárboles (*Subtree Crossover*):** Es el operador principal. Se seleccionan nodos de cruce aleatorios en dos padres y se intercambian los subárboles que cuelgan de dichos nodos. Esto permite combinar la lógica de dos programas distintos para crear uno nuevo.
5. **Mutación:** Se selecciona un nodo aleatorio en un padre, se elimina el subárbol existente y se reemplaza por uno nuevo generado aleatoriamente.

Uno de los desafíos técnicos más importantes en *GP* es el fenómeno del **Bloat** (hinchazón): la tendencia de los árboles a crecer en tamaño descontroladamente durante la evolución sin mejorar significativamente el *fitness*, a menudo generando "código muerto" (introns) [109]. Para contrarrestar esto, las implementaciones modernas utilizan **presión de parsimonia** (penalizar árboles grandes en la función de *fitness*) para buscar soluciones que sean no solo precisas, sino también compactas e interpretables.

2.4. *Satisfiability Modulo Theories (SMT)*

El paradigma de ***Satisfiability Modulo Theories (SMT)*** [21] extiende los fundamentos de la *Satisfiability* proposicional (*SAT*), incorporando la capacidad de razonar sobre dominios más expresivos que el estrictamente booleano. En esencia, *SMT* busca determinar la satisfacibilidad de una fórmula lógica de primer orden respecto de una o más teorías, permitiendo así modelar problemas que involucran números enteros, reales, arreglos, cadenas de caracteres u otros tipos de datos definidos formalmente.

A diferencia de los problemas *SAT*, que operan exclusivamente sobre variables booleanas y operadores lógicos, los problemas *SMT* introducen un marco semántico adicional mediante la noción de **teorías**. Cada teoría define un conjunto de símbolos, funciones e interpretaciones sobre un dominio particular, por ejemplo, Enteros (*Ints*), Reales (*Reals*), Vectores de Bits

(*Bit-Vectors*) o Arreglos (*Arrays*), lo que permite capturar relaciones matemáticas complejas de forma directa y sin recurrir a codificaciones artificiales.

La estandarización del lenguaje y las teorías en *SMT* se encuentra formalizada en la versión 2.6 de la biblioteca SMT-LIB [110], la cual establece un conjunto de teorías elementales que pueden combinarse para conformar las denominadas lógicas *SMT*:

- *ArraysEx*: Arreglos funcionales con extensionalidad.
- *FixedSizeBitVectors*: Vectores de bits de tamaño arbitrario.
- *Core* o *SAT*: Teoría central de *SMT*, define los operadores booleanos básicos.
- *FloatingPoint*: Números punto flotante.
- *Ints*: Números enteros.
- *Reals*: Números reales.
- *Reals_Ints* Números reales y enteros.
- *Strings*: Cadenas de caracteres *unicode* y expresiones regulares.

A partir de las teorías anteriores es posible componer las *lógicas SMT* mediante la combinación de una o más teorías, tal como se observa en la Figura 2.2:

El proceso de resolución en *SMT* se estructura en torno al framework *DPLL(T)* [50], una extensión del algoritmo *Davis–Putnam–Logemann–Loveland (DPLL)* utilizado en problemas *SAT* [46, 112, 113]. Este framework permite combinar un motor proposicional (*SAT solver*) con uno o más *Theory Solvers* [114], módulos especializados en verificar la consistencia de las restricciones dentro de una teoría matemática particular, como por ejemplo, aritmética lineal, enteros, arreglos o bit-vectors. Mientras el *SAT solver* opera sobre la abstracción booleana del problema, los *Theory Solvers* interpretan los mismos literales en su dominio y son capaces de detectar conflictos, propagar nuevas conclusiones o confirmar la satisfacibilidad parcial dentro de su teoría. Esta cooperación entre niveles proposicional y teórico es lo

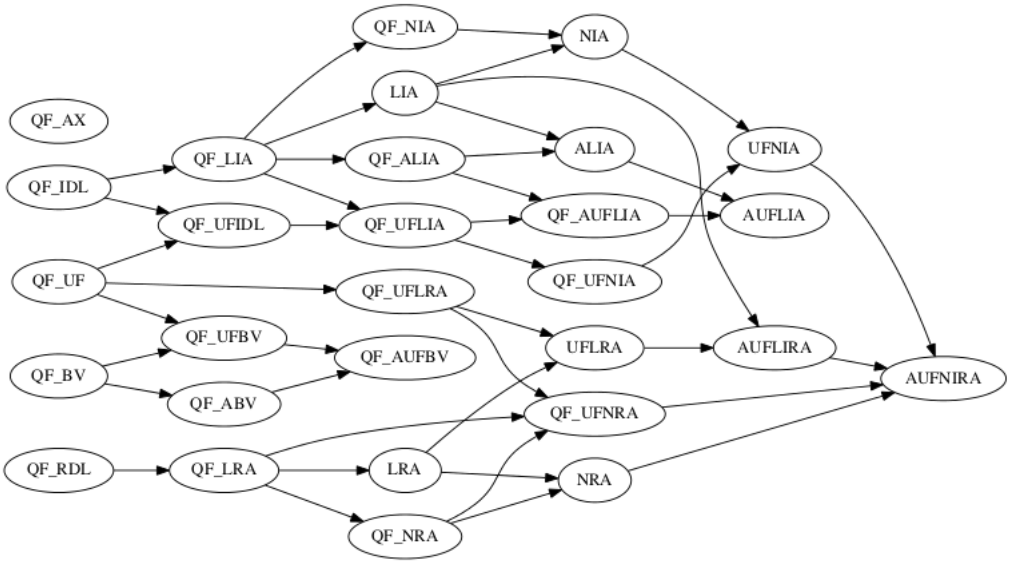


Figura 2.2: Diagrama con las lógicas *SMT* y la relación entre ellas [111]

.

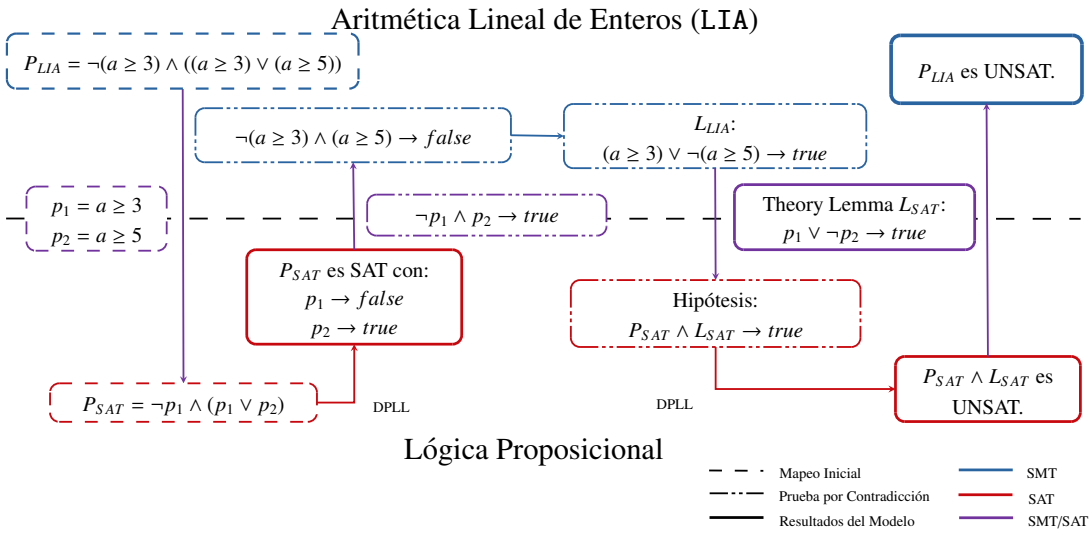


Figura 2.3: Ejemplo de resolución de una fórmula SMT mediante el framework DPLL(T) [46], donde se combinan lógica proposicional y teoría de Aritmética Lineal de Enteros (LIA). La figura ilustra el ciclo de búsqueda y prueba por contradicción al detectar insatisfacibilidad teórica [113].

que permite a *SMT* resolver problemas expresivos que combinan estructuras booleanas con restricciones de alto nivel.

La Figura 2.3 ilustra este proceso mediante un ejemplo que combina lógica proposicional con

aritmética lineal de enteros (LIA). Este mecanismo iterativo, basado en la interacción entre componentes proposicionales y teóricos, constituye el núcleo de los solvers SMT modernos, como Z3 [56], CVC5 [115] y Yices [116], el cuál puede resumirse de la siguiente manera:

1. Se define el modelo original, $\mathcal{P}_{LIA}$, con una o más lógicas de primer orden.
2. *SMT* separa el modelo entregado en sub-fórmulas únicas, y a cada una le asigna un predicado proposicional. Utilizando operadores booleanos, se construye la fórmula proposicional simplificada $\mathcal{P}_{\mathcal{SAT}}$, y a partir de esta se construye una tabla de verdad para determinar si la fórmula $\mathcal{P}_{\mathcal{SAT}}$ es `sat`. Si se determina que $\mathcal{P}_{\mathcal{SAT}}$ es `unsat`, entonces se concluye que el modelo original $\mathcal{P}_{LIA}$ también es `unsat` y el ciclo termina.
3. Si la fórmula $\mathcal{P}_{\mathcal{SAT}}$ es *sat*, se construye una nueva fórmula $\mathcal{P}_{\mathcal{L}}$, con las lógicas del modelo original, a partir de la combinación de predicados encontrados en la tabla de verdad de $\mathcal{P}_{\mathcal{SAT}}$, con la finalidad de determinar si los valores de verdad de los predicados siguen siendo consistente al ser evaluados en la lógica inicial.
4. A partir de las fórmulas $\mathcal{P}_{\mathcal{SAT}}$ y $\mathcal{P}_{\mathcal{L}}$ se plantea la siguiente hipótesis: $\mathcal{P}_{\mathcal{SAT}} \wedge \mathcal{P}_{\mathcal{L}} \rightarrow True$. Es decir, si ambas fórmulas son satisfacibles con la tabla de verdad encontrada, entonces se determina que hay consistencia y el ciclo puede continuar desde el paso número 2 hasta alcanzar convergencia o hasta generar una inconsistencia lógica. En caso contrario, si alguna de estas dos fórmulas no es satisfacible, se concluye que el modelo original $\mathcal{P}_{\mathcal{LIA}}$ es `unsat` y el ciclo termina.

La Figura 2.3 ejemplifica el funcionamiento del ciclo de resolución de fórmulas en el framework DPLL(T), el cual permite combinar lógica proposicional con teorías de primer orden, como la Aritmética Lineal de Enteros (*LIA*), para resolver problemas *SMT*.

El proceso comienza con una fórmula original $\mathcal{P}_{\mathcal{LIA}}$ modelada con una de las lógicas SMT (*LIA*), la cual es transformada a un modelo proposicional $\mathcal{P}_{\mathcal{SAT}}$ mediante una abstracción de predicados. A este modelo proposicional se le aplica un solver *SAT* para obtener una asignación candidata que satisfaga la fórmula.

Posteriormente, la asignación se valida en el contexto de la teoría (LIA). Si se detecta una contradicción entre los valores asignados y las restricciones teóricas, se genera un lemma

teórico que representa la incompatibilidad. Este lemma se incorpora al modelo proposicional y el proceso continúa iterativamente.

Cuando la conjunción del modelo proposicional y los lemas teóricos acumulados resulta insatisfacible o *unsat*, se concluye por prueba por contradicción que el modelo original también lo es. Esto permite garantizar la insatisfacibilidad teórica del problema modelado, cerrando el ciclo de búsqueda de forma eficiente.

En particular, el proceso de resolución llevado a cabo en la Figura 2.3 realiza los siguientes pasos:

1. **Modelo Original:**

   Se parte de una fórmula original en `LIA`

   $$\mathcal{P}_{\mathcal{LIA}} = \neg\,(a \geq 3) \wedge ((a \geq 3) \vee (a \geq 5))$$

2. **Traducción a Lógica Proposicional:**

   Se asignan predicados proposicionales a las siguientes sub-fórmulas:

   $$p_1 = (a \geq 3)\,, p_2 = (a \geq 5)$$

   Lo que genera la siguiente fórmula proposicional:

   $$\mathcal{P}_{\mathcal{SAT}} = \neg p_1 \wedge (p_1 \vee p_2)$$

   La cuál es satisfacible o *sat* cuando:

   $$p_1 = \text{false}, p_2 = \text{true}$$

3. **Verificación en *LIA*:**

   Con la asignación anterior, se valida la consistencia en *LIA*:

   $$\neg\,(a \geq 3) \wedge (a \geq 5) \rightarrow \text{false}$$

Lo que lleva a concluir que la interpretación no es válida en `LIA` dado que a no puede ser menor a 3 y al mismo tiempo mayor a 5.

4. **Generación de lemmas:**

   A partir del conflicto en `LIA` se genera el siguiente lemma teórico:

   $$p_1 \vee \neg p_2 \rightarrow \text{true}$$

   Lo cual implica que la combinación anterior no puede darse.

5. **Combinación de Lógica Proposicional y `LIA`:**

   Se plantea la hipótesis conjunta:

   $$\mathcal{P}_{\mathcal{SAT}} \wedge \mathcal{P}_{\mathcal{LIA}} \rightarrow \text{true}$$

   Pero al evaluarla, se encuentra que es insatisfacible o *unsat*.

6. **Conclusión:**

   Por lo tanto, se concluye que la fórmula original $\mathcal{P}_{\mathcal{LIA}}$ es *unsat*, cerrando el ciclo de razonamiento por prueba por contradicción.

SMT ha demostrado ser una herramienta altamente versátil para el modelamiento y la verificación de sistemas complejos, tanto en el ámbito del hardware como del software [13, 117]. Su principal fortaleza radica en la capacidad de integrar diversas teorías dentro de un mismo marco lógico, lo que permite representar de manera directa relaciones aritméticas, estructurales y simbólicas sin necesidad de recurrir a codificaciones exclusivamente booleanas. Esta flexibilidad se traduce en un equilibrio notable entre expresividad y eficiencia computacional, característica que ha impulsado su adopción en múltiples dominios de aplicación.

En los últimos años, el uso de *SMT* como alternativa a los enfoques basados en *SAT* para la resolución de problemas de optimización combinatoria ha experimentado un crecimiento sostenido [23]. Ello se debe a que las teorías disponibles en *SMT* permiten modelar problemas de acuerdo con su naturaleza semántica, por ejemplo, utilizando aritmética lineal,

bit-vectors o arreglos, en lugar de depender de representaciones puramente booleanas. Gracias a esta capacidad de abstracción, las aplicaciones de *SMT* y las mejoras en los solvers asociados han aumentado significativamente, dando lugar a investigaciones orientadas tanto a la optimización del rendimiento como a la ampliación de los dominios o teorías soportadas [118, 119, 120, 121].

Un aspecto especialmente relevante del enfoque de *SMT* es su utilidad para describir formalmente el comportamiento de sistemas dinámicos mediante fórmulas lógicas que representan estados y transiciones [112]. Este principio constituye la base de herramientas dedicadas al análisis, la verificación formal y las pruebas automatizadas de programas [117, 122, 123] y dispositivos físicos [124].

Por otra parte, los solvers *SMT* modernos, como Z3 [56], han ampliado su alcance más allá de la verificación de satisfacibilidad, incorporando soporte para múltiples teorías y lógicas dentro de un mismo framework. Estas herramientas permiten determinar la validez de expresiones lógicas complejas y realizar comprobaciones formales sobre modelos heterogéneos, consolidando a *SMT* como una de las tecnologías fundamentales en el desarrollo de métodos formales y en la investigación aplicada a la optimización simbólica [125, 126, 127, 128].

2.5. *Optimization Modulo Theories* (*OMT*)

Optimization Modulo Theories (OMT) [23] surge como una extensión natural del framework *SMT*, con el propósito de incorporar mecanismos de optimización sobre funciones objetivo, además de la mera verificación de satisfacibilidad. Mientras *SMT* responde a la pregunta de si existe una asignación que satisfaga un conjunto de restricciones, *OMT* busca, adicionalmente, encontrar aquella asignación que optimice una métrica cuantitativa, como el costo, el beneficio o la distancia, bajo las mismas restricciones lógicas y teóricas.

La primera implementación formal de *OMT* se enfocó en problemas de Aritmética Lineal de Enteros (`LIA`), abordando la optimización mediante dos estrategias complementarias:

1. Integrando un solver de Programación Lineal Entera externo en el ciclo *SMT*.

2. Modificando directamente el framework DPLL(T) para incorporar mecanismos de búsqueda del óptimo dentro del proceso principal, similar al proceso implementado en el *T-solver* [129]. Se considera *T-solver* dado que desde el 2006 hasta el 2011 (fecha de publicación de *OMT*) ha sido el *solver* utilizado por todos los ganadores en la categoría `LIA`/`LRA` de la *SMT-COMP* [130], evento en el que se compite por resolver problemas de satisfacción de restricciones usando *SMT*.

Desde entonces, el paradigma *OMT* ha evolucionado significativamente, dando origen a herramientas como *OptiMathSAT* [54], *Z3* [56, 131] y *CVC5* [115], las cuales permiten abordar problemas de optimización en dominios mixtos y combinados, abarcando teorías numéricas, discretas y simbólicas.

La principal ventaja de *OMT* reside en su capacidad para modelar problemas de satisfacción y optimización de restricciones (*CSP/CSOP*) utilizando estructuras matemáticas altamente expresivas. Esto permite representar de manera directa relaciones no lineales, condiciones lógicas dependientes y dominios mixtos. En la práctica, este enfoque se ha aplicado con éxito en áreas como verificación formal, planificación automática, análisis de seguridad y despliegue óptimo de recursos en sistemas distribuidos [132, 13].

No obstante, la flexibilidad que caracteriza al framework *OMT* también introduce desafíos relevantes en términos de desempeño. Distintos estudios han evidenciado que la elección de la lógica *SMT* utilizada para modelar un problema puede tener un impacto significativo en los tiempos de resolución y en la calidad de las soluciones [133, 24]. Por ejemplo, se ha observado que:

- Las lógicas basadas en Bit-Vectors (`QF_BV`) tienden a presentar un mejor rendimiento en problemas secuenciales o de planificación, como el *Travelling Salesman Problem* o el *Nurse Scheduling Problem*, debido a la eficiencia de sus representaciones binarias.
- Las lógicas `LRA` y `QF_ALIA` resultan más adecuadas para problemas con dominios numéricos amplios o estructuras de tipo arreglo, como el *Unbounded Knapsack Problem*.

- La lógica `LIA`, aunque semánticamente más natural para modelar problemas, no siempre conduce al mejor desempeño computacional.

Estos resultados ponen de manifiesto la relevancia de desarrollar mecanismos automáticos de selección y traducción de lógicas, capaces de aprovechar las diferencias de desempeño observadas entre ellas. La automatización de estos procesos constituye, precisamente, una de las principales motivaciones que orientan la presente investigación, en la cual se explora la posibilidad de combinar y traducir dinámicamente las restricciones de un problema hacia distintas lógicas *SMT* dentro de un mismo modelo de optimización.

2.6. Definición y formalización del problema

La resolución de Problemas de Satisfacción de Restricciones y/o Optimización *(CSP/CSOP)* mediante *Satisfiability Modulo Theories (SMT)* y su extensión *Optimization Modulo Theories (OMT)* se vuelve cada más frecuente en distintas áreas de la ingeniería, lo que ha impulsado múltiples estudios para mejorar el desempeño de los *solvers* y de las estrategias de búsqueda que hay por detrás de estos [134, 135, 136, 137, 138, 139, 140, 54, 141, 142, 143], logrando reducir en gran medida el tiempo requerido para encontrar soluciones al modelar y resolver estos problemas con *SMT/OMT*. Sin embargo, pese a los avances alcanzados en la resolución de problemas mediante estos frameworks, persisten limitaciones importantes relacionadas con la dependencia de la lógica empleada en el modelamiento. Estudios han evidenciado que el desempeño de un solver varía de forma significativa según la lógica o teoría utilizada para representar un mismo problema [133, 144, 24].

Esta dependencia implica que la elección de la lógica no es un aspecto meramente sintáctico, sino que constituye una decisión de modelamiento crítica con efectos directos en la eficiencia de la resolución y en la calidad de las soluciones obtenidas, cómo se discute al final de la Sección 2.5. Dentro de esta dependencia, se evidencian al menos tres limitaciones clave que dificultan el aprovechamiento de las ventajas asociadas al uso de múltiples lógicas en el modelamiento de problemas:

1. **Falta de un procedimiento de selección de lógica:** Actualmente no existe una metodología que permita determinar cuál o cuáles lógicas pueden ser más adecuada para modelar un problema específico, lo que obliga a realizar pruebas exhaustivas manualmente.

2. **Ausencia de traducción automática entre lógicas:** No se cuenta con mecanismos que permitan convertir automáticamente un modelo matemático desde una lógica a otra. Esta tarea requiere un alto nivel de conocimiento experto tanto en el dominio del problema como en las características de cada lógica.

3. **Bajo aprovechamiento del uso combinado de lógicas:** Aún no se ha explorado suficientemente el potencial de utilizar dos o más lógicas simultáneamente para modelar distintos componentes de un mismo problema, lo que podría conducir a mejoras significativas en tiempo de resolución y calidad de las soluciones.

En este contexto, surge la necesidad de desarrollar un marco metodológico y computacional que permita seleccionar, traducir y combinar lógicas *SMT* de forma automática, facilitando así la experimentación con diferentes configuraciones y la identificación de aquellas que conduzcan al mejor desempeño, sin alterar la semántica de las restricciones del problema original.

Capítulo 3

En camino hacia la codificación automática

Con el propósito de analizar el impacto que tiene la selección de la lógica *SMT* en el desempeño de los solvers al abordar problemas de satisfacción y optimización de restricciones, se plantea un conjunto de experimentos orientados a evaluar distintas estrategias de modelamiento sobre problemas representativos de la literatura. En particular, se busca determinar en qué medida la elección de la lógica influye en el tiempo de resolución, el consumo de memoria y la calidad de las soluciones obtenidas, considerando distintos dominios de aplicación, tales como planificación (*Planning*) y asignación de recursos (*Scheduling*).

Para ello, se aborda un conjunto de problemas clásicos de satisfacción y optimización de restricciones (*CSOP*), para los cuales se genera un número controlado de instancias con distintos niveles de complejidad. Cada problema se modela inicialmente utilizando la lógica *Linear Integer Arithmetic* (`LIA`), de modo que todas las variables y restricciones se encuentren definidas sobre el dominio de números enteros. Este modelamiento base sirve como punto de partida para la construcción de versiones equivalentes obtenidas a partir de la combinatoria entre múltiples lógicas para modelar las restricciones del problema.

En este trabajo nos centramos en las siguientes 4 lógicas de las discutidas en la Sección 2.4:

- **LIA** ***(Linear Integer Arithmetic)***: combina la teoría *Core* o *SAT* con *Enteros*, lo que permite representar combinaciones booleanas de inecuaciones entre polinomios lineales sobre variables con dominio en los números enteros.

- **LRA** ***(Linear Real Arithmetic)***: combina la teoría *Core* o *SAT* con *Reales*, lo que permite representar combinaciones booleanas de inecuaciones entre polinomios lineales sobre variables con dominio en los números reales.

- **QF_BV** ***(Quantifier-Free formulas over fixed-size BitVectors)***: combina la teoría *Core* o *SAT* con *Bit-Vectors de Tamaño Fijo*, se basa en operaciones aritméticas y lógicas sobre vectores de bits de tamaño fijo.

  Permite representar combinaciones booleanas de inecuaciones entre polinomios lineales sobre variables con dominio en los números binarios con una cantidad determinada de bits. Algunos *solvers* de *SMT*, como *Z3*, distinguen a los *unsigned bit-vectors* de los *signed bit-vectors* usando operadores exclusivos para cada tipo al momento de declarar una inecuación. En esta lógica el número de bits que posee una variable es fijo (*fixed-size*) y se define al momento de declararla, por lo que en el caso de querer realizar operaciones entre múltiples variables hay que tener en consideración la cantidad de bits de cada una y la cantidad de bits que se necesitarán para almacenar el resultado de la operación para así evitar perder información en caso de definir una cantidad de bits menor a la necesaria, o para evitar ocupar recursos innecesariamente en caso de definir una cantidad de bits mayor a la necesaria.

- **QF_ALIA** ***(Quantifier-Free formulas over Arrays with Extensionability over Linear Integer Arithmetic)***: combina la teoría *Core* o *SAT* con *Arreglos* extendiendo a la teoría de *Enteros*, incorpora el manejo de arreglos junto con la teoría de enteros y booleanos.

  QF_AX es, inicialmente, una extensión libre de cuantificadores de la teoría de arreglos (*ArraysEx*). Para poder representar índices, valores e inecuaciones es necesario extender al menos una teoría adicional que permita proporcionar un dominio a las variables del problema, siendo en este caso la teoría de números enteros. Así nace la lógica QF_ALIA, la cual permite representar inecuaciones entre polinomios lineales mediante uno o más arreglos cuyos índices y valores tienen dominio en los números enteros. Al

igual que las lógicas anteriores, inherentemente extiende a la teoría de booleanos, por lo que se puede representar combinaciones booleanas entre las inecuaciones usando los valores del arreglo.

Las transformaciones entre lógicas se realizan siguiendo un conjunto estructurado de reglas de traducción, las cuales preservan las restricciones del modelo original, pero modifican su representación interna para adecuarla al dominio de la lógica destino. Este proceso permite obtener una comparación directa y controlada del comportamiento del solver frente a modelos equivalentes bajo distintas lógicas *SMT*.

Finalmente, todas las instancias se resuelven empleando el solver *Z3* [56], utilizando el modo *Optimization Modulo Theories* (*OMT*) para abordar tanto problemas de satisfacción de restricciones como de optimización. Los resultados experimentales obtenidos se analizan posteriormente en el Capítulo 4, donde se presentan las métricas de desempeño y se discuten las observaciones derivadas de la comparación entre lógicas.

3.1. Problemas seleccionados

Con el fin de evaluar el desempeño de las distintas lógicas *SMT* al abordar problemas de satisfacción y optimización de restricciones, se seleccionó un conjunto representativo de problemas clásicos provenientes de la literatura y de la biblioteca de referencia CSPLib [145]. Estos problemas presentan diferentes estructuras combinatorias, tipos de restricciones y dominios de aplicación, lo que permite analizar el comportamiento del solver Z3 bajo condiciones variadas y comparables.

Las instancias utilizadas fueron seleccionadas de manera que cada problema cuente con configuraciones de distinta complejidad, permitiendo comparar el comportamiento de los *solvers* especializados en cada lógica a medida que aumenta la dificultad del modelo matemático.

Los problemas considerados en esta investigación se describen a continuación.

3.1.1. *N-Queens*

N-Queens o *N-Reinas* corresponde al problema 054 de la CSPLib [29] y posee un nivel de complejidad *NP-complete* [146]. Su formulación consiste en posicionar N reinas de un mismo color en un tablero de ajedrez de tamaño $N \times N$, de modo que ninguna de ellas pueda atacar o ser atacada por otra, como se observa en la Figura 3.1. Esto implica que no pueden coexistir dos reinas en la misma fila, columna o diagonal.

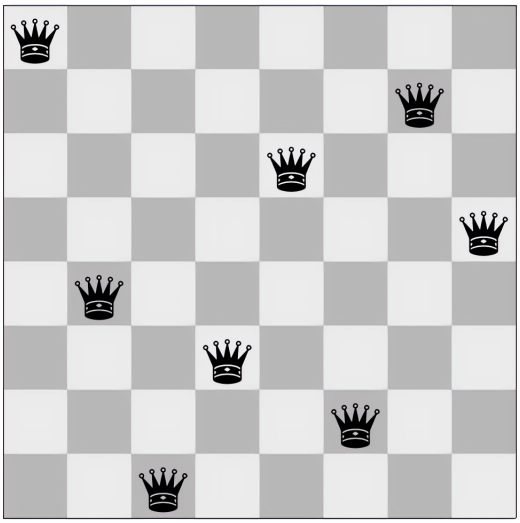


Figura 3.1: Representación visual de las *N-Reinas* [147], en donde se puede observar la distribución de 8 Reinas en un tablero de 8×8 de tal manera que ninguna reina sea atacada por otras.

El problema es ampliamente utilizado como caso de estudio en modelamiento declarativo y en la evaluación de solvers *CSP* y *SMT* [148, 149, 150], debido a su estructura combinatoria simple pero altamente sensible a la estrategia de representación utilizada.

Instancias

En este problema, cada instancia se define únicamente por el tamaño del tablero y la cantidad de reinas a ubicar, ambos determinados por el valor de N. Si $N = 4$, el tablero corresponde a un tamaño de 4×4 y se deben posicionar 4 reinas.

Con el objetivo de cubrir un rango amplio de complejidad, se consideraron los valores para

N descritos en la Tabla 3.1. Las columnas *Número de Variables* y *Número de Inecuaciones* anticipan los valores que se derivan del modelo presentado en la Sección 3.2.1: para cada instancia de tamaño N, el modelo define exactamente N variables (una por columna del tablero) y $N + \binom{N}{2} \cdot 2$ inecuaciones, correspondientes a las restricciones de dominio, unicidad de fila y unicidad de diagonal, respectivamente. Estos valores se incluyen aquí para ilustrar el crecimiento de la complejidad del modelo con el tamaño de la instancia.

Número de Reinas	Número de Variables	Número de Inecuaciones
4	4	22
8	8	92
12	12	210
16	16	376
20	20	590
50	50	3725
100	100	14950
120	120	21540
140	140	29330
160	160	38320
180	180	48510
200	200	59900

Tabla 3.1: Instancias base para las *N-Reinas*.

De esta manera, se generaron un total de 12 instancias únicas para este problema con complejidad ascendente según el número de reinas y el número de variables e inecuaciones que estas producen.

3.1.2. *Unbounded Knapsack Problem (UKP)*

UKP es una variación del problema 133 de la CSPLib [151], conocido como *Knapsack Problem*. En su formulación clásica, se dispone de una mochila con capacidad C y un conjunto de N objetos, cada uno con un peso W_n y una utilidad V_n. El objetivo es seleccionar una combinación de objetos que maximice la utilidad total sin superar la capacidad de la mochila, como se ilustra en la Figura 3.2.

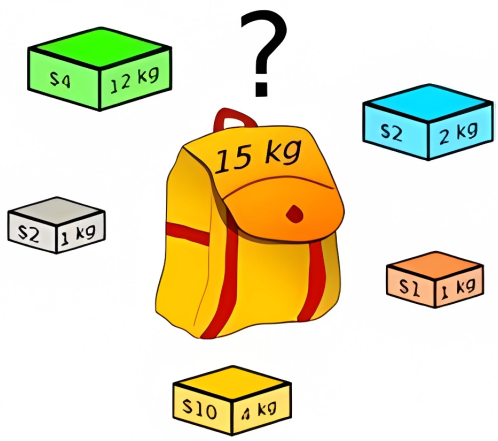


Figura 3.2: Representación visual del *Knapsack Problem* [152], en donde se puede observar la capacidad de la mochila, la cantidad de elementos, el peso de cada elemento y su utilidad.

A diferencia de la versión clásica, en *UKP* cada objeto puede seleccionarse un número ilimitado de veces. Este problema se considera uno de los *NP-complete* más elementales debido a su estructura unidimensional y al reducido número de restricciones que lo definen, lo cual lo convierte en un excelente punto de comparación para estudiar el impacto del modelamiento y la lógica utilizada [153, 154, 155, 156].

Instancias

Para este problema se generaron diez instancias artificiales utilizando un generador especializado [154], variando tanto la cantidad de objetos como la capacidad de la mochila:

Las semillas utilizadas en la generación de cada instancia se codifican en el nombre de archivo: "-0-" corresponde a la semilla 731232778, "-1-" a 594914841, "-2-" a 324673955, "-3-" a 300958364, y "-4-" a 182130978.

Cada instancia contiene los siguientes elementos:

- Número de elementos (N) y capacidad de la mochila (C), resumidos en la Tabla 3.2; los valores de variables e inecuaciones que allí se listan corresponden al conteo derivado del modelo de la Sección 3.2.2 para cada configuración (N, C).
- Lista con la utilidad v (entero) de cada objeto
- Lista con el peso w (entero) de cada objeto

Instancia	Número de elementos (N)	Capacidad de la mochila (C)	Número de variables	Número de inecuaciones
n10-0-c7370	10	7370	10	11
n20-1-c17050	20	17050	20	21
n30-0-c27795	30	27795	30	31
n40-3-c39588	40	39588	40	41
n50-4-c41934	50	41934	50	51
n60-2-c57601	60	57601	60	61
n70-3-c68135	70	68135	70	71
n80-0-c67603	80	67603	80	81
n90-2-c88213	90	88213	90	91
n100-2-c77552	100	77552	100	101

Tabla 3.2: Instancias generadas para el *UKP* [154].

El objetivo es seleccionar objetos, con posibilidad de repetición, de modo que la utilidad total sea máxima sin superar la capacidad de la mochila.

3.1.3. *Nurse Scheduling Problem (NSP)*

NSP es un problema clásico de asignación [157], que consiste en asignar turnos de trabajo diarios a un conjunto de enfermeras, cumpliendo tanto restricciones duras (por ejemplo, cobertura mínima de turnos o límites de horas de trabajo) como restricciones blandas (preferencias personales o continuidad de turnos), resultando en una planificación como la observada en la Tabla 3.3.

	Día 1			Día 2			...	Día 7			Número
Turnos	M	T	N	M	T	N	...	M	T	N	de turnos
Enfermera 1							...				6
...	...	...	...	...	...	...	...	...	...	...	...
Enfermera N							...				6
Número de enfermeras	15	18	13	13	18	13	...	13	14	13	314

Tabla 3.3:

Representación visual de la planificación empleada para el *Nurse Scheduling Problem* [158], en donde se puede observar la cantidad de *enfermeras* asignada para cada día y sus distribuciones según la oferta de turnos para cada día, separados en "Mañana (M)", "Tarde (T)" y "Noche (N)".

El objetivo es maximizar la calidad global de la asignación, equilibrando la satisfacción de las restricciones y la equidad entre las asignaciones. Dada su estructura combinatoria y su similitud con el problema de *Machine Scheduling*, se asume o espera que *NSP* tenga complejidad *NP-hard* [159].

Instancias

Para este problema se seleccionaron seis instancias de la *NSP-LIB* [160]. Estas instancias son generadas combinando restricciones duras y blandas relacionadas con cobertura y preferencias, lo que da origen a cuatro conjuntos de 7290 instancias de tipo ***diversas*** y dos conjuntos de 960 instancias de tipo ***realistas***.

Cada instancia incluye la siguiente información:

- **Tamaño del problema:** número de *enfermeras* (N), días (D) y turnos por día (S).
- **Matriz de cobertura:** matriz $D \times S$ cuyos valores indican la cantidad de *enfermeras* requeridos en cada turno.
- **Matriz de preferencias:** matriz $N \times (D \times S)$ cuyos valores expresan la preferencia de cada *enfermera* por trabajar en un turno específico.

Adicionalmente, la *NSP-LIB* define 16 configuraciones de restricciones adicionales, entre las que destacan:

- Límites mínimos y máximos de días asignados a cada *enfermera*.
- Límites mínimos y máximos de días consecutivos trabajados.
- Límites mínimos y máximos de días consecutivos en un mismo turno.
- Límites mínimos y máximos de asignaciones de cada turno a cada *enfermera*.

Además, el último turno de cada día se reserva como día de descanso, sin asignación de personal.

El objetivo en estas instancias es satisfacer todas las restricciones y maximizar la preferencia total, asignando a cada *enfermera* los turnos más valorados. Las columnas de variables e inecuaciones de la Tabla 3.4 se obtienen a partir del modelo descrito en la Sección 3.2.3, aplicado a cada combinación de enfermeras, días y turnos de la instancia correspondiente.

De las seis instancias seleccionadas, cuatro pertenecen al conjunto ***diversas*** (con 4 turnos por día y 7 días por jornada), diferenciándose por el número de *enfermeras* (25, 50, 75 y 100). Las dos restantes pertenecen al conjunto ***realistas*** (con 4 turnos por día y 28 días por jornada), con 30 y 60 *enfermeras*, respectivamente.

- Instancias ***diversas*** (*gen 8*): 25_1.nsp, 50_1.nsp, 75_1.nsp, 100_1.nsp
- Instancias ***realistas*** (*gen 16*): 30_1.nsp, 60_1.nsp

Instancia	Número de Enfermeras	Número de Días	Número de Turnos	Número de variables	Número de inecuaciones
25_1-8	25	7	4	700	1046
50_1-8	25	7	4	1400	2071
75_1-8	25	7	4	2100	3096
100_1-8	100	7	4	2800	4121
30_1-16	30	28	4	3360	4464
60_1-16	25	28	4	6720	8844

Tabla 3.4: Instancias para el *NSP*.

En la Tabla 3.4 se observan las instancias utilizadas junto a la cantidad de *enfermeras*, días y turnos de cada una, además de la cantidad de variables e inecuaciones para ilustrar la diferencia de complejidad entre cada instancia.

3.1.4. *Travelling Salesman Problem (TSP)*

TSP se define de la siguiente manera [161]: dada una lista de N ciudades y las distancias entre cada par de ellas, como se observa en la Figura 3.3, se busca determinar una ruta que visite cada ciudad exactamente una vez, regresando a la ciudad de origen, de manera que la distancia total recorrida sea mínima.

	1	2	3	4	5
1	∞	7	9	6	1
2	7	∞	3	4	9
3	9	3	∞	2	5
4	6	4	2	∞	3
5	1	9	5	3	∞

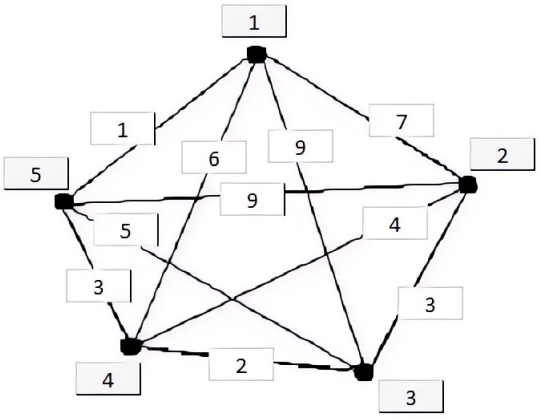


Figura 3.3: Representación visual del *Travelling Salesman Problem* [162], en donde se puede observar el grafo de la red de ciudades y su respectiva matriz de adyacencia, representando las ciudades a recorrer y sus distancias.

El *TSP* es *NP-complete* cuando se aborda como un problema de decisión, ya que puede obtenerse como una reducción del *Hamiltonian Cycle Problem*. Sin embargo, cuando se considera como problema de optimización, su complejidad se eleva a *NP-hard* [163], dado que verificar si una solución encontrada corresponde al ciclo más corto requiere una exploración exhaustiva del espacio de soluciones.

Debido a su relevancia histórica y su aplicabilidad en logística, planificación y análisis de rutas, el *TSP* constituye uno de los puntos de referencia más utilizados para evaluar algoritmos de optimización combinatoria [164, 165].

Instancias

Para este problema se seleccionaron siete instancias únicas, cinco de ellas provenientes de la *TSP-LIB* [166], y dos adicionales de fuentes complementarias [167]. La principal diferencia entre ellas radica en la cantidad de ciudades que conforman la red, como se observa en la Tabla 3.5. Las columnas *Número de variables* y *Número de inecuaciones* se derivan del modelo presentado en la Sección 3.2.4: para una instancia con N ciudades, el modelo define $N^2 + N$ variables (N^2 variables de decisión $x_{i,j}$ más N variables auxiliares u_i) y $N^2 + N + N^2 +$

$(N-1)^2+N$ inecuaciones, correspondientes a las restricciones de dominio, flujo, eliminación de subtours, unicidad de visita y prohibición de bucles, respectivamente.

Instancia	Número de ciudades	Longitud mínima del tour	Número de variables	Número de inecuaciones
five	5	19	25	100
burma14	14	30	196	721
p01	15	291	225	825
ulysses16	16	71	256	936
gr17	17	2085	289	1054
fri26	26	937	676	2431
att48	48	33523	2304	8184

Tabla 3.5: Instancias base para el *TSP*.

Cada instancia se representa mediante una matriz de adyacencia simétrica, donde cada celda indica la distancia entre dos ciudades. El objetivo es encontrar un *tour* que recorra todas las ciudades una única vez y minimice la distancia total recorrida.

Adicionalmente, con el propósito de disponer de instancias de distinta escala y mantener tiempos de resolución acotados, se generaron variantes adicionales a partir de las instancias *burma14* y *ulysses16*. Este proceso de generación consiste en construir sub-instancias progresivas tomando los primeros n nodos de cada conjunto de datos original, incrementando n de forma gradual hasta alcanzar la cantidad total de nodos de la instancia completa.

De esta manera, a partir de *burma14* se obtuvieron las instancias *burma5*, *burma6*, ..., *burma14*, y a partir de *ulysses16* se generaron las instancias *ulysses5*, *ulysses6*, ..., *ulysses16*. Este enfoque permite analizar de manera controlada el efecto del tamaño del problema en el desempeño de los distintos *solvers*.

3.1.5. *Balanced Academic Curriculum Problem (BACP)*

BACP corresponde al problema 030 de la CSPLib [168] y también es *NP-complete* [169]. La completitud se establece dado que la función objetivo es polinomialmente computable (*NP*), y porque es posible reducir una instancia del problema *3-Partition* al problema de decisión asociado a *BACP* [84].

El problema consiste en distribuir un conjunto de cursos a lo largo de varios periodos académicos, respetando los prerrequisitos entre ellos y procurando mantener un equilibrio en la carga académica de cada periodo. El objetivo es minimizar la variación en la cantidad de créditos por periodo, evitando así la existencia de semestres con cargas excesivas o insuficientes.

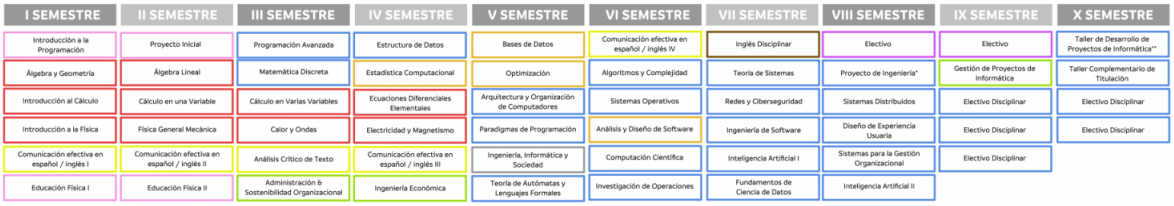


Figura 3.4: Representación visual del *Balanced Academic Curriculum Problem*. El problema busca generar una malla como la usada por la UTFSM [170] a partir de una lista de cursos, créditos o carga por curso, y sus pre-requisitos.

Instancias

Para este problema se utilizaron las tres instancias publicadas en la *CSPLib* [171]. Estas instancias fueron construidas a partir de las mallas curriculares de tres programas de Ingeniería Informática de la Universidad Técnica Federico Santa María. Cada instancia contiene la siguiente información:

- **Currículum académico:** conjunto de cursos pertenecientes al plan de estudios y las relaciones de precedencia entre ellos.
- **Número de periodos:** cantidad máxima de periodos académicos disponibles para distribuir todos los cursos.

- **Carga académica:** créditos asociados a cada curso, representando el esfuerzo requerido para completarlo.
- **Pre-requisitos:** pares ordenados (*curso*, *pre-requisito*) que definen el orden de precedencia.
- **Cargas mínima y máxima:** valores enteros que definen los límites inferior y superior de créditos por periodo.
- **Cantidad mínima y máxima de cursos:** límites en el número de asignaturas que un estudiante puede cursar por periodo.

Las columnas *Número de variables* y *Número de inecuaciones* de la Tabla 3.6 se obtienen a partir del modelo descrito en la Sección 3.2.5: el modelo define $M \times N + N + 1$ variables ($M \times N$ variables de asignación $x_{i,j}$, N variables de carga por periodo c_j y la variable global C) y un número de inecuaciones que depende del número de prerrequisitos y de los parámetros de carga y cantidad de cursos por periodo de cada instancia particular.

Instancia	Número de periodos	Número de cursos	Número de variables	Número de inecuaciones
bacp8	8	46	368	500
bacp10	10	42	420	556
bacp12	12	66	792	995

Tabla 3.6: Instancias para el *BACP*.

Las tres instancias descritas en la Tabla 3.6 se diferencian por la cantidad de periodos (8, 10 y 12, respectivamente) y por la estructura de cursos y pre-requisitos de cada currículum.

3.2. Modelamiento inicial de problemas usando `LIA`

Con el propósito de establecer una base común para la comparación entre distintas lógicas *SMT*, se desarrolló un modelamiento inicial de cada problema utilizando exclusivamente

álgebra lineal entera (LIA). Este enfoque permite garantizar la consistencia entre modelos, manteniendo la misma estructura de restricciones y variables en todos los casos, de modo que las diferencias observadas en los resultados puedan atribuirse directamente a las características de cada lógica y no a decisiones de diseño divergentes.

A partir de estos modelos lineales iniciales, se realizan posteriormente las transformaciones necesarias para expresarlos como combinaciones de las cuatro lógicas consideradas, preservando las formulaciones y restricciones originales. En las sub-secciones siguientes se describen los modelos construidos para cada problema, especificando los parámetros, variables, funciones objetivo y restricciones correspondientes.

3.2.1. *N-Queens*

El problema de las *N-reinas* consiste en ubicar N reinas idénticas en un tablero de ajedrez de tamaño $N \times N$ de manera que ninguna pueda atacar a otra. Este problema, descrito como *NP*-completo, se puede modelar en LIA utilizando una representación más compacta que la formulación estándar basada en variables binarias, lo que reduce significativamente el número de variables necesarias y la complejidad al tratarlo únicamente como un problema de satisfacción de restricciones:

Parámetros:

- N: Número de reinas que se deben posicionar en un tablero de tamaño $N \times N$, tal que $N \in \mathbb{Z}^+$.

Variables:

- x_c: Número de la fila en la columna c donde se posiciona una reina, $x_c \in \mathbb{Z}; \forall c \in \{1, ..., N\}$.

Restricciones del problema:

1. **Dominio de las variables:** Cada reina debe ubicarse dentro de los límites del tablero:

$$\bigvee_{k=1}^{N} x_c = k; \quad \forall c \in \{1, ..., N\}$$

2. **Una reina por fila:** No puede haber más de una reina en la misma fila:

$$x_{c_i} \neq x_{c_j}; \quad \forall c_i \in \{1, ..., N-1\}), \quad \forall c_j \in \{c_i + 1, ..., N\})$$

3. **Una reina por diagonal:** Dos reinas no pueden ubicarse en la misma diagonal:

$$x_{c_j} \neq x_{c_i} + (c_j - c_i)$$

$$x_{c_i} \neq x_{c_j} + (c_j - c_i)$$

$$\forall c_i \in \{1, ..., N-1\}, \quad \forall c_j \in \{c_i + 1, ..., N\}$$

3.2.2. *Unbounded Knapsack Problem (UKP)*

El modelo formulado para el *Unbounded Knapsack Problem (UKP)* representa una mochila unidimensional con capacidad finita, en la cual cada tipo de objeto puede incluirse un número ilimitado de veces, siempre que no se exceda la capacidad total. El objetivo es maximizar el valor total de los objetos seleccionados, respetando la restricción de peso. Este problema se modela como un caso clásico de optimización combinatoria con variables enteras no acotadas superiormente.

Parámetros:

- N: Número total de tipos de objetos, $N \in \mathbb{Z}^+$
- C: Capacidad máxima de la mochila, $C \in \mathbb{Z}^+$
- W_i: Peso asociado al objeto i, $W_i \in \mathbb{Z}; \quad \forall i \in \{1, \ldots, N\}$
- V_i: Valor asociado al objeto i, $V_i \in \mathbb{Z}; \quad \forall i \in \{1, \ldots, N\}$

Variables:

- x_i: Cantidad de veces que el objeto i es incluido en la mochila, $x_i \in \mathbb{Z}; \quad \forall i \in \{1, \ldots, N\}$

Función objetivo:

- **Maximizar el valor total en la mochila, v:**

$$max \quad v = \sum_{i=1}^{N} (V_i \times x_i)$$

Restricciones del problema:

1. **Dominio de las variables:** Cada objeto puede ser agregado cero o más veces. El límite superior se determina teóricamente a partir del peso del objeto y de la capacidad de la mochila:

$$\bigvee_{k=0}^{\lceil \frac{C}{W_i} \rceil} x_i = k; \quad \forall i \in \{1, \ldots, N\}$$

2. **Restricción de capacidad:** La suma total de los pesos de los objetos seleccionados no puede superar la capacidad máxima disponible de la mochila.

$$\sum_{i=1}^{N} (W_i \times x_i) \leq C$$

3.2.3. *Nurse Scheduling Problem (NSP)*

El *Nurse Scheduling Problem* (NSP) consiste en asignar turnos laborales a un conjunto de enfermeras durante un horizonte temporal determinado, cumpliendo con requisitos de cobertura, descansos mínimos y preferencias individuales.

El modelo desarrollado para este trabajo se basa en las restricciones y lineamientos propuestos en la *NSP-LIB* [160], adaptadas para representar de forma unificada las distintas variantes de instancias descritas en la Sección 3.1.3.

Parámetros:

- N : Número de *enfermeras*, $N \in \mathbb{Z}^+$
- D : Número de días del horizonte temporal de planificación, $D \in \mathbb{Z}^+$
- S : Número total de turnos posibles por día (el último representa el turno libre), $S \in \mathbb{Z}^+$
- $C_{d,s}$: Matriz de cobertura que indica la cantidad mínima de *enfermeras* requeridas para el día d y turno s, $C_{d,s} \in \mathbb{Z}_0^+; \quad \forall d \in \{1, ..., D\}, \quad \forall s \in \{1, ..., S\}$
- $P_{n,d,s}$: Matriz de preferencias, donde cada valor indica el grado de preferencia de la *enfermera* n para trabajar en el turno s del día d, $P_{n,d,s} \in \mathbb{Z}_0^+; \quad \forall n \in \{1, ..., N\}, \quad \forall d \in \{1, ..., D\}, \quad \forall s \in \{1, ..., S\}$
- min_wd: Cantidad mínima de días de trabajo por *enfermera*, $min\_wd \in \mathbb{Z}_0^+$
- max_wd: Cantidad máxima de días de trabajo por *enfermera*, $max\_wd \in \mathbb{Z}_0^+$
- min_cd : Cantidad mínima de días consecutivos de trabajo, $min\_cd \in \mathbb{Z}_0^+$
- max_cd : Cantidad máxima de días consecutivos de trabajo, $max\_cd \in \mathbb{Z}_0^+$
- min_cs_s : Cantidad mínima de días consecutivos que una *enfermera* puede trabajar en el turno s, $min\_cs_s \in \mathbb{Z}_0^+, s \in \{1, ..., S\}$
- max_cs_s : Cantidad máxima de días consecutivos que una *enfermera* puede trabajar en el turno s, $max\_cs_s \in \mathbb{Z}_0^+, s \in \{1, ..., S\}$
- min_s_s : Cantidad mínima de veces que se debe asignar el turno s a una misma *enfermera* durante el horizonte temporal, $min\_s_s \in \mathbb{Z}_0^+, s \in \{1, ..., S\}$
- max_s_s : Cantidad máxima de veces que se debe asignar el turno s a una misma *enfermera* durante el horizonte temporal, $max\_s_s \in \mathbb{Z}_0^+, s \in \{1, ..., S\}$

Variables:

- $x_{n,d,s} = \begin{cases} 1 & \text{si la } \textit{enfermera } n \text{ es asignado al día } d \text{ en el turno } s, \\ 0 & \text{en caso contrario} \end{cases}$

$$x_{n,d,s} \in \mathbb{Z}; \quad \forall n \in \{1, ..., N\}, \quad \forall d \in \{1, ..., D\}, \quad \forall s \in \{1, ..., S\}$$

Función objetivo:

- **Maximizar la satisfacción general de la planificación, H:**

$$max \quad H = \sum_{n=1}^{N} \sum_{d=1}^{D} \sum_{s=1}^{S} (P_{n,d,s} \times x_{n,d,s})$$

Restricciones del problema:

1. **Dominio de las variables:** Cada asignación es binaria, indicando si el turno es asignado o no.

$$\bigvee_{k=0}^{1} x_{n,d,s} = k; \quad \forall n \in \{1, \ldots, N\}, \quad \forall d \in \{1, \ldots, D\}, \quad \forall s \in \{1, \ldots, S\}$$

2. **Asignación única por día:** Cada *enfermera* debe estar asignada exactamente a un turno (incluido el libre) por día.

$$\sum_{s=1}^{S} x_{n,d,s} = 1; \quad \forall n \in \{1, ..., N\}, \quad \forall d \in \{1, ..., D\}$$

3. **Cobertura mínima de turnos:** Cada día y turno laboral debe tener al menos la cantidad requerida de *enfermeras*.

$$\sum_{n=1}^{N} x_{n,d,s} \geq C_{d,s}; \quad \forall d \in \{1, ..., D\}, \quad \forall s \in \{1, ..., S-1\}$$

4. **Límites de días totales de trabajo:** Cada *enfermera* debe cumplir con los límites de días trabajados en el horizonte temporal.

$$\sum_{d=1}^{D} \sum_{s=1}^{S-1} x_{n,d,s} \geq min\_wd; \quad \forall n \in \{1, ..., N\}$$

$$\sum_{d=1}^{D}\sum_{s=1}^{S-1} x_{n,d,s} \leq max\_wd; \quad \forall n \in \{1, ..., N\}$$

5. **Límites de asignación por turno:** Cada *enfermera* debe trabajar un turno s dentro de un rango específico de frecuencia.

$$\sum_{d=1}^{D} x_{n,d,s} \geq min\_s_s; \quad \forall n \in \{1, ..., N\}, \quad \forall s \in \{1, ..., S\}$$

$$\sum_{d=1}^{D} x_{n,d,s} \leq max\_s_s; \quad \forall n \in \{1, ..., N\}, \quad \forall s \in \{1, ..., S\}$$

6. **Restricción de días consecutivos de trabajo:** Cada *enfermera* debe trabajar al menos min_cd y a lo más max_cd días consecutivos.

$$\sum_{j=max\{i+1,2\}}^{min\{i+k+1,D\}}\sum_{s=1}^{S-1} x_{n,j-1,s} - 2 \times \sum_{s=1}^{S-1} x_{n,i-1,s} - 2 \times \sum_{s=1}^{S-1} x_{n,i+k-1,s} \leq k - 2;$$

$$\forall n \in \{1, ..., N\}, \quad \forall k \in \{min\{2, min\_cd\}, ..., min\_cd + 1\}, \quad \forall i \in \{1, ..., D + 2 - k\}$$

$$\sum_{d=j}^{min\{D,j+max\_cd+1\}}\sum_{s=1}^{S-1} x_{n,d,s} \leq max\_cd; \quad \forall n \in \{1, ..., N\}, \quad \forall j \in \{1, ..., D - max\_cd\}$$

7. **Restricción de días consecutivos por turno:** Si una *enfermera* trabaja en un turno s, debe repetirlo entre min_cs_s y max_cs_s veces consecutivas.

$$\sum_{j=max\{i+1,2\}}^{min\{i+k+1,D\}} x_{n,j-1,s} - 2 \times x_{n,i-1,s} - 2 \times x_{n,i+k-1,s} \leq k - 2;$$

$$\forall n \in \{1, ..., N\}, \quad \forall s \in \{1, ..., S - 1\},$$

$$\forall k \in \{min\{2, min\_cs_s\}, ..., min\_cs_s + 1\}, \quad \forall i \in \{1, ..., D + 2 - k\}$$

$$\sum_{d=j}^{min\{D,j+max\_cs_s+1\}} x_{n,d,s} \leq max\_cs_s;$$

$$\forall n \in \{1, ..., N\}, \quad \forall s \in \{1, ..., S - 1\}, \quad \forall j \in \{1, ..., D - max\_cd\}$$

3.2.4. *Travelling Salesman Problem (TSP)*

El modelo formulado para este problema se basa en el enfoque de programación lineal entera propuesto por Miller, Tucker y Zemlin [172], ampliamente utilizado en la literatura por su simplicidad y efectividad para evitar subtours mediante variables auxiliares. El objetivo consiste en determinar el recorrido de costo mínimo que visite cada nodo exactamente una vez y retorne al punto de partida:

Parámetros:

- N : Número de nodos del grafo, $N \in \mathbb{Z}^+$
- $D_{i,j}$: Distancia entre el nodo i y el nodo j, $D_{i,j} \in \mathbb{Z}_0^+, \quad \forall i, j \in \{1, ..., N\}$

Variables:

- $x_{i,j} = \begin{cases} 1 & \text{si el nodo } j \text{ es visitado inmediatamente después del nodo } i, \\ 0 & \text{en caso contrario} \end{cases}$

$$x_{i,j} \in \mathbb{Z}; \quad \forall i, j \in \{1, ..., N\}$$

- u_i : Variable auxiliar que representa el orden de visita del nodo i dentro del recorrido, utilizada para eliminar *sub-tours*, $u_i \in \mathbb{Z}, \forall i \in \{1, ..., N\}$

Función objetivo:

- **Minimizar la distancia total recorrida d:**

$$min \quad d = \sum_{i=1}^{N} \sum_{j=1}^{N} \left(D_{i,j} \times x_{i,j} \right)$$

Restricciones del problema:

1. **Dominio de las variables:** Cada variable $x_{i,j}$ solo puede tomar los valores 0 o 1, indicando la existencia o no de un trayecto directo entre los nodos i y j.:

$$\bigvee_{k=0}^{1} x_{i,j} = k; \quad \forall i, j \in \{1, ..., N\}$$

2. **Dominio de las variables auxiliares:** Las variables u_i se restringen a valores enteros entre 1 y N, representando la posición del nodo i en el recorrido.

   El dominio de u_i está restringido entre 1 y N:

$$\bigvee_{k=1}^{N} u_i = k; \quad \forall i \in \{1, ..., N\}$$

3. **Cada ciudad debe ser punto de partida una sola vez:** Desde cada nodo i debe existir exactamente una arista de salida hacia otro nodo j:

$$\sum_{j=1}^{N} x_{i,j} = 1; \quad \forall i \in \{1, ..., N\}, \quad i \neq j$$

4. **Cada ciudad debe ser punto de llegada una sola vez:** Hacia cada nodo j debe llegar exactamente una arista proveniente de otro nodo i:

$$\sum_{i=1}^{N} x_{i,j} = 1; \quad \forall j \in \{1, ..., N\}, \quad i \neq j$$

5. **Eliminación de subtours (restricciones de Miller–Tucker–Zemlin):** Estas restricciones impiden la formación de ciclos parciales que no incluyan a todos los nodos del grafo:

$$u_i - u_j + N \times x_{i,j} \leq N - 1;$$

$$\forall i, j \in \mathbb{Z}^+ : \quad 2 \leq i \leq N, \quad 1 \leq j \leq N$$

6. **Evitar visitas repetidas:** Dos nodos no pueden ocupar la misma posición dentro del recorrido:

$$u_i \neq u_j; \quad \forall i, j \in \{1, ..., N\}, \quad i \neq j$$

7. **Prohibición de bucles (self-loops):** No se permite que un nodo se visite a sí mismo:

$$x_{i,i} = 0; \quad \forall i \in \{1, ..., N\}$$

3.2.5. *Balanced Academic Curriculum Problem (BACP)*

El *Balanced Academic Curriculum Problem* (BACP) busca distribuir un conjunto de cursos dentro de un número fijo de periodos académicos, de manera que se respeten las relaciones de precedencia entre cursos y que la carga académica de cada periodo sea lo más equilibrada posible. El modelo utilizado en este trabajo se basa en la formulación de programación lineal entera propuesta por *Castro et al.* [171], la cual define un marco general para balancear tanto la cantidad de créditos como el número de cursos en cada periodo:

Parámetros:

- M : Número de cursos, $M \in \mathbb{Z}^+$
- N : Número de periodos académicos, $N \in \mathbb{Z}^+$
- α_i : Número de créditos del curso i, $\alpha_i \in \mathbb{Z}^+$, $\forall i \in \mathbb{Z}^+ : i \in \{1, ..., M\}$
- β : Carga académica **mínima** permitida por periodo, $\beta \in \mathbb{Z}^+$
- γ : Carga académica **máxima** permitida por periodo, $\gamma \in \mathbb{Z}^+$
- δ : Cantidad de cursos **mínima** por periodo, $\delta \in \mathbb{Z}^+$
- ϵ : Cantidad de cursos **máxima** por periodo, $\epsilon \in \mathbb{Z}^+$

Variables:

- $x_{i,j} = \begin{cases} 1 & \text{si el curso } i \text{ es asignado al periodo } j, \\ 0 & \text{en caso contrario} \end{cases}$

$$x_{i,j} \in \mathbb{Z}; \quad \forall i \in \{1, ..., M\}, \quad \forall j \in \{1, ..., N\}$$

- c_j : carga académica del periodo $j, c_j \in \mathbb{Z}; \quad \forall j \in \{1, ..., N\}$
- C : Mayor carga académica registrada a lo largo del currículum, $C \in \mathbb{Z}$

Función objetivo:

- Minimizar la mayor carga académica C entre todos los periodos, buscando un currículum equilibrado:

$$min \quad C = max\{c_1, ..., c_N\}$$

Restricciones del problema:

1. **Dominio de las variables:** Cada variable $x_{i,j}$ solo puede tomar los valores 0 o 1, indicando si el curso i es asignado o no al periodo j:

$$\bigvee_{k=0}^{1} x_{i,j} = k; \quad \forall i \in \{1, ..., M\}, \quad \forall j \in \{1, ..., N\}$$

2. **Definición de la carga máxima:** C se utiliza en la función objetivo como $C = max\{c_1, ..., c_N\}$, lo cual se logra representar mezclando la minimización de C junto a la siguiente inecuación:

$$C \geq c_j; \quad \forall j \in \{1, ..., N\}$$

3. **Cálculo de la carga académica por periodo:** La carga académica para el periodo j se define como:

$$c_j = \sum_{i=1}^{M} \left(\alpha_i \times x_{i,j}\right); \quad \forall j \in \{1, ..., N\}$$

4. **Asignación de curso por periodo:** Cada curso i debe ser asignados a un periodo j:

$$\sum_{j=1}^{N} x_{i,j} = 1; \quad \forall i \in \{1, ..., M\}$$

5. **Prerrequisitos de cursos:** si el curso b requiere como prerrequisito al curso a, entonces b no puede ser asignado a un periodo anterior al de a:

$$x_{b,j} \leq \sum_{r=1}^{j-1} x_{a,r} = 1; \quad \forall j \in \{2, ..., N\}, \quad \forall a \in \{1, ..., M-1\}, \quad \forall b \in \{a, ..., M\}$$

6. **Carga mínima por periodo:** La carga académica de cada periodo j debe ser mayor o igual a la carga mínima permitida:

$$c_j \geq \beta; \quad \forall j \in \{1, ..., N\}$$

7. **Carga máxima por periodo:** La carga académica de cada periodo j debe ser menor o igual a la carga máxima permitida:

$$c_j \leq \gamma; \quad \forall j \in \{1, ..., N\}$$

8. **Mínima cantidad de cursos por periodo:** El número de cursos del periodo j debe ser mayor o igual a la cantidad mínima permitida:

$$\sum_{i=1}^{M} x_{i,j} \geq \delta; \quad \forall j \in \{1, ..., N\}$$

9. **Máxima cantidad de cursos por periodo:** El número de cursos del periodo j debe ser menor o igual que la cantidad máxima permitida:

$$\sum_{i=1}^{M} x_{i,j} \leq \epsilon; \quad \forall j \in \{1, ..., N\}$$

En esta formulación, el equilibrio entre carga y precedencia se logra mediante la minimización del valor máximo de créditos C. Este enfoque asegura una distribución balanceada de los cursos, evitando periodos con sobrecarga o subutilización.

3.3. Traduciendo modelos hacia otras lógicas *SMT*

Una vez construidos los modelos a partir de *Linear Integer Arithmetic* (`LIA`), el siguiente paso consiste en extender su representación hacia otras lógicas *SMT*, con el objetivo de evaluar el comportamiento de los solvers frente a estos nuevos modelamientos que podrían ser puros (una sola lógica) o hibridos (múltiples lógicas). A diferencia de lo analizado en [24], donde cada modelo era completamente traducido desde `LIA` hacia una única lógica alternativa (`LRA`, `BV` o `ALIA`), en este trabajo se propone una estrategia combinada, en la que distintas partes de un mismo modelo pueden expresarse en diferentes lógicas.

El mecanismo de transformación propuesto se fundamenta en la modificación del dominio de las variables para inducir un cambio en la lógica subyacente de la función objetivo y de

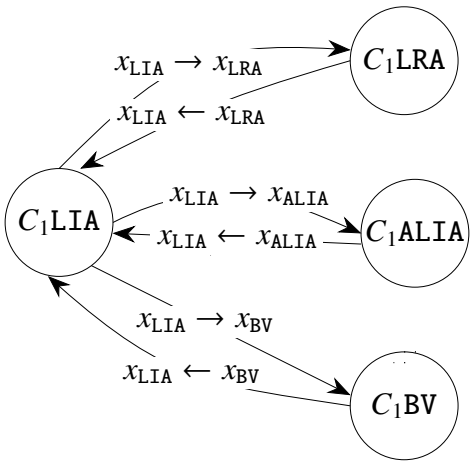


Figura 3.5: Esquema de transformación de restricciones utilizando `LIA` como núcleo central de interconexión con las lógicas `LRA`, `BV` y `ALIA`.

las restricciones del problema. El sistema adopta una topología de estrella (ver Figura 3.5), estableciendo a la lógica `LIA` como el núcleo semántico o *hub* de interconexión.

Sea C una restricción formulada originalmente en `LIA` sobre un conjunto de variables enteras $x_{\texttt{LIA}}$. El proceso de transformación hacia una lógica destino $\mathcal{L} \in \{\texttt{LRA}, \texttt{BV}, \texttt{ALIA}\}$ se define mediante una función de mapeo de tipos $\phi : \mathcal{D}(x_{\texttt{LIA}}) \to \mathcal{D}(x_{\mathcal{L}})$, donde $\mathcal{D}$ representa el dominio de las variables.

El diagrama de estados de la Figura 3.5 ilustra las transiciones permitidas para una restricción arbitraria C_1:

1. **Transformación Directa (`LIA` como Pivote):** Desde el estado central $C_1\texttt{LIA}$, es posible transformar la restricción directamente hacia Aritmética Real ($C_1\texttt{LRA}$), Vectores de Bits ($C_1\texttt{BV}$) o Aritmética de Arreglos de Enteros ($C_1\texttt{ALIA}$). Esta transición se ejecuta aplicando la proyección de variables correspondiente:

   $$x_{\texttt{LIA}} \to x_{\mathcal{L}} \implies C_1\texttt{LIA} \to C_1\mathcal{L}$$

   De manera análoga, el proceso inverso (retorno a `LIA`) se realiza mediante la retracción del dominio de las variables ($x_{\mathcal{L}} \leftarrow x_{\texttt{LIA}}$).

2. **Transformación Transitiva (Entre Lógicas Externas):** Debido a la arquitectura centralizada, no existen transiciones directas entre las lógicas periféricas (por ejemplo, de

LRA a ALIA). Para realizar dicha conversión, el sistema impone un paso intermedio obligatorio por el núcleo LIA. Si se desea transformar una restricción de LRA a ALIA, el flujo de transformación T se compone de dos etapas secuenciales:

$$T_{\texttt{LRA}\rightarrow\texttt{ALIA}} = (x_{\texttt{LRA}} \rightarrow x_{\texttt{LIA}}) \circ (x_{\texttt{LIA}} \rightarrow x_{\texttt{ALIA}}) \tag{3.1}$$

Este diseño garantiza que LIA actúe como el lenguaje común de intercambio, simplificando la implementación de los conversores al reducir la complejidad combinatoria de $N(N-1)$ traductores a solo $2N$ (donde N es el número de lógicas soportadas).

El proceso de transformación se basa en un conjunto de reglas generales que permiten mantener la semántica de las expresiones originales, garantizando equivalencia entre las restricciones del problema a lo largo de las diferentes lógicas. En particular, cada traducción se diseña de manera que:

- Las expresiones aritméticas, comparaciones y operaciones booleanas mantengan su validez formal.
- Se preserve el tipo de dato y el dominio lógico de las variables involucradas en el resultado final.
- Las restricciones mantengan la relación de factibilidad y correspondan a la estructura del modelo base.

Para explicar el proceso, se incluye un ejemplo de conversión parcial utilizando el modelo del *Travelling Salesman Problem (TSP)* (Sección 3.2.4). Este modelo se seleccionó dado que combina variables binarias y numéricas, lo que permite mostrar de forma clara la transición entre las distintas lógicas LIA, LRA y BV.

Las reglas de traducción aplicadas a cada tipo de componente (parámetros, función objetivo y restricciones) se presentan en las sub-secciones siguientes. Las traducciones específicas para los modelos de las *N-Reinas*, *UKP*, *NSP* y *BACP* se detallan en el Anexo A.1.

3.3.1. Transformación de LIA a LRA

La transformación de modelos formulados en LIA (*Linear Integer Arithmetic*) hacia LRA (*Linear Real Arithmetic*) se basa en un proceso de **relajación de dominio** de las variables, asegurando al mismo tiempo que las soluciones obtenidas en el modelo transformado continúen siendo factibles cuando se evalúan en el modelo original. En términos generales, esta conversión permite explorar el mismo espacio de soluciones discretas dentro de un dominio continuo, con el fin de aprovechar las capacidades de cálculo de los solvers optimizados para el álgebra lineal real.

La relajación de dominio consiste en reemplazar el dominio discreto de las variables enteras por un dominio continuo sobre los números reales. De este modo, si una variable en el modelo original se definía como $X \in \mathbb{Z}$, su versión transformada para LRA será $X \in \mathbb{R}$. Esta modificación amplía el espacio de búsqueda, lo que puede facilitar la convergencia del solver, pero también introduce el riesgo de obtener soluciones no enteras. Para contrarrestar este efecto, se mantienen las restricciones de factibilidad del modelo original mediante el uso de expresiones booleanas que acotan las variables a rangos discretos dentro del dominio real.

Una característica distintiva de las lógicas *SMT* es la integración nativa de álgebra booleana con álgebra numérica. Gracias a ello, es posible definir restricciones del tipo:

$$\bigvee_{k=0}^{1} x_{ij} = k; \quad \forall i, j \in \mathbb{Z}$$

las cuales obligan a las variables a adoptar únicamente valores en un conjunto finito previamente especificado. Esto permite realizar la transición desde LIA a LRA sin alterar la factibilidad del modelo ni requerir mecanismos adicionales de redondeo o penalización. En otras palabras, la lógica booleana actúa como un puente entre los dominios discretos y continuos, garantizando la equivalencia en las restricciones de ambas lógicas.

Para ilustrar el procedimiento, se considera nuevamente el modelo del *Travelling Salesman Problem* (TSP) descrito en la Sección 3.2.4. En su versión original basada en LIA, las variables se definen como:

- N: Cantidad de nodos o ciudades por visitar en la ruta.
- $x_{i,j}$: variable binaria que indica si el nodo j es visitado inmediatamente después del nodo i, con $x_{i,j} \in \mathbb{Z}; \quad \forall i, j \in \{1, ..., N\}$

  u_i : variable auxiliar que representa el orden de visita del nodo i dentro del recorrido, con $u_i \in \mathbb{Z}$.

Al transformar este modelo hacia `LRA`, ambas variables mantienen su interpretación, pero amplían su dominio hacia los números reales:

- $x_{i,j} \in \mathbb{R}$, donde $x_{i,j}$ puede adoptar valores continuos entre 0.0 y 1.0.

  $u_i \in \mathbb{R}$, permitiendo valores fraccionarios entre 1.0 y N.

No obstante, para preservar la factibilidad del modelo y evitar que las soluciones relajadas se desvíen del espacio discreto original, se mantienen las restricciones de dominio definidas en el modelo `LIA`. Estas restricciones, expresadas mediante álgebra booleana, acotan los posibles valores de las variables reales a un conjunto discreto:

$$\bigvee_{k=0}^{1} x_{i,j} = k; \quad \forall i, j \in \{1, ..., N\}$$

$$\bigvee_{k=1}^{N} u_i = k; \quad \forall i \in \{1, ..., N\}$$

De este modo, la estructura combinatoria del problema se conserva, aunque las operaciones aritméticas internas se ejecuten en un dominio continuo.

Esta estrategia de relajación y acotamiento booleano resulta particularmente útil para la integración entre modelos, ya que permite comparar directamente el desempeño de solvers especializados en dominios enteros y reales sobre el mismo conjunto de restricciones. Además, posibilita la construcción de modelos híbridos en los que distintas restricciones del problema se formulen en diferentes lógicas, sin pérdida de consistencia entre ellas.

El modelo resultante para el *TSP* completamente transformado a `LRA` mantiene la misma estructura matemática del modelo original en `LIA`, diferenciándose únicamente en el dominio

de sus variables y en el tipo de operaciones aritméticas evaluadas por el solver. Así, el proceso de transformación de `LIA` a `LRA` no modifica las restricciones del problema, sino únicamente la naturaleza del espacio de búsqueda, abriendo la posibilidad de una resolución más eficiente en problemas donde las restricciones lineales continuas sean más favorables para la heurística del solver.

A continuación se presenta el modelo completo usando `LRA` para todas las variables y todas las restricciones del *TSP*:

Parámetros:

- N : Número de nodos del grafo, $N \in \mathbb{Z}^+$
- $D_{i,j}$: Distancia entre el nodo i y el nodo j, $D_{i,j} \in \mathbb{Z}_0^+$; $\quad \forall i, j \in \{1, ..., N\}$

Variables:

- $x_{i,j} = \begin{cases} 1.0 & \text{si el nodo } j \text{ es visitado inmediatamente después del nodo } i, \\ 0.0 & \text{en caso contrario} \end{cases}$

$$x_{i,j} \in \mathbb{R}; \quad \forall i, j \in \{1, ..., N\}$$

- u_i : Variable auxiliar que representa el orden de visita del nodo i dentro del recorrido, utilizada para eliminar *sub-tours*, $u_i \in \mathbb{R}, \forall i \in \{1, ..., N\}$

Función objetivo:

- **Minimizar la distancia total recorrida d:**

$$min \quad d = \sum_{i=1}^{N} \sum_{j=1}^{N} \left(D_{i,j} \times x_{i,j} \right)$$

Restricciones del problema:

1. **Dominio de las variables:** Cada variable $x_{i,j}$ solo puede tomar los valores 0 o 1, indicando la existencia o no de un trayecto directo entre los nodos i y j.:

$$\bigvee_{k=0}^{1} x_{i,j} = k; \quad \forall i, j \in \{1, ..., N\}$$

2. **Dominio de las variables auxiliares:** Las variables u_i se restringen a valores enteros entre 1 y N, representando la posición del nodo i en el recorrido.

   El dominio de u_i está restringido entre 1 y N:

$$\bigvee_{k=1}^{N} u_i = k; \quad \forall i \in \{1, ..., N\}$$

3. **Cada ciudad debe ser punto de partida una sola vez:** Desde cada nodo i debe existir exactamente una arista de salida hacia otro nodo j:

$$\sum_{j=1}^{N} x_{i,j} = 1; \quad \forall i \in \{1, ..., N\} : \quad i \neq j$$

4. **Cada ciudad debe ser punto de llegada una sola vez:** Hacia cada nodo j debe llegar exactamente una arista proveniente de otro nodo i:

$$\sum_{i=1}^{N} x_{i,j} = 1; \quad \forall j \in \{1, ..., N\} : \quad i \neq j$$

5. **Eliminación de subtours (restricciones de Miller–Tucker–Zemlin):** Estas restricciones impiden la formación de ciclos parciales que no incluyan a todos los nodos del grafo:

$$u_i - u_j + N \times x_{i,j} \leq N - 1; \quad \forall i, j \in \mathbb{Z}^+ : \quad 2 \leq i \leq N, \quad 1 \leq j \leq N$$

6. **Evitar visitas repetidas:** Dos nodos no pueden ocupar la misma posición dentro del recorrido:

$$u_i \neq u_j; \quad \forall i, j \in \{1, ..., N\} : \quad i \neq j$$

7. **Prohibición de bucles (self-loops):** No se permite que un nodo se visite a sí mismo:

$$x_{i,i} = 0; \quad \forall i \in \{1, ..., N\}$$

3.3.2. Transformación de `LIA` a `QF_BV`

La transformación de modelos definidos en `LIA` (*Linear Integer Arithmetic*) hacia `QF_BV` (*Quantifier-Free Bit-Vector*) implica traducir las variables y expresiones aritméticas del dominio de los números enteros a un dominio binario representado mediante vectores de bits de tamaño fijo. Este proceso requiere una cuidadosa definición del rango de valores posibles para cada variable, con el fin de evitar pérdidas de información y asegurar que las operaciones aritméticas se mantengan consistentes con las del modelo original.

En términos generales, esta transformación depende directamente de la amplitud del dominio numérico de cada variable. Si las variables pueden asumir valores enteros positivos, la codificación binaria simple resulta suficiente; si en cambio el dominio incluye números negativos, se debe recurrir al uso de complemento de dos para preservar la representación de los signos. De esta manera, cada número entero de $X \in \mathbb{Z}$ se transforma en un vector de bits $\mathbf{X} \in \mathbb{Z}_2^N$, donde N corresponde al número mínimo de bits necesarios para representar el rango completo de valores posibles de X.

La transformación de variables pseudo-booleanas (binarias) es directa, ya que su dominio original $\{0, 1\}$ coincide con el dominio de los bit-vectors de tamaño 1. Por ejemplo, una variable $x_{i,j} \in \{0, 1\}$ puede representarse como $x_{i,j} \in \mathbb{Z}_2^1$, sin necesidad de modificaciones adicionales.

En cambio, las variables numéricas y los parámetros involucrados en combinaciones lineales requieren un proceso adicional para determinar su representación binaria óptima. La cantidad mínima de bits asignada a una variable depende de su valor máximo posible y se calcula utilizando el logaritmo en base dos, junto con la función techo:

$$N\_bits = \lceil log_2 (N) \rceil + 1$$

donde N representa el valor máximo posible de la variable. Por ejemplo, en el modelo del *Travelling Salesman Problem* (TSP) presentado en la Sección 3.2.4, la variable auxiliar u_i, que indica el orden de visita del nodo i y puede tomar valores enteros entre 0 y N, requiere exactamente N_bits bits para su representación:

$$u_i \in \mathbb{Z}_2^{N\_bits} : \mathbb{Z}_2 = \{0, 1\}; \quad \forall i \in \{1, \ldots, N\}$$

De esta forma, $\mathbb{Z}_2^N$ denota una lista de N bits, donde cada posición puede tomar el valor 0 o 1, y la concatenación de estos bits forma el número binario asociado.

Un aspecto crítico a considerar en este tipo de transformaciones es que la especificación estándar de *SMT-LIB* [110] define los vectores de bits con un tamaño fijo. Por ello, si una variable no dispone de suficientes bits para representar el valor máximo posible dentro del dominio del problema, se produce una pérdida de información por *overflow binario*. Para mitigar este riesgo, los solvers modernos —como *Z3*— incluyen operadores específicos que permiten extender temporalmente el tamaño de los vectores (rellenando con ceros a la izquierda) y distinguen entre operaciones con números con signo (*signed*) y sin signo (*unsigned*), asegurando así la consistencia aritmética del modelo.

La siguiente formulación muestra el modelo completo del *TSP* transformado completamente hacia `QF_BV`, utilizando codificación binaria para todas las variables y parámetros:

Parámetros:

- N : Número total de nodos, $N \in \mathbb{Z}^+$
- N^* : Representación binaria de N, $N^* \in \mathbb{Z}_2^{\lceil \log_2(N) \rceil + 1}$
- $D_{i,j}$: Distancia entre los nodos i y j, $D_{i,j} \in \mathbb{Z}_0^+$; $\quad \forall i, j \in \{1, ..., N\}$
- $D_{i,j}^*$: Representación binaria de las distancias, $D_{i,j}^* \in \mathbb{Z}_2^{\lceil \log_2(\text{máx}(D_{i,j})) \rceil + 1}$; $\quad \forall i, j \in \{1, ..., N\}$

Variables:

- $x_{i,j} \in \mathbb{Z}_2^1$: indica si el nodo j es visitado después del nodo i.
- $u_i \in \mathbb{Z}_2^{\lceil \log_2(N) \rceil + 1}$: variable auxiliar para definir el orden del recorrido y evitar *sub-tours*.

Función objetivo:

- Minimizar distancia total recorrida d

$$min \quad d = \sum_{i=1}^{N} \sum_{j=1}^{N} \left(D^*_{i,j} \times x_{i,j} \right)$$

Restricciones del problema:

1. **Dominio de las variables:** Cada variable $x_{i,j}$ solo puede tomar los valores 0_2 o 1_2, indicando la existencia o no de un trayecto directo entre los nodos i y j.:

$$\bigvee_{k=0_2}^{1_2} x_{i,j} = k; \quad \forall i, j \in \{1, ..., N\}$$

2. **Dominio de las variables auxiliares:** Las variables u_i se restringen a vectores de bits con valores entre 1_2 y N^*, representando la posición del nodo i en el recorrido.

$$\bigvee_{k=1_2}^{N^*} u_i = k; \quad \forall i \in \{1, ..., N\}$$

3. **Cada ciudad debe ser punto de partida una sola vez:** Desde cada nodo i debe existir exactamente una arista de salida hacia otro nodo j:

$$\sum_{j=1_2}^{N^*} x_{i,j} = 1_2; \quad \forall i \in \{1, ..., N\}, \quad i \neq j$$

4. **Cada ciudad debe ser punto de llegada una sola vez:** Hacia cada nodo j debe llegar exactamente una arista proveniente de otro nodo i:

$$\sum_{i=1_2}^{N^*} x_{i,j} = 1_2; \quad \forall j \in \{1, ..., N\}, \quad i \neq j$$

5. **Eliminación de subtours (restricciones de Miller–Tucker–Zemlin [172]):** Estas restricciones impiden la formación de ciclos parciales que no incluyan a todos los nodos del grafo:

$$u_i + N^* \times x_{i,j} \leq N^* - 1_2 + u_j; \quad \forall i, j \in \mathbb{Z}^+ : \quad 2 \leq i \leq N, \quad 1 \leq j \leq N$$

6. **Evitar visitas repetidas:** Dos nodos no pueden ocupar la misma posición dentro del recorrido:

$$u_i \neq u_j; \quad \forall i, j \in \{1, ..., N\} : \quad i \neq j$$

7. **Prohibición de bucles (self-loops):** No se permite que un nodo se visite a sí mismo:

$$x_{i,i} = 0_2; \quad \forall i \in \{1, ..., N\}$$

Esta formulación permite representar de manera compacta la estructura combinatoria del problema y reduce el espacio de búsqueda a operaciones bit a bit, lo que resulta especialmente eficiente en hardware y contextos donde el solver puede explotar paralelismo a nivel de bits. En consecuencia, la lógica `QF_BV` ofrece una alternativa potente frente a `LIA` en aquellos casos donde el dominio de las variables sea acotado y discreto, contribuyendo significativamente a mejorar los tiempos de resolución en entornos de optimización híbrida.

3.3.3. Transformación de `LIA` a `QF_ALIA`

La lógica `QF_ALIA` (*Quantifier-Free Arrays with Linear Integer Arithmetic*) extiende la teoría `QF_AX` mediante la integración de estructuras de arreglos y operadores aritméticos lineales sobre el dominio de los enteros. Esta combinación permite modelar y resolver problemas que involucran estructuras indexadas o de tipo matricial, manteniendo la expresividad de `LIA` y añadiendo capacidades para representar dependencias entre conjuntos de variables o elementos. En esencia, la lógica de arreglos con extensionalidad posibilita el manejo de estructuras con un número potencialmente infinito de estados, mientras que los solvers *SMT* aplican razonamiento deductivo para explorar únicamente los estados relevantes en cada instancia.

En su forma pura, la teoría `QF_AX` describe los arreglos como una triada formada por un índice, un elemento y la estructura contenedora —el propio arreglo—. La propiedad de extensionalidad garantiza que dos arreglos son iguales si y solo si contienen los mismos elementos en todas las posiciones indexadas. Sin embargo, la teoría básica `QF_AX` carece de operadores aritméticos, por lo que extenderla con teorías como `LIA` o `LRA` resulta más natural y apropiado para el modelamiento de problemas de optimización lineal como los considerados en este trabajo.

En este contexto, se adopta la extensión `QF_ALIA`, que combina el razonamiento aritmético lineal de `LIA` con la teoría de arreglos, permitiendo construir modelos estructurados sin

alterar sustancialmente las restricciones del problema original. La estructura formal de un arreglo en esta teoría puede expresarse como:

$$\mathcal{A}_I^E = (\Sigma, C)\,,$$

donde $\mathcal{A}$ denota la teoría de arreglos con índices I y elementos E, y los conjuntos Σ y C representan los símbolos y las constantes que relacionan la teoría de índices con la de elementos [173].

En el presente trabajo se asume que tanto los índices como los elementos pertenecen al conjunto de los números enteros, de manera que un índice I_i apunta siempre hacia un elemento E_i dentro del arreglo $\mathcal{A}$, preservando así la correspondencia ordenada entre índices y valores, $\forall i \in \mathbb{Z}$.

Para transformar un modelo formulado en `LIA` hacia `QF_ALIA`, cada variable se convierte en un arreglo cuyos índices y elementos pertenecen al dominio de los enteros. Por ejemplo, la variable auxiliar del modelo del *Travelling Salesman Problem* (TSP) descrita en la Sección 3.2.4,

$$u_i \in \mathbb{Z}, \quad \forall i \in \{1, \ldots, N\},$$

se representa en `QF_ALIA` como:

$$u \in \mathcal{A}_I^E, \quad \forall I, E \in \mathbb{Z}$$

De este modo, la variable u se convierte en un arreglo cuyos índices corresponden a los nodos del grafo, y cuyos elementos almacenan el orden de visita asociado a cada nodo.

Las variables sin subíndices se tratan como arreglos unidimensionales de tamaño uno, mientras que aquellas con dos o más subíndices se modelan como arreglos multidimensionales, en los que cada índice apunta a un nuevo arreglo o a un elemento numérico según la estructura original del modelo.

La teoría de arreglos introduce además una nueva notación para expresar las restricciones. Por ejemplo, una inecuación definida en `LIA` como:

$$x_{i,j} \geq 5, \quad \forall i, j \in \{1, \ldots 5\}$$

se traduce en `QF_ALIA` como:

$$\texttt{select}\left(\texttt{select}\left(x, i\right), j\right) \geq 5, \quad \forall i, j \in \{1, ...5\},$$

en donde la función `select` permite acceder al elemento almacenado en la posición (i, j) del arreglo.

Para mantener una notación más legible y cercana a la empleada en lenguajes de programación, se utiliza la forma indexada:

$$x[i][j] \geq 5, \quad \forall i, j \in \{1, ...5\},$$

conservando la semántica formal de la teoría y facilitando la comparación directa con los modelos basados en `LIA`.

A continuación se muestra el modelo completo del *TSP* transformado a `QF_ALIA`:

Parámetros:

- N : Número total de nodos, $N \in \mathbb{Z}^+$
- $D_{i,j}$: Distancia entre los nodos i y j, $D_{i,j} \in \mathbb{Z}_0^+, \forall i, j \in \{1, \ldots, N\}$

Variables:

- x: Arreglo bidimensional de enteros, donde $x[i][j] = 1$ si el nodo j es visitado inmediatamente después del nodo i, y 0 en caso contrario; $x \in \mathcal{A}_I^E, \quad \forall I \in \mathbb{Z}, \forall E \in \mathcal{A}_{I^*}^{E^*}, \forall I^*, E^* \in \mathbb{Z}$
- u: Arreglo unidimensional auxiliar de enteros, define el orden de visita de los nodos y evita la formación de *sub-tours*, $u \in \mathcal{A}_I^E, \forall I, E \in \mathbb{Z}$

Función objetivo:

- Minimizar distancia total recorrida d:

$$min \quad d = \sum_{i=1}^{N} \sum_{j=1}^{N} \left(D_{i,j} \times x[i][j] \right)$$

Restricciones del problema:

1. **Dominio de las variables:** Cada variable $x[i][j]$ solo puede tomar los valores 0 o 1, indicando la existencia o no de un trayecto directo entre los nodos i y j.:

$$\bigvee_{k=0}^{1} x[i][j] = k; \quad \forall i, j \in \{1, \ldots, N\}$$

2. **Dominio de las variables auxiliares:** Las variables $u[i]$ se restringen a valores entre 1 y n, representando la posición del nodo i en el recorrido.

$$\bigvee_{k=1}^{N} u[i] = k; \quad \forall i \in \{1, \ldots, N\}$$

3. **Cada ciudad debe ser punto de partida una sola vez:** Desde cada nodo i debe existir exactamente una arista de salida hacia otro nodo j:

$$\sum_{j=1}^{N} x[i][j] = 1; \quad \forall i \in \{1, \ldots, N\}, \quad i \neq j$$

4. **Cada ciudad debe ser punto de llegada una sola vez:** Hacia cada nodo j debe llegar exactamente una arista proveniente de otro nodo i:

$$\sum_{i=1}^{N} x[i][j] = 1; \quad \forall j \in \{1, \ldots, N\}, \quad i \neq j$$

5. **Eliminación de subtours (restricciones de Miller–Tucker–Zemlin):** Estas restricciones impiden la formación de ciclos parciales que no incluyan a todos los nodos del grafo:

$$u[i] + N \times x[i][j] \leq N - 1 + u[j]; \quad \forall i, j \in \mathbb{Z} : \quad 2 \leq i \leq N, \quad 1 \leq j \leq N$$

6. **Evitar visitas repetidas:** Dos nodos no pueden ocupar la misma posición dentro del recorrido:

$$u[i] \neq u[j]; \quad \forall i, j \in \{1, \ldots, N\} : \quad i \neq j$$

7. **Prohibición de bucles (self-loops):** No se permite que un nodo se visite a sí mismo:

$$x[i][i] = 0; \quad \forall i \in \{1, \ldots, N\}$$

La transformación de `LIA` a `QF_ALIA` mostrada conserva la estructura del modelo original introduciendo una representación matricial que facilita el manejo simbólico de relaciones entre elementos sin alterar la semántica de las restricciones. Esta formulación es especialmente útil para problemas donde la interacción entre múltiples variables puede representarse naturalmente como una estructura indexada, permitiendo a los solvers *SMT* aplicar estrategias de razonamiento más eficientes basadas en manipulación estructural de arreglos.

3.4. Combinando múltiples lógicas *SMT*

Una de las ventajas fundamentales del modelamiento de problemas mediante *SMT* es que no existe una restricción que obligue a emplear una única lógica a lo largo de todo el modelo. Esto permite formular distintas partes del problema usando lógicas diferentes —por ejemplo, `LIA`, `LRA` o `QF_BV`— y, de este modo, aprovechar las fortalezas particulares de cada solver especializado, obteniendo potencialmente soluciones en tiempos potencialmente menores que los que se requerirían al usar una sola lógica.

Para llevar a cabo esta estrategia de combinación de lógicas, se establece el siguiente marco de trabajo:

1. **Creación de variables por cada lógica utilizada:**

   Al modelar un problema con una única lógica *SMT*, cada variable de decisión posee un único dominio definido. Para la combinación de lógicas, se generan múltiples versiones de cada variable de decisión, una por cada lógica incluida en el modelo. Esto

asegura que cada restricción pueda vincularse con la versión de la variable correspondiente a la lógica seleccionada.

Por ejemplo, en un modelo de *TSP* que combine `LIA` y `LRA`, las variables se definen como:

$$x_{\texttt{LIA-}i,j} = \begin{cases} 1 & \text{si el nodo } j \text{ se visita inmediatamente después del nodo } i, \\ 0 & \text{en caso contrario} \end{cases}$$

$$x_{\texttt{LIA-}i,j} \in \mathbb{Z}; \quad \forall i, j \in \{1, \ldots, N\}$$

$$x_{\texttt{LRA-}i,j} = \begin{cases} 1.0 & \text{si el nodo } j \text{ se visita inmediatamente después del nodo } i, \\ 0.0 & \text{en caso contrario} \end{cases}$$

$$x_{\texttt{LRA-}i,j} \in \mathbb{R}; \quad \forall i, j \in \{1, ..., N\}$$

Variables auxiliares para eliminar subtours:

$$u_{\texttt{LIA-}i} \in \mathbb{Z}; \quad \forall i \in \{1, ..., N\}$$

$$u_{\texttt{LRA-}i} \in \mathbb{R}; \quad \forall i \in \{1, ..., N\}$$

2. **Asociación de cada variable con su dominio:**

   Para preservar la factibilidad del modelo final respecto al original, cada variable debe mantener su dominio acotado. Esto se logra replicando las restricciones de dominio para cada versión de la variable. En el ejemplo del *TSP* combinando `LIA` con `LRA`:

   - **Dominio de las variables binarias:** Cada variable $x_{\texttt{LIA-}i,j}$, $x_{\texttt{LRA-}i,j}$ solo puede tomar los valores 0 o 1, indicando la existencia o no de un trayecto directo entre los nodos i y j:

$$\bigvee_{k=0}^{1} x_{\texttt{LIA-}i,j} = k; \quad \forall i, j \in \{1, \ldots, N\}$$

$$\bigvee_{k=0}^{1} x_{\texttt{LRA-}i,j} = k; \quad \forall i, j \in \{1, \ldots, N\}$$

- **Dominio de las variables auxiliares para eliminar subtours:** Las variables $u_{\texttt{LIA-}i}$, $u_{\texttt{LRA-}i}$ se restringen a valores enteros entre 1 y N, representando la posición del nodo i en el recorrido:

$$\bigvee_{k=1}^{N} u_{\texttt{LIA-}i} = k; \quad \forall i \in \{1, ..., N\}$$

$$\bigvee_{k=1}^{n} u_{\texttt{LRA-}i} = k; \quad \forall i \in \{1, ..., N\}$$

3. **Asignación de cada restricción a una sola versión de la variable:**

   Cada restricción del modelo original se vincula únicamente con una versión de la variable de decisión. Esto garantiza que cada restricción se evalúe una sola vez, preservando la coherencia del espacio de soluciones. Por ejemplo, en el *TSP*, si se decide resolver las dos últimas restricciones con `LRA` y las demás con `LIA`:

   - **Restricciones del problema:**

     a) **Cada nodo debe ser punto de partida una sola vez:**
     Desde cada nodo i debe existir exactamente una arista de salida hacia otro nodo j:

$$\sum_{j=1}^{N} x_{\texttt{LIA-}i,j} = 1; \quad \forall i \in \{1, \ldots, N\}, \quad i \neq j$$

     b) **Cada nodo debe ser punto de llegada una sola vez:**

$$\sum_{i=1}^{N} x_{\texttt{LIA-}i,j} = 1; \quad \forall j \in \{1, \ldots, N\}, \quad i \neq j$$

     c) **Eliminación de subtours (restricciones de Miller–Tucker–Zemlin):**

$$u_{\texttt{LIA-}i} - u_{\texttt{LIA-}j} + N \times x_{\texttt{LIA-}i,j} \leq N - 1; \quad \forall i, j \in \mathbb{Z}^+ : \quad 2 \leq i \leq N, \quad 1 \leq j \leq N$$

     d) **Evitar visitas repetidas:**

$$u_{\texttt{LRA-}i} \neq u_{\texttt{LRA-}j}; \quad \forall i, j \in \{1, \ldots, N\} : \quad i \neq j$$

     e) **Prohibición de bucles (self-loops):**

$$x_{\texttt{LRA-}i,i} = 0; \quad \forall i \in \{1, ..., N\}$$

4. **Mapeo entre versiones de variables:**

   Al introducir múltiples versiones de una misma variable bajo distintas lógicas, se genera de manera implícita un grado de independencia entre las restricciones, ya que cada una podría operar sobre un conjunto distinto de variables sin relación directa entre sí. En ausencia de vínculos explícitos, el solver podría producir soluciones parciales o divergentes que, aunque sean válidas localmente en cada submodelo, no resultarían factibles al ser evaluadas en el modelo original unificado.

   Para evitar este problema, se agrega un conjunto de restricciones de mapeo que obliga a todas las versiones de una misma variable de decisión a mantener valores equivalentes, independientemente de la lógica con la que fueron modeladas. Este mecanismo de acoplamiento asegura la coherencia global del modelo y preserva la validez estructural de la solución combinada.

   Siguiendo con el ejemplo del *TSP* modelado mediante `LIA` y `LRA`, el mapeo se define de la siguiente forma:

   - **Mapeo para las variables binarias**:

   $$x_{\texttt{LIA-}i,j} = x_{\texttt{LRA-}i,j}; \quad \forall i, j \in \{1, \ldots, N\}$$

   - **Mapeo para las variables auxiliares:**

   $$u_{\texttt{LIA-}i} = u_{\texttt{LRA-}i}; \quad \forall i \in \{1, \ldots, N\}$$

   Estas restricciones adicionales de mapeo garantizan que todas las expresiones lógicas y numéricas del modelo compartan una base equivalente, permitiendo que cada parte del modelo, independiente de la lógica utilizada, contribuya de manera coherente al espacio global de soluciones.

En conjunto, las reglas presentadas en esta sección proporcionan un marco metodológico sólido para construir modelos híbridos capaces de integrar múltiples lógicas *SMT*. Más allá de su valor conceptual, estas reglas sientan las bases operativas para la búsqueda automática de configuraciones lógicas eficientes: al disponer de un modelo que puede mezclar y

relacionar distintas lógicas de manera controlada, se habilita la posibilidad de explorar sistemáticamente estas combinaciones en busca de aquellas que optimicen el desempeño del solver.

3.5. Búsqueda de combinaciones

El problema de encontrar la mejor combinación posible de lógicas *SMT* dentro de un modelo de optimización y satisfacción de restricciones puede formalizarse naturalmente como un **problema de búsqueda en el espacio de combinaciones de lógicas**. Cada restricción y función objetivo del modelo puede formularse bajo distintas lógicas —por ejemplo, `LIA`, `LRA`, `QF_BV` o `QF_ALIA`—, generando un espacio combinatorio de tamaño exponencial conforme aumenta el número de componentes del problema. Explorar exhaustivamente todas las combinaciones posibles sería computacionalmente inviable, ya que implicaría resolver un modelo distinto por cada configuración lógica posible, lo que se traduce en miles o incluso millones de ejecuciones del solver *OMT*.

Ante esta complejidad, el proceso de búsqueda requiere una estrategia que priorice la exploración hacia regiones del espacio de soluciones con mayor potencial de mejora, evitando evaluaciones redundantes o poco prometedoras. En este contexto, se propone utilizar el algoritmo de *Hill Climbing*, una técnica de búsqueda local que permite avanzar iterativamente desde una solución inicial —correspondiente a un modelo homogéneo en una única lógica— hacia configuraciones mixtas que presenten un desempeño mejor. Este enfoque se justifica por su bajo costo computacional en comparación a otras heurísticas, su simplicidad de implementación y su capacidad para descubrir combinaciones eficientes sin necesidad de recorrer el espacio completo.

De este modo, el problema de determinar qué combinación de lógicas *SMT* ofrece los mejores tiempos y resultados se aborda como una búsqueda sobre un espacio discreto, donde cada estado representa una asignación particular de lógicas a las restricciones del modelo. El uso de *Hill Climbing* permite guiar esta búsqueda mediante evaluaciones sucesivas del desempeño del solver, ajustando la configuración del modelo en función de los resultados

obtenidos y avanzando hacia combinaciones cada vez más competitivas.

3.5.1. Hill Climbing

Para el desarrollo de este sistema basado en *Hill Climbing* usando alguna mejora, se definen las siguientes componentes estructurales:

Representación de la Solución

Cada solución se representa como un vector de tamaño $n_o + n_c$, donde n_o corresponde al número de funciones objetivo del problema y n_c al número de restricciones. Los primeros n_o elementos del vector codifican la lógica utilizada para modelar la(s) función(es) objetivo, mientras que los n_c elementos restantes representan, en el mismo orden en el que fueron formuladas, las lógicas utilizadas en cada una de las restricciones del modelo matemático. Cada posición del vector almacena explícitamente la lógica *SMT* empleada para modelar esa expresión específica.

Por ejemplo, el vector [`LIA`, `LRA`, `LIA`, `QF_BV`, `QF_ALIA`] para un problema con $n_o = 1$ y $n_c = 4$ indica que la función objetivo (índice 1) y la segunda restricción (índice 3) fueron modeladas utilizando `LIA`, mientras que la primera restricción (índice 2) se expresó con `LRA`, la tercera (índice 4) con `QF_BV` y la cuarta y última restricción (índice 5) con `QF_ALIA`.

Inicialización de la Solución

La solución inicial o punto de partida del algoritmo de búsqueda corresponde al vector descrito en la Representación de la Solución, con tamaño $n_o + n_c$, pero con todos sus valores inicializados en `LIA`. Es decir, el sistema comienza con un modelo completamente formulado en lógica de enteros (`LIA`), y a partir de esta configuración base se exploran combinaciones alternativas de lógicas que puedan mejorar el desempeño.

Adicionalmente, se consideran tres puntos de partida alternativos para aumentar la probabilidad de encontrar soluciones superiores: se inicializa el mismo vector, pero asignando en su totalidad `LRA`, `QF_BV` y `QF_ALIA`, respectivamente. De este modo se obtienen cuatro soluciones iniciales puras —una por cada lógica— que servirán tanto como punto de inicio para la búsqueda como referencia comparativa frente a las soluciones híbridas generadas posteriormente.

Evaluación de la Solución

Cada solución candidata es evaluada utilizando el solver *OMT* de *Z3*. Para ello, se realiza primero la traducción del modelo original —formulado en dominio entero (`LIA`)— hacia las lógicas especificadas en cada posición del vector solución. Una vez finalizada la traducción, se genera el archivo correspondiente en formato `.smt2` y se ejecuta el solver sobre dicho modelo.

Antes de resolver cada instancia se establecen parámetros de control para acotar el tiempo máximo de ejecución y el uso máximo de memoria. Si alguno de estos límites es superado, la ejecución es detenida inmediatamente y se registran los valores observados hasta ese momento.

Los parámetros utilizados son los siguientes:

- Máximo uso de *RAM* permitido para el solver: 4048 *MB*
- Máximo tiempo de ejecución permitido: $1, 10, 100, 1000$ segundos

De esta forma, cada solución candidata es evaluada utilizando como máximo 4048 *MB* de memoria y uno de los límites de tiempo definidos, en función de la iteración del proceso de búsqueda.

Durante la resolución se registran el valor alcanzado por la función objetivo (o el estado de *SAT*, si no existe función objetivo), junto con el tiempo total y la memoria utilizada por el solver. Estos valores son empleados para comparar soluciones y determinar si una nueva configuración representa una mejora respecto de las anteriores.

Generación del Vecindario

En esta implementación se consideran dos estrategias alternativas para construir el vecindario de soluciones. La primera, denominada *abanico*, consiste en seleccionar aleatoriamente un índice del vector solución y generar tres nuevos vecinos reemplazando la lógica presente en ese índice por cada una de las otras tres lógicas posibles. Por ejemplo, si la solución actual es `LIA`, `LIA` y se selecciona aleatoriamente el índice 1, las soluciones vecinas serán `LRA`, `LIA`, `QF_BV`, `LIA` y `QF_ALIA`, `LIA`.

La segunda estrategia, llamada *aleatoria*, genera explícitamente cuatro vecinos. Para ello, se selecciona también un índice al azar, pero esta vez se reemplaza la lógica escogida por otra lógica distinta elegida aleatoriamente, repitiendo este proceso hasta construir cuatro vecinos. Por ejemplo, un posible vecindario para la solución `LIA`, `LIA` es: `LRA`, `LIA`, `LIA`, `QF_ALIA`, `QF_ALIA`, `LIA` y `LIA`, `LRA`.

Condición de Cambio de Solución y Condición de Término

Se utiliza el criterio clásico de *primera mejora*: la solución actual es reemplazada inmediatamente cuando se encuentra la primera solución vecina que presenta una evaluación superior.

Para ello, se recorren secuencialmente todas las soluciones del vecindario. Una solución candidata se considera mejor que la actual si (i) obtiene un valor mejor en la función objetivo, o bien, en caso de empate, (ii) presenta un menor tiempo de ejecución, o, si el empate persiste, (iii) requiere un menor consumo de memoria *RAM*. En cuanto una solución candidata cumple estas condiciones, se reemplaza a la solución actual y el proceso continúa generando un nuevo vecindario desde esta nueva configuración. Si, tras examinar todas las soluciones vecinas, ninguna resulta mejor que la actual, el algoritmo finaliza y devuelve dicha solución como resultado.

Iteraciones de *Hill Climbing*

El procedimiento de búsqueda mediante *Hill Climbing* se inicia siempre desde un vector solución inicial, en el cual todas las restricciones del modelo están formuladas utilizando una única lógica *SMT*. A partir de este vector, el algoritmo genera soluciones vecinas candidatas y, a medida que son evaluadas, actualiza de forma inmediata la mejor solución encontrada hasta ese momento.

En una primera etapa, se aplica la estrategia de generación de vecindario tipo *abanico*. Esta estrategia se ejecuta iterativamente bajo distintos límites de tiempo, utilizando siempre los mismos cuatro vectores iniciales como puntos de partida. Primero se realiza una búsqueda con un límite de 1 segundo; luego, este límite se incrementa a 10 segundos, posteriormente a 100 segundos, y finalmente a 1000 segundos. Para cada límite temporal, se realizan cuatro ejecuciones independientes (una por cada solución inicial), produciendo así 16 soluciones finales asociadas exclusivamente a la estrategia *abanico*.

Una vez completado este proceso, se repite exactamente el mismo esquema utilizando la estrategia alternativa de generación de vecindario tipo *aleatorio*. Esto implica nuevamente buscar primero con un límite de 1 segundo, luego con 10 segundos, después con 100 segundos y finalmente con 1000 segundos, siempre desde los mismos cuatro vectores solución iniciales. De este modo, se generan otras 16 soluciones finales adicionales. En total, el sistema produce 32 soluciones finales: 16 asociadas a la estrategia *abanico* y 16 a la estrategia *aleatoria*, todas con desempeño igual o superior al de sus respectivas soluciones de partida.

Pseudocódigo del procedimiento de búsqueda y selección de combinaciones de lógicas

Algoritmo 1 Procedimiento general de *Hill Climbing* con estrategias Abanico y Aleatoria

```
1:  por estrategia ∈ {ABANICO, ALEATORIO} hacer
2:      por tiempo_max ∈ {1, 10, 100, 1000} hacer
3:          por S_inicial ∈ SOLUCIONES_INICIALES hacer
4:              S_actual ← S_inicial
5:              Evaluar S_actual                          ▷ función objetivo, tiempo, RAM
6:              Vecinos ← GenerarVecindario(S_actual, estrategia)
7:              mejora ← falso
8:              por V ∈ Vecinos hacer
9:                  Evaluar V
10:                 si V mejora a S_actual entonces
11:                     S_actual ← V
12:                     mejora ← verdadero
13:                     detener                           ▷ criterio de primera mejora
14:                 fin si
15:             fin por
16:             si no mejora entonces
17:                 detener                               ▷ no hay mejor vecino, fin de la búsqueda
18:             fin si
19:             Guardar S_actual como solución final de (estrategia, tiempo_max, S_inicial)
20:         fin por
21:     fin por
22: fin por
```

El procedimiento descrito en el Algoritmo 1 constituye el núcleo experimental de este trabajo, ya que permite evaluar de manera sistemática la eficiencia y desempeño de modelos *SMT* formulados con combinaciones híbridas de lógicas. A través del algoritmo de *Hill Climbing*, se exploran distintos espacios de búsqueda definidos por la estructura de los modelos y las configuraciones lógicas posibles, identificando aquellas combinaciones que ofrecen un mejor

equilibrio entre calidad de solución, tiempo de ejecución y uso de recursos computacionales. Este enfoque iterativo y controlado no solo proporciona una base empírica sólida para comparar el impacto de cada lógica en el rendimiento del solver, sino que también posibilita avanzar hacia el objetivo central de esta investigación: desarrollar un mecanismo automatizado capaz de seleccionar, transformar y combinar lógicas *SMT* de forma inteligente, optimizando así la resolución de problemas de satisfacción de restricciones y optimización mediante *OMT*.

Una vez completada la búsqueda iterativa de combinaciones de lógicas mediante el algoritmo de *Hill Climbing*, se incorpora una etapa adicional de filtrado destinada a depurar los resultados obtenidos y conservar únicamente las configuraciones más representativas, procedimiento descrito en el Algoritmo 2.

Algoritmo 2 Procedimiento de filtrado de combinaciones de lógicas encontradas

```
1:  por problema ∈ {UKP, BACP, NSP, TSP, NQUEENS} hacer
2:      por instancia ∈ ObtenerInstancias(problema) hacer
3:          mejor_combinacion ← NULL
4:          por combinacion ∈ ObtenerCombisDeLogicas(problema, instancia) hacer
5:              si combinacion encontró solución entonces
6:                  mejor_combinacion ← ActualizarCombi(mejor_combinacion, combinacion)
7:              fin si
8:          fin por
9:          GuardarMejorCombinacion(problema, instancia, mejor_combinacion)
10:     fin por
11: fin por
```

En esta fase, el sistema analiza para cada instancia las 32 combinaciones de lógicas generadas durante la búsqueda y selecciona la combinación con el mejor desempeño. Se considera como "mejor" aquella configuración que logra una solución más cercana al óptimo; en caso de empate, se prioriza la que requiere un menor tiempo de resolución; y, si el empate persiste, se elige aquella que utiliza una menor cantidad de memoria *RAM*.

El resultado de este proceso de filtrado es un conjunto formado por las n mejores combinaciones de lógicas, o un número menor, en caso de que alguna instancia no haya podido

resolverse, donde *n* corresponde al total de instancias consideradas para el problema. Estas combinaciones seleccionadas avanzan luego a una etapa final de validación, en la cual, junto con los modelos construidos a partir de una sola lógica, se emplean para resolver nuevamente todas las instancias del problema bajo una configuración más relajada: un *timeout* de 3600 segundos y un límite máximo de 6144 *MB* de memoria *RAM*. Esta validación permite evaluar en condiciones homogéneas el desempeño real de cada combinación candidata y determinar cuáles presentan ventajas significativas frente a los modelos basados en una única lógica.

3.5.2. Comparación con *Hill Climbing* Convencional

El algoritmo descrito en la Sección 3.5.1 se basa en *Hill Climbing*, pero incorpora similitudes y modificaciones deliberadas respecto a su versión convencional. La Tabla 3.7 resumen las diferencias y similitudes entre el enfoque convencional y el utilizado en este trabajo.

A pesar de las extensiones incorporadas, el algoritmo propuesto mantiene las limitaciones inherentes al paradigma de búsqueda local. En primer lugar, ambas variantes son susceptibles a quedar atrapadas en *óptimos locales*: cuando ningún vecino inmediato mejora la solución actual, la búsqueda se detiene aunque existan configuraciones superiores en regiones más alejadas del espacio combinatorio. En segundo lugar, la calidad del resultado depende críticamente del punto de partida, aspecto que en el *Hill Climbing* convencional se mitiga típicamente con reinicios aleatorios y que aquí se aborda mediante múltiples inicializaciones haciendo uso de las combinaciones homogéneas de lógicas. Por último, el costo de evaluar cada vecino, que implica compilar y resolver un modelo *SMT* completo, introduce una granularidad de evaluación inusualmente costosa respecto al *Hill Climbing* estándar, lo que hace especialmente relevante la estrategia de *timeouts* progresivos para contener el presupuesto computacional total.

En conjunto, las extensiones descritas convierten al algoritmo propuesto en un esquema de búsqueda local multi-inicio y multi-estrategia, que retiene la simplicidad e interpretabilidad del *Hill Climbing* convencional mientras mitiga sus principales vulnerabilidades en el contexto específico de la selección de lógicas *SMT*, permitiendo encontrar un mayor número de soluciones posibles y aumentando la probabilidad de acercarse al óptimo global.

Tabla 3.7: Comparación entre *Hill Climbing* convencional y la variante propuesta en este trabajo, organizada por dimensiones de diseño del algoritmo de búsqueda.

Dimensión	**HC convencional**	**HC propuesto (este trabajo)**
Punto de partida	Un único estado inicial, típicamente aleatorio o fijo.	Cuatro soluciones iniciales puras (`LIA`, `LRA`, `QF_BV`, `QF_ALIA`), reduciendo el riesgo de quedar atrapado en regiones desfavorables del espacio desde el inicio.
Generación del vecindario	Un único tipo de movimiento, generalmente con tamaño de vecindario fijo.	Dos estrategias complementarias: *Abanico* (exhaustiva en una posición) y *Aleatoria* (diversificadora entre posiciones), aumentando la cobertura efectiva del espacio sin incrementar el costo por iteración.
Criterio de aceptación	Primera mejora o mejor vecino según una única métrica.	Primera mejora con criterio lexicográfico tri-objetivo: (1) calidad de la solución, (2) tiempo de CPU y (3) memoria RAM, permitiendo desempatar entre configuraciones con igual valor objetivo.
Presupuesto de evaluación	Número fijo de iteraciones o una condición de término única.	Cuatro etapas de *timeout* crecientes (1, 10, 100 y 1000 s) por estrategia, implementando un esquema de búsqueda progresiva que prioriza soluciones rápidas y refina iterativamente.
Número de ejecuciones	Una sola cadena de búsqueda por ejecución.	32 ejecuciones independientes (2 estrategias $\times$ 4 *timeouts* $\times$ 4 inicializaciones), cuyo resultado final es filtrado por el criterio de mejor solución global descrito en el Algoritmo 2.

Capítulo 4

Resultados

4.1. Configuración Experimental

Para resolver los modelos desarrollados en la Sección 3.2 y procesar las instancias descritas en la Sección 3.1, se utilizó el *solver* **Z3** de *Microsoft Research* [56], junto con su *API* para *Python* [174]. Esta herramienta permitió automatizar la generación y resolución de modelos, construyendo de manera programática los archivos en formato estándar *SMT-LIB 2.6* [110] a partir de las expresiones matemáticas previamente definidas.

Durante la generación de los modelos, se estableció explícitamente el estado inicial como `status unknown`, indicando al *solver* que debe iniciar la búsqueda desde un estado no determinado (*unknown* o *not sat*). Asimismo, no se fijó una lógica específica de manera manual, permitiendo que Z3 determine de forma automática la estrategia más apropiada en función de la estructura del modelo.

Una vez generados los archivos en formato `.smt2`, cada uno fue resuelto utilizando la aplicación de consola de Z3, ejecutando el *solver* con las configuraciones necesarias para la búsqueda de soluciones óptimas mediante técnicas de *Optimization Modulo Theories (OMT)*.

Las ejecuciones se realizaron con la siguiente configuración de parámetros:

- `-smt2`: Indica que el archivo de entrada está en el formato estándar *SMT-LIB 2.6*.
- `-st`: Solicita al *solver* un resumen técnico al finalizar la ejecución, incluyendo métricas relevantes como el tiempo total empleado y la memoria utilizada.
- `-model`: Ordena a Z3 incluir en la salida el modelo con los valores encontrados, ya que por defecto sólo reporta el estado `sat` o `unsat`.
- `-T < hard-timeout >`: Define un límite de tiempo máximo de búsqueda (*hard timeout*), tras el cual el proceso se interrumpe forzosamente.
- `-t < soft-timeout >`: Especifica un límite de tiempo flexible (*soft timeout*) que permite al solver ajustar dinámicamente su estrategia antes de alcanzar el *hard timeout*.
- `-memory 4048`: Restringe el uso máximo de memoria RAM disponible a 4048 MB para almacenar cálculos intermedios y estructuras de datos internas.

Lo que produce la siguiente línea de comando para resolver un modelo en formato `.smt2` con *Z3* con un tiempo límite de 1000 segundos:

```
z3 -st -model -t:1000000 -T:1000 -memory:4048 -smt2 ./modelo.smt2
```

Cada proceso de búsqueda mediante *Hill Climbing* se ejecutó bajo cuatro configuraciones de *timeout* distintas: 1, 10, 100 y 1000 segundos. Una vez alcanzado cada límite de tiempo (o una cantidad cercana a éste), se detiene la ejecución aún activa y se registran las soluciones parciales encontradas junto con las métricas de desempeño asociadas.

Configuración del entorno de ejecución

Los experimentos se llevaron a cabo en un equipo de cómputo con las siguientes características técnicas:

- **Sistema operativo:** Windows 11 (64 bits)

- **Versión del solver Z3:** 4.12.2
- **Procesador:** Intel Core i7-8700K (6 núcleos, 12 hilos, 3.7 GHz base / 4.7 GHz boost)
- **Memoria RAM disponible:** 39.9 GB DDR4 @ 3200 MHz

Esta configuración permite, mediante un entorno controlado y reproducible, garantizar la consistencia de los resultados obtenidos entre diferentes ejecuciones y configuraciones de búsqueda, manteniendo un equilibrio entre capacidad de cómputo y tiempos de resolución adecuados para el volumen total de instancias evaluadas.

4.2. Resultados Experimentales

A continuación se presentan los resultados obtenidos al aplicar el procedimiento de búsqueda basado en *Hill Climbing* sobre los modelos formulados en distintas combinaciones de lógicas *SMT*. Los resultados completos se encuentran en el directorio **Resultados de la búsqueda** del repositorio de resultados [175], donde se organizan por problema, instancia, lógica inicial y configuración de búsqueda. Cada registro incluye métricas de desempeño tales como la solución alcanzada, el tiempo de resolución, el uso de memoria y el número total de iteraciones evaluadas durante la búsqueda.

A partir de este conjunto exhaustivo de ejecuciones se aplicó el procedimiento de filtrado descrito previamente en la Sección 2, cuyo propósito es identificar la mejor combinación de lógicas entre las 32 exploradas para cada instancia por *Hill Climbing*. Las Tablas 4.1, 4.2, 4.3, 4.4, 4.5 y 4.6 presentan únicamente estas combinaciones ya filtradas, seleccionadas según los criterios definidos de calidad de solución, tiempo de resolución y uso de memoria. Cada tabla reporta exclusivamente las combinaciones correspondientes a instancias que lograron resolverse satisfactoriamente; aquellas para las cuales no se obtuvo ninguna solución válida durante el proceso de búsqueda se omiten explícitamente.

Posteriormente, las combinaciones seleccionadas fueron evaluadas nuevamente bajo una nueva configuración considerando un límite de 3600 segundos y 6144 *MB* de memoria, siguiendo el procedimiento descrito en la Sección 3.5.1. Esta etapa de validación adicional

Instancia	Tiempo (s)	Memoria (MB)	Solución	Combinacion de lógicas
4	0.01	17.72	SAT	lia-lia-alia
8	0.01	18.35	SAT	lra-lia-lra
12	0.05	19.11	SAT	lia-alia-alia
16	0.16	20.28	SAT	lia-lia-alia
20	0.19	21.45	SAT	lra-lia-lia
50	8.17	64.3	SAT	alia-alia-lra
100	249.91	225.19	SAT	alia-lia-lia
120	984.27	459.21	SAT	alia-lia-alia

Tabla 4.1:

Mejores combinaciones de lógicas encontradas para cada instancia de las *N-reinas*, generada tras filtrar las tablas registradas en el directorio **Resultados de la búsqueda** del repositorio de resultados [175]. Las instancias faltantes no lograron ser resueltas durante el proceso de búsqueda.

permite estimar su comportamiento en un escenario de resolución más amplio y verificar su estabilidad a lo largo de todas las instancias del problema.

4.2.1. Interpretación de Resultados

Las tablas con los resultados de la evaluación de las combinaciones de lógicas filtradas se encuentran en el Anexo A.2. A partir de estos datos se construyen las Figuras 4.1, 4.2, 4.3, 4.4 y 4.5, las cuales sintetizan el desempeño promedio de cada combinación de lógicas para cada problema. Cada gráfico resume tres métricas fundamentales: (i) la tasa de éxito o proporción de instancias resueltas, (ii) el tiempo promedio requerido para encontrar una solución y (iii) la memoria promedio utilizada por el solver. Estas visualizaciones permiten contrastar el rendimiento global de modelos homogéneos frente a modelos heterogéneos, identificando patrones y tendencias relevantes sobre la efectividad del modelamiento híbrido basado en lógicas *SMT*.

Instancia	Tiempo (s)	Memoria (MB)	Solución	Combinacion de lógicas
hi-n10-0-c7370	0.48	45.66	25727	alia-alia-lra
hi-n20-1-c17050	0.71	52.96	42498	alia-lra-lia
hi-n30-0-c27795	2.08	76.58	58350	lra-alia-alia
hi-n40-3-c39588	4.49	85.2	94603	alia-alia-lra
hi-n50-4-c41934	6.44	89.4	146750	alia-alia-lra
hi-n60-2-c57601	6.07	164.1	192888	alia-lia-alia
hi-n70-3-c68135	12.55	207.94	259981	lia-lia-alia
hi-n80-0-c67603	17.95	183.56	263670	alia-alia-lia
hi-n90-2-c88213	12.66	225.94	351120	alia-lra-alia
hi-n100-2-c77552	6.8	74.68	182536	alia-lia-alia

Tabla 4.2: Mejores combinaciones de lógicas encontradas para cada instancia de *UKP*, generada tras filtrar las tablas registradas en el directorio **Resultados de la búsqueda** del repositorio de resultados [175].

Instancia	Tiempo (s)	Memoria (MB)	Solución	Combinacion de lógicas
25-1-8	42.96	153.3	514	lia-lia-lia-lia-lia-lra-lia-alia-lia
50-1-8	119.73	491.88	960	lia-lia-alia-alia-lia-lra-lia-lia-lia
75-1-8	313.8	671.88	1432	alia-alia-alia-lia-alia-lia-alia-alia-alia

Tabla 4.3: Mejores combinaciones de lógicas encontradas para cada instancia de *NSP*, generada tras filtrar las tablas registradas en el directorio **Resultados de la búsqueda** del repositorio de resultados [175]. Las instancias faltantes no lograron ser resueltas durante el proceso de búsqueda.

Instancia	Tiempo (s)	Memoria (MB)	Solución	Combinacion de lógicas
five	0.02	38.24	19	`lra-lra-lia-lra-lra-lra-lra`
burma-5	0.03	36.32	22	`alia-alia-lra-alia-alia-alia-lia`
ulysses-5	0.03	36.48	34	`lia-lra-alia-lra-alia-lra-lra`
burma-6	0.03	37.26	22	`lia-lia-lia-lia-lia-lia-lra`
ulysses-6	0.04	37.46	35	`lra-lra-lra-lra-lra-lia-lra`
burma-7	0.05	38.14	22	`lra-lra-lra-lra-lra-lia-lra`
ulysses-7	0.04	38.08	35	`lra-lia-lra-lra-lra-lra-lra`
burma-8	0.63	39.04	22	`lra-alia-lra-alia-lia-lra-alia`
ulysses-8	0.3	44.38	36	`lia-lra-lia-lia-lra-lia-lia`
burma-9	1.17	43.38	23	`lia-lia-lia-lra-lia-lia-lia`
ulysses-9	0.48	47.54	45	`lra-lia-lia-lra-lia-lra-lia`

Tabla 4.4:

Mejores combinaciones de lógicas encontradas para cada instancia de *TSP* (parte 1), generada tras filtrar las tablas registradas en el directorio **Resultados de la búsqueda** del repositorio de resultados [175]. Las instancias faltantes no lograron ser resueltas durante el proceso de búsqueda.

Instancia	Tiempo (s)	Memoria (MB)	Solución	Combinacion de lógicas
burma-10	2.25	62.66	27	`lra-bv-lia-lra-alia-lra-lra`
ulysses-10	2.61	42.58	45	`lra-alia-lia-alia-lia-lra-alia`
burma-11	2.93	45.1	27	`lia-alia-lra-lia-alia-lra-lra`
ulysses-11	7.18	46.24	68	`lia-lra-lia-lra-lra-lia-lra`
burma-12	7.03	48.32	28	`lia-lra-lia-lia-lia-alia-alia`
ulysses-12	33.54	48.38	68	`alia-alia-lra-alia-alia-alia-alia`
burma-13	7.14	52.28	28	`lra-lra-lra-lia-lra-lra-alia`
ulysses-13	100.83	51.42	68	`alia-alia-alia-lia-alia-alia-lia`
burma-14	13.03	56.04	30	`alia-lia-lia-lia-lia-lia-lia`
ulysses-14	177.36	56.54	68	`alia-lra-alia-lra-lra-alia-lra`
p01	662.97	174.1	291	`lia-alia-lia-lia-lia-bv-lia`

Tabla 4.5:

Mejores combinaciones de lógicas encontradas para cada instancia de *TSP* (parte 2), generada tras filtrar las tablas registradas en el directorio **Resultados de la búsqueda** del repositorio de resultados [175]. Las instancias faltantes no lograron ser resueltas durante el proceso de búsqueda.

Instancia	Tiempo (s)	Memoria (MB)	Solución	Combinacion de lógicas
bacp8	2.01	54.46	17	`alia-alia-lia-alia-alia-alia-alia-alia-alia`
bacp10	4.36	64.34	14	`alia-lia-lra-lra-lia-lia-lia-lra-lra`
bacp12	25.62	117.52	17	`lra-lra-lra-alia-lra-lra-lra-lia-alia`

Tabla 4.6: Mejores combinaciones de lógicas encontradas para cada instancia de *BACP*, generada tras filtrar las tablas registradas en el directorio **Resultados de la búsqueda** del repositorio de resultados [175].

N-Reinas

La Figura 4.1 muestra el comportamiento promedio de cada combinación de lógicas aplicada al problema de las *N-reinas*. En general se observa un patrón estable y uniforme en los resultados: varias combinaciones alcanzan rendimientos razonablemente competitivos, aunque existen diferencias claras en el uso de recursos.

Entre todas las configuraciones evaluadas, la combinación homogénea

(`lia - lia - lia`)

destaca como la más eficiente, alcanzando una tasa de éxito perfecta (1.0) con un tiempo promedio de resolución de apenas 38.8 segundos—el menor entre todas las combinaciones—y con un uso moderado de memoria ($\approx$ 335 *MB*). Esto confirma que `LIA` captura de manera natural y altamente eficiente la estructura simétrica y matricial de las *N-reinas*.

Un segundo grupo de combinaciones exitosas lo conforman las siguientes variantes con tasas de éxito de 0.83:

(`alia - lia - lia`),

(lia - alia - alia),
(lia - lia - alia),
(alia - lia - alia),
(alia - alia - alia)

Estas combinaciones lograron resolver más del 80% de las instancias, manteniendo tiempos promedio entre 976 y 1032 segundos y consumos de memoria cercanos a 380–400 *MB*. Su desempeño confirma que la incorporación selectiva de lógicas alternativas permite sostener buenos niveles de rendimiento. Si bien, estas combinaciones no superan a la combinación puramente basada en LIA, logran superar ampliamente a las lógicas puramente basadas en números reales (LRA) y en bit-vectors (QF_BV), las que presentan tasas de éxito sustancialmente inferiores y tiempos promedio considerablemente mayores.

En contraste, las combinaciones que incluyen LRA presentan un rendimiento considerablemente inferior. Configuraciones como

(lra - lia - lra),
(lra - lra - lra),
(lra - lia - lia)

logran tasas de éxito entre 0.42 y 0.5, pero con tiempos promedio cercanos a los 2000 segundos. Esto sugiere que la relajación de dominio propia de LRA resulta poco adecuada para capturar la combinatoria estricta del problema, produciendo modelos más costosos computacionalmente y menos efectivos para resolver instancias complejas.

Finalmente, la configuración homogénea basada en QF_BV

(bv - bv - bv)

presenta un comportamiento particularmente costoso: si bien alcanza una tasa de éxito del 0.5, su uso promedio de memoria excede los 1600 *MB*, reflejando los costos computacionales asociados a la transformación de modelos basado enteros hacia bit-vectors.

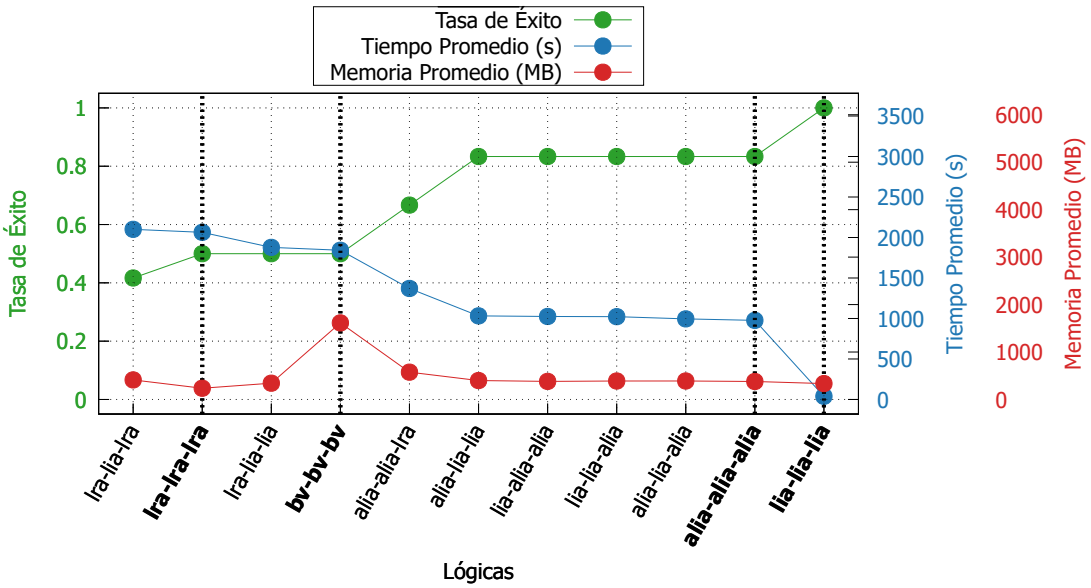


Figura 4.1: Desempeño promedio obtenido al resolver todas las instancias de las *N-Reinas* usando las mejores combinaciones de lógicas encontradas. La tabla de datos correspondiente se encuentra en el Anexo A.1. Los valores ennegrecidos en el eje horizontal representan las combinaciones lógicas homogéneas.

Unbounded Knapsack Problem (UKP)

La Figura 4.2 ilustra el comportamiento para el *Unbounded Knapsack Problem* (*UKP*), un problema dominado por combinaciones lineales simples. Esto se refleja en un patrón de resultados que privilegia las lógicas numéricas, especialmente las enteras.

Las combinaciones homogéneas muestran contrastes marcados:

- El modelo homogéneo basado en **QF_BV** obtiene una tasa de éxito igual a 0.0 y agota el tiempo máximo permitido, confirmando que la transformación de enteros a bit-vectors no fue adecuada para representar eficientemente estructuras lineales de magnitud variable típicas del *UKP*.
- **LRA** mejora marginalmente con una tasa de éxito de 0.2, pero con un tiempo promedio cercano a los 3000 segundos y un consumo de memoria mayor a 5100 *MB*, evidenciando la ineficiencia de la transformación hacia esta lógica para un problema

intrínsecamente discreto.

- **LIA** ofrece una mejora sustancial: resuelve el 100% de las instancias con un tiempo promedio de 581 segundos, alineándose con la naturaleza enteramente discreta del *UKP*.
- **QF_ALIA** muestra el mejor desempeño entre los modelos puros, logrando una tasa de éxito igual a 1.0 con un tiempo promedio de solo 7.4 segundos y un consumo de memoria de 112 *MB*. Este resultado evidencia el alto grado de eficiencia de esta lógica al manipular estructuras indexadas simples como las que aparecen en el *UKP*.

El comportamiento de las combinaciones heterogéneas resulta aún más interesante: múltiples combinaciones logran igualar la tasa de éxito perfecta de `QF_ALIA` y mantener tiempos muy similares (entre 7.8 y 8.0 segundos), además de consumos de memoria igualmente bajos. Entre las más destacadas se encuentran:

(`alia - lra - alia`),
(`alia - lra - lia`),
(`alia - alia - lia`),
(`alia - lia - alia`),
(`alia - alia - lra`)

Estos resultados sugieren que las combinaciones heterogéneas pueden aprovechar simultáneamente las capacidades aritméticas de `LIA`/`LRA` y la eficiencia indexada de `QF_ALIA`, produciendo modelos rápidos y estables.

Nurse Scheduling Problem (NSP)

La Figura 4.3 expone los resultados correspondientes al *NSP*, reconocido por su complejidad combinatorial y su estructura altamente restrictiva. En este caso, el contraste entre lógicas puras y heterogéneas se vuelve especialmente marcado.

Las configuraciones homogéneas basadas en `LIA`, `LRA` y `QF_BV` obtienen tasas de éxito iguales a 0.0, alcanzando sistemáticamente el tiempo máximo permitido de 3600 segundos y

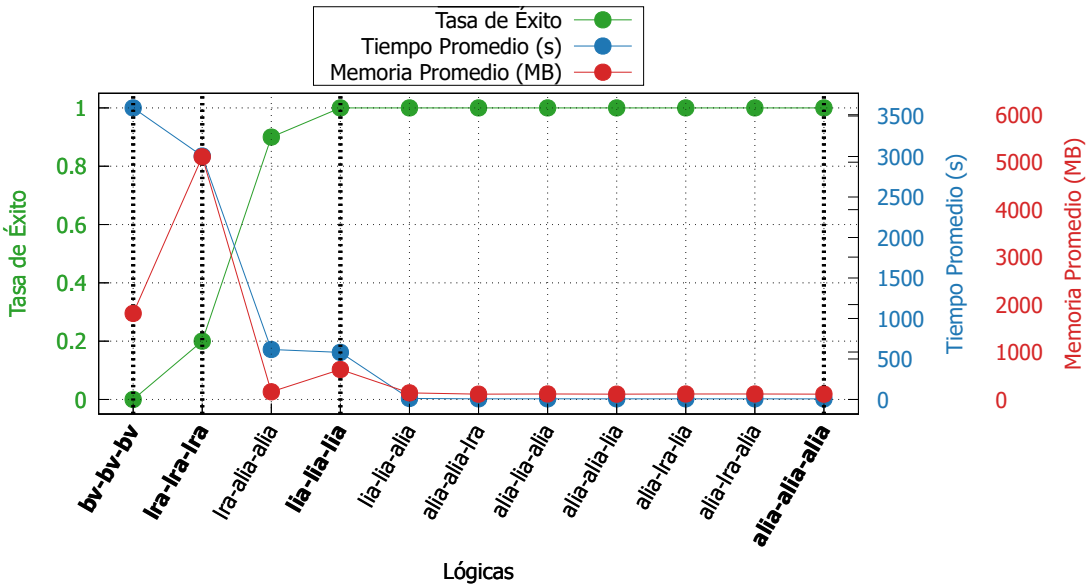


Figura 4.2: Desempeño promedio obtenido al resolver todas las instancias de *UKP* usando las mejores combinaciones de lógicas encontradas. La tabla de datos correspondiente se encuentra en el Anexo A.2. Los valores ennegrecidos en el eje horizontal representan las combinaciones lógicas homogéneas.

utilizando cantidades elevadas de memoria, en particular `LIA` y `LRA`, ambas por sobre los 4000 *MB*. Esto indica que ninguna de estas teorías, por sí sola, logra capturar adecuadamente las restricciones del *NSP*, tales como equidad, secuencias permitidas, cargas máximas y balance entre turnos.

En cambio, ciertas combinaciones heterogéneas alcanzan un rendimiento moderado. Dos de ellas:

(`lia - lia - alia - alia - lia - lra - lia - lia - lia`),
(`lia - lia - lia - lia - lia - lra - lia - alia - lia`)

logran una tasa de éxito de 0.33, con tiempos promedio entre 2430 y 2480 segundos y consumos de memoria entre 1800 y 1865 *MB*. Aunque estos valores siguen siendo elevados, representan mejoras significativas sobre los modelos homogéneos antes mencionados.

Finalmente, las combinaciones basadas mayoritariamente en `QF_ALIA` constituyen el punto

más alto de desempeño en este problema. Tanto la combinación homogénea como la heterogénea formada por las lógicas

(alia - alia - alia - lia - alia - lia - alia - alia - alia)

alcanzan una tasa de éxito de 0.5, con tiempos promedio cercanos a 1890 segundos y consumos de memoria reducidos (750 *MB*). Este resultado sugiere que el acceso indexado proporcionado por `QF_ALIA` es especialmente adecuado para capturar las dependencias secuenciales propias del *NSP*.

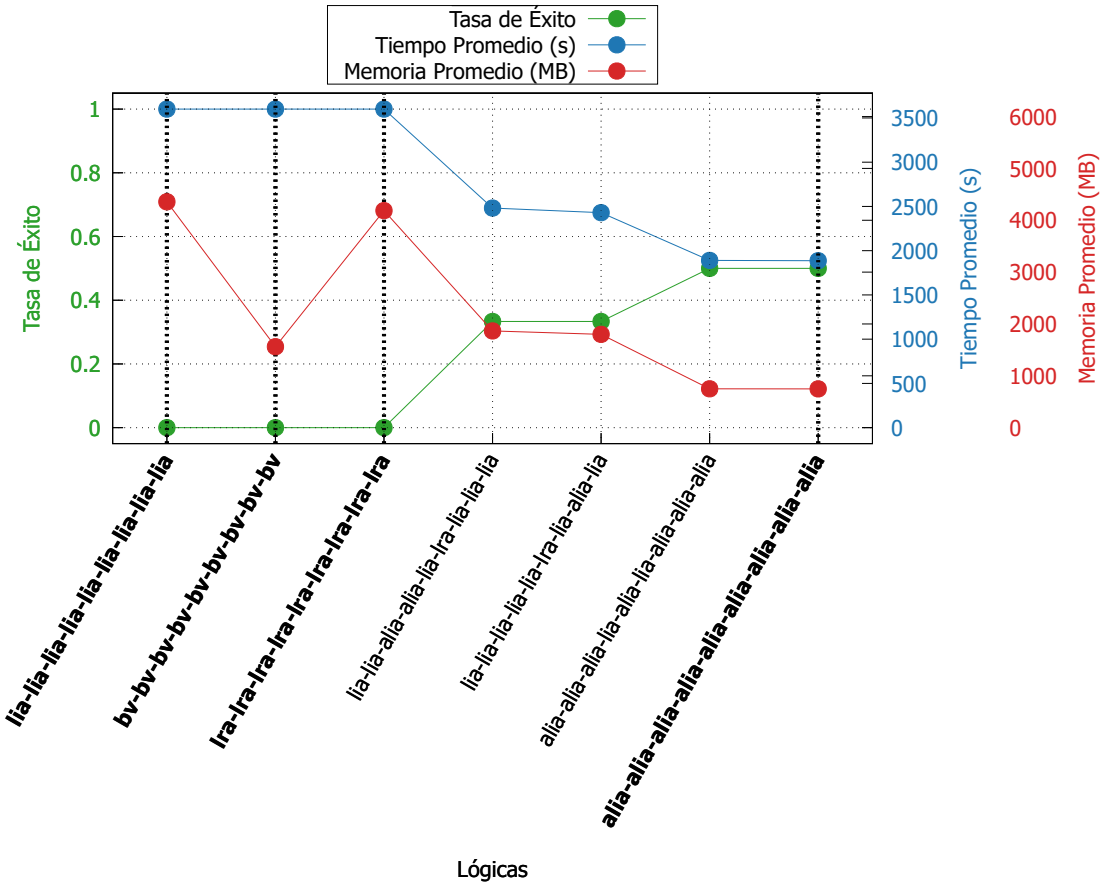


Figura 4.3: Desempeño promedio obtenido al resolver todas las instancias de *NSP* usando las mejores combinaciones de lógicas encontradas. La tabla de datos correspondiente se encuentra en el Anexo A.3. Los valores ennegrecidos en el eje horizontal representan las combinaciones lógicas homogéneas.

Travelling Salesman Problem (TSP)

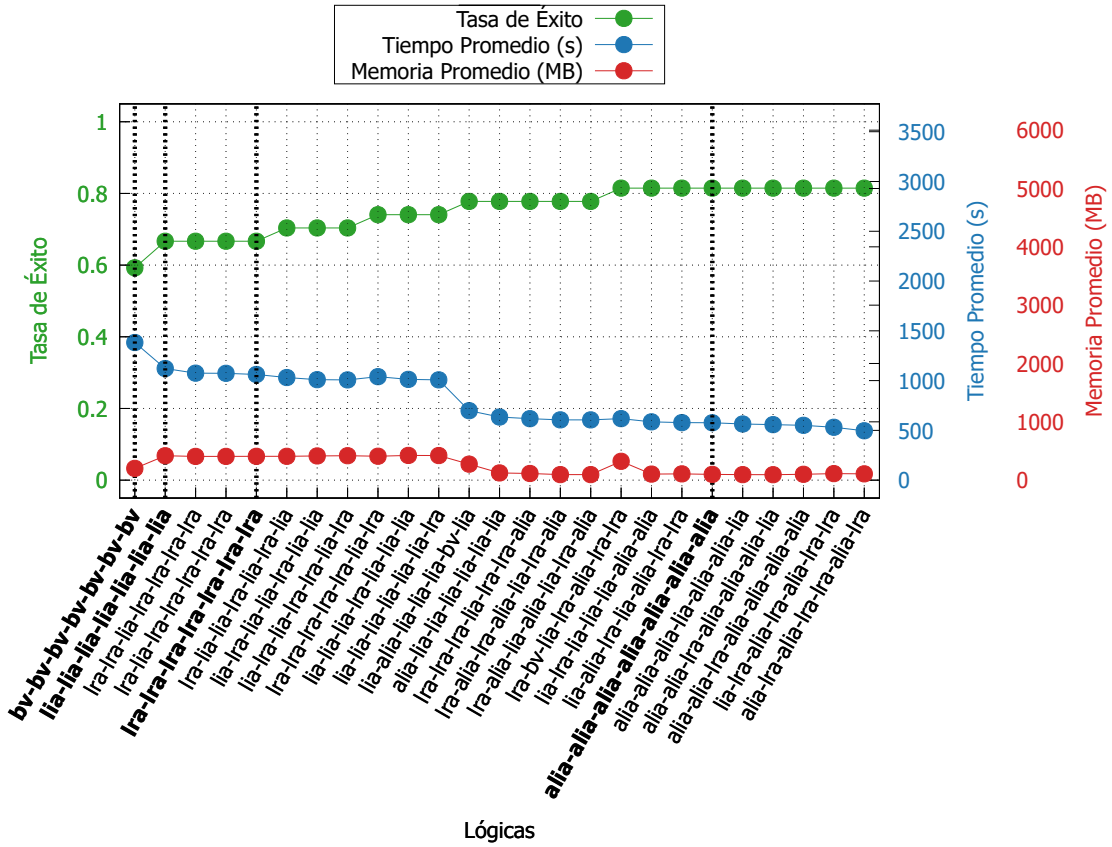


Figura 4.4: Desempeño promedio obtenido al resolver todas las instancias de *TSP* usando las mejores combinaciones de lógicas encontradas. Las tablas de datos correspondientes se encuentran en los Anexos A.4, A.5 y A.6. Los valores ennegrecidos en el eje horizontal representan las combinaciones lógicas homogéneas.

La Figura 4.4 sintetiza los resultados obtenidos para el *TSP*. A diferencia de otros problemas, este combina restricciones secuenciales, cardinalidades estrictas y dependencias lineales, lo que permite que varias teorías, tanto homogéneas como mixtas, sean capaces de modelarlo con éxito.

Las configuraciones homogéneas muestran los siguientes patrones:

- El modelo basado en `QF_BV` alcanza una tasa de éxito de 0.6, aunque con un tiempo promedio elevado de 1400 segundos.

- `LIA` y `LRA` logran tasas de éxito de 0.66, reduciendo los tiempos promedio a valores entre 1060 y 1120 segundos.
- `QF_ALIA` obtiene el mejor rendimiento entre todas las lógicas homogéneas: tasa de éxito de 0.81, tiempo promedio de 576 segundos y uso de memoria cercano a los 100 *MB*.

Sin embargo, la evidencia más relevante proviene de las combinaciones heterogéneas. Al menos cinco de ellas superan el desempeño del mejor modelo homogéneo, tanto en tasa de éxito como en eficiencia temporal. En particular,

(`alia - lra - alia - lra - lra - alia - lra`)

destaca como la mejor combinación global, alcanzando la tasa de éxito más alta entre todas las configuraciones evaluadas y reduciendo significativamente el tiempo promedio de resolución.

Balanced Academic Curriculum Problem (BACP)

La Figura 4.5 muestra los resultados para el *BACP*, uno de los problemas más sensibles a la lógica subyacente debido a su estructura secuencial y a la fuerte presencia de restricciones complejas.

Las lógicas homogéneas basadas en `LIA`, `LRA` y `QF_BV` muestran un desempeño limitado:

- `QF_BV` no resuelve ninguna instancia y agota el tiempo máximo;
- `LIA` y `LRA` resuelven solo un 33% de las instancias, con tiempos entre 2400 y 2500 segundos y memorias superiores a 500 *MB*.

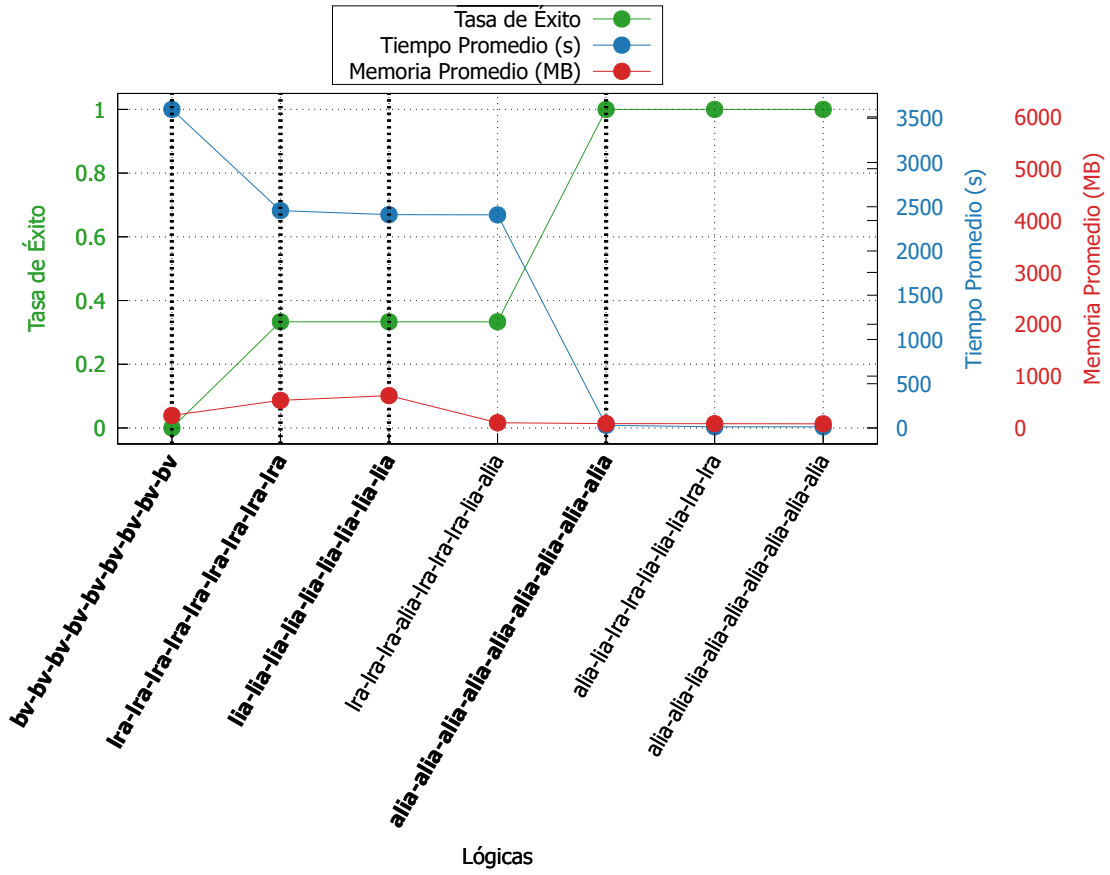


Figura 4.5: Desempeño promedio obtenido al resolver todas las instancias de *BACP* usando las mejores combinaciones de lógicas encontradas. La tabla de datos correspondiente se encuentra en el Anexo A.7. Los valores ennegrecidos en el eje horizontal representan las combinaciones lógicas homogéneas.

En cambio, `QF_ALIA` demuestra ser la teoría más adecuada para este problema. Su configuración homogénea

`(alia - alia - alia - alia - alia - alia - alia - alia - alia)`

alcanza una tasa de éxito perfecta (1.0), con un tiempo promedio de solo 30.13 segundos y memoria promedio de 83 *MB*.

Aún más destacable es el desempeño de las siguientes combinaciones heterogéneas:

(alia - lia - lra - lra - lia - lia - lia - lra - lra),
(alia - alia - lia - alia - alia - alia - alia - alia - alia)

dado que logran también una tasa de éxito igual a 1.0, pero reducen aún más el tiempo promedio de resolución a 12.7 y 13.23 segundos, respectivamente. Esto demuestra que la mezcla controlada de teorías, especialmente combinando QF_ALIA con LIA y LRA, permite capturar simultáneamente las ventajas expresivas de los arreglos y la eficiencia de las aritméticas reales y enteras, produciendo modelos sustancialmente más rápidos incluso que las mejores configuraciones homogéneas.

4.2.2. *Hill Climbing* como proceso de entrenamiento

El procedimiento de búsqueda basado en *Hill Climbing* no solo permite identificar combinaciones de lógicas eficientes, sino que además proporciona información valiosa acerca de la dificultad inherente de cada problema y de la robustez del modelo híbrido frente a distintos niveles de complejidad. Para ello, se analizan las configuraciones en las que cada combinación óptima fue descubierta, considerando dos dimensiones:

1. El límite máximo de tiempo asignado a la búsqueda (*timeout*)
2. La dificultad relativa de la instancia utilizada en esa etapa del entrenamiento

Bajo este marco, una combinación de lógicas se considera más "fácil de encontrar" cuando aparece en etapas con *timeout* reducido y en instancias de baja complejidad, mientras que configuraciones descubiertas únicamente en instancias avanzadas o con tiempos elevados reflejan una estructura de búsqueda más difícil y un espacio combinatorio más desafiante para *Hill Climbing*.

A grandes rasgos, el proceso de entrenamiento revela diferencias claras entre los problemas estudiados:

- **Problemas suaves (fáciles):** En las *N-reinas* y *UKP*, las mejores combinaciones se encontraron con límites de tiempo de 1 a 10 segundos y en las instancias más simples.

Esto indica un espacio de búsqueda relativamente suave y estable, donde pequeñas modificaciones produjeron mejoras rápidas.

- **Problemas de dificultad intermedia:** En *TSP*, las mejores combinaciones también se encuentran temprano, pero requieren mayor diversidad estructural y un conjunto más amplio de instancias para estabilizarse. El espacio de búsqueda es más amplio pero aún razonablemente transitable.
- **Problemas difíciles:** En *NSP* y *BACP*, las mejores combinaciones solo emergen en etapas tardías (100-1000 segundos) y en instancias complejas, lo que demuestra un espacio de búsqueda complejo, debido en gran parte al tamaño de las instancias utilizadas, especialmente en *NSP*.

A continuación se presenta un análisis comparativo para los cinco problemas evaluados.

N-Reinas

Los resultados de entrenamiento para las *N-reinas* muestran que las mejores combinaciones de lógicas son encontradas muy temprano en el proceso de búsqueda. La mayoría de las configuraciones óptimas aparecen con un *timeout* de 1 segundo, y en instancias pequeñas ($n = 4$, $n = 8$ y $n = 12$), lo que indica que el espacio de búsqueda es relativamente suave y que pequeñas modificaciones en la combinación de lógicas permiten mejoras rápidas y consistentes.

Combinaciones como:

(`lia - alia - alia`),
(`lia - lia - alia`),
(`alia - lia - lia`),
(`lra - lia - lia`)

emergen sistemáticamente en los primeros segundos de exploración, como se puede observar en la Figura 4.6. Esto sugiere que *Hill Climbing* encuentra fácilmente "rutas ascendentes"

hacia mejores configuraciones y que las teorías relevantes están ubicadas cerca de los puntos iniciales en el espacio combinatorio.

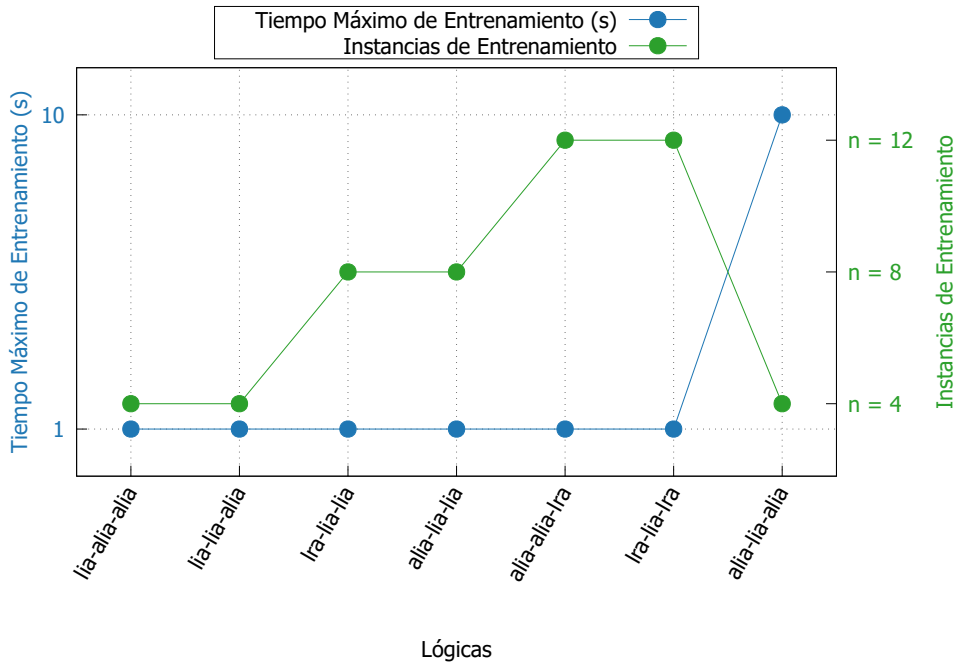


Figura 4.6: Etapas del entrenamiento en las que se encontraron por primera vez las mejores lógicas del problema las *N-Reinas*. Entre más cerca se encuentran ambos indicadores del eje X, más fácil fue dar con la combinación de lógicas.

En términos generales, las *N-reinas* se comporta como un problema altamente regular, con superficies de búsqueda suaves y pocos mínimos locales.

Unbounded Knapsack Problem (UKP)

El comportamiento observado en el *UKP* es aún más favorable para el procedimiento de búsqueda. Las mejores combinaciones heterogéneas basadas en `QF_ALIA` fueron encontradas casi exclusivamente en las etapas con *timeout* de 1 segundo, y siempre sobre la instancia más simple (`hi-n10-0-s731232778c7370`), como se puede ver en la Figura 4.7.

Incluso combinaciones más complejas, como las que incluyen componentes `LRA`, requieren solo 10 segundos o una instancia levemente más compleja para ser descubiertas.

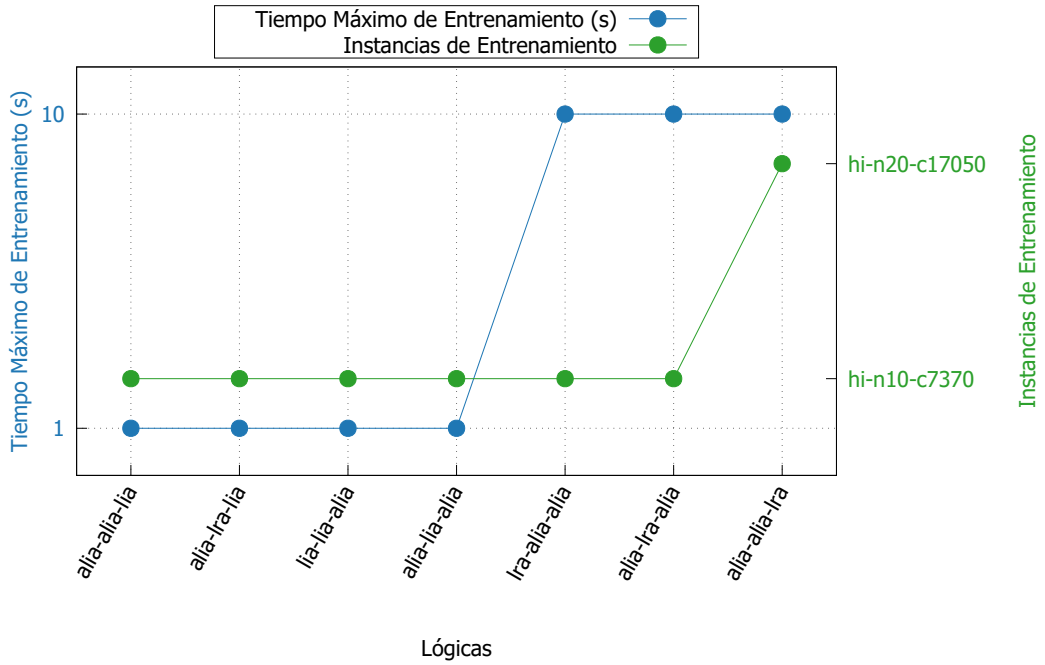


Figura 4.7: Etapas del entrenamiento en las que se encontraron por primera vez las mejores lógicas del problema *UKP*. Entre más cerca se encuentran ambos indicadores del eje X, más fácil fue dar con la combinación de lógicas.

Nurse Scheduling Problem (NSP)

El *NSP* presenta el comportamiento opuesto al problema anterior. Si bien, las mejores combinaciones se encontraron en las instancias más ligeras (`25-1-8` y `50-1-8`), ninguna de las mejores combinaciones aparece en etapas con *timeout* de 1 o 10 segundos, y todas requieren *timeouts* de 100 o 1000 segundos, como se observa en la Figura 4.8.

Las combinaciones más prometedoras, basadas en `QF_ALIA` o mezclas con `LIA`/`LRA`, emergen únicamente después de largos periodos de exploración, lo que refleja la complejidad estructural extrema del *NSP*, ya identificada en el análisis de desempeño promedio.

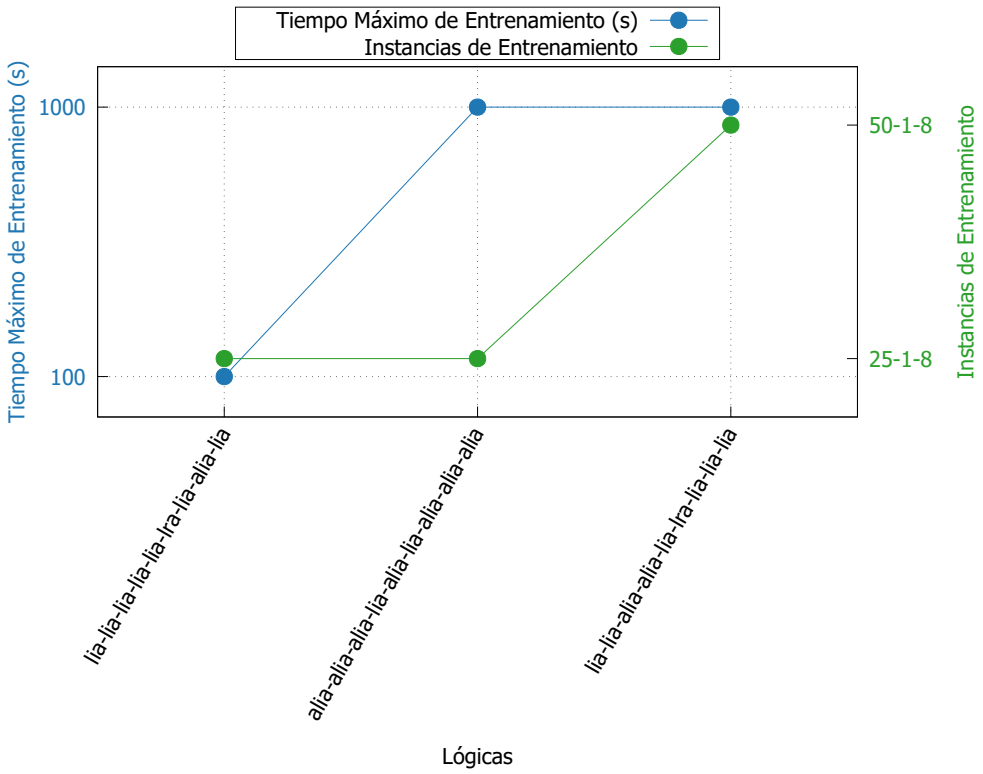


Figura 4.8: Etapas del entrenamiento en las que se encontraron por primera vez las mejores lógicas del problema *NSP*. Entre más cerca se encuentran ambos indicadores del eje X, más fácil fue dar con la combinación de lógicas.

Travelling Salesman Problem (TSP)

Para el *TSP*, el entrenamiento muestra un comportamiento más dinámico en comparación a los problemas anteriores. Las mejores combinaciones se encuentran distribuidas en distintos niveles de *timeout*:

- Varias combinaciones fueron encontradas en configuraciones con *timeout* de 1 segundo en instancias relativamente ligeras.
- Otras combinaciones requirieron 10 o 100 segundos de *timeout*, generalmente en las instancias intermedias.

- Solo algunas combinaciones específicas aparecen en etapas de 1000 segundos de *timeout*, indicando regiones más complejas del espacio de búsqueda.

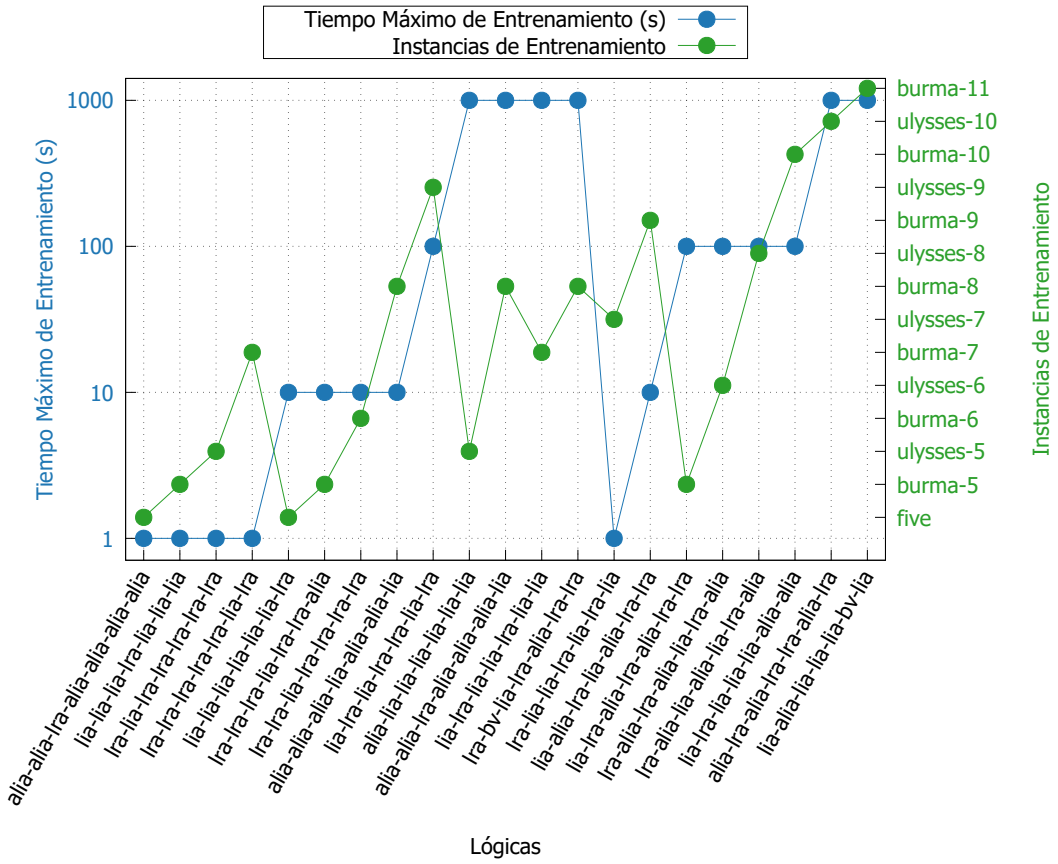


Figura 4.9: Etapas del entrenamiento en las que se encontraron por primera vez las mejores lógicas del problema *TSP*. Entre más cerca se encuentran ambos indicadores del eje X, más fácil fue dar con la combinación de lógicas.

Balanced Academic Curriculum Problem (BACP)

El entrenamiento del *BACP* ilustrado en la Figura 4.10 muestra que la combinación homogénea basada en `QF_ALIA` y la mejor combinación heterogénea son encontradas en etapas con *timeout* de 10 segundos sobre las instancias pequeñas (`bacp8` y `bacp10`).

Sin embargo, una de las combinaciones con peor desempeño aparece únicamente con un timeout de 1000 segundos y sobre la instancia más compleja (`bacp12`). Esto evidencia que,

si bien las mejores soluciones están relativamente “cerca” de los puntos de inicio, existen regiones profundas del espacio en las que la búsqueda con *Hill Climbing* converge hacia combinaciones sub-óptimas.

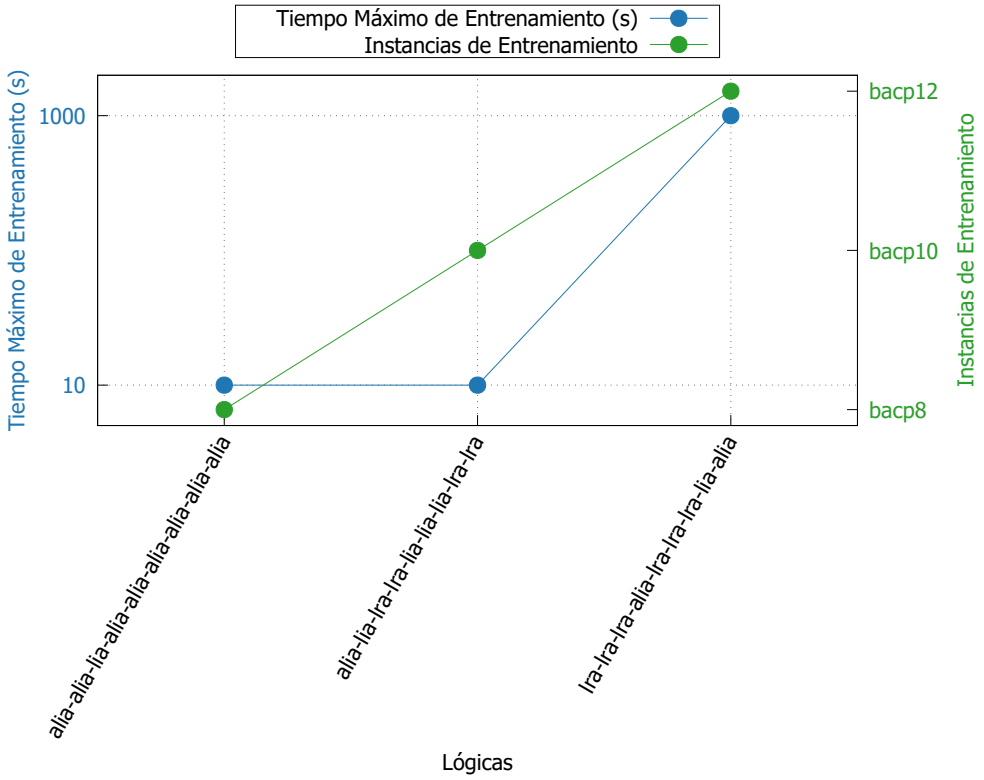


Figura 4.10: Etapas del entrenamiento en las que se encontraron por primera vez las mejores lógicas del problema *BACP*. Entre más cerca se encuentran ambos indicadores del eje X, más fácil fue dar con la combinación de lógicas.

Capítulo 5

Conclusiones y Trabajo Futuro

El presente trabajo abordó el desafío de mejorar el desempeño de solvers de *Optimization Modulo Theories* (*OMT*) mediante la selección y transformación automática de lógicas *SMT* aplicadas directamente sobre las restricciones de los modelos matemáticos construidos inicialmente utilizando la teoría de números enteros. La motivación central surgió de la observación de que distintas teorías *SMT* exhiben comportamientos diferentes dependiendo de la estructura del problema, el tipo de restricciones presentes y la forma en que estas interactúan durante el proceso de búsqueda. A partir de esta premisa, se planteó la hipótesis de que el uso de combinaciones híbridas de lógicas, junto con mecanismos automáticos de transformación y selección, podría superar a los modelos formulados haciendo uso de una sola lógica *SMT*.

Los resultados experimentales obtenidos confirman esta hipótesis de manera robusta. En prácticamente todos los problemas estudiados, *N-Reinas*, *UKP*, *TSP*, *BACP* y *NSP*, se identificaron combinaciones heterogéneas de lógicas capaces de mejorar el desempeño de múltiples configuraciones homogéneas, ya sea encontrando soluciones en un mayor número de instancias, reduciendo drásticamente los tiempos promedio de resolución, disminuyendo el uso de memoria, o logrando un equilibrio más favorable entre estas métricas. En particular, los modelos híbridos basados en la lógica `QF_ALIA`, combinada selectivamente con `LIA` y `LRA`, mostraron ganancias sustantivas en problemas con alta interacción entre restricciones discretas, secuenciales o indexadas, lo que evidencia el potencial de las arquitecturas híbridas para capturar propiedades expresivas que ninguna combinación homogénea puede

representar de manera aislada.

Los experimentos realizados permitieron comprobar que es factible modelar de manera uniforme todos los problemas utilizando un *framework* capaz de generar automáticamente variantes del modelo en distintas lógicas *SMT*. El uso de `LIA`, `LRA`, `QF_BV` y `QF_ALIA`, tanto en versiones homogéneas como heterogéneas, deja en evidencia que cada problema posee una combinación de lógicas que logra representar mejor su estructura interna y por consecuencia demostrar un mejor desempeño durante la búsqueda de la solución óptima, tal que el desempeño de esta combinación, ya sea homogénea o heterogénea, depende fuertemente del tipo de restricciones involucradas en el problema.

La transformación automática de restricciones y expresiones entre distintas lógicas *SMT* se hizo de tal manera que todos los modelos resultantes mantuvieron exactamente las mismas restricciones y componentes, preservando así el significado semántico y la factibilidad de las soluciones resultantes con respecto al modelo original. Esto se cumplió mediante el diseño e implementación de reglas de traducción para todas las combinaciones consideradas: `LIA` $\rightarrow$ `LRA`, `LIA` $\rightarrow$ `QF_BV` y `LIA` $\rightarrow$ `QF_ALIA`, junto con mecanismos adicionales para asegurar la consistencia entre las mismas variables modeladas en múltiples lógicas simultáneamente.

Sin embargo, este proceso reveló dificultades relevantes tanto en el modelamiento como en la resolución:

- La conversión a `QF_BV` requiere especificar un tamaño fijo de bit-vector, lo que introduce riesgos de *overflow* y limita la capacidad de representar dominios amplios si no se dimensionan adecuadamente los bits. Además, esta conversión en particular introduce un *overhead* adicional que las otras conversiones no consideran dado que el solver utilizado, *Z3*, cuenta con sus propios operadores internos para convertir variables de tipo entero a bit-vector y vice-versa, y hacer uso de este operador implica forzar al solver a utilizar una estrategia de búsqueda del óptimo menos eficiente, impactando directamente en los resultados observados para todos los modelos que incluyeron esta lógica dentro de sus combinaciones.

- La conversión hacia `QF_ALIA` exige representar cada variable como un arreglo indexado, lo que introduce sobrecarga estructural y puede aumentar la complejidad del modelo si no se gestiona cuidadosamente la indexación, sin embargo, al seguir utilizando la teoría de enteros para sus valores, no requiere de ninguna conversión adicional entre los tipos de datos más que la construcción de la estructura indexada.
- La transformación hacia `LRA`, si bien es sencilla ya que solo implica relajar el dominio de las variables, implica internamente expandir indebidamente el espacio de búsqueda al cambiarlo desde un dominio discreto hacia uno continuo, introduciendo una necesidad adicional de recursos (especialmente de memoria) para explorar estas soluciones.
- El uso de combinaciones heterogéneas obligó a incorporar restricciones que fuercen a que todas las variables tengan el mismo valor que sus contrapartes en las otras lógicas, garantizando que los valores asignados en una teoría sean compatibles con los de otra. Estas restricciones son esenciales, pero añaden complejidad estructural al modelo final.

A pesar de estas dificultades, las transformaciones implementadas fueron funcionales y permitieron construir modelos correctos y comparables entre sí. El análisis evidencia que la automatización del *encoding* es viable, pero requiere un diseño cuidadosamente calibrado, especialmente en dominios con restricciones no lineales o con alta carga combinatoria.

Asimismo, el procedimiento de búsqueda basado en *Hill Climbing* demostró ser una estrategia de entrada eficaz para explorar el espacio de combinaciones lógicas. Problemas como el *NSP* y el *TSP* exhiben una complejidad más alta en comparación a los otros problemas y requieren mayores tiempos para encontrar la solución óptima, y por consecuencia, mayores tiempos durante el entrenamiento para encontrar combinaciones de lógicas efectivas, pero aún así el algoritmo de búsqueda fue capaz en todos los casos de identificar combinaciones significativamente mejores que varios de los modelos base.

Hill Climbing fue capaz de encontrar combinaciones de lógicas superiores a múltiples de las configuraciones homogéneas, especialmente a la lógica empleada para construir el modelo original (`LIA`), en problemas como *NSP*, *TSP*, *UKP* y *BACP*, confirmando la utilidad práctica del método. Si bien, para las *N-Reinas* no logró encontrar ninguna combinación de lógicas

heterogénea que lograra superar al modelo original, varias de las combinaciones mixtas encontradas en cada problema lograron mejorar entre 1 y 2 ordenes de magnitud el tiempo empleado en la búsqueda con respecto a distintas combinaciones homogéneas.

En consecuencia, el rendimiento del método depende fuertemente del problema: la heurística es eficiente y efectiva, pero su desempeño puede mejorarse mediante vecindarios adaptativos, estrategias multiarranque o métodos híbridos que combinen exploración y explotación de manera más equilibrada, o derechamente con una metaheurística más compleja que pueda explorar múltiples vecindarios simultáneamente.

En cuanto al desempeño de las combinaciones heterogéneas, se presentan los siguientes hallazgos principales:

1. En todos los problemas evaluados existen combinaciones heterogéneas superiores a al menos dos de las configuraciones homogéneas, y en cuatro de los cinco problemas estas combinaciones fueron superiores a la lógica `LIA`, usada como base para construir el modelo original.

   Este es uno de los resultados más relevantes del estudio: la mezcla de teorías no solo es viable, sino sistemáticamente ventajosa.

2. En problemas estructuralmente complejos (*BACP* y *TSP*), las mejoras son más evidentes:

   - En *BACP*, dos combinaciones heterogéneas duplican la velocidad del mejor modelo homogéneo.
   - En *TSP*, al menos cinco combinaciones heterogéneas superan al mejor modelo homogéneo, tanto en tasa de éxito como en tiempo.

3. Las combinaciones heterogéneas pueden lograr configuraciones que requieren un menor uso de memoria.

En conjunto, estos resultados permiten concluir que la combinación automática de lógicas constituye una alternativa viable y efectiva para mejorar la capacidad de resolución de problemas de optimización y satisfacción de restricciones mediante *OMT*, especialmente en

escenarios donde el modelamiento tradicional en una lógica inicial transformado hacia otras lógicas homogéneas muestra limitaciones evidentes. El enfoque propuesto no solo aumenta la expresividad del modelamiento de restricciones complejas mediante distintas lógicas, sino que además permite que la selección de lógicas se adapte automáticamente a la estructura particular de cada problema. Esto se logra mediante la búsqueda guiada, que identifica qué combinaciones de lógicas resultan más eficientes para resolver cada problema en general.

5.1. Limitaciones del Trabajo Realizado

Tamaño y diversidad de las instancias

El conjunto de instancias empleado para evaluar los modelos, si bien representativo y adecuado para un estudio comparativo, presenta las siguientes limitantes:

- **Complejidad de las instancias:**

  En problemas como *NSP*, *TSP* y *BACP*, instancias de mayor escala (mayor número de días de planificación, ciudades o cursos) podrían exhibir comportamientos drásticamente más complejos que los observados en las instancias que lograron ser resueltas con las combinaciones encontradas para las configuraciones empleadas en el solver.

- **Diversidad estructural:**

  Algunos problemas, como *UKP* o las *N-Reinas*, poseen una estructura relativamente regular en cuanto al tipo de restricciones empleadas, lo que permite que las conclusiones sean estables entre instancias, pero no necesariamente seguirán siéndolo para instancias con restricciones adicionales no contempladas en los modelos base.

Conjunto limitado de lógicas consideradas

El trabajo se centra exclusivamente en cuatro teorías fundamentales: `LIA`, `LRA`, `QF_BV` y `QF_ALIA`.

Si bien estas lógicas representan un espectro amplio de expresividad, desde la aritmética lineal continua hasta razonamiento indexado y operaciones *bitwise*, existe un conjunto mucho mayor de teorías disponibles en la familia de teorías *SMT* (no lineales, cuantificadas, de strings, etc.) que no fueron consideradas.

Adicionalmente:

- No se incorporaron teorías no lineales, como `NIA` o `NRA`.
- No se exploraron combinaciones de mayor profundidad que las generadas por la estructura de traducción adoptada (los modelos construidos se compusieron por 3, 7 y 9 componentes lógicas)

Por lo tanto, aunque las combinaciones estudiadas son relevantes y suficientes para validar la hipótesis planteada, el universo de combinaciones posibles es mucho más amplio.

Uso de una única metaheurística: *Hill Climbing*

El procedimiento de búsqueda se apoyó exclusivamente en *Hill Climbing* con el criterio de "alguna mejora" usando una vecindad acotada limitada a dos configuraciones de búsqueda posibles, "abanico" y "aleatoria", lo que introduce varias limitaciones metodológicas:

- **Sensibilidad a óptimos locales:**

  Se evidencia una rápida convergencia por parte del algoritmo de búsqueda empleado dado el criterio de término de *Hill Clibing*: finalizar cuando ninguna solución del vecindario produce alguna mejora, y el acotado conjunto de vecinos: misma solución que la mejor, pero con una lógica distinta, provocando que quede una gran porción del espacio de búsqueda sin explorar, especialmente en problemas con un mayor número de restricciones como *NSP*, *BACP* y *TSP*,

- **Ausencia de mecanismos de diversificación:**

  Para el *Hill Climbing* se consideraron 4 puntos de inicio (uno por cada combinación de lógicas homogéneas), pero además de eso no se emplearon técnicas adicionales

como reinicios aleatorios, búsqueda tabú, *Simulated Annealing* (recocido simulado) o algoritmos genéticos, que podrían explorar regiones del espacio inaccesibles para la heurística empleada por si sola.

- **Dependencia fuerte del orden de exploración:**

  Dado que los vecinos se definen transformando una sola de las lógicas usadas en la mejor solución, la calidad de las combinaciones generadas se ve directamente influenciada por la calidad de la mejor solución encontrada.

El rendimiento del enfoque podría mejorar significativamente si se adoptaran estrategias de búsqueda más robustas o híbridas, o bien, estrategias poblacionales capaces de explorar múltiples zonas del espacio de búsqueda simultáneamente, permitiendo también generar combinaciones más interesantes que las obtenidas al transformar solo una de las componentes.

Restricciones centradas en modelos lineales

El proceso de traducción y transformación de lógicas estuvo diseñado, de manera explícita para manejar restricciones lineales, expresiones aritméticas afines, accesos indexados a estructuras discretas, operaciones de dominio entero y real dentro de marcos lineales.

En consecuencia:

- No se abordaron restricciones no lineales, cuadráticas o polinómicas.
- No se evaluaron problemas con componentes simbólicas, cuantificadores o expresiones no estructuradas.
- Las reglas de transformación se construyen bajo el supuesto de linealidad y monotonicidad en la forma de las expresiones.

Esto implica que la metodología desarrollada es sólida para problemas lineales y discretos, pero no necesariamente aplicable a dominios con dinámicas más complejas como optimización no convexa, verificación simbólica o razonamiento sobre strings.

Limitaciones computacionales y configuraciones fijas

No se estudia el impacto de otros solvers *OMT* (como *OptiMathSAT*, *cvc5*, etc.) para validar el comportamiento observado y evaluar las combinaciones encontradas para un mismo problema.

La reproducibilidad del rendimiento mostrado en los resultados puede verse afectada por diferencias en *hardware* al momento de ejecutar el solver.

5.2. Trabajo Futuro

Los resultados obtenidos en este trabajo abren una serie de líneas de investigación prometedoras, tanto en el ámbito del modelamiento híbrido con lógicas *SMT* como en el diseño de estrategias de búsqueda para seleccionar automáticamente configuraciones eficientes. Las siguientes direcciones representan extensiones naturales del marco propuesto y apuntan a fortalecer su aplicabilidad, robustez y generalización.

Nuevas heurísticas de búsqueda

El presente estudio se basó exclusivamente en una variante monótona de *Hill Climbing*, lo cual permitió validar la hipótesis central, pero también reveló limitaciones para problemas con espacios de búsqueda más grande, como *NSP*, *BACP* y *TSP*.

Futuras investigaciones podrían explorar metaheurísticas más sofisticadas, tales como:

- ***Simulated Annealing* (Recocido Simulado)**, para escapar de óptimos locales mediante transiciones controladas hacia soluciones peores.
- **Búsqueda Tabú**, introduciendo memoria para evitar ciclos y ampliar la exploración del espacio.
- **Algoritmos Genéticos**, combinando subconjuntos de lógicas exitosas mediante operadores inspirados en la recombinación y mutación.

- **Aprendizaje Reforzado**, permitiendo que el algoritmo aprenda políticas de selección de lógicas basadas en el desempeño histórico o por medio de simulaciones.

Estas heurísticas podrían mejorar la calidad de las combinaciones encontradas, reducir el número de iteraciones y permitir explorar regiones del espacio inaccesibles para un método estrictamente ascendente.

Extender el marco de transformación a otras teorías *SMT*

La metodología actual se centra en transformar restricciones lineales entre cuatro teorías principales (`LIA`, `LRA`, `QF_BV`, `QF_ALIA`). Sin embargo, el ecosistema de *SMT* ofrece un conjunto mucho más amplio de teorías cuya integración abriría nuevas posibilidades de modelamiento.

Líneas relevantes incluyen:

- Aritmética no lineal (`NIA`, `NRA`), para problemas con multiplicaciones, cuadráticas o restricciones no convexas.
- Teorías de igualdad con funciones no interpretadas (*UF*), aplicables en modelamiento simbólico y verificación.
- *Strings* y expresiones regulares, para problemas de análisis de texto, síntesis o validación de patrones.
- Teorías de conjuntos (*Sets*) o mapas (*Maps*), útiles en problemas de asignación y distribución.
- Modelamiento híbrido con cuantificadores, permitiendo expresar restricciones globales o paramétricas.

Extender el marco de transformación a estas teorías permitiría abordar una gama más amplia de problemas y evaluar si las combinaciones inter-lógicas conservan o amplifican las mejoras observadas en las teorías lineales.

Evaluación sobre instancias de mayor escala y *benchmarks* industriales

Los resultados obtenidos evidencian patrones claros, pero están limitados por el tamaño de las instancias y por la complejidad moderada de los problemas evaluados. Una extensión natural consiste en aplicar el enfoque a:

- Instancias de escala industrial en planificación, logística o verificación.
- *Benchmarks* completos de la *SMT-LIB*, que contienen miles de problemas con variaciones teóricas y estructurales.
- Problemas con cientos o miles de restricciones, para estudiar el comportamiento del solver en escenarios de alta carga combinatoria.

Este análisis permitiría determinar la robustez del método frente al crecimiento de la dimensionalidad, así como caracterizar qué teorías se degradan o escalan mejor cuando las instancias aumentan significativamente en tamaño.

Integración directa con *solvers SMT* modernos

Actualmente, la generación y evaluación de modelos se realiza mediante archivos *SMT2* estáticos con dependencias en la librería de *Z3* para conversiones de lógicas. Un enfoque más avanzado consistiría en integrar el sistema de búsqueda directamente con este u otro solver, con el fin de poder:

- Modificar dinámicamente la lógica de cada restricción durante el proceso de resolución, en respuesta al desempeño observado en lugar de resolver la instancia por completo con cada combinación del vecindario.
- Aprovechar *APIs* internas de optimización, como *Z3-tactics*, *OptiMathSAT strategies*, o *cvc5 portfolios*, lo que permitiría producir un mayor impacto a bajo nivel haciendo uso completo de todas las capacidades de cada solver.

Otra trabajo interesante en esta línea sería la construcción de un solver híbrido autoajustable que fuera capaz de detectar cuándo una teoría se vuelve ineficiente, reescribir partes del modelo en tiempo real y continuar la búsqueda con un modelo adaptado.

Este tipo de integración abriría la puerta a sistemas de razonamiento *logic-aware*, donde la selección de teorías no es fija, sino parte del proceso iterativo del solver.

Bibliografía

[1] J. Hooker, "Logic, optimization, and constraint programming," *INFORMS J. on Computing*, vol. 14, p. 295–321, Nov. 2002.

[2] A. Neumaier, "Complete search in continuous global optimization and constraint satisfaction," *Acta Numerica*, vol. 13, p. 271–369, 2004.

[3] A. Falkner, G. Friedrich, A. Haselböck, G. Schenner, and H. Schreiner, "Twenty-five years of successful application of constraint technologies at siemens," *AI Mag.*, vol. 37, p. 67–80, Dec. 2016.

[4] P. Kolhe and H. Christensen, "Planning in logistics: A survey," pp. 48–53, 09 2010.

[5] S. D'Amours, M. Rönnqvist, and A. Weintraub Pohorille, "Using Operational Research for Supply Chain Planning in the Forest Products Industry," Nov. 2008.

[6] M. Salido and F. Rossi, "New trends in constraint satisfaction, planning, and scheduling: A survey," *Knowledge Eng. Review*, vol. 25, pp. 249–279, 09 2010.

[7] S. Yu, L. You, and S. Zhou, "A review of optimization modeling and solution methods in renewable energy systems," *Frontiers of Engineering Management*, vol. 10, pp. 640–671, Dec 2023.

[8] A. Esparza Aponte, M. Blondin, and J. P. Trovao, "A review of optimization strategies for energy management in microgrids," *Energies*, vol. 18, p. 3245, 06 2025.

[9] S. Dev Gupta, B. Genc, and B. O'Sullivan, "Explanation in constraint satisfaction: A survey," in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21* (Z.-H. Zhou, ed.), pp. 4400–4407, International Joint Conferences on Artificial Intelligence Organization, 8 2021. Survey Track.

[10] J. R. Banga, "Optimization in computational systems biology," *BMC Systems Biology*, vol. 2, p. 47, May 2008.

[11] G. An, B. G. Fitzpatrick, S. Christley, P. Federico, A. Kanarek, R. M. Neilan, M. Oremland, R. Salinas, R. Laubenbacher, and S. Lenhart, "Optimization and control

of agent-based models in biology: A perspective," *Bulletin of Mathematical Biology*, vol. 79, pp. 63–87, Jan 2017.

[12] T. Yamakami, "Optimization, randomized approximability, and boolean constraint satisfaction problems," 2011.

[13] A. B. Trindade and L. C. Cordeiro, "Applying smt-based verification to hardware/software partitioning in embedded systems," *Design Automation for Embedded Systems*, vol. 20, pp. 1–19, Mar 2016.

[14] Y.-F. Liu, T.-H. Chang, M. Hong, Z. Wu, A. M.-C. So, E. A. Jorswieck, and W. Yu, "A survey of recent advances in optimization methods for wireless communications," 2024.

[15] K. Çelikbilek, Z. Saleem, R. Morales Ferre, J. Praks, and E. S. Lohan, "Survey on optimization methods for leo-satellite-based networks with applications in future autonomous transportation," *Sensors*, vol. 22, no. 4, 2022.

[16] H. Fei, L. Qian, and D. Sun, "A survey of recent research on optimization models and algorithms for operations management from the process view," *Scientific Programming*, vol. 2017, pp. 1–19, 07 2017.

[17] M. P. Castro, A. A. Cire, and J. C. Beck, "Decision diagrams for discrete optimization: A survey of recent advances," 2022.

[18] B. Aslani, S. Mohebbi, and E. Oughton, "A systematic review of optimization methods for recovery planning in cyber–physical infrastructure networks: Current state and future trends," *Computers amp; Industrial Engineering*, vol. 192, p. 110224, June 2024.

[19] F. Enayaty Ahangar, L. Albert, and E. DuBois, "A survey of optimization models and methods for cyberinfrastructure security," *IISE Transactions*, vol. 53, pp. 1–48, 06 2020.

[20] W. L. Winston, *Operations Research: Applications and Algorithms*. Belmont, California: Thomson Brooks/Cole, 4th ed., 2004.

[21] C. Tinelli, "A dpll-based calculus for ground satisfiability modulo theories," in *Logics in Artificial Intelligence* (S. Flesca, S. Greco, G. Ianni, and N. Leone, eds.), (Berlin, Heidelberg), pp. 308–319, Springer Berlin Heidelberg, 2002.

[22] S. A. Cook, "The complexity of theorem-proving procedures," *Proceedings of the third annual ACM symposium on Theory of computing*, vol. STOC '71, p. 151–158, 1971. Archived (PDF) from the original on 2022-10-09.

[23] R. Oliver, "Optimization modulo theories," Master's thesis, Universitat Politècnica de Catalunya, 01 2011.

[24] I. Jorquera, N. Gálvez Ramírez, F. Saubion, E. Monfroy, and C. Castro, "Addressing classic constraint satisfaction problems through optimisation modulo theories," in *Optimization and Learning* (B. Dorronsoro, E.-G. Talbi, D. Weraikat, and S. Bouamama, eds.), (Cham), pp. 267–278, Springer Nature Switzerland, 2026.

[25] R. M. Karp, *Reducibility among Combinatorial Problems*, pp. 85–103. Boston, MA: Springer US, 1972.

[26] J. P. Vielma, "Mixed integer linear programming formulation techniques," *SIAM Review*, vol. 57, no. 1, pp. 3–57, 2015.

[27] N. Ejaz and S. Choudhury, "A comprehensive survey of linear, integer, and mixed-integer programming approaches for optimizing resource allocation in 5g and beyond networks," 2025.

[28] V. Mak-Hau, J. Yearwood, and W. Moran, "Knowledge engineering mixed-integer linear programming: constraint typology," 2021.

[29] B. S. Hussain, "CSPLib problem 054: N-queens." `http://www.csplib.org/Problems/prob054`.

[30] T. Kunt, "The *n*-queens problem in higher dimensions," 2024.

[31] M. Al-Rudaini, "N-queens problem solving using linear programming in gnu linear programming kit (glpk)," 11 2016.

[32] "GLPK – GNU Linear Programming Kit." urlhttp://www.gnu.org/software/glpk/, release year or current year. Accessed: [Date of access, e.g., 24 Nov. 2025].

[33] A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," *Econometrica*, vol. 28, no. 3, pp. 497–520, 1960.

[34] G. Pataki and M. Tural, "Basis reduction, and the complexity of branch-and-bound," 2009.

[35] S. Shoja, D. Arnström, and D. Axehill, "Overall complexity certification of a standard branch and bound method for mixed-integer quadratic programming," 2022.

[36] A. Khajavirad and N. V. Sahinidis, "A hybrid lp/nlp paradigm for global optimization relaxations," *Mathematical Programming Computation*, vol. 10, pp. 383–421, Sep 2018.

[37] G. Cabrera G., E. Cabrera, R. Soto, L. J. M. Rubio, B. Crawford, and F. Paredes, "A hybrid approach using an artificial bee algorithm with mixed integer programming applied to a large-scale capacitated facility location problem," *Mathematical Problems in Engineering*, vol. 2012, no. 1, p. 954249, 2012.

[38] P. R. Pinheiro and P. R. Oliveira, "A hybrid approach of bundle and benders applied large mixed linear integer problem," *Journal of Applied Mathematics*, vol. 2013, no. 1, p. 678783, 2013.

[39] N. Eén and N. Sörensson, "An extensible sat-solver," in *Theory and Applications of Satisfiability Testing* (E. Giunchiglia and A. Tacchella, eds.), (Berlin, Heidelberg), pp. 502–518, Springer Berlin Heidelberg, 2004.

[40] G. Audemard and L. Simon, "On the glucose sat solver," *International Journal on Artificial Intelligence Tools*, vol. 27, no. 01, p. 1840001, 2018.

[41] A. Biere, "Lingeling, plingeling and treengeling entering the sat competition 2013," in *In Proceedings of SAT Competition 2013* ({A. Balint, A. Belov, M. Heule, M. Järvisalo}, ed.), vol. B-2013-1, Department of Computer Science Series of Publications, 2013.

[42] A. Gupta, M. K. Ganai, and C. Wang, "Sat-based verification methods and applications in hardware verification," in *Formal Methods for Hardware Verification* (M. Bernardo and A. Cimatti, eds.), (Berlin, Heidelberg), pp. 108–143, Springer Berlin Heidelberg, 2006.

[43] M. Abdulaziz and F. Kurz, "Formally verified sat-based ai planning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, pp. 14665–14673, Jun. 2023.

[44] S. Eggergluss, G. Fey, and R. Drechsler, "Sat-based atpg for path delay faults in sequential circuits," in *2007 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 3671–3674, 2007.

[45] A. Biere and D. Kröning, *SAT-Based Model Checking*, pp. 277–303. Cham: Springer International Publishing, 2018.

[46] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, p. 394–397, July 1962.

[47] J. Marques-Silva and K. Sakallah, "Grasp: a search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 506–521, 1999.

[48] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient sat solver," in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pp. 530–535, 2001.

[49] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI'09, (San Francisco, CA, USA), p. 399–404, Morgan Kaufmann Publishers Inc., 2009.

[50] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. NLD: IOS Press, 2009.

[51] V. Nicolet, "Csc410 tutorial: z3 - sat solving." http://www.cs.toronto.edu/ victorn/tutorials/z3/SAT.html, 2017. Last accessed on 15-10-2023.

[52] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt, "Cadical 2.0," in *Computer Aided Verification* (A. Gurfinkel and V. Ganesh, eds.), (Cham), pp. 133–152, Springer Nature Switzerland, 2024.

[53] N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, "Sat competition 2020," *Artificial Intelligence*, vol. 301, p. 103572, 2021.

[54] R. Sebastiani and P. Trentin, "Optimathsat: A tool for optimization modulo theories," pp. 447–454, 07 2015.

[55] R. Nieuwenhuis and A. Oliveras, "On sat modulo theories and optimization problems," in *Theory and Applications of Satisfiability Testing - SAT 2006* (A. Biere and C. P. Gomes, eds.), (Berlin, Heidelberg), pp. 156–169, Springer Berlin Heidelberg, 2006.

[56] L. de Moura and N. Bjørner, "Z3: an efficient smt solver," vol. 4963, pp. 337–340, 04 2008.

[57] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, (Berlin, Heidelberg), p. 171–177, Springer-Verlag, 2011.

[58] J. N. Hooker, "Toward unification of exact and heuristic optimization methods," *International Transactions in Operational Research*, vol. 22, no. 1, pp. 19–48, 2015.

[59] R. Alkhalifa, F. Alkhomayes, B. Almazroua, D. Alhaidan, M. Alothman, and J. Almuhaidib, "A comparative review of parallel exact, heuristic, metaheuristic, and hybrid optimization techniques for the traveling salesman problem," 2025.

[60] G. B. Dantzig, *Origins of the simplex method*, p. 141–151. New York, NY, USA: Association for Computing Machinery, 1990.

[61] A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," *Econometrica*, vol. 28, pp. 497–520, 2025/11/04/ 1960. Full publication date: Jul., 1960.

[62] R. Gomory, "Outline of an algorithm for integer solutions to linear programs," *Bulletin of the American Mathematical Society*, vol. 64, pp. 275–278, 09 1958.

[63] J. Karlof, *Integer programming: Theory and practice*. 01 2005.

[64] J. Teghem *European Journal of Operational Research*, vol. 205, no. 2, pp. 486–487, 2010.

[65] S. Khalfi, F. Caraffini, and G. Iacca, "Metaheuristics in the balance: A survey on memory-saving approaches for platforms with seriously limited resources," *International Journal of Intelligent Systems*, vol. 2023, no. 1, p. 5708085, 2023.

[66] P. Mills, E. Tsang, Q. Zhang, and J. Ford, "A survey of ai-based meta-heuristics for dealing with local optima in local search," 10 2004.

[67] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[68] K. Rajwar, K. Deep, and S. Das, "An exhaustive review of the metaheuristic algorithms for search and optimization: taxonomy, applications, and open challenges," *Artificial Intelligence Review*, vol. 56, pp. 13187–13257, Nov 2023.

[69] D. B. Fogel, *Artificial Intelligence through Simulated Evolution*, pp. 227–296. 1998.

[70] I. Rechenberg and B. Toms, *Cybernetic Solution Path of an Experimental Problem:*. Library translation / Royal Aircraft Establishment, Ministry of Aviation, 1965.

[71] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press, 04 1992.

[72] J. R. Koza, "Genetic programming as a means for programming computers by natural selection," *Statistics and Computing*, vol. 4, pp. 87–112, Jun 1994.

[73] G. B. Dantzig, *Linear Programming and Extensions*. United States Air Force Project RAND, The RAND Corporation, Aug. 1963.

[74] S. I. Gass, *Linear programming: methods and applications (5th ed.)*. USA: McGraw-Hill, Inc., 1985.

[75] R. Bland, "New finite pivoting rules for the simplex method," *Math. Oper. Res.*, vol. 2, p. 103–107, May 1977.

[76] R. Deval, "Exact algorithms in combinatorial optimization.," *International Review of Applied Engineering Research 2248-9967*, vol. 3, pp. 24–28, 07 2013.

[77] I. Dumitrescu and T. Stützle, "A survey of methods that combine local search and exact algorithms,"

[78] G. B. Dantzig, *The Simplex Method.* Santa Monica, CA: RAND Corporation, 1956.

[79] A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," *Econometrica*, vol. 28, p. 497, 1960.

[80] M. Padberg and G. Rinaldi, "A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems," *SIAM Review*, vol. 33, no. 1, pp. 60–100, 1991.

[81] S. Nickel, C. Steinhardt, H. Schlenker, W. Burkart, and M. Reuter-Oppermann, *IBM ILOG CPLEX Optimization Studio*, pp. 9–23. 05 2020.

[82] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2024.

[83] S. Bolusani, M. Besançon, K. Bestuzheva, A. Chmiela, J. Dionísio, T. Donkiewicz, J. van Doornmalen, L. Eifler, M. Ghannam, A. Gleixner, C. Graczyk, K. Halbig, I. Hedtke, A. Hoen, C. Hojny, R. van der Hulst, D. Kamp, T. Koch, K. Kofler, J. Lentz, J. Manns, G. Mexi, E. Mühmer, M. E. Pfetsch, F. Schlösser, F. Serrano, Y. Shinano, M. Turner, S. Vigerske, D. Weninger, and L. Xu, "The scip optimization suite 9.0," 2024.

[84] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition ed., 1979.

[85] C. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, vol. 32. 01 1982.

[86] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Comput. Surv.*, vol. 35, p. 268–308, Sept. 2003.

[87] E.-G. Talbi, *Common Concepts for Metaheuristics*, ch. 1, pp. 1–86. John Wiley and Sons, Ltd, 2009.

[88] F. Glover, G. A. Kochenberger, and F. S. Hillier, eds., *Handbook of Metaheuristics*, vol. 57 of *International Series in Operations Research and Management Science*. Boston, MA: Springer US, 2003.

[89] I. H. Osman and G. Laporte, "Metaheuristics: A bibliography," *Annals of Operations Research*, vol. 63, pp. 511–623, Oct 1996.

[90] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. USA: Prentice Hall Press, 3rd ed., 2009.

[91] Z. Michalewicz and D. B. Fogel, *How to Solve It: Modern Heuristics*. Berlin, Heidelberg: Springer-Verlag, 2004.

[92] V. Cerny, "Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm," *Journal of Optimization Theory and Applications*, vol. 45, pp. 41–51, Jan 1985.

[93] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[94] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Comput. Oper. Res.*, vol. 13, p. 533–549, May 1986.

[95] F. Glover, "Tabu search—part i," *ORSA Journal on Computing*, vol. 1, no. 3, pp. 190–206, 1989.

[96] F. Glover and M. Laguna, *Tabu Search*, pp. 2093–2229. Boston, MA: Springer US, 1998.

[97] H. R. Lourenco, O. C. Martin, and T. Stutzle, "Iterated local search," 2001.

[98] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. Springer Publishing Company, Incorporated, 2nd ed., 2015.

[99] T. Back, *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. USA: Oxford University Press, Inc., 1996.

[100] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1989.

[101] P. Moscato, "On evolution, search, optimization, genetic algorithms and martial arts - towards memetic algorithms," *Caltech Concurrent Computation Program*, 10 2000.

[102] M. Srinivas and L. Patnaik, "Adaptive probabilities of crossover and mutation in genetic algorithms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 24, no. 4, pp. 656–667, 1994.

[103] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *Trans. Evol. Comp*, vol. 6, p. 182–197, Apr. 2002.

[104] D. B. Fogel, *Evolutionary Computation: Towards a New Philosophy of Machine Intelligence*. Wiley-IEEE Press, 2nd ed., 1999.

[105] H.-P. P. Schwefel, *Evolution and Optimum Seeking: The Sixth Generation*. USA: John Wiley & Sons, Inc., 1993.

[106] H.-G. Beyer and H.-P. Schwefel, "Evolution strategies –a comprehensive introduction," *Natural Computing: An International Journal*, vol. 1, p. 3–52, May 2002.

[107] N. Hansen and A. Ostermeier, "Completely derandomized self-adaptation in evolution strategies," *Evol. Comput.*, vol. 9, p. 159–195, June 2001.

[108] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. Cambridge, MA, USA: MIT Press, 1992.

[109] R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008.

[110] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," 2017. Available at `www.SMT-LIB.org`.

[111] SMT-LIB Initiative, "Smt-lib: The logic specifications." `https://smt-lib.org/logics.shtml`. Accessed: 2025-02-05.

[112] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, vol. 54, p. 69–77, sep 2011.

[113] N. Gálvez Ramírez, *A Framework for Autonomous Generation of Strategies in Satisfiability Modulo Theories*. PhD thesis, 12 2018.

[114] G. Nelson and D. C. Oppen, "Simplification by cooperating decision procedures," *ACM Trans. Program. Lang. Syst.*, vol. 1, p. 245–257, Oct. 1979.

[115] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems* (D. Fisman and G. Rosu, eds.), (Cham), pp. 415–442, Springer International Publishing, 2022.

[116] B. Dutertre and L. M. de Moura, "The yices smt solver," 2006.

[117] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using smt solvers instead of sat solvers," in *Model Checking Software* (A. Valmari, ed.), (Berlin, Heidelberg), pp. 146–162, Springer Berlin Heidelberg, 2006.

[118] M. Bofill, J. Suy, M. Palahí, and M. Villaret, "Solving constraint satisfaction problems with sat modulo theories," *Constraints*, vol. 17, pp. 273–303, 2012.

[119] M. Bofill, J. Suy, and M. Villaret, "A system for solving constraint satisfaction problems with smt," in *Theory and Applications of Satisfiability Testing – SAT 2010* (O. Strichman and S. Szeider, eds.), (Berlin, Heidelberg), pp. 300–305, Springer Berlin Heidelberg, 2010.

[120] M. Palahí i Sitges, *Reformulation of constraint models into SMT*. PhD thesis, Spain, 2015.

[121] H. Hosobe, "Solving hierarchical soft constraints with an smt solver," in *Proceedings of the 2020 12th International Conference on Computer and Automation Engineering*, ICCAE 2020, (New York, NY, USA), p. 42–46, Association for Computing Machinery, 2020.

[122] N. Bjørner, "Smt in verification, modeling, and testing at microsoft," in *Hardware and Software: Verification and Testing* (A. Biere, A. Nahir, and T. Vos, eds.), (Berlin, Heidelberg), pp. 3–3, Springer Berlin Heidelberg, 2013.

[123] L. Cordeiro, B. Fischer, and J. Marques-Silva, "Smt-based bounded model checking for embedded ansi-c software," in *2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 137–148, 2009.

[124] H. Kim, *Efficient Smt Solving for Hardware Model Checking*. PhD thesis, USA, 2010. AAI3433303.

[125] C. Ansótegui, M. Bofill, F. Manyà, and M. Villaret, "Automated theorem provers for multiple-valued logics with satisfiability modulo theory solvers," *Fuzzy Sets and Systems*, vol. 292, pp. 32–48, 2016. Special Issue in Honor of Francesc Esteva on the Occasion of his 70th Birthday.

[126] C. Ansótegui, M. Bofill, F. Manyà, and M. Villaret, "Building automated theorem provers for infinitely-valued logics with satisfiability modulo theory solvers," in *2012 IEEE 42nd International Symposium on Multiple-Valued Logic*, pp. 25–30, 2012.

[127] S. Böhme, *Proving Theorems of Higher-Order Logic with SMT Solvers*. PhD thesis, Technische Universität München, 2012.

[128] K. R. M. Leino, "Automating theorem proving with smt," in *Interactive Theorem Proving* (S. Blazy, C. Paulin-Mohring, and D. Pichardie, eds.), (Berlin, Heidelberg), pp. 2–16, Springer Berlin Heidelberg, 2013.

[129] B. Dutertre and L. de Moura, "A fast linear-arithmetic solver for dpll(t)," in *Computer Aided Verification* (T. Ball and R. B. Jones, eds.), (Berlin, Heidelberg), pp. 81–94, Springer Berlin Heidelberg, 2006.

[130] C. Barrett, L. de Moura, and A. Stump, "Smt-comp: Satisfiability modulo theories competition," in *Computer Aided Verification* (K. Etessami and S. K. Rajamani, eds.), (Berlin, Heidelberg), pp. 20–23, Springer Berlin Heidelberg, 2005.

[131] N. S. Bjørner and A.-D. Phan, "*νz* - maximal satisfaction with z3," in *International Symposium on Symbolic Computation in Software Science*, 2014.

[132] M. Eraşcu, F. Micota, and D. Zaharie, "Scalable optimal deployment in the cloud of component-based applications using optimization modulo theory, mathematical programming and symmetry breaking," *Journal of Logical and Algebraic Methods in Programming*, vol. 121, p. 100664, 2021.

[133] E. Davidson, Ö. Akgün, J. Espasa, and P. Nightingale, "Effective encodings of constraint programming models to smt," in *Principles and Practice of Constraint Programming* (H. Simonis, ed.), (Cham), pp. 143–159, Springer International Publishing, 2020.

[134] N. Gálvez Ramírez, Y. Hamadi, E. Monfroy, and F. Saubion, "Evolving smt strategies," in *2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 247–254, 11 2016.

[135] N. G. Ramírez, E. Monfroy, F. Saubion, and C. Castro, "Optimizing smt solving strategies by learning with an evolutionary process," in *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 816–820, 2018.

[136] N. Gálvez Ramírez, Y. Hamadi, E. Monfroy, and F. Saubion, "Towards automated strategies in satisfiability modulo theory," in *European Conference on Genetic Programming*, pp. 230–245, 03 2016.

[137] N. Gálvez Ramírez, E. Monfroy, F. Saubion, and C. Castro, "Improving complex smt strategies with learning," *International Transactions in Operational Research*, vol. 27, no. 2, pp. 1162–1188, 2020.

[138] Q. Z. Benjamin Mikek, "Speeding up smt solving via compiler optimization," in *In Proceedings of the Symposium on Foundations of Software Engineering (FSE).*, 2023.

[139] J. Schmidt and M. Leuschel, "Improving smt solver integrations for the validation of b and event-b models," in *Formal Methods for Industrial Critical Systems* (A. Lluch Lafuente and A. Mavridou, eds.), (Cham), pp. 107–125, Springer International Publishing, 2021.

[140] Lu, Zhengyang, "Alphasmt: A reinforcement learning guided smt solver," Master's thesis, 2023.

[141] M. Balunovic, P. Bielik, and M. Vechev, "Learning to solve smt formulas," in *Advances in Neural Information Processing Systems* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), vol. 31, Curran Associates, Inc., 2018.

[142] R. Sebastiani and P. Trentin, "On optimization modulo theories, maxsmt and sorting networks," 02 2017.

[143] T. Gawlitza and D. Monniaux, "Improving strategies via SMT solving," *CoRR*, vol. abs/1101.2812, 2011.

[144] I. Jorquera, N. Gálvez Ramírez, and C. Castro, "Distintos enfoques de modelamiento matemático para resolver problemas de optimización usando optimization modulo theories," October 2023.

[145] "CSPLib: A problem library for constraints." `http://www.csplib.org`, 1999.

[146] I. P. Gent, C. Jefferson, and P. Nightingale, "Complexity of n-queens completion," *Journal of Artificial Intelligence Research*, vol. 59, pp. 815–848, 2017.

[147] S. Eddins, "The eight queens problem." blog post on MathWorks Blogs, Apr. 2017. Accessed: 2025-07-12.

[148] N. Bjorner, C. Eisenhofer, and L. Kovács, "User-Propagation for Custom Theories in SMT Solving," vol. 3185, pp. 71–79, CEUR-WS.org, Aug. 2022.

[149] L. Kotthoff, "Constraint solvers: An empirical evaluation of design decisions," 2010.

[150] M. A. Ayub, K. A. Kalpoma, H. T. Proma, S. M. Kabir, and R. I. H. Chowdhury, "Exhaustive study of essential constraint satisfaction problem techniques based on N-Queens problem," in *2017 20th International Conference of Computer and Information Technology (ICCIT)*, (Dhaka, Bangladesh), pp. 1–6, IEEE, Dec. 2017.

[151] Özgür Akgün, "CSPLib problem 133: Knapsack problem." `http://www.csplib.org/Problems/prob133`.

[152] S. Xu, X. Chen, X. Pi, C. Joe-Wong, P. Zhang, and H. Y. Noh, "Incentivizing vehicular crowdsensing system for large scale smart city applications," p. 51, 03 2019.

[153] T. C. Hu, L. Landa, and M.-T. Shing, *The Unbounded Knapsack Problem*, pp. 201–217. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[154] H. Becker and L. S. Buriol, "An empirical analysis of exact algorithms for the unbounded knapsack problem," *European Journal of Operational Research*, vol. 277, no. 1, pp. 84–99, 2019.

[155] V. Poirriez, N. Yanev, and R. Andonov, "A hybrid algorithm for the unbounded knapsack problem," *Discrete Optimization*, vol. 6, no. 1, pp. 110–124, 2009.

[156] R. Andonov, V. Poirriez, and S. Rajopadhye, "Efficient dynamic programming for the unbounded knapsack problem," vol. 123, 01 1997.

[157] R. Lagatie, S. Haspeslagh, and P. D. Causmaecker, "Negotiation protocols for distributed nurse rostering." Eindhoven University of Technology Department of Computer Science, 2009.

[158] A. A. El Adoly, M. Gheith, and M. Nashat Fors, "A new formulation and solution for the nurse scheduling problem: A case study in egypt," *Alexandria Engineering Journal*, vol. 57, no. 4, pp. 2289–2298, 2018.

[159] S. J. den Hartog, H. Hoogeveen, and T. C. van der Zanden, "On the complexity of nurse rostering problems," *Operations Research Letters*, vol. 51, no. 5, pp. 483–487, 2023.

[160] M. Vanhoucke and B. Maenhout, "Nsplib–a nurse scheduling problem library: a tool to evaluate (meta-) heuristic procedures," 01 2007.

[161] J. S. Arora, "16 - genetic algorithms for optimum design," in *Introduction to Optimum Design (Second Edition)* (J. S. Arora, ed.), pp. 531–542, San Diego: Academic Press, second edition ed., 2004.

[162] B. Melnikov, "The first acquaintance of schoolchildren with hard programming problems," pp. 1649–1658, 03 2019.

[163] W. J. Cook, B. Korte, L. Lovász, A. Wigderson, and G. M. Ziegler, eds., *The Traveling Salesman Problem*, pp. 527–562. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[164] G. Reinelt, "TSPLIB—A Traveling Salesman Problem Library," *ORSA Journal on Computing*, vol. 3, pp. 376–384, Nov. 1991.

[165] T. Weise, R. Chiong, J. Lassig, K. Tang, S. Tsutsui, W. Chen, Z. Michalewicz, and X. Yao, "Benchmarking Optimization Algorithms: An Open Source Framework for the Traveling Salesman Problem," *IEEE Computational Intelligence Magazine*, vol. 9, pp. 40–52, Aug. 2014.

[166] G. Reinelt, "TSPLIB–a traveling salesman problem library," *ORSA Journal on Computing*, vol. 3, no. 4, pp. 376–384, 1991.

[167] J. Burkardt, "Data for the traveling salesperson problem." http://people.sc.fsu.edu/jburkardt/datasets/tsp/tsp.html, 2011.

[168] B. Hnich, Z. Kiziltan, and T. Walsh, "CSPLib problem 030: Balanced academic curriculum problem (bacp)." `http://www.csplib.org/Problems/prob030`.

[169] M. Chiarandini, L. D. Gaspero, S. Gualandi, and A. Schaerf, "The balanced academic curriculum problem revisited," *Journal of Heuristics*, vol. 18, p. 119–148, 2012.

[170] Universidad Técnica Federico Santa María, "Pregrado en ingeniería informática." `https://informatica.usm.cl/pregrado/`, n.d. Accessed: 2025-12-02.

[171] C. Castro and S. Manzano, "Variable and value ordering when solving balanced academic curriculum problems," 2001.

[172] C. E. Miller, A. W. Tucker, and R. A. Zemlin, "Integer programming formulation of traveling salesman problems," *J. ACM*, vol. 7, p. 326–329, oct 1960.

[173] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli, "Towards smt model checking of array-based systems," in *Automated Reasoning* (A. Armando, P. Baumgartner, and G. Dowek, eds.), (Berlin, Heidelberg), pp. 67–82, Springer Berlin Heidelberg, 2008.

[174] N. Bjørner, L. Moura, L. Nachmanson, and C. Wintersteiger, *Programming Z3*, pp. 148–201. 04 2019.

[175] I. Jorquera, “Resultados Experimentales: Automatizando Lógicas SMT - Evaluación de Combinaciones.” https://gitlab.com/ignacio.jorquera/automatizando-logicas-smt-resultados, 2025. Repositorio de GitLab. Accedido: 12-12-2025.

Apéndice A

Anexos

A.1. Transformaciones de modelos

En este Anexo se incluyen las transformaciones de los modelos no abordados en la Sección 3.3.

A.1.1. N-Queens

Transformación de `LIA` a `LRA`

Parámetros:

- N: Número de reinas que se deben posicionar en un tablero de tamaño $N \times N$, tal que $N \in \mathbb{Z}^+$.

Variables:

- x_c: Número de la fila en la columna c donde se posiciona una reina, $x_c \in \mathbb{R}; \forall c \in \{1, ..., N\}$.

Restricciones del problema:

1. **Dominio de las variables:** Cada reina debe ubicarse dentro de los límites del tablero:

$$\bigvee_{k=1}^{N} x_c = k; \quad \forall c \in \{1, ..., N\}$$

2. **Una reina por fila:** No puede haber más de una reina en la misma fila:

$$x_{c_i} \neq x_{c_j}; \quad \forall c_i \in \{1, ..., N-1\}), \quad \forall c_j \in \{c_i + 1, ..., N\})$$

3. **Una reina por diagonal:** Dos reinas no pueden ubicarse en la misma diagonal:

$$x_{c_j} \neq x_{c_i} + (c_j - c_i)$$

$$x_{c_i} \neq x_{c_j} + (c_j - c_i)$$

$$\forall c_i \in \{1, ..., N-1\}, \quad \forall c_j \in \{c_i + 1, ..., N\}$$

Transformación de `LIA` a `QF_BV`

Parámetros:

- N: Número de reinas que se deben posicionar en un tablero de tamaño $N \times N$, tal que $N \in \mathbb{Z}^+$.

Variables:

- x_c: Número de la fila en la columna c donde se posiciona una reina, $x_c \in \mathbb{Z}_2^{\lceil \log_2(N) \rceil + 1}$; $\forall c \in \{1, ..., N\}$.

Restricciones del problema:

1. **Dominio de las variables:** Cada reina debe ubicarse dentro de los límites del tablero:

$$\bigvee_{k=1}^{N} x_c = k_2; \quad \forall c \in \{1, ..., N\}$$

2. **Una reina por fila:** No puede haber más de una reina en la misma fila:

$$x_{c_i} \neq x_{c_j}; \quad \forall c_i \in \{1, ..., N-1\}), \quad \forall c_j \in \{c_i + 1, ..., N\})$$

3. **Una reina por diagonal:** Dos reinas no pueden ubicarse en la misma diagonal:

$$x_{c_j} \neq x_{c_i} + (c_j - c_i)_2$$

$$x_{c_i} \neq x_{c_j} + (c_j - c_i)_2$$
$$\forall c_i \in \{1, ..., N-1\}, \quad \forall c_j \in \{c_i + 1, ..., N\}$$

Transformación de `LIA` a `QF_ALIA`

Parámetros:

- N: Número de reinas que se deben posicionar en un tablero de tamaño $N \times N$, tal que $N \in \mathbb{Z}^+$.

Variables:

- x: Arreglo unidimensional de enteros, define en que fila se posicionará una reina para cada columna señalada en el índice del arreglo, $x \in \mathcal{A}_I^E, \forall I, E \in \mathbb{Z}$

Restricciones del problema:

1. **Dominio de las variables:** Cada reina debe ubicarse dentro de los límites del tablero:

$$\bigvee_{k=1}^{N} x[c] = k; \quad \forall c \in \{1, ..., N\}$$

2. **Una reina por fila:** No puede haber más de una reina en la misma fila:

$$x[c_i] \neq x[c_j]; \quad \forall c_i \in \{1, ..., N-1\}), \quad \forall c_j \in \{c_i + 1, ..., N\})$$

3. **Una reina por diagonal:** Dos reinas no pueden ubicarse en la misma diagonal:

$$x[c_j] \neq x[c_i] + (c_j - c_i)$$

$$x[c_i] \neq x[c_j] + (c_j - c_i)$$
$$\forall c_i \in \{1, ..., N-1\}, \quad \forall c_j \in \{c_i + 1, ..., N\}$$

A.1.2. UKP

Transformación de `LIA` a `LRA`

Parámetros:

- N: Número total de tipos de objetos, $N \in \mathbb{Z}^+$
- C: Capacidad máxima de la mochila, $C \in \mathbb{Z}^+$
- W_i: Peso asociado al objeto i, $W_i \in \mathbb{Z}; \quad \forall i \in \{1, \ldots, N\}$
- V_i: Valor asociado al objeto i, $V_i \in \mathbb{Z}; \quad \forall i \in \{1, \ldots, N\}$

Variables:

- x_i: Cantidad de veces que el objeto i es incluido en la mochila, $x_i \in \mathbb{R}; \quad \forall i \in \{1, \ldots, N\}$

Función objetivo:

- **Maximizar el valor total en la mochila, v:**

$$max \quad v = \sum_{i=1}^{N} (V_i \times x_i)$$

Restricciones del problema:

1. **Dominio de las variables:** Cada objeto puede ser agregado cero o más veces. El límite superior se determina teóricamente a partir del peso del objeto y de la capacidad de la mochila:

$$\bigvee_{k=0}^{\lceil \frac{C}{W_i} \rceil} x_i = k; \quad \forall i \in \{1, \ldots, N\}$$

2. **Restricción de capacidad:** La suma total de los pesos de los objetos seleccionados no puede superar la capacidad máxima disponible de la mochila.

$$\sum_{i=1}^{N} (W_i \times x_i) \leq C$$

Transformación de `LIA` a `QF_BV`

Parámetros:

- N: Número total de tipos de objetos, $N \in \mathbb{Z}^+$
- C: Capacidad máxima de la mochila, $C \in \mathbb{Z}^+$
- W_i: Peso asociado al objeto i, $W_i \in \mathbb{Z}; \quad \forall i \in \{1, \ldots, N\}$
- W_i^*: Representación binaria del peso asociado al objeto i, $W_i^* \in \mathbb{Z}_2^{\lceil \log_2(\text{máx}(W_i)) \rceil + 1}; \quad \forall i \in \{1, \ldots, N\}$
- V_i: Valor asociado al objeto i, $V_i \in \mathbb{Z}; \quad \forall i \in \{1, \ldots, N\}$
- V_i^* : Representación binaria del valor asociado a cada objeto, $V_i^* \in \mathbb{Z}_2^{\lceil \log_2(\text{máx}(V_i)) \rceil + 1}; \quad \forall i \in \{1, ..., N\}$

Variables:

- x_i: Cantidad de veces que el objeto i es incluido en la mochila, $x_i \in \mathbb{Z}_2^{\lceil \log_2(N) \rceil + 1}$

Función objetivo:

- **Maximizar el valor total en la mochila, v:**

$$max \quad v = \sum_{i=1}^{N} (V_i^* \times x_i)$$

Restricciones del problema:

1. **Dominio de las variables:** Cada objeto puede ser agregado cero o más veces. El límite superior se determina teóricamente a partir del peso del objeto y de la capacidad de la mochila:
$$\bigvee_{k=0}^{\lceil \frac{C}{W_i} \rceil} x_i = k_2; \quad \forall i \in \{1, \ldots, N\}$$
2. **Restricción de capacidad:** La suma total de los pesos de los objetos seleccionados no puede superar la capacidad máxima disponible de la mochila.
$$\sum_{i=1}^{N} (W_i^* \times x_i) \leq C$$

Transformación de `LIA` a `QF_ALIA`

Parámetros:

- N: Número total de tipos de objetos, $N \in \mathbb{Z}^+$
- C: Capacidad máxima de la mochila, $C \in \mathbb{Z}^+$
- W_i: Peso asociado al objeto i, $W_i \in \mathbb{Z}; \quad \forall i \in \{1, \ldots, N\}$
- V_i: Valor asociado al objeto i, $V_i \in \mathbb{Z}; \quad \forall i \in \{1, \ldots, N\}$

Variables:

- x: Arreglo unidimensional de enteros, define la cantidad de veces que se agrega el objeto indicado por el índice en la mochila, $x \in \mathcal{A}_I^E, \forall I, E \in \mathbb{Z}$

Función objetivo:

- **Maximizar el valor total en la mochila, v:**

$$max \quad v = \sum_{i=1}^{N} (V_i \times x[i])$$

Restricciones del problema:

1. **Dominio de las variables:** Cada objeto puede ser agregado cero o más veces. El límite superior se determina teóricamente a partir del peso del objeto y de la capacidad de la mochila:
$$\bigvee_{k=0}^{\lceil \frac{C}{W_i} \rceil} x[i] = k; \quad \forall i \in \{1, \ldots, N\}$$
2. **Restricción de capacidad:** La suma total de los pesos de los objetos seleccionados no puede superar la capacidad máxima disponible de la mochila.
$$\sum_{i=1}^{N} (W_i \times x[i]) \leq C$$

A.1.3. NSP

Transformación de `LIA` a `LRA`

Parámetros:

- N : Número de *enfermeras*, $N \in \mathbb{Z}^+$
- D : Número de días del horizonte temporal de planificación, $D \in \mathbb{Z}^+$
- S : Número total de turnos posibles por día (el último representa el turno libre), $S \in \mathbb{Z}^+$
- $C_{d,s}$: Matriz de cobertura que indica la cantidad mínima de *enfermeras* requeridas para el día d y turno s, $C_{d,s} \in \mathbb{Z}_0^+; \quad \forall d \in \{1, ..., D\}, \quad \forall s \in \{1, ..., S\}$
- $P_{n,d,s}$: Matriz de preferencias, donde cada valor indica el grado de preferencia de la *enfermera* n para trabajar en el turno s del día d, $P_{n,d,s} \in \mathbb{Z}_0^+; \quad \forall n \in \{1, ..., N\}, \quad \forall d \in \{1, ..., D\}, \quad \forall s \in \{1, ..., S\}$
- min_wd: Cantidad mínima de días de trabajo por *enfermera*, $min\_wd \in \mathbb{Z}_0^+$
- max_wd: Cantidad máxima de días de trabajo por *enfermera*, $max\_wd \in \mathbb{Z}_0^+$
- min_cd : Cantidad mínima de días consecutivos de trabajo, $min\_cd \in \mathbb{Z}_0^+$
- max_cd : Cantidad máxima de días consecutivos de trabajo, $max\_cd \in \mathbb{Z}_0^+$
- min_cs_s : Cantidad mínima de días consecutivos que una *enfermera* puede trabajar en el turno s, $min\_cs_s \in \mathbb{Z}_0^+, s \in \{1, ..., S\}$
- max_cs_s : Cantidad máxima de días consecutivos que una *enfermera* puede trabajar en el turno s, $max\_cs_s \in \mathbb{Z}_0^+, s \in \{1, ..., S\}$
- min_s_s : Cantidad mínima de veces que se debe asignar el turno s a una misma *enfermera* durante el horizonte temporal, $min\_s_s \in \mathbb{Z}_0^+, s \in \{1, ..., S\}$
- max_s_s : Cantidad máxima de veces que se debe asignar el turno s a una misma *enfermera* durante el horizonte temporal, $max\_s_s \in \mathbb{Z}_0^+, s \in \{1, ..., S\}$

Variables:

- $x_{n,d,s} = \begin{cases} 1.0 & \text{si la } \textit{enfermera } n \text{ es asignado al día } d \text{ en el turno } s, \\ 0.0 & \text{en caso contrario} \end{cases}$

$$x_{n,d,s} \in \mathbb{R}; \quad \forall n \in \{1, ..., N\}, \quad \forall d \in \{1, ..., D\}, \quad \forall s \in \{1, ..., S\}$$

Función objetivo:

- **Maximizar la satisfacción general de la planificación, H:**

$$max \quad H = \sum_{n=1}^{N} \sum_{d=1}^{D} \sum_{s=1}^{S} (P_{n,d,s} \times x_{n,d,s})$$

Restricciones del problema:

1. **Dominio de las variables:** Cada asignación es binaria, indicando si el turno es asignado o no.

$$\bigvee_{k=0}^{1} x_{n,d,s} = k; \quad \forall n \in \{1, \ldots, N\}, \quad \forall d \in \{1, \ldots, D\}, \quad \forall s \in \{1, \ldots, S\}$$

2. **Asignación única por día:** Cada *enfermera* debe estar asignada exactamente a un turno (incluido el libre) por día.

$$\sum_{s=1}^{S} x_{n,d,s} = 1; \quad \forall n \in \{1, ..., N\}, \quad \forall d \in \{1, ..., D\}$$

3. **Cobertura mínima de turnos:** Cada día y turno laboral debe tener al menos la cantidad requerida de *enfermeras*.

$$\sum_{n=1}^{N} x_{n,d,s} \geq C_{d,s}; \quad \forall d \in \{1, ..., D\}, \quad \forall s \in \{1, ..., S-1\}$$

4. **Límites de días totales de trabajo:** Cada *enfermera* debe cumplir con los límites de días trabajados en el horizonte temporal.

$$\sum_{d=1}^{D} \sum_{s=1}^{S-1} x_{n,d,s} \geq min\_wd; \quad \forall n \in \{1, ..., N\}$$

$$\sum_{d=1}^{D} \sum_{s=1}^{S-1} x_{n,d,s} \leq max\_wd; \quad \forall n \in \{1, ..., N\}$$

5. **Límites de asignación por turno:** Cada *enfermera* debe trabajar un turno s dentro de un rango específico de frecuencia.

$$\sum_{d=1}^{D} x_{n,d,s} \geq min\_s_s; \quad \forall n \in \{1, ..., N\}, \quad \forall s \in \{1, ..., S\}$$

$$\sum_{d=1}^{D} x_{n,d,s} \leq max\_s_s; \quad \forall n \in \{1, ..., N\}, \quad \forall s \in \{1, ..., S\}$$

6. **Restricción de días consecutivos de trabajo:** Cada *enfermera* debe trabajar al menos min_cd y a lo más max_cd días consecutivos.

$$\sum_{j=max\{i+1,2\}}^{min\{i+k+1,D\}} \sum_{s=1}^{S-1} x_{n,j-1,s} - 2 \times \sum_{s=1}^{S-1} x_{n,i-1,s} - 2 \times \sum_{s=1}^{S-1} x_{n,i+k-1,s} \leq k-2;$$

$$\forall n \in \{1, ..., N\}, \quad \forall k \in \{min\{2, min\_cd\}, ..., min\_cd + 1\}, \quad \forall i \in \{1, ..., D + 2 - k\}$$

$$\sum_{d=j}^{min\{D, j+max\_cd+1\}} \sum_{s=1}^{S-1} x_{n,d,s} \leq max\_cd; \quad \forall n \in \{1, ..., N\}, \quad \forall j \in \{1, ..., D - max\_cd\}$$

7. **Restricción de días consecutivos por turno:** Si una *enfermera* trabaja en un turno s, debe repetirlo entre min_cs_s y max_cs_s veces consecutivas.

$$\sum_{j=max\{i+1,2\}}^{min\{i+k+1,D\}} x_{n,j-1,s} - 2 \times x_{n,i-1,s} - 2 \times x_{n,i+k-1,s} \leq k-2;$$

$$\forall n \in \{1, ..., N\}, \quad \forall s \in \{1, ..., S-1\},$$
$$\forall k \in \{min\{2, min\_cs_s\}, ..., min\_cs_s + 1\}, \quad \forall i \in \{1, ..., D + 2 - k\}$$

$$\sum_{d=j}^{min\{D, j+max\_cs_s+1\}} x_{n,d,s} \leq max\_cs_s;$$

$$\forall n \in \{1, ..., N\}, \quad \forall s \in \{1, ..., S-1\}, \quad \forall j \in \{1, ..., D - max\_cd\}$$

Transformación de `LIA` a `QF_BV`

Parámetros:

- N : Número de *enfermeras*, $N \in \mathbb{Z}^+$
- D : Número de días del horizonte temporal de planificación, $D \in \mathbb{Z}^+$
- S : Número total de turnos posibles por día (el último representa el turno libre), $S \in \mathbb{Z}^+$
- $C_{d,s}$: Matriz de cobertura que indica la cantidad mínima de *enfermeras* requeridas para el día d y turno s, $C_{d,s} \in \mathbb{Z}_0^+$; $\quad \forall d \in \{1, ..., D\}, \quad \forall s \in \{1, ..., S\}$
- $C^*_{d,s}$: Representación binaria de $C_{d,s}$, $C^*_{d,s} \in \mathbb{Z}_2^{\lceil \log_2(max(C_{d,s})) \rceil + 1}$

- $P_{n,d,s}$: Matriz de preferencias, donde cada valor indica el grado de preferencia de la *enfermera n* para trabajar en el turno s del día d, $P_{n,d,s} \in \mathbb{Z}_0^+; \quad \forall n \in \{1, ..., N\}, \quad \forall d \in \{1, ..., D\}, \quad \forall s \in \{1, ..., S\}$
- $P^*_{n,d,s}$: Representación binaria de $P_{n,d,s}$, $P^*_{n,d,s} \in \mathbb{Z}_2^{\lceil \log_2(max(P_{n,d,s})) \rceil + 1}$
- min_wd: Cantidad mínima de días de trabajo por *enfermera*, $min\_wd \in \mathbb{Z}_0^+$
- min_wd^* : Representación binaria de min_wd, $min\_wd^* \in \mathbb{Z}_2^{\lceil \log_2(min\_wd) \rceil + 1}$
- max_wd: Cantidad máxima de días de trabajo por *enfermera*, $max\_wd \in \mathbb{Z}_0^+$
- max_wd^* : Representación binaria de max_wd, $max\_wd^* \in \mathbb{Z}_2^{\lceil \log_2(max\_wd) \rceil + 1}$
- min_cd : Cantidad mínima de días consecutivos de trabajo, $min\_cd \in \mathbb{Z}_0^+$
- min_cd^* : Representación binaria de min_cd, $min\_cd^* \in \mathbb{Z}_2^{\lceil \log_2(min\_cd) \rceil + 1}$
- max_cd : Cantidad máxima de días consecutivos de trabajo, $max\_cd \in \mathbb{Z}_0^+$
- max_cd^* : Representación binaria de max_cd, $max\_cd^* \in \mathbb{Z}_2^{\lceil \log_2(max\_cd) \rceil + 1}$
- min_cs_s : Cantidad mínima de días consecutivos que una *enfermera* puede trabajar en el turno s, $min\_cs_s \in \mathbb{Z}_0^+$, $s \in \{1, ..., S\}$
- $min\_cs_s^*$: Representación binaria de min_cs_s, $min\_cs_s* \in \mathbb{Z}_2^{\lceil \log_2(max(min\_cs_s)) \rceil + 1}$
- max_cs_s : Cantidad máxima de días consecutivos que una *enfermera* puede trabajar en el turno s, $max\_cs_s \in \mathbb{Z}_0^+$, $s \in \{1, ..., S\}$
- $max\_cs_s^*$: Representación binaria de max_cs_s, $max\_cs_s* \in \mathbb{Z}_2^{\lceil \log_2(max(max\_cs_s)) \rceil + 1}$
- min_s_s : Cantidad mínima de veces que se debe asignar el turno s a una misma *enfermera* durante el horizonte temporal, $min\_s_s \in \mathbb{Z}_0^+$, $s \in \{1, ..., S\}$
- $min\_s_s^*$: Representación binaria de min_s_s, $min\_s_s* \in \mathbb{Z}_2^{\lceil \log_2(max(min\_s_s)) \rceil + 1}$
- max_s_s : Cantidad máxima de veces que se debe asignar el turno s a una misma *enfermera* durante el horizonte temporal, $max\_s_s \in \mathbb{Z}_0^+$, $s \in \{1, ..., S\}$
- $max\_s_s^*$: Representación binaria de max_s_s, $max\_s_s* \in \mathbb{Z}_2^{\lceil \log_2(max(max\_s_s)) \rceil + 1}$

Variables:

- $$x_{n,d,s} = \begin{cases} 1_2 & \text{si la } \textit{enfermera } n \text{ es asignado al día } d \text{ en el turno } s, \\ 0_2 & \text{en caso contrario} \end{cases}$$

$$x_{n,d,s} \in \mathbb{Z}_2^1; \quad \forall n \in \{1, ..., N\}, \quad \forall d \in \{1, ..., D\}, \quad \forall s \in \{1, ..., S\}$$

Función objetivo:

- **Maximizar la satisfacción general de la planificación, H:**

$$max \quad H = \sum_{n=1}^{N} \sum_{d=1}^{D} \sum_{s=1}^{S} \left(P^*_{n,d,s} \times x_{n,d,s} \right)$$

Restricciones del problema:

1. **Dominio de las variables:** Cada asignación es binaria, indicando si el turno es asignado o no.

$$\bigvee_{k=0}^{1} x_{n,d,s} = k_2; \quad \forall n \in \{1, \ldots, N\}, \quad \forall d \in \{1, \ldots, D\}, \quad \forall s \in \{1, \ldots, S\}$$

2. **Asignación única por día:** Cada *enfermera* debe estar asignada exactamente a un turno (incluido el libre) por día.

$$\sum_{s=1}^{S} x_{n,d,s} = 1_2; \quad \forall n \in \{1, ..., N\}, \quad \forall d \in \{1, ..., D\}$$

3. **Cobertura mínima de turnos:** Cada día y turno laboral debe tener al menos la cantidad requerida de *enfermeras*.

$$\sum_{n=1}^{N} x_{n,d,s} \geq C^*_{d,s}; \quad \forall d \in \{1, ..., D\}, \quad \forall s \in \{1, ..., S-1\}$$

4. **Límites de días totales de trabajo:** Cada *enfermera* debe cumplir con los límites de días trabajados en el horizonte temporal.

$$\sum_{d=1}^{D} \sum_{s=1}^{S-1} x_{n,d,s} \geq min\_wd^*; \quad \forall n \in \{1, ..., N\}$$

$$\sum_{d=1}^{D} \sum_{s=1}^{S-1} x_{n,d,s} \leq max\_wd^*; \quad \forall n \in \{1, ..., N\}$$

5. **Límites de asignación por turno:** Cada *enfermera* debe trabajar un turno s dentro de un rango específico de frecuencia.

$$\sum_{d=1}^{D} x_{n,d,s} \geq min\_s^*_s; \quad \forall n \in \{1, ..., N\}, \quad \forall s \in \{1, ..., S\}$$

$$\sum_{d=1}^{D} x_{n,d,s} \leq max\_s^*_s; \quad \forall n \in \{1, ..., N\}, \quad \forall s \in \{1, ..., S\}$$

6. **Restricción de días consecutivos de trabajo:** Cada *enfermera* debe trabajar al menos min_cd y a lo más max_cd días consecutivos.

$$\sum_{j=max\{i+1,2\}}^{min\{i+k+1,D\}} \sum_{s=1}^{S-1} x_{n,j-1,s} - 2_2 \times \sum_{s=1}^{S-1} x_{n,i-1,s} - 2_2 \times \sum_{s=1}^{S-1} x_{n,i+k-1,s} \leq k_2 - 2_2;$$

$$\forall n \in \{1, ..., N\}, \quad \forall k \in \{min\{2, min\_cd\}, ..., min\_cd + 1\}, \quad \forall i \in \{1, ..., D + 2 - k\}$$

$$\sum_{d=j}^{min\{D,j+max\_cd+1\}} \sum_{s=1}^{S-1} x_{n,d,s} \leq max\_cd^*; \quad \forall n \in \{1, ..., N\}, \quad \forall j \in \{1, ..., D - max\_cd\}$$

7. **Restricción de días consecutivos por turno:** Si una *enfermera* trabaja en un turno s, debe repetirlo entre min_cs_s y max_cs_s veces consecutivas.

$$\sum_{j=max\{i+1,2\}}^{min\{i+k+1,D\}} x_{n,j-1,s} - 2_2 \times x_{n,i-1,s} - 2_2 \times x_{n,i+k-1,s} \leq k_2 - 2_2;$$

$$\forall n \in \{1, ..., N\}, \quad \forall s \in \{1, ..., S - 1\},$$
$$\forall k \in \{min\{2, min\_cs_s\}, ..., min\_cs_s + 1\}, \quad \forall i \in \{1, ..., D + 2 - k\}$$

$$\sum_{d=j}^{min\{D,j+max\_cs_s+1\}} x_{n,d,s} \leq max\_cs_s^*;$$

$$\forall n \in \{1, ..., N\}, \quad \forall s \in \{1, ..., S - 1\}, \quad \forall j \in \{1, ..., D - max\_cd\}$$

Transformación de `LIA` a `QF_ALIA`

Parámetros:

- N : Número de *enfermeras*, $N \in \mathbb{Z}^+$
- D : Número de días del horizonte temporal de planificación, $D \in \mathbb{Z}^+$
- S : Número total de turnos posibles por día (el último representa el turno libre), $S \in \mathbb{Z}^+$
- $C_{d,s}$: Matriz de cobertura que indica la cantidad mínima de *enfermeras* requeridas para el día d y turno s, $C_{d,s} \in \mathbb{Z}_0^+$; $\forall d \in \{1, ..., D\}, \quad \forall s \in \{1, ..., S\}$

- $P_{n,d,s}$: Matriz de preferencias, donde cada valor indica el grado de preferencia de la *enfermera n* para trabajar en el turno s del día d, $P_{n,d,s} \in \mathbb{Z}_0^+; \quad \forall n \in \{1, ..., N\}, \quad \forall d \in \{1, ..., D\}, \quad \forall s \in \{1, ..., S\}$
- min_wd: Cantidad mínima de días de trabajo por *enfermera*, $min\_wd \in \mathbb{Z}_0^+$
- max_wd: Cantidad máxima de días de trabajo por *enfermera*, $max\_wd \in \mathbb{Z}_0^+$
- min_cd : Cantidad mínima de días consecutivos de trabajo, $min\_cd \in \mathbb{Z}_0^+$
- max_cd : Cantidad máxima de días consecutivos de trabajo, $max\_cd \in \mathbb{Z}_0^+$
- min_cs_s : Cantidad mínima de días consecutivos que una *enfermera* puede trabajar en el turno s, $min\_cs_s \in \mathbb{Z}_0^+, s \in \{1, ..., S\}$
- max_cs_s : Cantidad máxima de días consecutivos que una *enfermera* puede trabajar en el turno s, $max\_cs_s \in \mathbb{Z}_0^+, s \in \{1, ..., S\}$
- min_s_s : Cantidad mínima de veces que se debe asignar el turno s a una misma *enfermera* durante el horizonte temporal, $min\_s_s \in \mathbb{Z}_0^+, s \in \{1, ..., S\}$
- max_s_s : Cantidad máxima de veces que se debe asignar el turno s a una misma *enfermera* durante el horizonte temporal, $max\_s_s \in \mathbb{Z}_0^+, s \in \{1, ..., S\}$

Variables:

- x: Arreglo tridimensional de enteros, donde $x[n][d][s] = 1$ si *enfermera n* es asignado al día d en el turno s, y 0 en caso contrario; $x \in \mathcal{A}_I^E, \quad \forall I \in \mathbb{Z}, \forall E \in \mathcal{A}_{I^*}^{E^*}, \forall I^* \in \mathbb{Z}, \forall E^* \in \mathcal{A}_{I^{**}}^{E^{**}}, \forall I^{**}, E^{**} \in \mathbb{Z}$

$$x[n][d][s] = \begin{cases} 1 & \text{si la } \textit{enfermera } n \text{ es asignado al día } d \text{ en el turno } s, \\ 0 & \text{en caso contrario} \end{cases}$$

$$x[n][d][s] \in \mathbb{Z}; \quad \forall n \in \{1, ..., N\}, \quad \forall d \in \{1, ..., D\}, \quad \forall s \in \{1, ..., S\}$$

Función objetivo:

- **Maximizar la satisfacción general de la planificación, H:**

$$max \quad H = \sum_{n=1}^{N} \sum_{d=1}^{D} \sum_{s=1}^{S} (P_{n,d,s} \times x[n][d][s])$$

Restricciones del problema:

1. **Dominio de las variables:** Cada asignación es binaria, indicando si el turno es asignado o no.

$$\bigvee_{k=0}^{1} x_{n,d,s} = k; \quad \forall n \in \{1, \ldots, N\}, \quad \forall d \in \{1, \ldots, D\}, \quad \forall s \in \{1, \ldots, S\}$$

2. **Asignación única por día:** Cada *enfermera* debe estar asignada exactamente a un turno (incluido el libre) por día.

$$\sum_{s=1}^{S} x[n][d][s] = 1; \quad \forall n \in \{1, ..., N\}, \quad \forall d \in \{1, ..., D\}$$

3. **Cobertura mínima de turnos:** Cada día y turno laboral debe tener al menos la cantidad requerida de *enfermeras*.

$$\sum_{n=1}^{N} x[n][d][s] \geq C_{d,s}; \quad \forall d \in \{1, ..., D\}, \quad \forall s \in \{1, ..., S-1\}$$

4. **Límites de días totales de trabajo:** Cada *enfermera* debe cumplir con los límites de días trabajados en el horizonte temporal.

$$\sum_{d=1}^{D} \sum_{s=1}^{S-1} x[n][d][s] \geq min\_wd; \quad \forall n \in \{1, ..., N\}$$

$$\sum_{d=1}^{D} \sum_{s=1}^{S-1} x[n][d][s] \leq max\_wd; \quad \forall n \in \{1, ..., N\}$$

5. **Límites de asignación por turno:** Cada *enfermera* debe trabajar un turno s dentro de un rango específico de frecuencia.

$$\sum_{d=1}^{D} x[n][d][s] \geq min\_s_s; \quad \forall n \in \{1, ..., N\}, \quad \forall s \in \{1, ..., S\}$$

$$\sum_{d=1}^{D} x[n][d][s] \leq max\_s_s; \quad \forall n \in \{1, ..., N\}, \quad \forall s \in \{1, ..., S\}$$

6. **Restricción de días consecutivos de trabajo:** Cada *enfermera* debe trabajar al menos *min_cd* y a lo más *max_cd* días consecutivos.

$$\sum_{j=max\{i+1,2\}}^{min\{i+k+1,D\}} \sum_{s=1}^{S-1} x[n][j-1][s] - 2 \times \sum_{s=1}^{S-1} x[n][i-1][s] - 2 \times \sum_{s=1}^{S-1} x[n][i+k-1][s] \leq k-2;$$

$$\forall n \in \{1, ..., N\}, \quad \forall k \in \{min\{2, min\_cd\}, ..., min\_cd + 1\}, \quad \forall i \in \{1, ..., D + 2 - k\}$$

$$\sum_{d=j}^{min\{D, j+max\_cd+1\}} \sum_{s=1}^{S-1} x[n][d][s] \leq max\_cd; \quad \forall n \in \{1, ..., N\}, \quad \forall j \in \{1, ..., D - max\_cd\}$$

7. **Restricción de días consecutivos por turno:** Si un *enfermera* trabaja en un turno s, debe repetirlo entre min_cs_s y max_cs_s veces consecutivas.

$$\sum_{j=max\{i+1,2\}}^{min\{i+k+1,D\}} x[n][j-1][s] - 2 \times x[n][i-1][s] - 2 \times x[n][i+k-1][s] \leq k - 2;$$

$$\forall n \in \{1, ..., N\}, \quad \forall s \in \{1, ..., S-1\},$$
$$\forall k \in \{min\{2, min\_cs_s\}, ..., min\_cs_s + 1\}, \quad \forall i \in \{1, ..., D + 2 - k\}$$

$$\sum_{d=j}^{min\{D, j+max\_cs_s+1\}} x[n][d][s] \leq max\_cs_s;$$

$$\forall n \in \{1, ..., N\}, \quad \forall s \in \{1, ..., S-1\}, \quad \forall j \in \{1, ..., D - max\_cd\}$$

A.1.4. BACP

Transformación de `LIA` a `LRA`

Parámetros:

- M : Número de cursos, $M \in \mathbb{Z}^+$
- N : Número de periodos académicos, $N \in \mathbb{Z}^+$
- α_i : Número de créditos del curso i, $\alpha_i \in \mathbb{Z}^+$, $\forall i \in \mathbb{Z}^+ : i \in \{1, ..., M\}$
- β : Carga académica **mínima** permitida por periodo, $\beta \in \mathbb{Z}^+$
- γ : Carga académica **máxima** permitida por periodo, $\gamma \in \mathbb{Z}^+$
- δ : Cantidad de cursos **mínima** por periodo, $\delta \in \mathbb{Z}^+$
- ϵ : Cantidad de cursos **máxima** por periodo, $\epsilon \in \mathbb{Z}^+$

Variables:

- $x_{i,j} = \begin{cases} 1.0 & \text{si el curso } i \text{ es asignado al periodo } j, \\ 0.0 & \text{en caso contrario} \end{cases}$

$$x_{i,j} \in \mathbb{R}; \quad \forall i \in \{1, ..., M\}, \quad \forall j \in \{1, ..., N\}$$

- c_j : carga académica del periodo $j, c_j \in \mathbb{R}; \quad \forall j \in \{1, ..., N\}$
- C : Mayor carga académica registrada a lo largo del currículum, $C \in \mathbb{R}$

Función objetivo:

- Minimizar la mayor carga académica C entre todos los periodos, buscando un currículum equilibrado:

$$min \quad C = max\{c_1, ..., c_N\}$$

Restricciones del problema:

1. **Dominio de las variables:** Cada variable $x_{i,j}$ solo puede tomar los valores 0.0 o 1.0, indicando si el curso i es asignado o no al periodo j:

$$\bigvee_{k=0}^{1} x_{i,j} = k; \quad \forall i \in \{1, ..., M\}, \quad \forall j \in \{1, ..., N\}$$

2. **Definición de la carga máxima:** C se utiliza en la función objetivo como $C = max\{c_1, ..., c_n\}$, lo cual se logra representar mezclando la minimización de C junto a la siguiente inecuación:

$$C \geq c_j; \quad \forall j \in \{1, ..., N\}$$

3. **Cálculo de la carga académica por periodo:** La carga académica para el periodo j se define como:

$$c_j = \sum_{i=1}^{M} \left(\alpha_i \times x_{i,j}\right); \quad \forall j \in \{1, ..., N\}$$

4. **Asignación de curso por periodo:** Cada curso i debe ser asignados a un periodo j:

$$\sum_{j=1}^{N} x_{i,j} = 1; \quad \forall i \in \{1, ..., M\}$$

5. **Prerrequisitos de cursos:** si el curso b requiere como prerrequisito al curso a, entonces b no puede ser asignado a un periodo anterior al de a:

$$x_{b,j} \leq \sum_{r=1}^{j-1} x_{a,r} = 1; \quad \forall j \in \{2, ..., N\}, \quad \forall a \in \{1, ..., M-1\}, \quad \forall b \in \{a, ..., M\}$$

6. **Carga mínima por periodo:** La carga académica de cada periodo j debe ser mayor o igual a la carga mínima permitida:

$$c_j \geq \beta; \quad \forall j \in \{1, ..., N\}$$

7. **Carga máxima por periodo:** La carga académica de cada periodo j debe ser menor o igual a la carga máxima permitida:

$$c_j \leq \gamma; \quad \forall j \in \{1, ..., N\}$$

8. **Mínima cantidad de cursos por periodo:** El número de cursos del periodo j debe ser mayor o igual a la cantidad mínima permitida:

$$\sum_{i=1}^{M} x_{i,j} \geq \delta; \quad \forall j \in \{1, ..., N\}$$

9. **Máxima cantidad de cursos por periodo:** El número de cursos del periodo j debe ser menor o igual que la cantidad máxima permitida:

$$\sum_{i=1}^{M} x_{i,j} \leq \epsilon; \quad \forall j \in \{1, ..., N\}$$

Transformación de `LIA` a `QF_BV`

Parámetros:

- M : Número de cursos, $M \in \mathbb{Z}^+$
- M^* : Representación binaria de M, $M^* \in \mathbb{Z}_2^{\lceil \log_2(M) \rceil + 1}$
- N : Número de periodos académicos, $N \in \mathbb{Z}^+$
- N^* : Representación binaria de N, $N^* \in \mathbb{Z}_2^{\lceil \log_2(N) \rceil + 1}$
- α_i : Número de créditos del curso i, $\alpha_i \in \mathbb{Z}^+$, $\forall i \in \mathbb{Z}^+ : i \in \{1, ..., M\}$
- α_i^* : Representación binaria de α_i, $\alpha_i^* \in \mathbb{Z}_2^{\lceil \log_2(max(\alpha_i)) \rceil + 1}$

- β : Carga académica **mínima** permitida por periodo, $\beta \in \mathbb{Z}^+$
- β^* : Representación binaria de β, $\beta^* \in \mathbb{Z}_2^{\lceil \log_2(\beta) \rceil + 1}$
- γ : Carga académica **máxima** permitida por periodo, $\gamma \in \mathbb{Z}^+$
- γ^* : Representación binaria de γ, $\gamma^* \in \mathbb{Z}_2^{\lceil \log_2(\gamma) \rceil + 1}$
- δ : Cantidad de cursos **mínima** por periodo, $\delta \in \mathbb{Z}^+$
- δ^* : Representación binaria de δ, $\delta^* \in \mathbb{Z}_2^{\lceil \log_2(\delta) \rceil + 1}$
- ϵ : Cantidad de cursos **máxima** por periodo, $\epsilon \in \mathbb{Z}^+$
- ϵ^* : Representación binaria de ϵ, $\epsilon^* \in \mathbb{Z}_2^{\lceil \log_2(\epsilon) \rceil + 1}$

Variables:

- $x_{i,j} = \begin{cases} 1_2 & \text{si el curso } i \text{ es asignado al periodo } j, \\ 0_2 & \text{en caso contrario} \end{cases}$

$$x_{i,j} \in \mathbb{Z}_2^1; \quad \forall i \in \{1, ..., M\}, \quad \forall j \in \{1, ..., N\}$$

- c_j : carga académica del periodo j, $c_j \in \mathbb{Z}_2^{\lceil \log_2(max(\sum \alpha, \beta, \gamma)) \rceil + 1}$; $\quad \forall j \in \{1, ..., N\}$
- C : Mayor carga académica registrada a lo largo del currículum, $C \in \mathbb{Z}_2^{\lceil \log_2(max(\sum \alpha, \beta, \gamma)) \rceil + 1}$

Función objetivo:

- Minimizar la mayor carga académica C entre todos los periodos, buscando un currículum equilibrado:

$$min \quad C = max\{c_1, ..., c_N\}$$

Restricciones del problema:

1. **Dominio de las variables:** Cada variable $x_{i,j}$ solo puede tomar los valores 0 o 1, indicando si el curso i es asignado o no al periodo j:

$$\bigvee_{k=0}^{1} x_{i,j} = k_2; \quad \forall i \in \{1, ..., M\}, \quad \forall j \in \{1, ..., N\}$$

2. **Definición de la carga máxima:** C se utiliza en la función objetivo como $C = max\{c_1, ..., c_n\}$, lo cual se logra representar mezclando la minimización de C junto a la siguiente inecuación:

$$C \geq c_j; \quad \forall j \in \{1, ..., N\}$$

3. **Cálculo de la carga académica por periodo:** La carga académica para el periodo j se define como:

$$c_j = \sum_{i=1}^{M} \left(\alpha_i^* \times x_{i,j}\right); \quad \forall j \in \{1, ..., N\}$$

4. **Asignación de curso por periodo:** Cada curso i debe ser asignados a un periodo j:

$$\sum_{j=1}^{N} x_{i,j} = 1_2; \quad \forall i \in \{1, ..., M\}$$

5. **Prerrequisitos de cursos:** si el curso b requiere como prerrequisito al curso a, entonces b no puede ser asignado a un periodo anterior al de a:

$$x_{b,j} \leq \sum_{r=1}^{j-1} x_{a,r} = 1_2; \quad \forall j \in \{2, ..., N\}, \quad \forall a \in \{1, ..., M-1\}, \quad \forall b \in \{a, ..., M\}$$

6. **Carga mínima por periodo:** La carga académica de cada periodo j debe ser mayor o igual a la carga mínima permitida:

$$c_j \geq \beta^*; \quad \forall j \in \{1, ..., N\}$$

7. **Carga máxima por periodo:** La carga académica de cada periodo j debe ser menor o igual a la carga máxima permitida:

$$c_j \leq \gamma^*; \quad \forall j \in \{1, ..., N\}$$

8. **Mínima cantidad de cursos por periodo:** El número de cursos del periodo j debe ser mayor o igual a la cantidad mínima permitida:

$$\sum_{i=1}^{M} x_{i,j} \geq \delta^*; \quad \forall j \in \{1, ..., N\}$$

9. **Máxima cantidad de cursos por periodo:** El número de cursos del periodo j debe ser menor o igual que la cantidad máxima permitida:

$$\sum_{i=1}^{M} x_{i,j} \leq \epsilon^*; \quad \forall j \in \{1, ..., N\}$$

Transformación de `LIA` a `QF_ALIA`

Parámetros:

- M : Número de cursos, $M \in \mathbb{Z}^+$
- N : Número de periodos académicos, $N \in \mathbb{Z}^+$
- α_i : Número de créditos del curso i, $\alpha_i \in \mathbb{Z}^+$, $\forall i \in \mathbb{Z}^+ : i \in \{1, ..., M\}$
- β : Carga académica **mínima** permitida por periodo, $\beta \in \mathbb{Z}^+$
- γ : Carga académica **máxima** permitida por periodo, $\gamma \in \mathbb{Z}^+$
- δ : Cantidad de cursos **mínima** por periodo, $\delta \in \mathbb{Z}^+$
- ϵ : Cantidad de cursos **máxima** por periodo, $\epsilon \in \mathbb{Z}^+$

Variables:

- x: Arreglo bidimensional de enteros, donde $x[i][j] = 1$ si el curso i es asignado al periodo j, y 0 en caso contrario; $x \in \mathcal{A}_I^E, \quad \forall I \in \mathbb{Z}, \forall E \in \mathcal{A}_{I^*}^{E^*}, \forall I^*, E^* \in \mathbb{Z}$
- c: Arreglo unidimensional de enteros, define la carga académica de cada periodo, $c \in \mathcal{A}_I^E, \forall I, E \in \mathbb{Z}$
- C: Arreglo unidimensional de enteros, representa la mayor carga académica entre todos los periodos del curriculum, $C \in \mathcal{A}_I^E, \forall I, E \in \mathbb{Z}$

Función objetivo:

- Minimizar la mayor carga académica C entre todos los periodos, buscando un currículum equilibrado:

$$min \quad C = max\{c[1], ..., c[N]\}$$

Restricciones del problema:

1. **Dominio de las variables:** Cada variable $x[i][j]$ solo puede tomar los valores 0 o 1, indicando si el curso i es asignado o no al periodo j:

$$\bigvee_{k=0}^{1} x[i][j] = k; \quad \forall i \in \{1, ..., M\}, \quad \forall j \in \{1, ..., N\}$$

2. **Definición de la carga máxima:** C se utiliza en la función objetivo como $C = max\{c[1], ..., c[N]\}$, lo cual se logra representar mezclando la minimización de C junto a la siguiente inecuación:

$$C[0] \geq c[j]; \quad \forall j \in \{1, ..., N\}$$

3. **Cálculo de la carga académica por periodo:** La carga académica para el periodo j se define como:

$$c[j] = \sum_{i=1}^{M} (\alpha_i \times x[i][j]); \quad \forall j \in \{1, ..., N\}$$

4. **Asignación de curso por periodo:** Cada curso i debe ser asignados a un periodo j:

$$\sum_{j=1}^{N} x[i][j] = 1; \quad \forall i \in \{1, ..., M\}$$

5. **Prerrequisitos de cursos:** si el curso b requiere como prerrequisito al curso a, entonces b no puede ser asignado a un periodo anterior al de a:

$$x[b][j] \leq \sum_{r=1}^{j-1} x[a][r] = 1; \quad \forall j \in \{2, ..., N\}, \quad \forall a \in \{1, ..., M-1\}, \quad \forall b \in \{a, ..., M\}$$

6. **Carga mínima por periodo:** La carga académica de cada periodo j debe ser mayor o igual a la carga mínima permitida:

$$c[j] \geq \beta; \quad \forall j \in \{1, ..., N\}$$

7. **Carga máxima por periodo:** La carga académica de cada periodo j debe ser menor o igual a la carga máxima permitida:

$$c[j] \leq \gamma; \quad \forall j \in \{1, ..., N\}$$

8. **Mínima cantidad de cursos por periodo:** El número de cursos del periodo j debe ser mayor o igual a la cantidad mínima permitida:

$$\sum_{i=1}^{M} x[i][j] \geq \delta; \quad \forall j \in \{1, ..., N\}$$

9. **Máxima cantidad de cursos por periodo:** El número de cursos del periodo j debe ser menor o igual que la cantidad máxima permitida:

$$\sum_{i=1}^{M} x[i][j] \leq \epsilon; \quad \forall j \in \{1, ..., N\}$$

A.2. Desempeño promedio por problema

A continuación se presentan las tablas que contienen los desempeños promedios de las combinaciones de lógicas usadas para resolver todas las instancias de cada problema usando máximo de ejecución de 3600 segundos y un límite de memoria de 6144 *MB*. Estos promedios son calculados directamente a partir de las tablas registradas en el directorio `Evaluación mejores combinaciones` del repositorio de resultados [175].

Cada tabla corresponde a un problema, y contienen una fila por cada combinación de lógicas resultante del filtrado. Con ello se busca evaluar el desempeño de cada combinación a nivel general del problema.

La columna **Tiempo Promedio (s)** reporta la cantidad de segundos utilizados en promedio por el solver para resolver todas las instancias del problema usando la combinación indicada, mientras que la columna **Memoria Promedio (MB)** indica la cantidad de memoria *RAM* consumida en promedio durante la ejecución.

La columna **Tasa de Éxito** indica la razón entre la cantidad de instancias resueltas exitosamente y la cantidad total de instancias del problema. Un valor de 0 indica que la combinación elegida no logró resolver ninguna instancia, mientras que un valor de 1 indica que la combinación logró resolver todas las instancias.

Combinación de lógicas	Tiempo Promedio (s)	Memoria Promedio (MB)	Tasa de Éxito
lra-lia-lra	2100.61	412.17	0.42
lra-lra-lra	2063.66	236.53	0.5
lra-lia-lia	1878.19	343.25	0.5
bv-bv-bv	1842.7	1613.95	0.5
alia-alia-lra	1370.65	575.69	0.67
alia-lia-lia	1032.14	398.91	0.83
lia-alia-alia	1024.74	380.85	0.83
lia-lia-alia	1022.51	389.3	0.83
alia-lia-alia	994.96	389.58	0.83
alia-alia-alia	976.75	379.04	0.83
lia-lia-lia	38.84	335.28	1.0

Tabla A.1:

Desempeño promedio de las mejores combinaciones encontradas para el problema *N-QUEENS*

Combinación de lógicas	Tiempo Promedio (s)	Memoria Promedio (MB)	Tasa de Éxito
bv-bv-bv	3600.01	1813.78	0.0
lra-lra-lra	3005.98	5114.34	0.2
lra-alia-alia	617.26	161.57	0.9
lia-lia-lia	581.38	631.64	1.0
lia-lia-alia	12.82	138.79	1.0
alia-alia-lra	7.97	112.09	1.0
alia-lia-alia	7.97	116.12	1.0
alia-alia-lia	7.92	112.05	1.0
alia-lra-lia	7.9	115.64	1.0
alia-lra-alia	7.88	115.63	1.0
alia-alia-alia	7.4	112.04	1.0

Tabla A.2: Desempeño promedio de las mejores combinaciones encontradas para el problema *UKP*

Combinación de lógicas	Tiempo Promedio (s)	Memoria Promedio (MB)	Tasa de Éxito
lia-lia-lia-lia-lia-lia-lia-lia-lia-lia	3600.01	4356.23	0.0
bv-bv-bv-bv-bv-bv-bv-bv-bv-bv	3600.01	1561.44	0.0
lra-lra-lra-lra-lra-lra-lra-lra-lra-lra	3600.01	4185.83	0.0
lia-lia-alia-alia-lia-lra-lia-lia-lia	2481.67	1865.59	0.33
lia-lia-lia-lia-lia-lra-lia-alia-lia	2429.44	1799.65	0.33
alia-alia-alia-lia-alia-lia-alia-alia-alia	1890.11	749.13	0.5
alia-alia-alia-alia-alia-alia-alia-alia-alia-alia	1886.54	747.54	0.5

Tabla A.3: Desempeño promedio de las mejores combinaciones encontradas para el problema *NSP*

Combinación de lógicas	Tiempo Promedio (s)	Memoria Promedio (MB)	Tasa de Éxito
bv-bv-bv-bv-bv-bv-bv-bv	1382.39	201.61	0.59
lia-lia-lia-lia-lia-lia-lia-lia	1121.39	418.47	0.67
lra-lra-lia-lra-lra-lra-lra-lra	1074.83	407.09	0.67
lra-lia-lra-lra-lra-lra-lra-lra	1073.15	407.29	0.67
lra-lra-lra-lra-lra-lra-lra-lra	1061.6	408.65	0.67
lra-lia-lia-lra-lia-lra-lia-lra	1030.81	408.76	0.7
lia-lra-lia-lia-lra-lia-lia-lia	1009.38	414.94	0.7
lia-lra-lia-lra-lra-lia-lia-lra	1006.91	418.84	0.7

Tabla A.4:

Desempeño promedio de las mejores combinaciones encontradas para el problema *TSP* (Parte 1)

Combinación de lógicas	Tiempo Promedio (s)	Memoria Promedio (MB)	Tasa de Éxito
lra-lra-lra-lra-lra-lia-lra	1039.53	410.08	0.74
lia-lia-lia-lra-lia-lia-lia	1012.32	423.81	0.74
lia-lia-lia-lia-lia-lia-lra	1007.42	422.01	0.74
lia-alia-lia-lia-lia-bv-lia	698.05	274.34	0.78
alia-lia-lia-lia-lia-lia-lia	634.13	123.6	0.78
lra-lra-lra-lia-lra-lra-alia	617.47	114.18	0.78
lra-alia-lra-alia-lia-lra-alia	605.51	94.91	0.78
lra-alia-lia-alia-lia-lra-alia	605.39	94.93	0.78

Tabla A.5:

Desempeño promedio de las mejores combinaciones encontradas para el problema *TSP* (Parte 2)

Combinación de lógicas	Tiempo Promedio (s)	Memoria Promedio (MB)	Tasa de Éxito
lra-bv-lia-lra-alia-lra-lra	617.97	323.07	0.81
lia-lra-lia-lia-lia-alia-alia	585.54	103.94	0.81
lia-alia-lra-lia-alia-lra-lra	577.98	107.59	0.81
alia-alia-alia-alia-alia-alia-alia	576.64	98.81	0.81
alia-alia-alia-lia-alia-alia-lia	564.05	94.79	0.81
alia-alia-lra-alia-alia-alia-lia	557.7	93.63	0.81
alia-alia-lra-alia-alia-alia-alia	550.83	100.46	0.81
lia-lra-alia-lra-alia-lra-lra	531.04	112.19	0.81
alia-lra-alia-lra-lra-alia-lra	495.92	108.06	0.81

Tabla A.6:

Desempeño promedio de las mejores combinaciones encontradas para el problema *TSP* (Parte 3)

Combinación de lógicas	Tiempo Promedio (s)	Memoria Promedio (MB)	Tasa de Éxito
bv-bv-bv-bv-bv-bv-bv-bv-bv-bv	3600.01	240.57	0.0
lra-lra-lra-lra-lra-lra-lra-lra-lra-lra	2456.12	532.49	0.33
lia-lia-lia-lia-lia-lia-lia-lia-lia-lia	2410.33	623.57	0.33
lra-lra-lra-alia-lra-lra-lra-lia-alia	2408.15	101.06	0.33
alia-alia-alia-alia-alia-alia-alia-alia-alia-alia	30.13	83.51	1.0
alia-lia-lra-lra-lia-lia-lia-lra-lra	13.24	82.53	1.0
alia-alia-lia-alia-alia-alia-alia-alia-alia-alia	12.75	79.51	1.0

Tabla A.7:

Desempeño promedio de las mejores combinaciones encontradas para el problema *BACP*