UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Graduado en Ingeniería del Software

Implementación de Metaheurísticas Multi-objetivo en Julia asistida por LLMs

Implementation of Multi-objective Metaheuristics in Julia assisted by LLMs

Realizado por
Agustín Romero Ladrón de Guevara

Tutorizado por
Antonio Jesús Nebro Urbaneja

Departamento
Lenguajes y Ciencias de la Computación

MÁLAGA, septiembre de 2024

Resumen

El uso de metaheurísticas aplicado a la optimización multiobjetivo ha aumentado considerablemente en las últimas décadas. Los algoritmos basados en metaheurísticas permiten obtener soluciones cercanas a las óptimas, sin requerir para ello una cantidad de recursos desorbitada. En la Universidad de Málaga radica el proyecto de código libre jMetal, un marco de trabajo escrito en Java para la optimización de problemas multiobjetivo mediante el uso de metaheurísticas.

En este Trabajo de Fin de Grado se parte de dos bases, el ya mencionado jMetal y un proyecto similar, aunque a menor escala llamado MetaJul, el cual está escrito en Julia, un lenguaje emergente. Este segundo nace de la idea de explorar la adaptación de una arquitectura modular por componente a un lenguaje nuevo muy diferente al original, pero con un rendimiento muy prometedor.

El objetivo de este trabajo, sin embargo, tiene un enfoque distinto. La idea es explorar las capacidades de distintos modelos de inteligencia artificial para convertir código de un lenguaje de programación a otro, con la dificultad de la disparidad entre uno y otro, pues Julia no es un lenguaje orientado a objetos y carece de características imprescindibles en Java, como podría ser la herencia.

De esta manera, el trabajo consta de dos partes diferenciadas. La primera se centra en, dado el mismo contexto, comparar las implementaciones adaptadas de Java a Julia que proporcionan diferentes LLMs, para una colección de problemas con aplicaciones en el mundo real. La segunda se aleja un poco de esa comparación para poner el foco en cuán útiles son estos mismos LLMs a la hora de traducir un algoritmo complejo de un lenguaje a otro.

Palabras clave: metaheuristicas, optimización multi-objetivo, software de optimización, LLM, petición.

Abstract

The usage of metaheuristics applied to multi-objective optimization has considerably increased in the last couple of decades. Algorithms based on metaheuristics allow almost optimal solutions to be obtained without requiring immense amounts of resources. An open source project called jMetal is rooted in Universidad de Málaga. This is a framework written in Java for optimizing multi-objective problems by using metaheuristics.

This Final Year Dissertation consists of two baseline projects, the aforementioned jMetal and a similar one, although on a smaller scale named MetaJul, written in Julia, an emerging language. The second one originates from the idea of exploring the adaptation from a modular architecture to a new, very different from the original, language, that shows a lot of potential in terms of performance.

However, the goal of this dissertation has a different focus. The idea is exploring the capabilities of different artificial intelligence models to convert code from one language to another, with the complication that the disparity between both creates, as Julia is not an object oriented language and lacks fundamental traits from Java, such as inheritance.

As such, the dissertation consists on two differentiated parts. The first one focuses on, given the same context, comparing the implementations adapted from Java to Julia that different LLMs offer, for a collection of real-world application problems. The second one strays a little from that comparison and sets the focus on how useful are those same LLMs when it comes to translating a complex algorithm from one language to the other.

Keywords: metaheuristics, multi-objective optimization, optimization framework, LLM, prompt.

Índice

1 Introducción

1.1. Motivación

La optimización multiobjetivo con metaheurísticas es una disciplina que ha experimentado un gran auge desde los últimos 25 años. Las razones van desde el interés por optimizar problemas con objetivos contrapuestos aplicables al mundo real, hasta la aparición de algoritmos no exactos basados en metaheurísticas [3]. Estos algoritmos se caracterizan principalmente por ser más flexibles en cuanto a las soluciones encontradas, pues no necesariamente serán capaces de encontrar siempre las soluciones óptimas, pero sí próximas a ellas y de manera rápida.

En este contexto, comienza en la Universidad de Málaga el desarrollo de jMetal [7][9], un framework de optimización multiobjetivo con metaheurísticas programado en Java. Con el tiempo, jMetal se ha convertido en una de las herramientas más usadas en este campo.

Durante los casi 20 años que lleva en desarrollo, el proyecto se ha expandido a otros lenguajes de programación, aunque no en su totalidad. Tampoco siguen teniendo mantenimiento varios de ellos, estando en activo actualmente jMetal en Java, jMetalPy en Python [1], y el más reciente de todos, MetaJul en Julia [10]. Este trabajo de fin de grado se centra en MetaJul, y plantea llevar a cabo un estudio para implementar metaheurísticas multiobjetivo en Julia a partir de las implementaciones ya existentes en jMetal, asistido por inteligencia artificial generativa (LLMs - *Large Languages Models*). En concreto, la metaheurística en la que se centra este Trabajo Fin de Grado es MOEA/D [12], que es uno de los algoritmos multiobjetivo de referencia.

La elección de Julia como lenguaje para el proyecto no es arbitraria. Tradicionalmente, en el ámbito científico se elije el lenguaje de programación dependiendo de si se busca

sencillez o rendimiento. Julia trata de ofrecer la sencillez y flexibilidad de lenguajes como Python, sin sacrificar el rendimiento que ofrecen lenguajes como C. Además, presenta un desafío adicional a la hora de desarrollar aplicaciones, y es que Julia no es orientado a objetos, al no presentar clases ni herencia. Otra dificultad añadida de usar Julia es que al no ser un lenguaje tan extendido como Java o Python, la base de conocimiento sobre este lenguaje con la que se ha entrenado un LLM posiblemente sea bastante menor y que si hubiese elegido un lenguaje más popular.

De esta manera, la idea principal será utilizar una o varias herramientas de LLM para asistir en el proceso de adaptación del algoritmo y sus componentes y analizar hasta qué punto estas herramientas facilitan el proceso.

Se parte de lo ya implementado en el proyecto MetaJul, el cual ya cuenta con algunos componentes y metaheurísticas.

1.2. Objetivos

1. Analizar los principales componentes del proyecto jMetal y estudiar cómo se podrían implementar en MetaJul. Mientras que Java es un lenguaje orientado a objetos, Julia no lo es en el sentido de que carece de los conceptos de clases y herencia.

2. Abordar la implementación de los problemas multiobjetivo continuos de jMetal en Julia usando LLMs. Para problemas sencillos se espera que consigan dar una traducción directa con el contexto adecuado. Para problemas más complejos el interés es identificar las limitaciones y tratar de conseguir, al menos, una implementación funcional.

3. Centrarse en la metaheurística multiobjetivo MOEA/D. Ver la utilidad de los LLM a la hora de traducir los distintos componentes necesarios como operadores, funciones agregativas o vecindarios.

1.3. Estructura del documento

La memoria se divide en cinco capítulos bien diferenciados, los cuales se describen brevemente a continuación.

El capítulo primero se trata de una introducción al proyecto. Se detallan la motivación y los objetivos del mismo, así como las tecnologías empleadas y la metodología seguida.

El capítulo segundo describe los conceptos en los que se basa el trabajo, los proyectos utilizados y otras herramientas utilizadas.

El capítulo tercero expone los resultados de los LLMs al traducir el código de los problemas de lenguaje Java a lenguaje Julia.

El capítulo cuarto explica el proceso de traducir el algoritmo MOEA/D y sus componentes necesarios con la ayuda de los LLMs.

Por último, en el capítulo quinto, se exponen las conclusiones y posibles líneas de trabajo futuras que este Trabajo de Fin de Grado facilita.

1.4. Tecnologías usadas

A continuación se recopilan las tecnologías utilizadas:

- Lenguajes de programación
  - Java: se utiliza el proyecto jMetal como referencia.
  - Julia: se utiliza el proyecto MetaJul como punto de partida.
- Entornos de programación y editores de texto
  - Intellij IDEA para Java.
  - Visual Studio Code para Julia.
  - Overleaf como editor LaTeX.
- Git y GitHub para administrar repositorios.
- LLMs
  - ChatGPT4o-Turbo
  - Gemini 1.5 Pro
  - Claude 3.5 Sonnet

1.5. Metodología empleada

Para la elaboración de este Trabajo de Fin de Grado se ha seguido una metodología de Feature-Driven Development, centrada en la finalización de funcionalidades específicas. Esta metodología permite considerar cada problema o componente como una característica independiente.

2

Conceptos y Herramientas

2.1. El Lenguaje de Programación Julia

Tradicionalmente, la programación científica ha estado dividida entre dos tipos de lenguajes: por un lado, los lenguajes de alta productividad, como Python, MATLAB o R, que priorizan la facilidad de desarrollo y la legibilidad del código; y por otro, los lenguajes de alto rendimiento, como C, C++ o Fortran, que ofrecen una ejecución rápida y una mayor cercanía al hardware, pero con mayor complejidad y menor expresividad. Esta separación obliga a menudo a los científicos e ingenieros a escribir prototipos en un lenguaje y posteriormente re-implementarlos en otro para obtener el rendimiento necesario.

Julia surge precisamente para cerrar esta brecha entre productividad y rendimiento [2]. Es un lenguaje de programación de alto nivel, diseñado específicamente para el cómputo numérico y científico, que combina una sintaxis sencilla y expresiva con un motor de ejecución muy eficiente. Para lograr este equilibrio, Julia se basa en el compilador LLVM (*Low-Level Virtual Machine*), lo que le permite generar código nativo altamente optimizado desde el propio código fuente, sin necesidad de escribir extensiones en C o recurrir a otros lenguajes para acelerar partes críticas del programa.

Una de las principales virtudes de Julia es su modelo de tipado dinámico con inferencia de tipos, lo que permite escribir código sin declarar tipos explícitos, pero sin sacrificar la velocidad de ejecución. Además, Julia está diseñado con el concepto de *multiple dispatch* como paradigma central: los métodos se seleccionan en tiempo de ejecución según los tipos concretos de los argumentos, lo que permite escribir funciones genéricas y re-utilizables. También ofrece una interoperabilidad sencilla con otras plataformas científicas: puede

llamar a bibliotecas escritas en C, Python o R sin necesidad de wrappers complicados.

Finalmente, Julia cuenta con un gestor de paquetes integrado y una comunidad activa que ya ha desarrollado un ecosistema creciente de librerías para áreas como álgebra lineal, optimización, aprendizaje automático, simulaciones, procesamiento de señales o análisis de datos. Por todo ello, Julia se está consolidando como una alternativa moderna para quienes buscan combinar expresividad y rendimiento en sus aplicaciones científicas y técnicas.

2.2. Algoritmos Evolutivos

Los algoritmos evolutivos forman parte de un conjunto más amplio de estrategias conocidas como metaheurísticas, las cuales se utilizan para resolver problemas de optimización con uno o varios objetivos [3]. A diferencia de los métodos exactos, que garantizan encontrar la solución óptima de un problema, las metaheurísticas adoptan un enfoque aproximado y generalmente no determinista, lo que les permite encontrar soluciones suficientemente buenas en tiempos razonables, especialmente en escenarios donde el espacio de búsqueda es muy amplio o la evaluación de soluciones es computacionalmente costosa.

Dentro de las metaheurísticas, los algoritmos evolutivos son las técnicas más conocidas y usadas con gran diferencia. Estos algoritmos están inspirados en mecanismos propios de la evolución biológica, como la selección natural, la reproducción y la mutación. Su funcionamiento se basa en una población de soluciones candidatas, conocidas como individuos, cada uno de los cuales codifica una posible solución al problema mediante un cromosoma, que representa las variables de decisión del problema.

El proceso evolutivo se desarrolla a lo largo de múltiples generaciones. En cada iteración, se evalúa la calidad de las soluciones mediante una función de aptitud (*fitness*), que indica qué tan adecuada es una solución con respecto al objetivo de optimización. Posteriormente, se seleccionan los individuos más aptos para reproducirse, generando nuevas soluciones mediante operadores genéticos como el cruce y la mutación. Estas nuevas soluciones se incorporan a la población, reemplazando total o parcialmente a la anterior, y el ciclo continúa hasta que se alcanza un criterio de parada, como un número máximo de generaciones o una mejora mínima entre iteraciones. Gracias a este esquema iterativo y adaptativo, los algoritmos evolutivos pueden explorar el espacio de búsqueda de manera

eficiente en busca de soluciones cercanas al óptimo.

2.3. Optimización Multiobjetivo

Los problemas de optimización multiobjetivo son aquellos en los que se pretende optimizar simultáneamente varias funciones objetivo, las cuales suelen estar en conflicto entre sí. Esto significa que mejorar el valor de una de las funciones puede implicar necesariamente el deterioro del valor de otra. Esta característica hace que el concepto de solución óptima sea fundamentalmente distinto del que se maneja en la optimización mono-objetivo, donde típicamente se busca una única solución que maximice o minimice una función determinada.

En el contexto multiobjetivo, una solución se considera óptima en el sentido de Pareto si no existe otra solución alternativa que mejore al menos uno de los objetivos sin empeorar alguno de los demás. El conjunto de todas las soluciones no dominadas, es decir, las que no son superadas por ninguna otra según este criterio, constituye el conjunto óptimo de Pareto. Por su parte, la representación gráfica de los valores que alcanzan estas soluciones en el espacio de los objetivos se conoce como el frente de Pareto. Este frente ofrece una colección de soluciones de compromiso, todas igualmente válidas desde el punto de vista matemático, y permite que un experto seleccione la alternativa que mejor se ajuste a sus prioridades o restricciones prácticas.

En el caso de problemas reales es muy frecuente que el frente de Pareto no pueda calcularse exactamente debido a la complejidad computacional y a la alta dimensionalidad del espacio de búsqueda, por lo que la alternativa es recurrir a métodos aproximados como las metaheurísticas, entre las que destacan los algoritmos evolutivos multiobjetivo (por ejemplo, NSGA-II [5], MOEA/D [12] o SPEA2). Estos algoritmos están diseñados para encontrar una buena aproximación del frente de Pareto, equilibrando dos criterios clave [6][4]: la convergencia, que evalúa cómo de cerca están las soluciones encontradas del frente óptimo, y la diversidad, que se refiere a cómo de bien repartidas están las soluciones a lo largo del frente. Una buena distribución permite ofrecer al usuario final una mayor variedad de opciones con distintos niveles de compromiso entre objetivos.

2.4. El Algoritmo MOEA/D

El algoritmo MOEA/D es un algoritmo evolutivo multiobjetivo propuesto por Q. Zhang y J. Li en 2007 [12]. A diferencia de otros algoritmos evolutivos, en lugar de buscar directamente un conjunto de soluciones óptimas (conocido como frente de Pareto), MOEA/D descompone el problema multiobjetivo en múltiples subproblemas de optimización de un solo objetivo, que se resuelven simultáneamente. Cada subproblema representa una combinación ponderada de los objetivos originales, lo que permite repartir el esfuerzo de búsqueda a lo largo del espacio de soluciones posibles.

Para resolver estos subproblemas, MOEA/D utiliza una población de soluciones candidatas que evolucionan con el tiempo. Cada solución está asociada a uno de los subproblemas y se mejora mediante operadores evolutivos como la recombinación y la mutación, típicos de los algoritmos genéticos. A diferencia de otros enfoques evolutivos, MOEA/D aprovecha la idea de vecindad: las soluciones asociadas a subproblemas similares se ayudan entre sí durante el proceso de evolución, compartiendo información útil para mejorar el rendimiento de forma colaborativa.

Una de las ventajas clave de MOEA/D es su eficiencia al escalar a problemas con muchos objetivos, donde otros algoritmos tradicionales tienen dificultades. Además, su estructura modular permite adaptar fácilmente distintas estrategias de evolución, selección o representación, según las características específicas del problema. Gracias a su diseño basado en la descomposición, MOEA/D se ha convertido en una herramienta versátil y eficaz para abordar una amplia variedad de problemas reales en ingeniería, logística, finanzas, y otras áreas donde hay que tomar decisiones equilibrando múltiples criterios al mismo tiempo.

El siguiente pseudo-código describe el funcionamiento de un algoritmo MOEA/D genérico:

Algorithm 1 Esquema general de MOEA/D

Input: Número de subproblemas N, función de agregación, vecindario T, número máximo de iteraciones $MaxGen$
Output: Conjunto aproximado del frente de Pareto
Inicializar pesos $\{\lambda_1, \lambda_2, \ldots, \lambda_N\}$ para los subproblemas
Determinar vecindarios $B(i)$ para cada subproblema i
Inicializar población $\{x_1, x_2, \ldots, x_N\}$ aleatoriamente
Evaluar las soluciones iniciales
Inicializar el conjunto de referencia (por ejemplo, el punto ideal)
for $gen = 1$ to $MaxGen$ **do**
    for $i = 1$ to N **do**
        Seleccionar padres de $B(i)$
        Generar una nueva solución y mediante recombinación y mutación
        Evaluar y
        Actualizar el punto de referencia si es necesario
        for all $j \in B(i)$ **do**
            if la agregación de y mejora la de x_j **then**
                Reemplazar $x_j \leftarrow y$
            end if
        end for
    end for
end for
return Población final como aproximación del frente de Pareto

2.5. El Framework jMetal

jMetal es un framework de código abierto, desarrollado en Java, diseñado específicamente para la investigación y el desarrollo de algoritmos de optimización, especialmente en el contexto de la optimización multiobjetivo mediante metaheurísticas [7][9]. Su arquitectura modular y extensible permite a los usuarios implementar, probar y comparar diferentes algoritmos evolutivos, técnicas basadas en inteligencia de enjambre y métodos de descomposición como NSGA-II y MOEA/D. Desde su aparición, jMetal se ha consolidado como una herramienta ampliamente utilizada tanto en entornos académicos como industriales dentro del ámbito de la optimización multiobjetivo.

El diseño de jMetal se basa en la separación clara de los componentes fundamentales de un algoritmo de optimización: problemas, algoritmos, operadores (como cruce, mutación y selección) y soluciones. Esta organización permite al usuario experimentar con distintas configuraciones sin necesidad de reescribir grandes bloques de código. Por ejemplo, definir un nuevo problema de optimización es tan sencillo como crear una clase que especifique

el número de variables, objetivos y el método de evaluación de las soluciones.

Además, jMetal incluye una colección de problemas estándar para estudios comparativos, indicadores de rendimiento (como el hipervolumen o la distancia generacional) y soporte para ejecución paralela y distribuida.

2.6. El Paquete MetaJul

MetaJul es un proyecto de código abierto desarrollado en Julia, orientado a facilitar la implementación de metaheurísticas, especialmente algoritmos evolutivos y multiobjetivo [10]. Surgió con la intención de explorar cómo adaptar la arquitectura modular y componentizada de jMetal en Julia, aprovechando las capacidades de rendimiento que ofrece el lenguaje.

El proyecto MetaJul incluye implementaciones de diferentes tipos de codificaciones (reales, enteros, binario, permutaciones), problemas de optimización, operadores genéticos y algoritmos como NSGAII, SMPSO y algoritmos genéticos mono-objetivos clásicos. Además, incluye ejemplos en de uso en *notebooks* Jupyter y pruebas unitarias que muestran cómo configurar y ejecutar estos algoritmos.

2.7. LLMs (Modelos Masivos de Lenguaje)

Los Modelos Masivos de Lenguaje o, por sus siglas en inglés, LLMs, han demostrado recientemente excelentes capacidades en procesamiento de tareas expresadas en lenguaje natural. Su éxito ha sido tal, que ha promovido una gran cantidad de contribuciones a su investigación en los últimos años. [8]

La necesidad de modelos con capacidades generales surge de un crecimiento en la demanda a los sistemas para manejar tareas de lenguaje complejas, tales como la traducción, el resumen o las interacciones conversacionales. Nuevos descubrimientos en modelos de lenguaje, principalmente transformadores, capacidad de cómputo y disponibilidad de datos de entrenamiento a gran escala, han permitido la creación de LLMs con rendimientos cercanos al que podrían tener humanos en diversas tareas.

Es necesario destacar las desventajas de los LLMs actualmente, pues no son perfectos, ni mucho menos que eso. La capacidad para resolver estas tareas a nivel humano es cos-

toso. Los LLMs necesitan un entrenamiento lento, hardware extenso y caro de ejecutar. Estos costes han limitado su adopción, pero también ha dado pie a que se diseñen mejores arquitecturas, estrategias de entrenamiento, y una gran cantidad de otras posibles mejoras. Todavía quedan tareas, como el razonamiento o la planificación que pueden resultar sencillas para los humanos, que a los modelos actuales no se les da bien.

Por último, es importante mencionar las notorias alucinaciones. Se llama alucinaciones a las respuestas generadas que suenan plausibles, pero son incorrectas o no se alinean con el input proporcionado. Esto, junto a las preocupaciones por la privacidad de las personas por entrenamientos que comprenden datos personales, son dos de las mayores preocupaciones que tiene la sociedad actual respecto a estos modelos masivos de lenguaje, pues da lugar a que el público general tenga acceso a información incorrecta, presentada de manera convincente.

3

Implementación de Problemas Multiobjetivo

En esta sección se afronta la adaptación de problemas multiobjetivo continuos de Java a Julia mediante el uso de LLMs. Se ha elegido un conjunto de problemas denominado Aplicaciones del Mundo Real, los cuales serán referidos por las siglas en inglés RWA, *Real-World Applications* [11]. A continuación se describen los problemas tratados.

1. **RWA1 - Subasi2016** (5 variables, 2 objetivos): Estructuras de colmena aplicadas a disipadores.

2. **RWA2 - Liao2008** (5 variables, 3 objetivos): Optimización de estructuras para vehículos que deben resistir impactos.

3. **RWA3 - Ganesan2013** (3 variables, 3 objetivos): Producción de gases sintéticos.

4. **RWA4 - Padhi2016** (5 variables, 3 objetivos): Determinación de parámetros para cortadores de geometrías complejas en materiales industriales.

5. **RWA5 - Gao2020** (9 variables, 3 objetivos): Estudio del rendimiento térmico de sistemas de almacenamiento de calor latente con lecho empacado.

6. **RWA6 - Xu2020** (4 variables, 3 objetivos): Selección de parámetros para el proceso de corte de acero de ultra alta resistencia.

7. **RWA7 - Goel2007** (4 variables, 3 objetivos): Mejora del rendimiento y vida útil de inyectores monopuerto para cohetes de combustible líquido.

8. **RWA8 - Vaidyanathan2004** (4 variables, 4 objetivos): Mejora del rendimiento y vida útil de inyectores monopuerto considerando la temperatura a tres pulgadas de la superficie del inyector.

9. **RWA9 - Chen2015** (6 variables, 5 objetivos): Diseño de antenas de banda ultra ancha.

10. **RWA10 - Ahmad2017** (3 variables, 7 objetivos): Producción de tejidos hidrofóbicos.

La naturaleza de estos problemas permite facilitar la verificación de la corrección de las implementaciones proporcionadas si se tiene acceso a los valores de los objetivos. Por esta razón, es conveniente realizar primero una evaluación de todos los problemas en jMetal.

```
1  package org.jmetal_test;
2  import org.uma.jmetal.problem.doubleproblem.impl.AbstractDoubleProblem;
3  import org.uma.jmetal.problem.multiobjective.rwa.*;
4  import org.uma.jmetal.solution.doublesolution.DoubleSolution;
5
6  import java.util.Arrays;
7
8  public class rwa_testing {
9      public static void main(String[] args) {
10         AbstractDoubleProblem[] problems = {new Subasi2016(), new
               Liao2008(), new Ganesan2013(), new Padhi2016(), new Gao2020(),
               new Xu2020(), new Goel2007(), new Vaidyanathan2004(), new
               Chen2015(), new Ahmad2017()};
11
12         for (AbstractDoubleProblem problem : problems) {
13             print_objectives(problem);
14         }
15     }
16
17     private static void print_objectives(AbstractDoubleProblem problem) {
18         int variables = problem.numberOfVariables();
19
20         DoubleSolution solution1 = problem.createSolution();
21         DoubleSolution solution2 = problem.createSolution();
22         DoubleSolution solution3 = problem.createSolution();
23
24         for (int i = 0; i < variables; i++) {
25             double min = problem.variableBounds().get(i).getLowerBound();
26             double max = problem.variableBounds().get(i).getUpperBound();
27             double avg = (min + max) / 2;
28
29             solution1.variables().set(i, min);
30             solution2.variables().set(i, avg);
31             solution3.variables().set(i, max);
32         }
33
34         evaluate_rwa(problem, solution1);
35         evaluate_rwa(problem, solution2);
36         evaluate_rwa(problem, solution3);
37         System.out.println();
38     }
39
```

```
40      private static void evaluate_rwa(AbstractDoubleProblem problem,
            DoubleSolution solution) {
41          DoubleSolution evaluated_solution = problem.evaluate(solution);
42          System.out.println(problem.name() + solution.variables() + ": " +
                Arrays.toString(evaluated_solution.objectives()));
43      }
44  }
```

En este caso, se realizan tres ejecuciones: una asignando a las variables sus valores mínimos, otra asignando sus valores máximos, y una tercera asignando sus valores intermedios. Por ejemplo, si el límite inferior de un problema de dos variables es (1, 2) y el límite superior es (2, 4), esta tercera ejecución asignará los valores (1.5, 3).

Se evaluarán tres peticiones con diferentes contextos. Cada petición se realizará a ChatGPT 4-turbo, Gemini 1.5 Pro y Claude 3.5 Sonnet. Debido a la gran cantidad de peticiones que fueron realizadas, estas se pueden encontrar en el siguiente repositorio para una mayor comodidad: PeticionesTFG. Este repositorio contiene también las peticiones realizadas para la sección posterior a esta.

3.1. Petición primera

Para la primera petición, directamente se incluirá la implementación en Java y se pedirá que se adapte a Julia.

```
1  Take this implementation in java and translate it to julia
2  package org.uma.jmetal.problem.multiobjective.rwa;
3
4  import java.util.List;
5  import org.uma.jmetal.problem.doubleproblem.impl.AbstractDoubleProblem;
6  import org.uma.jmetal.solution.doublesolution.DoubleSolution;
7
8  /**
9   * Problem Subasi2016 (RWA1) described in the paper "Engineering
        applications of
10  * multi-objective evolutionary algorithms: A test suite of
        box-constrained real-world
11  * problems". DOI: https://doi.org/10.1016/j.engappai.2023.106192
12  */
13 public class Subasi2016 extends AbstractDoubleProblem {
14   public Subasi2016() {
15     numberOfObjectives(2);
16     List<Double> lowerLimit = List.of(20.0, 6.0, 20.0, 0.0, 8000.0) ;
17     List<Double> upperLimit = List.of(60.0, 15.0, 40.0, 30.0, 25000.0) ;
18     name("Subasi2016");
19
20     variableBounds(lowerLimit, upperLimit);
21   }
22
23   @Override
24   public DoubleSolution evaluate(DoubleSolution solution) {
25     double H = solution.variables().get(0);
```

```
    double t = solution.variables().get(1);
    double Sy = solution.variables().get(2);
    double theta = solution.variables().get(3);
    double Re = solution.variables().get(4);

    double f;
    double Nu;

    Nu = 89.027 + 0.300 * H - 0.096 * t - 1.124 * Sy - 0.968 * theta
        + 4.148 * 10e-3 * Re + 0.0464 * H * t - 0.0244 * H * Sy
        + 0.0159 * H * theta + 4.151 * 10e-5 * H * Re + 0.1111 * t * Sy
        - 4.121 * 10e-5 * Sy * Re + 4.192 * 10e-5 * theta * Re;

    f = 0.4753 - 0.0181 * H + 0.0420 * t + 5.481 * 10e-3 * Sy - 0.0191 *
       theta
        - 3.416 * 10e-6 * Re - 8.851 * 10e-4 * H * Sy
        + 8.702 * 10e-4 * H * theta + 1.536 * 10e-3 * t * theta
        - 2.761 * 10e-6 * t * Re - 4.400 * 10e-4 * Sy * theta
        + 9.714 * 10e-7 * Sy * Re + 6.777 * 10e-4 * H * H;

    solution.objectives()[0] = -Nu;
    solution.objectives()[1] = f;

    return solution ;
  }
}
```

3.2. Petición segunda

Para la segunda petición, en lugar de proporcionar la clase entera, se hará una descripción y se proporcionará las fórmulas pertinentes para el cálculo de los objetivos.

```
I have the implementation in Java for the problem RWA1 Subasi2016. The
    class consists of a constructor and an evaluate method.
The constructor for the problem sets the number of objectives to 2, the
    name to "Subasi2016" and the lower and upper limits like this:
List<Double> lowerLimit = List.of(20.0, 6.0, 20.0, 0.0, 8000.0) ;
List<Double> upperLimit = List.of(60.0, 15.0, 40.0, 30.0, 25000.0) ;

The evaluate method recieves a DoubleSolution, updates the objectives
    and returns it back. This method gets the 5 variables in order and
    applies the formula for each objective.
These are the formulas for the objectives:
Nu = 89.027 + 0.300 * H - 0.096 * t - 1.124 * Sy - 0.968 * theta
  + 4.148 * 10e-3 * Re + 0.0464 * H * t - 0.0244 * H * Sy
  + 0.0159 * H * theta + 4.151 * 10e-5 * H * Re + 0.1111 * t * Sy
  - 4.121 * 10e-5 * Sy * Re + 4.192 * 10e-5 * theta * Re;

f = 0.4753 - 0.0181 * H + 0.0420 * t + 5.481 * 10e-3 * Sy - 0.0191 *
    theta
  - 3.416 * 10e-6 * Re - 8.851 * 10e-4 * H * Sy
  + 8.702 * 10e-4 * H * theta + 1.536 * 10e-3 * t * theta
  - 2.761 * 10e-6 * t * Re - 4.400 * 10e-4 * Sy * theta
  + 9.714 * 10e-7 * Sy * Re + 6.777 * 10e-4 * H * H;

Nu = -Nu;
f = f;

Translate this Java class into an implementation in Julia, considering
    that the DoubleSolution is already implemented as a struct and is
    called ContinuousSolution:
```

```
mutable struct ContinuousSolution{T<:Number} <: Solution
    variables::Array{T}
    objectives::Array{Real}
    constraints::Array{Real}
    attributes::Dict
    bounds::Array{Bounds{T}}
end
```

3.3. Petición tercera

La tercera petición proporcionará un ejemplo de cómo otro problema es implementado, tanto en Java como en Julia, para después proporcionar la clase entera que se busca traducir en Java y solicitarla en Julia. El objetivo de añadir otro problema como referencia es ampliar el contexto y definir qué forma debería tener la implementación.

```
This is the Java implementation for the Fonseca problem:
package org.uma.jmetal.problem.multiobjective;

import java.util.ArrayList;
import java.util.List;
import org.uma.jmetal.problem.doubleproblem.impl.AbstractDoubleProblem;
import org.uma.jmetal.solution.doublesolution.DoubleSolution;

/**
 * Class representing problem Fonseca
 */
@SuppressWarnings("serial")
public class Fonseca extends AbstractDoubleProblem {
  ...
}


And this is how it is implemented in Julia:
function fonseca()
  ...
end

Translate the Subasi2016 problem to Julia, which implementation in Java
    is this:
package org.uma.jmetal.problem.multiobjective.rwa;

import java.util.List;
import org.uma.jmetal.problem.doubleproblem.impl.AbstractDoubleProblem;
import org.uma.jmetal.solution.doublesolution.DoubleSolution;

/**
 * Problem Subasi2016 (RWA1) described in the paper "Engineering
    applications of
 * multi-objective evolutionary algorithms: A test suite of
    box-constrained real-world
 * problems". DOI: https://doi.org/10.1016/j.engappai.2023.106192
 */
public class Subasi2016 extends AbstractDoubleProblem {
  public Subasi2016() {
    numberOfObjectives(2);
    List<Double> lowerLimit = List.of(20.0, 6.0, 20.0, 0.0, 8000.0) ;
    List<Double> upperLimit = List.of(60.0, 15.0, 40.0, 30.0, 25000.0) ;
    name("Subasi2016");
```

```
41
42      variableBounds(lowerLimit, upperLimit);
43    }
44
45    @Override
46    public DoubleSolution evaluate(DoubleSolution solution) {
47      double H = solution.variables().get(0);
48      double t = solution.variables().get(1);
49      double Sy = solution.variables().get(2);
50      double theta = solution.variables().get(3);
51      double Re = solution.variables().get(4);
52
53      double f;
54      double Nu;
55
56      Nu = 89.027 + 0.300 * H - 0.096 * t - 1.124 * Sy - 0.968 * theta
57          + 4.148 * 10e-3 * Re + 0.0464 * H * t - 0.0244 * H * Sy
58          + 0.0159 * H * theta + 4.151 * 10e-5 * H * Re + 0.1111 * t * Sy
59          - 4.121 * 10e-5 * Sy * Re + 4.192 * 10e-5 * theta * Re;
60
61      f = 0.4753 - 0.0181 * H + 0.0420 * t + 5.481 * 10e-3 * Sy - 0.0191 *
            theta
62          - 3.416 * 10e-6 * Re - 8.851 * 10e-4 * H * Sy
63          + 8.702 * 10e-4 * H * theta + 1.536 * 10e-3 * t * theta
64          - 2.761 * 10e-6 * t * Re - 4.400 * 10e-4 * Sy * theta
65          + 9.714 * 10e-7 * Sy * Re + 6.777 * 10e-4 * H * H;
66
67      solution.objectives()[0] = -Nu;
68      solution.objectives()[1] = f;
69
70      return solution ;
71    }
72  }
```

3.4. Resultados

Se muestra en el Cuadro 1 una recopilación de los resultados obtenidos:

Petición	ChatGPT 4-Turbo	Gemini 1.5 Pro
Petición 1	8/10	0/10
Petición 2	9/10	8/10
Petición 3	10/10	2/10

Cuadro 1: Tasa de aciertos de las peticiones en los distintos LLMs

Es fácil darse cuenta de que en la tabla no aparece Claude AI. Por desgracia, tras tan solo cuatro peticiones, las cuales ni siquiera fueron realizadas en el mismo día, apareció la notificación de la Figura 1.

En base a esto, se ha decidido dejar de usar este LLM en esta comparativa, al no proporcionar suficiente capacidad para la alta cantidad de peticiones necesaria. No ha

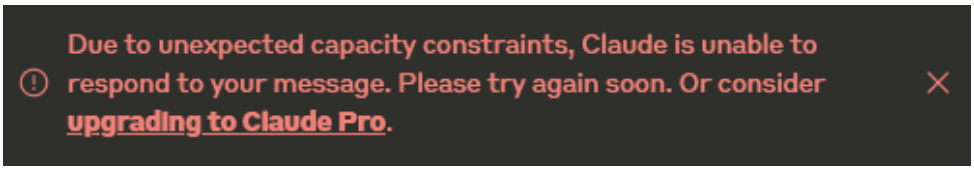

Figura 1: Aviso de Claude AI de capacidad superada

sido este el caso con ChatGPT ni Gemini, aunque sí es cierto que un mensaje similar ha aparecido en alguna ocasión al usar ChatGPT, pero sin llegar a limitar su uso. Gemini no ha presentado ningún síntoma similar.

En cuanto al rendimiento de ChatGPT y Gemini, se puede apreciar en sus tasas de aciertos una clara diferencia. ChatGPT supera a Gemini en todas las peticiones.

4

Implementación del Algoritmo MOEA/D

Esta sección se centra en el desarrollo de la metaheurística multiobjetivo MOEA/D. El objetivo principal es obtener una implementación completa del algoritmo con la ayuda de los LLMs, mientras que la comparación entre ellos pasa a un segundo plano. De tal manera, si el primer LLM utilizado para escribir el código de un componente proporciona código directamente funcional o fácilmente adaptable, no será necesario realizar la misma consulta a otros modelos. Sin embargo, se valorarán todas las implementaciones obtenidas siguiendo las valoraciones especificadas:

- 5 puntos: implementación directamente utilizable.
- 4 puntos: implementación utilizable tras realizar pocos cambios.
- 3 puntos: implementación utilizable tras realizar muchos cambios.
- 2 puntos: alguna parte de la implementación aprovechable.
- 1 punto: implementación no aprovechable.

Se parte de la implementación existente en jMetal, así como de los componentes ya implementados en MetaJul. En la Figura 2 se muestra un diagrama que representa los tipos no oficiales de Java de los cuales hace uso la clase *MOEADBuilder* existente en jMetal. De igual manera se muestran los tipos necesarios para construir estos tipos. Se representan encuadrados con línea continua las clases y con línea discontinua las interfaces.

El correcto funcionamiento de cada una de las implementaciones ha sido comprobado gracias al uso de pruebas unitarias. Estas pruebas han sido también generadas en su

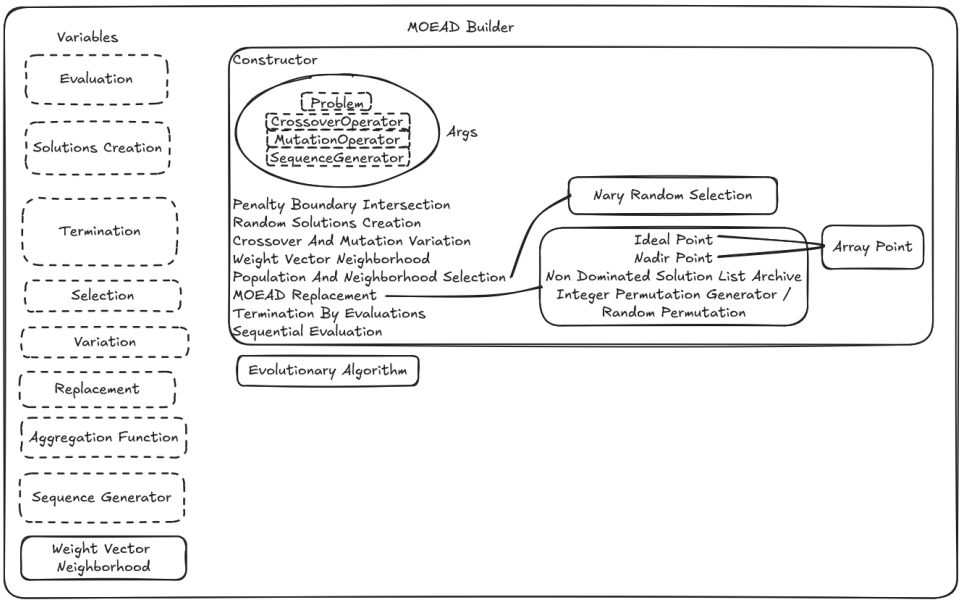

Figura 2: Clase MOEADBuilder en jMetal

mayor parte por los LLM y al igual que cada petición realizada, se pueden encontrar en el siguiente repositorio destinado a recopilarlas: PeticionesTFG. Este repositorio contiene también las peticiones realizadas para la sección anterior. En este caso, no se han incluido ciertas peticiones insatisfactorias, pues no supusieron ninguna aportación para el proyecto.

A continuación se detallan los puntos a destacar de cada implementación que ha sido requerida para finalmente obtener un algoritmo completo y funcional.

4.1. Interfaz AggregationFunction

La interfaz *AggregationFunction* en jMetal se implementa de la siguiente forma:

```
1 public interface AggregationFunction {
2   double compute(double[] vector, double[] weightVector, IdealPoint
       idealPoint, NadirPoint nadirPoint) ;
3   boolean normalizeObjectives() ;
4   void epsilon(double value) ;
5 }
```

Se puede observar que hay dos clases, *NadirPoint* e *IdealPoint*, que han de ser implementadas para ser usadas en el método *compute()*.

4.1.1. Interfaz Point y subclases

Al analizar las clases *NadirPoint* e *IdealPoint* en jMetal se observa que ambas implementan la interfaz *Point*:

```
public interface Point {
  int dimension();
  double[] values() ;
  double value(int index) ;
  void value(int index, double value) ;
  void update(double[] point) ;
  void set(double[] point) ;
}
```

pero lo hacen extendiendo la clase *ArrayPoint*, que utiliza arrays para implementar el concepto de punto. Dado que en Julia no existe el mecanismo de herencia de clases se asume que los LLMs no van a generar un código Julia 100 % a partir de los disponibles en jMetal. En primer lugar, se plantea tanto a ChatGPT4 como a Gemini1.5 generar un equivalente a la interfaz *Point*. La respuesta de Gemini es un código Julia inválido, mientras que ChatGPT sí devuelte una implementación que tiene sentido:

```
abstract type Point end

function dimension(p::Point)::Int
    throw(MethodError(dimension, (p,)))
end

function values(p::Point)::Vector{Float64}
    throw(MethodError(values, (p,)))
end

function value(p::Point, index::Int)::Float64
    throw(MethodError(value, (p, index)))
end

function value!(p::Point, index::Int, val::Float64)
    throw(MethodError(value!, (p, index, val)))
end

function update!(p::Point, point::Vector{Float64})
    throw(MethodError(update!, (p, point)))
end

function set!(p::Point, point::Vector{Float64})
    throw(MethodError(set!, (p, point)))
end
```

En vista de este resultado, se realizan las siguientes consultas a ChatGPT:

1. Escribir el código Julia equivalente a la clase *ArrayPoint* de jMetal.

2. Escribir el código Julia equivalente a la clase *IdealPoint*, que hereda de ArrayPoint

3. Buscar si hay paquetes en Julia que ofrezcan estructuras y funciones similares a las de la interfaz *Point* y resto de implementaciones.

La respuesta a la tercera consulta sugiere usar tres paquetes ya existentes, lo que aparenta ser muy prometedor aunque los códigos que se indican son más complicados que los generados en las consultas primera y segunda. Además, se observa un error en el código generado (simula una herencia de estructuras en Julia, lo que no está permitido en el lenguaje). En consecuencia, se opta por usar los códigos de las consultas primera y segunda, aunque es necesario realizar algunas correcciones relacionadas con los índices de los puntos: en Java los índices de arrays y listas empiezan por 0, mientras que en Julia empiezan por 1 y el código generado de *IdealPoint* no incluye las implementaciones de los métodos que hay que reusar de *ArrayPoint*, por lo que hay que escribirlos manualmente.

No es necesario solicitar la implementación de *NadirPoint*, ya que es el mismo código que IdealPoint, simplemente cambiando los nombres.

4.1.2. Funciones agregativas

Una vez que se dispone de una versión en MetaJul de las clases *NadirPoint* e *IdealPoint*, el siguiente paso es desarrollar en Julia el código que implementa la interfaz *AggregationFunction* que utilizará el algoritmo, *PenaltyBoundaryIntersection*. Se muestra a continuación el código en Java:

```
1  public class PenaltyBoundaryIntersection implements AggregationFunction {
2    private double epsilon = 0.000001 ;
3    private boolean normalizeObjectives ;
4
5    private final double theta ;
6
7    public PenaltyBoundaryIntersection(double theta, boolean
        normalizeObjectives) {
8      this.theta = theta ;
9      this.normalizeObjectives = normalizeObjectives ;
10   }
11
12   @Override
13   public double compute(double[] vector, double[] weightVector,
        IdealPoint idealPoint, NadirPoint nadirPoint) {
14     double d1, d2, nl;
15
16     d1 = d2 = nl = 0.0;
17
18     for (int i = 0; i < vector.length; i++) {
19       double value ;
20       if (normalizeObjectives) {
21         value = (vector[i] -
              idealPoint.value(i))/(nadirPoint.value(i)-idealPoint.value(i)+epsilon)
              ;
```

```
      } else {
        value = vector[i] - idealPoint.value(i) ;
      }
      d1 += value * weightVector[i];
      nl += Math.pow(weightVector[i], 2.0);
    }
    nl = Math.sqrt(nl);
    d1 = Math.abs(d1) / nl;

    for (int i = 0; i < vector.length; i++) {
      double value ;
      if (normalizeObjectives) {
        value = (vector[i] -
            idealPoint.value(i))/(nadirPoint.value(i)-idealPoint.value(i))
            ;
      } else {
        value = vector[i] - idealPoint.value(i);
      }
      d2 += Math.pow(value - d1 * (weightVector[i] / nl), 2.0);
    }
    d2 = Math.sqrt(d2);

    return (d1 + theta * d2) ;
  }

  @Override
  public void epsilon(double epsilon) {
    this.epsilon = epsilon ;
  }

  @Override
  public boolean normalizeObjectives() {
    return this.normalizeObjectives ;
  }
}
```

A priori no es un código complejo, por lo que cabría esperar que los LLMs puedan generar gran parte del código en Julia. Las consultas que se plantean consisten simplemente en dar las clases Java e indicar que se genere una implementación en Julia. La respuesta de Gemini es, de nuevo, código Julia incorrecto, mientras que el código generado por ChatGPT, salvo ligeras modificaciones para adaptarlo al proyecto MetaJul, es completamente válido.

```
# Define the PenaltyBoundaryIntersection struct
struct PenaltyBoundaryIntersection <: AggregationFunction
    normalizeObjectives::Bool
    epsilon::Float64
    theta::Float64

    PenaltyBoundaryIntersection() = new(false, 1e-6, 5.0)
    PenaltyBoundaryIntersection(theta::Float64,
        normalizeObjectives::Bool) = new(normalizeObjectives, 1e-6, theta)
end

# Compute function
function compute(
    aggFunction::PenaltyBoundaryIntersection, vector::Vector{Float64},
```

```
        weightVector::Vector{Float64},
    idealPoint::IdealPoint, nadirPoint::NadirPoint
)
    d1 = 0.0
    d2 = 0.0
    nl = 0.0

    # Calculate d1 and nl
    for i in 1:length(vector)
        tmpValue = if aggFunction.normalizeObjectives
            (vector[i] - value(idealPoint, i)) /
            (value(nadirPoint, i) - value(idealPoint, i) +
                aggFunction.epsilon)
        else
            vector[i] - value(idealPoint, i)
        end
        d1 += tmpValue * weightVector[i]
        nl += weightVector[i]^2
    end

    nl = sqrt(nl)
    d1 = abs(d1) / nl

    # Calculate d2
    for i in 1:length(vector)
        tmpValue = if aggFunction.normalizeObjectives
            (vector[i] - value(idealPoint, i)) /
            (value(nadirPoint, i) - value(idealPoint, i))
        else
            vector[i] - value(idealPoint, i)
        end
        d2 += (tmpValue - d1 * (weightVector[i] / nl))^2
    end
    d2 = sqrt(d2)

    return d1 + aggFunction.theta * d2
end
```

Clase/Interfaz	GPT	Claude	Gemini
Point	5		1
ArrayPoint	4		
IdealPoint	3		
PenaltyBoundaryIntersection	5		

Cuadro 2: Puntuación Interfaz AggregationFunction

4.2. Operadores

4.2.1. Operador NaryRandomSelection

Antes de adaptar el operador de selección *PopulationAndNeighborhoodSelection*, conviene tener preparado *NaryRandomSelection*, ya que el primero utiliza el segundo. Esta es la implementación en Java:

```java
public class NaryRandomSelection<S> implements
    SelectionOperator<List<S>, List<S>> {
  private int numberOfSolutionsToBeReturned;

  /** Constructor */
  public NaryRandomSelection() {
    this(1);
  }

  /** Constructor */
  public NaryRandomSelection(int numberOfSolutionsToBeReturned) {
    this.numberOfSolutionsToBeReturned = numberOfSolutionsToBeReturned;
  }

  /** Execute() method */
  public List<S> execute(List<S> solutionList) {
    Check.notNull(solutionList);
    Check.collectionIsNotEmpty(solutionList);
    Check.that(
        solutionList.size() >= numberOfSolutionsToBeReturned,
        "The solution list size ("
            + solutionList.size()
            + ") is less than "
            + "the number of requested solutions ("
            + numberOfSolutionsToBeReturned
            + ")");

    return ListUtils.randomSelectionWithoutReplacement(
        numberOfSolutionsToBeReturned, solutionList);
  }
}
```

No debería ser complicado para los LLMs de traducir pues es bastante simple.

```julia
struct NaryRandomSelectionOperator <: SelectionOperator
  numberOfSolutionsToBeReturned::Int
  # Constructor to handle default value
  NaryRandomSelectionOperator(n::Int = 1) = new(n)
end

function select(x::Vector,
    selectionOperator::NaryRandomSelectionOperator)
  @assert length(x) >= selectionOperator.numberOfSolutionsToBeReturned
      "Solution list size ($length(x)) is less than the number of
      requested solutions
      ($(selectionOperator.numberOfSolutionsToBeReturned))"

  return
      x[randperm(length(x))[1:selectionOperator.numberOfSolutionsToBeReturned]]
end
```

En este caso, la solución es una combinación del código generado por ChatGPT y Gemini. Se ha seleccionado el struct y el constructor de ChatGPT y la función select de Gemini.

4.2.2. IntegerBoundedSequenceGenerator y IntegerPermutationGenerator

Con respecto a los generadores de secuencia, no se espera que resulten problemáticos, pues son bastante simples.

```
1  public class IntegerBoundedSequenceGenerator implements
       SequenceGenerator<Integer> {
2    private int index;
3    private int size ;
4
5    public IntegerBoundedSequenceGenerator(int size) {
6      Check.that(size > 0, "Size " + size + " is not a positive number
           greater than zero");
7      this.size = size ;
8      index = 0;
9    }
10
11   @Override
12   public void generateNext() {
13     index++;
14     if (index == size) {
15       index = 0;
16     }
17   }
18 }
19
20 public class IntegerPermutationGenerator implements
       SequenceGenerator<Integer> {
21   private int[] sequence;
22   private int index;
23   private int size ;
24
25   public IntegerPermutationGenerator(int size) {
26     Check.that(size > 0, "Size " + size + " is not a positive number
           greater than zero");
27     this.size = size ;
28     sequence = randomPermutation(size) ;
29
30     index = 0;
31   }
32
33   @Override
34   public void generateNext() {
35     index++;
36     if (index == sequence.length) {
37       sequence = randomPermutation(size) ;
38       index = 0;
39     }
40   }
41
42   private int[] randomPermutation(int size) {
43     int[] permutation = new int[size] ;
44     JMetalRandom randomGenerator = JMetalRandom.getInstance() ;
45     int[] index = new int[size];
46     boolean[] flag = new boolean[size];
47
```

```
48      for (int n = 0; n < size; n++) {
49        index[n] = n;
50        flag[n] = true;
51      }
52
53      int num = 0;
54      while (num < size) {
55        int start = randomGenerator.nextInt(0, size - 1);
56        while (true) {
57          if (flag[start]) {
58            permutation[num] = index[start];
59            flag[start] = false;
60            num++;
61            break;
62          }
63          if (start == (size - 1)) {
64            start = 0;
65          } else {
66            start++;
67          }
68        }
69      }
70      return permutation ;
71    }
72  }
```

Las implementaciones de ChatGPT son lo suficientemente buenas. El único cambio realizado es el índice inicial, pues en Julia los índices de las estructuras de datos empiezan en 1, a diferencia de Java.

```
1  struct IntegerBoundedSequenceGenerator <: SequenceGenerator
2      index::Int
3      size::Int
4
5      function IntegerBoundedSequenceGenerator(size::Int)
6          @assert size > 0 "Size $size is not a positive number greater
               than zero"
7          new(1, size)
8      end
9  end
10
11 mutable struct IntegerPermutationGenerator <: SequenceGenerator
12     sequence::Vector{Int}
13     index::Int
14     size::Int
15
16     function IntegerPermutationGenerator(size::Int)
17         @assert size > 0 "Size $size is not a positive number greater
               than zero"
18         new(randomPermutation(size), 1, size)
19     end
20 end
```

Aunque en su mayoría el código es correcto, es un desliz importante teniendo en cuenta que es una característica del lenguaje conocida.

4.2.3. PopulationAndNeighborhoodSelection

Habiendo implementado estos componentes, ya es posible continuar con *PopulationAndNeighborhoodSelection*. El código de Gemini está incompleto y no es lo suficientemente bueno en comparación con el de ChatGPT. En este caso, ha sido necesario eliminar la parametrización, pues estaba siendo la causa de fallo de las pruebas. La función *getNeighbors* no es necesario llamarla a través de *selection.neighborhood*, sino que se pasa como parámetro a la función. Esto se debe al del proyecto. Pasa lo mismo con *selectionOperator* y *getValue*.

Adicionalmente, como simplificación del proyecto, el tipo *NeighborType* ha sido reemplazado por un booleano, pues solo se hará uso de dos de los valores de la numeración, *NEIGHBOR* y *POPULATION*.

```
1  # Selection method for PopulationAndNeighborhoodSelection
2  function select(selection::PopulationAndNeighborhoodSelection,
       solutionList::Vector{T})::Vector{T} where T
3      matingPool = Vector{T}()
4      randomValue = rand()
5
6      if randomValue < selection.neighborhoodSelectionProbability
7          # Select from neighborhood
8          neighborType = true
9          neighbors = getNeighbors(selection.neighborhood, solutionList,
               getValue(selection.solutionIndexGenerator))
10         matingPool = select(neighbors, selection.selectionOperator)
11     else
12         # Select from population
13         neighborType = false
14         matingPool = select(solutionList, selection.selectionOperator)
15     end
16
17     if selection.selectCurrentSolution
18         # Add the current solution to the mating pool
19         currentSolution =
               solutionList[getValue(selection.solutionIndexGenerator)]
20         push!(matingPool, currentSolution)
21     end
22
23     @assert length(matingPool) == selection.matingPoolSize string("The
           mating pool size ", length(matingPool), " is not equal to the
           required size ", selection.matingPoolSize)
24
25     return matingPool
26 end
```

4.3. WeightVectorNeighborhood

Lamentablemente, las peticiones exactas, así como el código generado, no fueron guardadas para esta sección. Sin embargo, eran similares al resto, siguiendo el patrón de

Clase/Interfaz	GPT	Claude	Gemini
NaryRandomSelection	5		5
IntegerBoundedSequenceGenerator	4		
IntegerPermutationGenerator	4		
PopulationAndNeighborhoodSelection	3		1

Cuadro 3: Puntuación Operadores

aportar la implementación en Java y solicitarla en Julia. Según las notas recogidas, el contructor generado por Claude es completamente utilizable, la función *minFastSort!* fue generado correctamente por Claude y Gemini, y los métodos *initializeNeighborhood* son correctos tal cual los escribió ChatGPT.

Cabe destacar nuevamente la simplificación de *NeighborType* por un booleano.

```
# Gemini and Claude code
function minFastSort!(x::Vector{Float64}, idx::Vector{Int}, n::Int,
    m::Int)
    for i in 1:m
        for j in i+1:n
            if x[i] > x[j]
                x[i], x[j] = x[j], x[i]
                idx[i], idx[j] = idx[j], idx[i]
            end
        end
    end
end

struct WeightVectorNeighborhood <: Neighborhood
    numberOfWeightVectors::Int
    weightVectorSize::Int
    neighborhoodSize::Int
    neighborhood::Array
    weightVector::Array{Float64,2}
    # false=Population, true=Neighbor
    neighborType::Bool

    function WeightVectorNeighborhood(numberOfWeightVectors::Int,
        neighborhoodSize::Int)
        weightVectorSize = 2

        neighborhood = Array{Int}(undef, numberOfWeightVectors,
            neighborhoodSize)
        weightVector = Array{Float64}(undef, numberOfWeightVectors,
            weightVectorSize)

        for n in 1:numberOfWeightVectors
            a = 1.0 * (n - 1) / (numberOfWeightVectors - 1)
            weightVector[n, 1] = a
            weightVector[n, 2] = 1 - a
        end

        weightVectorNeighborhood = new(numberOfWeightVectors,
            weightVectorSize, neighborhoodSize, neighborhood,
```

```
            weightVector, false)
35          initializeNeighborhood(weightVectorNeighborhood)
36
37          return weightVectorNeighborhood
38      end
39
40      function WeightVectorNeighborhood(numberOfWeightVectors::Int,
            weightVectorSize::Int, neighborhoodSize::Int,
            vectorDirectoryName::String)
41
42          neighborhood = Array{Int}(undef, numberOfWeightVectors,
                neighborhoodSize)
43          #weightVector = Array{Float64}(undef, numberOfWeightVectors,
                weightVectorSize)
44
45          weightVectorFileName = joinpath(vectorDirectoryName,
                "W$(weightVectorSize)D_$(numberOfWeightVectors).dat")
46          weightVector = readWeightVectors(weightVectorFileName)
47
48          weightVectorNeighborhood = new(numberOfWeightVectors,
                weightVectorSize, neighborhoodSize, neighborhood,
                weightVector, false)
49
50          initializeNeighborhood(weightVectorNeighborhood)
51
52          return weightVectorNeighborhood
53      end
54
55  end
56
57  function
        initializeNeighborhood(weightVectorNeighborhood::WeightVectorNeighborhood)
58      numberOfWeightVectors = weightVectorNeighborhood.numberOfWeightVectors
59      neighborhoodSize = weightVectorNeighborhood.neighborhoodSize
60      weightVector = weightVectorNeighborhood.weightVector
61      neighborhood = weightVectorNeighborhood.neighborhood
62
63      x = Vector{Float64}(undef, numberOfWeightVectors)
64      idx = Vector{Int}(undef, numberOfWeightVectors)
65
66      for i in 1:numberOfWeightVectors
67          # calculate the distances based on weight vectors
68          for j in 1:numberOfWeightVectors
69              x[j] = norm(weightVector[i, :] - weightVector[j, :])
70              idx[j] = j
71          end
72
73          # find 'niche' nearest neighboring sub problems
74          minFastSort!(x, idx, numberOfWeightVectors, neighborhoodSize)
75
76          neighborhood[i, :] .= idx[1:neighborhoodSize]
77      end
78  end
79
80  function
        getNeighbors(weightVectorNeighborhood::WeightVectorNeighborhood,
        solutionList::Vector{S}, solutionIndex::Int) where {S <: Solution}
81      neighbourSolutions = Vector{S}()
82      for neighborIndex in
            weightVectorNeighborhood.neighborhood[solutionIndex, :]
83          push!(neighbourSolutions, solutionList[neighborIndex])
84      end
85      return neighbourSolutions
86  end
```

Clase/Interfaz	GPT	Claude	Gemini
minFastSort!	5	5	5
constructor	4	5	4
initializeNeighborhood	5	4	4
getNeighbors	5	5	3

Cuadro 4: Puntuación WeightVectorNeighborhood

4.4. MOEAD

4.4.1. MOEADReplacement

La clase *MOEADReplacement* ha resultado complicada de adaptar a Julia. Ninguna de las implementaciones proporcionadas por los distintos LLM eran lo suficientemente buenas, y finalmente fue necesaria bastante intervención humana para detectar los fallos generados por los múltiples errores.

Entre los errores corregidos se encontraron tipos mal especificados, funciones que modifican los argumentos recibidos sin necesidad, funciones no implementadas, las llamadas a la función *compute* pasan argumentos incorrectos y uso incorrecto de las permutaciones. Tras correcciones humanas, pues los LLMs tampoco fueron capaces de arreglar los errores, se logró alcanzar la siguiente implementación funcional.

```
mutable struct MOEADReplacement <: Replacement
    matingPoolSelection::PopulationAndNeighborhoodSelection
    weightVectorNeighborhood::WeightVectorNeighborhood
    aggregationFunction::AggregationFunction
    sequenceGenerator::SequenceGenerator
    maximumNumberOfReplacedSolutions::Int
    normalize::Bool
    idealPoint::IdealPoint
    nadirPoint::NadirPoint
    nonDominatedArchive::NonDominatedArchive
    firstReplacement::Bool

    MOEADReplacement(matingPoolSelection, weightVectorNeighborhood,
        aggregationFunction, sequenceGenerator, maxReplaced, normalize) =
        new(
        matingPoolSelection,
        weightVectorNeighborhood,
        aggregationFunction,
        sequenceGenerator,
        maxReplaced,
        normalize,
        IdealPoint(),
        NadirPoint(),
        NonDominatedArchive(ContinuousSolution{Float64}),
        true
```

```
    )
end

function replace_(replacement::MOEADReplacement, population::Vector{T},
    offspring::Vector{T}) where {T <: Solution}
    new_solution = offspring[1]

    update_ideal_point!(replacement, population, new_solution)
    update_nadir_point!(replacement, population, new_solution)

    neighborType =
        replacement.matingPoolSelection.neighborhood.neighborType
    # NeighborType Neighbor = true
    randomPermutation = if neighborType == true
        IntegerPermutationGenerator(
      size(replacement.{...}.neighborhoodSize, 1)
    )
    else
        IntegerPermutationGenerator(length(population))
    end

    replacements = 0
    while replacements < replacement.maximumNumberOfReplacedSolutions &&
        hasNext(randomPermutation)
        k = 1
        if neighborType
            k = replacement.weightVectorNeighborhood
        .neighborhood[sequenceGenerator.Value][randomPermutation.value]
        else
            k = getValue(randomPermutation)
        end

        generateNext!(randomPermutation)

        f1 = compute(replacement.aggregationFunction,
            population[k].objectives,
            replacement.weightVectorNeighborhood.weightVector[k, :],
            replacement.idealPoint, replacement.nadirPoint)
        f2 = compute(replacement.aggregationFunction,
            new_solution.objectives,
            replacement.weightVectorNeighborhood.weightVector[k, :],
            replacement.idealPoint, replacement.nadirPoint)

        if f2 < f1
            population[k] = copySolution(new_solution)
            replacements += 1
        end
    end

    generateNext!(replacement.sequenceGenerator)
    return population
end
```

Nota: el código ha tenido que ser alterado para poder ser visualizado correctamente en este documento. La versión original se encuentra en el fichero src/component/evolutionaryAlgorithm/replacement.jl.

4.4.2. MOEAD

Completamente opuesto es el caso de *MOEAD*. Se trata de código bastante simple, compuesto mayormente por llamadas a funciones ya existentes, pues ya se dispone de todos los componentes necesarios para implementar el algoritmo.

```
problem::Problem
    populationSize::Int
    neighborhoodSize::Int
    maxReplacedSolutions::Int
    neighborhoodSelectionProbability::Float64
    aggregationFunction::AggregationFunction
    crossover::CrossoverOperator
    mutation::MutationOperator
    sequenceGenerator::SequenceGenerator
    termination::Termination
    normalize::Bool

    solver::EvolutionaryAlgorithm

    function MOEAD(
        problem::ContinuousProblem;
        populationSize=100,
        neighborhoodSize=20,
        maxReplacedSolutions=2,
        neighborhoodSelectionProbability=0.9,
        aggregationFunction=PenaltyBoundaryIntersection(5.0, false),
        crossover=SBXCrossover(probability=1.0, distributionIndex=20.0,
            bounds=problem.bounds),
        mutation=PolynomialMutation(probability=1.0 /
            numberOfVariables(problem), distributionIndex=20.0,
            bounds=problem.bounds),
        sequenceGenerator=IntegerPermutationGenerator(populationSize),
        termination=TerminationByEvaluations(25000),
        normalize=false
    )
        algorithm = new()
        algorithm.solver = EvolutionaryAlgorithm()
        algorithm.solver.name = "MOEA/D"
        algorithm.problem = problem
        algorithm.populationSize = populationSize
        algorithm.neighborhoodSize = neighborhoodSize
        algorithm.maxReplacedSolutions = maxReplacedSolutions
        algorithm.neighborhoodSelectionProbability =
            neighborhoodSelectionProbability
        algorithm.aggregationFunction = aggregationFunction
        algorithm.crossover = crossover
        algorithm.mutation = mutation
        algorithm.sequenceGenerator = sequenceGenerator
        algorithm.termination = termination
        algorithm.normalize = normalize

        return algorithm
    end

    function optimise!(moead::MOEAD)
        solver = moead.solver
        problem = moead.problem

        solver.solutionsCreation = DefaultSolutionsCreation(problem,
            moead.populationSize)
```

```
51          solver.evaluation = SequentialEvaluation(problem)
52          solver.termination = moead.termination
53
54          variation = CrossoverAndMutationVariation(1, moead.crossover,
                moead.mutation)
55          solver.variation = variation
56
57          if problem.numberOfObjectives == 2
58              neighborhood = WeightVectorNeighborhood(moead.populationSize,
                    moead.neighborhoodSize)
59          else
60              #throw(DomainError("Number of objectives > 2 is not currently
                    supported."))
61              neighborhood = WeightVectorNeighborhood(moead.populationSize,
                    problem.numberOfObjectives, moead.neighborhoodSize,
                    "path/to/weight-vectors.dat")
62          end
63
64          selection = PopulationAndNeighborhoodSelection(
65              variation.matingPoolSize,
66              moead.sequenceGenerator,
67              neighborhood,
68              moead.neighborhoodSelectionProbability,
69              true
70          )
71          solver.selection = selection
72
73          replacement = MOEADReplacement(
74              selection,
75              neighborhood,
76              moead.aggregationFunction,
77              moead.sequenceGenerator,
78              moead.maxReplacedSolutions,
79              moead.normalize
80          )
81          solver.replacement = replacement
82
83          return evolutionaryAlgorithm(solver)
84      end
```

Clase/Interfaz	GPT	Claude	Gemini
MOEADReplacement	3	3	2
MOEAD	5	5	5

Cuadro 5: Puntuación MOEAD

4.5. Ejecución del algoritmo y resultados

Se procede a continuación a realizar pruebas de rendimiento para comparar el algoritmo implementado en Java y el mismo implementado en Julia. Se ha cogido como problema el conocido como ZDT1, que es un problema de optimización continua bi-objetivo que se caracterizar por ser escalable, lo que significa que el número de variables de decisión pue-

de ser variable pero el óptimo siempre es el mismo. De ese modo, se puede aumentar el esfuerzo computacional de evaluar cada solución del problema aumentando el número de variables.

```
function main()
    problem = ZDT1(numberOfVariables=2000)

    solver::EvolutionaryAlgorithm = EvolutionaryAlgorithm()
    solver.name = "MOEA/D"

    populationSize = 100
    neighborhoodSize = 20
    offspringPopulationSize = 1
    maximumNumberOfReplacedSolutions = 2
    normalizeObjectives = false

    solver.solutionsCreation = DefaultSolutionsCreation(problem,
        populationSize)
    solver.evaluation = SequentialEvaluation(problem)
    solver.termination = TerminationByEvaluations(100000)

    mutation = PolynomialMutation(probability = 1.0 /
        numberOfVariables(problem), distributionIndex = 20.0, bounds =
        problem.bounds)
    crossover = SBXCrossover(probability = 0.9, distributionIndex = 20.0,
        bounds = problem.bounds)

    """
    mutation = UniformMutation(1.0/numberOfVariables(problem), 20.0,
        problem.bounds)
    crossover = BLXAlphaCrossover(probability = 1.0, alpha = 0.5, bounds
        = problem.bounds)
    """


    solver.variation =
        CrossoverAndMutationVariation(offspringPopulationSize, crossover,
        mutation)

    neighborhood = WeightVectorNeighborhood(populationSize,
        neighborhoodSize)

    sequenceGenerator = IntegerPermutationGenerator(populationSize)

    selectCurrentSolution = true
    solver.selection =
        PopulationAndNeighborhoodSelection(solver.variation.matingPoolSize,
        sequenceGenerator, neighborhood, 0.9, selectCurrentSolution)

    aggregationFunction = PenaltyBoundaryIntersection(5.0,
        normalizeObjectives)
    #aggregationFunction = Tschebyscheff()

    solver.replacement = MOEADReplacement(solver.selection, neighborhood,
        aggregationFunction, sequenceGenerator,
        maximumNumberOfReplacedSolutions, normalizeObjectives)

    optimize!(solver)

    foundSolutions = solver.foundSolutions

    objectivesFileName = "FUN.csv"
    variablesFileName = "VAR.csv"

```

```
46      println("Algorithm: ", name(solver))
47      println("Computing time: ", computingTime(solver))
48
49      println("Objectives stored in file ", objectivesFileName)
50      printObjectivesToCSVFile(objectivesFileName, foundSolutions)
51
52      println("Variables stored in file ", variablesFileName)
53      printVariablesToCSVFile(variablesFileName, foundSolutions)
54  end
```

Para realizar la comparación se parte como referencia de los resultados obtenidos sobre ese problema con otro algoritmo, NSGA-II, que ya está incluido en MetaJul. Los resultados de este algoritmo se muestran en la Figura 3. En la figura se muestra el tiempo de ejecución cuando el problema ZDT1 se configura con 30, 100, 500, 1000 y 2000 variables.

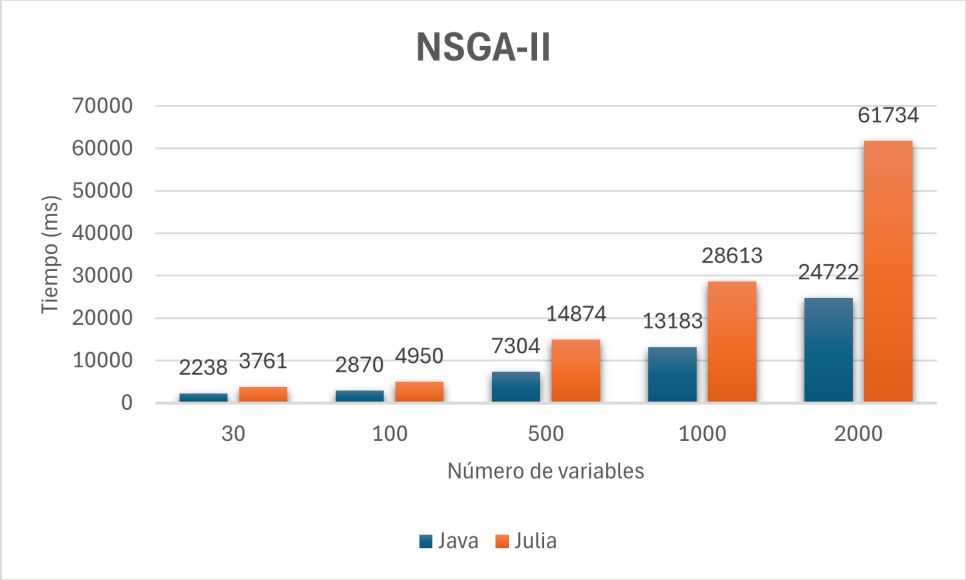

Figura 3: Comparativa de rendimiento NSGA-II en Java (jMetal) y Julia (MetaJul)

Se puede observar en la Figura 3 como el algoritmo NSGA-II se ejecuta siempre más rápido en Java que en Julia tardando este último 1,68 veces más que el primero para 30 variables, y que asciende a 2,50 veces para 2000 variables.

En el caso de MOEA/D en la Figura 4, Julia es más rápido para un número bajo de variables, equiparándose en 500 variables y finalmente termina siendo más lento a partir de ese punto, igual que lo era NSGA-II.

Estos resultados no eran los esperados antes de implementar el algoritmo en Julia. Los resultados obtenidos presentados en [10] para el algoritmo NSGA-II y problema ZDT1 son

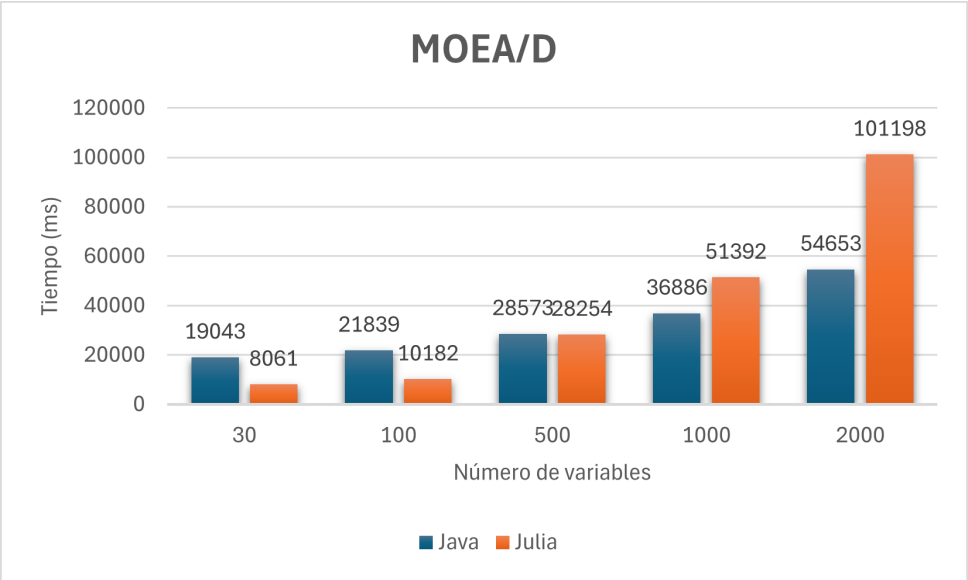

Figura 4: Comparativa de rendimiento MOEA/D en Java (jMetal) y Julia (MetaJul)

muy dispares como se puede observar en la Figura 5. Para pocas variables el rendimiento de Java es mejor que el de Julia, en algún punto pasado las 100 variables se igualan los tiempos de ejecución, y finalmente Julia acaba siendo más eficiente por un margen bastante amplio de 2,71 veces.

Estas diferencias pueden deberse a una multitud de factores, aunque el más obvio es la diferencia de hardware y sistema operativo de ejecución. Las pruebas realizadas para este trabajo han sido realizadas en Windows 11 utilizando Ubuntu 22.04.2 LTS sobre WSL y un procesador AMD Ryzen 7 5700G, con 16GB de RAM. Mientras tanto, las pruebas del artículo [10] fueron realizadas en macOS Sonoma 14.1.2, un procesador Apple M1 Max y 35GB de RAM.

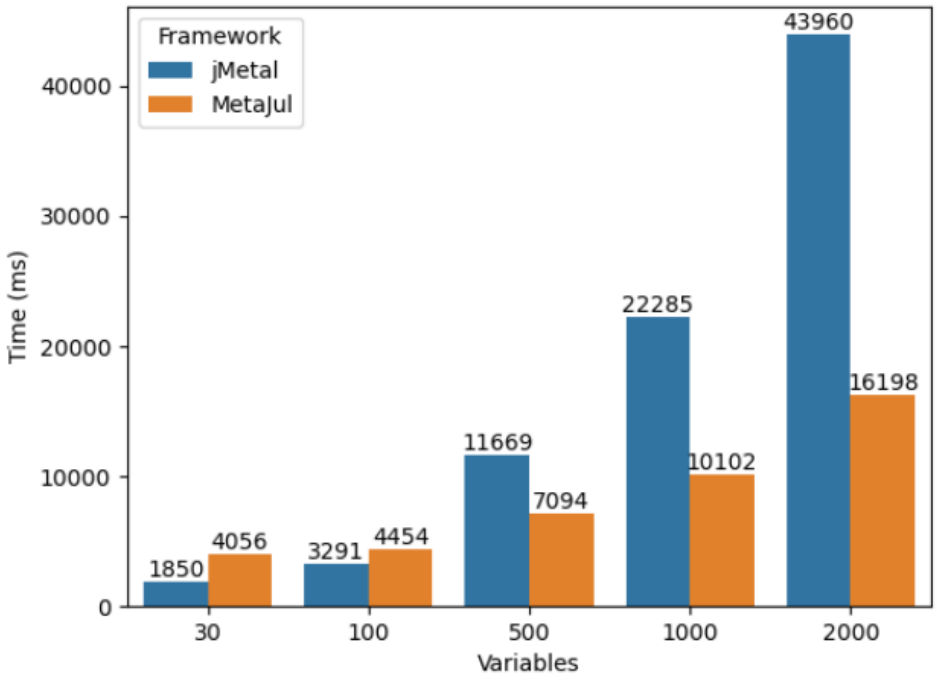

Figura 5: Comparativa de rendimiento realizada por Antonio de NSGA-II en Java (jMetal) y Julia (MetaJul)[10]

5

Conclusiones y Líneas Futuras

5.1. Conclusiones

5.1.1. Utilidad de la inteligencia artificial generativa en la programación

El uso de los distintos LLMs como asistentes al programar ha resultado cumplir con las expectativas. Si bien en ocasiones el código es erróneo o necesita cambios, siempre ha sido un buen punto de referencia. Esto es especialmente cierto para programadores con poca experiencia en el lenguaje con el que están trabajando, pues es muy poco frecuente que el LLM falle en la sintaxis.

Es cierto que, a día de hoy, los LLMs aún no tienen la capacidad de generar grandes cantidades de código y que éste sea funcional. Sin embargo, mientras se siga una filosofía de programación que permita separar las funciones en tareas simples, es posible disminuir en gran medida el tiempo necesario para escribir código.

Incluso en las ocasiones en las que el código proporcionado por los LLMs no es suficiente, generalmente se les puede pedir que traten de arreglar el error. El problema viene cuando no son capaces de solucionarlo por ellos mismos, pues queda como tarea del programador revisar y depurar un programa que no han escrito por si mismos y, posiblemente, es la primera vez que se enfrentan a tal código.

5.1.2. Rendimiento del algoritmo MOEA/D y aprovechamiento de las capacidades de Julia

En un primer momento, el rendimiento del algoritmo MOEA/D en Julia no coincidía con el esperado, pues la comparación con Java es completamente inversa a aquella del

algoritmo NSGA-II que se puede ver en el artículo redactado por Antonio J. Nebro y Xavier Gandibleux [10].

Sin embargo, tras repetir la comparación del artículo en el mismo hardware utilizado para el algoritmo MOEA/D, se puede observar una gran mejora en el rendimiento obtenido con respecto a Java para pocas variables, mientras que el rendimiento, conforme aumentan las variables, sigue siendo mejor que aquel obtenido por el algoritmo NSGA-II para el mismo problema. Poniendo todo en consideración, aunque el rendimiento de MOEA/D no sea el esperado inicialmente, sí que ha mejorado con respecto a NSGA-II.

Respecto al aprovechamiento del lenguaje Julia, queda mucho por optimizar. Las implementaciones son en su mayoría traducciones directas de Java, lo cual no es un acercamiento apropiado para lenguajes tan dispares.

5.2. Líneas Futuras

5.2.1. Expansión de MetaJul

El proyecto MetaJul es aún pequeño. Aunque parezca natural compararlo con su hermano mayor, jMetal, esta no es una comparación justa debido al diferencia de edad entre ambos proyectos. Son varios los proyectos derivados de jMetal que han cesado su desarrollo, y aunque acercar MetaJul a jMetal suena poco menos que utópico a día de hoy, un futuro en el que ambos proyectos mantienen su desarrollo es posible.

Se propone como línea seguir ampliando MetaJul siguiendo los pasos de jMetal. Actualmente Julia es un lenguaje con poco renombre que mucha gente desconoce. Python, pese a su bajo rendimiento, destaca en el ámbito científico y matemático por encima de lenguajes con mayor rendimiento como C o R por su sencillez; Julia es un buen candidato para llamar la atención y MetaJul es una oportunidad para que el lenguaje se de a conocer en el sector.

5.2.2. Mejora del rendimiento

En este proyecto el rendimiento no se ha tenido en cuenta a la hora del desarrollo. Debido al papel protagonista de la implementación asistida por inteligencia artificial, gran cantidad del código producido es una adaptación directa del código Java a Julia.

Con un acercamiento de este estilo, se pone el foco en el correcto funcionamiento de los componentes y problemas, dejando a un lado el rendimiento.

Por esta razón, se propone una línea en la que se tome como base el proyecto de MetaJul y se haga un análisis de rendimiento en profundidad, observando los puntos débiles de la implementación actual y tratando de hacer justicia a la filosofía de Julia, sencillez sin sacrificar el rendimiento.

Referencias

[1] Antonio Benítez-Hidalgo, Antonio J. Nebro, José García-Nieto, Izaskun Oregi, and Javier Del Ser. jmetalpy: A python framework for multi-objective optimization with metaheuristics. *Swarm Evol. Comput.*, 51, 2019.

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[3] C. Blum and A. Roli. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[4] C.A. Coello Coello, G.B. Lamont, and D.A. van Veldhuizen. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, Inc. 2nd Ed., NY, USA, 2007.

[5] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[6] Kalyanmoy Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley Sons, Inc., USA, 2001.

[7] J.J. Durillo and A.J. Nebro. jmetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760 – 771, 2011.

[8] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A comprehensive overview of large language models, 2024.

[9] A.J. Nebro, Juan J. Durillo, and M. Vergne. Redesigning the jMetal multi-objective optimization framework. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, pages 1093–1100, New York, NY, USA, 2015. ACM.

[10] Antonio J. Nebro and Xavier Gandibleux. Experiences using julia for implementing multi-objective evolutionary algorithms. In Marc Sevaux, Alexandru-Liviu Olteanu, Eduardo G. Pardo, Angelo Sifaleras, and Salma Makboul, editors, *Metaheuristics - 15th International Conference, MIC 2024, Lorient, France, June 4-7, 2024, Proceedings, Part II*, volume 14754 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 2024.

[11] Saúl Zapotecas-Martínez, Abel García-Nájera, and Adriana Menchaca-Méndez. Engineering applications of multi-objective evolutionary algorithms: A test suite of box-constrained real-world problems. *Engineering Applications of Artificial Intelligence*, 123:106192, 2023.

[12] Qingfu Zhang and Hui Li. MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. *IEEE Transactions on Evolutionary Computation*, 11(6):712–731, 2007.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA