UNIVERSIDAD DE GRANADA

Facultad de Ciencias y Escuela Técnica Superior de Ingenierías Informática y de Telecomunicaciones

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO DE FIN DE GRADO

Algoritmos para la aproximación de un conjunto a partir de otros. Caracterización matemática del problema y estudio experimental.

Presentado por:
Laura Lázaro Soraluce

Curso académico 2025-2026

Algoritmos para la aproximación de un conjunto a partir de otros. Caracterización matemática del problema y estudio experimental.

Laura Lázaro Soraluce

Laura Lázaro Soraluce *Algoritmos para la aproximación de un conjunto a partir de otros. Caracterización matemática del problema y estudio experimental.*.
Trabajo de fin de Grado. Curso académico 2025-2026.

Responsable de tutorización

Nicolás Marín Ruiz
Ciencias de la Computación e Inteligencia Artificial

Daniel Sánchez Fernández
Ciencias de la Computación e Inteligencia Artificial

Doble Grado en Ingeniería Informática y Matemáticas

Facultad de Ciencias y Escuela Técnica Superior de Ingenierías Informática y de Telecomunicaciones

Universidad de Granada

Declaración de originalidad

Dña. Laura Lázaro Soraluce

Declaro explícitamente que el trabajo presentado como Trabajo de Fin de Grado (TFG), correspondiente al curso académico 2025-2026, es original, entendido esto en el sentido de que no he utilizado para la elaboración del trabajo fuentes sin citarlas debidamente.

En Granada a 25 de noviembre de 2025

Fdo: Laura Lázaro Soraluce

A mamá, papá e Iker.

Agradecimientos

A mi padre, por cada cena que convierte en un viaje por teoremas, paradojas y preguntas sin respuesta. A mi madre, que, aunque el mundo de los números no sea el suyo, nunca ha dejado de acompañarme en el mío. Y a mi hermano, cuya curiosidad y criterio admiro más de lo que digo. Gracias por construir siempre un lugar al que querer volver, un espacio donde pensar en voz alta, debatir sin miedo y reír sin reservas.

A mis tutores, Daniel y Nicolás, por su tiempo, su paciencia, y su orientación y apoyo constantes a lo largo de este trabajo. Y en especial por ser, antes que tutores, excelentes profesores, capaces de equilibrar la transmisión de conocimiento, el respeto y la cercanía.

A Carmen y María, las mejores compañeras de carrera —amigas— con quienes podría haber compartido estos años. Me han dado risa, desahogo y empujón en los momentos más necesarios.

A Cristóbal, Manu, Adri, Lucía y Álvaro, por ser parte del paisaje de esta carrera. Me han hecho el camino más amable.

Por último, a mí misma. Por el trabajo constante y la perseverancia que he mantenido en cada etapa. La satisfacción de hoy también viene de lo que he sido capaz de volcar en este camino: cabeza, corazón y fondo.

Summary

The problem of selecting the "best" approximation for a dataset is a fundamental challenge in computational science. In many practical domains, such an approximation must not only be accurate, but also simple, interpretable, and computationally not expensive. This Final Degree Project focuses on formalizing this problem as set approximation by expressions.

The general objective of this work has been, on the one hand, to formalize this set approximation problem; and on the other, to design, implement, and validate an experimental environment to solve it and compare different algorithmic approaches. We have divided the specific objectives into a theoretical-mathematical block and a practical-computational block:

- **Theoretical-mathematical block**: establishing a formal characterization of the problem, studying its relationship with classic problems, its algebraic properties, the different evaluation measures, and studying its theoretical complexity.
- **Practical-computational block**: defining algorithmic approaches to the problem (exhaustive search with limited depth, Greedy-MO, and NSGA-II genetic algorithm), studying their complexity, designing a reproducible experimental framework, and illustrating the correct functioning of the algorithms through different experiments.

The initial hypotheses regarding the algorithms' behavior were that the exhaustive method would be computationally too expensive to compute, the Greedy-MO heuristic would be fast but would get trapped in local optima, and the NSGA-II metaheuristic would be significantly more robust.

The theoretical framework has two main points. On the one hand, we have treated the problem as a multi-objective optimization problem. Instead of a single solution, we seek a Pareto front that offers a balance between the different objectives. Secondly, we address the computational complexity analysis, where the main theoretical result is that this problem is NP-Complete. For its demonstration, we use the fact that our problem is a generalization of others, such as SAT or Set Cover. This theoretical limit justifies why the use of heuristics and approximations is necessary.

We focused on implementing and comparing three algorithms. Given that the search space is infinite, all of them operate on a subspace bounded by a maximum depth k, that is, they only explore expressions with $\leq k$ operators.

1. **Exhaustive Search with limited depth:** it is an exact algorithm that deterministically explores the bounded subspace. It guarantees finding the true optimal Pareto front of the said subspace. Due to its high complexity, its purpose in this work is to serve as a validation baseline in very small instances ($k = 3$).
2. **Greedy-MO:** it is a deterministic and constructive heuristic. It generates solutions sequentially (adding a pair (op, F_i) in each step), maintaining only a front of non-

dominated solutions at each level. It is designed to be extremely fast, but its nature makes it prone to getting trapped in local optima.

3. **NSGA-II (Genetic):** it is a stochastic metaheuristic selected for being a benchmark in multi-objective optimization problems. It maintains a population of solutions that evolves through tournament selection, crossover, and mutation. It uses non-dominance sorting and crowding distance to guide the search towards a global and diverse Pareto front.

To support these algorithms, we implemented a reproducible experimental laboratory with a modular architecture and a centralized configuration framework. The scripts are responsible for managing compilation, the iterative execution of experiments per seed, and saving the results, ensuring traceability.

The experimental results illustrated the behavior of the algorithms, aligning with the hypotheses in these experiments specifically. In Experiment 1 on small instances, NSGA-II was able to replicate the true optimal Pareto front found by the exhaustive search with limited depth, whereas Greedy-MO got trapped in a local optimum. Experiment 2, on instances with at least one expression reaching a Jaccard index$= 1.0$, measured robustness. The NSGA-II proved to be significantly more robust than Greedy-MO in finding at least one expression that also reached Jaccard$= 1.0$. Experiment 3, without any expression as a reference, was the main comparison. The NSGA-II demonstrated clear superiority over Greedy-MO, which we proved to be statistically significant using a Wilcoxon Signed-Rank Test.

The main conclusion is that the NSGA-II metaheuristic is a significantly more robust and effective solution than the Greedy heuristic for this NP-Complete problem. The trade-off has been confirmed: Greedy is instantaneous but limited to simple solutions, whereas NSGA-II uses its time to explore more complex solutions that result in much higher quality.

The contribution of this Final Degree Project is, on the one hand, the theoretical formalization of the set approximation by expressions problem and the study of its properties from different mathematical areas; and on the other hand, the construction of a reproducible software environment that implements, compares, and validates the chosen approaches for this problem using different algorithms.

Future work focuses on new experiments (such as an analysis of the time needed for NSGA-II to dominate the Greedy algorithm), algorithmic improvements (such as an exhaustive analysis to choose the parameters of NSGA-II), and on adding new measures to compare.

Resumen

Los problemas de seleccionar la "mejor" aproximación para un conjunto de datos son un desafío fundamental en la ciencia computacional. En gran cantidad de dominios prácticos, dicha aproximación no solo debe ser precisa, sino también simple, interpretable y barata de calcular. Este Trabajo de Fin de Grado se centra en dicho desafío formalizando un problema de la aproximación de conjuntos mediante expresiones.

El objetivo general de este trabajo ha sido, por un lado, formalizar este problema de aproximación de conjuntos; y por otro, diseñar, implementar y validar un entorno experimental para resolverlo y comparar distintas aproximaciones algorítmicas. Dividimos los objetivos específicos en un bloque teórico-matemático y uno práctico-computacional:

- **Bloque teórico-matemático**: establecer una caracterización formal del problema, estudiando su relación con problemas clásicos, sus propiedades algebraicas, las distintas medidas de evaluación y, estudiar su complejidad teórica.
- **Bloque práctico-computacional**: definir aproximaciones algorítmicas del problema (búsqueda exhaustiva con profundidad limitada, Greedy-MO y algoritmo genético NSGA-II), estudiar su complejidad, diseñar un marco experimental reproducible, e ilustrar el correcto funcionamiento de los algoritmos a través de distintos experimentos.

Las hipótesis de partida del comportamiento de los algoritmos eran que el método exhaustivo sería computacionalmente inviable, la heurística Greedy-MO sería rápida pero quedaría atrapada en óptimos locales, y la metaheurística NSGA-II sería significativamente más robusta.

El marco teórico tiene dos puntos principales. Por un lado, hemos tratado el problema como un problema de optimización multiobjetivo. En lugar de una única solución, buscamos un frente de Pareto que ofrece un equilibrio entre los distintos objetivos. En segundo lugar, abordamos el análisis de complejidad computacional, donde el hallazgo teórico central es que este problema es NP-Completo. Para su demostración hemos utilizado que nuestro problema es una generalización de otros conocidos, como el problema SAT o el *Set Cover*. Esta intratabilidad teórica justifica formalmente el por qué del uso de heurísticas y aproximaciones.

Nos centramos en implementar y comparar tres algoritmos. Dado que el espacio de búsqueda $\mathcal{E}$ es infinito, todos ellos operan sobre un subespacio acotado por una profundidad máxima k, es decir, exploran únicamente las expresiones con $\leq k$ operadores.

1. **Búsqueda Exhaustiva con profundidad limitada:** es un algoritmo exacto que explora de forma determinista el subespacio acotado. Garantiza encontrar el frente de Pareto óptimo real de dicho subespacio. Debido a su alta complejidad, su único propósito en este trabajo es servir como referencia en instancias muy pequeñas.

2. **Greedy-MO:** es una heurística determinista y constructiva. Genera soluciones añadiendo un par (op, F_i) en cada paso, manteniendo en cada nivel únicamente un frente de soluciones no dominadas. Está diseñado para ser extremadamente rápido, pero su naturaleza lo expone a quedar atrapado en óptimos locales.
3. **NSGA-II (Genético):** es una metaheurística estocástica, seleccionada por ser una referencia en problemas optimización multiobjetivo. Mantiene una población de soluciones que evoluciona mediante selección por torneo, cruce y mutación. Utiliza la clasificación por no-dominancia y la distancia de aglomeración para guiar la búsqueda hacia un frente de Pareto global y diverso.

Para soportar estos algoritmos, implementamos un laboratorio experimental reproducible con una arquitectura modular y un archivo que centraliza todos los parámetros. Los *scripts* desarrollados se encargan de gestionar la compilación, la ejecución iterativa por semilla de los experimentos, y de guardar los resultados, garantizando la trazabilidad.

Los resultados experimentales sirvieron para ilustrar el comportamiento de los algoritmos, corroborando las hipótesis en los siguientes experimentos concretos. En el Experimento 1 sobre instancias pequeñas, el NSGA-II fue capaz de replicar el frente de Pareto óptimo real encontrado por la búsqueda exhaustiva con profundidad limitada, mientras que el Greedy-MO quedó atrapado en óptimos locales. El Experimento 2, sobre instancias con al menos una expresión que alcanza el índice de Jaccard$= 1.0$, sirvió para medir la robustez. El NSGA-II demostró ser significativamente más robusto que el Greedy-MO, encontrando al menos una expresión que también alcanza indice de Jaccard$= 1.0$. El Experimento 3, sin ninguna expresión de referencia, fue la comparativa principal. El NSGA-II demostró una superioridad clara con respecto al Greedy, que demostramos estadísticamente significativa a través de un Test de los Rangos Signados de Wilcoxon.

La conclusión principal es que la metaheurística NSGA-II es una solución significativamente más robusta y eficaz que la heurística Greedy para este problema NP-Completo. Se ha confirmado el balance entre coste y calidad: el Greedy es instantáneo pero se limita a soluciones simples, mientras que el NSGA-II usa su tiempo para explorar soluciones más complejas que resultan en una calidad muy superior.

La aportación de este Trabajo de Fin de Grado es por un lado la formalización teórica del problema de aproximación de conjuntos mediante expresiones y el estudio de sus propiedades desde diferentes áreas matemáticas; y por otro, la construcción de un entorno de software reproducible que implementa, compara y valida las aproximaciones escogidas para dicho problema mediante distintos algoritmos.

Las líneas futuras se centran en nuevos experimentos (como un análisis del tiempo necesario para que el NSGA-II domine al algoritmo Greedy), mejoras algorítmicas (como un análisis exhaustivo para elegir los hiperparámetros del algoritmo genético), y en añadir nuevas medidas para comparar.

Índice general

Introducción

El problema de seleccionar una "buena" aproximación para un conjunto de datos aparece en numerosos campos, desde la logística (optimizar rutas de camiones de reparto) y las redes de comunicación (colocar antenas que cubran un área geográfica), hasta la biología computacional o el análisis de negocio. En todos estos contextos, una aproximación útil puede tener varias propiedades que optimizar, como la simplicidad o la cobertura del conjunto objetivo. En este Trabajo de Fin de Grado estudiamos precisamente este problema, formalizándolo como aproximación de conjuntos mediante expresiones. Este consiste en construir, sobre un universo, una expresión a partir de una familia base de conjuntos y un conjunto de operadores, cuya evaluación se aproxime a un conjunto objetivo.

Nuestro proyecto aborda este problema desde una perspectiva de optimización multiobjetivo, en la que no existe una única solución óptima, sino un conjunto de compromisos representados mediante un frente de Pareto. La literatura reciente destaca la utilidad de las metaheurísticas en problemas donde coexisten objetivos (en nuestro caso, la calidad de la aproximación y la complejidad de la expresión). Además, el interés de nuestro problema es añadido, pues es una generalización de problemas clásicos como Set Cover o Exact Cover, lo que sugiere una dificultad computacional elevada y motiva la elección de algoritmos aproximados.

Objetivos del trabajo

El objetivo principal de este Trabajo de Fin de Grado ha sido doble: primero, formalizar y estudiar desde el punto de vista matemático este problema de aproximación de conjuntos mediante expresiones; y segundo, diseñar, implementar y validar un entorno experimental para comparar distintas aproximaciones algorítmicas en su resolución. Para ello, hemos dividido los objetivos generales en dos bloques principales:

- **Bloque teórico-matemático**:
  - Establecer una caracterización formal del problema de aproximación de conjuntos así como sus elementos fundamentales.
  - Estudiar la relación de nuestro problema con otros problemas clásicos de la literatura matemática.
  - Analizar las propiedades y estructuras algebraicas del universo, la familia de subconjuntos y las operaciones disponibles.
  - Estudiar distintas medidas como herramientas para definir restricciones y evaluar la calidad de las soluciones.
  - Explorar la complejidad computacional del problema, y su pertenencia a clases de complejidad conocidas, a través de su posible relación con problemas NP-completos.

- **Bloque práctico-computacional**:
  - Definir distintas aproximaciones algorítmicas del problema, adaptando la formulación teórica a variantes que permitan su resolución práctica.
  - Analizar diversos algoritmos potencialmente aplicables y justificar la selección de aquellos que resulten más adecuados para su estudio.
  - Estudiar la complejidad computacional de los algoritmos escogidos.
  - Diseñar e implementar un marco experimental reproducible que integre la generación automática de instancias, la ejecución controlada de los algoritmos seleccionados (búsqueda exhaustiva con profundidad limitada, Greedy multiobjetivo y genético NSGA-II) y el registro estructurado de resultados.
  - Validar el correcto funcionamiento de los algoritmos implementados mediante pruebas ilustrativas, y demostrar la capacidad para construir un entorno computacional completo y extensible.

Dado que el espacio de búsqueda de todas las expresiones $\mathcal{E}$ es infinito (se pueden anidar operaciones indefinidamente), el primer paso para cualquier algoritmo de búsqueda es acotarlo. Por ello, nos centramos en un subespacio, definido por una profundidad máxima k. Estudiamos, no solo un método exhaustivo como la búsqueda exhaustiva con profundidad limitada, sino también una heurística Greedy-MO y una metaheurística NSGA-II, para enriquecer la comparativa y proporcionar un punto de referencia intermedio entre la velocidad pura y la calidad de la solución.

Para alcanzar estos objetivos, aplicamos conocimientos adquiridos en distintas asignaturas cursadas a lo largo del doble grado en ingeniería informática y matemáticas, como Modelos Avanzados de Computación (para el análisis de NP-Completitud), Metaheurísticas (para el diseño del NSGA-II), Algorítmica (para el análisis de complejidad y el Greedy), Álgebra (para la fundamentación del problema) o Inferencia Estadística (para realizar tests sobre los resultados).

Estructura de la memoria

La memoria tiene la siguiente estructura:

- El Capítulo 1 define y formaliza el problema y todos sus elementos, así como algunas particularizaciones.
- El Capítulo 2 establece el marco teórico-matemático, define el problema y analiza su complejidad.
- El Capítulo 3 describe la metodología algorítmica, detallando la implementación y el análisis de complejidad de los tres algoritmos. Además, presenta el diseño del laboratorio experimental, la arquitectura del software, los protocolos de los experimentos y el análisis detallado de los resultados.
- El Capítulo 4 cierra el trabajo, resumiendo los hallazgos, realizando una valoración personal y proponiendo vías futuras de investigación.

- Finalmente, el documento incluye varios anexos con información complementaria (la planificación temporal y el presupuesto económico del proyecto, datos adicionales de los experimentos, y la guía de reproducibilidad y el repositorio de código).

1. Introducción al problema

En este trabajo abordamos el problema de aproximar un conjunto de elementos partiendo de una colección de subconjuntos de su mismo universo. La idea principal consiste en utilizar aproximaciones algorítmicas para seleccionar y combinar algunos de estos subconjuntos mediante ciertas operaciones, respetando una serie de restricciones impuestas por el usuario y optimizando un conjunto de medidas, previamente acordados. Entre las restricciones podemos incluir, por ejemplo, un número máximo de operaciones utilizadas o un tamaño mínimo requerido para la solución. En cuanto a las medidas, estas pueden estar relacionadas con la sobrecobertura, la infracobertura o la redundancia entre conjuntos, entre otras.

Como veremos, aunque sea deseable, no siempre va a ser posible obtener una coincidencia exacta con el conjunto objetivo. En muchos casos puede ser suficiente con obtener una aproximación que cumpla con los requisitos. Por ello, formulamos el problema de forma flexible y general, y lo iremos particularizando a lo largo del trabajo, estudiando distintas variantes y aspectos concretos según el contexto.

1.1. Definición formal del problema

Una vez hemos introducido el problema de forma global y los objetivos del trabajo en la sección de Introducción, comenzamos centrándonos en el primero de ellos: su formulación general. Queremos definir un marco amplio que englobe todas las particularizaciones que analizaremos a lo largo del trabajo. Para ello, comenzamos enumerando los elementos principales de nuestro problema:

1. **Universo** U : conjunto finito de elementos, con $|U| = n$. Convenimos sin pérdida de generalidad que $U \neq \emptyset$.

2. **Conjunto de operaciones** $\mathcal{O}$: el conjunto formado por las operaciones binarias sobre conjuntos bien conocidas de unión, intersección y diferencia:

   $$\mathcal{O} = \{\cup, \cap, \setminus\}.$$

   No incluimos la operación de complemento, ya que hemos optado por considerar únicamente operaciones binarias. Además, la operación de complemento puede expresarse como una diferencia, sin aumentar el número de operaciones necesarias. Eliminar otra operación como la intersección no resultaría conveniente, pues reproducir su efecto mediante diferencias requeriría al menos una operación adicional y aumentaría la complejidad de las expresiones resultantes, lo cual no siempre es deseable.

Definición 1.1 (Familia generadora)**.** Sea U el conjunto universo. Llamamos familia generadora al conjunto $F \subseteq \mathcal{P}(U)$ (donde $\mathcal{P}(U)$ denota el conjunto de las partes de U), tal que $U \in F$. Denotamos por $|F| = m$ el número de subconjuntos de la familia y por $n_i = |F_i|$ el cardinal de cada subconjunto $F_i \in F$.
Cuando no haya lugar a confusión, nos referiremos a la familia generadora simplemente como familia F.

F es la familia de conjuntos que podemos utilizar para construir la aproximación que buscamos.

Definición 1.2 (Expresión)**.** Dado un universo U, llamamos expresión a toda combinación finita bien formada obtenida a partir de elementos de subconjuntos de $\mathcal{P}(U)$ mediante las operaciones de $\mathcal{O}$. Denotamos por $\mathcal{E}$ al conjunto de todas las expresiones posibles sobre $\mathcal{P}(U)$.

Cabe señalar de esta definición que las expresiones pueden utilizar paréntesis para alterar el orden de evaluación. Al evaluar una expresión, las subexpresiones contenidas entre paréntesis tienen la máxima prioridad y se resuelven primero, comenzando desde los pares más anidados hacia los más externos. Si una expresión carece de paréntesis para indicar un orden, se aplica la regla de evaluación por defecto, que es la resolución de izquierda a derecha. Es importante destacar que toda expresión puede siempre representarse mediante una notación en la que cada subexpresión con operaciones está delimitada por paréntesis.

Definimos las dos funciones siguientes sobre $\mathcal{E}$, que nos permitirán extraer información sobre la forma en que se construyeron sus expresiones:

Definición 1.3 (Función de subconjuntos utilizados)**.** Definimos la aplicación

$$\mathcal{F} : \mathcal{E} \longrightarrow \mathcal{P}(\mathcal{P}(U))$$

que asocia a cada expresión $e \in \mathcal{E}$ la colección de subconjuntos de U que aparecen en e.

Definición 1.4 (Función de operaciones utilizadas)**.** Definimos la aplicación

$$\mathcal{O}p : \mathcal{E} \longrightarrow \mathcal{P}(\mathcal{O})$$

que asocia a cada expresión $e \in \mathcal{E}$ el conjunto de operaciones de $\mathcal{O}$ que aparecen en e.

Aprovechando estas funciones, podemos hacer la siguiente definición:

Definición 1.5 (Espacio de expresiones generadas por una familia F)**.** Sea U el conjunto universo y $\mathcal{E}$ el conjunto de expresiones asociado a U según la definición 1.2. Dada una familia $F \subseteq \mathcal{P}(U)$, definimos el espacio de expresiones generadas por F como

$$\mathcal{E}_F = \{e \in \mathcal{E}; \quad \mathcal{F}(e) \subseteq F\}$$

Es decir las expresiones formadas a partir de conjuntos de F.

Adicionalmente, definimos una función que asocia a cada expresión su resultado:

Definición 1.6 (Función de evaluación)**.** La función de evaluación

$$\text{eval} : \mathcal{E} \longrightarrow \mathcal{P}(U)$$

es aquella que asigna a cada expresión $e \in \mathcal{E}$ el subconjunto de U que resulta de aplicar las operaciones de $\mathcal{O}$ en la forma indicada por e.

Denotamos por $\mathcal{R} = \mathit{eval}(\mathcal{E})$ a la imagen de $\mathcal{E}$ por esta función, es decir, la familia de todos los conjuntos que producen las expresiones de $\mathcal{E}$. Es importante tener en cuenta que, debido a las propiedades del Álgebra de Boole, distintas expresiones pueden dar lugar al mismo resultado, por lo que no podemos asegurar que la aplicación *eval* sea inyectiva [Men70].

Pasamos ahora a ver dos conceptos que nos van a permitir delimitar el espacio de expresiones de nuestro interés y calibrar la calidad de una expresión:

Definición 1.7 (Restricción)**.** Una restricción C es un predicado definido sobre el conjunto de expresiones $\mathcal{E}$:

$$C : \mathcal{E} \to \{0, 1\}.$$

Decimos que una expresión $e \in \mathcal{E}$ satisface la restricción C si $C(e) = 1$.

Definición 1.8 (Conjunto de expresiones restringidas por $\mathcal{C}$)**.** Sea $\mathcal{C} = \{C_1, \ldots, C_r\}$ un conjunto finito de restricciones. Denotamos por

$$\mathcal{E}^{\mathcal{C}} = \{e \in \mathcal{E};\, C_i(e) = 1 \;\forall i \in \{1, \ldots, r\}\}$$

al conjunto de expresiones que satisfacen todas las restricciones de $\mathcal{C}$.

Denotaremos por $\mathcal{E}_F^{\mathcal{C}}$ al conjunto de expresiones de $\mathcal{E}_F$ que satisfacen todas las restricciones de $\mathcal{C}$, es decir,

$$\mathcal{E}_F^{\mathcal{C}} = \mathcal{E}_F \cap \mathcal{E}^{\mathcal{C}}.$$

En general, las restricciones pueden derivar de las necesidades o intereses del usuario, para modelar el problema de acuerdo con una aplicación práctica concreta; o pueden introducirse con el fin de adaptar la formulación del problema de manera que pueda resolverse posteriormente mediante un algoritmo específico. Además, aunque todas las restricciones son predicados sobre el conjunto de expresiones, es fácil ver que también pueden definirse restricciones que se refieran a $\mathcal{R} = \mathit{eval}(\mathcal{E})$. Por ejemplo, un tamaño máximo del conjunto aproximado, sobre la evaluación de la expresión de la solución.

Definición 1.9 (Medida)**.** Una medida es una función

$$M : \mathcal{E} \to \mathbb{R}$$

que asigna a cada expresión $e \in \mathcal{E}$ un valor numérico que cuantifica algún aspecto o propiedad de interés, determinado por la naturaleza de la propia medida.

De igual forma que las restricciones, aunque las medidas están definidas sobre $\mathcal{E}$, algunas podemos verlas también como funciones sobre su imagen por *eval*, es decir, sobre $\mathcal{R} = eval(\mathcal{E})$.

En la práctica, las medidas son herramientas que pueden ayudarnos a imponer las restricciones de $\mathcal{C}$, rechazando expresiones que no cumplan un umbral; o a cuantificar la calidad de una posible solución encontrada.

Más adelante, estudiaremos ejemplos concretos de medidas, entre las que veremos:

- Medidas de asociación
- Medidas direccionales
- Medidas de cobertura
- Medidas de redundancia
- Medidas de ruido
- Medidas de error global
- Medidas de tamaño
- Medidas ponderadas

Por último, antes de poder dar la definición formal de nuestro problema, necesitamos rescatar las definiciones de Dominancia de Pareto y de frente de Pareto que podemos encontrar en [ED18], y que adaptamos a los elementos de nuestro problema:

Definición 1.10 (Dominancia de Pareto en un espacio de expresiones $\mathcal{E}$ en base a una serie de medidas)**.** Sea $\mathcal{E}$ un espacio de expresiones y $\mathcal{M} = \{M_1, \ldots, M_s\}$ un conjunto de medidas definidas sobre $\mathcal{E}$. Sea

$$y(e) = (M_1(e), \ldots, M_s(e)) \in \mathbb{R}^s$$

el vector de valores de las medidas de $\mathcal{M}$ evaluadas en e.

Dadas $e_1, e_2 \in \mathcal{E}$ dos expresiones, decimos que e_1 domina de Pareto a e_2 si y solo si

$$\forall i \in \{1, \ldots, s\},\ y_i(e_1) \geq y_i(e_2) \qquad \text{y} \quad \exists j \in \{1, \ldots, s\};\ y_j(e_1) > y_j(e_2).$$

Es decir, la expresión e_1 domina a otra e_2 si e_1 no es peor en ninguna medida de $\mathcal{M}$ y es mejor en al menos una de ellas. Sin pérdida de generalidad, estamos suponiendo que queremos maximizar las medidas, suponiendo la adaptación trivial de las medidas en caso de que interese minimizarlas.

Definición 1.11 (Frente de Pareto inducido por un conjunto de medidas sobre un espacio de expresiones)**.** Sea $\mathcal{E}$ un espacio de expresiones y $\mathcal{M} = \{M_1, \ldots, M_s\}$ un conjunto de medidas definidas sobre él. El frente de Pareto asociado a $\mathcal{E}$ es el conjunto de expresiones no dominadas, es decir:

$$\mathcal{E}^{\mathcal{M}} = \{\, e \in \mathcal{E};\ \nexists e' \in \mathcal{E} \text{ tal que } e' \text{ domine de Pareto a } e \,\}.$$

Denotaremos por $\mathcal{E}_F^{\mathcal{M}}$ al frente de Pareto correspondiente al espacio $\mathcal{E}_F \subseteq \mathcal{E}$.

Antes de enunciar formalmente nuestro problema particular, consideramos primero la formulación general de un problema de optimización definido sobre un espacio de expresiones:

Definición 1.12 (Problema de optimización sobre un espacio de expresiones)**.** Dado un conjunto de expresiones $\mathcal{E}$, un conjunto de restricciones $\mathcal{C}$ definidas sobre él y un conjunto de medidas $\mathcal{M}$, el problema de optimización sobre expresiones consiste en determinar una expresión $e^* \in \mathcal{E}^{\mathcal{C}}$ que pertenezca al frente de Pareto asociado $\mathcal{E}^{\mathcal{M}}$, es decir,

$$e^* \in \mathcal{E}^{\mathcal{C}} \cap \mathcal{E}^{\mathcal{M}}.$$

Llamamos $\tilde{G} = eval(e^*)$ a la evaluación de dicha expresión por la función $eval$, y denotamos $(\mathcal{E}^{\mathcal{C}})^{\mathcal{M}}$ al frente de Pareto correspondiente a $\mathcal{E}^{\mathcal{C}}$.

Procedemos a dar una definición formal de nuestro problema. Para ello, consideramos un conjunto objetivo $G \subseteq U$, no vacío, que representa el conjunto que queremos aproximar mediante combinaciones de los elementos de la familia generadora F.

Definición 1.13 (Problema de aproximación de G mediante expresiones)**.** Dado un universo U, un espacio de expresiones $\mathcal{E}$, un conjunto objetivo $G \subseteq U$, un conjunto de restricciones $\mathcal{C}$ sobre $\mathcal{E}$ y un conjunto de medidas $\mathcal{M}$ sobre $\mathcal{E}$, planteamos el problema de aproximación de G como un caso particular del problema anterior en el que al menos una de las medidas de $\mathcal{M}$ determina algún aspecto de aproximación entre una expresión e y el conjunto objetivo G.

Dichas medidas pertenecen a las denominadas medidas de asociación, en las que profundizaremos en la Sección 2.2 más adelante.

El objetivo es determinar una expresión

$$e^* \in (\mathcal{E}^{\mathcal{C}})^{\mathcal{M}}.$$

Es decir, en el frente de Pareto asociado a $\mathcal{E}^{\mathcal{C}}$.

Antes de presentar el siguiente ejemplo sencillo, que servirá para visualizar todos estos conceptos que acabamos de definir, hacemos dos nuevas definiciones de funciones sobre $\mathcal{E}$:

Definición 1.14 (Función de secuencia de subconjuntos utilizados)**.** Definimos la aplicación

$$\mathcal{F}^* : \mathcal{E} \longrightarrow \bigcup_{n \geq 1} F^n$$

que asocia a cada expresión $e \in \mathcal{E}$ la n-upla de elementos de F que intervienen en ella, manteniendo el orden en el que aparecen y las posibles repeticiones.

Denotamos por $F^{(i)}(e)$ al elemento i-ésimo de la secuencia $\mathcal{F}^*(e)$.

Definición 1.15 (Función de secuencia de operaciones utilizados)**.** Definimos la aplicación

$$\mathcal{O}p^* : \mathcal{E} \longrightarrow \bigcup_{n\geq 1} \mathcal{O}^n$$

que asocia a cada expresión $e \in \mathcal{E}$ la n-upla de operaciones de $\mathcal{O}$ que intervienen en ella, manteniendo el orden en el que aparecen y las posibles repeticiones.

Denotamos por $O^{(i)}(e)$ al elemento i-ésimo de la secuencia $\mathcal{O}p^*(e)$.

Expresamos el codominio de ambas aplicaciones como una unión $\bigcup_{n\geq 1} F^n$ (análogo para $\mathcal{O}$) porque el número de subconjuntos u operaciones presentes en una expresión puede variar. De este modo, cada $\mathcal{F}^*(e)$ y $\mathcal{O}p^*(e)$ puede tener distinta longitud según la complejidad de la expresión.

Ejemplo 1.16. Ahora sí, planteamos el siguiente ejemplo sencillo:

- $U = \{1, 2, 3, 4, 5\}$.
- $F = \{\{1, 2\}, \{2, 3, 4\}\}$.
- $G = \{2, 4, 5\}$.
- $\mathcal{E}_F$: es el conjunto de todas las expresiones válidas que se pueden formar combinando subconjuntos de F (con posible repetición) mediante operaciones de unión, intersección y diferencia. Por ejemplo:
  - $\{1, 2\} \cup \{2, 3, 4\}$
  - $\{1, 2\} \setminus \{2, 3, 4\}$
- $eval : \mathcal{E}_F \to \mathcal{P}(U)$: función de evaluación que asigna a cada expresión su evaluación. Por ejemplo:

$$eval(\{1, 2\} \cup \{2, 3, 4\}) = \{1, 2, 3, 4\}, \quad eval(\{1, 2\} \setminus \{2, 3, 4\}) = \{1\}.$$

- $\mathcal{C} = \{C_1, C_2, C_3, C_4\}$: conjunto de restricciones. Donde:

$$C_1(e) : |\mathcal{O}p^*(e)| \leq 1$$
$$C_2(e) : \mathcal{O}p(e) \subseteq \{\cap\}$$
$$C_3(e) : \forall i \neq j,\ F^{(i)}(e) \neq F^{(j)}(e)$$
$$C_4(e) : \emptyset \notin \mathcal{F}(e)$$

  La tercera restricción es equivalente a decir que los elementos de $\mathcal{F}(e)$ aparecen sin repetición en e.

$\mathcal{E}_F^{\mathcal{C}}$ será el subconjunto de expresiones de $\mathcal{E}_F$ que cumplen dichas restricciones:

$$\mathcal{E}^{\mathcal{C}} = \{\{1,2,3,4,5\}, \{1,2\}, \{2,3,4\}, \{1,2\} \cap \{2,3,4\}, \{2,3,4\} \cap \{1,2\}, \\ \{1,2,3,4,5\} \cap \{1,2\}, \{1,2,3,4,5\} \cap \{2,3,4\}, \{1,2\} \cap \{1,2,3,4,5\}, \\ \{2,3,4\} \cap \{1,2,3,4,5\}.$$

- $\mathcal{M} = \{M_{\text{Tamaño}}, M_{\text{Precisión}}\}$: conjunto de medidas donde $M_{\text{Tamaño}}(e) = |eval(e)|$, y $M_{\text{Precisión}}(e) = \frac{|eval(e) \cap G|}{|eval(e)|}$.

Lo que buscamos es una expresión $e^* \in \mathcal{E}_F^{\mathcal{C}}$ que pertenezca al frente de Pareto respecto a las medidas de $\mathcal{M}$, en relación con el conjunto objetivo G, es decir, $e^* \in (\mathcal{E}_F^{\mathcal{C}})^{\mathcal{M}}$.

Llamamos e_i con $i \in \{1, \ldots, 9\}$ a las expresiones de $\mathcal{E}^{\mathcal{C}}$, en el orden en que las hemos escrito. Calculamos en la siguiente tabla las medidas para dichas expresiones, donde cada celda (i, j) será el valor $M_i(e_j)$ de evaluar la medida M_i en la expresión e_j:

	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9
$M_{\text{Tamaño}}$	5	2	3	1	1	2	3	2	3
$M_{\text{Precisión}}$	0.6	0.5	0.66	1	1	0.5	0.66	0.5	0.66

Esta tabla nos ayuda a visualizar qué expresiones pertenecen al frente de Pareto:

$$\mathcal{E}^{\mathcal{M}} = \{e_1, e_3, e_4, e_5, e_7, e_9\}$$

Cada una de estas soluciones constituye una solución a nuestro problema.

Con este ejemplo vemos que, mientras que $\mathcal{E}$ describe cómo se construyen combinaciones de los conjuntos de F y potenciales soluciones, $\mathcal{R} = eval(\mathcal{E})$ describe los conjuntos finales que se obtienen al evaluarlas. Como sobre $\mathcal{E}$ no hay restricciones impuestas, el tamaño de $\mathcal{E}$ es en general infinito. Tras aplicar las restricciones, el tamaño de $\mathcal{E}^{\mathcal{C}}$ sí puede ser finito. El tamaño de $\mathcal{R}$, sin embargo, está siempre superiormente acotado por el tamaño de $\mathcal{P}(U)$ como veremos en la sección de estructuras algebraicas al presentar el concepto de conjunto potencia.

Este ejemplo es una particularización extremadamente sencilla del problema, que nos ha permitido visualizar de forma inmediata las posibles soluciones y su evaluación mediante distintas medidas. En la práctica, los espacios de expresiones suelen ser mucho más amplios y complejos, por lo que la identificación del frente de Pareto o incluso de una buena aproximación resulta un proceso no trivial.

1.2. Contexto y Motivación

Una vez que hemos definido nuestro problema, los elementos que intervienen en él, y los objetivos que nos hemos impuesto para este trabajo, resulta natural presentar la motivación para estudiarlo, así como el contexto histórico en el que surge y el de las áreas que incumbe.

Podemos decir que el marco para nuestro problema empieza a formularse con el desarrollo del concepto de recubrimiento de conjuntos, que puede entenderse como una particularización de nuestro problema, y merece ser destacado por su antigua historia en las matemáticas y sus grandes aportaciones. Este surge en el siglo XIX, cuando se empieza a formalizar el concepto en teoría de conjuntos [Bou94], incluyendo algunas nociones que hoy son de gran importancia como álgebra de conjuntos y anillos de conjuntos, en las que profundizaremos más adelante al analizar las estructuras que induce la familia de conjuntos F en el conjunto de expresiones evaluadas $\mathcal{R}$. Más tarde, en el siglo XX aparece también en el ámbito topológico, cuando se habla de recubrimientos abiertos de espacios topológicos. Aunque nuestro problema no se formula en ese ámbito, esta conexión nos permite reforzar la idea de la importancia del concepto de recubrimiento.

Como hemos indicado, otro objetivo fundamental de nuestro problema es el análisis de la complejidad del mismo, que haremos en la última sección de la parte matemática. Esto podemos hacerlo gracias a un salto decisivo para nuestro trabajo que se produce más recientemente, en la década de 1970, con el desarrollo de la teoría de la complejidad computacional, con el Teorema de Cook y el desarrollo del concepto NP-Completo. Estas nociones son las que nos permitirán clasificar nuestro problema según las distintas clases de complejidad existentes, para saber cómo abordar su resolución en la práctica, al relacionarlo con otros cuya clase de complejidad es conocida. Por ejemplo, la publicación de la lista de 21 problemas NP-Completos de Richard Karp, nos presenta el problema del conjunto de cobertura (Set Cover Problem), popularizando así en el ámbito de la informática el concepto de recubrimiento de conjuntos. Este consiste en: dado un conjunto de elementos, U, y una familia F de subconjuntos de U, ¿se puede encontrar un subconjunto $F' \subseteq F$ de k o menos conjuntos con unión U? Este problema de decisión también tiene una versión de optimización: en las mismas condiciones, minimizar el número de subconjuntos que usa $F' \subseteq F$ [HS04].

A partir del Set Cover Problem han surgido otras variantes como el Exact Cover, que consiste en: dado un conjunto U y una familia F de subconjuntos de U, encontrar un subconjunto $F' \subseteq F$ donde cada elemento de U aparece en un y solo un subconjuntos de F'. Otra variante es el Exact Cover By 3-Sets (X3C), que consiste en: dado un conjunto U, con cardinal múltiplo de tres, y una familia F de subconjuntos de U de tres elementos, encontrar un subconjunto $F' \subseteq F$ donde cada elemento de U aparece en un y solo un elemento de F' [GJ90]. Todas estas variantes que pueden verse como particularizaciones de nuestro problema en las que los conjuntos $\mathcal{C}$ y $\mathcal{M}$ son muy concretos. Estos problemas han sido objeto de amplios estudios, lo que demuestra que el marco en el que trabajamos es una de las líneas clásicas de estudio en matemáticas y ciencias de la computación.

Aunque el problema que abordamos en este trabajo está estrechamente relacionado con estas formulaciones clásicas, es importante recalcar que no se trata exactamente del mismo problema. Como ya hemos presentado, en nuestro caso somos más flexibles permitiendo también el uso de otras operaciones aparte de la unión, con el objetivo de aproximar G lo mejor posible según medidas específicas, que no siempre se traduce en hallar una partición suya o una cobertura exacta.

Por otro lado, además del interés teórico, nuestro problema tiene también muchas aplicaciones prácticas en distintas áreas, especialmente en la toma de decisiones y la optimización de recursos. Entre ellas destacan los problemas de recubrimiento o cobertura espacial, en los que

se busca garantizar que una determinada región física quede completamente cubierta por un conjunto limitado de recursos, como por ejemplo el despliegue de antenas para ofrecer cobertura Wi-Fi, la ubicación óptima de cámaras o sensores, etc. Veamos algunos ejemplos relevantes:

- En marketing: colocar anuncios de forma que cubran un público objetivo, sin redundancias (sin que las personas los vean varias veces) y sin desperdiciar recursos (evitando que los vean personas a las que no va dirigido). En términos de nuestro problema:
  - Universo U: todos los clientes potenciales de un mercado.
  - Conjunto objetivo G: el público objetivo. Por ejemplo, mujeres de entre 20 y 30 años.
  - Familia F: los conjuntos de clientes alcanzados por cada canal publicitario. Por ejemplo, F_1 =“lectores de la revista X”, F_2 =“espectadores del canal Y”, etc.
  - Problema: encontrar la expresión que cubra G (maximizar el alcance) minimizando la redundancia y el coste (no alcanzar a gente fuera de G).

- En logística: minimizar el número de camiones utilizados en la distribución de mercancías [MMW24]. En términos de nuestro problema:
  - Universo U: todas las ubicaciones de entrega de una ciudad.
  - Conjunto objetivo G: el conjunto de ubicaciones que deben ser cubiertas en una jornada.
  - Familia F: las rutas. Cada F_i es el conjunto de ubicaciones que puede cubrir el camión i.
  - Problema: encontrar la expresión que cubra G, sujeta a la restricción (un objetivo) de minimizar el número de F_i utilizados (minimizar camiones).

- En problemas de clasificación en informática: un ejemplo es una práctica de la asignatura de Inteligencia de Negocio del 4º curso del grado en Ingeniería Informática, que hemos cursado recientemente: predicción de aprobación de créditos.
  - Universo U: todos los solicitantes de crédito.
  - Conjunto objetivo G: el conjunto de solicitantes de bajo riesgo.
  - Familia F: los conjuntos de solicitantes que cumplen una característica. Por ejemplo, F_1 =“ingresos altos”, F_2 =“tiene propiedad”, F_3 =“historial de créditos limpio”, etc.
  - Problema: encontrar la expresión que mejor se solape con G.

  Encontrar esta combinación puede ayudar en la predicción en casos futuros, así cómo en la interpretabilidad de la decisión de aceptar o rechazar un crédito.

- En redes de comunicación: colocar antenas o puntos de acceso para asegurar cobertura en todo el área.
  - Universo U: todos los puntos discretos (coordenadas) de una región.

  - Conjunto objetivo G: el conjunto de ubicaciones que deben tener cobertura. Por ejemplo, hospitales, centros urbanos, etc.
  - Familia F: el área de cobertura de cada antena. F_i es el conjunto de puntos que la antena en la posición i puede cubrir.
  - Problema: encontrar la expresión que cubra G (un problema de recubrimiento) con el menor número de conjuntos posible (minimizar el número de antenas).

- En biología computacional: similar al problema de selección de créditos, consiste en identificar las causas genéticas de una enfermedad.
  - Universo U: todos los pacientes participantes en el estudio clínico.
  - Conjunto objetivo G: el grupo de pacientes que han desarrollado cierta enfermedad.
  - Familia F: las mutaciones genéticas. Cada conjunto contiene a los pacientes que presentan una variante genética concreta. Por ejemplo, $F_i =$"pacientes con la mutación en el gen i").
  - Problema: encontrar la combinación de mutaciones que mejor explica o predice qué pacientes desarrollan la enfermedad (G).

Estas aplicaciones muestran la importancia del problema de aproximación de conjuntos y sus distintas particularizaciones, en la toma de decisiones eficientes y la optimización de recursos, lo que lo convierte en un problema de gran interés no solo en el ámbito computacional.

1.3. Metodología

Como hemos visto, este trabajo está dividido en dos bloques principales: uno más teórico, matemático; y otro más práctico, computacional.

En la primera parte, partimos de la formalización matemática del problema de aproximación de conjuntos que hemos hecho y de todos sus elementos, para hacer un estudio de sus propiedades. Dado que el problema es muy general, recurrimos a particularizaciones que nos permitan analizar con mayor claridad estructuras concretas.

Para este análisis nos apoyamos en diferentes áreas de las matemáticas, empezando por el álgebra de conjuntos y las estructuras algebraicas, al estudiar conceptos básicos que nos permitirán conocer y entender bien la base de nuestro problema, incidiendo especialmente en los elementos U, F, $\mathcal{O}$ y en el conjunto $\mathcal{R}$. Analizaremos tanto las propiedades del universo, como las estructuras que induce la familia de conjuntos F en $\mathcal{R}$. También, en relación con el conjunto de medidas $\mathcal{M}$, haremos uso de la teoría de la medida del análisis al explorar distintas medidas como herramientas para definir restricciones y medidas de calidad, entre las que aparecerán las medidas de asociación y tablas de contingencia de la estadística. Por último, para terminar el estudio teórico del problema, estudiaremos la NP-Completitud y la teoría de la complejidad computacional, para poder clasificar nuestro problema y saber cómo abordarlo de la mejor manera en la práctica.

La idea es dar un contexto histórico y teórico general de todas estas áreas, profundizando solo en los conceptos de cada una que son directamente relevantes en nuestro problema. En este sentido, nuestro trabajo se trata más bien de un estudio transversal por distintas ramas matemáticas, en lugar de un análisis en profundidad de una sola de ellas. Este enfoque nos permite conectar distintos conceptos trabajados a lo largo del Grado en Matemáticas, estudiando sus interconexiones e integrándolos en un solo problema práctico.

En la segunda parte, vamos a abordar el problema desde un punto de vista práctico, en el que implementaremos y estudiaremos el comportamiento de distintas aproximaciones algorítmicas y analizaremos su comportamiento en varias particularizaciones del problema. Utilizaremos en cada caso las medidas adecuadas correspondientes para guiar el proceso de construcción de la solución, y asegurar que se cumplen las restricciones teóricas impuestas. En particular, los algoritmos que incluiremos serán: búsqueda exhaustiva con profundidad limitada, Greedy, y un algoritmo genético.

Finalmente, evaluaremos los resultados utilizando algunas de las medidas previamente definidas, para comparar de forma objetiva la calidad de las soluciones obtenidas y el rendimiento de cada algoritmo, así como plantear posibles mejoras y perspectivas futuras.

1.4. Particularizaciones del problema

Hasta ahora hemos presentado nuestro problema general de forma muy amplia, y hemos visto que se trata de una generalización de problemas ampliamente conocidos en las matemáticas, que llevan muchos años siendo estudiados por su gran complejidad. Es por ello que a lo largo de este trabajo, recurrimos a particularizaciones de nuestro problema general, que nos facilitarán su estudio y nos permitirán abordarlo de forma más intuitiva. Estas particularizaciones consisten en aplicar restricciones específicas ($\mathcal{E}^{\mathcal{C}}$ de nuestro problema), de manera que cada una define un subconjunto del espacio general de soluciones, y permite adaptar el problema a estrategias de resolución prácticas que no podríamos aplicar sobre el problema general.

En las siguientes secciones del trabajo iremos presentando distintas particularizaciones asociadas a conceptos teóricos específicos, como los recubrimientos, las particiones o las estructuras algebraicas. No obstante, en esta sección introductoria incluimos dos casos generales que ilustran la lógica del proceso de particularización y que pueden aplicarse en diferentes contextos:

- Particularización sin repetición:
  - $\mathcal{C} = \{C_1\}$, donde $C_1(e) : \forall i, j \in \{1, \ldots, |\mathcal{F}^*(e)|\};\ i \neq j, \quad F^{(i)}(e) \neq F^{(j)}(e)$

  Imponemos que los subconjuntos de F utilizados en una expresión no se repitan.

- Particularización sin conjunto vacío en F:
  - $\mathcal{C} = \{C_1\}$, donde $C_1(e) : \emptyset \notin \mathcal{F}(e)$

Es decir, el conjunto vacío no se utiliza en la expresión. En el planteamiento general de nuestro problema permitimos que el conjunto vacío pertenezca a la familia F, lo cual es necesario para poder trabajar con determinadas estructuras algebraicas (como anillos o álgebras de conjuntos) como veremos en la Sección 2.1 de este trabajo. Sin embargo, en un contexto práctico, puede ser útil excluir $\emptyset$ de F, ya que no contribuye a mejorar la aproximación al conjunto objetivo G y puede generar soluciones triviales, como expresiones en las que el resultado de la evaluación es $\emptyset$.

2. Fundamentos matemáticos

2.1. Estructuras algebraicas

Nuestro problema puede abordarse desde distintas áreas de las matemáticas, cada una aportando perspectivas e información diferente. Empezar por entender las estructuras de los conjuntos básicos que aparecen en nuestro problema puede ser lo más natural, para posteriormente, poder comenzar a medirlos, compararlos o estudiar la complejidad del problema en conjunto. Por ello, en primer lugar, nos enfocamos en el álgebra y en las estructuras algebraicas que intervienen, analizando tanto sus propiedades como la interacción entre ellas y las posibles estructuras derivadas. Esta sección se centrará principalmente en los elementos U y F del problema, así como en el conjunto $\mathcal{R} = eval(\mathcal{E})$. Más adelante introduciremos medidas que nos permitirán comparar subconjuntos y asignarles valores en función de ciertas propiedades.

Históricamente, la teoría de conjuntos comienza a desarrollarse a finales del siglo XIX con los trabajos de George Cantor en la rama del análisis, donde introdujo los cardinales infinitos entre otros conceptos. En esos momentos existían muchas paradojas alrededor de dicha teoría, como la de Russell, que motivaron la búsqueda de axiomas más rigurosos. Gottlob Frege propuso una primera formalización en 1903, pero Russell evidenció sus inconsistencias. En 1908, Ernst Zermelo propone una axiomatización que termina Abraham Fraenkel en 1922, dando lugar a la teoría de conjuntos moderna. Además, a Zermelo se le atribuye el conocido axioma de elección. Hoy en día, la mayor parte del álgebra se sigue basando dicha teoría de conjuntos. Aunque no se puede demostrar si la teoría de conjuntos moderna es consistente de acuerdo con el segundo teorema de incompletitud de Gödel, no se han encontrado contradicciones en el último siglo, y por tanto se sigue basando la mayor parte de álgebra en ella [FD07].

2.1.1. Conjunto potencia

Esta teoría de conjuntos moderna y su formulación axiomática, nos da el marco formal que necesitamos para trabajar con estructuras y operaciones que conocemos hoy en día sobre bases sólidas. Cuando trabajamos con subconjuntos en el contexto de estructuras algebraicas aparecen conceptos básicos que es fundamental introducir. Es el caso del conjunto potencia, que ya hemos nombrado antes y que en nuestro problema es especialmente relevante por distintas razones. A continuación recordamos su definición, que podemos encontrar en [Dev93], entre otros:

Definición 2.1 (Conjunto Potencia)**.** El conjunto potencia de U, denotado $\mathcal{P}(U)$, es el conjunto cuyos elementos son todos los subconjuntos de U. Se escribe

$$\mathcal{P}(U) = \{X | X \subseteq U\}$$

Es interesante conocer este concepto fundamental, pues nos permite deducir que el tamaño de F está limitado superiormente por el de $\mathcal{P}(U)$. De la misma manera, también limita el tamaño de $\mathcal{R}$. Esto se debe a que cualquier operación entre subconjuntos de U siempre produce otro subconjunto de U. Es decir, no se generan elementos ajenos a U, y por tanto, ningún conjunto fuera de $\mathcal{P}(U)$.

El conjunto potencia de un conjunto de cardinal n contiene 2^n conjuntos [Lip98]. Este resultado se obtiene de un razonamiento combinatorio: cada elemento puede incluirse o no en un subconjunto dado, por lo que el número total de subconjuntos es la suma de todas las combinaciones posibles,

$$|\mathcal{P}(U)| = \sum_{k=0}^{n} \binom{n}{k} = 2^n.$$

Lo vemos sobre el siguiente ejemplo al que haremos referencia a lo largo del trabajo para mostrar diferentes conceptos de forma intuitiva:

Ejemplo 2.2. Consideramos los siguientes elementos:

- $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
- $F = \{\{1, 2\}, \{2, 3, 4\}, \{2, 5\}, \{4, 7\}, \{4, 5, 6\}, \{6, 7, 8\}, \{7, 10\}, \{8, 9\}, \{10\}\}$
- $G = \{2, 4, 5, 7, 10\}$

En este ejemplo, el conjunto potencia de U, $\mathcal{P}(U)$, está formado por los $2^{10} = 1024$ subconjuntos posibles de U, entre los que están:

- el conjunto vacío: $\emptyset$
- todos los conjuntos unitarios: $\{1\}, \ldots, \{10\}$
- todas las combinaciones de entre 2 y 9 elementos: $\{1, 2\}, \{2, 5, 7\}, \ldots$
- el conjunto total: U

En este caso, la familia de conjuntos resultantes $\mathcal{R}$ tendrá a lo sumo 1024 elementos. Esto implica que, por muchas expresiones distintas que se construyan en $\mathcal{E}$, al evaluarlas nunca obtendremos más de 1024 conjuntos diferentes. Esto es de gran utilidad en nuestro problema, pues puede permitir, por ejemplo, detener el proceso de construcción de expresiones una vez alcanzado el número máximo de conjuntos resultantes posibles, en particularizaciones del problema donde las restricciones no afectan a la estructura de la expresión.

Además de aportarnos información sobre el elemento F de nuestro problema y sobre $\mathcal{R}$, el conjunto potencia es la base de estructuras matemáticas más complejas que nos permiten añadir propiedades a nuestro problema, así como enriquecer su análisis.

2.1.2. Retículo y álgebra de Boole

Una de las estructuras que se construyen a partir del conjunto potencia es el retículo, que estudiamos para posteriormente presentar el concepto de álgebra de Boole. Antes de presentarlo, recordamos algunos conceptos subyacentes como las relaciones de orden y el conjunto ordenado que podemos encontrar, por ejemplo, en [DP02]:

Definición 2.3 (Relación de orden)**.** Sea X un conjunto. Una operación binaria $\leq$ sobre X es una relación de orden si cumple las siguientes propiedades:

- Reflexividad: $\forall x \in X,\ x \leq x$.
- Antisimetría: $\forall x, y \in X, \quad x \leq y \ \text{ y } \ y \leq x \Rightarrow x = y$.
- Transitividad: $\forall x, y, z \in X, \quad x \leq y \ \text{ y } \ y \leq z \Rightarrow x \leq z$.

En nuestro caso, $\subseteq$ es una relación de orden sobre $\mathcal{P}(U)$, pues cumple:

- Reflexividad: $\forall A \in \mathcal{P}(U)$ se tiene $A \subseteq A$.
- Antisimetría: si $A \subseteq B$ y $B \subseteq A$, entonces $A = B$.
- Transitividad: si $A \subseteq B$ y $B \subseteq C$, entonces $A \subseteq C$.

Como tenemos una relación de orden sobre $\mathcal{P}(U)$, decimos que $(\mathcal{P}(U), \subseteq)$ es un conjunto parcialmente ordenado. En este caso, la inclusión $\subseteq$ define un orden parcial, ya que no todos los subconjuntos de U son comparables entre sí.

Para representar cualquier conjunto ordenado, se puede utilizar un diagrama de Hasse [Gri99]. En nuestro caso, como en el ejemplo base 2.2, $\mathcal{P}(U)$ tiene demasiados elementos lo que dificulta la visualización, lo mostramos para un nuevo ejemplo aún más sencillo:

Ejemplo 2.4. $U = \{1, 2, 3, 4\}$

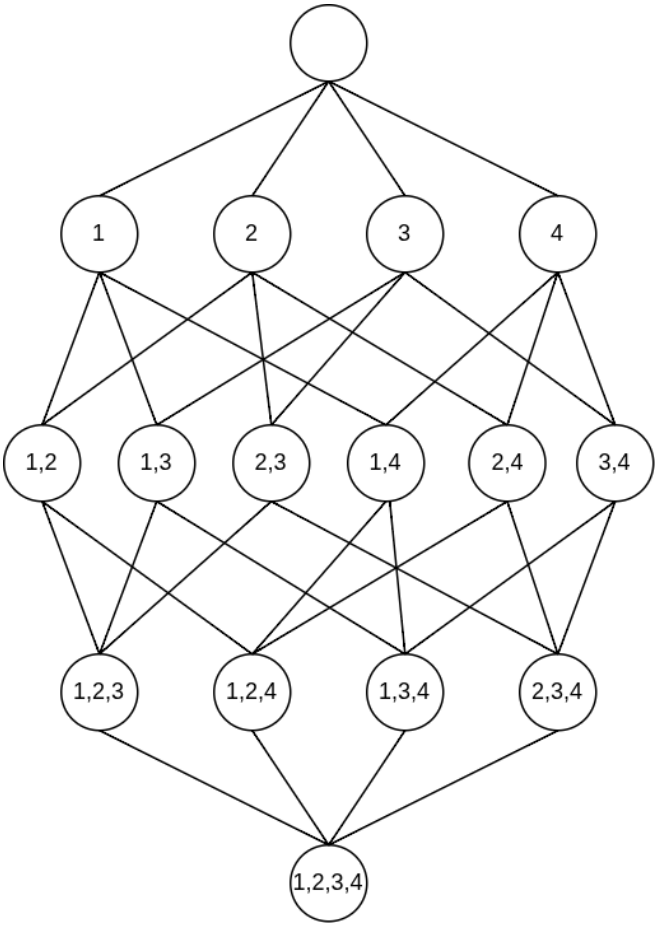

Figura 2.1.: Diagrama de Hasse del conjunto potencia de $U = \{1, 2, 3, 4\}$

El último paso antes de poder dar la definición formal de retículo, es recordar el concepto de cota superior e inferior en el conjunto $\mathcal{P}(U)$, utilizado la relación de inclusión $\subseteq$ en lugar de la relación usual $\leq$ que aparece, por ejemplo, en [DP02].

Definición 2.5 (Cota superior y supremo)**.** Sea U un universo y $X \subseteq \mathcal{P}(U)$ una familia de subconjuntos. Decimos que $A \in \mathcal{P}(U)$ es una cota superior de X si contiene a todos sus elementos:

$$\forall X_i \in X, \quad X_i \subseteq A.$$

Al menor de dichos conjuntos lo denominamos supremo de X y lo denotamos por $\sup(X)$.

De manera análoga, se tiene la siguiente definición:

Definición 2.6 (Cota inferior e ínfimo)**.** Sea $B \in \mathcal{P}(U)$. Decimos que B es una cota inferior de X si está contenido en todos los elementos de la familia:

$$\forall X_i \in X, \quad B \subseteq X_i.$$

Al mayor de dichos conjuntos lo denominamos ínfimo de X y lo denotamos por $\inf(X)$.

Existen distintas formulaciones generales del concepto de retículo en teoría de órdenes, que pueden consultarse en [DP02], así cómo algunas de sus propiedades. A continuación presentamos la definición general:

Definición 2.7. (Retículo) Un retículo es un conjunto ordenado $(L, \leq)$ en el que cualquier conjunto finito tiene supremo o ínfimo.

Si además un retículo es distributivo y todos sus elementos tienen complemento, decimos que se trata de un álgebra de Boole [DP02].

En nuestro caso, nos centraremos en el caso particular del retículo de conjuntos, considerando la estructura que se obtiene sobre el conjunto potencia de U con la relación de inclusión:

$$(\mathcal{P}(U), \subseteq).$$

Podemos comprobar fácilmente que se trata de un retículo, pues para cualquier colección finita de subconjuntos $X \subseteq \mathcal{P}(U)$ existen los elementos:

$$\sup(X) = \bigcup_{A \in X} A, \qquad \inf(X) = \bigcap_{A \in X} A.$$

Por tanto, la unión actúa como supremo y la intersección como ínfimo en $\mathcal{P}(U)$. Esto es fácil de ver en un diagrama de Hasse. Para encontrar el supremo de una familia de conjuntos basta con bajar en el diagrama hasta encontrar el menor conjunto que contiene a todos ellos. De la misma manera, para determinar el ínfimo, ascendemos en el diagrama hasta encontrar el mayor conjunto que está contenido en todos los considerados. Siguiendo con nuestro ejemplo 2.4 y utilizando nuestra Figura 2.1, si consideramos $X = \{\{3\}, \{1, 2\}\}$, siguiendo hacia abajo, encontramos el conjunto $\{1, 2, 3\}$, que es el menor conjunto que contiene a todos los de X. Este será el supremo. Para el ínfimo, vemos que el único conjunto contenido en ambos de X es $\emptyset$.

Por lo tanto, cualquier colección de subconjuntos de $\mathcal{P}(U)$ siempre va a tener ínfimo y supremo, que en los casos extremos serán el conjunto vacío y el total respectivamente. Concluimos así que $(\mathcal{P}(U), \subseteq)$ es un retículo.

Vemos que, como la unión y la intersección son distributivas la una con respecto de la otra y, además, para todo $A \in \mathcal{P}(U)$ se cumple que $A \cup (U \setminus A) = U$ y $A \cap (U \setminus A) = \emptyset$, concluimos que $\mathcal{P}(U)$ es un retículo distributivo en el que todo elemento tiene complemento, y por tanto, un álgebra de Boole.

2.1.3. Relación con la lógica proposicional

El hecho de que $\mathcal{P}(U)$ sea un álgebra de Boole, nos permite conectarlo directamente con la lógica proposicional, lo cual relaciona nuestro problema estrechamente con la lógica matemática y la teoría de la computación [Hal74a]. A cada elemento de $\mathcal{P}(U)$, es decir, a cada

subconjunto de U, se le puede asociar una proposición lógica. Además, las operaciones de unión, intersección y complemento corresponden a la disyunción, la conjunción y la negación, respectivamente.

En la siguiente tabla mostramos a qué operación lógica corresponde cada operación de conjuntos:

Operación en conjuntos	Símbolo	Operación lógica
Unión	$A \cup B$	Disyunción ($p \vee q$)
Intersección	$A \cap B$	Conjunción ($p \wedge q$)
Complemento	$U \setminus A$	Negación ($\neg p$)
Conjunto vacío	$\emptyset$	Falso ($\bot$)
Conjunto total	U	Verdadero ($\top$)

Tabla 2.1.: Correspondencia entre álgebra de Boole y lógica proposicional.

Por ejemplo, cada subconjunto de U corresponde a una proposición lógica p_A que indica si un elemento u pertenece o no a A. Para el ejemplo 2.4, el subconjunto $A = \{1,3\}$, se asocia con la proposición p_A, que vale 0 si $u \notin \{1,3\}$ y 1 en caso contrario. Sea $B = \{2,3\}$ otro subconjunto de U, tenemos que

$$A \cup B = \{1,2,3\}, \qquad p_{A\cup B}(u) = p_A(u) \vee p_B(u).$$

Con la intersección:

$$A \cap B = \{3\}, \qquad p_{A\cap B}(u) = p_A(u) \wedge p_B(u).$$

Y con el complemento:

$$U \setminus A = \{2,4\}, \qquad p_{U\setminus A}(u) = \neg p_A(u).$$

Así, tenemos un marco equivalente a la lógica proposicional, de manera que trabajar con familias de subconjuntos equivale a razonar con fórmulas proposicionales. Esto nos será de utilidad más adelante, en la Subsección 2.3.5 para plantear nuestro problema en términos del problema de satisfacción booleana, un problema ampliamente reconocido y estudiado.

2.1.4. Recubrimientos y particiones

Sobre este marco algebraico, podemos introducir particularizaciones nuevas basadas en la relación entre la familia F y el conjunto objetivo G. Veamos primero algunas definiciones que describen distintas formas en que los subconjuntos de F pueden combinarse para representar o aproximar G. Presentamos en primer lugar la definición de recubrimiento de G, que podemos encontrar en [Wil04]:

Definición 2.8 (Recubrimiento de un conjunto)**.** La colección $F' = \{F_{i_1}, \ldots, F_{i_k}\} \subseteq F$, distinta del vacío, es un recubrimiento del conjunto G si la unión de sus elementos contiene a G, es decir, si $G \subseteq \bigcup_{j=1}^{k} F_{i_j}$.

Surge así la siguiente particularización:

Particularización de Recubrimiento de G:

- $\mathcal{C} = \{C_1, C_2\}$, donde

$$C_1(e) : \mathcal{O}p(e) \subseteq \{\cup\}$$
$$C_2(e) : G \subseteq eval(e)$$

  Es decir, se impone que la evaluación de e sea un recubrimiento de G.

En nuestro problema, llamaremos $F' = \mathcal{F}^*(e)$ para una expresión $e \in \mathcal{E}$.

Veamos ahora una definición un poco más restrictiva que podemos encontrar en [Lip98]:

Definición 2.9 (Partición)**.** Sea $F' = F_{i_1}, \ldots, F_{i_k}$ una colección de elementos de F. Diremos que F' es una partición de G si cumple:

- Forma un recubrimiento exacto de G:

$$\bigcup_{j=1}^{k} F_{i_j} = G,$$

- Sus elementos son disjuntos dos a dos:

$$\forall p \neq q,\ F_{i_p} \cap F_{i_q} = \emptyset.$$

Sin embargo, obtener una partición de G no siempre va a ser de interés, necesario o incluso posible, pues puede que haya elementos de G que no queden cubiertos por ningún elemento de F, que algunos de los subconjuntos de F escogidos se solapen entre sí, o bien que se tengan en cuenta elementos que se salen del conjunto G.

Además, el número de particiones posibles para un conjunto finito depende de su cardinal, n, y viene dado por el número de Bell [Aig07]:

- $B(0) = 1$
- $B(n) = \sum_{k=0}^{n-1} \binom{n-1}{k} B(k)$

Es claro que este número crece rápidamente con n, lo cuál nos da una idea de la complejidad del problema de trabajar con particiones incluso en casos sencillos, y justifica la utilidad y necesidad de otras variantes y criterios. Por ello, en este trabajo nos centramos en encontrar la mejor aproximación posible priorizando diferentes objetivos en lugar de limitándonos a solo uno.

En muchos casos puede ser preferible relajar la condición de partición y permitir solapamientos entre los subconjuntos. Dado que toda partición es un caso particular de recubrimiento,

esta relajación amplía el espacio de soluciones y puede facilitar la búsqueda de expresiones que aproximen G de forma eficiente.

Podemos definir varias particularizaciones a partir del concepto de partición:

- **Particularización al Set Cover Problem**:
  - $\mathcal{C} = \{C_1, C_2\}$, donde

$$C_1(e) : \mathcal{O}p(e) \subseteq \{\cup\}$$
$$C_2(e) : G = eval(e)$$

  Es decir, partiendo de la particularización de Recubrimiento de G, imponemos además que la evaluación de e sea un recubrimiento exacto de G. Esto simula el bien conocido y ampliamente estudiado problema del Set Cover.

- **Particularización al Exact Cover**:
  - $\mathcal{C} = \{C_1, C_2, C_3\}$, donde

$$C_1(e) : \mathcal{O}p(e) \subseteq \{\cup\}$$
$$C_2(e) : G = eval(e)$$
$$C_3(e) : \forall i \neq j, \quad F^{(i)}(e) \cap F^{(j)}(e) = \emptyset$$

  Añadiendo una restricción adicional sobre la particularización anterior, C_3, exigimos que los subconjuntos de F utilizados en una expresión, sean disjuntos dos a dos. Esta condición más fuerte da lugar al conocido Exact Cover Problem.

Volviendo a nuestro Ejemplo 2.2, donde $G = \{2, 4, 5, 7, 10\}$, podemos encontrar:

- Recubrimiento de G: $F' = \{\{2, 3, 4\}, \{4, 5, 6\}, \{6, 7, 8\}, \{10\}\}$
- Recubrimiento exacto de G: $F' = \{\{2, 5\}, \{4, 7\}, \{7, 10\}\}$
- Partición exacta de G: $F' = \{\{2, 5\}, \{4, 7\}, \{10\}\}$

2.1.5. Estructuras inducidas

En la subsección anterior introdujimos distintas particularizaciones del problema general a partir de conceptos centrados en la relación entre la familia F y el conjunto G. En esta sección continuaremos con el mismo enfoque de particularización, pero ahora basándonos en estructuras algebraicas.

En el ámbito del álgebra, una estructura surge cuando dotamos a los conjuntos de ciertas leyes de composición [Bou04], y por lo tanto se define por:

- Un conjunto subyacente.
- Una o varias leyes de composición definidas sobre el conjunto.

- Unos axiomas que deben verificar dichas leyes y explican cómo se comportan los elementos entre ellos.

En nuestro caso, a partir de F, podemos considerar varias estructuras algebraicas de conjuntos que surgen de cerrar F bajo distintas operaciones. Estas estructuras generadas por F definen subconjuntos de $\mathcal{P}(U)$, y por tanto también subconjuntos de $\mathcal{R}$, la familia de conjuntos resultantes de evaluar las expresiones en $\mathcal{E}$. Para la construcción de estas estructuras, hacemos todas las posibles combinaciones de elementos de F mediante las operaciones permitidas en cada caso, incluyendo también combinaciones previas. Presentamos a continuación las definiciones de anillo, semianillo y álgebra de conjuntos, que pueden encontrarse en [Hal74b]:

Definición 2.10 (Anillo de conjuntos)**.** Un anillo de conjuntos es una colección no vacía $\mathcal{A}$ de conjuntos, tal que si para todo $A, B \in \mathcal{A}$ se cumple:

$$A \cup B \in \mathcal{A} \quad \text{y} \quad A \setminus B \in \mathcal{A}.$$

En particular, como en nuestro caso $\mathcal{R}$ es finito, deducimos que también es cerrado bajo uniones e intersecciones finitas:

$$\bigcup_{A \in \mathcal{R}} A \in \mathcal{R} \quad \text{y} \quad \bigcap_{A \in \mathcal{R}} A \in \mathcal{R}.$$

Teniendo esto en cuenta, el conjunto vacío y el conjunto $\mathcal{P}(U)$ también son ejemplos de anillos de conjuntos.

Volviendo a nuestro marco, podemos imponer la estructura de anillo mediante la siguiente particularización:

Particularización a anillo de conjuntos:

- $\mathcal{C} = \{C_1\}$, donde $C_1(e) : \mathcal{O}p(e) \subseteq \{\cup, \setminus\}$

Ejemplo sencillo:

- $U = \{1, 2, 3, 4\}$
- $F = \{\{1\}, \{1, 2\}, \{3, 4\}, U\}$
- Operaciones permitidas: unión y diferencia

El espacio $\mathcal{E}$ de expresiones que se genera es: $\mathcal{E} = \{\{1\}, \{1, 2\}, \{3, 4\}, \{1\} \cup \{1, 2\}, \{1\} \cup \{3, 4\}, \{1, 2\} \cup \{3, 4\}, \{1\} \setminus \{3, 4\}, \{1\} \setminus \{1, 2\}, \{1, 2\} \setminus \{3, 4\}, \dots\}$

En este caso, $\mathcal{E}$ tiene infinitos elementos, pues no hemos impuesto restricciones sobre el número máximo de operaciones o elementos de F que podemos utilizar.

La familia de conjuntos resultantes que se genera es:

$$\mathcal{R} = \{\emptyset, \{1\}, \{2\}, \{1, 2\}, \{3, 4\}, \{1, 3, 4\}, \{2, 3, 4\}, U\}$$

Esto es, el anillo de conjuntos generado por F.

Continuando ahora con la estructura de semianillo:

Definición 2.11 (Semianillo de conjuntos)**.** Un semianillo es una colección no vacía $\mathcal{A}$ de conjuntos tal que:

- si $A, B \in \mathcal{A}$ entonces $A \cap B \in \mathcal{A}$
- si $A, B \in \mathcal{A}$ y $A \subset B$, entonces $\exists\{D_1, \ldots, D_n\}$ una colección finita de conjuntos de $\mathcal{A}$, tal que:
$$B \setminus A = \bigcup_{i=1}^{n} D_i.$$

De nuevo, todos los semianillos siempre contienen al conjunto vacío.

En nuestro marco, la estructura de semianillo puede imponerse mediante la siguiente particularización:

Particularización a semianillo de conjuntos:

- $\mathcal{C} = \{C_1\}$, donde $C_1(e) : \mathcal{O}p(e) \subseteq \{\cap, \setminus\}$

Ejemplo sencillo:

- $U = \{1, 2, 3, 4\}$
- $F = \{\{1\}, \{2, 3\}, \{1, 2, 3\}, U\}$
- Operaciones permitidas: intersección y diferencia.

La familia de conjuntos resultantes que se genera es:

$$\mathcal{R} = \{\emptyset, \{1\}, \{4\}, \{1, 4\}, \{2, 3\}, \{1, 2, 3\}, \{2, 3, 4\}, U\}$$

Esto es, el semianillo generado por F.

Presentamos ahora la definición de álgebra de conjuntos:

Definición 2.12 (Álgebra de conjuntos)**.** Un álgebra de conjuntos (también a veces conocido como campo de conjuntos) es una colección no vacía $\mathcal{A}$ de conjuntos tal que:

- Si $A, B \in \mathcal{A}$, entonces $A \cup B \in \mathcal{A}$.
- Si $A \in \mathcal{A}$, entonces su complemento respecto a U, $U \setminus A \in \mathcal{A}$.

A partir de esta definición, presentamos una nueva particularización para simular esta estructura en nuestro problema:

Particularización a álgebra de conjuntos::

- $\mathcal{C} = \{C_1, C_2\}$, donde

$$C_1(e) : \mathcal{O}p(e) \subseteq \{\cap, \setminus\}$$
$$C_2(e) : \forall (e_1 \setminus e_2) \text{ subexpresión de } e, \quad e_1 = U$$

La segunda restricción es equivalente a decir que, todas las subexpresiones de e que utilicen la operación de diferencia, lo hacen con respecto a U. Esto nos permite simular la operación complemento aunque en la formulación general de nuestro problema las operaciones disponibles sean únicamente la unión, la intersección y la diferencia.

Ejemplo sencillo:

- $U = \{1, 2, 3, 4\}$
- $F = \{\{2, 3\}, \{1, 2, 3\}, U\}$
- Operaciones permitidas: complemento y unión.

La familia de conjuntos resultantes que se genera es:

$$\mathcal{R} = \{\emptyset, \{1\}, \{4\}, \{1, 4\}, \{2, 3\}, \{1, 2, 3\}, \{2, 3, 4\}, U\}$$

Esto es, el álgebra de conjuntos generado por F. Recordamos que podemos utilizar el complemento con respecto a U porque lo podemos ver como la diferencia con respecto a U.

La siguiente proposición es un resultado clásico en teoría de conjuntos, que puede encontrarse en [Lip98]:

Teorema 2.13 (Relación entre álgebra de conjuntos y anillo de conjuntos)**.** *Toda álgebra de conjuntos es también un anillo de conjuntos, pero el recíproco no es cierto.*

Demostración. Sea F un álgebra de conjuntos sobre U. Por definición, F está cerrada bajo complemento y unión finita, y contiene al conjunto vacío. Se tiene también que F es cerrada bajo intersección y diferencia, ya que:

$$A \cap B = \overline{(\overline{A} \cup \overline{B})}, \quad A \setminus B = A \cap \overline{B}$$

Por tanto, toda álgebra es también un anillo, ya que cumple las propiedades necesarias.
Sin embargo, no todo anillo de conjuntos es un álgebra, ya que un anillo no exige que $U \in F$. Por ejemplo, sea:

$$F = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\} \subseteq \mathcal{P}(\{1, 2, 3\})$$

Esta familia es un anillo (cerrada bajo diferencia y unión), pero no es un álgebra, ya que $\{1, 2, 3\} \notin F$. Por tanto, la implicación contraria no se cumple. □

Analizamos también el concepto de σ-álgebra, que va a ser esencial cuando queramos aplicar el concepto de medida clásico, y que se puede encontrar definido en [Hal74b].

Definición 2.14 (σ-álgebra)**.** Una σ-álgebra es un álgebra $F \subseteq \mathcal{P}(U)$ que además verifica:

$$\{F_n\}_{n \in \mathbb{N}} \subseteq F \quad \Longrightarrow \quad \bigcup_{n \in \mathbb{N}} F_n \in F.$$

Es decir, esta estructura es cerrada bajo uniones numerables. Además, tenemos en cuenta el siguiente teorema que se puede encontrar en [Lip98]:

Teorema 2.15 (Relación entre σ-álgebra de conjuntos y anillo de conjuntos)**.** *Todo σ-álgebra de conjuntos es también un anillo, pero el recíproco no es cierto.*

Demostración. Sea $F \subseteq \mathcal{P}(U)$ una σ-álgebra. Por definición, cumple:

- $U \in F$
- *Si $A \in F$, entonces $\overline{A} \in F$*
- *Si $\{A_n\}_{n\in\mathbb{N}} \subseteq F$, entonces $\bigcup_{n=1}^{\infty} A_n \in F$*

Como F está cerrada bajo complementos y uniones numerables, también lo está bajo intersecciones numerables (por las leyes de De Morgan), y en particular bajo intersección finita y unión finita (son un caso particular de las numerables). Además, está cerrada bajo diferencia, ya que:

$$A \setminus B = A \cap \overline{B}$$

Por tanto, F cumple las propiedades necesarias para ser un anillo de conjuntos.

Para demostrar que el recíproco no es cierto, consideramos el siguiente contraejemplo. Sea $U = \mathbb{N}$, definimos:

$$F = \{A \subseteq \mathbb{N} \mid A \text{ es finito o } \overline{A} \text{ es finito}\}$$

Esta familia es un anillo, ya que está cerrada bajo diferencia y unión finita:

- *La unión de dos conjuntos finitos es finita.*
- *La diferencia de un conjunto finito y otro finito es finita.*
- *La diferencia de un conjunto cofinito (su complemento es un conjunto finito) y otro cualquiera sigue siendo finita o cofinita.*

Sin embargo, F no es una σ-álgebra, ya que no está cerrada bajo uniones numerables. Por ejemplo, sea $A_n = \{2n\}$ para cada $n \in \mathbb{N}$. Entonces $A_n \in F$ porque cada conjunto es finito, pero:

$$\bigcup_{n=1}^{\infty} A_n = \{2, 4, 6, 8, \dots\}$$

es un conjunto infinito con complemento también infinito, por tanto:

$$\bigcup_{n=1}^{\infty} A_n \notin F$$

Queda demostrado que F no es una σ-álgebra. □

Cabe destacar que, el concepto de álgebra de conjuntos implica estar cerrada bajo complementos y uniones finitas, y el concepto de σ-álgebra implica ser cerrada bajo complementos y uniones numerables (no necesariamente finitas). En nuestro caso U es finito, por lo que toda

unión numerable es finita. Por ello, toda álgebra de conjuntos sobre U es automáticamente una σ-álgebra y ambos conceptos coinciden en este marco.

Para concluir esta sección dedicada a estructuras algebraicas, presentamos a continuación una tabla resumen que recoge las particularizaciones introducidas y las restricciones asociadas a cada una de ellas:

Tabla 2.2.: Resumen de particularizaciones y sus restricciones asociadas.

Particularización	**Restricciones**
Sin repetición	$C_1(e) : \forall i \neq j,\ F^{(i)}(e) \neq F^{(j)}(e)$.
Sin conjunto vacío en F	$C_1(e) : \emptyset \notin \mathcal{F}(e)$.
Recubrimiento de G	$C_1(e) : \mathcal{O}p(e) \subseteq \{\cup\}$, $C_2(e) : G \subseteq eval(e)$.
Set Cover Problem	$C_1(e) : \mathcal{O}p(e) \subseteq \{\cup\}$, $C_2(e) : G = eval(e)$.
Exact Cover	$C_1(e) : \mathcal{O}p(e) \subseteq \{\cup\}$, $C_2(e) : G = eval(e)$, $C_3(e) : \forall i \neq j,\ F^{(i)}(e) \cap F^{(j)}(e) = \emptyset$.
Anillo de conjuntos	$C_1(e) : \mathcal{O}p(e) \subseteq \{\cup, \setminus\}$.
Semianillo de conjuntos	$C_1(e) : \mathcal{O}p(e) \subseteq \{\cap, \setminus\}$.
Álgebra de conjuntos	$C_1(e) : \mathcal{O}p(e) \subseteq \{\cap, \setminus\}$, $C_2(e) : \forall (e_1 \setminus e_2)$ subexpresión de e, $e_1 = U$.

2.2. Medidas

Una vez que hemos analizado las estructuras algebraicas de los distintos conjuntos presentes en la definición de nuestro problema, introducimos las medidas. Tal y como mencionamos en la definición de nuestro problema, estas son herramientas que nos permiten imponer restricciones (que definen el conjunto $\mathcal{E}^{\mathcal{C}}$ de nuestro problema), así como medir la calidad de nuestras soluciones. Esto último, nos permite por un lado ver cómo de buena es una aproximación y por otro, comparar entre varias potenciales soluciones y escoger una en base a un criterio determinado. Adelantamos que en la parte informática de este trabajo, veremos otro uso adicional de las medidas: su incorporación dentro de los propios algoritmos. Sin embargo, para ello es necesario anticipar algunos aspectos computacionales, por lo que esta sección se limita exclusivamente a su caracterización matemática.

Las medidas que vamos a estudiar no pretenden ser una clasificación exhaustiva, sino una selección representativa de las más empleadas en el análisis de conjuntos. Antes de presentarlas, conviene aclarar que estas pueden clasificarse de dos formas. Por un lado, según su forma o naturaleza matemática, que determina el tipo de relación que establecen en-

tre los conjuntos implicados (asociativa, direccional u otras). Por otro, según su función dentro del problema, distinguiendo entre medidas empleadas para imponer restricciones, para evaluar la calidad de las soluciones o incluso, en un contexto más computacional en el que profundizaremos más adelante, como parte del propio diseño heurístico de un algoritmo.

Las medidas de calidad u optimización son aquellas cuyo valor buscamos maximizar o minimizar según el objetivo concreto; mientras que las medidas de restricción son las que nos permiten imponer restricciones en nuestro problema, estableciendo un valor umbral que ha de cumplirse para constituir una solución válida.

Recordamos lo mencionado al definir el elemento $\mathcal{M}$ de nuestro problema, en relación a que algunas medidas pueden verse como una composición con la función *eval*. En los siguientes ejemplos que planteamos, cuando escribimos $M(\tilde{G})$ para alguna medida $M \in \mathcal{M}$, queremos simbolizar dicha composición con *eval*. Además, veremos que pasa de forma similar con algunas medidas que se ven como composiciones con la función $\mathcal{F}$, que asocia a cada expresión el conjunto de subconjuntos de F que utiliza.

2.2.1. Medidas de asociación

Las medidas de asociación son ampliamente utilizadas en el ámbito de la estadística y el análisis de datos para cuantificar la relación entre variables categóricas. Cuando tenemos una población clasificada según dos o más variables, es decir, clasificación cruzada, surgen preguntas sobre el grado de asociación o dependencia que hay entre ellas, y en qué grado conocer el valor de una variable nos da información sobre la otra. La elección de la medida de asociación debe hacerse en función del contexto y de los objetivos específicos del problema y no de manera generalizada [BL21]. De hecho, no siempre es posible definir un único objetivo para una investigación. Muchas veces buscamos simplemente resumir de forma compacta una gran cantidad de datos o identificar patrones.

En términos de probabilidad, si X e Y son variables categóricas, una medida de asociación suele expresar cuánto se desvía la distribución conjuntos $P(X, Y)$ del producto de las marginales $P(X)P(Y)$, lo cual equivale a la independencia. Una opción común es construir modelos probabilísticos de actividad predictiva, y definir la medida de asociación como una probabilidad derivada de dicho modelo [GK79]. Esto mide la reducción relativa en la probabilidad de error al predecir una variable conociendo la otra.

Trasladando esto a nuestro problema de aproximación de conjuntos, la pertenencia de los elementos al conjunto objetivo G y a la aproximación final $\tilde{G} = eval(e^*)$ juega un papel análogo al de las categorías de las variables anteriores. Es decir, medimos hasta qué punto ambas asignaciones de pertenencia están asociadas, sin asumir prioridad de una sobre la otra. Para ello, no necesitamos construir explícitamente los modelos probabilísticos que hemos mencionado, sino que nos centraremos en métricas más sencillas que pueden interpretarse como medidas de asociación derivadas de las tablas de contingencia 2×2 entre el conjunto objetivo G y la aproximación $\tilde{G} = eval(e^*)$:

	En G	**No en** G	**Suma por fila**
En $\tilde{G}$	Verdaderos Positivos (TP)	Falsos Positivos (FP)	$\lvert\tilde{G}\rvert$
No en $\tilde{G}$	Falsos Negativos (FN)	Verdaderos Negativos (TN)	$\lvert U \setminus \tilde{G}\rvert$
Suma por columna	$\lvert G\rvert$	$\lvert U \setminus G\rvert$	$\lvert U\rvert$

Tabla 2.3.: Tabla de contingencia 2×2 entre el conjunto objetivo G y la aproximación $\tilde{G}$.

Hemos tomado como clase "positiva" la pertenencia a G, pues es nuestra realidad. Por lo tanto, los Verdaderos Positivos (o True Positives) son los elementos que están tanto en la aproximación como en el conjunto objetivo; los Falsos Positivos (o False Positives) son los elementos que están en la aproximación pero no en G; los Falsos Negativos (o False Negatives) son los elementos de G que no están en la aproximación; y por último, los Verdaderos Negativos (o True Negatives) son los elementos que no están ni en $\tilde{G}$ ni en G.

En la definición general hemos considerado las medidas como aplicaciones $M : \mathcal{E} \to \mathbb{R}$, pues interpretamos el conjunto objetivo G como una constante externa al dominio de la función en los casos en que interviene en la medida. Esto simplifica la notación sin pérdida de generalidad y permite englobar tanto las medidas que dependen de G como las que no.

De la Tabla 2.3, podemos sacar algunas de las medidas de asociación más comunes:

- Índice de Jaccard: mide la proporción de elementos que pertenecen simultáneamente a dos conjuntos respecto al total de elementos que pertenecen al menos a uno de ellos. Es una de las medidas más utilizadas en problemas de cobertura y análisis de similitud, y se emplea en campos como la minería de datos, la recuperación de información y la ecología. Damos su fórmula clásica que se puede encontrar en [RV96]:

$$\mu = \frac{TP}{TP + FN + FP}.$$

  A continuación definimos esta medida para nuestro problema, donde resulta útil cuando se quiere evaluar la calidad de la aproximación sin priorizar ningún tipo de error:

  Definición 2.16 (Índice de Jaccard en el problema de aproximación de conjuntos)**.** Sea $G \subseteq U$ el conjunto objetivo y $e \in \mathcal{E}$ una expresión cuya evaluación $eval(e)$ constituye una aproximación a G. Definimos el índice de Jaccard de e como una medida M tal que:

$$M(e) = \frac{|G \cap eval(e)|}{|G \cup eval(e)|}.$$

  La medida toma valores en $[0, 1]$, alcanzando el valor 1 cuando la aproximación coincide exactamente con el conjunto objetivo y 0 cuando son disjuntas.

- Coeficiente de Sørensen-Dice: es muy similar al índice de Jaccard pero pondera más los valores correctamente cubiertos (TP), de modo que la intersección tiene el doble

de importancia que los errores por omisión o inclusión. Es muy utilizada en ámbitos como el aprendizaje automático, donde se emplea para comparar conjuntos, cadenas o estructuras de datos. Su fórmula clásica es:

$$\mu = \frac{2TP}{2TP + FP + FN}.$$

Damos su definición adaptándola a nuestro problema:

Definición 2.17 (Coeficiente de Sørensen-Dice en el problema de aproximación de conjuntos)**.** Sea $G \subseteq U$ el conjunto objetivo y $e \in \mathcal{E}$ una expresión cuya evaluación $eval(e)$ constituye una aproximación a G. Definimos el Coeficiente de Sørensen-Dice de e como una medida M tal que:

$$M(e) = \frac{2|G \cap eval(e)|}{|G| + |eval(e)|}.$$

Esta medida toma valores en $[0, 1]$, alcanzando 1 cuando la aproximación coincide con el conjunto objetivo y 0 cuando son disjuntas. En comparación con el índice de Jaccard, tiende a asignar valores más altos cuando la intersección entre G y $\tilde{G} = eval(e)$ es grande, por lo que es más tolerante a errores.

- Coeficiente de correlación ϕ: mide la calidad de clasificaciones binarias, teniendo en cuenta tanto los aciertos como los errores de ambos tipos. Es muy utilizado en ámbitos como la bioinformática y el aprendizaje automático, donde es más conocido por sus siglas en inglés MCC (*Matthews correlation coefficient*). A diferencia de otras medidas que solo consideran una parte de la tabla de contingencia, el coeficiente ϕ utiliza toda la información disponible, incluyendo los verdaderos negativos (TN). Es especialmente útil cuando existe un desbalanceo de clases. Su fórmula general que podemos encontrar en [BB01], es la siguiente:

$$\mu = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}.$$

Damos la definición adaptada a nuestro problema:

Definición 2.18 (Coeficiente de correlación ϕ en el problema de aproximación de conjuntos)**.** Sea $G \subseteq U$ el conjunto objetivo y $e \in \mathcal{E}$ una expresión cuya evaluación $eval(e)$ constituye una aproximación de G. Definimos el coeficiente de correlación ϕ de e como una medida μ tal que:

$$M(e) = \frac{(|G \cap eval(e)|) \cdot |U \setminus (G \cup eval(e))| - |eval(e) \setminus G| \cdot |G \setminus eval(e)|}{\sqrt{(|eval(e)|)\,(|G|)\,(|U \setminus G|)\,(|U \setminus eval(e)|)}}.$$

Toma valores en $[-1, 1]$, siendo $+1$ cuando la predicción es perfecta, 0 cuando es aleatoria, y -1 cuando es inversa (predice justo lo contrario a la realidad). En el contexto de nuestro problema, es especialmente informativa cuando G representa una fracción pequeña de U.

Todas estas medidas asociativas únicamente utilizan el conjunto objetivo y las potenciales soluciones, sin importar cómo se construyeron estas [HM15]. Por ello, nos dicen cómo de buenas son estas soluciones respecto a G, y nos permiten usarlas como criterio discriminador para seleccionar la solución óptima de entre todas las candidatas. No obstatnte, para interpretar correctamente estos valores numéricos, es fundamental distinguir la escala que emplea cada métrica. En la literatura se suelen diferenciar dos convenciones principales:

- Convención con signo: La medida toma valores entre -1 y $+1$, alcanzando estos extremos en caso de asociación completa, y siendo el 0 el caso de independencia. Es el caso del coeficiente de correlación ϕ. Esta convención es útil cuando existe un orden natural en las categorías que nos permita asignar un sentido a la relación.
- Convención sin signo: La medida toma valores entre 0 y +1, alcanzando el +1 en caso de asociación completa, y el 0 en caso de independencia. Es el caso de índice de Jaccard. Esta convención se utiliza cuando no hay un orden natural entre las categorías y no tiene sentido hablar de direcciones positivas o negativas.

Es importante destacar que la idea de "asociación completa" puede ser ambigua en contextos complejos. En cambio, el concepto de independencia sí es claro: por ejemplo, si la inclusión de un elemento en G es estadísticamente independiente de su aparición en $\tilde{G}$.

Finalmente, conviene y es interesante recordar que una vez definida una medida de asociación dentro de uno de estos rangos, es posible generar otras medidas cumpliendo la misma convención mediante transformaciones (como elevarla a una potencia o aplicarle funciones monótonas), sin perder su interpretación general.

2.2.2. Medidas direccionales

A diferencia de las medidas de asociación, las medidas direccionales no son simétricas, sino que evalúan una variable respecto de otra. Toman una como "realidad" y otra como su "predicción", por lo que son especialmente útiles cuando existe un criterio de referencia, como es nuestro caso, donde queremos aproximar G.

De la Tabla 2.3, podemos extraer algunas de las medidas direccionales más utilizadas cuyas fórmulas clásicas podemos encontrar en [MRS08]:

- Precisión: mide la proporción de elementos predichos que son correctos. Damos su fórmula clásica:
$$\mu = \frac{TP}{TP + FP}.$$
La adaptamos a nuestro problema:

  Definición 2.19 (Precisión en el problema de aproximación de conjuntos)**.** Sea $G \subseteq U$ el conjunto objetivo y $e \in \mathcal{E}$ una expresión cuya evaluación $eval(e)$ constituye una aproximación de G. Definimos la precisión de e como:
$$\mu(e) = \frac{|G \cap eval(e)|}{|eval(e)|}.$$

La medida toma valores en $[0, 1]$, siendo 1 el valor correspondiente a una aproximación perfecta y 0 cuando no existe ningún acierto.

Está directamente relacionada con la sobrecobertura ($|eval(e) \setminus G| = FP$), pues cuanto mayor es la precisión, menor es la sobrecobertura de G, es decir, el número de elementos de $U \setminus G$ incluidos en la aproximación. La diferencia es que la sobrecobertura se expresa en términos absolutos , mientras que la precisión es relativa, pues normaliza respecto al tamaño de $\tilde{G} = eval(e)$:

$$1 - \mu(e) = \frac{|eval(e) \setminus G|}{|eval(e)|}$$

.

- Exhaustividad[1]: mide la proporción de elementos reales que han sido correctamente predichos. Su fórmula clásica es:

$$\mu = \frac{TP}{TP + FN}.$$

La adaptamos a nuestro problema:

Definición 2.20. Sea $G \subseteq U$ el conjunto objetivo y $e \in \mathcal{E}$ una expresión cuya evaluación $eval(e)$ constituye una aproximación de G. Definimos la exhaustividad de e como una medida M tal que:

$$\mu(e) = \frac{|G \cap eval(e)|}{|G|}$$

La medida toma valores en $[0, 1]$, siendo 1 el valor de una cobertura completa del conjunto objetivo y 0 cuando ningún elemento de G ha sido cubierto.

Está directamente relacionada con la infracobertura, pues cuanto mayor es la exhaustividad, menor es la infracobertura de G, es decir, el número de elementos de G que quedan sin cubrir por la aproximación. Al igual que en el caso anterior, la infracobertura es absoluta ($|G \setminus \tilde{G}| = FN$), mientras que la exhaustividad es relativa, al normalizar respecto al tamaño de G:

$$1 - \mu(e) = \frac{|G \setminus eval(e)|}{|G|}.$$

- Especificidad[2]: mide la proporción de negativos correctamente identificados como tales. Su fórmula clásica es:

$$\mu = \frac{TN}{TN + FP}.$$

La definimos adaptada a nuestro problema:

Definición 2.21. Sea $G \subseteq U$ el conjunto objetivo y $e \in \mathcal{E}$ una expresión cuya evaluación $eval(e)$ constituye una aproximación de G. Definimos la especificidad de e como una medida M tal que:

$$M(e) = \frac{|U \setminus (G \cup eval(e))|}{|U \setminus G|}.$$

[1] Del inglés, *recall.*
[2] Del inglés, *specificity.*

La medida toma valores en $[0,1]$, siendo 1 el valor de una exclusión perfecta de los elementos ajenos a G y 0 cuando todos los elementos a G se incluyen erróneamente en la aproximación.

Esta medida evalúa hasta qué punto los elementos que no pertenecen al conjunto objetivo han sido correctamente dejados fuera de $\tilde{G}$. Por ello, no siempre resulta de interés por sí sola, ya que ignora lo que ocurre con los elementos que sí pertenecen a G. En muchos casos es preferible combinarla con otras métricas que reflejen tanto la inclusión como la exclusión de elementos.

- *F1-Score*: es la media armónica de precisión y exhaustividad, y una de las métricas más utilizadas para evaluar la calidad global de una clasificación binaria. Equivale al coeficiente de Dice aplicado a la tabla de contingencia, por lo que puede verse como una versión balanceada entre los aciertos positivos y los errores de omisión o inclusión. Su fórmula clásica es:

$$\mu = \frac{2TP}{2TP + FP + FN} = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}.$$

La definimos en el contexto de nuestro problema:

Definición 2.22. Sea $G \subseteq U$ el conjunto objetivo y $e \in \mathcal{E}$ una expresión cuya evaluación $eval(e)$ constituye una aproximación de G. Definimos el *F1-Score* de e como una medida M tal que:

$$M(e) = \frac{2|G \cap eval(e)|}{2|G \cap eval(e)| + |eval(e) \setminus G| + |G \setminus eval(e)|}.$$

La medida toma valores en $[0,1]$, siendo 1 cuando la aproximación coincide exactamente con el conjunto objetivo y 0 cuando no existe intersección entre ambos.

En la práctica se ha prestado mucha atención al equilibrio entre la precisión y la exhaustividad [MRS08], especialmente en conjuntos de datos desbalanceados y en problemas como la detección de fraude. Aunque este análisis es más común en el ámbito del aprendizaje automático, donde se usan curvas precisión-exhaustividad para ajustar el umbral en cada iteración de entrenamiento, nos muestra su importancia en nuestro problema. No conviene despreciar una de estas dos medidas por completo y solo enfocarnos en la otra. Si tenemos una precisión alta con una exhaustividad baja, los elementos incluidos en la aproximación realmente pertenecen a G, pero se pueden estar ignorando muchos otros elementos de G. Por el contrario, si tenemos una exhaustividad alta con una precisión baja, aproximamos la mayoría de elementos de G, pero estamos incluyendo también muchos elementos de fuera de G. Esto nos muestra lo necesario que es considerar ambas medidas conjuntamente, según nuestro objetivo.

Así surge el *F1-Score*, que merece la pena analizar con mayor detalle. Esta medida aparece por primera vez en el libro *Information Retrieval de Van Rijsbergen*, en 1979 [CHK23]. Antes de llegar a su fórmula definitiva se plantean distintas alternativas, aunque esta medida pronto se consolidó y empezó a utilizarse también en otros ámbitos de la informática, como la extracción de información y la clasificación de textos, convirtiéndose en una de las más

conocidas y utilizadas. En cuanto a su naturaleza, el *F1-Score* ha generado cierto debate sobre si debe considerarse de asociación (simétrica) o direccional. En la mayoría de estudios se presenta como una medida direccional [HST+22], pues es la media armónica de precisión y exhaustividad, ambas medidas direccionales. Ahora bien, se trata de una medida simétrica, cuyo resultado no cambia dependiendo de qué clase se defina como positiva y negativa. Es decir, es invariante a intercambiar el conjunto objetivo G con la aproximación $\tilde{G}$. Además, no depende de TN, sino que mide el parecido que hay entre G y $\tilde{G}$ sin tener en cuenta el resto del universo. De ahí su paralelismo con el coeficiente Dice, y su posible interpretación como medida asociativa.

Volviendo a las medidas direccionales presentadas, al igual que ocurría con las medidas asociativas, comparan el conjunto objetivo con las potenciales soluciones [HM15]. Todas estas pueden expresarse de forma natural como probabilidades condicionadas calculadas a partir de la Tabla 2.3. Por ejemplo, la precisión corresponde a $P(G|\tilde{G})$, mientras que la sensibilidad es $P(\tilde{G}|G)$. Esto conecta directamente con la literatura de clasificación binaria y de tests diagnósticos, donde las medidas que hemos mencionado se definen en los mismos términos. En los campos como la medicina y epidemiología, se utilizan al definir la calidad de un test diagnóstico en los que se clasifican pacientes sanos y enfermos, de ahí la gran relación de nuestro problema con la biología computacional, como ya mencionamos en el apartado de aplicaciones e impacto.

2.2.3. Otras medidas

Dentro de esta sección incluimos algunas medidas que no son ni asociativas ni direccionales. No se trata de métricas estandarizadas en la literatura como lo son las mencionadas hasta ahora, sino que las definimos a partir de propiedades observables en nuestro problema. Estas evalúan diferentes criterios:

- Medidas de tamaño: evalúan el cardinal de un conjunto o de un grupo de subconjuntos.
  - Tamaño local: evalúa el número de elementos contenidos en cada subconjunto empleado en la construcción de la aproximación. Lo definimos para nuestro problema:

    Definición 2.23 (Tamaño local en el problema de aproximación de conjuntos)**.** Sea $e \in \mathcal{E}$ una expresión y $\mathcal{F}(e)$ el conjunto de subconjuntos de F utilizados en su construcción. Para cada $F_i \in \mathcal{F}(e)$, definimos su tamaño local como una medida M tal que:

    $$M(e) = |F_i|.$$

    Esta medida cuantifica el cardinal del subconjunto F_i y permite penalizar o priorizar ciertos subconjuntos según su tamaño. Puede utilizarse, por ejemplo, para favorecer subconjuntos más grandes (que cubran más elementos) o para reducir el número total de subconjuntos empleados en la aproximación.

  - Tamaño de la aproximación: esta medida nos da el cardinal de la aproximación final, o bien el número de subconjuntos utilizados. Es útil en casos en los que cada elemento o cada nuevo subconjunto aumente el coste total, y queramos minimizarlo. Damos su definición de acuerdo con nuestro problema:

Definición 2.24 (Tamaño de la aproximación en el problema de aproximación de conjuntos)**.** Sea $e \in \mathcal{E}$ una expresión. Definimos el tamaño de la aproximación de e como una medida M tal que:

$$M(e) = |eval(e)|,$$

o, alternativamente, como el número de subconjuntos utilizados en su construcción:

$$M(e) = |\mathcal{F}(e)|.$$

Resultan especialmente útiles en escenarios donde el tamaño de la solución afecta al coste o la complejidad del modelo.

- Medidas ponderadas: asignan a cada conjunto $F_i \in \mathcal{F}(e)$ un peso o coste $w_i \in \mathbb{R}_0^+$, que puede representar, por ejemplo, la dificultad, el riesgo o el coste asociado a su uso. Estas medidas permiten modelar escenarios más realistas, en los que los subconjuntos no son igualmente accesibles. Damos su definición para nuestro problema:

  Definición 2.25 (Medida ponderada para el problema de aproximación de conjuntos)**.** Sea $e \in \mathcal{E}$ una expresión y $\mathcal{F}(e)$ el conjunto de subconjuntos empleados en su construcción. Definimos la medida ponderada de e como una medida M tal que:

  $$M(e) = \sum_{F_i \in \mathcal{F}(e)} w_i, \qquad w_i \in \mathbb{R}_0^+.$$

  Equivalentemente, puede verse como una medida local sobre cada subconjunto:

  $$M_i(e) = w_i, \qquad M(e) = \sum_{F_i \in \mathcal{F}(e)} M_i(e).$$

  Esta medida refleja el coste total de la expresión e.

- Medidas globales: son medidas que, si bien no se ven afectadas por cuál se toma como clase de referencia, no entran dentro de la categoría de medidas asociativas porque no miden directamente el grado de parecido entre los dos conjuntos, sino si coinciden o discrepan respecto a todo el universo. Se trata, por tanto, de medidas simétricas de carácter externo, porque trabajan sobre los conjuntos finales completos.
  - Tasa de error[3]: mide la proporción de elementos mal clasificados sobre el total. Es una medida muy conocida y extendida en estadística y en el ámbito de aprendizaje automático. Su fórmula clásica podemos encontrarla en [Tin10], y es la siguiente:

    $$\mu = \frac{FP + FN}{TP + FP + FN + TN}.$$

    La definimos en el contexto de nuestro problema:

    Definición 2.26 (Tasa de error en el problema de aproximación de conjuntos)**.** Sea $G \subseteq U$ el conjunto objetivo y $e \in \mathcal{E}$ una expresión cuya evaluación $eval(e)$

[3]Del inglés, *error rate.*

constituye una aproximación a G. Definimos la tasa de error de e como una medida M tal que:

$$M(e) = \frac{|eval(e) \setminus G| + |G \setminus eval(e)|}{|U|}.$$

En nuestro problema, mide los elementos que están o bien en G o bien en $\tilde{G}$ pero no en ambos ni en ninguno. Da una visión del número de elementos mal aproximados, independientemente del tipo de error cometido. Toma valores en el intervalo $[0, 1]$, siendo 0 una aproximación perfecta y 1 un desacuerdo total entre G y $eval(e)$.

- Exactitud[4]: es la proporción de elementos clasificados correctamente sobre el total. En [MRS08] podemos encontrar su fórmula clásica es:

$$\mu = \frac{TP + TN}{TP + FP + FN + TN} = 1 - \text{tasa de error.}$$

La definimos en el contexto de nuestro problema:

Definición 2.27 (Exactitud para el problema de aproximación de conjuntos)**.** Sea $G \subseteq U$ el conjunto objetivo y $e \in \mathcal{E}$ una expresión cuya evaluación $eval(e) \subseteq U$ constituye una aproximación a G. Definimos la exactitud de e como una medida M tal que:

$$M(e) = \frac{|G \cap eval(e)| + |U \setminus (G \cup eval(e))|}{|U|}.$$

En nuestro problema, mide cuántos de los elementos de todo el universo (ya sean de dentro de G o de $U \setminus G$) han sido correctamente aproximados. Toma valores en $[0, 1]$, siendo 0 la coincidencia perfecta entre G y $eval(e)$, y 0 la discrepancia total. Es fácil ver que la exactitud y la tasa de error son medidas complementarias.

- Número de operaciones: cuantifica la complejidad estructural de una aproximación en términos del número de operaciones entre conjuntos que han sido empleadas para construirla. No evalúa similitud ni relación direccional entre G y $\tilde{G}$, sino la simplicidad o complejidad de la expresión generada. Damos su definición para nuestro problema:

Definición 2.28 (Número de operaciones en el problema de aproximación de conjuntos)**.** Sea $e \in \mathcal{E}$ una expresión. Denotamos por $\mathcal{O}p(e)$ al conjunto de operaciones utilizadas en la construcción de e. Definimos el número de operaciones de e como una medida M tal que:

$$M(e) = |\mathcal{O}p(e)|.$$

Valores bajos de esta medida indicarn expresiones simples mientras que valores altos corresponden a construcciones más complejas.

Cabe destacar que tanto la exactitud como la especificidad dependen de los verdaderos negativos (TN), es decir, los elementos que no pertenecen ni a G ni a su aproximación, $\tilde{G}$. En

[4]Del inglés, *accuracy.*

campos como el diagnóstico médico, donde los negativos son tan relevantes como los positivos, este tipo de medidas son cruciales. En nuestro problema, sin embargo, los TN son los elementos de $(U \setminus G) \cap (U \setminus \tilde{G})$, que suelen ser muy numerosos y aportan poca información sobre la aproximación. Por ello, muchas métricas (como el *F1-Score* o Jaccard) los ignoran por completo.

Habiendo visto diferentes ejemplos de medidas, señalamos que siempre existe la posibilidad de definir nuevas medidas según nuestros intereses particulares. Una forma natural consiste en combinar varias de las medidas mencionadas aplicadas a una misma expresión e:

$$\mu(e) = w_1\mu_1(e) + w_2\mu_2(e) + \cdots + w_n\mu_n(e),$$

donde los coeficientes $w_i \in \mathbb{R}^+$ actúan como pesos que reflejan la relevancia de cada una de las medidas parciales μ_i. Este enfoque permite equilibrar distintos criterios (por ejemplo, precisión, cobertura o coste) dentro de una única función de evaluación.

En situaciones más complejas en las que nos interesa medir propiedades de distintas expresiones, podemos recurrir a una medida multidimensional. Por ejemplo, para dos expresiones relevantes e_1 y e_2:

$$\mu(e_1, e_2) = w_1\gamma_1(e_1) + w_2\delta_1(e_2) + \ldots,$$

donde las γ_i son medidas sobre e_1 y las δ_i son medidas sobre e_2. Esta idea es extrapolable a dimensiones superiores, permitiendo construir medidas más flexibles.

2.2.4. Restricción vs. Optimización

Como hemos mencionado, en la teoría de nuestro problema de aproximación de conjuntos, las medidas pueden desempeñar tres papeles en cuanto a su uso. Por un lado, pueden actuar como restricción: fijamos un umbral (mínimo o máximo aceptable) y, por ejemplo, descartamos todas las soluciones que no lo respeten. Por otro lado, pueden servir como objetivo de optimización, es decir, buscamos maximizar o minimizar su valor dentro del espacio de soluciones. Esta dualidad es muy común en problemas de optimización combinatoria, donde se busca hallar la mejor solución posible en un espacio discreto. Por último, sirven como heurísticas dentro de ciertos algoritmos, como veremos en el capítulo informático de este trabajo.

Algunas de las medidas presentadas se utilizan preferentemente en una de estas modalidades. Ciertas medidas tienen un significado natural como restricciones, como ocurre con el número de operaciones utilizadas. En este caso, resulta más adecuado fijar un máximo de operaciones permitidas para evitar que se dispare la complejidad computacional o el tiempo de ejecución, como veremos en la siguiente subsección. Así, las aproximaciones que excedan dicho umbral podrían, por ejemplo, ser automáticamente descartadas. En cambio, no resulta tan útil utilizar esta medida como criterio de optimización, pues si buscamos minimizar el número de operaciones, obtendríamos trivialmente el valor cero; mientras que si maximizarlo conduciría a expresiones más largas y complejas, sin aportar una mejora real en la calidad de la aproximación.

Las medidas con carácter restrictivo juegan un papel crucial en nuestro problema, pues permiten implementar las restricciones estructurales asociadas a cada particularización. Por ejemplo, pueden emplearse para restringir las operaciones disponibles en los casos de semianillo o anillo. Dentro de este grupo es habitual distinguir entre restricciones duras[5] y blandas[6], siguiendo la terminología de la programación lineal [Ken75]. Las duras son las restricciones estructurales del problema, que han de cumplirse necesariamente, mientras que las blandas son restricciones deseables pero no estrictamente obligatorias, que pueden relajarse si el coste de su cumplimiento es excesivo.

Por el contrario, el resto de medidas que hemos presentado tienen más sentido como criterios de optimización. Por ejemplo, buscamos maximizar el índice de Jaccard, el coeficiente de Dice, la precisión o la exhaustividad, o bien minimizar la redundancia y la tasa de error.

De cualquier manera, la decisión de tratar una medida para imponer una restricción o como función de calidad a optimizar afecta directamente al espacio de soluciones aceptables $\mathcal{E}^{\mathcal{C}}$, determinando su tamaño, estructura y propiedades.

2.3. Complejidad computacional

Una vez que hemos definido las medidas, sus distintos tipos e interpretaciones, y hemos presentado diferentes ejemplos y aplicaciones, terminamos la parte matemática de nuestro trabajo enfocándonos en analizar el coste de calcularlas y en la complejidad de nuestro problema en conjunto.

La teoría de la computación surge en el siglo XX a partir de cuestiones fundamentales sobre los límites del cálculo y las matemáticas. En los años veinte, Hilbert planteó su programa de formalización matemática, que fue posteriormente limitado por los teoremas de incompletitud de Gödel. Estos resultados llevaron a investigadores como Turing y Church a preguntarse qué problemas podían resolverse de forma algorítmica [Zac06].

En este contexto, Alan Turing introdujo el concepto de Máquina de Turing, un modelo teórico que formaliza la noción de cálculo. Se trata de un autómata con una cinta infinita y un cabezal de lectura/escritura que, siguiendo un conjunto finito de reglas, permite describir cualquier procedimiento computacional [Tur37]. Este modelo sentó las bases de la computabilidad moderna y sigue siendo la referencia fundamental para definir qué significa que un problema sea resoluble mediante un algoritmo.

Con el desarrollo de los ordenadores, la atención se trasladó hacia la eficiencia en la resolución de problemas [FH03]. Esto dio lugar a la teoría de la complejidad computacional, donde se definieron clases como P y NP, para clasificar problemas en función de los recursos necesarios para resolverlos.

Por un lado tenemos la complejidad espacial que se refiere a la cantidad de memoria que requiere un algoritmo. Por otro tenemos la complejidad temporal, que es el tiempo de eje-

[5] Del inglés, *hard*.
[6] Del inglés, *soft*.

cución necesario de dicho algoritmo. Sin embargo, veremos que ambas están estrechamente relacionadas.

2.3.1. Complejidad temporal y notación asintótica

Aún cuando un problema sea resoluble computacionalmente en la teoría, puede no serlo en la práctica si el tiempo o la memoria requeridos resultan excesivos. Muchas veces es difícil calcular el tiempo exacto que tarda un algoritmo en ejecutarse, por lo que normalmente se recurre a estimaciones asintóticas [Sip96]. En particular, nos interesa el crecimiento de la función de tiempo de ejecución en el peor caso, es decir, el término de mayor orden cuando el tamaño de la entrada tiende a infinito. Para expresarlo, utilizamos la notación conocida como "big-O" que podemos encontrar en [Sip96]:

Definición 2.29 (Big-O)**.** Sean f y g funciones tales que $f, g : \mathbb{N} \to \mathbb{R}^+$. Decimos que $f(n) = O(g(n))$ si existen constantes positivas c, n_0 tales que para cada $n \in \mathbb{N}$, con $n \geq n_0$, $f(n) \leq cg(n)$. En ese caso, $g(n)$ actúa como cota superior asintótica para $f(n)$.

Mientras que $O(g(n))$ indica que una función $f(n)$ no crece más rápido que $g(n)$ salvo constantes, existe una notación más restrictiva que expresa que $f(n)$ crece estrictamente más despacio que $g(n)$ cuando $n \to \infty$, y que también podemos encontrar en [Sip96]:

Definición 2.30 (Small-o)**.** Sean f y g funciones tales que $f, g : \mathbb{N} \to \mathbb{R}^+$. Decimos que $f(n) = o(g(n))$ si para todo $c \in \mathbb{R}^+$, existe un n_0, tal que $f(n) < cg(n) \quad \forall n \geq n_0$. En este caso, $g(n)$ es una cota estricta para $f(n)$.

Veamos los órdenes de algunas de las medidas que mencionamos en la sección anterior, así como de algunas operaciones necesarias en nuestro problema:

La medida de tamaño local es un gran ejemplo de orden constante, $O(1)$, pues basta con leer el contador de elementos $|F_i|$, independientemente del tamaño de la entrada. Ahora bien, si tenemos una solución $\tilde{G}$ y el conjunto objetivo G, y queremos comprobar si un elemento de la solución está en el objetivo, podemos usar dos estrategias distintas: una búsqueda binaria en una estructura ordenada, con coste logarítmico, $O(log(|G|))$; o recorrer todos los elementos de G, con orden lineal, $O(|G|)$. Si repetimos este procedimiento para cada elemento de $\tilde{G}$, el coste asciende a $O(|\tilde{G}| \cdot log|G|)$ y $O(|\tilde{G}| \cdot |G|) \approx O(|G|^2)$ respectivamente. Finalmente, para un orden exponencial, $O(2^n)$, basta con calcular todas las posibles combinaciones de subconjuntos de F y ver si cubren G. Esto corresponde a una búsqueda exhaustiva, que veremos en la parte informática.

Es claro en estos ejemplos abstractos la importancia de estudiar y comparar en detalle las estructuras de datos que podemos utilizar en nuestro problema, así como las operaciones asociadas. La elección adecuada puede marcar la diferencia entre una solución ineficiente, y otra mucho más rápida. Más adelante, en la parte informática, se analizará la complejidad concreta de los algoritmos que implementaremos para resolver el problema.

Utilizamos los siguientes cuatro gráficos para presentar todo esto de forma más visual. En ellos analizamos cómo cambia el tiempo de ejecución de un algoritmo/programa/instrucciones de distintos órdenes, según el tamaño de entrada. Lo hacemos para distintos tamaños de n en cada gráfico.

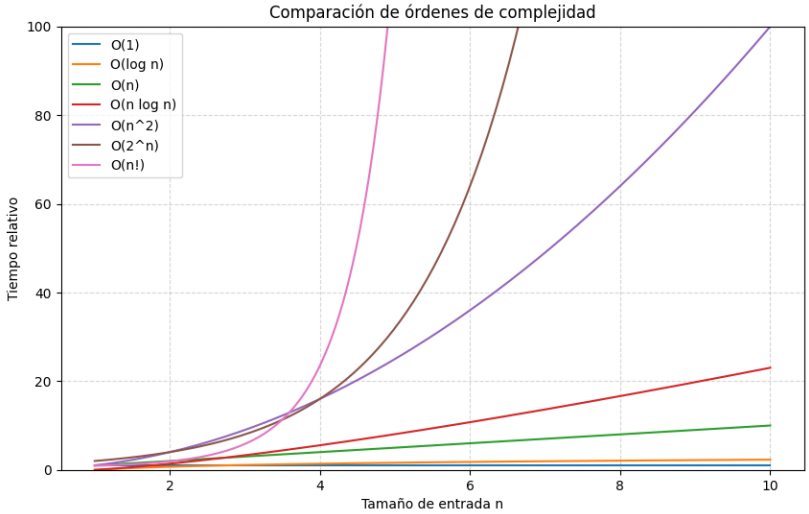

Figura 2.2.: Comparación de órdenes de complejidad para n hasta 10 (escala lineal).

En la Figura 2.2, centrada en tamaños muy pequeños de entrada, vemos que incluso con n tan reducidos, los órdenes factorial y exponencial empiezan a crecer rápidamente. A partir de $n = 4$, $O(n!)$ supera drásticamente al resto, y a partir de $n = 6$, el orden $O(2^n)$ también se dispara. Los órdenes polinomiales como $O(n^2)$, resultan mejorables en este rango, en comparación con los lineales y logarítmicos, que practicamente no se distinguen entre sí. Con este gráfico vemos que, aún para entradas pequeñas, los algoritmos de complejidad exponencial o factorial dejan de ser prácticos.

Rescatamos nuestro ejemplo base 2.2 donde teníamos $|F| = 9, |G| = 5, |U| = 10$. Si quisiéramos comprobar si un elemento de $\tilde{G}$ está en el objetivo, utilizando una estructura ordenada, que conlleva un orden $O(log|G|) = O(log(5))$, vemos en el gráfico que el tiempo es prácticamente el mismo que en el caso constante. Por su lado, si recorremos los 5 elementos de G, con orden $O(5)$, vemos que el tiempo crece, separándose del orden logarítmico, aún siendo un ejemplo con conjuntos muy pequeños.

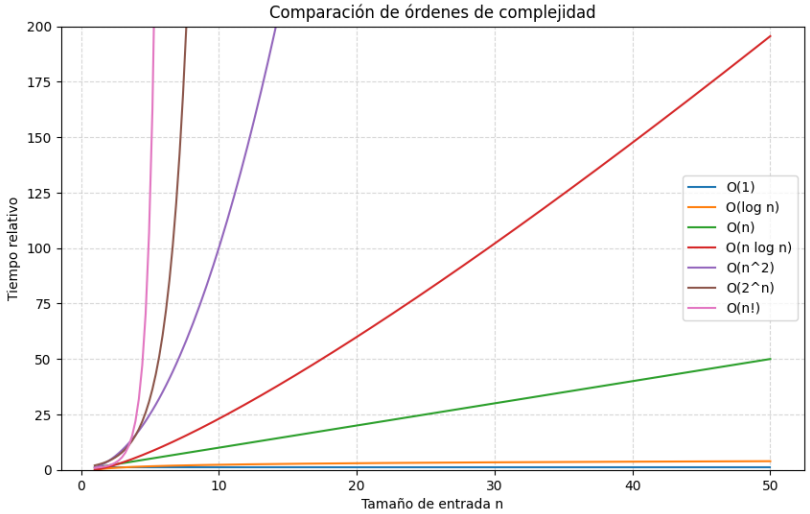

Figura 2.3.: Comparación de órdenes de complejidad para n hasta 50 (escala lineal).

Si ampliamos el rango hasta $n = 50$, observamos en la Figura 2.3 cómo los órdenes cuadráticos se evidencian como poco eficientes en comparación con los lineales y logarítmicos. El orden $O(nlogn)$ comienza a separarse de manera clara, poniéndose en una posición intermedia: más costoso que los lineales, pero aún muy inferior a los exponenciales. Este gráfico muestra cómo, a medida que aumenta el tamaño de entrada, incluso órdenes considerados razonables, pueden llegar a ser limitantes.

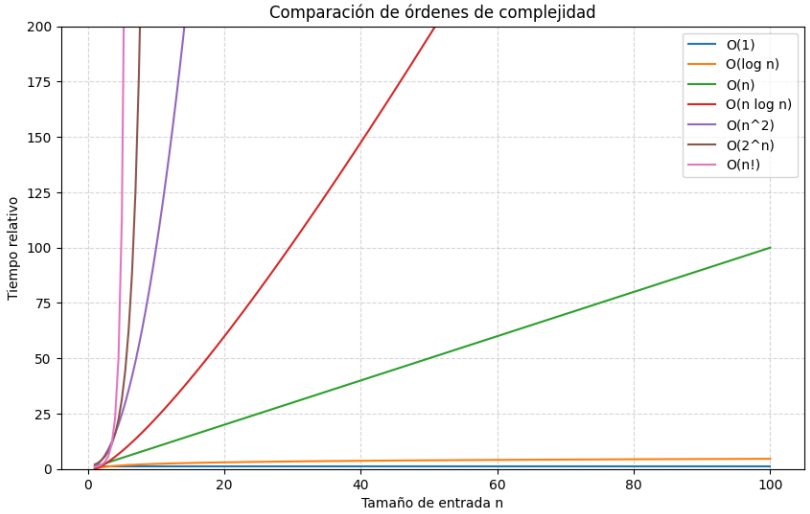

Figura 2.4.: Comparación de órdenes de complejidad para n hasta 100 (escala lineal).

Si analizamos tamaños hasta $n = 100$ en la Figura 2.4, la diferencia entre los órdenes es aún más evidente. Los factoriales, exponenciales y cuadráticos crecen tanto que dejan de ser representativos en la escala lineal, lo cual confirma su inviabilidad en la práctica. El orden

$O(nlogn)$, aunque también se dispara, todavía se mantiene en un rango más asumible. Los órdenes lineal y logarítmico continúan siendo los únicos realmente escalables. En este gráfico vemos por qué los algoritmos cuadráticos, aunque más realistas que los exponenciales, pueden convertirse en un cuello de botella con entradas de tamaño medio.

Dado que nuestro problema surge de un caso real en el que se analizan coberturas de conjuntos de tamaño muy grande (miles o incluso millones de elementos), nos interesa estudiar qué ocurre en dichos casos. Para ello, en la Figura 2.5 utilizamos una escala logarítmica que nos permite representar valores de n mucho mayores. Omitimos los órdenes factorial y exponencial, por su desproporción:

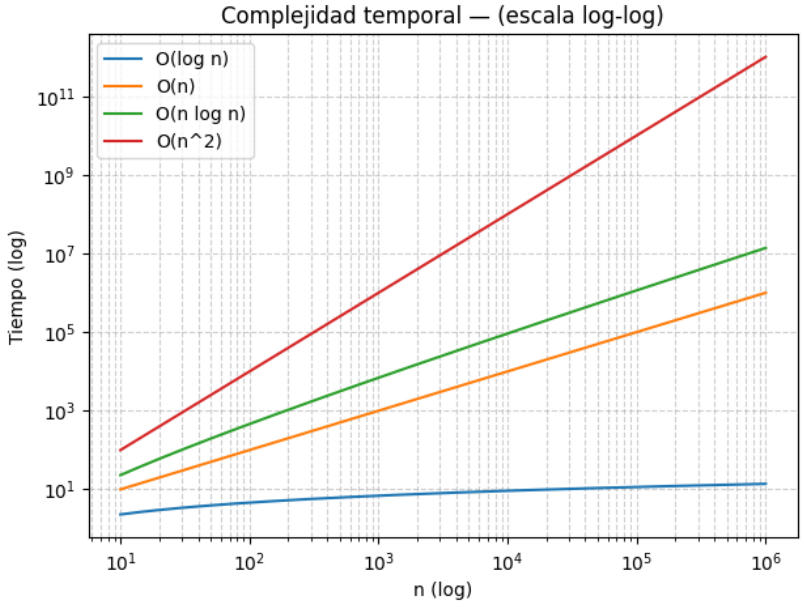

Figura 2.5.: Comparación de órdenes polinomiales en rango grande de n (escala log–log).

La escala logarítmica nos permite apreciar cómo los órdenes polinomiales se convierten en rectas con distinta pendiente. Aunque al principio parecen próximos, a medida que n crece la distancia entre ellos aumenta. Por ejemplo, para $n = 1000$, un algoritmo lineal puede requerir en torno a 10^3 pasos, uno $O(nlogn)$ en torno a 10^4, y uno cuadrático alrededor de 10^6. El orden $O(logn)$ se mantiene alrededor de 10 pasos para entradas de tamaño hasta $n = 10^5$, resultando mucho inferior que los otros órdenes.

A parte de los distintos órdenes temporales que hemos analizado y ejemplificado, de forma más en general se define la clase de complejidad temporal $TIME(t(n))$ como el conjunto de todos los problemas decidibles en tiempo $O(t(n))$ por una máquina de Turing. A partir de aquí se derivan las conocidas clases de complejidad P y NP.

2.3.2. Clases de complejidad y NP-Completitud

Las siguientes definiciones de clases P y NP podemos encontrarlas en [Sip96].

Definición 2.31 (Clase P)**.** Es la clase de todos los problemas de decisión resolubles por máquinas de Turing deterministas que trabajan en un tiempo polinomial. Siguiendo con la notación que veníamos usando:

$$P = \bigcup_k TIME(n^k)$$

De forma más intuitiva, son todos los problemas cuya solución se puede encontrar rápidamente. Un ejemplo es la multiplicación de matrices, de orden $O(n^3)$, o buscar un elemento en un vector, de orden $O(n)$.

Antes de presentar la siguiente clase recordamos que, mientras que una Máquina de Turing determinista solo tiene una trayectoria posible en cada paso de la ejecución, una Máquina no determinista, es aquella que puede tener varias transiciones posibles.

Definición 2.32 (Clase NP)**.** Es la clase de todos los problemas de decisión resolubles por máquinas de Turing no deterministas que trabajan en un tiempo polinomial.

De forma más intuitiva, son todos los problemas cuya solución se puede verificar en tiempo polinómico. Uno de los ejemplos más conocidos es el problema del viajante de comercio. En su versión de optimización: dado un conjunto de ciudades y las distancias entre ellas, encontrar el camino que visite cada ciudad una, y solo una, vez, regrese al origen y minimice el coste total. La versión de decisión pregunta si existe un camino de coste $\leq k$ constante. En la Figura 2.6 vemos un ejemplo de grafo asociado a este problema:

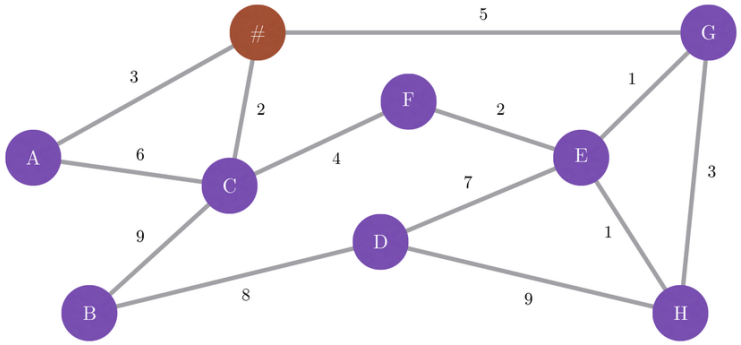

Figura 2.6.: Ejemplo de grafo de TSP. Fuente: [Bre13].

De forma intuitiva, decimos que un problema A se reduce a otro problema B cuando una solución para B puede ser utilizada para obtener una solución para A [Sip96]. Para ello se utiliza una función de reducción intermedia que convierte las instancias de A en instancias de

B, de forma que resolver B implica automáticamente haber resuelto A. Se dice que un problema es NP-Hard, o NP-Duro, si todo problema de NP se reduce polinómicamente a él [Sip96].

Con esto podemos presentar el concepto de NP-Completitud, disponible en [Sip96]:

Definición 2.33 (NP-Completo)**.** Un problema es NP-Completo si pertenece a la clase NP y además todos los otros problemas de NP son reducibles a él. Es decir, pertenece a NP y es NP-Hard.

En 1971, Stephen Cook demostró con el Teorema de Cook, que ciertos problemas, como el de satisfacibilidad booleana (SAT), son NP-completos, introduciendo la noción de reducibilidad polinómica. Un año después, el informático Richard Karp, publicó un artículo que contenía un con otros 21 problemas que también son NP-Completos. Esto supuso un gran avance, demostrando que eran equivalentes entre ellos y que lo que Cook había encontrado no era un problema puntual, sino parte de una clase mayor que permitiría resolver muchos otros problemas relevantes [Kar72]. Este marco constituye la base del análisis moderno de complejidad y motiva el estudio de problemas como la cobertura de conjuntos, aunque no fue reconocido inmediatamente tras su publicación. Además, ayudó a plantear el famoso problema P vs NP. Este, aún sin resolver, es uno de los grandes desafíos de la teoría de la complejidad y ha sido reconocido como uno de los Problemas del Milenio por el *Clay Mathematics Institute*.

La cuestión tiene distintas interpretaciones, asociadas a las diferentes definiciones de ambas clases, que son equivalentes: desde el punto de vista teórico, se trata de comparar lo que una Máquina de Turing no determinista puede resolver rápidamente con lo que una determinista puede hacer; desde el punto de vista práctico, se quiere comprobar si la capacidad de verificar soluciones rápidamente implica también la capacidad de encontrarlas con la misma eficiencia [Sip96].

Es bien conocido, y no nos detendremos en su demostración (pues no es directamente útil a nuestro problema), que $P \subseteq NP$, pues toda máquina determinista es también no determinista, siendo la inclusión trivial. Sin embargo, a lo que aún no hay respuesta es a si dicha inclusión es estricta, cosa que tendría consecuencias profundas en múltiples campos. Si $P = NP$, muchos problemas considerados intratables podrían resolverse de manera eficiente, lo que transformaría áreas como la inteligencia artificial, la optimización de recursos o la biología computacional. Un ejemplo más concreto e interesante, son los sistemas como RSA, que se basa en la dificultad de ciertos problemas, como la factorización de enteros grandes, que se consideran intratables en la práctica [ABo9]. Si se demostrase que $P = NP$, se pondría en riesgo la seguridad de estos sistemas, al hacerlos resolubles de forma eficiente.

Además, la frontera entre las clases P y NP, no está del todo clara y hay problemas que han pasado de una clase a otra cuando se ha descubierto un algoritmo que lo resolvía eficientemente. Es el caso del problema de demostrar si un número es primo. Hasta 2002, se consideraba que este problema pertenecía a la clase NP, pero tres científicos indios desarrollaron un algoritmo que demostró que el problema también pertenece a la clase P. Este cambio evidencia la importancia de encontrar algoritmos más eficientes, incluso para problemas que parecen intratables.

2.3.3. Complejidad teórica de nuestro problema

Nuestro problema es muy general y flexible, por lo que no resulta adecuado analizar su complejidad directamente, sino que conviene hacerlo sobre sus distintas variantes, donde podemos hablar en términos de medidas y objetivos concretos. No obstante, haremos un razonamiento general, que abarque todas sus variantes, sin perder generalidad y analizando la complejidad del problema base.

Para estudiar nuestro problema en términos de las clases P y NP, hace falta convertirlo en un problema de decisión que tenga el mismo grado de dificultad, cosa que siempre es posible [GJ90]. Esto ocurre de forma natural cuando la formulación incluye un umbral sobre una medida con carácter restrictivo. Por ejemplo, en la variante de Recubrimiento de G, la cuestión es: ¿es posible encontrar una familia de subconjuntos $F' \subseteq F$ cuya unión forme un recubrimiento de G? Cuando por el contrario se trate de optimizar una medida, la transformación en problema de decisión la conseguimos imponiendo también un umbral k sobre la medida en cuestión. Por ejemplo, la variante sin repetición: ¿es posible encontrar una aproximación de G, sin repetir subconjuntos de F, cuya precisión sea al menos 0.8?

Recordamos el problema del Set Cover que ya mencionamos en la introducción. En su versión de decisión consiste en responder a lo siguiente: dado un conjunto universo U, una colección de m subconjuntos $S_i \subseteq U$, y un entero k, ¿es posible encontrar una familia C de como máximo k de estos subconjuntos S_i, tal que al hacer su unión, cubran todo U? [KT05]. Actualmente se considera un problema NP-Completo, por lo que no se conoce un algoritmo polinómico que lo resuelva en todos los casos. Es por esto que en la práctica, se emplean algoritmos aproximados y heurísticos para su resolución. Pasa igual con el Exact Cover Problem, que también se ha demostrado como NP-Completo.

Nuestro problema puede entenderse como una generalización de ambos. A diferencia de Set Cover o Exact Cover, no buscamos exclusivamente una partición ni un recubrimiento exacto, sino que definimos el objetivo en términos de una medida de aproximación más flexible. En algunas de nuestras variantes, sin embargo, sí que es posible formular de manera explícita estas condiciones, de tal manera que tanto el Exact Cover Problem como el Set Cover Problem se pueden formular como variantes concretas de nuestro problema.

Como estas dos variantes son NP-Completas, es claro que nuestro problema es NP-Duro, pues cualquier generalización de un problema NP-Duro hereda su complejidad. En particular, el problema Set Cover se reduce al nuestro, ya que cualquier instancia de Set Cover puede formularse como una instancia válida de nuestro problema simplemente restringiendo los operadores a la unión ($\cup$) y definiendo la medida de evaluación como la cobertura exacta. Por tanto, si nuestro problema pudiese resolverse en tiempo polinómico, también se podría resolver el Set Cover, lo cual contradice su pertenencia a NP.

Queda por analizar si nuestro problema, además, pertenece a la clase NP, para determinar si puede considerarse NP-Completo. Es decir, queremos ver si es posible comprobar una solución candidata en tiempo polinómico. En nuestro caso, la entrada al verificador son la expresión candidata e, el conjunto objetivo G, el universo U y la familia F. Primero hemos de calcular el conjunto resultante $\tilde{G} = eval(e)$. Dado que la expresión tiene una longitud finita acotada por el tamaño de la entrada, esta evaluación es polinómica respecto a $|U|$ y $|F|$.

En segundo lugar, debemos calcular las medidas de $\mathcal{M}$ sobre la expresión. Todas las medidas externas de asociación y direccionales que hemos explicado en la sección anterior, dependen de la tabla de contingencia 2×2 que presentamos con anterioridad. Construir esta tabla requiere, en el peor de los casos, recorrer todos los elementos de G para cada elemento de $\tilde{G}$. Esto da lugar a una complejidad es $O(|\tilde{G}| \cdot |G|) = O(\tilde{g} \cdot g) \approx O(n^2)$, es decir, polinómica.

En cuanto a otras medidas externas que ya estudiamos, su complejidad es la siguiente:

- Tamaño de la aproximación: una única consulta al número de elementos de la estructura de datos correspondiente, con coste $O(1)$. Si la estructura no almacena directamente el tamaño, habría que recorrer el conjunto, lo que supondría un coste lineal.
- Error global y exactitud: se definen igualmente a partir de la tabla de contingencia, siendo como mucho un cálculo polinómico.

Visto todo esto, podemos concluir que la versión de decisión de nuestro problema, utilizando las medidas computables en tiempo polinómico que consideramos, pertenece a NP. Dado que además hemos demostrado su NP-dureza, afirmamos que nuestro problema es NP-Completo. Esto implica que, si nuestro problema tuviera un algoritmo que lo decidiese en tiempo polinómico en todos los casos, también lo tendrían otros como el Set Cover o el Exact Cover, lo cual equivaldría a demostrar que $P = NP$.

Al ser NP-Completo nuestro problema, surge la necesidad de recurrir a algoritmos aproximados y heurísticos para resolverlos, pues salvo que se demostrase que $P = NP$, no esperamos encontrar un algoritmo exacto en tiempo polinómico que lo resuelva en todos los casos. Por ello, en la práctica resulta más realista desarrollar métodos que aporten soluciones suficientemente buenas en un tiempo razonable, aunque no siempre sean óptimas.

2.3.4. Complejidad espacial

Aunque en nuestro problema nuestro foco principal es la complejidad temporal, es también conveniente entender como afecta a la memoria necesaria en escenarios especialmente grandes, para poder estudiar aproximaciones viables. Como ya anticipamos, hay una estrecha relación entre las clases de complejidad espacial y temporal que veremos a continuación.

Se define $SPACE(t(n))$ como el conjunto de problemas que pueden resolverse en espacio $O(t(n))$ por una Máquina de Turing determinista [AB09]. Más concretamente, para cada entrada x, el número total de celdas de la cinta que en algún momento han tenido algún símbolo durante la ejecución, es como mucho $c \cdot t(|x|)$, con $c > 0$ constante. De la misma manera, existe una clase $PSPACE$ definida como el conjunto de problemas que pueden resolverse utilizando espacio polinómico:

$$PSPACE = \bigcup_k SPACE(n^k)$$

En cuanto a su relación con las clases temporales, se tiene la siguiente cadena de inclusiones, que no nos detendremos a demostrar [ABo9]:

$$P \subseteq NP \subseteq PSPACE \subseteq EXPTIME = \bigcup_k TIME(2^{n^k}).$$

Estas inclusiones implican que, como en la sección anterior vimos que nuestro problema pertenece a la clase NP, también pertenece a la clase $PSPACE$. Esto significa que puede resolverse utilizando, en el peor de los casos, un espacio polinómico en el tamaño de la entrada, lo cual es perfectamente asumible desde el punto de vista práctico.

2.3.5. Relación con SAT

Aparte de los ya mencionados como Set Cover o Exact Cover, otro problema en el que merece la pena reparar es el conocido como SAT o Problema de Satisfacibilidad Booleana. Este fue el primer problema demostrado como NP-Completo por el propio Stephen Cook en su famoso teorema. Esto significa que fue el primer problema conocido al que se pueden reducir todos los demás problemas de la clase NP, lo cual le aporta un papel central en el ámbito de la teoría de la computación.

Este problema de decisión puede encontrarse en [Sip96] y consiste en lo siguiente: dada una fórmula de lógica booleana, ¿es posible encontrar una asignación de variables de manera que la fórmula se cumpla sin inconsistencias? Es decir, para n variables, el espacio de búsqueda consta de 2^n asignaciones posibles.

Si bien nuestro problema no es exactamente SAT, y a priori puede no parecer similar, sí que existen paralelismos. Como mencionamos en la sección de estructuras algebraicas, en nuestro problema, $\mathcal{P}(U)$ es un álgebra de Boole y podemos traducir subconjuntos a proposiciones y operadores a conectores lógicos. Esto supone que las variantes de nuestro problema, podrían formularse como instancias de SAT.

Asociamos una variable proposicional x_i a cada conjunto de la familia F, de manera que si x_i es verdadero, significa que el subconjunto F_i es seleccionado en la expresión.

Añadimos F y G para seguir con el ejemplo 2.4:

$$U = \{1,2,3,4\}, \qquad G = \{1,3\}, \qquad F = \{\{1\},\{2,3\},\{3\}\}$$

Asignamos las variables proposicionales a los elementos de F: x_1 corresponde a $\{1\}$, x_2 a $\{2,3\}$ y x_3 a $\{3\}$.

Como ejemplo, formulamos la variante Exact Cover de G (particularización de nuestro problema). Imponemos lo siguiente:

1. El elemento 1 debe aparecer una y solo una vez. Cómo solo aparece en un elemento de F: $\{1\} = x_1$, forzamos su elección.

2. El elemento 3 debe aparecer una y solo una vez. Cómo aparece en dos elemento de F:

$\{3\} = x_3$ y $\{2,3\} = x_2$, exigimos incluir exactamente uno de los dos:

$$(x_3 \vee x_2) \ \wedge \ (\neg x_3 \vee \neg x_2).$$

3. Hay que asegurarse de no incluir elementos de F que estén fuera de G. En particular, de no incluir el 2:
$$(\neg x_2).$$

4. El elemento $4 \in G$ no debe ser cubierto. Como no aparece en ningún F_i, no genera ninguna restricción adicional.

Agrupando todas estas condiciones, obtenemos la siguiente fórmula proposicional para SAT:

$$\Phi = x_1 \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (\neg x_2)$$

Simplificando, esto sería equivalente a decir:

$$x_1 \wedge x_3 \wedge (\neg x_2)$$

que corresponde a seleccionar exactamente $\{1\}$ y $\{3\}$, cuya unión es G y sin cubrir nada fuera de G.

En este caso, resolver esta instancia de SAT que hemos construido equivale a decidir la existencia de una solución de Exact Cover para G con los subconjuntos de F, de manera que los subconjuntos utilizados en la cobertura de G son precisamente aquellos cuyas proposiciones toman el valor de verdad en la versión SAT.

De forma análoga, podemos formular la variante Set Cover reformulando sin exigir que los elementos estén presentes en un único subconjunto. En el ejemplo anterior, la condición del punto 2 pasaría a ser la siguiente:
$$(x_2 \vee x_3)$$
Por la simplicidad de este ejemplo, la fórmula proposicional resultante es la misma que en el caso de Exact Cover, pero no siempre tiene por qué serlo.

Comparar SAT con variantes de nuestro porblema es especialmente interesante por dos razones. Primero, refuerza la utilidad de las condiciones estructurales que habíamos visto en la sección de estructuras algebraicas: como $\mathcal{P}(U)$ es álgebra de Boole, se pude conmutar fácilmente entre operaciones de conjuntos y operaciones lógicas. Por otro lado, esta equivalencia nos ayuda a en el estudio de la complejidad de nuestro problema, a través de reducciones, como lo hicimos a partir del Exact Cover o Set Cover Problem. Además de todo esto, añade otra conexión entre nuestro problema y problemas centrales en el mundo de la teoría de la computación, siendo las soluciones de uno soluciones también del otro.

3. Implementación algorítmica

Hasta ahora, hemos presentado y estudiado nuestro problema desde el punto de vista matemático. Ya conocemos en detalle las estructuras que intervienen, cómo se relacionan entre ellas, cómo utilizar medidas para medir la calidad de nuestras soluciones e imponer restricciones según la particularización, y cuál es el coste temporal de utilizar ciertas medidas y de resolver nuestro problema.

Es natural ahora, enfocarnos en su resolución práctica. Para ello, notamos que la formulación matemática que establecimos en la Sección 1.1, en la que buscamos encontrar una expresión e^* que cumpla las restricciones del conjunto $\mathcal{E}^{\mathcal{C}}$ y que esté en el frente de Pareto asociado $(\mathcal{E}^{\mathcal{C}})^{\mathcal{M}}$, está formalmente dentro del campo de la optimización multiobjetivo[1]. Los componentes de un problema MOP son los siguientes [Ehro5]:

- Conjunto factible: es el conjunto de elementos que satisface las restricciones del problema. En nuestro caso se corresponde con $\mathcal{E}^{\mathcal{C}}$.
- Vector de funciones objetivo: es la principal diferencia con la optimización mono-objetivo, donde la función a optimizar es escalar. En nuestro caso, el conjunto de medidas $\mathcal{M}$ actúa como el conjunto de funciones objetivo.
- Espacio objetivo: el espacio de valores donde se proyectan los resultados de nuestras medidas.
- El conjunto ordenado $(\mathbb{R}^p, \preceq)$: define la relación de orden utilizada para comparar vectores de objetivos. En nuestro caso, la dominancia de Pareto.

Por lo tanto, un problema se clasifica como MOP cuando requiere optimizar simultáneamente $k \geq 2$ funciones objetivo, cumpliendo una serie de restricciones. Cuando dichas funciones están en conflicto, no existe una única solución óptima, sino un conjunto de soluciones no dominadas. Esta correspondencia es fundamental, pues determina el tipo de estrategias algorítmicas necesarias para calcular o aproximar dicho frente de Pareto.

En el ámbito de la algorítmica y la informática, los problemas MOP se abordan normalmente con dos tipos de métodos: los métodos de agregación clásicos [CdlF22], que transforman el problema en uno mono-objetivo, usando por ejemplo ponderación; y los métodos específicos de MOP, que operan directamente en el espacio multiobjetivo, como los basados en el concepto de dominancia de Pareto. En este capítulo nos centraremos en los segundos, y adaptaremos la formulación teórica del problema a distintas aproximaciones algorítmicas que permitan la aplicación práctica de estas estrategias. Presentaremos y analizaremos tres implementaciones: una búsqueda exhaustiva con profundidad limitada, una heurística Greedy, y un algoritmo genético NSGA-II. A partir de sus pseudocódigos, estudiaremos la complejidad temporal de cada uno, discutiendo su viabilidad en distintos tamaños de instancias.

[1] Del inglés, *Multiobjective Programming, MOP.*

Desarrollaremos además un generador aleatorio de instancias parametrizable, que nos servirá para evaluar el rendimiento de los algoritmos en diversos escenarios. Este generador será capaz de producir diferentes configuraciones del problema, controlando parámetros clave como:

- $|G|$: el cardinal del conjunto objetivo G que buscamos aproximar.
- $|F|$: el cardinal de la familia generadora F.
- $|F_i|$: el cardinal de cada $F_i \in F$.

Para garantizar la diversidad y, sobre todo, la reproducibilidad de los experimentos, la generación de estas instancias se hará mediante el uso de generadores de números pseudoaleatorios basados en semillas[2]. El registro de la semilla utilizada nos permite replicar exactamente la secuencia de aleatoriedad, haciendo que cualquier experimento se pueda verificar. Además, el entorno experimental nos permite almacenar las instancias generadas en un formato estructurado, para que se puedan probar otras implementaciones algorítmicas sobre los conjuntos de datos utilizados en este trabajo. Sobre esta base, construiremos dicho entorno experimental automatizado que permita controlar parámetros, ejecutar múltiples experimentos y registrar los resultados obtenidos de forma estructurada.

Con el objetivo de validar el funcionamiento del marco desarrollado y observar el comportamiento de los algoritmos en distintos contextos, realizaremos tres experimentos ilustrativos:

- Comparación de cómo los tres algoritmos (búsqueda exhaustiva con profundidad limitada, Greedy y NSGA-II) abordan el problema, observando la estructura de las soluciones alcanzadas por cada uno, sobre un conjunto reducido de instancias generadas con distintas semillas.
- Experimento de reconstrucción en el que construiremos G a partir de F y de una expresión de referencia. El objetivo es validar la capacidad de convergencia de los algoritmos Greedy y NSGA-II en problemas para los que sabemos que es posible obtener al menos una solución exacta.
- Evaluación comparativa entre el algoritmo Greedy y el genético sobre 50 instancias generadas aleatoriamente. El objetivo de este experimento no es una evaluación exhaustiva, sino mostrar las diferencias de rendimiento entre una heurística simple y una metaheurística MOP más sofisticada en un conjunto acotado de pruebas en las que no hay garantías de que exista una solución exacta.

Finalmente, garantizaremos que el marco experimental desarrollado sea reproducible y fácilmente extensible a nuevas configuraciones, instancias o algoritmos. Esto lo conseguimos no solo con el registro de semillas, sino también documentando el entorno de ejecución (bibliotecas, versiones de software, y parámetros fijos del NSGA-II). Estas características aseguran que nuestros resultados pueden ser validados de forma independiente y permiten construir sobre nuestra base, ajustando, extendiendo o comparando nuevos métodos algorítmicos. Conseguir precisamente esto, es decir, la construcción de un laboratorio computacional riguroso adaptado a nuestro problema, es uno de los objetivos relevantes de este capítulo.

[2] Del inglés, *seeds*.

3.1. Funciones objetivo

Antes de adentrarnos en los algoritmos, es importante concretar el conjunto de medidas $\mathcal{M}$ que actúan como funciones objetivo de nuestro problema MOP. Tal y como establecimos en la Sección 2.2, estas medidas se pueden utilizar para determinar la calidad de las soluciones y definir el frente de Pareto.

En esta parte del Trabajo de Fin de Grado, el conjunto de medidas considerado, $\mathcal{M} = (M_1, M_2, M_3)$, está compuesto por:

- $M_1 \equiv$, índice de Jaccard: lo utilizamos para cuantificar la similitud entre la expresión evaluada y el conjunto objetivo G, buscando su maximización. Recordamos su fórmula:

  $$\mu = \frac{TP}{TP + FN + FP},$$

  que adaptada a nuestro problema es:

  $$M_1(e) = \frac{|G \cap eval(e)|}{|G \cup eval(e)|}.$$

- $M_2 \equiv$ número de operaciones utilizadas en la expresión: es una medida que buscamos minimizar, y que estudia la complejidad y estructura de la expresión. Recordamos su fórmula:

  $$M_2(e) = |\mathcal{O}p^*(e)|.$$

- $M_3 \equiv$ número de subconjuntos distintos de F_i utilizados en la expresión: es una medida que buscamos minimizar, y que también nos estudia la estructura de la expresión, así como la repetición de subconjuntos de F. Recordamos su fórmula:

  $$M_3(e) = |\mathcal{F}(e)|.$$

La coexistencia de estos tres objetivos (maximizar M_1, minimizar M_2 y minimizar M_3) puede generar conflicto, lo cual es típico de los problemas multiobjetivo. Por ello, buscamos encontrar el conjunto de soluciones no dominadas que conforman el frente de Pareto.

A partir de este punto, por claridad, nos referiremos a estas medidas por su notación más descriptiva: M_{Jaccard} (para M_1), $|\mathcal{O}p^*|$ (para M_2) y $|\mathcal{F}|$ (para M_3).

Sea x e y dos soluciones distintas. En la implementación decimos que x domina a y si:

$$\underbrace{M_{\text{Jaccard}}(x) \geq M_{\text{Jaccard}}(y)}_{\text{no peor en Jaccard}} \wedge \underbrace{|\mathcal{F}(x)| \leq |\mathcal{F}(y)|}_{\text{no peor en tamaño}} \wedge \underbrace{|\mathcal{O}p^*(x)| \leq |\mathcal{O}p^*(y)|}_{\text{no peor en \#ops}}$$

$$\wedge \underbrace{\left(M_{\text{Jaccard}}(x) > M_{\text{Jaccard}}(y) \vee |\mathcal{F}(x)| < |\mathcal{F}(y)| \vee |\mathcal{O}p^*(x)| < |\mathcal{O}p^*(y)|\right)}_{\text{estrictamente mejor en al menos una}}.$$

La construcción del frente completo a partir de un conjunto de $|V|$ soluciones candidatas tiene una complejidad de $O(|V|^2 \cdot 3)$ (ya que tenemos 3 medidas a optimizar). Si bien, como

veremos, la complejidad de la generación de candidatos es mayor, esta complejidad cuadrática en $|V|$ se convierte en un cuello de botella cuando los algoritmos generan un número extremadamente alto de soluciones o cuando el frente de Pareto final contiene muchas soluciones, afectando directamente al tiempo total de ejecución.

Dado que la evaluación de soluciones se hace un número muy elevado de veces, especialmente en los algoritmos heurísticos, la eficiencia temporal de las medidas que utilizamos es crítica. Por la forma en que hemos implementado las medidas, su complejidad es la siguiente:

Tabla 3.1.: Eficiencia Computacional de las Funciones Objetivo ($\mathcal{M}$)

Medida	**Complejidad Temporal**
M_{Jaccard} (Índice de Jaccard)	$O(n)$
$\lvert\mathcal{O}p^*\rvert$ (Número de operaciones)	$O(1)$
$\lvert\mathcal{F}\rvert$ (Número de F_i distintos)	$O(1)$

Si bien el índice de Jaccard (M_{Jaccard}) tiene una complejidad lineal $O(n)$, donde $n = |U|$ el tamaño del universo. Esto se debe a que para calcular el coeficiente es necesario realizar operaciones bit-a-bit (un *AND* para la intersección y un *OR* para la unión) sobre la totalidad del universo de n bits, y posteriormente contar los bits activados en ambos resultados.

Por otro lado, las medidas $|\mathcal{O}p^*|$ y $|\mathcal{F}|$ operan en un tiempo constante $O(1)$. Esto lo hemos conseguido gracias a utilizar estructuras de datos que cuentan con funciones de consulta $O(1)$, y a precalcular los valores, actualizándolos durante la construcción de la expresión y guardándolos junto a la expresión. Así podemos simplemente acceder directamente a los valores para calcular las medidas, evitando tener que recalcularlos en cada consulta.

Cabe destacar que M_{Jaccard} no se precalcula de esta forma. La razón es que $|\mathcal{O}p^*|$ y $|\mathcal{F}|$ son métricas estructurales que pueden ser compuestas de forma incremental. En cambio, M_{Jaccard} es una métrica que depende de la comparación con el conjunto objetivo G y no se va componiendo paso a paso.

3.2. Aproximaciones algorítmicas

Una vez hemos definido el marco teórico de nuestro problema como MOP, podemos proceder a presentar las aproximaciones algorítmicas escogidas.

La primera aproximación a considerar sería un método exacto mediante búsqueda exhaustiva. Sin embargo, sabemos que esa estrategia no es viable para nuestro problema, ya que el espacio de búsqueda de expresiones $\mathcal{E}$ es infinito, pues las operaciones se pueden aplicar de forma recursiva, permitiendo la construcción de expresiones arbitrariamente complejas.

Una alternativa sería acotar el espacio de búsqueda estableciendo un número máximo de operaciones. Si bien esta búsqueda exhaustiva acotada sí genera un espacio finito, al ser nues-

tro problema NP-Completo, el tamaño de este espacio acotado crece de forma combinatoria, como veremos más adelante, haciendo la aproximación inviable en la práctica para valores de k o $|F|$ mínimamente grandes. Es por ello que debemos hacer uso de aproximaciones heurísticas que nos permitan obtener soluciones razonablemente buenas, es decir, aproximaciones del frente de Pareto, en un tiempo asumible. Nosotros estudiaremos tres enfoques distintos: una búsqueda exhaustiva con profundidad limitada (que emplearemos como referencia y validación para niveles de profundidad k pequeños), una heurística Greedy, y un algoritmo genético multiobjetivo basado en NSGA-II. Cada uno de ellos requiere adaptar la formulación del problema e imponer restricciones específicas sobre la forma en que se explora el espacio de expresiones $\mathcal{E}$, de modo que sea posible aplicar dichas técnicas.

A continuación definimos las aproximaciones algorítmicas asociadas a cada uno de los métodos de resolución implementados, detallando sus fundamentos, complejidad y modo de integración en el marco experimental propuesto.

Cabe destacar que, en la práctica empleamos una notación en la que todas las subexpresiones con operaciones están delimitadas por paréntesis. De esta forma, cada operación op $\in \mathcal{O}$ sobre dos subexpresiones e_i y e_j se construye explícitamente como $(e_i \,\mathrm{op}\, e_j)$.

3.2.1. Búsqueda exhaustiva con profundidad limitada

Como hemos mencionado, los algoritmos de fuerza bruta resultan útiles en problemas con un espacio de búsqueda pequeño y bien definido, caso muy alejado del nuestro. Sin embargo, es posible adaptarlos para realizar exploraciones completas hasta cierta profundidad. En nuestro caso, implementamos una búsqueda exhaustiva con profundidad limitada, que recorre sistemáticamente el espacio de expresiones hasta una profundidad máxima k (el número máximo de operaciones permitido). De esta manera no estamos recorriendo todo el espacio $\mathcal{E}$, pero sí una parte manejable que permite analizar el comportamiento del problema manteniendo un tiempo de ejecución y un consumo de recursos menor.

La búsqueda exhaustiva tiene la ventaja de ser sencilla de implementar, pues genera y evalúa cada expresión posible. Este algoritmo no requiere estructuras de datos complejas y constituye una referencia útil para validar algunas de las aproximaciones heurísticas posteriores.

Los componentes básicos de un algoritmo de búsqueda exhaustiva son:

1. Generador de candidatos: función que produce, de forma sistemática, cada elemento del conjunto de soluciones posibles, siguiendo un orden determinado.

2. Función objetivo: función que evalúa cada solución completa generada, ya sea mediante una o varias métricas.

3. Criterio de parada: condición que determina cuándo debe detenerse la exploración.

Definimos ahora una aproximación algorítmica de nuestro problema, que surge del funcionamiento de un algoritmo de búsqueda exhaustiva con profundidad limitada:

Definición 3.1 (Aproximación con profundidad acotada k)**.** Sea U un conjunto finito, $F \subseteq \mathcal{P}(U)$ una familia, $G \subseteq U$ un conjunto objetivo y $\mathcal{M}$ un conjunto de medidas. Fijamos un $k \in \mathbb{N}$. Definimos el problema de **aproximación con profundidad acotada** k como un caso particular del problema de aproximación de G mediante expresiones (definición 1.13), en el que el conjunto de restricciones $\mathcal{C}$ es:

$$C_1(e) : \text{la expresión utiliza a lo sumo } k \text{ operaciones.}$$

Buscamos, por lo tanto, el conjunto de expresiones no dominadas, es decir, aquellas que pertenecen al frente de Pareto asociado $(\mathcal{E}_F^{\mathcal{C}})^{\mathcal{M}}$.

Esta definición establece la aproximación menos restrictiva del problema para poder aplicar una búsqueda exhaustiva con profundidad limitada, pero este modelo general podría complicarse más si se añadiesen restricciones adicionales al conjunto $\mathcal{C}$. De hecho, la aproximación Greedy que veremos en la siguiente subsección es un ejemplo de ello, pues añade restricciones a la forma en que se construyen las expresiones, limitando así el espacio de búsqueda.

Vemos que la búsqueda exhaustiva es un claro ejemplo de aproximación algorítmica en la que las medidas siguen teniendo sentido como criterio para imponer restricciones o para evaluar la calidad de las posibles soluciones, pero no como criterio interno. En este caso no se toman decisiones durante la ejecución ya que el algoritmo recorre todas las combinaciones posibles hasta un nivel de profundidad k, sin guiar su exploración mediante ninguna medida.

Podemos ahora relacionar los componentes generales de un algoritmo de búsqueda exhaustiva con profundidad limitada con los elementos de nuestro problema:

- Generador de candidatos: aunque en nuestra formulación teórica describimos el espacio de expresiones $\mathcal{E}$ como ya construido, algorítmicamente este componente corresponde a la función que lo genera de forma recursiva, partiendo de los subconjuntos de F, hasta enumerar todas las expresiones con un máximo de k operaciones.
- Función objetivo: función que calcula las medidas definidas sobre cada expresión $e \in \mathcal{E}^{\mathcal{C}}$ y devuelve su vector de valores.
- Criterio de parada: se cumple cuando se ha explorado el nivel de profundidad k, es decir, cuando se han generado y evaluado todas las expresiones posibles en $\mathcal{E}^{\mathcal{C}}$.

Volviendo al funcionamiento concreto de la búsqueda exhaustiva con profundidad limitada adaptada a nuestra aproximación, el algoritmo comienza construyendo el espacio de expresiones de forma recursiva. Primero incluye todas las expresiones elementales $F_i \in F$ y el conjunto U. A partir de ellas, genera nuevas expresiones combinando todas las posibles parejas $(e_i \text{ op } e_j)$, con e_i, e_j expresiones ya exploradas y $\text{op} \in \{\cup, \cap, \setminus\}$, añadiendo cada nueva expresión al espacio. Este proceso continúa hasta que se termina de explorar el nivel de profundidad k, que es el número máximo de operaciones que permitimos en las expresiones generadas. Cada expresión obtenida se evalúa mediante la función objetivo, que en este caso agrupa las medidas definidas en $\mathcal{M}$, evaluando la calidad de cada solución en el sentido multiobjetivo. Durante la exploración se van almacenando los valores obtenidos y, al final, se calculan las expresiones no dominadas. El criterio de parada se cumple cuando se han

explorado las soluciones de k operaciones, momento en el que se han generado y evaluado todas las expresiones posibles en $\mathcal{E}^{\mathcal{C}}$. El resultado final del algoritmo no es una única solución óptima, sino el conjunto de expresiones no dominadas, es decir, el frente de Pareto correspondiente al espacio explorado.

Veamos su funcionamiento de forma más intuitiva mediante su pseudocódigo:

Algoritmo 1 Búsqueda Exhaustiva

```
1:  Expr ← lista de listas, indexado 0..k
2:  Soluciones ← ∅
3:  Para cada B ∈ F hacer
4:      e ← CREAR_EXPRESION(B)
5:      Expr[0] ← Expr[0] ∪ {e}
6:      m⃗ ← EVALUAR(e, G, M)
7:      Soluciones ← Soluciones ∪ {(e, m⃗)}
8:  Fin Para
9:  Para s ← 1 hasta k hacer
10:     Para cada op ∈ {∪, ∩, \} hacer
11:         Para a ← 0 hasta s − 1 hacer
12:             b ← s − 1 − a
13:             Para cada e_i ∈ Expr[a] hacer
14:                 Para cada e_j ∈ Expr[b] hacer
15:                     e_nueva ← COMBINAR(e_i, op, e_j)
16:                     Expr[s] ← Expr[s] ∪ {e_nueva}
17:                     m⃗ ← EVALUAR(e_nueva, G, M)
18:                     Soluciones ← Soluciones ∪ {(e_nueva, m⃗)}
19:                 Fin Para
20:             Fin Para
21:         Fin Para
22:     Fin Para
23: Fin Para
24: PF ← FRENTEPARETO(Soluciones)
25: Devolver PF
```

El funcionamiento de esta implementación se basa en una estrategia de generación de abajo hacia arriba[3], construyendo las expresiones por niveles de complejidad.

En las líneas 1 y 2, el algoritmo comienza inicializando dos estructuras de datos principales. En la línea 1, se reserva `Expr`, una lista de listas que guardará las expresiones generadas, indexadas por su número de operaciones s (de 0 a k). En la línea 2, se inicializa `Soluciones` como una lista vacía, que contendrá todas las expresiones generadas y evaluadas, independientemente de su nivel.

De la línea 3 a la 8, se construye y evalúa el nivel $s = 0$, que son las expresiones sin operaciones. El bucle de la línea 3 itera sobre todos los conjuntos iniciales de la familia F, creando

[3]Del inglés, *Bottom-Up*

una expresión elemental e para cada uno de ellos mediante la función CREAR_EXPRESION y almacenándola en `Expr[0]`. Se evalúa inmediatamente esa expresión mediante la función EVALUAR para obtener el vector de medidas m y se añade a la lista global de soluciones.

A continuación, en el bucle de las líneas 9-23 tenemos la generación iterativa. El bucle de la línea 9 itera para cada nivel de profundidad s, desde 1 hasta el límite k. En las líneas 10-12, para un s dado, el algoritmo itera sobre las tres operaciones y sobre todas las posibles expresiones de los niveles a y b tales que $a + b + 1 = s$. En las líneas 13-14, los bucles anidados recorren todas las combinaciones posibles de expresiones e_i (del nivel a) y e_j (del nivel b) que ya fueron generadas en pasos anteriores. En la línea 15 se genera una e_{nueva} al combinar e_i y e_j con el operador op. En la siguiente línea, la nueva expresión (que por construcción tiene s operaciones), se almacena en `Expr[s]` para ser utilizada en niveles superiores. En las líneas 18-19, de forma análoga al caso base, cada e_{nueva} se evalúa inmediatamente y se añade a la lista global de `Soluciones`.

Por último, una vez que el bucle principal concluye, la lista `Soluciones` contiene todas las expresiones evaluadas desde el nivel 0 hasta el k. En la línea 24 se aplica la función FRENTE-PARETO sobre dicha lista que nos devuelve el frente de Pareto. Finalmente, se devuelve el conjunto `PF`, que representa el frente de Pareto exacto para el espacio explorado.

Complejidad

Ahora que tenemos el pseudocódigo del algoritmo de búsqueda exhaustiva con profundidad limitada, podemos analizar su complejidad. Empezamos determinando cuántas expresiones se generan en cada nivel s:

$$|\mathsf{Expr}[s]| = |\mathcal{O}| \sum_{a=0}^{s-1} |\mathsf{Expr}[a]||\mathsf{Expr}[s-1-a]|$$

Esta recurrencia es análoga a la de los números de Catalan [Cam94]:

$$C_{j+1} = \sum_{a=0}^{j} C_a C_{j-a}, \quad C_0 = 1,$$

que, entre otras cosas, cuenta el número de posibles árboles binarios con j nodos internos o, equivalentemente, con $j + 1$ hojas.

Cada una de nuestras expresiones se puede expresar como un árbol binario, donde los nodos internos son las operaciones binarias $(\{\cup, \cap, \setminus\})$ y las hojas son los conjuntos $F_i \in F$. Vemos en la Figura 3.1 figura un ejemplo sencillo de esta representación:

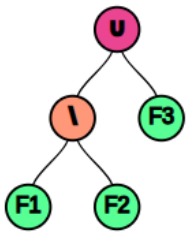

Figura 3.1.: Árbol binario de la expresión $(F_1 \setminus F_2) \cup F_3$.

De esta manera, contar el número de expresiones con s operaciones, equivale a contar el número de árboles binarios de s nodos internos. El número de posibles estructuras de esos árboles viene dado por el número de Catalan: $C_s = \frac{1}{s+1}\binom{2 \cdot s}{s}$ [GC20]. Sin embargo, en nuestro problema, además tenemos que considerar las etiquetas que puede tener cada nodo y cada hoja:

- Cada nodo interno puede ser cualquiera de las operaciones de $\mathcal{O}$ ($|\mathcal{O}| = 3$). Por lo que, las formas de rellenar los nodos internos es equivalente a calcular las variaciones con repetición de 3 elementos cogidos de s en s. Son variaciones con repetición porque se pueden repetir operaciones en una expresión y no tenemos por qué usar todas las operaciones. Por lo tanto:
$$VR_3^s = 3^s$$
- Cada hoja puede ser cualquiera de las expresiones iniciales, es decir, los conjuntos de F. Sea $m = |F|$ el cardinal de la familia generadora. Así, tenemos m opciones para cada una de las $s + 1$ hojas:
$$VR_m^{s+1} = m^{s+1}$$

Multiplicando por estos dos valores, obtenemos una cota exacta del número de expresiones con s operaciones:

$$|\mathsf{Expr}[s]| = C_s \cdot 3^s \cdot m^{s+1}$$

Por lo que el número total de expresiones generadas hasta el nivel k, denotado como $|V|$, es:

$$|V| = \sum_{s=0}^{k} |\mathsf{Expr}[s]| = \sum_{s=0}^{k} C_s \cdot 3^s \cdot m^{s+1}; \quad C_s = \frac{1}{s+1}\binom{2s}{s}.$$

C_s se aproxima asintóticamente a $\frac{2^{2s}}{s\sqrt{\pi s}} = \frac{4^s}{s\sqrt{\pi s}}$ [FS09]. Por lo tanto, tomando los términos dominantes, el número total de expresiones generadas, $|V|$, es $O((4 \cdot 3 \cdot m)^k \cdot m)$. Sin embargo, este valor $|V|$ representa solo el número de candidatos, no el tiempo total de ejecución. Como el coste de evaluación de cada expresión no es $O(1)$, debemos analizar los costes computacionales por separado:

- Coste de generación y evaluación: el algoritmo genera $|V|$ expresiones. La evaluación de cada una está dominada por el cálculo del índice de Jaccard cuya complejidad es $O(n)$ (donde $n = |U|$). Por lo tanto, la cota superior teórica de este coste es $O(|V| \cdot n)$.

- Coste de Filtrado de Pareto: como hemos visto, el coste de filtrar el frente de Pareto es $O(3 \cdot |V|^2)$ (ya que hay 3 medidas a optimizar).

Por lo tanto, la complejidad temporal total del algoritmo es la suma de estos componentes: $O(|V| \cdot n + |V|^2)$.

Cabe destacar que esta complejidad corresponde a una implementación de filtrado en lote, donde primero se generan las $|V|$ soluciones y, al final, se aplica el filtrado comparándolas todas entre sí. Esta implementación es la más directa para un algoritmo de fuerza bruta y nos permite reutilizar fácilmente la función de filtrado de Pareto en otros algoritmos. Se podría considerar una optimización mediante un filtrado incremental, que reduciría la complejidad temporal en el caso promedio. Sin embargo, esta no alteraría el cuello de botella fundamental de este algoritmo, que se debe a la complejidad espacial. Para poder generar las expresiones de nivel s, nuestra implementación debe almacenar en memoria todas las expresiones de los niveles $a < s$. Esta tabla de generación (`Expr` en el pseudocódigo) tiene un tamaño total de $O(|V|)$, creciendo exponencialmente con k. En la práctica, la limitación de memoria $O(|V|)$ es un cuello de botella que se alcanza antes que el límite de tiempo $O(|V|^2)$. Por lo tanto, aunque el filtrado incremental es una optimización válida, la intratabilidad fundamental del algoritmo se debe al crecimiento exponencial de $|V|$. Esto limita la viabilidad práctica del algoritmo a valores de k muy pequeños.

3.2.2. Greedy

Precisamente debido a las limitaciones de la búsqueda exhaustiva, surge la necesidad de recurrir a alternativas heurísticas que ofrezcan soluciones de buena calidad en tiempos razonables. Por ello, en esta sección proponemos un enfoque heurístico al que nos referimos como Greedy-MO.

Un algoritmo Greedy[4] tradicional (mono-objetivo) elige, en cada paso, la opción más prometedora según un criterio local, sin tener en cuenta las posibles consecuencias futuras de su elección [KT05]. Esta estrategia limita la búsqueda a una única rama, lo que la hace rápida pero propensa a quedar atrapada en óptimos locales. Sin embargo, esta estrategia no es directamente aplicable a MOP, ya que no existe una única opción más prometedora, sino un conjunto de soluciones de compromiso (es decir, soluciones en el frente de Pareto, que representan distintos equilibrios entre los objetivos y no son estrictamente mejores ni peores entre sí).

El algoritmo Greedy-MO que proponemos es una heurística híbrida que ajusta y combina conceptos de varias familias de algoritmos. Hemos mantenido el término "Greedy" por varias razones. Por un lado, así como el Greedy clásico se queda con una sola solución óptima local, nuestra aproximación selecciona aquel único conjunto de expresiones que es localmente óptimo (es decir, el frente de Pareto) y descarta todas las demás soluciones dominadas. Por otro lado, mantenemos el término por la naturaleza voraz de podar todas aquellas soluciones que no nos interesen sin pensar en las consecuencias futuras.

[4]También denominado algoritmo voraz.

Nuestro enfoque además tiene similitudes con la Búsqueda en Haz[5], que es un algoritmo de búsqueda en árbol que, en lugar de explorar todos los nodos, solo considera un cierto número de mejores nodos en un nivel determinado, descartando el resto [LFSJ22]. Esto es una generalización del algoritmo Greedy tradicional (que equivale a escoger en cada nivel 1 solo nodo). A diferencia de la búsqueda en haz clásica que es mono-objetivo y opera con un haz de ancho fijo, nuestra adaptación se distingue por tres aspectos. En primer lugar, la selección no se basa en una única medida, sino en un vector de medidas mediante la dominancia de Pareto. En segundo lugar, el número de nodos que expandimos en cada iteración no es fijo, sino que se define por la cardinalidad del frente de Pareto local. En tercer lugar, para seleccionar las soluciones que expandimos, comparamos cada nuevo candidato contra todo el espacio de las mejores soluciones hasta el momento, no solo las de la iteración actual.

Por tanto, nuestra adaptación al dominio multiobjetivo es un enfoque híbrido. Aunque comparte conceptos de otras heurísticas como la Búsqueda de Haz Iterativa[6] o el Greedy Iterativo[7], la combinación específica de condiciones que proponemos no la hemos localizado en la literatura consultada. La Búsqueda de Haz Iterativa ejecuta múltiples Búsquedas en Haz, aumentando el tamaño del haz en cada reinicio hasta alcanzar un límite de tiempo [LFSJ22]. Por su parte, el Greedy Iterativo es una metaheurística que genera una solución inicial completa y luego itera en un bucle de dos fases: una destrucción (donde se eliminan componentes) y una construcción (donde se repara la solución). Después, se utiliza un criterio de aceptación para decidir qué solución se mantiene para la siguiente iteración, permitiendo escapar de óptimos locales [MPR18].

Definimos ahora una aproximación algorítmica de nuestro problema, imponiendo restricciones en la forma en que se construyen las expresiones, las cuales surgen del funcionamiento del propio algoritmo Greedy-MO. Al igual que la búsqueda exhaustiva, esta aproximación debe estar acotada, ya que el espacio de expresiones es infinito.

Definición 3.2 (Aproximación secuencial)**.** Sea U un universo, F una familia suya, $G \subseteq U$ un conjunto objetivo y $\mathcal{M}$ un conjunto de medidas. Fijamos una profundidad máxima $k \in \mathbb{N}$. Definimos la **aproximación secuencial** como un caso particular del problema de aproximación de G mediante expresiones (definición 1.13), donde el conjunto de restricciones $\mathcal{C}$ tiene los siguientes elementos:

- $C_1 : \emptyset \notin \mathcal{F}(e)$,
- $C_2 : |\mathcal{O}p^*(e)| \leq k$,
- $C_3 : e = (((F_{i_0} \ \text{op}_1 \ F_{i_1}) \ \text{op}_2 \ F_{i_2}) \cdots)$,

de modo que toda expresión e está construida añadiendo en cada paso un nuevo operador y un nuevo conjunto siempre por la derecha de la expresión parcial. Esto equivale a prescindir de los paréntesis, evaluando siempre de izquierda a derecha.

[5]Del inglés, *Beam Search*.
[6]Del inglés, *Iterative Beam Search*.
[7]Del inglés, *Iterated Greedy*.

Denotamos por $e^{[t]}$ a una expresión construida en el paso t, por PF_i al frente de Pareto del paso i, y definimos una aproximación para resolver el problema anterior:

Definición 3.3 (Aproximación Greedy-MO)**.** Es una aproximación del problema de la definición 3.2 donde el objetivo es encontrar el frente de Pareto del conjunto de todas las expresiones $e \in \mathcal{E}^{\mathcal{C}}$ generadas por el siguiente proceso constructivo por niveles $t = 0 \dots k$:

- Nivel 0: PF_0 se calcula como el frente de Pareto de las soluciones elementales. Es decir, las soluciones construidas sin operaciones, solo con un elemento de F.
- Nivel $t > 0$: El conjunto de candidatos de este nivel se genera combinando únicamente las soluciones del frente anterior (PF_{t-1}) con un elemento de F mediante una operación: $C_t = \{e^{[t-1]} \operatorname{op} F_i \mid e^{[t-1]} \in \mathsf{PF}_{t-1}, F_i \in F\}$. El frente de este nivel, PF_t, es el conjunto de soluciones no dominadas de estos nuevos candidatos. Se calcula el nuevo frente de Pareto global, $\mathsf{PF}_{\mathsf{Global}}$, a partir del que teníamos y el PF_{t}.

Por lo tanto, el objetivo es encontrar el frente de Pareto del conjunto de todas las expresiones $e \in \mathcal{E}^{\mathcal{C}}$ generadas de manera que en cada paso $t = 1, \dots, m$, el par (op, F_{i_t}) es aquel que convierte a $e^{[t]}$ en una solución del frente de Pareto local de la iteración t. El estado inicial se define escogiendo el elemento de F que convierte a $e^{[0]}$ en un elemento del frente de Pareto del nivel 0.

Esta definición nos permite ver claramente el nuevo uso de las medidas como criterio utilizado dentro del algoritmo. Para poder monitorizar cómo se construye una expresión parcial $e^{[i]}$ en cada iteración, y elegir efectivamente aquellas no dominadas en dicha iteración, es necesario usar un conjunto de medidas en cada una de dichas iteraciones. Estas medidas pueden ser cualesquiera de las medidas ya mencionadas que se pueda adaptar para que actúe o bien sobre la solución parcial, o bien sobre los elementos del conjunto candidato, pudiendo siempre comparar con el conjunto objetivo. También pueden ser medidas no presentadas antes, como las medidas de redundancia:

$$M(e^{[n]}) = |F_i \cap eval(e^{[n]})|.$$

Estas miden la intersección entre un conjunto candidato F_i y la solución parcial en el paso n, $\tilde{G}_n$. Esta medida permite priorizar soluciones donde los conjuntos seleccionados tienen poca intersección entre sí, lo cual favorece la diversidad. Es útil en contextos donde la redundancia es costosa o no deseable, como en casos de segmentación o clasificación. Sin embargo, puede llevar a descartar conjuntos útiles que, aunque se solapen, aportan una mejor cobertura sobre G.

Vemos así, que las medidas que presentábamos, no solo tienen un interés teórico para permitirnos aplicar restricciones para particularizar el problema, o para medir la calidad de una solución obtenida, sino que existe un gran interés y necesidad práctica, para el propio funcionamiento del algoritmo.

Haciendo uso de la definición que hemos dado, analizamos los elementos de nuestro problema y cómo se corresponden con los del algoritmo:

- Conjunto candidato: corresponde al producto cartesiano

$$(\mathcal{O} \times (F \setminus \{\emptyset\})) = \{\, (\mathrm{op}_j, F_i);\ F_i \in F \setminus \{\emptyset\},\ \mathrm{op}_j \in \mathcal{O}\,\}.$$

  Cada elemento representa una acción que se puede aplicar sobre una solución parcial actual. En la primera iteración, las primeras expresiones válidas son de la forma $e_i = F_{i_j}$, seleccionadas según el criterio inicial que hemos descrito en la definición.

- Función objetivo: Es el vector de métricas $\mathcal{M} = (M_1, M_2, M_3)$. Se usa para medir la calidad de cada solución.

- Función de selección: determina con cuáles de los candidatos generados nos quedamos. Viene dada por el proceso que calcula el frente de Pareto global utilizando la función objetivo. En cada iteración, selecciona aquellos candidatos que son globalmente no dominados.

- Función de solución: el algoritmo termina cuando se alcanza el límite de profundidad, es decir, $|\mathcal{O}p^*(e)| = k$, y por tanto no pueden explorarse más expresiones bajo las restricciones impuestas; o cuando en la iteración anterior no se han podido obtener soluciones nuevas no dominadas y por tanto sabemos que ya no se conseguirán.

Volviendo al funcionamiento de Greedy-MO, aplicándolo en concreto para nuestra aproximación, el algoritmo parte del frente de Pareto inicial calculado sobre F. En cada iteración $iter = 1, \ldots, k$, se generan todos los candidatos combinando el frente anterior con pares de la forma (op, F_i), con op$\in \mathcal{O}$, $F_i \in F$. Para cada candidato, se evalúa el vector de medidas M. A continuación se seleccionan todos los candidatos no dominados del nivel t. Estos formarán el nuevo frente de Pareto de dicho nivel, que se usará como base para la siguiente iteración. El proceso continúa hasta alcanzar las k iteraciones, devolviendo el frente de Pareto global.

Lo vemos de forma más intuitiva con el pseudocódigo 2:

En las líneas 1-4 inicializamos: las listas de soluciones `PF_anterior` y `PF_actual`, que almacenarán el frente de Pareto local de la iteración anterior y la actual, respectivamente; `FrenteGlobal` que almacenará el frente de Pareto acumulado de todos los niveles; y `BloquesBase` contendrá las expresiones iniciales. En las líneas 5-8, el algoritmo guarda en `BloquesBase` todos los elementos de F junto con su evaluación. En las líneas 9-10 se hace la inicialización: `PF_anterior` se calcula como el frente de Pareto de los bloques base, y `FrenteGlobal` se inicializa con este mismo frente.

En la línea 12 comienza el bucle principal, que se ejecutará mientras no se supere la profundidad k y mientras el frente de la iteración anterior (`PF_anterior`) no esté vacío. En la línea 13 se crea una lista temporal `Candidatos_s` para todas las nuevas expresiones de ese nivel. En las líneas 14-22, ocurre la expansión Greedy. El algoritmo genera candidatos combinando únicamente las soluciones del frente anterior `PF_anterior` con los `BloquesBase` mediante alguna operación. En la línea 23 se hace un filtrado local, calculando el `FrenteLocal_s` a partir de los `Candidatos_s` generados, quedándonos solo con los no dominados en este nivel.

En las líneas 24-25 hacemos un filtrado global, combinando los "ganadores" locales (los de `FrenteLocal_s`) con los "ganadores" de todos los niveles anteriores (`FrenteGlobal`). Al filtrar

esta lista combinada, obtenemos el `NuevoFrenteGlobal`. En las líneas 27-31, preparamos la siguiente iteración. El algoritmo necesita saber qué soluciones usar en la próxima iteración, así que se reconstruye el `PF_actual` extrayendo del `NuevoFrenteGlobal` solo aquellas soluciones que se construyeron en la iteración actual. Para ello, filtramos aquellas expresiones e cuyo número de operaciones, $|\mathcal{O}p^*(e)|$, es exactamente s.

Finalmente, se actualiza el `FrenteGlobal` y se asigna `PF_actual` a `PF_anterior` para la siguiente iteración del bucle. El algoritmo termina devolviendo el `FrenteGlobal`, que contiene todas las soluciones no dominadas encontradas en cualquier nivel de profundidad.

Algoritmo 2 Greedy-MO

1: PF_anterior $\leftarrow \emptyset$
2: PF_actual $\leftarrow \emptyset$
3: FrenteGlobal $\leftarrow \emptyset$
4: BloquesBase $\leftarrow \emptyset$
5: **Para cada** $e \in F$ **hacer**
6: $\quad \vec{m} \leftarrow \textsc{Evaluar}(e, G, \mathcal{M})$
7: $\quad$ BloquesBase $\leftarrow$ BloquesBase $\cup \{(e, \vec{m})\}$
8: **Fin Para**
9: PF_anterior $\leftarrow \textsc{FrentePareto}$(BloquesBase)
10: FrenteGlobal $\leftarrow$ PF_anterior
11: $s \leftarrow 1$
12: **Mientras** $s \leq k$ **y** PF_anterior $\neq \emptyset$ **hacer**
13: $\quad$ Candidatos$_s \leftarrow \emptyset$
14: $\quad$ **Para cada** op $\in \{\cup, \cap, \setminus\}$ **hacer**
15: $\quad\quad$ **Para cada** $(e_i, \vec{m}_i) \in$ PF_anterior **hacer**
16: $\quad\quad\quad$ **Para cada** $(e_j, \vec{m}_j) \in$ BloquesBase **hacer**
17: $\quad\quad\quad\quad e_{nueva} \leftarrow \textsc{Combinar}(e_i, \text{op}, e_j)$
18: $\quad\quad\quad\quad \vec{m}_{nuevo} \leftarrow \textsc{Evaluar}(e_{nueva}, G, \mathcal{M})$
19: $\quad\quad\quad\quad$ Candidatos$_s \leftarrow$ Candidatos$_s \cup \{(e_{nueva}, \vec{m}_{nuevo})\}$
20: $\quad\quad\quad$ **Fin Para**
21: $\quad\quad$ **Fin Para**
22: $\quad$ **Fin Para**
23: $\quad$ FrenteLocal$_s \leftarrow \textsc{FrentePareto}$(Candidatos$_s$)
24: $\quad$ FrenteCombinado $\leftarrow$ FrenteGlobal $\cup$ FrenteLocal$_s$
25: $\quad$ NuevoFrenteGlobal $\leftarrow \textsc{FrentePareto}$(FrenteCombinado)
26: $\quad$ PF_actual $\leftarrow \emptyset$
27: $\quad$ **Para cada** $(e, \vec{m}) \in$ NuevoFrenteGlobal **hacer**
28: $\quad\quad$ **Si** $|\mathcal{O}p^*(e)| = s$ **Entonces**
29: $\quad\quad\quad$ PF_actual $\leftarrow$ PF_actual $\cup \{(e, \vec{m})\}$
30: $\quad\quad$ **Fin Si**
31: $\quad$ **Fin Para**
32: $\quad$ FrenteGlobal $\leftarrow$ NuevoFrenteGlobal
33: $\quad$ PF_anterior $\leftarrow$ PF_actual
34: $\quad s \leftarrow s + 1$
35: **Fin Mientras**
36: **Devolver** FrenteGlobal

Complejidad

Ahora que disponemos del pseudocódigo, podemos analizar la complejidad del algoritmo Greedy-MO.

Usamos las siguientes variables:

- k: la profundidad máxima.
- m: el número de elementos base ($|F|$).
- n: el tamaño del universo ($|U|$).
- P_s: el tamaño del frente de Pareto local en el nivel s, es decir, $|\mathsf{FrenteNivel}[s]|$.
- $P_{\max}$: el tamaño máximo que alcanza P_s en cualquier nivel.
- G_s: el tamaño del frente de Pareto global en el nivel s, es decir, $|\mathsf{FrenteGlobal}|$.
- $G_{\max}$: el tamaño máximo que alcanza G_s.

El coste total es la suma de los costes del bucle principal, que se ejecuta k veces. Analicemos el coste de una única iteración s:

- Coste de generación y evaluación: El algoritmo genera un conjunto de candidatos $\mathsf{Candidatos}_s$. El número de candidatos es el producto de las tres operaciones ($|\mathcal{O}| = 3$), el frente anterior (P_{s-1}) y los bloques base (m).

  $$|C_s| = 3 \cdot P_{s-1} \cdot m \approx O(P_{\max} \cdot m)$$

  Cada candidato se genera (combinación Bitset (ver Subsección 3.3.1), $O(n)$) y se evalúa (dominado por el índice de Jaccard, $O(n)$). El coste por candidato es $O(n)$. Por lo tanto, el coste de esta fase es: $O(P_{\max} \cdot m \cdot n)$.

- Coste de filtrado: El algoritmo debe filtrar los $|C_s|$ candidatos. El coste de FrentePareto es cuadrático respecto al número de candidatos: $O(|C_s|^2 \cdot 3) = O((P_{\max} \cdot m)^2 \cdot 3)$ (ya que hay 3 medidas a optimizar). El algoritmo filtra la unión del frente global anterior y el nuevo frente local. El coste es: $O((G_{s-1} + P_s)^2 \cdot 3) \approx O((G_{\max} + P_{\max})^2 \cdot 3)$.

Por lo tanto, la complejidad temporal total del algoritmo es el producto del número de iteraciones (k) por el coste de la iteración más costosa. Dicho coste viene dominado por el término más grande de los tres que hemos analizado (generación/evaluación, filtrado local y filtrado global):

$$T = O\left(k \cdot \max\left(P_{\max} \cdot m \cdot n, (P_{max} \cdot m)^2, (G_{\max} + P_{\max})^2\right)\right),$$

que depende linealmente de k, del tamaño del universo n, y de forma polinómica de m, del frente de Pareto local ($P_{\max}$) y del frente de Pareto global (G_{max}).

3.2.3. Algoritmo Genético

Tras haber analizado e implementado un método de búsqueda exhaustiva limitada, capaz de encontrar el frente de Pareto verdadero dentro de dicho subespacio acotado, pero limitada por su coste computacional; y una heurística como Greedy-MO, que obtiene una aproximación a dicho frente en tiempos mucho menores; resulta natural presentar ahora un enfoque metaheurístico que busca equilibrar ambas perspectivas, como es el caso de los algoritmos genéticos.

Los algoritmos genéticos son una clase de metaheurísticas estocásticas que se inspiran en procesos de selección natural y genética, como la selección, el cruce y la mutación, para explorar el espacio de soluciones de forma eficiente [Gol89]. Funcionan sobre poblaciones en las que las soluciones son individuos, combinando la supervivencia del individuo más fuerte con un intercambio aleatorio de información para guiar la búsqueda.

A diferencia de las heurísticas clásicas, las metaheurísticas no se limitan a una única trayectoria de búsqueda, sino que operan sobre una población de posibles soluciones. Su naturaleza probabilística les permite explorar de forma más global el espacio de soluciones, siendo especialmente útiles en problemas de gran dimensión, con espacios de búsqueda complejos o discontinuos. Esto les permite escapar de óptimos locales y alcanzar resultados de mayor calidad en un tiempo razonable, aunque al no tratarse de métodos exactos no garantizan la obtención del óptimo [Hol92].

Dentro de las metaheurísticas evolutivas, los algoritmos genéticos destacan por ser sencillos de implementar. Otras técnicas como las estrategias evolutivas se han desarrollado principalmente para problemas de optimización continua, mientras que los algoritmos genéticos resultan especialmente adecuados para problemas discretos, donde las soluciones pueden representarse de forma natural como conjuntos de elementos.

En este trabajo, el problema está formulado precisamente en esos términos, lo que encaja de forma directa con los algoritmos genéticos. Sus versiones multiobjetivo extienden este enfoque evolutivo clásico a problemas en los que se deben optimizar simultáneamente varias funciones objetivo en conflicto. En lugar de buscar una única solución óptima, el propósito es aproximar el frente de Pareto.

Dado que nuestro problema es multiobjetivo, un algoritmo genético clásico con una única función de idoneidad[8] no es aplicable. En su lugar, utilizamos un modelo evolutivo que genere poblaciones de soluciones no dominadas. Algunos de los algoritmos más conocidos en esta área se pueden encontrar en [KCS06]:

- MOGA (*Multi-Objective Optimization Genetic Algorithm*): asigna la aptitud a cada individuo basándose en su ránking de Pareto (cuántas soluciones de la población lo dominan). Su principal desventaja es que suele tener una convergencia lenta.
- NPGA (*Niched Pareto Genetic Algorithm*): no asigna un ranking explícito, sino que utiliza una selección por torneo donde los individuos son comparados contra un subconjunto de la población. La diversidad se mantiene usando el conteo de nichos[9] como criterio

[8] Del inglés, *fitness*.
[9] Del inglés, *niching*.

de desempate si los dos individuos seleccionados para el torneo son no-dominados entre sí. Su simplicidad se ve afectada por la necesidad de definir un parámetro de tamaño de nicho.

- NSGA (*Non-dominated Sorting Genetic Algorithm*): es el primero en proponer la clasificación por no-dominancia, que divide a toda la población en "frentes"($F_1, F_2, \dots$) según su nivel de dominancia. Su principal desventaja es su alta complejidad computacional ($O(n^3)$).
- NSGA-II (*Non-dominated Sorting Genetic Algorithm II*): es la evolución del NSGA. Resuelve los problemas de su versión anterior introduciendo dos mecanismos clave: un algoritmo de clasificación rápida ($O(n^2)$) y el uso de la distancia de aglomeración[10] para mantener la diversidad.
- SPEA (*Strength Pareto Evolutionary Algorithm*): introduce el elitismo mediante el uso de un archivo externo (una población de élite) que almacena las mejores soluciones no dominadas encontradas hasta el momento. La aptitud de un individuo se basa en el número de soluciones que domina en el archivo.

Hemos decidido implementar nuestro propio algoritmo NSGA-II desde cero, lo cual nos permite utilizar directamente los conocimientos adquiridos en la asignatura optativa de Metaheurísticas del Grado en Ingeniería Informática.

El NSGA-II es uno de los algoritmos de referencia en la optimización multiobjetivo. A diferencia de otros métodos, equilibra los dos objetivos clave de un algoritmo genético multiobjetivo: la convergencia hacia el frente de Pareto (en nuestro caso, expresiones con alto M_1 y bajos M_2, M_3) y la diversidad de las soluciones, garantizando en nuestro problema considerar expresiones estructuralmente diferentes. Todo ello con una complejidad computacional $O(n^2)$ (mejor que NSGA) y sin requerir archivos externos (como SPEA).

El espacio de búsqueda y el objetivo del NSGA-II son los mismos que los de la búsqueda exhaustiva con profundidad limitada (Definición 3.1), pues el espacio de expresiones sigue estando acotado a un máximo de k operaciones. Sin embargo, y esta es la ventaja fundamental, la naturaleza heurística y no exhaustiva del algoritmo genético permite explorar en la práctica valores de k más elevados. A continuación detallamos los componentes del NSGA-II y su adaptación a nuestra implementación específica.

El NSGA-II tiene los siguientes componentes [DPAM02][KCS06]:

1. Población de cromosomas: un cromosoma (o individuo) es la representación de una solución candidata. Un aspecto clave del diseño ha sido la coherencia con la representación de las soluciones empleadas en los algoritmos de búsqueda exhaustiva con profundidad limitada y Greedy-MO. En lugar de una estructura de árbol explícita, el cromosoma almacena una representación plana de la expresión: cadena de caracteres. Los genes de este cromosoma son los componentes elementales de la expresión, tanto los subconjuntos base F_i como los operadores que los unen. Además, utilizamos paréntesis alrededor de cada subexpresión, para definir la precedencia, pero estos no se

[10] Del inglés, *crowding distance.*

consideran genes en sí, ya que su posición es fija respecto a la estructura del árbol y su modificación arbitraria produciría expresiones matemáticamente inválidas. El individuo almacena esta expresión candidata junto con sus valores de idoneidad (rango de no-dominancia y distancia de aglomeración).

En la Figura 3.2 ilustramos dos individuos. Aunque internamente se almacenan como las cadenas de texto que aparecen en el pie, se muestran visualmente como árboles sintácticos para facilitar su comprensión.

(a) Individuo P_1: $((F_0 \cap F_1) \cup F_2)$ (b) Individuo P_2: $(F_3 \setminus (F_4 \cup F_5))$

Figura 3.2.: Individuos P_1 y P_2.

2. Mecanismo de selección: define quién sobrevive y quién se reproduce.

   - Elitismo fuerte: define cómo se crea la población de cada iteración. Se combinan la población de padres P_t y la de hijos Q_t en una "super-población" R_t. Los N mejores individuos de R_t (seleccionados según rango y distancia) forman la nueva población P_{t+1}.

   - Selección por torneo: define cómo se eligen los padres para crear los hijos. Para elegir un padre, se seleccionan dos individuos al azar de la población y compiten. El ganador es aquel con el mejor rango (menor) o, en caso de empate, la mayor distancia de aglomeración.

3. Función de cruce: simula la mezcla de material genético que ocurre en la reproducción animal, donde los cromosomas de los padres se entremezclan, de manera que algunos de los genes del hijo vienen del padre y otros de la madre. Dados dos padres, el operador de cruce genera un nuevo hijo combinándolos con una de las operaciones de conjuntos que barajamos. Esta operación solo es válida si la nueva expresión H no viola la restricción de profundidad máxima k. Como se combinan dos subexpresiones mediante un operador nuevo, los paréntesis se añaden alrededor de esta expresión final.

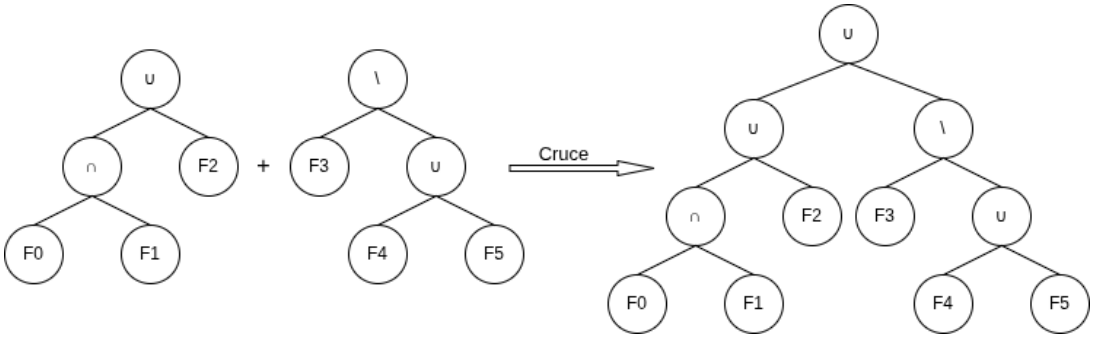

Figura 3.3.: Ejemplo del operador de cruce. Los individuos P_1 y P_2 (izquierda) se combinan para generar la nueva expresión: $(((F_0 \cap F_1) \cup F_2) \cup (F_3 \setminus (F_4 \cup F_5)))$ (derecha).

En la Figura 3.3 podemos ver un ejemplo en el que se crea un nuevo individuo utilizando la operación de cruce sobre P_1 y P_2 de la Figura 3.2 combinando sus expresiones mediante un operador aleatorio (en este caso, $\cup$).

4. Función de mutación: simula la deformación genética de un individuo que puede deberse a diferentes factores, aplicando pequeños cambios aleatorios a un individuo para introducir nueva información genética y evitar óptimos locales. Para equilibrar la exploración y la explotación, hemos implementado dos tipos de mutación:

   - Mutación de crecimiento (80 %): combina la expresión actual del individuo con un bloque base (F_i) mediante una operación aleatoria, añadiendo el nuevo bloque al principio o al final de la expresión original con igual probabilidad. Al igual que el operador de cruce, se añaden paréntesis alrededor de la nueva expresión resultante.

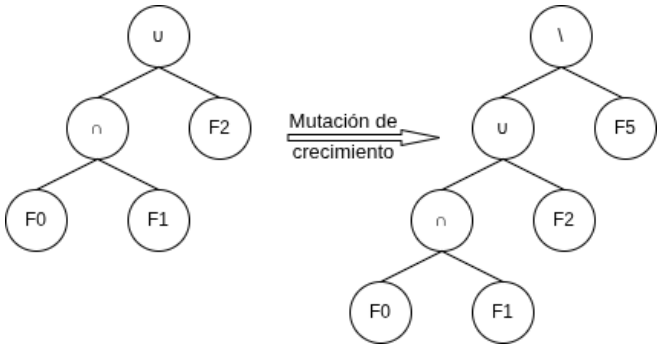

Figura 3.4.: Ejemplo de mutación de crecimiento. El individuo P_1 (izquierda) se combina, mediante una operación, con un bloque base aleatorio para generar la nueva expresión: $(((F_0 \cap F_1) \cup F_2) \setminus F_5)$ (derecha).

En la Figura 3.4 vemos un ejemplo de expresión que surge de aplicar el operador de mutación de crecimiento a P_1 (de la Figura 3.2a). En este caso, añade la operación $\setminus$ y el subconjunto F_5 al final de P_1.

- Mutación de reestructuración (20 %): A diferencia de la anterior, este operador afecta a la estructura global del individuo. Su funcionamiento se basa en conservar los subconjuntos F_i pero renovar completamente la forma en que se combinan.

  El proceso consta de dos fases:

  a) Modificación del contenido: se altera ligeramente la lista de subconjuntos base que componen el individuo (añadiendo, eliminando o sustituyendo uno de ellos).

  b) Regeneración estructural: se descarta la estructura anterior (operadores y paréntesis) y se genera una nueva expresión aleatoria utilizando únicamente la lista de subconjuntos modificada.

  Este mecanismo permite explorar nuevas formas de combinar los mismos elementos básicos, facilitando saltos grandes en el espacio de búsqueda para escapar de óptimos locales donde la selección de conjuntos es correcta pero la fórmula que los une no lo es. Además, la inclusión de este operador de mutación asegura que cualquier solución válida en el espacio de búsqueda tiene una probabilidad no nula de ser generada. Es decir, el espacio de soluciones es totalmente alcanzable.

  En la Figura 3.5 vemos un ejemplo de este proceso: en el individuo P_1 (de la Figura 3.2a) su gen F_2 es reemplazado por F_6, y se reconstruye la expresión a partir de todos sus genes, formando $((F_1 \cup F_6) \cap F_0)$.

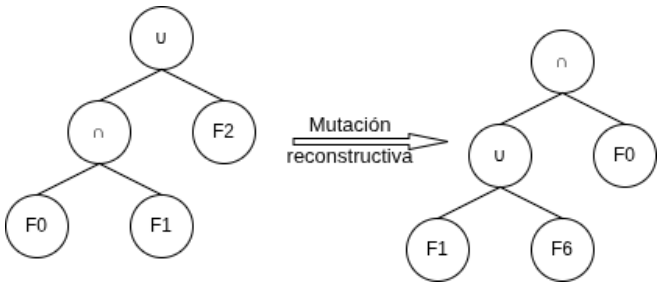

Figura 3.5.: Ejemplo de mutación destructiva. El individuo P_1 (izquierda) sufre una mutación en sus genes y, a partir de la nueva lista de genes, se reconstruye la expresión (derecha).

Los porcentajes (80/20) son parámetros que hemos escogido tras realizar pruebas preliminares para calibrar el algoritmo.

5. Función de evaluación: utiliza dos mecanismos para evaluar la calidad de las soluciones en la población.
   - Clasificación rápida por no-dominancia: es el criterio de selección primario. El algoritmo clasifica toda la población en frentes de Pareto. El rango (número de frente) es la principal medida de aptitud y se almacena en el individuo.
   - Distancia de aglomeración: es el criterio de desempate, diseñado para mantener la diversidad en el frente de Pareto. Mide la densidad de soluciones calculando el "espacio vacío" promedio alrededor de un individuo con respecto a sus vecinos. Un individuo en una región poco poblada obtiene una puntuación alta, mientras

que uno en una región densa obtiene una puntuación baja. Cuando dos individuos tienen el mismo rango, el algoritmo prioriza al que tenga mayor distancia de aglomeración, fomentando así un frente de soluciones bien distribuido.

El algoritmo genético comienza creando una población inicial de N individuos, generados de forma totalmente aleatoria. Si bien algunas implementaciones utilizan otra heurística (como Greedy) para crear esta población, en este trabajo hemos optado por la inicialización aleatoria, ya que uno de los objetivos es comparar el rendimiento de los algoritmos de forma independiente.

A partir de esta población, comienza el bucle evolutivo. La filosofía de NSGA-II se basa en el elitismo: en cada generación, se crea una nueva población de descendientes que se combina con la población de padres actual. Sobre esta "superpoblación" de tamaño $2N$ se aplica el proceso de selección completo (no-dominancia y distancia de aglomeración) para elegir a los N supervivientes que formarán la siguiente generación, asegurando que las mejores soluciones nunca se pierden.

En este caso, la aproximación algorítmica de nuestro problema para poder resolverlo con un algoritmo genético es la misma que en la definición 3.1 que utilizamos con la búsqueda exhaustiva con profundidad limitada, pues ambas tienen el mismo espacio de búsqueda y el mismo objetivo. Por ello, los espacios $\mathcal{E}^{\mathcal{C}}$ y $\mathcal{E}^{\mathcal{M}}$ son los mismos para ambos algoritmos, aunque estos utilicen distintos métodos de exploración.

Procedemos a ver el comportamiento del NSGA-II que hemos implementado, cuyo pseudocódigo se detalla en el algoritmo 3.

En las líneas 1-3 se crea la población inicial aleatoria y se preparan las variables que servirán para controlar el tiempo de ejecución y las generaciones.

En la línea 4 tenemos el bucle principal, que se detiene cuando se ha llegado al máximo de generaciones o cuando llega al límite de tiempo permitido. En las líneas 6-18, el bucle interno crea N hijos. Lo hace eligiendo los padres con la función TorneoSelección. Si procede (por probabilidad), los cruza con la función Cruzar y genera el hijo. Si no, el hijo será el primer padre. Si procede (por probabilidad), se muta (con la función Mutar) el hijo con uno de los métodos vistos. En las líneas 17, se añade el nuevo hijo a la población de descendientes.

En las líneas 19-20 unimos la población anterior a esta nueva de descendientes y asignamos el rango a cada individuo mediante la función ClasificaciónNoDominada.

En las líneas 23-34, se aplica la lógica de elitismo para crear la nueva población. Dentro de cada frente calculamos la distancia de aglomeración con la función CalcularDistAglomeración y añadimos en orden de mejor a peor, todos los que podamos hasta llenar la nueva población. Esta población es la que se utilizará en la siguiente generación.

Finalmente, devolvemos el frente de Pareto de la última población que se llegó a calcular.

Algoritmo 3 Genético (NSGA-II)

```
1: Poblacion ← InicializarPoblacion(F, U, G, k, N)
2: tiempo_inicio ← RelojActual()
3: generacion ← 0
4: Mientras generacion < G_max y (RelojActual() − tiempo_inicio < T_lim) hacer
5:     Descendencia ← ∅
6:     Mientras |Descendencia| < N hacer
7:         p_1 ← TorneoSeleccion(Poblacion)
8:         p_2 ← TorneoSeleccion(Poblacion)
9:         Si Aleatorio() < Probabilidad_cruce Entonces
10:            hijo ← Cruzar(p_1, p_2, F, U, G, k)
11:        Si No
12:            hijo ← p_1
13:        Fin Si
14:        Si Aleatorio() < Probabilidad_mutación Entonces
15:            hijo ← Mutar(hijo, F, U, G, k)
16:        Fin Si
17:        Descendencia ← Descendencia ∪ {hijo}
18:    Fin Mientras
19:    R_t ← Poblacion ∪ Descendencia
20:    Frentes ← ClasificaciónNoDominada(R_t)
21:    P_{t+1} ← ∅
22:    i ← 0
23:    Mientras i < |Frentes| y |P_{t+1}| < N hacer
24:        Si Frentes[i] ≠ ∅ Entonces
25:            CalcularDistAglomeracion(Frentes[i])
26:            Si |P_{t+1}| + |Frentes[i]| ≤ N Entonces
27:                P_{t+1} ← P_{t+1} ∪ Frentes[i]
28:            Si No
29:                OrdenarPorDistAglomeracion(Frentes[i])
30:                P_{t+1} ← P_{t+1} ∪ CopiarPrimeros(Frentes[i], N − |P_{t+1}|)
31:            Fin Si
32:        Fin Si
33:        i ← i + 1
34:    Fin Mientras
35:    Poblacion ← P_{t+1}
36:    generacion ← generacion + 1
37: Fin Mientras
38: Devolver FrentePareto(Poblacion)
```

Complejidad

A diferencia de los algoritmos deterministas, la complejidad de un algoritmo evolutivo como NSGA-II no se mide por su tiempo total de ejecución. Dada la naturaleza estocástica de estos algoritmos, no existen garantías de alcanzar el óptimo global en un tiempo finito, por lo que es imprescindible establecer una condición de terminación explícita (como un número máximo de generaciones o un límite de tiempo), que no depende del problema en sí. Por lo tanto, la práctica habitual es analizar la complejidad computacional de una única generación, para asegurar que el algoritmo es eficiente [ES15].

A continuación mostramos una versión simplificada de los pseudocódigos de las funciones de mutación y cruce, para facilitar el estudio de la complejidad sin entrar en detalles de implementación:

Algoritmo 4 Operador de Cruce Simplificado

```
op ← ALEATORIO()
CONJUNTO(hijo) ← CONJUNTO(P1) op CONJUNTO(P2)        ▷ Coste: O(n)
CADENA(hijo) ← CADENA(P1) + "op" + CADENA(P2)        ▷ Coste: O(k)
USADOS(hijo) ← USADOS(P1) ∪ USADOS(P2)               ▷ Coste: O(C log C)
EVALUAR(hijo)                                        ▷ Coste: O(n)
Devolver hijo
```

Algoritmo 5 Operador de Mutación Simplificado

```
tipo ← ALEATORIO([0, 1])
Si tipo < Prob_Reestructuracin Entonces
    Disponibles ← MODIFICAR(USADOS(hijo))
    Mientras |Disponibles| > 1 hacer
        A, B ← EXTRAERDOS(Disponibles)
        op ← ALEATORIO()
        CONJUNTO(nuevo) ← CONJUNTO(A) op CONJUNTO(B)     ▷ Coste: O(n)
        CADENA(nuevo) ← CADENA(A) + op + CADENA(B)       ▷ Coste: O(k)
        USADOS(nuevo) ← USADOS(A) ∪ USADOS(B)            ▷ Coste: O(C log C)
        INSERTAR(Disponibles, nuevo)
    Fin Mientras
    hijo ← Disponibles.top()
Si No
    hijo ← CRECIMIENTO()                             ▷ Coste menor: O(n + k + log C)
Fin Si
EVALUAR(hijo)                                        ▷ Coste: O(n)
```

Para cada generación (hasta un máximo de G_{max}) hacemos lo siguiente:

- Guardamos todas las expresiones vistas de la población (esta última de tamaño N) con un coste de $O(N \cdot k)$ para evitar duplicados. Para ello utilizamos cadenas, cada una con un coste de inserción proporcional a su longitud máxima. Como cada expresión puede tener a lo sumo k operadores y $k+1$ subconjuntos, la inserción de una cadena tiene una cota superior de orden $O(2k+1) \approx O(k)$.

- Creamos la población de descendientes (también de tamaño N), haciendo para cada hijo dos torneos de selección de coste $O(T_{tam})$, donde T_{tam} es el tamaño del torneo.

  Llamemos $C = \min\{m, k+1\}$ al número máximo de conjuntos base utilizados (cada expresión utiliza como mucho $\min\{m, k+1\}$ elementos de F). Para cada hijo, en el peor de los casos se cruza y se muta. La función de cruce, tiene una cota superior de $O(n + k + C \log C)$, como podemos ver en el pseudocódigo del algoritmo 4. En el pseudocódigo del algoritmo 5, vemos que la función de mutación queda acotada por el coste de la mutación de reestructuración, donde cada una de las C iteraciones del bucle tienen orden $O(n + k + C \log C)$, siendo así $O(C \cdot (n + k + C \log C))$ el coste total de esta función.

- Utilizamos la función ClasificaciónNoDominada sobre la población de tamaño $2N$. Este procedimiento requiere comparar cada solución con todas las demás para determinar su dominancia. En el peor de los casos, se tiene un coste de $O(M \cdot N^2)$, donde M es el número de objetivos.

- Ordenamos cada frente de Pareto con CalcularDistAglomeración. Esto requiere ordenar a los individuos del frente para cada uno de los M objetivos. Utilizando un algoritmo de ordenación eficiente, se tiene un coste de $O(M \cdot N \log N)$.

Sumando estos valores, el coste de una única generación es la suma de los anteriores. El término $O(M \cdot N^2)$ domina asintóticamente a $O(M \cdot N \log N)$, por lo que este último puede omitirse en la suma total. Agrupando los costes de la creación de los N hijos, la cota superior del coste de una generación es:

$$O(N(C \cdot n + C \cdot k + C^2 \log C) + M \cdot N^2)$$

Esto puede interpretarse de dos maneras:

- Escalabilidad del algoritmo: si tratamos los parámetros del problema como constantes (es decir, n, k, C y M son fijos), el coste de generar un individuo es una constante. La complejidad de una generación es: $O(N^2)$.

- Escalabilidad del problema: si tratamos los parámetros del algoritmo como constantes (es decir, N, M y T_{tam} son fijos), el término $O(M \cdot N^2)$ se vuelve una constante. Por lo tanto, la complejidad de una generación es: $O(C \cdot (n + k))$.

En conclusión, el análisis demuestra que el algoritmo escala cuadráticamente con el tamaño de la población ($O(N^2)$) y linealmente con la profundidad máxima permitida en las expresiones ($O(k)$).

Parámetros del algoritmo

Como venimos viendo en el pseudocódigo y la complejidad, la implementación del NSGA-II requiere calibrar varios parámetros. En este trabajo, hemos establecido los valores mediante un proceso de ajuste empírico, buscando una configuración que ofrezca un buen equilibrio

entre la explotación (converger a buenas soluciones) y la exploración (evitar óptimos locales) en cada uno de los experimentos planteados.

- Población (N): 150 individuos. Este es un tamaño de población estándar. Es lo suficientemente grande para mantener una diversidad genética saludable (evitando la convergencia prematura), pero lo suficientemente pequeño para que el coste de clasificación $O(N^2)$ del NSGA-II en cada generación sea manejable.
- Probabilidad de cruce (P_c): 0.8. Utilizamos una alta probabilidad de cruce para favorecer la explotación. Esto asegura que los rasgos de los individuos "fuertes" (aquellos en buenos frentes de Pareto) se recombinen frecuentemente, permitiendo que la población converja hacia las regiones más prometedoras.
- Probabilidad de mutación (P_m): 0.5. Hemos elegido una probabilidad de mutación relativamente alta para fomentar la exploración. Esto permitirá introducir "saltos" aleatorios y ayudar a la población a escapar de óptimos locales.
- Tamaño de torneo: 5. Este valor determina la "presión selectiva". Un valor de 5 (en lugar de 2, por ejemplo) significa que las soluciones buenas tienen una mayor probabilidad de ser seleccionadas para el cruce, acelerando la convergencia, pero sin eliminar prematuramente a individuos diversos.
- Generaciones máximas: se establece un número muy alto ya que la parada se controla en la práctica mediante el límite de tiempo.

Estos valores se mantendrán constantes en todos los experimentos, variando únicamente el límite de tiempo de ejecución para adaptarse a la complejidad de cada escenario.

3.3. Diseño, validación y reproducibilidad experimental

Hasta ahora, hemos estudiado tres algoritmos que hacen una cobertura del espacio totalmente distinta:

- Búsqueda exhaustiva con profundidad limitada: cobertura completa y determinista del espacio $\mathcal{E}^{\mathcal{C}}$. Garantiza encontrar el frente de Pareto óptimo real.
- Greedy-MO: Cobertura parcial y determinista. Explora un subespacio mucho menor (definido por la restricción C_3) y poda en cada nivel.
- NSGA-II: Cobertura parcial y estocástica. Explora el mismo espacio $\mathcal{E}^{\mathcal{C}}$ que la Exhaustiva, pero usando una búsqueda probabilística basada en población en lugar de recorriéndolo todo. Por tanto, el resultado no garantiza encontrar el frente de Pareto óptimo, sino una aproximación a dicho frente.

El objetivo principal de esta fase de diseño y experimentación, no es hacer una evaluación de rendimiento comparativa y exhaustiva entre los algoritmos. Esa tarea requeriría un diseño más complejo, con la generación de miles de instancias y un análisis de parámetros que se escapan del alcance de este trabajo. En su lugar, nuestro objetivo es la construcción de un

entorno experimental completo, reproducible y extensible. Para garantizar este último punto, en el Apéndice C se proporciona una guía detallada de compilación, ejecución y acceso al repositorio de código. Queremos diseñar una infraestructura coherente que permita generar instancias controladas, ejecutar distintos métodos de resolución, registrar automáticamente los resultados y analizarlos bajo distintos criterios. En este sentido, el énfasis está en la organización del laboratorio de experimentación: la correcta integración del código, la gestión de parámetros y semillas, la trazabilidad de los datos y la validación funcional del sistema.

Para ello, nos planteamos dos objetivos principales:

- Construir un entorno experimental reproducible que permita generar instancias aleatorias, ejecutar distintos algoritmos (búsqueda exhaustiva con profundidad limitada, Greedy y genético) y registrar sus resultados bajo una misma configuración.
- Verificar la consistencia interna de las métricas y del comportamiento de los algoritmos, comprobando que las soluciones generadas por cada enfoque concuerdan con lo esperado teóricamente.

En relación con este segundo objetivo, planteamos las siguientes hipótesis:

- La búsqueda exhaustiva con profundidad limitada alcanza el frente óptimo y sirve como referencia de validación para las demás aproximaciones.
- El algoritmo genético es capaz de reproducir soluciones exactas o casi óptimas bajo configuraciones razonables, lo que valida tanto su implementación como la interacción entre parámetros.
- El algoritmo Greedy-MO es más rápido, produce resultados coherentes con las medidas utilizadas, pero queda atrapado en óptimos locales.

Para poner a prueba estas hipótesis, es necesario definir los factores y variables de nuestro diseño experimental:

- Variables independientes:
  - Algoritmo empleado: búsqueda exhaustiva con profundidad limitada, Greedy-MO o genético (NSGA-II).
  - Tamaño y rango de las instancias: tamaño del universo $|U|$, número de elementos en $|G|$, número de conjuntos en $|F|$, tamaño de los subconjuntos $|F_i|$ y número máximo de operaciones k.
  - Parámetros de control del algoritmo genético: tamaño de la población, probabilidad de cruce, probabilidad de mutación, tamaño del torneo y tiempo máximo de ejecución.
  - Semilla de generación aleatoria (para reproducibilidad).
- Variables dependientes:
  - Correcto funcionamiento del entorno (tiempos de ejecución y ausencia de errores).

- Calidad de la solución: índice de Jaccard (a maximizar), número de subconjuntos distintos F_i utilizados (a minimizar) y número de operaciones en la expresión (a minimizar).
- Estructura del frente de Pareto: número total de soluciones no dominadas.

El entorno de ejecución sobre el que se realizan estos experimentos tiene la siguiente configuración de software y hardware:

- Hardware: procesador `Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz` (4 núcleos físicos, 8 hilos lógicos) con velocidad máxima de 4.90GHz, 15.4GiB de RAM.
- Software: compilador `g++` (C++17), sistema operativo Ubuntu 20.04.
- Código: definimos la configuración experimental en el archivo `config.yaml`, incluido en la Sección C.3, donde definimos los parámetros del universo, rangos de generación, semillas y rutas de salida.

Finalmente, el proceso de generación de instancias sigue este protocolo:

- El universo U tiene un tamaño fijo $|U| = 128$. Hemos elegido este valor porque es lo suficientemente grande como para simular un espacio de búsqueda complejo, pero lo suficientemente pequeño como para que las operaciones sean eficientes al usar el tipo de dato `std::bitset`. Es importante notar que, al fijar este valor, todos los términos de complejidad teórica que dependen de $n = |U|$ (como $O(n)$) se vuelven de orden constante en nuestro análisis práctico.
- El conjunto objetivo G se genera como un subconjunto aleatorio de U, con cardinal uniforme en el rango $[G_{min} = 10, |U|]$ y muestreo sin reemplazo.
- La familia de conjuntos $F = \{F_1, \ldots, F_m\}$ se genera con m uniformemente distribuido en $[F_{n,min}, F_{n,max}]$, y cada subconjunto F_i con tamaño uniforme en $[F_{i,min}, F_{i,max}]$.
- Los operadores disponibles en las expresiones son $\mathcal{O} = \{\cup, \cap, \setminus\}$.
- El parámetro k es la profundidad máxima de la expresión, es decir, el número de operaciones binarias permitidas. Como veremos a continuación, su valor variará según el experimento.

3.3.1. Arquitectura del entorno experimental y prácticas de ingeniería

Correspondiendo al primer objetivo principal de esta sección (construir un entorno experimental reproducible y extensible), el desarrollo no se centró únicamente en los algoritmos, sino también en la creación de una arquitectura de software robusta que lo soportara. En esta sección, detallamos las prácticas de ingeniería aplicadas y la arquitectura interna del código que fueron fundamentales para construir dicho entorno.

3.3.1.1. Requisitos

Comenzamos exponiendo los requisitos funcionales, que definen las capacidades operativas del sistema:

- **RF1 - Configuración externa:** el sistema debe permitir la configuración de los parámetros variables de los experimentos (número de conjuntos de la familia F, tamaños de los subconjuntos F_i, tamaño del conjunto objetivo G, semillas, profundidad de las expresiones k) mediante archivos externos (formato YAML) sin necesidad de recompilar el código para cada ejecución.
- **RF2 - Generación de instancias:** el sistema debe ser capaz de generar instancias del problema de forma pseudoaleatoria, respetando estrictamente todos los parámetros de configuración definidos.
- **RF3 - Ejecución de algoritmos:** el sistema debe permitir la ejecución selectiva de los tres algoritmos implementados (Exhaustivo, Greedy-MO, NSGA-II) sobre las mismas instancias para facilitar su comparación.
- **RF4 - Trazabilidad de resultados:** el entorno experimental debe encargarse de capturar y almacenar automáticamente los resultados generados por los algoritmos, estructurando la información (frentes de Pareto, métricas y datasets) en archivos de texto organizados para poder analizarlos posteriormente. Además de las semillas generadoras, el sistema debe mantener los valores concretos de las instancias utilizadas, garantizando la reproducibilidad incluso ante posibles variaciones en las implementaciones de las bibliotecas pseudoaleatorias.
- **RF5 - Reproducibilidad:** el sistema debe garantizar que, al ejecutar un algoritmo sobre una misma instancia y configuración, los resultados obtenidos sean deterministas e idénticos.

En cuanto a los requisitos no funcionales, que definen las restricciones y atributos de calidad, hemos definido los siguientes:

- **RNF1 - Eficiencia computacional:** la implementación de las estructuras de datos centrales debe mantener una arquitectura de datos que sea coherente y compatible con todos los algoritmos utilizados en el proyecto.
- **RNF2 - Extensibilidad:** la arquitectura debe ser modular, permitiendo la incorporación de nuevas medidas (M_i) o nuevos algoritmos sin alterar la lógica existente.
- **RNF3 - Portabilidad:** el código debe utilizar exclusivamente características estándar de `C++17` y bibliotecas estándar (STL) para asegurar su compilación y ejecución en distintos entornos sin dependencias complejas.

3.3.1.2. Prácticas de ingeniería aplicadas

Exponemos a continuación una serie de prácticas de ingeniería de software que aseguran la calidad del código, la fiabilidad de las ejecuciones y la trazabilidad de los resultados:

- **Modularidad y separación de responsabilidades:** dividimos el proyecto en varios componentes principales:
  - **`tfgcore/`:** contiene toda la lógica algorítmica (src/) y las definiciones de datos (include/). La lógica de cada algoritmo (genetico.cpp, greedy.cpp, etc.) está definida en su propio archivo, separada de la lógica de los demás.
  - **`scripts/`:** contiene los *scripts* de alto nivel que gestionan la experimentación. Esta capa es responsable de compilar, ejecutar y registrar los resultados.
  - **`config.yaml`:** centraliza todos los parámetros de los experimentos (semillas, k, N, m, límites de tiempo), permitiendo modificar las ejecuciones sin tocar el código fuente. Podemos encontrar el contenido de este archivo en la Sección C.3.
  - **`results/` y `build/`:** se separan del código fuente los resultados y *datasets* y los ejecutables compilados.
- **Automatización de experimentos:** hemos desarrollado dos *scripts* principales en Python (experimentos.py y exp_genetico.py) que automatizan todo el proceso de experimentación:
  1. Leen el `config.yaml`.
  2. Compilan el ejecutable (build/main).
  3. Ejecutan el programa C++ iterativamente para cada semilla definida.
  4. Capturan la salida estándar de cada ejecución.
  5. Parsean la salida para separar los resultados (en resultados.txt) de los datos de la instancia (en dataset.txt), asegurando la trazabilidad.
- **Validación (pruebas de caja blanca y negra):** durante el desarrollo, combinamos pruebas de "caja blanca" con pruebas de "caja negra" para asegurar el correcto funcionamiento de los algoritmos, así como la calibración del genético.
- **Validación científica (experimento 1):** diseñamos el experimento 1 como una prueba de validación del sistema completo. Al comparar las heurísticas (Greedy-MO, NSGA-II) contra la búsqueda exhaustiva de profundidad limitada en instancias pequeñas ($k = 3$), se pudo verificar empíricamente que los algoritmos se comportaban según lo esperado teóricamente.

3.3.1.3. Entorno de desarrollo y bibliotecas utilizadas

Toda la implementación la hemos realizado en `C++17`, compilado con `g++`, y sin dependencias externas. Hemos utilizado únicamente las bibliotecas de la STL[11], lo cual asegura portabilidad y reproducibilidad del código en cualquier entorno con un compilador compatible con el estándar `C++17` o superior.

Las bibliotecas utilizadas son las siguientes:

[11] En inglés, *Standard Template Library.*

- Contenedores de representación: `<bitset>` para la representación de conjuntos finitos, `<vector>`, `<set>`, y `<unordered_set>` para almacenar familias de conjuntos, poblaciones de candidatos y para garantizar la operación $O(1)$ de la medida de los $|F_i|$ usados.

- Gestión y tipos: utilizamos bibliotecas genéricas como `<string>`, `<iostream>` y `<algorithm>`, así como `<stdexcept>`, para el control de excepciones y `<cstddef>` para la gestión de tipos enteros.

- Aleatoriedad y reproducibilidad: `<random>` para la generación de números pseudoaleatorios utilizada en el algoritmo genético y en la inicialización de instancias.

3.3.1.4. Estructuras de datos

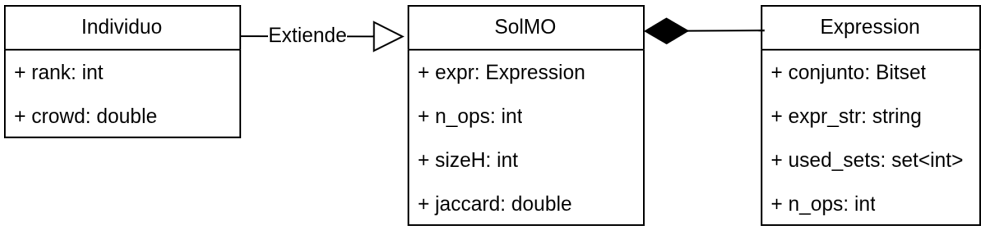

Figura 3.6.: Diagrama de clases simplificado de las estructuras de datos principales.

En la Figura 3.6 podemos ver una representación de las tres estructuras de datos principales de nuestra implementación:

- **`Expression`** es la base, la expresión en sí. Almacena el resultado de evaluar la expresión (el `Bitset` resultante) y su forma simbólica (un `string`). También almacena algunos datos estructurales como el número de operaciones y el conjunto de índices de los F_i distintos utilizados. Estos dos datos son útiles al calcular las medidas, pues su precálculo busca mejorar la eficiencia de los algoritmos.
- **`SolMO`** contiene la `Expression` y añade los valores de evaluarla mediante el vector de medidas. Es lo que usan la búsqueda exhaustiva con profundidad limitada y el Greedy-MO.
- **`Individuo`** extiende a `SolMO` porque es también una solución evaluada, que además añade información que solo el NSGA-II necesita (*rank* para el rango, *crowd* para la distancia de aglomeración).

La separación que hemos hecho entre estas tres estructuras, nos permite distinguir la solución (la `Expression`) de su evaluación en el espacio multiobjetivo (la `SolMO`), que es la que se utiliza en el filtrado de Pareto, y de una representación que contiene elementos que solo el algoritmo NSGA-II necesita.

En la Figura 3.6, es evidente como las tres estructuras están relacionadas entre sí, siendo `Expression` un atributo de `SolMO`, y `Individuo` una extensión de esta última.

3.3.1.5. Archivos principales

Como hemos mencionado, el código fuente del proyecto se organiza en una estructura modular que separa la implementación (`tfgcore`) de los *scripts* (`scripts/`) y de los resultados (`results/`). La Figura 3.7 muestra esta organización.

```
tfg-exp/
├── build/
│   └── main
├── results/
│   ├── batch
│   ├── exp_genetico
│   └── small
├── scripts/
│   ├── experimento_genetico.py
│   └── experimentos.py
├── tfgcore/
│   ├── include/
│   │   ├── domain.hpp
│   │   ├── exhaustiva.hpp
│   │   ├── expr.hpp
│   │   ├── generator.hpp
│   │   ├── genetico.hpp
│   │   ├── greedy.hpp
│   │   ├── ground_truth.hpp
│   │   ├── metrics.hpp
│   │   └── solutions.hpp
│   └── src/
│       ├── exhaustiva.cpp
│       ├── generator.cpp
│       ├── genetico.cpp
│       ├── greedy.cpp
│       ├── ground_truth.cpp
│       ├── main.cpp
│       └── metrics.cpp
├── CMakeLists.txt
└── config.yaml
```

Figura 3.7.: Diagrama de la estructura de directorios del proyecto.

La lógica central del proyecto (dentro de `tfgcore/`) se divide en los siguientes módulos de cabecera (`inc/`), que separan la lógica del dominio y los algoritmos, y que facilitan la reutilización y extensión del código:

- **`domain.hpp`**: define el alias `Bitset = std::bitset<U_SIZE>`, donde `U_SIZE` es un parámetro configurable que permite adaptar el tamaño del universo. El archivo también define las funciones de operaciones de Bitsets (por ejemplo, `set_union`).

- **`expr.hpp`**: define la estructura `Expression`.
- **`solutions.hpp`**: define las estructuras que "envuelven" a una `Expression`: `SolMO` e `Individuo`.
- **`metrics.hpp`**: define el enumerado `Metric` (para darle nombre a las medidas) y la función principal de evaluación `M(...)`. También implementa la lógica de dominancia (dominates) y el algoritmo de filtrado de Pareto (pareto_front).
- **`generator.hpp`**: contiene la lógica para generar instancias aleatorias del problema (crear G y la familia F).
- **`exhaustiva.hpp`, `greedy.hpp`, `genetico.hpp`:** cada uno de estos archivos contiene la implementación del algoritmo correspondiente.
- **`ground_truth.hpp`**: contiene la lógica para generar las instancias (conjunto F, expresión de referencia y conjunto G a partir de dicha expresión) del experimento 2.

La Figura 3.8 muestra qué archivos utilizan a qué otros. Destacamos que los módulos de los algoritmos son independientes entre sí, lo que valida la modularidad y facilita la extensión.

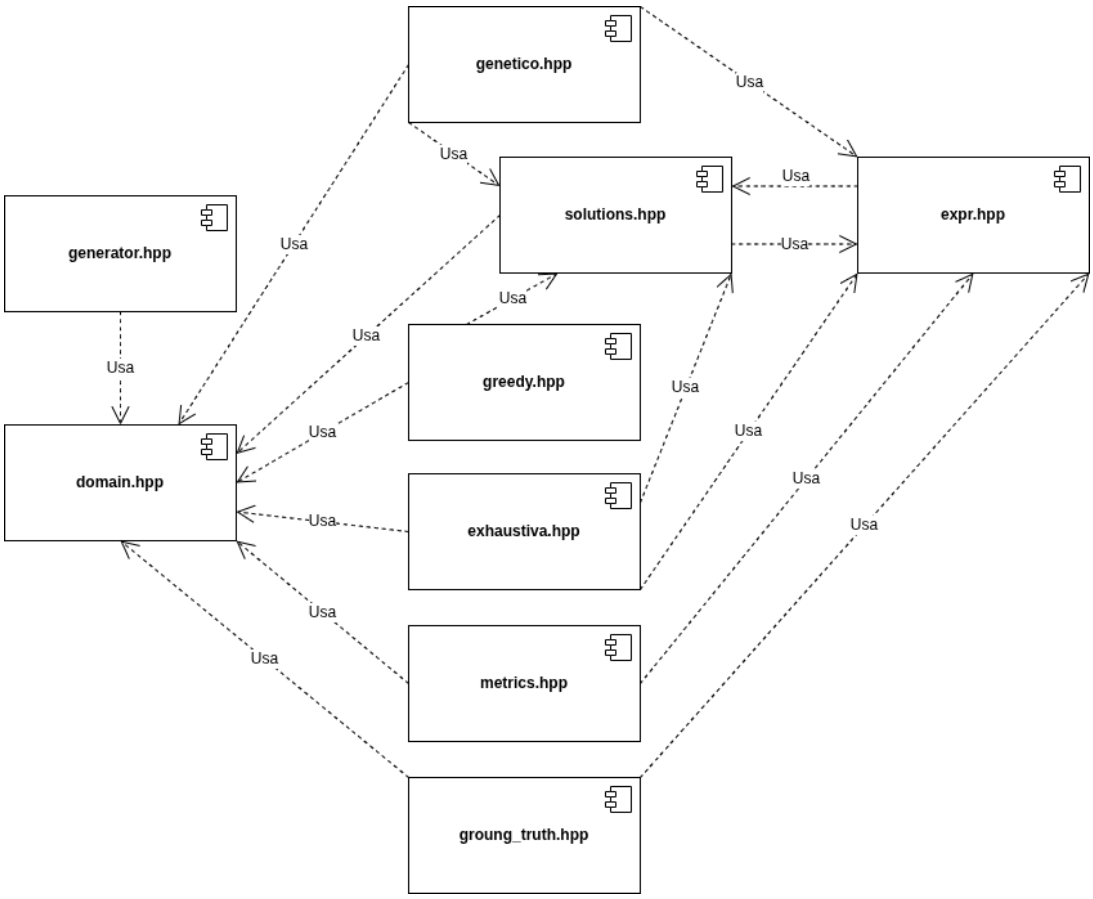

Figura 3.8.: Diagrama que muestra la relación de inclusión entre los archivos principales.

3.3.1.6. Implementación de las funciones objetivo y dominancia

A parte de las tres estructuras de datos principales ya mencionadas, gestionamos también las funciones objetivo $\mathcal{M}$ mediante dos componentes clave. El primero es el enumerado `Metric`, que proporciona un identificador único para cada una de las medidas (M_1, M_2, M_3). Esto nos permite definir de forma flexible qué subconjunto de medidas se emplea en un experimento dado, así como incorporar fácilmente nuevas medidas. Este identificador se utiliza en la función de evaluación que recibe la expresión a evaluar, el conjunto objetivo y el identificador de la medida, y se encarga de llevar a cabo el cálculo correspondiente asegurando los órdenes que habíamos visto en la Sección 3.1.

Una vez que hemos evaluado una solución y sus tres objetivos están almacenados en la estructura `SolMO`, se procede al filtrado de Pareto. La lógica de dominancia (definida matemáticamente en la Sección 3.1) se implementa en una función booleana que determina si una `SolMO` domina a otra. A su vez, un algoritmo de filtrado de Pareto utiliza esta función de dominancia para iterar sobre un conjunto de soluciones y obtener el frente final. Finalmente, para garantizar una presentación determinista y ordenada de las soluciones en los resultados, las ordenamos según la siguiente jerarquía de prioridad: Jaccard descendente, número de F_i distintos ascendente y número de operaciones ascendente.

3.3.1.7. Guía de extensibilidad (añadir una métrica)

El diseño modular (específicamente `metrics.hpp`) permite añadir nuevas medidas de forma sencilla. Para añadir una nueva métrica (por ejemplo, M_4):

1. **Definir la métrica:** añadir el nombre de la nueva medida a `enum class Metric` en `metrics.hpp`.
2. **Actualizar funciones (en `metrics.hpp`):** añadir el `case` correspondiente en las funciones `metric_name` (para los registros) y `is_maximization` (para la lógica de dominancia).
3. **Implementar la lógica (en `metrics.cpp`):** añadir un nuevo `case` al `switch` de la función `M(const Expression& H, ...)` que calcule y devuelva el valor de M_4.
4. **Actualizar la dominancia:** modificar la función `dominates(a, b)` en `metrics.cpp` para que compare el nuevo cuarto objetivo.
5. **Actualizar el crowding (para el NSGA-II):** añadir un nuevo bloque de ordenación (`std::sort`) para M_4 en la función `calcular_crowding_distance` en `genetico.cpp` para que la diversidad se mida en 4 dimensiones.

Cabe destacar que si la nueva métrica M_4 es estructural (como lo son M_2 y M_3), calcularla dentro de la función M en cada evaluación, puede ser ineficiente y crear un cuello de botella. Para dichos casos más complejos, añadir la nueva métrica podría requerir pasos adicionales para precalcular el valor, como modificar la estructura `Expression`; actualizar las funciones `build_random_expr`, `crossover` y `mutar` (todas en genetico.cpp); y actualizar la función `M(...)` para que simplemente devuelva el valor precalculado.

3.3.2. Protocolos por experimento

Una vez detallada la arquitectura del entorno experimental (correspondiente al primer objetivo de esta sección), procedemos a describir los protocolos diseñados para cumplir con el segundo objetivo: verificar la consistencia interna y el comportamiento de los algoritmos.

- Experimento 1 (instancias pequeñas y validación de referencia):
  - Parámetros: $|U| = 128$, $G_{\min} = 10$, $F_n \in [5, 20]$, $|F_i| \in [5, 50]$, $k = 3$, límite de tiempo del NSGA-II $= 150$ s.
  - Semillas: 4 (1002 a 1005, este rango fue escogido simplemente por diferenciar del rango de otros experimentos y separar los conjuntos de pruebas).
  - Algoritmos: búsqueda exhaustiva con profundidad limitada, Greedy y genético (NSGA-II).
  - Salida: tiempo de ejecución, número de soluciones no dominadas, índice de Jaccard máximo y frente de Pareto completo.
  - Propósito: verificar la integración del entorno y la coherencia entre los tres métodos en instancias donde el frente óptimo puede calcularse exhaustivamente.
- Experimento 2 (validación de alcance de óptimo conocido):
  - Cada instancia se genera de forma que se garantiza la existencia de al menos una expresión con Jaccard = 1 (un óptimo global conocido).
  - Parámetros: $|U| = 128$, $G_{\min} = 10$, $F_n \in [5, 100]$, $F_i \in [5, 100]$, $k = 10$, límite de tiempo del NSGA-II $= 900$ s.
  - Instancias: 20 instancias generadas aleatoriamente.
  - Algoritmos: genético (NSGA-II).
  - Propósito: comparar la capacidad del Greedy y el NSGA-II para converger al óptimo global en un escenario donde sabemos que este existe al menos una expresión con Jaccard=1.
- Experimento 3 (robustez y escalabilidad del entorno):
  - Parámetros: $|U| = 128$, $G_{\min} = 10$, $F_n \in [5, 100]$, $F_i \in [5, 100]$, $k = 10$, límite de tiempo del NSGA-II $= 900$ s.
  - Semillas: 50 (2001-2050, de nuevo, este rango fue escogido simplemente por diferenciar del rango de otros experimentos y separar los conjuntos de pruebas).
  - Algoritmos: Greedy y genético (NSGA-II).
  - Salida: tiempo de ejecución, número de soluciones no dominadas y mejor índice de Jaccard.
  - Propósito: comparar el rendimiento y la robustez de ambas heurísticas en un escenario de propósito general, y evaluar la escalabilidad y estabilidad del laboratorio experimental ante un mayor número de instancias y una mayor complejidad ($k = 10$).

3.3.3. Experimentos

Podemos ahora adentrarnos en la explicación de cada experimento, así como en ver los resultados que producen y las conclusiones que podemos sacar de ellos. Para garantizar la total reproducibilidad de este trabajo, todos los archivos de código, los *datasets* de instancias generados y los resultados de todos los experimentos se han publicado en un repositorio de GitHub, cuya dirección y contenido se detallan en el Apéndice C.

3.3.3.1. Experimento 1 - Instancias pequeñas

Este primer experimento tiene como objetivo ilustrar el proceso de construcción de soluciones de cada algoritmo y, a la vez, validar nuestras hipótesis de comportamiento en un entorno controlado. Para ello, lo ejecutamos en un escenario de complejidad reducida, de modo que la búsqueda exhaustiva con profundidad limitada pueda ejecutarse en un tiempo razonable para devolver el frente de Pareto real del subespacio. Al disponer de esta referencia óptima, podemos ilustrar y verificar empíricamente el comportamiento de las heurísticas. El objetivo, por tanto, no es una comparación de rendimiento, sino una validación cualitativa del comportamiento esperado de los algoritmos.

Para ello, ejecutamos cuatro instancias independientes (semillas 1002-1005) con:

$$|U| = 128, \quad k = 3, \quad m \sim \mathcal{U}(5, 20), \quad |F_i| \sim \mathcal{U}(5, 50), \quad |G| \geq 10.$$

Los parámetros del NSGA-II para este experimento son los definidos en la Sección 3.2.3 ($N = 150$, $P_c = 0.8$, $P_m = 0.5$, Tamaño de torneo $= 5$). Establecemos el límite de tiempo en 150 s, pues tras ejecuciones preliminares, encontramos que era suficiente para encontrar el frente de Pareto óptimo del subespacio.

Registramos en la Tabla 3.2 el frente de Pareto resultante, definido por el espacio de objetivos $\mathcal{M}$, junto con el tiempo de ejecución total (en milisegundos) y el número de soluciones no dominadas obtenidas.

Para ilustrar el comportamiento de los algoritmos, analizamos en detalle la primera instancia (Semilla 1002). Los datos de entrada generados para esta instancia, extraídos del archivo del dataset, son los siguientes:

- Tamaño universo U: 128
- Conjunto G: $x_5, x_{13}, x_{17}, x_{19}, x_{23}, x_{27}, x_{31}, x_{42}, x_{43}, x_{44}, x_{45}, x_{53}, x_{59}, x_{67}, x_{69}, x_{70}, x_{77}, x_{78}, x_{84}, x_{88}, x_{90}, x_{103}, x_{113}, x_{116}, x_{124}$
- Conjuntos F_i:
  - $F_0 : x_{13}, x_{19}, x_{53}, x_{59}, x_{77}, x_{124}$
  - $F_1 : x_{17}, x_{27}, x_{31}, x_{42}, x_{43}, x_{44}, x_{53}, x_{67}, x_{78}, x_{84}, x_{103}, x_{113}, x_{116}$
  - $F_2 : x_1, x_4, x_5, x_6, x_{19}, x_{20}, x_{23}, x_{26}, x_{29}, x_{33}, x_{37}, x_{43}, x_{44}, x_{45}, x_{48}, x_{53}, x_{56}, x_{69}, x_{78}, x_{84}, x_{86}, x_{88}, x_{90}, x_{96}, x_{97}, x_{103}, x_{106}, x_{109}, x_{111}, x_{115}$
  - $F_3 : x_{49}, x_{94}, x_{102}, x_{113}, x_{115}$

- $F_4 : x_3, x_4, x_5, x_8, x_{22}, x_{28}, x_{30}, x_{31}, x_{34}, x_{36}, x_{38}, x_{44}, x_{50}, x_{53}, x_{57}, x_{58}, x_{74}, x_{77}, x_{78}, x_{95}, x_{107}, x_{108}, x_{116}, x_{119}, x_{125}, x_{126}$
- $F_5 : x_{11}, x_{12}, x_{32}, x_{58}, x_{90}, x_{93}, x_{94}, x_{96}, x_{103}, x_{106}, x_{107}, x_{112}, x_{113}, x_{115}$
- $F_6 : x_{31}, x_{44}, x_{46}, x_{67}, x_{95}, x_{100}, x_{103}$

Tabla 3.2.: Lista completa de soluciones no dominadas en la instancia ilustrativa (Semilla 1002, $k = 3$). Correspondiente al archivo results/small/20251108_211724_resultados_small.txt.

Algoritmo	**Expresión**	**Jaccard**	$\lvert\mathcal{F}(e)\rvert$	$\lvert\mathcal{O}p^*(e)\rvert$
Búsqueda exhaustiva de profundidad limitada				
Solución 1	$(F0 \cup (F1 \cup (F2 \cap F4)))$	0.731	4	3
Solución 2	$(F0 \cup (F1 \cup (F4 \cap F2)))$	0.731	4	3
Solución 3	$(F0 \cup ((F2 \cap F4) \cup F1))$	0.731	4	3
Solución 4	$(F0 \cup ((F4 \cap F2) \cup F1))$	0.731	4	3
Solución 5	$(F1 \cup (F0 \cup (F2 \cap F4)))$	0.731	4	3
Solución 6	$(F1 \cup (F0 \cup (F4 \cap F2)))$	0.731	4	3
Solución 7	$(F1 \cup ((F2 \cap F4) \cup F0))$	0.731	4	3
Solución 8	$(F1 \cup ((F4 \cap F2) \cup F0))$	0.731	4	3
Solución 9	$((F0 \cup F1) \cup (F2 \cap F4))$	0.731	4	3
Solución 10	$((F0 \cup F1) \cup (F4 \cap F2))$	0.731	4	3
Solución 11	$((F1 \cup F0) \cup (F2 \cap F4))$	0.731	4	3
Solución 12	$((F1 \cup F0) \cup (F4 \cap F2))$	0.731	4	3
Solución 13	$((F2 \cap F4) \cup (F0 \cup F1))$	0.731	4	3
Solución 14	$((F2 \cap F4) \cup (F1 \cup F0))$	0.731	4	3
Solución 15	$((F4 \cap F2) \cup (F0 \cup F1))$	0.731	4	3
Solución 16	$((F4 \cap F2) \cup (F1 \cup F0))$	0.731	4	3
Solución 17	$((F0 \cup (F2 \cap F4)) \cup F1)$	0.731	4	3
Solución 18	$((F0 \cup (F4 \cap F2)) \cup F1)$	0.731	4	3
Solución 19	$((F1 \cup (F2 \cap F4)) \cup F0)$	0.731	4	3
Solución 20	$((F1 \cup (F4 \cap F2)) \cup F0)$	0.731	4	3
Solución 21	$(((F2 \cap F4) \cup F0) \cup F1)$	0.731	4	3
Solución 22	$(((F2 \cap F4) \cup F1) \cup F0)$	0.731	4	3
Solución 23	$(((F4 \cap F2) \cup F0) \cup F1)$	0.731	4	3
Solución 24	$(((F4 \cap F2) \cup F1) \cup F0)$	0.731	4	3
Solución 25	$(F0 \cup F1)$	0.720	2	1
Solución 26	$(F1 \cup F0)$	0.720	2	1
Solución 27	$F1$	0.520	1	0
Solución 28	U	0.195	0	0
Greedy-MO				
Solución 1	$(F1 \cup F0)$	0.720	2	1
Solución 2	$F1$	0.520	1	0
Solución 3	U	0.195	0	0

Continúa en la siguiente página...

Tabla 3.2 – continuación de la página anterior

Algoritmo	Expresión	$M_{Jaccard}$	$\lvert\mathcal{F}(e)\rvert$	$\lvert\mathcal{O}p^*(e)\rvert$
Genético (NSGA-II)				
Solución 1	$(((F2 \cap F4) \cup F1) \cup F0)$	0.731	4	3
Solución 2	$(F1 \cup ((F2 \cap F4) \cup F0))$	0.731	4	3
Solución 3	$((F4 \cap F2) \cup (F1 \cup F0))$	0.731	4	3
Solución 4	$(F0 \cup ((F2 \cap F4) \cup F1))$	0.731	4	3
Solución 5	$(F0 \cup ((F4 \cap F2) \cup F1))$	0.731	4	3
Solución 6	$((F0 \cup (F2 \cap F4)) \cup F1)$	0.731	4	3
Solución 7	$((F0 \cup (F4 \cap F2)) \cup F1)$	0.731	4	3
Solución 8	$(F1 \cup ((F4 \cap F2) \cup F0))$	0.731	4	3
Solución 9	$((F2 \cap F4) \cup (F1 \cup F0))$	0.731	4	3
Solución 10	$(((F4 \cap F2) \cup F0) \cup F1)$	0.731	4	3
Solución 11	$(F1 \cup (F0 \cup (F2 \cap F4)))$	0.731	4	3
Solución 12	$((F1 \cup (F2 \cap F4)) \cup F0)$	0.731	4	3
Solución 13	$((F2 \cap F4) \cup (F0 \cup F1))$	0.731	4	3
Solución 14	$(((F4 \cap F2) \cup F1) \cup F0)$	0.731	4	3
Solución 15	$((F1 \cup (F4 \cap F2)) \cup F0)$	0.731	4	3
Solución 16	$(F1 \cup (F0 \cup (F4 \cap F2)))$	0.731	4	3
Solución 17	$((F1 \cup F0) \cup (F2 \cap F4))$	0.731	4	3
Solución 18	$((F0 \cup F1) \cup (F2 \cap F4))$	0.731	4	3
Solución 19	$((F4 \cap F2) \cup (F0 \cup F1))$	0.731	4	3
Solución 20	$((F1 \cup F0) \cup (F4 \cap F2))$	0.731	4	3
Solución 21	$(F0 \cup (F1 \cup (F2 \cap F4)))$	0.731	4	3
Solución 22	$(F0 \cup (F1 \cup (F4 \cap F2)))$	0.731	4	3
Solución 23	$((F0 \cup F1) \cup (F4 \cap F2))$	0.731	4	3
Solución 24	$(((F2 \cap F4) \cup F0) \cup F1)$	0.731	4	3
Solución 25	$(F0 \cup F1)$	0.720	2	1
Solución 26	$(F1 \cup F0)$	0.720	2	1
Solución 27	$F1$	0.520	1	0
Solución 28	U	0.195	0	0

La ejecución de los tres algoritmos sobre esta instancia $(k = 3, m = 7)$ produjo los siguientes resultados:

- La búsqueda exhaustiva con profundidad limitada encontró el frente óptimo (28 soluciones) en 430 ms.
- El Greedy encontró 3 soluciones en menos de 1 ms.
- El NSGA-II encontró el frente óptimo (28 soluciones) en el tiempo límite asignado de 150 s.

De la Tabla 3.2 extraemos varios hallazgos clave. El primero es que tanto la búsqueda exhaustiva con profundidad limitada como el NSGA-II encontraron 28 soluciones no dominadas que, debido a la conmutatividad y asociatividad de las operaciones, se reducen a 4 valores únicos en el espacio de objetivos.

El segundo hallazgo valida nuestras hipótesis:

- El Greedy-MO quedó atrapado en un óptimo local. Su mejor solución fue $(F1 \cup F0)$ ($M_{Jaccard} = 0.720$, $k = 1$). Vemos que de todas las expresiones en el frente de Pareto de la búsqueda exhaustiva de profundidad limitada, ninguna esta construida de forma secuencial a partir de $(F_1 \cup F_0)$. Las soluciones de $k = 2$ que generó el Greedy-MO fueron dominadas por la solución con $k = 1$, que tuvo mejor índice de Jaccard y menor complejidad, haciendo que el algoritmo terminara, demostrando la limitación de su generación secuencial.
- El NSGA-II fue capaz de converger al frente de Pareto óptimo real del subespacio correspondiente. Este resultado ilustra que nuestra implementación explora el espacio de búsqueda de forma efectiva y valida la hipótesis de su convergencia para instancias de esta complejidad.

Los resultados de las otras tres semillas confirman este patrón de comportamiento:

- Semilla 1003: la búsqueda exhaustiva con profundidad limitada (6.4 s) y el NSGA-II (150 s) encontraron el mismo frente óptimo de 136 soluciones ($M_{Jaccard}$ máx. 0.975). El Greedy ($<$ 1ms) quedó atrapado en un frente subóptimo ($M_{Jaccard}$ máx. 0.681), encontrando 4 soluciones en su frente de Pareto.
- Semilla 1004: esta instancia fue más simple. La búsqueda exhaustiva con profundidad limitada (125 ms) y el NSGA-II (150 s) encontraron el mismo frente óptimo de 4 soluciones ($M_{Jaccard}$ máx. 0.545). El Greedy ($<$ 1 ms) encontró un frente de 3 soluciones, fallando en encontrar la mejor solución.
- Semilla 1005: la búsqueda exhaustiva con profundidad limitada (8.0 s) y el NSGA-II (150 s) encontraron el mismo frente óptimo de 135 soluciones ($M_{Jaccard}$ máx. 0.945). El Greedy ($<$ 1 ms) encontró solo la solución trivial U ($M_{Jaccard} = 0.6875$).

Observamos que las soluciones generadas por el algoritmo Greedy se construyen de manera secuencial, añadiendo en cada paso un par (op, F_i) sobre el estado actual, lo que refleja su naturaleza determinista y local, como ya hemos explicado. En cambio, tanto la búsqueda exhaustiva con profundidad limitada como el genético generan expresiones con estructuras más variadas, que permiten alcanzar, en este subespacio, el frente completo de soluciones no dominadas.

3.3.3.2. Experimento 2 - Validación de alcance de óptimo conocido

El segundo experimento tiene un propósito diferente: validar la capacidad de convergencia de la heurística Greedy-MO frente a la metaheurística NSGA-II en un escenario donde se conoce que existe al menos una expresión que alcanza $M_{Jaccard} = 1$. Para ello generamos 20 instancias independientes con:

$$|U| = 128, \quad k = 10, \quad m \sim \mathcal{U}(5, 100), \quad |F_i| \sim \mathcal{U}(5, 100), \quad |G| \geq 10.$$

En cada una, el conjunto objetivo G se construye a partir de una expresión de referencia e^* (con $k \leq 10$) generada previamente ($G = eval(e^*)$). Este proceso garantiza que para cada

una de las 20 instancias existe una solución óptima en $M_{Jaccard}$, pero no necesariamente en $|\mathcal{O}p^*(e)|$ o $|\mathcal{F}(e)|$.

Los parámetros del NSGA-II para este experimento son los definidos en la Sección 3.2.3 ($N = 150$, $P_c = 0.8$, $P_m = 0.5$, tamaño de torneo $= 5$). Estos son los mismos parámetros utilizados en el Experimento 1. La única diferencia es que en este experimento ampliamos el límite de tiempo a 900 segundos por instancia. Este límite lo escogimos tras ejecuciones preliminares, y representa un compromiso realista dentro del tiempo disponible para la experimentación de este trabajo, pues es suficiente para permitir la convergencia en la mayoría de los casos, sin conllevar en un coste computacional excesivo por mejoras pequeñas.

Dado que el espacio de búsqueda con $k = 10$ y m hasta 100 es inmenso, las hipótesis son las siguientes:

- El Greedy-MO (determinista y local) quedará atrapado en óptimos locales de baja calidad (índice de Jaccard bajo) de forma consistente.
- El NSGA-II (estocástico y global) demostrará su robustez al encontrar frentes de Pareto que dominen sistemáticamente a los frentes del Greedy en todas las instancias.

La Tabla 3.3 resume los resultados de la ejecución de ambos algoritmos sobre las 20 instancias. Registramos el mejor índice de Jaccard alcanzado por cualquier solución en el frente final, el número de soluciones que alcanzan dicho valor, y el tiempo de ejecución. Además, mostramos las medidas $|\mathcal{O}p^*|$ y $|\mathcal{F}|$ de dichas soluciones con mejor $M_{Jaccard}$. Para el Greedy-MO mostramos también el tiempo de ejecución.

Tabla 3.3.: Resultados del Experimento 2 (Validación de alcance de $M_{Jaccard} = 1.0$, $k = 10$). Correspondientes al archivo results/exp_genetico/2025-11-17_18-10-05.

	NSGA-II				**Greedy-MO**				
Semilla	**Mejor** $M_{Jaccard}$	**Nº Sols.**	$\lvert\mathcal{O}p^*\rvert$	$\lvert\mathcal{F}\rvert$	**Mejor** $M_{Jaccard}$	**Nº Sols.**	$\lvert\mathcal{O}p^*\rvert$	$\lvert\mathcal{F}\rvert$	**Tiempo** (ms)
1000	**1.0**	8	2	3	0.957	1	0	1	< 1
1010	**1.0**	2	4	5	0.755	1	2	3	< 1
1020	**1.0**	147	2	3	**1.0**	8	2	3	4
1030	0.975	1	4	5	0.843	2	3	4	1
1040	0.892	1	10	6	0.800	1	1	2	< 1
1050	0.9	2	9	6	0.862	1	1	2	1
1060	**1.0**	1	5	5	0.867	1	2	2	< 1
1070	0.8	31	5	6	0.600	1	4	5	1
1080	**1.0**	128	3	4	**1.0**	4	3	4	2
1090	**1.0**	2	1	2	**1.0**	1	1	2	< 1
2000	0.959	23	4	5	0.861	2	6	5	4
2010	**1.0**	1	2	3	0.833	1	1	2	< 1
2020	0.83	88	4	5	0.667	2	2	3	3
2030	0.979	9	4	5	0.787	1	1	2	< 1
2040	**1.0**	15	3	4	0.891	1	0	0	< 1
2050	0.913	1	4	5	0.652	1	2	3	< 1
2060	**1.0**	92	4	5	**1.0**	1	4	5	1
2070	**1.0**	2	4	4	0.750	1	2	3	< 1
2080	**1.0**	28	3	4	0.963	1	0	1	< 1
2090	0.853	1	0	1	0.853	1	0	1	< 1
Tasa éxito	**11 / 20 (55 %)**				**4 / 20 (20 %)**				

Estos resultados están en línea con las hipótesis del experimento. Por un lado, en ninguna de las 20 instancias el Greedy-MO obtuvo un índice de Jaccard superior al del NSGA-II, haciendo evidente la dominancia de este último en el objetivo $M_{Jaccard}$. El NSGA-II (55 % de éxito) fue significativamente más robusto que el Greedy-MO (20 % de éxito) para encontrar la solución de $M_{Jaccard} = 1$. Esta robustez, sin embargo, tiene un coste evidente. El algoritmo Greedy-MO fue extremadamente rápido (su tiempo máximo fue de 4 ms y en la mayoría de los casos fue < 1 ms) mientras que el NSGA-II utilizó los 900 s que se le asignaron. En 9 de las 20 instancias (45 %), el NSGA-II no logró encontrar la solución con $M_{Jaccard} = 1$ en el límite de 900 s, lo cual demuestra la alta complejidad del espacio de búsqueda. Sin embargo, en esos casos, el algoritmo genético se aproximó mucho al óptimo, en particular, más que el algoritmo Greedy-MO (por ejemplo, semilla 1030, $M_{Jaccard} = 0.975$ vs. $M_{Jaccard} = 0.843$; semilla 2030, $M_{Jaccard} = 0.979$ vs $M_{Jaccard} = 0.787$).

Podemos destacar que el algoritmo Greedy-MO sí tuvo éxito en 4 instancias (20 %). Esto indica que, para ciertas configuraciones de instancias, el camino determinista del Greedy-MO coincide con el camino hacia al menos una solución con el óptimo en índice de Jaccard. Analizando la Tabla 3.3, nos damos cuenta de que en las 4 instancias donde ambos algoritmos alcanzaron el $M_{Jaccard} = 1.0$, las soluciones del Greedy-MO fueron de complejidad idéntica a las del NSGA-II. Sin embargo, la hipótesis sobre la simplicidad del Greedy-MO se valida en algunos de los casos en que no se alcanzó $M_{Jaccard} = 1.0$. Por ejemplo, en la Semilla 1050, el Greedy-MO se quedó atascado en un óptimo local con $M_{Jaccard} = 0.862$ (y $|\mathcal{O}p^*| = 1$), mientras que el NSGA-II aceptó una solución de mayor complejidad ($|\mathcal{O}p^*| = 9$) para alcanzar un índice de Jaccard más alto (0.9). Esto demuestra que el NSGA-II es capaz de navegar soluciones más complejas para optimizar el objetivo $M_{Jaccard}$, mientras que el Greedy-MO se ve limitado a óptimos locales que, a menudo, coinciden con expresiones más simples.

3.3.3.3. Experimento 3 - Instancias grandes

En este último experimento nos centramos en evaluar el comportamiento de las heurísticas en un escenario de optimización y complejo. La diferencia fundamental con el experimento anterior es que ahora las instancias se generan de forma puramente aleatoria, sin garantía de que exista una solución con $M_{Jaccard} = 1.0$.

Generamos 50 instancias independientes (semillas 2001–2050) con:

$$|U| = 128, \quad k = 10, \quad m \sim \mathcal{U}(5, 100), \quad |F_i| \sim \mathcal{U}(5, 100), \quad |G| \geq 10.$$

Los parámetros del NSGA-II son los definidos en la Sección 3.2.3 ($N = 150$, $P_c = 0.8$, $P_m = 0.5$, Torneo= 5) y, al igual que en el experimento anterior, se utilizó un límite de tiempo de 900 segundos por instancia.

Dado que el espacio de búsqueda es, de nuevo, inmenso y no hay garantía de una solución con $M_{Jaccard} = 1.0$, las hipótesis son las mismas que en el experimento anterior:

- El Greedy-MO (determinista y local) quedará atrapado en óptimos locales de baja calidad (Jaccard bajo) de forma consistente.

- El NSGA-II (estocástico y global) demostrará su robustez al encontrar frentes de Pareto que dominen sistemáticamente a los frentes del Greedy-MO en todas las instancias.

Los resultados estadísticos de las 50 ejecuciones, se resumen en la Tabla 3.4:

Tabla 3.4.: Resumen Estadístico del Experimento 3 (50 Instancias, $k = 10$).
Correspondientes al archivo results/batch/20251121_001943_resultados_batch.

Métrica	**Algoritmo**	**Media**	**Desv. Típica**	**Mediana**	**Mín**	**Máx**
Mejor $M_{Jaccard}$	Greedy-MO	0.722	0.245	0.780	0.167	0.992
	NSGA-II	0.802	0.231	0.926	0.184	1.000
Tamaño Frente	Greedy-MO	2.8	1.6	3	1	7
	NSGA-II	79.2	56	65	2	150
$\lvert\mathcal{O}p^*(e)\rvert$	Greedy-MO	1.0	1.1	1	0	3
(del mejor Jaccard)	NSGA-II	6.7	3.4	8	0	10
$\lvert\mathcal{F}(e)\rvert$	Greedy-MO	1.6	1.4	2	0	4
(del mejor Jaccard)	NSGA-II	5.8	2.1	7	1	8
Tiempo (ms)	Greedy-MO	0.3	1.5	< 1	< 1	10
	NSGA-II	900000	1.1	900000	900000	900002

Los datos de la tabla validan las hipótesis planteadas. Observamos un claro compromiso entre la calidad de la solución y el coste computacional. El NSGA-II es significativamente superior. Obtuvo un índice de Jaccard igual o superior en el 100 % de las 50 instancias. La mediana del "mejor $M_{Jaccard}$" del NSGA-II (0.926) es mucho mayor que la del Greedy-MO (0.780), lo que demuestra su robustez para escapar de óptimos locales. El Greedy-MO fue prácticamente instantáneo (mediana de < 1 ms), mientras que el NSGA-II utilizó su límite de 900 segundos. El NSGA-II produjo frentes de Pareto mucho más diversos (mediana de 65 soluciones) que el Greedy-MO (mediana de 3 soluciones).

Finalmente, los datos de complejidad ($|\mathcal{O}p^*|$ y $|\mathcal{F}|$) validan las hipótesis. El NSGA-II exploró soluciones mucho más complejas (mediana de 8 operaciones y 7 conjuntos) para encontrar sus mejores resultados, mientras que el Greedy-MO se limitó a soluciones simples (mediana de 1 operación y 2 conjuntos).

Es en este contexto donde se explica el valor mínimo de 0 en la tabla para ambas métricas: este 0 corresponde a la solución trivial U (0 operaciones y 0 conjuntos F_i). Aunque ambos algoritmos la encontraron (demostrando que el espacio de búsqueda la contenía), esta expresión suele tener un índice de Jaccard muy bajo y, por tanto, no es representativa de la calidad de las soluciones que el NSGA-II es capaz de descubrir.

A pesar de que la tabla anterior muestra una clara diferencia en el rendimiento medio (media de $M_{Jaccard}$ de 0.802 en NSGA-II frente a 0.722 en Greedy), para validar que esta diferencia no es cuestión de azar, es necesario aplicar un test de hipótesis estadístico.

Para seleccionar el test adecuado, debemos considerar la naturaleza de nuestros datos. En primer lugar, los datos no siguen una distribución normal (ya que los valores del índice de Jaccard se agrupan en el extremo superior), por lo que debemos descartar test paramétricos.

En segundo lugar, y más importante, nuestras muestras son dependientes, ya que ambos algoritmos se ejecutaron sobre el mismo conjunto de 50 semillas. En estas condiciones, el Test de los Rangos Signados de Wilcoxon es el ideal [HWC14]. Este test nos permite comprobar si la mediana de las diferencias entre los pares de resultados es significativamente distinta de cero, dándonos una medida robusta de si un algoritmo es consistentemente superior al otro.

Para realizar el test, establecemos nuestra hipótesis nula y nuestra hipótesis alternativa. Dado que queremos comprobar si el NSGA-II es mejor que el Greedy, utilizamos un test de una cola. Sea

$$D_i = Jaccard_{NSGA-II,i} - Jaccard_{Greedy,i}.$$

Si los dos algoritmos fueran iguales, la mediana de estas diferencias debería ser cero. por lo tanto, nuestro contraste es de la forma:

- Hipótesis nula ($H_0 : M_D = 0$): la mediana de las diferencias entre los índices de Jaccard es cero (NSGA-II - Greedy). Es decir, no hay diferencia estadística entre los dos algoritmos.
- Hipótesis alternativa ($H_1 : M_D > 0$): la mediana de las diferencias es mayor que cero. Es decir, el NSGA-II produce resultados de índice de Jaccard estadísticamente superiores a los del Greedy-MO.

Establecemos el nivel de significación estándar $\alpha = 0.05$. Esto significa que si el test devuelve un p (p-valor) inferior a 0.05, tendremos la confianza estadística suficiente (95 %) para rechazar la hipótesis nula (H_0) y aceptar que la mejora del NSGA-II es significativa.

Aplicamos el test a los 50 pares dados del "mejor $M_{Jaccard}$" del experimento, cuyos valores completos se pueden consultar en la Tabla B.1. Calculamos las diferencias $D_i = X_i - m$. Nuestra $m = 0$, por lo que $X_i = D_i$. Primero descartamos los pares donde la diferencia entre NSGA-II y Greedy-MO es cero. Esto se da en cuatro casos: las semillas 2002, 2005, 2010, y 2027. Por ello, el test se realiza sobre las $N = 46$ instancias restantes. Este descarte de empates es un paso estándar del test de Wilcoxon. Las diferencias nulas (empates exactos) se eliminan del análisis porque no tienen signo (ni positivo ni negativo) y, por lo tanto, no contribuyen al estadístico de prueba.

El siguiente paso es calcular el estadístico T^+. Para ello, primero se asignan rangos a las $N = 46$ diferencias no nulas: se toma su valor absoluto $|D_i|$, se ordenan de menor a mayor, y se les asigna un rango de 1 (a la diferencia más pequeña) hasta $N = 46$ (a la más grande).

Una vez asignados, el estadístico $T^+ \equiv$ se define como la suma de los rangos que corresponden a las diferencias D_i positivas. Como podemos observar en la tabla del Apéndice B, las 46 diferencias no nulas son positivas. Por ello, T^+ es la suma de todos los rangos posibles:

$$T^+_{exp} = \sum_{i=1}^{46} r(|D_i|) = \sum_{i=1}^{46} i = \frac{N(N+1)}{2} = \frac{46 \times (46+1)}{2} = 1081$$

Ahora, para la aproximación asintótica, sabemos que si $N > 15$, se puede aproximar por $N(\mu, \sigma^2)$. Este es nuestro caso, ya que $N = 46$. Calculamos la media y la varianza de T^+ bajo

H_0.

- Media: $\mu_T = \frac{N(N+1)}{4} = \frac{46(47)}{4} = 540.5$.
- Varianza: $\sigma_T^2 = \frac{N(N+1)(2N+1)}{24} = \frac{46(47)(93)}{24} = 8377.75$
- Desviación típica: $\sigma_T = 91.53$

Calculamos el p-valor. Queremos calcular la probabilidad de obtener un resultado como el nuestro (1081), asumiendo que la media debería ser 540.5. Para ello, tipificamos $T_{exp}^+ : Z = \frac{(T_{exp}^+ - 0.5) - \mu_T}{\sigma_T} = \frac{(1081-0.5)-540.5}{91.53} \approx 5.90$.

Por lo tanto, el p-valor asociado a $Z = 5.90$ es $p \approx 1.8 \times 10^{-9}$. Dado que $p < 0.05$, rechazamos la hipótesis nula H_0. Esto nos demuestra que la mejora del NSGA-II en el índice de Jaccard no es cuestión de azar, sino que es estadísticamente significativa.

3.4. Discusión y observaciones finales

El análisis de los tres experimentos descritos en este capítulo nos ha permitido ilustrar empíricamente las hipótesis de comportamiento de los algoritmos.

- Experimento 1: este experimento (instancias pequeñas, $k = 3$) sirvió para establecer una referencia óptima (dentro del subespacio) usando la búsqueda exhaustiva con profundidad limitada. El hallazgo clave fue que el NSGA-II convergió al mismo frente de Pareto en todas las instancias. Esto aporta una evidencia empírica que valida nuestra hipótesis de que el NSGA-II es capaz de converger al óptimo global en espacios de búsqueda acotados.
- Experimento 2: este experimento (20 instancias con $M_{Jaccard} = 1.0$ conocido) confirmó la superioridad de la exploración global del NSGA-II (70 % de tasa de éxito) frente a la exploración local del Greedy-MO (20 % de tasa de éxito). También demostró la rapidez del Greedy-MO, que en todos los casos encontró el óptimo local en pocos milisegundos.
- Experimento 3: este experimento (50 instancias aleatorias) fue el más revelador. El análisis estadístico no solo probó que la superioridad del NSGA-II en calidad (índice de Jaccard) es estadísticamente significativa ($p < 0.001$), sino que también captó el equilibrio multiobjetivo:
  - El Greedy-MO es instantáneo (media de 0.3 ms) pero produce soluciones de baja calidad (media $M_{Jaccard} = 0.722$) que fueron sistemáticamente dominadas, quedando atrapado en óptimos locales muy simples (mediana de 1 operación).
  - El NSGA-II demostró que para alcanzar esa calidad de Jaccard superior, tuvo que encontrar soluciones estructuralmente más complejas (mediana de 8 operaciones).

  Además, el frente de Pareto obtenido por el NSGA-II es mucho más diverso que el del Greedy-MO.

En conclusión, el análisis de los experimentos ha evidenciado la intratabilidad del problema en casos complejos, siendo la búsqueda exhaustiva con profundidad limitada inviable.

La heurística Greedy-MO es rápida pero subóptima, quedando limitada por el espacio de búsqueda que explora. Finalmente, la metaheurística NSGA-II demuestra ser el enfoque más robusto, capaz de explorar eficazmente espacios de búsqueda más complejos y encontrar un buen equilibrio entre los objetivos en conflicto.

4. Conclusiones y vías futuras

En este capítulo final presentamos las conclusiones generales del trabajo, resumiendo los logros que hemos obtenido tanto en la definición del problema (Capítulo 1), como en la formulación y estudio matemático (Capítulo 2), y en la implementación algorítmica y experimental (Capítulo 3). Finalmente, expondremos las posibles vías de desarrollo posterior, tanto experimentales como teóricas, que surgen a partir del trabajo realizado.

4.1. Conclusiones generales

En este Trabajo de Fin de Grado hemos seguido una narrativa lógica que abarca desde la fundamentación matemática de un problema de aproximación de conjuntos hasta su implementación práctica y validación experimental. La conclusión principal no es un único hallazgo, sino la propia conexión entre la teoría, la complejidad y la algorítmica.

A continuación, dividimos las conclusiones en los dos grandes bloques del trabajo.

Conclusiones del bloque teórico-matemático

En el trabajo, hemos construido una caracterización formal y completa del problema de aproximación de conjuntos objeto de estudio, identificando sus componentes fundamentales y las estructuras algebraicas que lo sustentan. El estudio de anillos, semianillos, σ-álgebra, álgebra de Boole, retículos y relaciones entre los subconjuntos de U, ha proporcionado un marco sólido para entender cómo se pueden combinar y aproximar conjuntos.

Un resultado significativo ha sido la identificación de conexiones entre nuestro problema y problemas clásicos de la literatura, como Set Cover o Exact Cover. Esta relación ha permitido demostrar que el problema general es NP-Completo, estableciendo así sus límites computacionales y justificando la necesidad de recurrir a algoritmos aproximados y técnicas heurísticas. Esta parte teórica ha sido clave, pues conecta el análisis matemático con la complejidad computacional y la elección posterior de los algoritmos.

Finalmente, hemos formalizado el problema como un caso particular de problema multiobjetivo, analizando el papel y la naturaleza de diferentes medidas (de asociación, direccionales, etc.) que podrían ser utilizadas.

Conclusiones del bloque práctico-computacional

En este bloque, tomamos la decisión de abordar el problema en su versión de optimización multiobjetivo, asumiendo el desafío añadido que esto conlleva. Seleccionamos tres algoritmos

para abarcar distintas ramas de la algorítmica: una búsqueda exhaustiva de profundidad limitada, para servir como referencia; una heurística Greedy-MO, que explora solo la región secuencial del espacio de búsqueda; y una metaheurística como lo es NSGA-II, que explora estocásticamente el mismo espacio que la búsqueda exhaustiva limitada.

A nivel de implementación, el resultado ha sido un laboratorio experimental que se ha mostrado robusto en los experimentos y pruebas realizados. Hemos construido una arquitectura modular con experimentos automatizados y configurables, que ha demostrado ser estable en los experimentos realizados y extensible como hemos indicado en la Subsección 3.3.1 (por ejemplo, para añadir nuevas métricas, o algoritmos).

Finalmente, en la fase de experimentación, los resultados obtenidos están en sintonía con las hipótesis: el algoritmo Greedy-MO fue el más rápido pero a menudo quedó atrapado en óptimos locales; el NSGA-II demostró ser significativamente más robusto, encontrando frentes de Pareto que dominaban al Greedy-MO en índice de Jaccard y en diversidad; y la búsqueda exhaustiva no ha sido viable en escenarios mínimamente complejos.

En resumen, este Trabajo de Fin de Grado ha conectado la fundamentación matemática con la validación empírica, entregando distintas soluciones validadas para su aproximación, todo ello utilizando un entorno de software reproducible y extensible.

Limitaciones del estudio

Este trabajo presenta dos limitaciones principales relativas a su alcance. En primer lugar, no hemos realizado un análisis sistemático sobre cómo el rendimiento de los algoritmos escala al variar las características del problema (tamaño de U, de F, de G, de los F_i), por lo que la generalización de los resultados a instancias con propiedades distintas queda como una vía de trabajo futuro. En segundo lugar, nos hemos enfocado en comparar las estrategias algorítmicas (Greedy-MO vs. NSGA-II) y no en escoger de manera exhaustiva sus hiperparámetros. La calibración del NSGA-II la realizamos de forma empírica y, aunque es robusta, no se puede garantizar su optimalidad, la cual podría explorarse con otras técnicas en un trabajo posterior.

Valoración personal y competencias adquiridas

A nivel personal, este Trabajo de Fin de Grado ha sido un desafío, no solo en el ámbito puramente académico. Por primera vez me he enfrentado a un proyecto de esta envergadura, lo que ha requerido desarrollar competencias clave como la búsqueda autodidacta de información, la evaluación de fuentes fiables, y la habilidad para estructurar y conectar ideas de forma coherente.

Una de las dificultades principales fue la abstracción, es decir, aprender a explicar los conceptos sin asumir conocimientos previos, y a separar limpiamente los detalles de implementación, de la fundamentación teórica y la validación experimental.

Académicamente, este trabajo ha sido un puente entre varias asignaturas cursadas a lo largo del doble grado en ingeniería informática y matemáticas. Para el análisis de la intra-

tabilidad del problema, hemos utilizado "Modelos Avanzados de Computación" (para la NP-Completitud) y "Algorítmica" (para el análisis de complejidad). La solución heurística ha requerido aplicar los conocimientos de "Metaheurísticas" (para el NSGA-II) y "Estructuras de Datos" (para una implementación eficiente), y todo ello sobre las bases de "Álgebra" (teoría de conjuntos y retículos). Además, hemos utilizado conceptos de "Inferencia Estadística" en el test de Wilcoxon sobre los resultados del Experimento 3.

En el aspecto práctico, el reto más significativo fue la implementación y calibración del algoritmo genético. Más allá de la propia codificación, el proceso de ajustar los operadores, las probabilidades y los parámetros para que el algoritmo convergiera de forma robusta supuso una gran dificultad que requirió paciencia y un enfoque metódico.

4.2. Perspectivas futuras y posibles avances

El análisis de complejidad y los resultados experimentales de este trabajo abren varias líneas de investigación futuras, tanto para estudiar otros aspectos de los algoritmos como para profundizar en el análisis del problema.

4.2.1. Extensiones directas y mejoras algorítmicas

Existen varias mejoras directas que continuarían la línea de este trabajo:

- **Análisis de rendimiento:** una vía de gran interés sería diseñar un experimento adicional entre el Greedy-MO y el NSGA-II. Se ejecutaría primero el Greedy-MO para obtener su frente de Pareto PF_{Greedy}. Luego, se ejecutaría el NSGA-II midiendo el tiempo exacto que tarda en generar un frente $PF_{NSGA-II}$ que domine completamente a PF_{Greedy}. Esto proporcionaría una métrica clara del tiempo necesario hasta la superación, al contrario que ahora, que el NSGA-II completa el tiempo límite establecido.
- **Análisis estadístico multiobjetivo avanzado:** otro análisis que merecería la pena hacer en el experimento 3 es calcular el indicador de hipervolumen (HV) para cada frente, que no hemos implementado en este trabajo debido a su alta complejidad algorítmica. Esta métrica mide tanto convergencia como diversidad y permitiría una comparación estadística más robusta del rendimiento global de los algoritmos.
- **Extensión del espacio de objetivos:** actualmente el problema se ha definido con tres objetivos. Una extensión natural sería incluir nuevas métricas, como la interpretabilidad de la expresión (por ejemplo, penalizando la anidación excesiva) o el coste computacional de evaluarla, como un cuarto objetivo a minimizar.
- **Inclusión de nuevas operaciones en las expresiones:** el trabajo actual se limita a los operadores binarios clásicos $\{\cup, \cap, \setminus\}$. Una vía futura de gran interés teórico sería estudiar el impacto de añadir nuevos operadores, como la diferencia simétrica ($F_i \oplus F_j$) o un operador de implicación lógica ($F_i \to F_j$). Aunque estos operadores pueden expresarse como combinaciones de los que ya utilizamos ($\cup, \cap, \setminus$), y por tanto no amplían el espacio de conjuntos alcanzables, su inclusión permitiría reducir la complejidad estructural de las expresiones resultantes en algunos casos y poder explorar potencialmente más conjuntos manteniendo la misma restricción de tamaño en las expresiones.

A. Anexo: Planificación y presupuesto

En este apéndice proporcionamos documentación detallada de la gestión y la viabilidad económica del proyecto. Está dividido en dos partes principales: el cronograma de desarrollo, donde podemos visualizar la planificación temporal inicial de las tareas y la distribución final del esfuerzo a lo largo de los 7 meses de trabajo; y el presupuesto, que recoge la estimación de costes y la justificación de los recursos materiales.

A.1. Planificación

Inicialmente, estimamos una carga de trabajo de 500 horas para completar este Trabajo de Fin de Grado. La planificación original contemplaba distribuir este esfuerzo en un plazo de 4 meses, con la siguiente estructura:

- Mes 1: fundamentos y definición del problema y sus elementos. Estudio de la complejidad teórica, medidas y estructuras algebraicas.
- Meses 2 y 3: desarrollo de las tres aproximaciones algorítmicas escogidas. Diseño e implementación del entorno de experimentación.
- Mes 4: ejecución de experimentos, recogida e interpretación de resultados y conclusiones.

Sin embargo, durante el desarrollo del proyecto, debido a la complejidad del mismo y al tiempo disponible para su realización, se hizo necesario replantear la distribución de dichas 500 horas de trabajo.

El cronograma final del proyecto, ilustrado en la Figura A.1, refleja la distribución real del esfuerzo dividido en las etapas principales que abarcan los 7 meses de trabajo.

Como podemos ver, comenzamos definiendo el problema y sus aplicaciones, así como estudiando las estructuras algebraicas asociadas. En los meses 2 y 3, nos centramos en el estudio de medidas y de la complejidad teórica, a la vez que empezamos a desarrollar el algoritmo Greedy por la parte informática.

En los meses 4 y 5, desarrollamos los otros dos algoritmos (búsqueda exhaustiva de profundidad limitada y NSGA-II), realizando también pruebas preliminares sobre los tres métodos.

En los últimos dos meses, llevamos a cabo la ejecución de todos los experimentos, ajustando posibles inconvenientes. Finalmente, nos centramos en el análisis de sus resultados, el cual fue clave para la redacción de las secciones finales de la memoria y la corrección de detalles en las secciones iniciales.

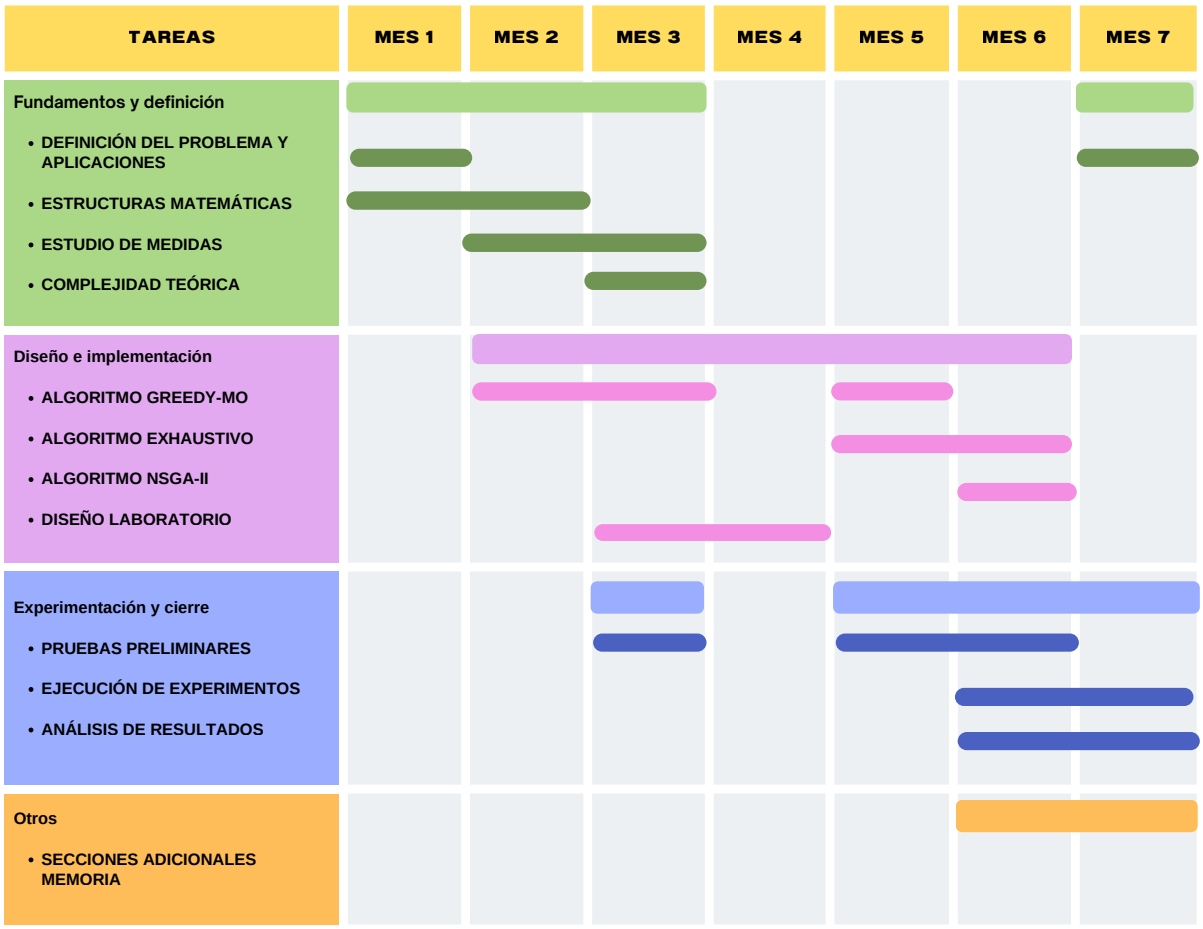

Figura A.1.: Diagrama de Gantt del proyecto. Se muestra la distribución temporal de las tareas a lo largo de los 7 meses de desarrollo.

A.2. Presupuesto

En la Tabla A.1 se muestra el desglose detallado.

Tabla A.1.: Presupuesto desglosado del proyecto.

Concepto	**Coste total**
Recursos Humanos	
Ingeniero Junior	8.000,00 €
Hardware	
Amortización Portátil	150,00 €
Software y Servicios	
Licencia Overleaf (2 meses)	38,00 €
Compilador G++	0,00 €
Python y Librerías	0,00 €
Repositorio GitHub	0,00 €
TOTAL	**8.188,00 €**

Hemos calculado el presupuesto del proyecto estimando los costes de recursos humanos y materiales necesarios para su ejecución.

Para el cálculo del coste de personal, hemos tomado como referencia el salario medio de un Ingeniero Informático Junior en la ciudad de Granada (aproximadamente 22.000€ brutos anuales[1]). La parte de recursos humanos refleja el valor de las 500 horas invertidas en la investigación, implementación y experimentación.

En cuanto a los recursos materiales, contemplamos la amortización del equipo informático personal utilizado (calculada proporcionalmente a los 7 meses de uso sobre una vida útil de 5 años[2]). Además, incluimos el coste directo de la licencia *Standard* de la plataforma Overleaf, necesaria para la gestión y compilación de la memoria debido a su extensión. Cabe destacar que el uso de tecnologías de código abierto y gratuitas (`G++`, `Python`) nos ha permitido reducir los costes de licencias de software.

[1] Estimación extraida del portal de empleo Glassdoor en noviembre del 2025.
[2] Vida útil estimada según las especificaciones oficiales del fabricante (ASUS).

B. Anexo: Datos completos del experimento 3

A continuación se presentan los datos pareados del Experimento 3, utilizados para el Test de Wilcoxon en la sección 3.3.3.3. La Tabla B.1 muestra el mejor índice de Jaccard obtenido por cada algoritmo en cada una de las 46 semillas en las que la diferencia (D_i) entre ambos era no nula, así como el valor de D_i.

Tabla B.1.: Datos pareados de Mejor Jaccard (Experimento 3, 50 Instancias).

Semilla	Jaccard (Greedy)	Jaccard (NSGA-II)	Diferencia (NSGA-II - Greedy)
2001	0.7568	**0.8000**	+0.0432
2003	0.7444	**0.8354**	+0.0910
2004	0.7711	**0.8795**	+0.1084
2006	0.4250	**0.5424**	+0.1174
2007	0.9767	**0.9884**	+0.0117
2008	0.4756	**0.5938**	+0.1182
2009	0.4667	**0.6522**	+0.1855
2011	0.9375	**0.9917**	+0.0542
2012	0.2857	**0.3846**	+0.0989
2013	0.9263	**0.9474**	+0.0211
2014	0.5536	**0.6279**	+0.0743
2015	0.7578	**0.9600**	+0.2022
2016	0.9063	**1.0000**	+0.0937
2017	0.1935	**0.2200**	+0.0265
2018	0.8906	**1.0000**	+0.1094
2019	0.9141	**1.0000**	+0.0859
2020	0.9922	**1.0000**	+0.0078
2021	0.9878	**1.0000**	+0.0122
2022	0.1667	**0.1837**	+0.0170
2023	0.7344	**0.8364**	+0.1020
2024	0.9747	**1.0000**	+0.0253
2025	0.5385	**0.6667**	+0.1282
2026	0.4688	**0.5714**	+0.1026
2028	0.3659	**0.4688**	+0.1029
2029	0.4634	**0.5610**	+0.0976
2030	0.8077	**0.9231**	+0.1154
2031	0.7222	**0.7910**	+0.0688
2032	0.9609	**1.0000**	+0.0391
2033	0.7051	**0.8154**	+0.1103
2034	0.3824	**0.5172**	+0.1348
2035	0.3235	**0.4762**	+0.1527

Continúa en la siguiente página...

Tabla B.1 – continuación de la página anterior

Semilla	Jaccard (Greedy)	Jaccard (NSGA-II)	Diferencia (NSGA-II - Greedy)
2036	0.8594	**0.9910**	+0.1316
2037	0.9518	**0.9875**	+0.0357
2038	0.9375	**0.9756**	+0.0381
2039	0.9615	**1.0000**	+0.0385
2040	0.9922	**1.0000**	+0.0078
2041	0.9453	**0.9758**	+0.0305
2042	0.7969	**0.9808**	+0.1839
2043	0.2000	**0.2593**	+0.0593
2044	0.6364	**0.7179**	+0.0815
2045	0.8594	**1.0000**	+0.1406
2046	0.9063	**0.9915**	+0.0852
2047	0.7886	**0.9592**	+0.1706
2048	0.8276	**0.9286**	+0.1010
2049	0.5333	**0.6897**	+0.1564
2050	0.7344	**0.8393**	+0.1049

C. Anexo: Repositorio de Código y Guía de Reproducibilidad

C.1. Repositorio de código

El código fuente completo de este trabajo, incluyendo los scripts de experimentación, el archivo `config.yaml` y los archivos .cpp/.hpp, está disponible públicamente en el siguiente repositorio:

https://github.com/lauralsoraluce/TFG

El código está publicado bajo licencia MIT, que permite el uso, copia, modificación y distribución del código, manteniendo únicamente el aviso de *copyright* y la propia licencia.

C.2. Guía de compilación y reproducibilidad

Esta guía detalla los pasos para compilar y ejecutar los experimentos descritos en esta memoria.

Requisitos del Entorno

- **Compilador:** g++ con soporte para C++17.
- **Sistema Operativo:** desarrollado y probado en Ubuntu 20.04.
- **Python:** versión 3.8 o superior.
- **Librerías de Python:** PyYAML (se puede instalar con pip install PyYAML).

Compilación

La compilación es automática y la gestionan los scripts de Python. Cada vez que se lanza un experimento, el script invoca a g++ para compilar el ejecutable `build/main`, asegurando que se utiliza la última versión del código. Si se quiere compilar manualmente, el comando (ejecutado desde la raíz `tfg-exp/`) es:

```
g++ -std=c++17 -O3 -I tfgcore/include/ tfgcore/src/*.cpp -o build/main
```

Código C.1: Comando de compilación manual

Ejecución de experimentos

El repositorio contiene dos scripts principales en la carpeta `scripts/` para lanzar las tandas de experimentos. El ejecutable de C++ (build/main) acepta el parámetro `- -algo` para determinar qué algoritmos ejecutar.

- **`experimentos.py`:** lanza el experimento 1 (instancias pequeñas, $k = 3$) y/o el experimento 3 (instancias grandes, $k = 10$, 50 semillas). Al ejecutarlo, ofrece un menú interactivo para elegir qué experimento ejecutar.
- **`exp_genetico.py`:** lanza el experimento 2 ($M_{Jaccard} = 1.0$ conocido, $k = 10$, 20 semillas).

Para lanzar, por ejemplo, los experimentos 1 y 3 (que es lo que hace el experimentos.py):

```
cd /ruta/al/proyecto/tfg-exp/scripts python3 experimentos.py
```

Código C.2: Ejecución del script experimentos.py

Estructura de resultados

Los scripts crean las siguientes carpetas dentro de `results/`:

- `results/small/`: contiene los resultados del experimento 1, así como los *datasets* utilizados en él (los ficheros tienen los nombres: `..._resultados.txt` y `..._dataset.txt`, según el momento en el que se crearon (timestamp)).
- `results/exp_genetico/`: contiene los resultados del experimento 3, así como los *datasets* utilizados en él (de nuevo, los ficheros tienen los nombres: `..._resultados.txt` y `..._dataset.txt`, según el momento en el que se crearon (timestamp)).
- `results/batch/`: contiene los resultados del experimento 2 así como los *datasets* utilizados en él. Este script crea un subdirectorio con timestamp (ej. `2025-11-09_183000/`) que contiene los archivos `resultados.txt` y `dataset.txt`.

En los archivos de *datasets*, además de las semillas generadoras, mantenemos los valores concretos de las instancias utilizadas. Esto lo hacemos para garantizar la reproducibilidad incluso ante posibles variaciones en las implementaciones de las bibliotecas pseudoaleatorias.

C.3. Archivo de configuración experimental (`config.yaml`)

Todos los experimentos de este trabajo se han ejecutado de forma automatizada. Para garantizar la reproducibilidad, todos los parámetros del problema, la generación de instancias, las semillas y los límites de tiempo (el único parámetro del genético que ajustamos según el experimento) se guardan en un único archivo de configuración `config.yaml`. A continuación, se muestra el contenido de dicho archivo, seguido de una explicación de sus componentes.

```yaml
U_size: 128   # modificable

small_config:
  G_size_min: 10
  F_n_min: 5
  F_n_max: 20
  Fi_size_min: 5
  Fi_size_max: 50
  k: 3
  seeds: [1002, 1003, 1004, 1005]
  instances: 4
  timeouts:
    ga_default_sec: 150

genetico_config:
  G_size_min: 10
  k: 10
  F_n_min: 5
  F_n_max: 80
  Fi_size_min: 5
  Fi_size_max: 100
  seeds: [1000, 1010, 1020, 1030, 1040, 1050, 1060, 1070, 1080, 1090, 2000, 2010,
      ↪ 2020, 2030, 2040, 2050, 2060, 2070, 2080, 2090]
  timeouts:
    ga_default_sec: 900

batch_config:
  G_size_min: 10
  F_n_min: 5
  F_n_max: 80
  Fi_size_min: 5
  Fi_size_max: 100
  k: 10
  seed_start: 2001
  instances: 50
  timeouts:
    ga_default_sec: 900

# --- Rutas ---
paths:
  results_small: "results/small"
  results_batch: "results/batch"
  results_genetico: "results/exp_genetico"
```

Código C.3: Archivo de configuración `config.yaml`

Descripción de los parámetros

- U_size: parámetro global que fija el tamaño del universo ($|U|$) en 128 para todos los experimentos, coincidiendo con la implementación de `std::bitset<128>`.
- `small_config`: este bloque define la configuración utilizada en el experimento 1 (instancias pequeñas).
  - `k: 3`: se establece una profundidad máxima de $k = 3$. Este valor es crucial para permitir la ejecución de la Búsqueda Exhaustiva y obtener una referencia óptima.
  - `F_n_max: 20`: el número de conjuntos base (m) se limita a 20.

- `seeds`: se especifican las 4 semillas exactas $(1002 - 1005)$ para este experimento de validación.
- `ga_default_sec: 150`: límite de tiempo (150 s) para el NSGA-II en este escenario.

- `genetico_config`: este bloque define la configuración para el experimento 2 (validación de alcance de $Jaccard = 1.0$).
  - `k: 10`: se incrementa la profundidad a $k = 10$.
  - `F_n_max: 80`: se establece el número máximo de conjuntos base en 80.
  - `seeds`: se define la lista explícita de las 20 semillas utilizadas para generar las instancias con óptimo conocido.
  - `ga_default_sec: 900`: límite de tiempo para el NSGA-II.
- `batch_config`: este bloque define la configuración para el experimento 3 (instancias grandes).
  - `k: 10`: se mantiene la profundidad en $k = 10$.
  - `F_n_max: 80`: se establece el número máximo de conjuntos base en 80.
  - `seed_start: 2001` e `instances: 50`: define que se deben generar 50 instancias, utilizando semillas correlativas a partir de la 2001 $(2001 - 2050)$.
  - `ga_default_sec: 900`: Se mantiene el límite de tiempo de 900 s.
- `paths`: define las rutas donde el sistema de experimentación guarda los archivos de resultados para cada uno de los tres experimentos.

Bibliografía

[AB09] Sanjeev Arora y Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, USA, 1st edición, 2009. ISBN: 0521424267. https://dl.acm.org/doi/10.5555/1540612.

[Aig07] Martin Aigner. *A Course in Enumeration*, volumen 238 de *Graduate Texts in Mathematics*. Springer, Berlin, Heidelberg, 2007. ISBN: 978-3-540-39032-4. DOI: 10.1007/978-3-540-39035-0. https://link.springer.com/book/10.1007/978-3-540-39035-0.

[BB01] P. Baldi y S. Brunak. *Bioinformatics, second edition: The Machine Learning Approach*. Adaptive Computation and Machine Learning series. MIT Press, 2001. ISBN: 9780262255707. https://ieeexplore.ieee.org/servlet/opac?bknumber=6267217.

[BL21] Eric J. Beh y Rosaria Lombardo. *An introduction to correspondence analysis*. Wiley Series in Probability and Statistics. John Wiley and Sons, Incorporated, Hoboken, New Jersey, 2021. ISBN: 9781119041948. https://onlinelibrary.wiley.com/doi/book/10.1002/9781119044482.

[Bou94] Nicolas Bourbaki. *Elements of the history of mathematics*. Berlin ; New York : Springer, 1994. ISBN: 3540647678. https://archive.org/details/elementsofhistor0000bour.

[Bou04] Nicolas Bourbaki. *Theory of Sets*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. DOI: 10.1007/978-3-642-59309-3. ISBN: 978-3-540-22525-6.

[Bre13] Lucas Bremond. *An Optimal Tour Generation Strategy for a Multiple Rendezvous Mission to the Trojan Asteroids*. PhD thesis, Universite Paul Sabatier - Toulouse III, August 2013. https://doi.org/10.13140/RG.2.1.2904.0246.

[Cam94] Peter J. Cameron. *Combinatorics*. Cambridge University Press, Cambridge, 1994. ISBN: 9780521457613. DOI: 10.1017/CBO9780511803888.

[CdlF22] Carlos Carazo de la Fuente. Problemas de programación lineal multi-objetivo, June 2022. Trabajo de Fin de Grado. Director: Pedro Mariano Mateo Collazos. https://zaguan.unizar.es/record/125505/files/TAZ-TFG-2022-2789.pdf.

[CHK23] Peter Christen, David J. Hand, y Nishadi Kirielle. A Review of the F-Measure: Its History, Properties, Criticism, and Alternatives. *ACM Comput. Surv.*, 56(3), October 2023. DOI: 10.1145/3606367.

[Dev93] Keith Devlin. *The Joy of Sets*. Undergraduate Texts in Mathematics. Springer New York, New York, NY, 1993. ISBN: 978-1-4612-6941-0 978-1-4612-0903-4. DOI: 10.1007/978-1-4612-0903-4. http://link.springer.com/10.1007/978-1-4612-0903-4.

[DP02] B. A. Davey y H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, 2 edición, 2002. ISBN: 9780511809088. DOI: 10.1017/CBO9780511809088.

[DPAM02] K. Deb, A. Pratap, S. Agarwal, y T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002. DOI: 10.1109/4235.996017.

[ED18] Michael T. M. Emmerich y André H. Deutz. A tutorial on multiobjective optimization: fundamentals and evolutionary methods. *Natural Computing*, 17(3):585–609, September 2018. DOI: 10.1007/s11047-018-9685-y. http://link.springer.com/10.1007/s11047-018-9685-y.

[Ehr05] Matthias Ehrgott. *Multicriteria Optimization*. Springer, Berlin, Heidelberg, 2 edición, 2005. ISBN: 978-3-540-21398-7 978-3-540-27659-3. DOI: 10.1007/3-540-27659-9.

[ES15] A.E. Eiben y J.E. Smith. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. DOI: 10.1007/978-3-662-44874-8.

[FD07] José Ferreirós Domínguez. *Labyrinth of thought: a history of set theory and its role in modern mathematics José Ferreirós*. Birkäuser, Basel, 2nd revised ed edición, 2007. ISBN: 978-3-7643-8349-7. https://link.springer.com/book/10.1007/978-3-7643-8350-3.

[FH03] L. Fortnow y S. Homer. A Short History of Computational Complexity. *Bulletin of the European Association for Theoretical Computer Science*, 80, June 2003. https://lance.fortnow.com/papers/files/history.pdf.

[FS09] Philippe Flajolet y Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, Cambridge, 2009.

[GC20] Javier González Castillo. Números de Catalan y Aplicaciones, noviembre 2020. Trabajo de Fin de Grado. Director: Pedro J. Miana Sanz. https://zaguan.unizar.es/record/97683/files/TAZ-TFG-2020-5191.pdf.

[GJ90] Michael R. Garey y David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990. ISBN: 0-7167-1045-5. https://dl.acm.org/doi/book/10.5555/574848.

[GK79] Leo A. Goodman y Herny Kruskal. *Measures of Association for Cross Classifications*. Number 1 en Springer Series in Statistics. Springer New York, New York, NY, 1979. ISBN: 978-1-4612-9997-4 978-1-4612-9995-0. DOI 10.1007/978-1-4612-9995-0. https://link.springer.com/book/10.1007/978-1-4612-9995-0.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edición, 1989. ISBN: 0-201-15767-5. https://dl.acm.org/doi/10.5555/534133.

[Gri99] Ralph P. Grimaldi. *Discrete and Combinatorial Mathematics: An Applied Introduction*. Addison-Wesley Longman, Reading, MA, 4th edition edición, 1999. https://archive.org/details/discretecombinat0000grim.

[Hal74a] Paul R. Halmos. *Lectures on Boolean Algebras*. Undergraduate Texts in Mathematics. Springer, New York, NY, 1 edición, 1974. Originally published by Van Nostrand (1968). ISBN: 978-0-387-90094-0. DOI: 10.1007/978-1-4612-9855-7.

[Hal74b] Paul R. Halmos. *Measure Theory*. Graduate Texts in Mathematics. Springer-Verlag, Berlin, first edición, 1974. DOI: 10.1007/978-1-4684-9440-2. ISBN: 978-0-387-90088-9.

[HM15] Mohammad Hossin y Sulaiman M.N. A Review on Evaluation Metrics for Data Classification Evaluations. *International Journal of Data Mining & Knowledge Management Process*, 5:01–11, 03 2015. DOI: 10.5121/ijdkp.2015.5201.

[Hol92] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press, Cambridge, MA, 1992. ISBN: 9780262275552. DOI: 10.7551/mitpress/1090.001.0001.

[HS04] Holger Hoos y Thomas Stützle. *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004. ISBN: 1558608729. DOI: 10.5555/983505.

[HST+22] Steven A. Hicks, Inga Strümke, Vajira Thambawita, Mohamed Hammou, Michael A. Riegler, Pål Halvorsen, y Sravanthi Parasa. On evaluation metrics for medical applications of artificial intelligence. *Scientific Reports*, 12(1):5979, 2022. DOI: 10.1038/s41598-022-09954-8.

[HWC14] Myles Hollander, Douglas A. Wolfe, y Eric Chicken. *Nonparametric statistical methods* . Wiley series in probability and statistics. John Wiley & Sons, New York, 3rd ed. edición, 2014. ISBN: 9780470387375. DOI: 10.1002/9781119196037.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. En R. E. Miller, J. W. Thatcher, y J. D. Bohlinger, editores, *Complexity of Computer Computations*, páginas 85–103. Plenum, New York, 1972. ISBN: 978-1-4684-2003-6. DOI: 10.1007/978-1-4684-2001-2_9. https://link.springer.com/book/10.1007/978-1-4684-2001-2.

[KCS06] Abdullah Konak, David W. Coit, y Alice E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety*, 91(9):992–1007, 2006. Special Issue - Genetic Algorithms and Reliability.

[Ken75] JW Kendall. Hard and soft constraints in linear programming. *Omega*, 3(6):709–715, 1975. https://www.sciencedirect.com/science/article/pii/0305048375900730.

[KT05] Jon Kleinberg y Éva Tardos. *Algorithm Design*. Pearson/Addison Wesley, Boston, 2005. ISBN: 0-321-29535-8.

[LFSJ22] Luc Libralesso, Pablo Andres Focke, Aurélien Secardin, y Vincent Jost. Iterative beam search algorithms for the permutation flowshop. *European Journal of Operational Research*, 301(1):217–234, 2022. ISBN: 0377-2217. https://www.sciencedirect.com/science/article/pii/S0377221721008602.

[Lip98] Seymour Lipschutz. *Set Theory and Related Topics*. Schaum's Outline. McGraw-Hill, 2nd edition edición, 1998. ISBN: 0070381593. https://www.mheducation.com/highered/mhp/product/schaum-s-outline-set-theory-related-topics.html?viewOption=student.

[Men70] Elliott Mendelson. *Schaum's Outline of Boolean Algebra and Switching Circuits*. Schaum's Outline Series. McGraw-Hill, New York, 1970. ISBN: 0070414602. https://www.mheducation.com/highered/mhp/product/schaum-s-outline-boolean-algebra-switching-circuits.html?viewOption=student#mh-ecomm-he-product-accordion-248675630-heading-mh-ecomm-accordion-list-e6a1dab3c4.

[MMW24] Nima Moradi, Fereshteh Mafakheri, y Chun Wang. Set Covering Routing Problems: A review and classification scheme. *Computers & Industrial Engineering*, 198:110730, 2024. DOI: 10.1016/j.cie.2024.110730.

[MPR18] Rafael Martí, Panos M. Pardalos, y Mauricio G. C. Resende, editores. *Handbook of Heuristics*. Springer International Publishing, Cham, 2018. DOI: 10.1007/978-3-319-07124-4. ISBN: 978-3-319-07123-7 978-3-319-07124-4. http://link.springer.com/10.1007/978-3-319-07124-4.

[MRS08] Christopher D. Manning, Prabhakar Raghavan, y Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, 2008. DOI: 10.1017/CBO9780511809071. ISBN: 9780521865715.

[RV96] Raimundo Real y Juan M. Vargas. The Probabilistic Basis of Jaccard's Index of Similarity. *Systematic Biology*, 45(3):380–385, 09 1996. DOI: 10.1093/sysbio/45.3.380.

[Sip96] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edición, 1996. ISBN: 053494728X. https://dl.acm.org/doi/10.5555/524279.

[Tin10] Kai Ming Ting. Error Rate. En Claude Sammut y Geoffrey I. Webb, editores, *Encyclopedia of Machine Learning*, páginas 331–331. Springer US, Boston, MA, 2010. DOI: 10.1007/978-0-387-30164-8_262. ISBN: 978-0-387-30164-8.

[Tur37] Alan Mathison Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(1):230–265, 1937. https://doi.org/10.1112/plms/s2-42.1.230.

[Wil04] Stephen Willard. *General Topology*. Dover Publications, Mineola, NY, unabridged republication of the 1970 edition edición, 2004. ISBN: 9780486131788. https://store.doverpublications.com/products/9780486131788.

[Zac06] Richard Zach. Kurt Gödel and Computability Theory. En Arnold Beckmann, Ulrich Berger, Benedikt Löwe, y John V. Tucker, editores, *Logical Approaches to Computational Barriers*, páginas 575–583, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN: 978-3-540-35468-0. DOI: https://doi.org/10.1007/11780342_59.