GRADO EN INGENIERÍA DEL SOFTWARE

Diseño e implementación de una herramienta software para meta-optimización multiobjetivo

Design and implementation of a meta-optimization framework

Realizado por
Pablo Alonso Burgos

Tutorizado por
Antonio Jesús Nebro Urbaneja

Departamento
Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO 2025

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA DEL SOFTWARE

Diseño e implementación de una herramienta software para meta-optimización multiobjetivo

Design and implementation of a meta-optimization framework

Realizado por:

Pablo Alonso Burgos

Tutorizado por:

Antonio Jesús Nebro Urbaneja

Departamento:

Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO DE 2025

Resumen

La optimización multiobjetivo con algoritmos evolutivos es una disciplina que ha experimentado un gran auge en los últimos 25 años. Este auge se debe a la aparición de algoritmos no exactos como los algoritmos evolutivos, capaces de encontrar soluciones, si bien no necesariamente óptimas, sí próximas al óptimo de este tipo de problemas de forma rápida. En este contexto, una línea de investigación actual es el diseño y configuración de parámetros de los algoritmos evolutivos para problemas multi-objetivo de forma automática, y una aproximación a este tema es la meta-optimización, que consiste en definir dicho diseño como un problema de optimización que pueda ser resuelto por un algoritmo evolutivo multi-objetivo. En este contexto, en la Universidad de Málaga se ha desarrollado un software de meta-optimización llamado Evolver.

Este Trabajo de Fin de Grado está centrado en rediseñar y mejorar Evolver con el fin de suplir varias deficiencias de diseño y limitaciones del mismo. La idea consiste en incorporar a la herramienta Evolver una serie de mejoras tales como la generación automática de código para evitar la configuración manual mediante la especificación del espacio de parámetros de un algoritmo en *YAML*, el rediseño de algoritmos auto-configurables para permitir usar cualquier codificación y el desarrollo de una nueva codificación en árbol a la hora de trabajar con los distintos algoritmos. El resultado del trabajo fin de grado se evaluará usando el algoritmo NSGA-II, que es el algoritmo evolutivo de referencia en optimización multi-objetivo.

Palabras clave: Algoritmos evolutivos, Optimización multiobjetivo, Generación automática de código, Codificación en árbol, Evolver.

Abstract

Multi-objective optimization with evolutionary algorithms is a discipline that has experienced significant growth in the last 25 years. This growth is due to the emergence of non-exact algorithms, such as evolutionary algorithms, which are capable of finding solutions, though not necessarily optimal, close to the optimum for these types of problems quickly. In this context, a current line of research is the automatic design and parameter configuration of evolutionary algorithms for multi-objective problems, and one approach to this topic is meta-optimization, which involves defining this design as an optimization problem that can be solved by a multi-objective evolutionary algorithm. In this context, the University of Málaga has developed a meta-optimization software called Evolver.

This Bachelor's Thesis focuses on redesigning and improving Evolver to address several design deficiencies and limitations. The idea is to incorporate a series of improvements into the Evolver tool, such as automatic code generation to avoid manual configuration by specifying an algorithm's parameter space in *YAML*, the redesign of self-configurable algorithms to allow the use of any encoding, and the development of a new tree-based encoding when working with different algorithms. The results of the thesis will be evaluated using the NSGA-II algorithm, which is the reference evolutionary algorithm in multi-objective optimization.

Keywords: Evolutionary algorithms, Multi-objective optimization, Automatic code generation, Tree encoding, Evolver.

Índice

Resumen **5**

Abstract **7**

1. Introducción **15**

1.1. Motivación 15

1.2. Objetivos 15

1.3. Tecnologías usadas 16

1.4. Metodología empleada 17

1.5. Estructura del documento 18

2. Conceptos básicos **21**

2.1. Algoritmos evolutivos 21

2.2. Optimización Multi-Objetivo 22

2.3. El algoritmo evolutivo NSGA-II 23

2.4. Meta-optimización de metaheurísticas multiobjetivo 23

2.5. YAML 24

2.6. jMetal: Un framework para optimización multiobjetivo 26

3. Análisis de Evolver **31**

3.1. ¿Qué es Evolver? 31

3.2. Estructura de Evolver 32

3.3. Limitaciones de Evolver 33

4. Diseño e implementación de REvolver **37**

4.1. Arquitectura 37

4.2. Espacio de parámetros 37

4.3. Archivos YAML 39

4.4. Documentación de código 39

5. Generación automática de código en REvolver **41**

5.1. Problema Evolver original 41

5.2. Proceso de desarrollo 43

5.3. Desarrollo de la solución previa 44
5.4. Estructura de la solución definitiva 45
5.5. Desarrollo de la solución definitiva (DoubleSolution) 45
5.6. Desarrollo de la solución definitiva (PermutationSolution) 51

6. Implementación de una codificación en árbol 53
6.1. Motivación 53
6.2. Problema jMetal 54
6.3. Proceso de desarrollo 54
6.4. Arquitectura 55
6.5. Implementación de la clase YAMLProblem 56
6.6. Implementación de la clase YamlSolution 57
6.7. Definición de los nuevos operadores: mutation 58
6.8. Definición de los nuevos operadores: crossover 59
6.9. Adaptación de Evolver a la codificación en árbol 59

7. Evaluación de la nueva codificación en árbol 65
7.1. Metodología experimental 65
7.2. Métricas consideradas para la comparación de los algoritmos 67
7.3. Comparativa gráfica de los algoritmos 68
7.4. Comparativa de las mejores configuraciones encontradas 74
7.5. Comparativa de algoritmos empleando SAES 76

8. Conclusiones y Líneas Futuras 85
8.1. Conclusiones 85
8.2. Líneas Futuras 85

Referencias 87

Índice de figuras

1. Tablero Trello empleado 18
2. Funcionamiento de un algoritmo evolutivo. 21
3. Soluciones posibles del problema del coche. 22
4. Enfoque metaoptimizador de Evolver 24
5. Espacios de solución y decisión en el problema de metaoptimización 24
6. Diagrama de clases con la estructura de jMetal-core. 29
7. Componentes de Evolver Aldana-Martín et al. [2023] 32
8. Estructura de paquetes del proyecto Evolver. 33
9. Estructura de paquetes de REvolver. 37
10. Espacio completo de parámetros de REvolver. 38
11. Estructura de paquetes del generador de ConfigurableNSGAII 44
12. Estructura de paquetes para la generación de los parámetros de primer nivel de *EvNSGAIIDoubleProblem* 45
13. Estructura de paquetes de doubleproblem 46
14. Diagrama de clases con las clases implementadas en el contexto de la arquitecture de jMetal. 56
15. Interfaz generada por OutputYAMLResults para el la solución YAML. . . 61
16. Información en tiempo real de las evaluaciones del metaoptimizador 61
17. Evolución de las soluciones para los problemas DTLZ tras 1000 evaluaciones empleando EvNSGAII. 69
18. Evolución de las soluciones para el problema DTLZ1 a lo largo de las 1000 evaluaciones empleando Y-NSGAII-RC. 70
19. Evolución de las soluciones para el problema DTLZ1 a lo largo de las 1000 evaluaciones empleando Y-NSGAII-SBC. 70
20. Problema DTLZ1 en la evaluación 1000 con EvNSGAII 72
21. Problema DTLZ1 en la evaluación 1000 con Y-NSGAII-RC 72
22. Problema DTLZ1 en la evaluación 1000 con Y-NSGAII-SBC 73
23. Comparación de las soluciones iniciales de EVNGSAII, Y-NSGAII-RC y Y-NSGAII.SBC. 73
24. Comparación de las soluciones finales de EVNGSAII, Y-NSGAII-RC y Y-NSGAII.SBC. 74

25. Critical distance ranking 78
26. Boxplot para DTLZ1 79
27. Boxplot para DTLZ2 80
28. Boxplot para DTLZ3 81
29. Boxplot para DTLZ4 81
30. Boxplot para DTLZ5 82
31. Boxplot para DTLZ6 83
32. Boxplot para DTLZ7 83

1. Introducción

1.1. Motivación

La optimización multiobjetivo con algoritmos evolutivos es una disciplina que ha experimentado un gran auge en los últimos 25 años Deb [2001]Coello Coello et al. [2007]. Al margen del interés de optimizar problemas con varios objetivos contrapuestos, debido a su aplicabilidad en infinidad de problemas del mundo real, este auge se debe a la aparición de algoritmos no exactos como las metaheurísticas, que son capaces de encontrar soluciones, si bien no necesariamente óptimas, sí próximas al óptimo de este tipo de problemas de forma rápida Blum and Roli [2003]. Las metaheurísticas son técnicas estocásticas (tienen componentes internos que se basan en números generados aleatoriamente) que tienen, al igual que ocurre con otros algoritmos de la Inteligencia Artificial como los algoritmos de *clústering* y de clasificación en aprendizaje automático, una serie de parámetros que hay que ajustar de forma precisa para que la búsqueda de soluciones para un conjunto de problemas dado sea eficaz. Dentro de las metaheurísticas, los algoritmos evolutivos son la subfamilia de técnicas más ampliamente conocida y usada.

En este contexto, una línea de investigación actual es el diseño e implementación de herramientas que permitan ajustar los parámetros de algoritmos evolutivos para problemas multiobjetivo de forma automática, y una aproximación a este tema es la **meta-optimización**, que consiste en definir la búsqueda de configuraciones como un problema de optimización que pueda ser resuelto por una metaheurística multiobjetivo. En la Universidad de Málaga ya se ha investigado en ese tema y se ha desarrollado un software de meta-optimización llamado *Evolver* (`https://github.com/jMetal/Evolver`). Sin embargo, este software tiene un diseño deficiente, lo que lo hace muy difícil de usar por parte de los usuarios, ya que requiere la escritura manual de métodos complejos que podrían generarse de forma automática, presenta varias limitaciones en cuanto a los tipos de problemas que puede resolver y se basa en una codificación que potencialmente es ineficiente.

1.2. Objetivos

El objetivo principal de este TFG es diseñar e implementar una herramienta de meta-optimización, inspirada en *Evolver*, pero que resuelva sus limitaciones. En concreto, se

plantea abordar los siguientes aspectos:

1. Se deben poder contemplar problemas de optimización con diversas codificaciones (*Evolver* sólo admite soluciones continuas).
2. A partir de la descripción del espacio de parámetros en un fichero *YAML*, se ha de generar de forma automática el código de una metaheurística auto configurable que pueda ser directamente usable por un usuario. Se trabajará en este aspecto con el algoritmo *NSGA-II*, que es el estándar en el área.
3. Implementar una codificación alternativa a la usada en *Evolver*, que está basada en codificar el espacio de parámetros de una metaheurística como un array de valores reales en el rango [0,0, 1,0], en un esquema más eficiente basado en árbol de parámetros. Esta tarea requiere la definición e implementación de operadores específicos de cruce y mutación de soluciones.
4. Revisar todos los componentes de *Evolver* para plantear diseños alternativos que mejoren su legibilidad, usabilidad y extensibilidad.
5. Mejorar la calidad del código incluyendo documentación.

1.3. Tecnologías usadas

Para el desarrollo de este trabajo se han utilizado las siguientes tecnologías y herramientas:

- Lenguajes de programación: Java
- Entorno de desarrollo: IntelliJ IDEA
- Frameworks: jMetal y Evolver
- Formatos de archivo: YAML y Texto
- Gestión del proyecto: Trello
- Sistema de control de versiones: GitHub

1.4. Metodología empleada

La metodología seguida a lo largo del proyecto ha sido *Kanban*. Se intentó inicialmente aplicar *Scrum*, pero se abandonó esta metodología al tratarse de un equipo únicamente compuesto por mí y no poderse aplicar correctamente.

Se ha implementado un flujo de trabajo **Kanban** en Trello, con una estructura minimalista compuesta por tres columnas principales:

- *Lista de tareas*: Tareas pendientes de iniciar.
- *Hecho*: Tareas finalizadas (pero aún no validadas).
- *Completado*: Tareas revisadas y cerradas.

El flujo lineal de una determinada tarea sigue el siguiente ciclo:

$$\text{Lista de tareas} \rightarrow \text{Hecho} \rightarrow \text{Completado}$$

De esta manera, se hace muy sencillo el seguimiento del proyecto por parte del tutor y para mí a la hora de ser consciente de lo hecho y lo que queda por hacer. Gracias a la facilidad que nos proporciona *Trello* a la hora de añadir y gestionar tareas en el tablero, no se ha perdido el rumbo del proyecto en ningún momento.

- *Sencillez*: Fácil de entender para equipos nuevos en Kanban.
- *Enfoque en finalización*: La columna *Completado* evita reabrir tareas innecesariamente.
- *Visualización clara*: Trello permite arrastrar tarjetas entre columnas (*drag-and-drop*).

A continuación, se muestra el tablero *Trello* empleado:

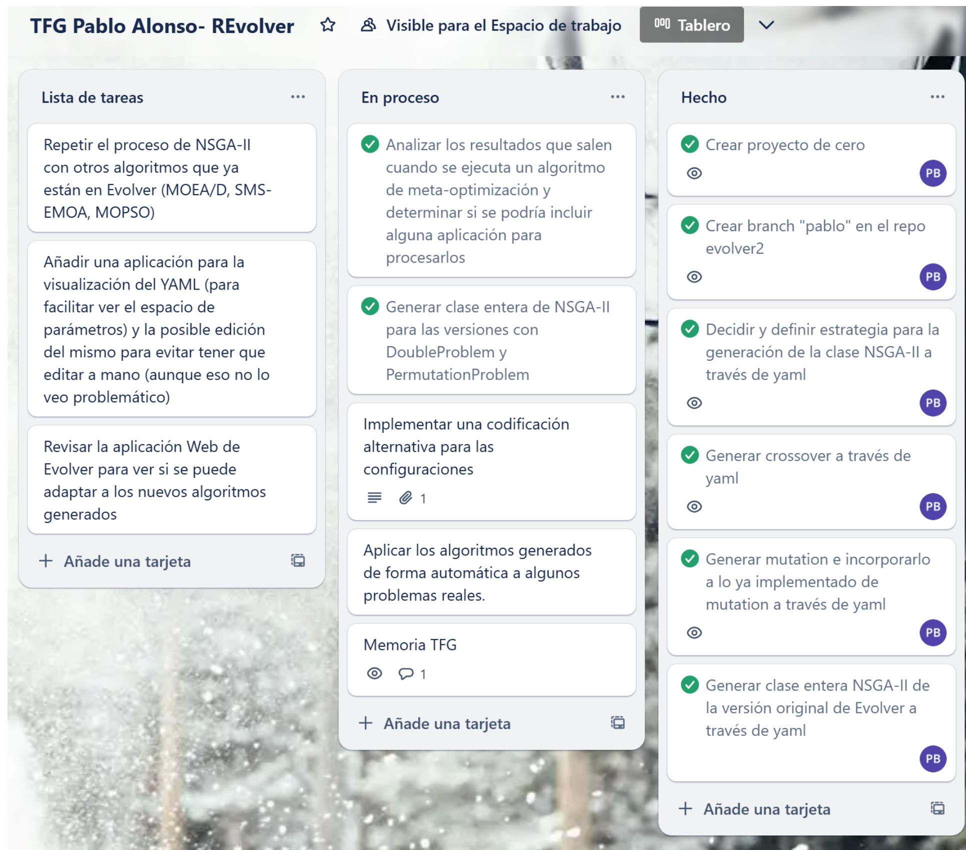

Figura 1: Tablero Trello empleado

1.5. Estructura del documento

Esta memoria está organizada de la siguiente manera:

- En el capítulo 2 se presenta el contexto teórico sobre conceptos destacados y de revelancia en este Trabajo de Fin de Grado.
- El capítulo 3 describe la herramienta *Evolver* original y su arquitectura. Además, se incluye un análisis detallado de las limitaciones de *Evolver*.
- El capítulo 4 presenta la nueva implementación *REvolver* con las mejoras realizadas.
- El capítulo 5 expone y explica cómo se ha elaborado la generación automática de código.
- El capítulo 6 trata lo relativo a la creación e implementación de la codificación en árbol.

- El capítulo 7, está destinado a evaluar la nuevas codificación en árbol y enfrentarla con la ya existente.
- Finalmente en el capítulo 8 se abordarán las conclusiones y líneas futuras de trabajo.

2. Conceptos básicos

En este capítulo se presentan los principales conceptos en los que se sustenta el Trabajo Fin de Grado realizado.

2.1. Algoritmos evolutivos

Los algoritmos evolutivos pertenecen a una conocida familia de técnicas de optimización denominada metaheurísticas. Estos tipos de algoritmos destacan porque se inspiran en la selección natural, donde los individuos más aptos sobreviven y se reproducen, mientras que los menos aptos desaparecen.

El funcionamiento de un algoritmo evolutivo genérico se muestra en la figura 2. Se parte de una población inicial de individuos, que están compuestos por:

- Un *cromosoma*: valores que representan una solución del problema
- Un *fitness*: valor que indica la calidad de la solución

El fitness de un individuo se evalúa aplicando la formulación del problema de optimización a la solución que representa el cromosoma.

El algoritmo evolutivo tiene un bucle que se ejecuta de forma repetida hasta que se cumpla una condición de terminación. En el bucle se llevan a cabo una serie de etapas: se seleccionan individuos de la población para emparejarlos, se aplican operadores de variación (típicamente, cruce y mutación) sobre las parejas para generar una población de descendientes, que es evaluada, y se reemplaza (actualiza) la población actual con los descendientes.

Algorithm 1 Pseudo-code of an evolutionary algorithm

```
1:  P(0) ← GenerateInitialSolutions()
2:  t ← 0
3:  Evaluate(P(0))
4:  while not StoppingCriterion() do
5:      P′(t) ← Selection(P(t))
6:      Q(t) ← Variation(P(t′))
7:      Evaluate(Q(t))
8:      P(t + 1) ← Update(P(t), Q(t))
9:      t ← t + 1
10: end while
```

Figura 2: Funcionamiento de un algoritmo evolutivo.

2.2. Optimización Multi-Objetivo

Los problemas de optimización multiobjetivo consisten en maximizar o minimizar varias funciones u objetivos a la vez Zhou et al. [2011]. Además, dichos objetivos suelen ser contrapuestos, es decir, mejorar uno de ellos implica empeorar el resto. Es por ello que no hay una única solución, sino que existe un conjunto de soluciones que representan los mejores compromisos posibles entre los diferentes objetivos.

Pongamos un ejemplo sencillo. Se quiere ir a Madrid en coche. Son 535 km. Como todo problema de optimización multi-objetivo, el problema se caracteriza por objetivos, variables y restricciones: Objetivo 1: Minimizar el tiempo de viaje. Objetivo 2: Minimizar el consumo de gasolina. Restricciones: Velocidad mínima: 60 km/h. Velocidad máxima: 120 km/h. Variables de decisión: la media de velocidad del vehículo.

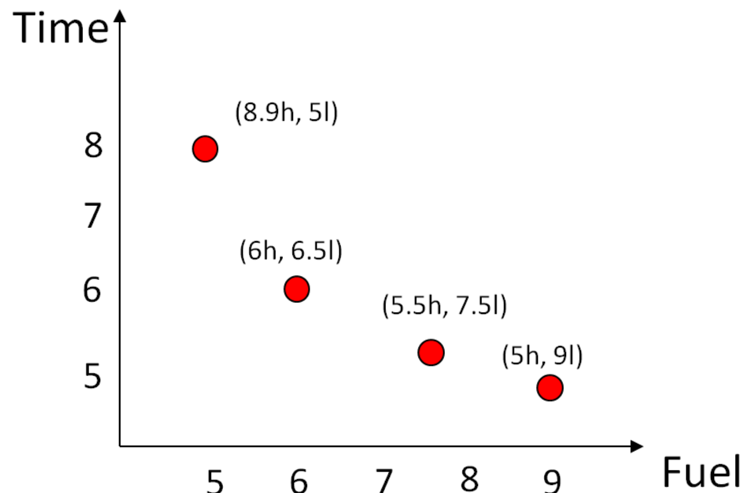

Figura 3: Soluciones posibles del problema del coche.

Dado que obtenemos un conjunto de soluciones, hemos de decidir cuál nos conviene más. Si el tiempo es nuestra prioridad, escogeríamos la solución de (5h, 9l). Si el consumo es lo que más nos importa, escogeríamos la solución de (8h, 6l). Sin embargo, estas soluciones representan las soluciones extremas; el resto son soluciones intermedias que buscan alcanzar un término medio entre los dos objetivos.

En optimización multi-objetivo se usa el concepto de dominancia de Pareto. Se dice que una solución domina a otra cuando es mejor o igual en todos los objetivos. Cuando una solución es mejor que otra en algunos objetivos y peor en otros, se dice que ambas soluciones son no dominadas.

Usando dominancia de Pareto, el óptimo de un problema de optimización multi-objetivo está compuesto por aquél conjunto de soluciones de todo el espacio de búsqueda que son no dominadas entre sí y que no son dominadas por ninguna otra.

2.3. El algoritmo evolutivo NSGA-II

NSGA-II es uno de los algoritmos evolutivos multi-objetivo más usados. Es bastante conocido desde su aparición a comienzos del año 2000 porque es bastante robusto, que significa que funciona razonablemente bien con muchos problemas en su configuración por defecto Deb et al. [2002].

Al ser un algoritmo evolutivo. se basa en el algoritmo de la figura 2, pero para poder resolver problemas multi-objetivo aplica dos técnicas:

- Un algoritmo para ordenar las soluciones según el concepto de dominancia de Pareto.
- Un estimador de densidad denominado (*crowding distance*) que usa en para seleccionar las soluciones que están en regiones menos pobladas.

2.4. Meta-optimización de metaheurísticas multiobjetivo

Los algoritmos evolutivos son muy sensibles a los parámetros de control que los caracterizan, por lo que su eficacia depende del ajuste concreto de estos parámetros cuando se aplica a un problema o conjunto de problemas determinado.

Este ajuste se ha hecho tradicionalmente de forma manual, lo que es tedioso y poco riguroso. En los últimos años se está investigando en el uso de técnicas de configuración automática de algoritmos evolutivos (y de metaheurísticas en general). Una estrategia prometedora es la meta-optimización, que consiste en formular la búsqueda de configuraciones de parámetros de un algoritmo evolutivo como un problema de optimización multi-objetivo que se optimiza con un algoritmo evolutivo. Es en este contexto en el que se desarrolló la herramienta Evolver. El enfoque de metaoptimización de Evolver sigue el esquema mostrado en la Figura 4 . El objetivo es encontrar la mejor configuración de una metaheurística multiobjetivo de nivel base para resolver eficientemente un conjunto de problemas de nivel base, al que llamamos conjunto de entrenamiento. Este objetivo se formula como un problema de metaoptimización, donde el espacio de soluciones está compuesto por configuraciones del algoritmo para el optimizador de nivel base, y el espacio

objetivo está definido por un conjunto de indicadores de calidad que deben minimizarse, como se ilustra en la Figura 5.

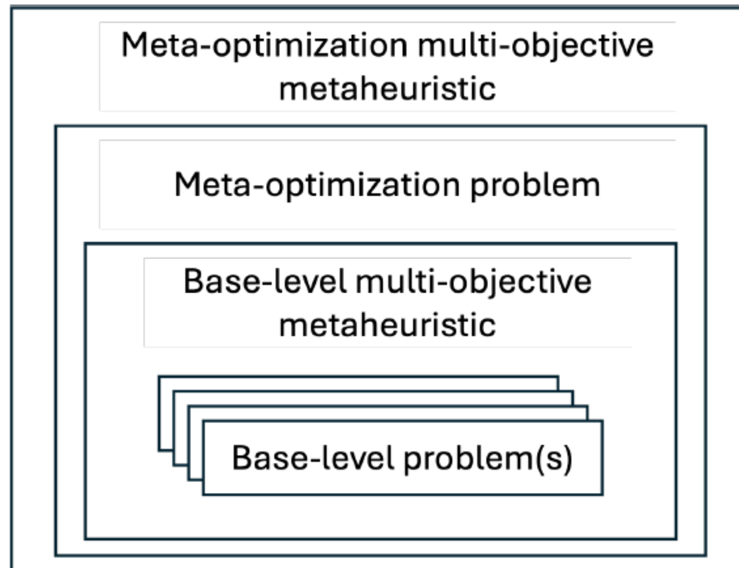

Figura 4: Enfoque metaoptimizador de Evolver

Figura 5: Espacios de solución y decisión en el problema de metaoptimización

2.5. YAML

En el panorama actual del desarrollo de software, *YAML* (YAML Ain't Markup Language) ha emergido como el lenguaje silencioso que alimenta la infraestructura digital moderna. Su ascenso no fue accidental: combina la legibilidad de un documento de texto con la precisión que requieren los sistemas automatizados.

Nacido en 2001 como alternativa a XML, *YAML* adoptó una filosofía muy concreta: la configuración debería ser intuitiva para quienes la escriben y robusta para las máquinas que la interpretan. Esto se materializa en su sintaxis limpia, donde dos espacios reemplazan llaves anidadas, y donde los comentarios (con #) permiten documentar cada parámetro si así se desea.

```yaml
version: '3.8'

services:
  webapp:
    image: nginx:latest
    container_name: my_nginx
    ports:
      - "8080:80"
    volumes:
      - ./html:/usr/share/nginx/html
    environment:
      - NGINX_ENV=production
    restart: unless-stopped

  database:
    image: postgres:13
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: securepassword
      POSTGRES_DB: app_db
    volumes:
      - pg_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"

volumes:
  pg_data:
```

Listing 1: Ejemplo YAML docker-compose

Lo que comenzó como formato alternativo hoy impulsa gigantes tecnológicos. Kubernetes lo adoptó como lingua franca para definir recursos, Docker lo utiliza para orquestar contenedores en docker-compose.yml, y herramientas como Ansible lo transforman en scripts de automatización. Su secreto reside en tres virtudes fundamentales:

- Jerarquía visual: La indentación crea mapas mentales inmediatos, donde la estructura se deduce de la alineación.
- Flexibilidad tipográfica: Soporta desde números y booleanos hasta listas multilínea con preservación de saltos.
- Ecosistema universal: Bibliotecas nativas en Python, Ruby, Java y JavaScript garantizan interoperabilidad.

Pero *YAML* trasciende lo técnico. En equipos DevOps, se ha convertido en un lenguaje común que unifica desarrolladores y sysadmins. Un archivo *YAML* bien estructurado funciona simultáneamente como:

- Documentación ejecutable.
- Registro histórico de cambios.

- Contrato entre componentes del sistema.

Los desafíos no son menores. La sintaxis permisiva puede generar ambigüedades, y la falta de variables nativas complica la reutilización de código. Herramientas como Helm para Kubernetes introducen plantillas para superar estas limitaciones, demostrando la adaptabilidad del ecosistema.

En proyectos modernos, un archivo *YAML* típico recorre un ciclo de vida completo: desde el portátil del desarrollador hasta los pipelines de CI/CD, pasando por revisiones de equipo y auditorías de seguridad. Esta transversalidad lo ha convertido en el formato de configuración por defecto para la era de la nube, donde la agilidad y la claridad no son opcionales.

2.6. jMetal: Un framework para optimización multiobjetivo

jMetal se ha consolidado como uno de los frameworks más completos y versátiles en el ámbito de la optimización multiobjetivo mediante metaheurísticas Durillo and Nebro [2011]. Desarrollado en Java como proyecto open source, esta herramienta se ha convertido en referencia tanto para investigadores como para profesionales que trabajan en problemas de optimización complejos.

El proyecto, que inició su desarrollo en 2006, ha evolucionado significativamente hasta su versión actual 6.7 (disponible públicamente en `https://github.com/jMetal/jMetal`), con una versión 6.8-SNAPSHOT en constante desarrollo que incorpora las últimas mejoras y algoritmos. Esta evolución ha sido documentada en varias publicaciones científicas clave que han marcado hitos en su desarrollo:

- *jMetal: A Java framework for multi-objective optimization*" sentó las bases iniciales del framework.
- *Redesigning the jMetal Multi-Objective Optimization Framework*" presentó una importante reestructuración arquitectónica.
- *Automatic configuration of NSGA-II with jMetal and irace*" introdujo capacidades avanzadas de configuración automática.
- *Evolving a Multi-objective Optimization Framework*" documentó las últimas innovaciones.

jMetal proporciona un amplio espectro de algoritmos implementados. En el ámbito multiobjetivo, incluye desde clásicos como NSGA-II y SPEA2 hasta variantes más recientes como NSGA-III y MOEA/D-STM.

Para problemas uniobjetivo, jMetal ofrece igualmente un conjunto robusto de metaheurísticas, incluyendo versiones modificables de algoritmos genéticos, estrategias evolutivas y diferentes variantes de PSO (Particle Swarm Optimization). Particularmente interesante es su soporte para modelos de computación paralela, que permite ejecutar algoritmos tanto en modo síncrono (usando multihilo o Apache Spark) como asíncrono.

La flexibilidad de jMetal se manifiesta en su soporte para múltiples esquemas de representación:

- *Binaria*: Ideal para problemas combinatorios.
- *Real*: Para espacios de búsqueda continuos.
- *Entera*: Optimización discreta.
- *Permutación*: Problemas de ordenamiento.

Esta versatilidad permite abordar desde problemas académicos clásicos hasta aplicaciones industriales complejas.

Además, jMetal incluye una extensa colección de problemas benchmark organizados en categorías:

- *Familias estándar*: ZDT, DTLZ, WFG para evaluación comparativa.
- *Problemas con restricciones*: Como los conjuntos de Srinivas y Tanaka.
- *Problemas combinatorios*: Incluyendo TSP multiobjetivo.
- *Casos académicos*: OneMax, OneZeroMax para validación.

Para medir la calidad de las soluciones, implementa indicadores como:

- Hipervolumen (y su versión normalizada).
- Spread (diversidad de soluciones).
- Distancia generacional (GD).

- Distancia generacional inversa (IGD+).
- Épsilon aditivo.

Más allá de los algoritmos básicos, jMetal ofrece funcionalidades que lo distinguen:

- *Automatización experimental*: Configuración sistemática de estudios.
- *Diseño automático*: Generación de metaheurísticas.
- *Análisis estadístico*: Evaluación rigurosa de resultados.
- *Interoperabilidad*: Integración con otras herramientas como irace.

Este conjunto de características hace de jMetal una plataforma ideal tanto para investigación académica como para desarrollo de soluciones industriales, manteniendo un equilibrio entre rendimiento computacional y flexibilidad conceptual. Su arquitectura modular permite, además, extender sus capacidades para abordar nuevos desafíos en el campo de la optimización.

El núcleo de jMetal está compuesto por tres elementos. Dentro del paquete jMetal-core, encontramos *Problem*, *Solution* y *Operator*, donde en este último destacan crossover y mutation.

1. *Problem*:Define el problema de optimización mediante:
   - *Espacio de búsqueda*: Límites para cada variable.
   - *Funciones objetivo*: Métricas a optimizar.
   - *Restricciones* (opcional): Condiciones adicionales.
2. *Solution* Representación de una solución candidata:
   - *Variables*: Valores de decisión.
   - *Objetivos*: Resultados evaluados.
   - *Métodos*: Para acceso/modificación.
3. *Operator* Modifican soluciones durante la optimización:

   Por un lado destaca el *Crossover*. Combina soluciones padres:

$$\text{Hijo} = \text{Crossover}(\text{Parent}_1, \text{Parent}_2) \tag{1}$$

A su vez, la *Mutation* también adquiere importancia. Aplica cambios aleatorios a la solución:

$$\text{MutatedSolution} = \text{Mutate}(\text{Solution}) \quad (2)$$

La estructura que siguen estas componentes clave fue rediseñada en la versión *jMetal 5.0* Nebro et al. [2015]:

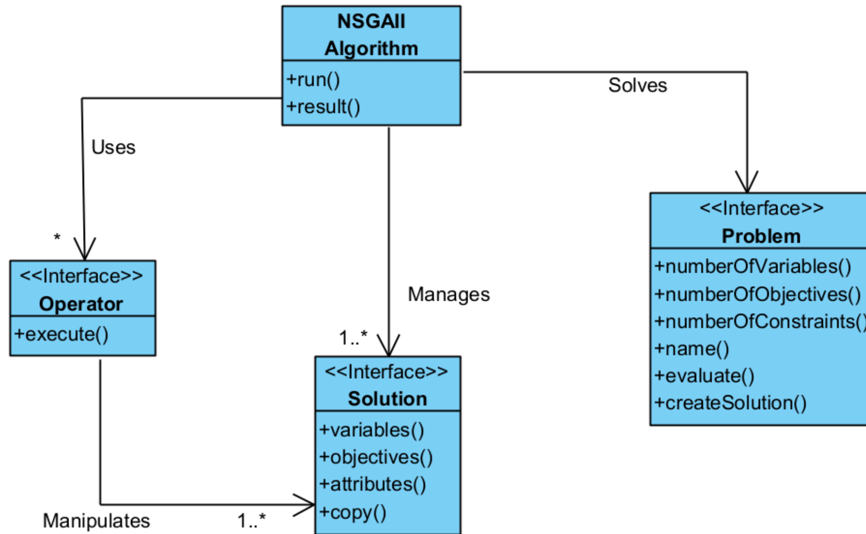

Figura 6: Diagrama de clases con la estructura de jMetal-core.

Si quisiésemos trabajar con una codificación de arrays de dobles, se crearía un *DoubleProblem*, que extendería a la interfaz *Problem*, un *DoubleSolution*, que también extendería a la interfaz *Solution* y los operadores correspondientes de *Crossover* y *Mutation*.

Se aplicaría este mismo patrón para cualquier tipo de codificación que se quisiese implantar.

3. Análisis de Evolver

Antes de comenzar a explicar lo que ha sido realizado en este TFG, se va a dedicar una sección a aclarar brevemente en qué consiste Evolver así como su estructura. Finalmente se presentarán las limitaciones y puntos débiles de Evolver, los cuales juntos representan el objeto de este TFG.

3.1. ¿Qué es Evolver?

Evolver es una herramienta software diseñada para la configuración automática de metaheurísticas multiobjetivo, que consiste en, dado un conjunto de problemas (denominado *conjunto de entrenamiento*), buscar la mejor configuración de un algoritmo para optimizarlos en conjunto de forma eficiente.

Su enfoque principal es la meta-optimización, donde se lleva a cabo un proceso de ajuste de parámetros de un algoritmo evolutivo y se plantea como un problema multiobjetivo que se resuelve mediante un optimizador multiobjetivo. En este problema, la codificación de las variables representa una configuración particular del algoritmo, y evaluar una solución implica ejecutar el algoritmo evolutivo bajo dicha configuración; el frente de soluciones resultante se evalúa mediante una combinación de indicadores de calidad, que constituyen las funciones objetivo del problema multiobjetivo resultante.

Por tanto, las bases de *Evolver* son:

- *Algoritmo configurable*: Algoritmo metaheurístico multiobjetivo que, dado un conjunto de entrenamiento, busca una configuración óptima.
- *Espacio de diseño*: Define los parámetros y componentes configurables del algoritmo.
- *Indicadores de calidad*: Objetivos a minimizar al aplicar el algoritmo configurable.
- *Meta-optimizador*: Algoritmo que resuelve el problema de optimización definido por los indicadores de calidad.

La figura **??** incluye un diagrama que muestra estos componentes y cómo se relacionan entre sí.

El proyecto *Evolver* consta de dos partes:

1. *Evolver* (biblioteca Java):

- Implementa el enfoque de meta-optimización
- Incluye algoritmo configurable, espacio de diseño, indicadores de calidad y metaoptimizador
- Construido con Maven

2. *Evolver Dashboard* (Python):

- Interfaz web para configurar y ejecutar *Evolver*
- Desarrollado con Streamlit

Evolver está descrito en el artículo *Evolver: Meta-optimizing multi-objective metaheuristics* Aldana-Martín et al. [2023], publicado en la revista *SoftwareX* `https://www.sciencedirect.com/science/article/pii/S2352711023002479`.

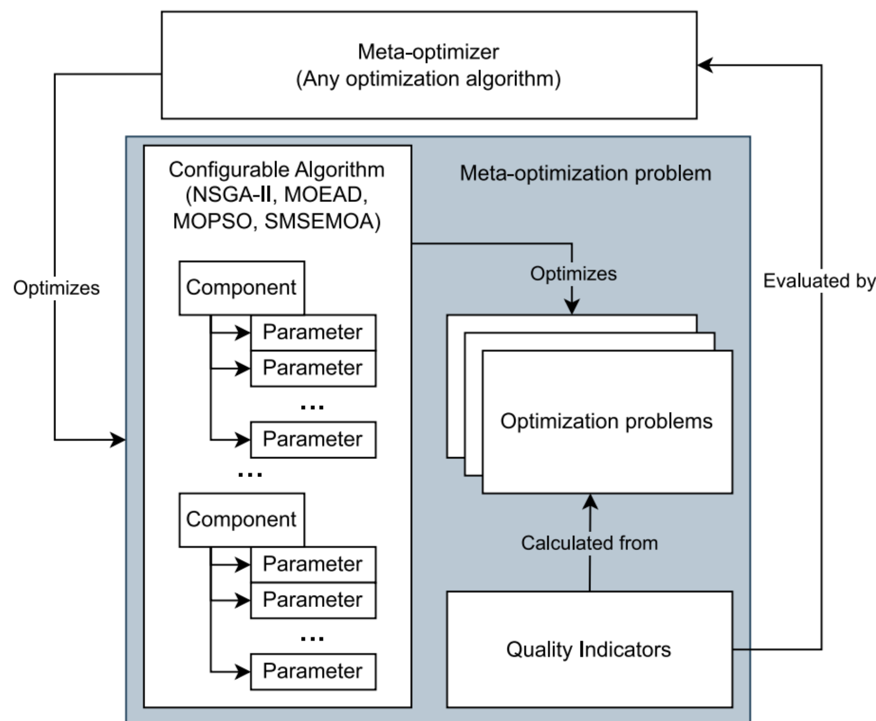

Figura 7: Componentes de Evolver Aldana-Martín et al. [2023]

3.2. Estructura de Evolver

La estructura de *Evolver* se muestra en la figura 8. Los paquetes que lo componen son:

component/ Elementos reutilizables de algoritmos

configurablealgorithm/ Interfaces para algoritmos configurables. Implementaciones de algoritmos usando *YAMLs*

examples/ Ejemplos de meta-optimización.

factory/ Crea instancias de algoritmos.

parameter/ Define las interfaces e implementaciones de los parámetros que componen los algoritmos evolutivos.

parameterdescriptiongenerator/ Genera descripciones de los espacios de parámetros tanto en formato textual como *YAML*.

problem/ Clases que definen e implementan un problema de meta-optimización.

referencefrontupdate/ Paquete experimental.

util/ Funciones auxiliares y utilidades.

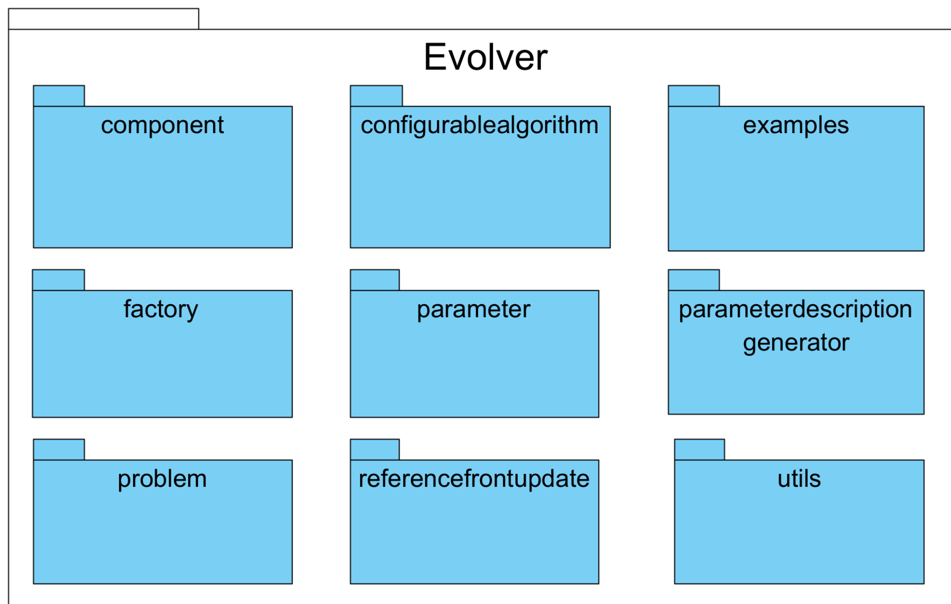

Figura 8: Estructura de paquetes del proyecto Evolver.

3.3. Limitaciones de Evolver

Evolver presenta varias limitaciones y áreas de mejora que se van a detallar a continuación.

En primer lugar, la estructura del proyecto es muy confusa y poco intuitiva, en parte porque no hay una separación clara de las partes que componen lo que es la funcionalidad de meta-optimización en sí con la del dashboard. En este sentido, el paquete *factory* sólo se usa en el dashboard.

Por otro lado, el paquete *problem* incluye tres cases para implementar el problema de meta-optimización, cuando realmente sólo hace falta uno. Además, esas clases contienen métodos muy largos que son difíciles de entender.

En segundo lugar, Evolver sólo permite usar meta-optimización cuando los problemas del conjunto de entrenamiento son continuos, lo que es una limitación importante, dado que no se puede usar para problemas combinatorios.

En tercer lugar, los parámetros de los algoritmos configurables están codificados de forma explícita, por lo que cualquier pequeño cambio requiere modificar su código. Sin embargo, el espacio de parámetros se obtiene en formato *YAML* a partir de un generador, por lo que se podría pensar en generar los algoritmos de forma automática a partir de dicha descripción de parámetros.

En cuarto lugar, la codificación del problema de meta-optimización es continua, de forma que todos los parámetros se codifican en un array de elementos (uno por parámetro) de valores entre 0.0 y 1.0, lo que requiere una posterior decodificación a los valores originales (hay que indicar que hay parámetros enteros, reales y categóricos). Eso se hace así porque es un esquema muy fácil de implementar y permite además que cualquier algoritmo evolutivo multi-objetivo pueda ser usado como meta-optimizador.

Sin embargo, esta codificación presenta dos problemas potenciales. Suponamos un parámetro categórico con dos categorías. Al codificarlo en el intervalo $[0{,}0, 1{,}0]$, cualquier valor menor que 0.5 representa a la primera categoría y uno mayor a la segunda, por lo que al aplicar un operador de mutación al valor actual del parámetro puede ocurrir que el cambio no tenga efecto al decodificar (por ejemplo, si el valor actual es 0.2 y se muta a 0.4, siendo representando la primera categoría). Otro inconveniente es que todos los parámetros se codifican en el array, pero hay parámetros que dependen de otros que tienen determinados valores; en este caso, se puede mutar un parámetro que no se usa en la configuración en la que está.

Por último, el proyecto no posee una documentación clara ni ordenada y, en algunos casos, inexistente.

Estas limitaciones constituyen la motivación principal para el desarrollo de este TFG, cuyo objetivo es superar estas deficiencias mediante una nueva implementación.

4. Diseño e implementación de REvolver

En esta sección se detalla el diseño y aspectos de implementación relevantes de *REvolver*, el nuevo proyecto resultante de ampliar las funcionalidades de *Evolver* cumpliendo los objetivos propuestos del TFG, así como resolviendo las limitaciones y mejoras observadas.

4.1. Arquitectura

La arquitectura de *REvolver* mejora notablemente a la de *Evolver*. Se han quitado paquetes y clases inutilizadas así como sustituido nombres de clases poco claros por otros más acordes, explicativos y coherentes. La nueva arquitectura diseñada es la siguiente:

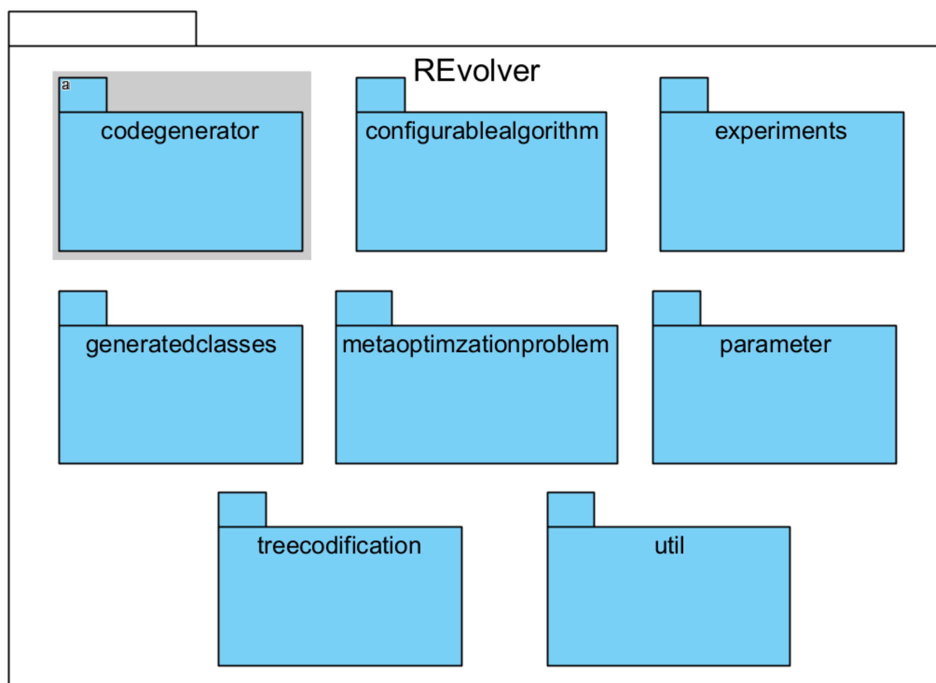

Figura 9: Estructura de paquetes de REvolver.

4.2. Espacio de parámetros

El espacio de parámetros se asemeja bastante al de *Evolver*.

Poseemos cinco parámetros de primer nivel:

- *algorithmResult*
- *createInitialSolutions*
- *offSpringPopulationSize*

- *variation*
- *selection*

Cada parámetro de primer nivel, a su vez, posee una serie de subparámetros. Todos estos últimos son fácilmente manipulables ya que se describen a través de un archivo *YAML*. Así se explica en la sección 4.3 El espacio de completo de parámetros en *REvolver* es el siguiente:

```
algorithmResult:
  type: categorical
  values:
    - population:
    - externalArchive:
        specific_parameter:
          - populationSizeWithArchive:
              type: integer
              values: [10, 200]
          - externalArchive:
              type: categorical
              values:
                - crowdingDistanceArchive:
                - unboundedArchive:

createInitialSolutions:
  type: categorical
  values: [random:, latinHypercubeSampling:, scatterSearch:]

offspringPopulationSize:
  type: categorical
  values: [1, 2, 5, 10, 20, 50, 100, 200, 400]

selection:
  type: categorical
  values:
    - tournament:
        specific_parameter:
          - selectionTournamentSize:
              type: integer
              values: [2, 10]
    - random:
```

```
variation:
  type: categorical
  values:
    - crossoverAndMutationVariation:
        specific_parameter:
          - crossover:
              type: categorical
              global_parameters:
                - crossoverProbability:
                    type: real
                    values: [0.0, 1.0]
                - crossoverRepairStrategy:
                    type: categorical
                    values: [random:, round:, bounds:]
              values:
                - SBX:
                    specific_parameter:
                      - sbxDistributionIndex:
                          type: real
                          values: [5.0, 400.0]
                - BLX_ALPHA:
                    specific_parameter:
                      - blxAlphaCrossoverAlphaValue:
                          type: real
                          values: [0.0, 1.0]
                - wholeArithmetic:
          - mutation:
              type: categorical
              global_parameters:
                - mutationProbabilityFactor:
                    type: real
                    values: [0.0, 2.0]
                - mutationRepairStrategy:
                    type: categorical
                    values: [random:, round:, bounds:]
              values:
                - uniform:
                    specific_parameter:
                      - uniformMutationPerturbation:
                          type: real
                          values: [0.0, 1.0]
                - polynomial:
                    specific_parameter:
                      - polynomialMutationDistributionIndex:
                          type: real
                          values: [5.0, 400.0]
                - linkedPolynomial:
                    specific_parameter:
                      -
      linkedPolynomialMutationDistributionIndex:
                          type: real
                          values: [5.0, 400.0]
                - nonUniform:
                    specific_parameter:
                      - nonUniformMutationPerturbation:
                          type: real
                          values: [0.0, 1.0]
```

Figura 10: Espacio completo de parámetros de REvolver.

Si no se desea contar con un parámetro en completo, simplemente tendríamos que borrarlo del archivo *YAML*.

4.3. Archivos YAML

Los archivos tienen un valor esencial en *REvolver*. En este proyecto, toda funcionalidad nueva incorporada ha sido realizada empleando archivos *YAML* como base.

Se ha optado por emplear los archivos *YAML* debido a:

- Notación sencilla.
- Representación visual de jerarquías de parámetros.
- Facilidad para modificaciones.

Además, los archivos *YAML* permiten modificar fácilmente el espacio de parámetros, algo fundamental en este TFG.

Se valoró emplear otro tipo de archivos, como el *JSON*, pero el carácter jerárquico de los *YAML* fue diferenciador en la elección.

4.4. Documentación de código

Toda herramienta software debe estar debidamente documentada, algo que le faltaba a *Evolver*. Por ello, se ha dotado al proyecto de una documentación correcta y adecuada. Se ha decidido elaborar toda la documentación en inglés y sigue un formato del estilo *Javadoc*.

5. Generación automática de código en REvolver

Uno de los objetivos de este TFG es la generación automática, o dinámica, del código de los algoritmos configurables, en particular de NSGA-II. El funcionamiento es simple: se describen en un archivo *YAML* aquellos parámetros con los que queremos contar en el algoritmo y se generará una clase nueva teniendo únicamente los parámetros y límites detallados en el archivo YAML.

5.1. Problema Evolver original

Tal y como se mencionó en la sección 3.3, Evolver tiene un gran inconveniente. Si se quería prescindir o contar con algún parámetro concreto, esto debía hacerse a mano, es decir, modificando el código. Esta opción no es válida, ya que hay usuarios de Evolver que no saben programar, además de no ser para nada práctico.

Veamos esto con un ejemplo. Imaginemos que, para el cruce (*crossover*) de dos soluciones, no queremos contar con el parámetro *SBX*. Para ello, hasta ahora, habría que borrar *SBX* a mano. Esto puede llegar a ser un poco tedioso, ya que, en muchos casos, hay trozos de código que dependen de ciertos parámetros y, por tanto, si dicho parámetro no aparece, ese trozo de código también debe desaparecer.

Para quitar lo relativo a *SBX* habría que editar la línea 2 y eliminar las líneas 12 y 13:

```
1 private void variation() {
2     CrossoverParameter crossoverParameter = new CrossoverParameter(
3         List.of("SBX", "BLX_ALPHA", "wholeArithmetic"));
4     ProbabilityParameter crossoverProbability =
5         new ProbabilityParameter("crossoverProbability");
6     crossoverParameter.addGlobalParameter(crossoverProbability);
7     RepairDoubleSolutionStrategyParameter crossoverRepairStrategy =
8         new RepairDoubleSolutionStrategyParameter(
9             "crossoverRepairStrategy", Arrays.asList("random", "round", "bounds"));
10    crossoverParameter.addGlobalParameter(crossoverRepairStrategy);
11
12    RealParameter distributionIndex = new RealParameter("sbxDistributionIndex",5.0,400.0);
13    crossoverParameter.addSpecificParameter("SBX", distributionIndex);
14
15    RealParameter alpha = new RealParameter("blxAlphaCrossoverAlphaValue",0.0,1.0);
16    crossoverParameter.addSpecificParameter("BLX_ALPHA", alpha);
17    //resto del código...
```

Fragmento de código 1: Crossover del método variation() de la clase ConfigurableNSGAII

Asimismo, si se quisiese modificar el *upperBound* de *BLX_ALPHA* (línea 15), también

habría que hacerlo todo a mano, algo que funciona, pero que es poco práctico si se va a estar constantemente tocando y variando parámetros.

Por ello, una solución a este problema es partir de la definición de un archivo *YAML* que contenga el espacio entero de parámetros y modificarlo según se desee. Aplicando este enfoque se podrá escoger de manera rápida, sencilla e intuitiva los parámetros y valores límite (*lowerBound* y *upperBound*) con los que queremos contar.

```
1         crossover:
2           type: categorical
3           global_parameters:
4             crossoverProbability:
5               type: real
6               values: [ 0.0, 1.0 ]
7             crossoverRepairStrategy:
8               type: categorical
9               values:
10                random:
11                round:
12                bounds:
13          values:
14            SBX:
15              specific_parameter:
16                sbxDistributionIndex:
17                  type: real
18                  values: [ 5.0, 400.0 ]
19            BLX_ALPHA:
20              specific_parameter:
21                blxAlphaCrossoverAlphaValue:
22                  type: real
23                  values: [ 0.0, 1.0 ]
24            wholeArithmetic:
```

Listing 2: YAML con el espacio de parámetros completo de crossover

Quitando lo relativo a *SBX* y estableciendo el *upperBound* de *BLX_ALPHA* a 1.5 del archivo *YAML* 2:

```
1         crossover:
2           type: categorical
3           global_parameters:
4             crossoverProbability:
5               type: real
6               values: [ 0.0, 1.0 ]
7             crossoverRepairStrategy:
8               type: categorical
9               values:
10                random:
11                round:
12                bounds:
13          values:
14            BLX_ALPHA:
15              specific_parameter:
16                blxAlphaCrossoverAlphaValue:
17                  type: real
18                  values: [ 0.0, 1.5 ]
19            wholeArithmetic:
```

Listing 3: YAML sin SBX y BLX ALPHA upperBound a 1.5

El resultado final del código sería:

```java
private void variation() {
    CrossoverParameter crossoverParameter = new CrossoverParameter(
        List.of("BLX_ALPHA", "wholeArithmetic"));
    ProbabilityParameter crossoverProbability =
        new ProbabilityParameter("crossoverProbability");
    crossoverParameter.addGlobalParameter(crossoverProbability);
    RepairDoubleSolutionStrategyParameter crossoverRepairStrategy =
        new RepairDoubleSolutionStrategyParameter(
            "crossoverRepairStrategy", Arrays.asList("random", "round", "bounds"));
    crossoverParameter.addGlobalParameter(crossoverRepairStrategy);

    RealParameter alpha = new RealParameter("blxAlphaCrossoverAlphaValue", 0.0, 1.5);
    crossoverParameter.addSpecificParameter("BLX_ALPHA", alpha);
```

Fragmento de código 2: Generación dinámica del método variation() de la clase ConfigurableNSGAII a través del *YAML* 3

De esta manera, se está generando el código de manera automática desde un archivo *YAML* en base a nuestras necesidades concretas. Aplicando este enfoque, se pueden generar dinámicamente las clases que implementan versiones configurables de NSGA-II con los parámetros seleccionados en el fichero YAML.

Además, permite que una persona que no sepa de programación pueda usar Evolver y hacer cambios en los algoritmos.

A continuación, se presentará el espacio de parámetros completo, contando con los cinco parámetros de primer nivel:

Si se quiere eliminar algún parámetro, basta con eliminarlo del *YAML* y viceversa.

5.2. Proceso de desarrollo

Los pasos seguidos para elaborar una solución viable para la generación automática de código han sido los siguientes:

1. Estudio de los archivos YAML.
2. Investigación y pruebas con ejemplos simples de lectura de archivos YAML.
3. Elección del procedimiento a seguir para la generación dinámica.
4. Validación por parte del tutor.
5. Lectura de *YAML* y generación automática de código con un único parámetro para valorar si el procedimiento elegido resulta ser adecuado.

6. Validación por parte del tutor.

7. Escalar la solución al resto de parámetros de primer nivel.

8. Muestra en la consola de la generación automática de código para valorar y perfilar el resultado final.

9. Validación por parte del tutor.

10. Creación de una nueva clase, llamada *EvNSGAIIDoubleProblem*, una vez se está contento con el resultado final.

11. Validación por parte del tutor.

5.3. Desarrollo de la solución previa

Para diseñar la estrategia de generación automática de código se ha desarrollado en primera instancia y, a partir de ella, la solución definitiva.

El objetivo inicial de la primera versión es generar automáticamente la clase *ConfigurableNSGAII*, que sólo permite generar configuraciones para problemas continuos. Esta tarea se completó, de forma que el algoritmo se generó correctamente y se dio por cumplido el objetivo. Sin embargo, se replanteó el refactorizar y simplificar la clase *ConfigurableNSGAII* para que pudiese soportar varias codificaciones, por ejemplo, la DoubleSolution y PermutationSolution.

Por ello, aunque se logró generar dinámicamente la clase en cuestión, ese esquema se desechó. La estructura de los paquetes quedó de la siguiente manera, en la que se procesaron y generaron los parámetros de primer nivel por separado, excepto *offspringpopulationsize*:

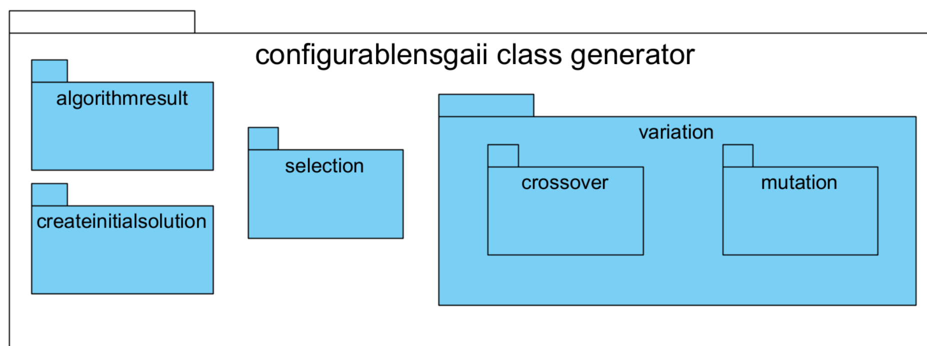

Figura 11: Estructura de paquetes del generador de ConfigurableNSGAII

No se va a hablar acerca de esta solución ya que sigue exactamente el mismo enfoque que la solución definitiva. La única diferencia entre esta solución y la definitiva es el código en sí generado.

Haber generado correctamente esta solución ha sido muy importante ya que ha servido para adquirir experiencia, abordar cómo resolver el problema, lo que ha servido como base para implementar la solución que se detalla en el siguiente apartado. Aunque nos vamos a centrar en la otra versión del algoritmo NSGAII implementada, se ha decidido también incluir el generador automático de la clase *ConfigurableNSGAII*.

5.4. Estructura de la solución definitiva

La solución definitiva está englobada dentro del paquete *codegenerator*. Este paquete, a su vez, contiene tres paquetes, cada uno supone una clase distinta generada.

1. *configurablensgaii*: véase la sección 5.3.
2. *doubleproblem*: genera la clase *EvNSGAIIDoubleProblem* (versión nueva de la clase *ConfigurableNSGAII*). Explicado en la sección 5.5.
3. *permutationproblem*: comentado en la sección 5.6.

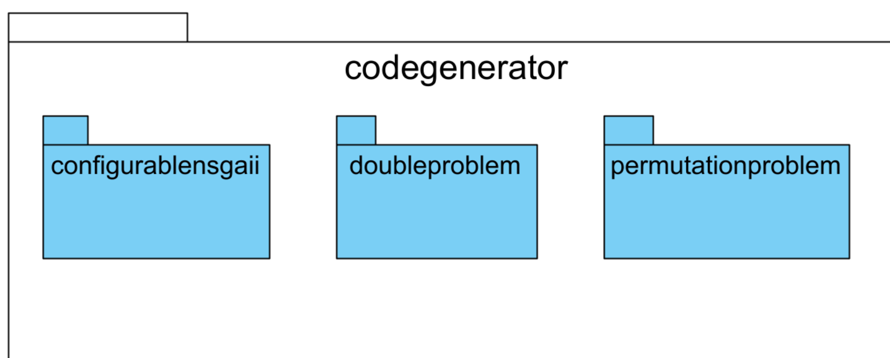

Figura 12: Estructura de paquetes para la generación de los parámetros de primer nivel de *EvNSGAIIDoubleProblem*

5.5. Desarrollo de la solución definitiva (DoubleSolution)

En esta solución definitiva, se va a intentar generar automáticamente la clase EvNSGAIIDoubleProblem en función de lo descrito en el archivo YAML. Se trata de una clase

refactorizada de *ConfigurableNSGAII* mucho más legible e intuitiva para llevar a cabo la generación de código.

Dada la complejidad de lo que se quiere hacer, el primer paso es ver si es viable generar el código automáticamente y comprobar si se puede hacer. Por ello, simplificamos el problema y lo reducimos a generar dinámicamente un único operador. El elegido fue el *crossover* (subparámetro de *variation*), ya que es bastante completo. Si se consigue poder generarlo automáticamente, esto podría hacerse con cualquier operador y entonces generar la clase entera EvNSGAIIDoubleProblem debería ser viable.

En este punto, lo que nos importa de la clase *EvNSGAIIDoubleProblem* son los métodos *setParameterSpace()* y *setParameterRelationships()* ya que son los únicos que varían en función del archivo YAML. En este caso, nos vamos a centrar en el método *setParameterSpace()*. Para ello se creó en primer lugar el paquete *codegenerator*, dentro del cual se incluyó el subpaquete *doubleproblem*, destinado a gestionar problemas con variables de tipo *double*.

Uno de los tres subpaquetes de la figura 12 *doubleproblem*. Éste indica explícitamente el tipo de problema con el que se está trabajando, que es, en este caso, con una codificación de arrays de dobles.

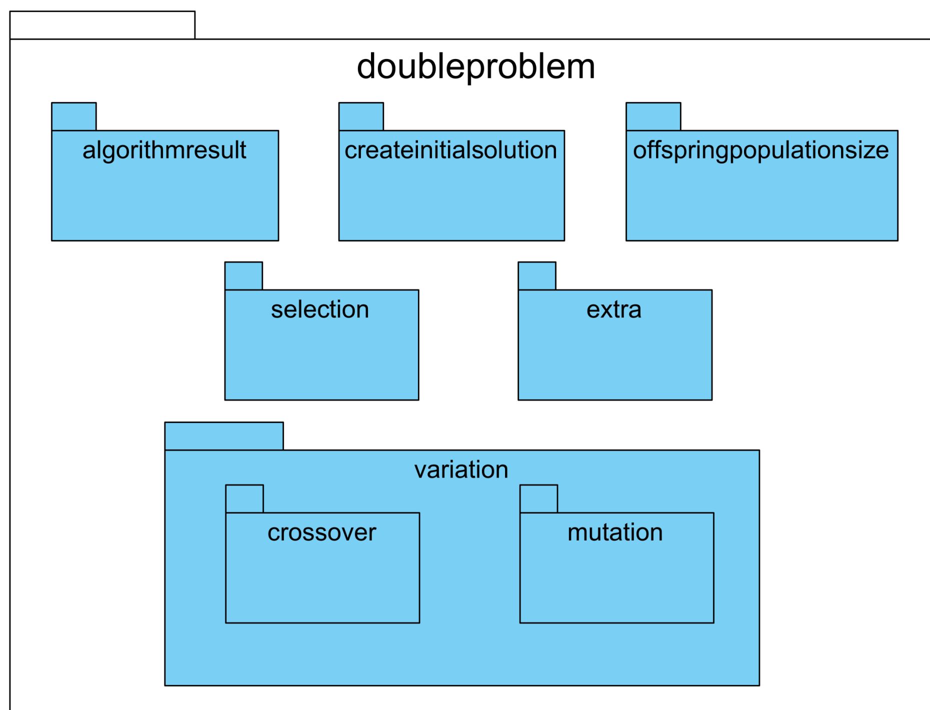

Figura 13: Estructura de paquetes de doubleproblem

Cada parámetro de primer nivel es tratado de forma independiente. Para su generación, cada paquete con el nombre de un parámetro de primer nivel cuenta con una arquitectura modular basada en tres componentes principales:

- Un processor, encargado de leer y procesar el archivo *YAML* de entrada.
- Una clase para generar el método *setParameterRelationships*, responsable de establecer las relaciones entre parámetros.
- Una clase para generar el método *setParameterSpace*, que define el espacio de búsqueda correspondiente al parámetro.

Esta separación de responsabilidades mejora la escalabilidad y el mantenimiento del código, permitiendo extender fácilmente el soporte a nuevos tipos de parámetros o problemas.

Por tanto, ante este *YAML* de entrada:

```yaml
crossover:
  type: categorical
  global_parameters:
    crossoverProbability:
      type: real
      values: [ 0.0, 1.0 ]
    crossoverRepairStrategy:
      type: categorical
      values:
        random:
        round:
        bounds:
  values:
    BLX_ALPHA:
      specific_parameter:
        blxAlphaCrossoverAlphaValue:
          type: real
          values: [ 0.0, 1.5 ]
    wholeArithmetic:
```

Listing 4: Ejemplo YAML de entrada para generar código Java

El *YAML* 4 genera el siguiente código del método *setParameterSpace()*:

```java
private void setParameterSpace() {
    parameterSpace.put(
            ParameterNames.VARIATION,
            new VariationParameter(List.of("crossoverAndMutationVariation")));

    parameterSpace.put(
            ParameterNames.CROSSOVER,
            new CrossoverParameter(List.of("BLX_ALPHA", "wholeArithmetic")));

    parameterSpace.put(
            ParameterNames.CROSSOVER_PROBABILITY,
            new RealParameter(ParameterNames.CROSSOVER_PROBABILITY, 0.0, 1.0));

    parameterSpace.put(
            ParameterNames.CROSSOVER_REPAIR_STRATEGY,
            new RepairDoubleSolutionStrategyParameter(
                ParameterNames.CROSSOVER_REPAIR_STRATEGY, List.of("random", "round")));

    parameterSpace.put(
            ParameterNames.BLX_ALPHA_CROSSOVER_ALPHA_VALUE,
            new RealParameter(ParameterNames.BLX_ALPHA_CROSSOVER_ALPHA_VALUE, 0.0, 1.5));
}
```

Fragmento de código 3: Generación dinámica del método setParameterSpace en base al *YAML* 4

Como vemos, se genera todo lo relativo al crossover en el método setParameterSpace(). Lo mismo ocurriría para el método *setParameterRelationships()*.

A continuación, se mostrará el código desarrollado a alto nivel para hacer posible la generación automática.

```java
public void VariationProcessorSetParameterSpaceMain() {
// Process crossover
    Map<String, Object> crossover = (Map<String, Object>) specificParams.get("crossover");
    CrossoverProcessor crossoverProcessor = new CrossoverProcessor();
    crossoverProcessor.readAndParseCrossover(crossover);
    crossoverProcessor.printSetParameterSpaceCode();
// mismo procedimiento para mutation
}
```

Fragmento de código 4: Estructura generación dinámica del método setParameterSpace()

Después, lo que realmente genera el código es la clase *CrossoverProcessor*, donde se llaman a los métodos *readAndParseCrossover()* y *printSetParameterSpaceCode()*.

La clase *CrossoverProcessor* tiene varios pilares. Uno de ellos es, sin duda, la definición de unas variables globales:

```java
public class CrossoverProcessor {
    // Variables globales
    private String formatedCrossoverNames;
    private List<Double> crossoverProbRange;
    private String formatedRepairStrategies;
    private Map<String, List<Object>> specificParamValues;

    // Resto de la implementacion...
}
```

Fragmento de código 5: Variables en CrossoverProcessor

A continuación, se explicarán las variables empleadas en 5, ya que desempeñan un papel fundamental en el funcionamiento del sistema:

1. El string *formattedCrossoverNames* almacena los valores de cruce elegidos en el *YAML* (*SBX*, *BLX_ALPHA* y/o *wholeArithmetic*).

2. La lista *crossoverProbRange* registra los límites de la probabilidad de crossover, entre 0 y 1.

3. El string *formattedRepairStrategies* recopila las estrategias de reparación del crossover (*random*, *round* y/o *bounds*).

4. El mapa *specificParamValues* *<String, List<Object>>* tiene como clave cada uno de los elementos de *formattedCrossoverNames*, y como valor una lista que representa los valores mínimo y máximo del parámetro específico del operador correspondiente, en el caso de caso de que estos valores existan.

Podemos ver la manera en la que se emplean las variables explicadas anteriormente en el código Java mostrado a continuación. Así se generaría lo relativo al crossover en el método setParameterSpace():

```java
1 public void printSetParameterSpaceCode() {
2     System.out.println("parameterSpace.put(\n" +
3             "        ParameterNames.CROSSOVER,\n" +
4             "        new CrossoverParameter(List.of(" +
5             formattedCrossoverNames + ")));\n");
6
7     System.out.println("parameterSpace.put(\n" +
8             "        ParameterNames.CROSSOVER_PROBABILITY,\n" +
9             "        new RealParameter(ParameterNames.CROSSOVER_PROBABILITY, "
10            + crossoverProbRange.get(0) + ", " + crossoverProbRange.get(1) + "));\n");
11
12    System.out.println("parameterSpace.put(\n" +
13            "        ParameterNames.CROSSOVER_REPAIR_STRATEGY,\n" +
14            "        new RepairDoubleSolutionStrategyParameter(\n" +
15            "            ParameterNames.CROSSOVER_REPAIR_STRATEGY, List.of(" +
16            formattedRepairStrategies + ")));\n");
17
18    // Dynamically invoke printsetParameterSpaceCode from each crossover class
19    for (Map.Entry<String, List<Object>> entry : specificParamValues.entrySet()) {
20        try {
21            String className = "org.uma.evolver.parameter.catalogue.crossover." +
22            entry.getKey();
23            Class<?> clazz = Class.forName(className);
24            Object instance = clazz.getDeclaredConstructor().newInstance();
25            Method method = clazz.getMethod("printsetParameterSpaceCode",
26            Double.class, Double.class);
27
28            Double min = (Double) entry.getValue().get(0);
29            Double max = (Double) entry.getValue().get(1);
30
31            String result = (String) method.invoke(instance, min, max);
32            System.out.println(result);
33
34        } catch (ClassNotFoundException e) {
35            System.err.println("Class not found: " + entry.getKey());
36        } catch (NoSuchMethodException e) {
37            System.err.println("Method 'printsetParameterSpaceCode' not found in class: "
38            + entry.getKey());
39        } catch (Exception e) {
40            e.printStackTrace();
41        }
42    }
43 }
```

Fragmento de código 6: Ejemplo código del generador del método setParameterSpace

La clave ha sido aplicar la reflexión (a partir de la línea 17). Todo el código que dependía de los valores del crossover (*SBX*, *BLX_ALPHA*, *wholeArithmetic*), y sus respectivos valores inferiores y superiores, se ha generado gracias a este interesante y útil enfoque. El funcionamiento de la reflexión empleado en REvoler es sencillo: si un valor de *crossover* aparecía en el archivo YAML, se llama a una clase que se encargaría de generar el código relativo a dicho valor de cruce. Véamoslo con un ejemplo, pongámonos en el caso de que en el archivo *YAML* aparece *SBX* como valor de *crossover*:

```
1 public class SBX {
2
3     public String printsetParameterSpaceCode(Double minValue, Double maxValue) {
4         return "\nparameterSpace.put(\n" +
5                 "         ParameterNames.SBX_DISTRIBUTION_INDEX,\n" +
6                 "         new RealParameter(ParameterNames.SBX_DISTRIBUTION_INDEX,"+
7                 minValue+", "+maxValue+"));\n";
8     }
9
10    public String printsetParameterRelationshipsCode() {
11        return "\nparameterSpace\n" +
12                "             .get(ParameterNames.CROSSOVER)\n" +
13                "             .addSpecificParameter(\"SBX\",
14                parameterSpace.get(ParameterNames.SBX_DISTRIBUTION_INDEX));";
15    }
16 }
```

Fragmento de código 7: Métodos de SBX llamados en la reflexion

A través de la reflexión, se llamaría a los métodos de la clase SBX.

Hemos visto que es posible generar el *crossover* dinámicamente en función del YAML, por tanto, la solución es totalmente adaptable y se pueden generar dinámicamente los métodos *setParameterSpace()* y *setParameterRelationships()*.

Si se aplica este mismo enfoque para la mutación, obteniendo de este modo la variación, así como el resto de parámetros de primer nivel, podemos generar la clase *EvNSGAII-DoubleProblem* completa.

Este ha sido todo el procedimiento seguido para generar dinámicamente el *crossover* en función de lo descrito en el archivo YAML. Al ser una solución tan clara y sólida, para generar la clase entera dinámicamente habría que aplicar este mismo procedimiento para el *mutation*, con ello obtendríamos la *variation*, y el resto de parámetros de primer nivel.

5.6. Desarrollo de la solución definitiva (PermutationSolution)

Debido a que un objetivo del TFG es poder trabajar con varias codificaciones, se decidió generar dinámicamente la clase EvNSGAIIPermutationSolution. En esta clase se trabaja también con el algoritmo NSGA-II pero con una codificación distinta, con la *PermutationSolution*. Se trata de una clase muy parecida a *EvNSGAIIDoubleProblem* pero adaptada a la codificación de permutaciones. No se va a explicar en detalle ya que el procedimiento es idéntico al apartado 5.3, en el que se generó un algoritmo NSGA-II para *DoubleSolution*.

6. Implementación de una codificación en árbol

Uno de los inconvenientes de Evolver, comentado en la sección 3.4, es el uso de una codificación de los parámetros basada en un array de números reales, con una posición por parámetro, con valores entre cero y uno, así como la inclusión de todos los parámetros del espacio de parámetros, incluyendo los que no son necesarios en determinadas configuraciones. El motivo es que puede que los cambios de una configuración a nivel de codificación, pero que eso no provoque ningún cambio al decodificar.

Una alternativa es codificar las configuraciones en una estructura de árbol Christoph Neumüller and Affenzeller [2012], en la que se reflejen los tipos de los parámetros (categórico, entero, real), los subparámetros que tienen y las relaciones entre ellos. Dado que se está usando *YAML* en este TFG como tecnología sobre la que gira el almacenamiento y procesamiento de configuraciones, la idea es definir una codificación basada en ese formato (clase *YamlSolution*). Esto implica que hay que implementar una clase para los problemas de esta codificación (clase *YamlProblem*) y los correspondientes operadores de cruce y mutación.

6.1. Motivación

La investigación siempre me ha llamado la atención; sin embargo, en la carrera no se ha visto nada al respecto. Por ello, el TFG es el momento ideal para sumergirnos en ella.

En un algoritmo evolutivo el coste computacional, cuando se aplica a problemas reales, viene dado por la evaluación de las nuevas soluciones que se generan. En el caso de Evolver/REvolver, las soluciones son configuraciones del algoritmo NSGA-II, por lo que cada vez que genera una configuración hay que ejecutar NSGA-II sobre cada uno de los problemas de entrenamiento. Este puede requerir un tiempo de cómputo del orden de decenas de segundos o incluso minutos, lo que puede hacer que un proceso de optimización requiera muchas horas de cómputo en una máquina con decenas de *cores*.

Por tanto, dado que la codificación original de Evolver, basada en arrays de números reales, puede dar lugar a configuraciones nuevas, pero que, al ser decodificadas, en realidad sean iguales que las originales, puede hacer que sí se pierda tiempo. La codificación basada en árbol debería, en teoría, ser más eficiente, pero no hay ningún trabajo de investigación que haya abordado el impacto real de usar la codificación en array. En esta parte del TFG

se plantea realizar una primera etapa de lo que sería un estudio experimental que conduciría a determinar si la codificación en árbol hace que el proceso de meta-optimización sea más eficiente que usando *arrays*.

6.2. Problema jMetal

Está claro que *jMetal* es una herramienta maravillosa; sin embargo, un inconveniente que tenía era que, al integrarlo con *Evolver*, solo soportaba soluciones de números reales (*DoubleSolution*). Está basada en codificar el espacio de parámetros de una metaheurística como un array de valores reales en el rango [0,0, 1,0]. La codificación de números reales tiene un gran potencial; en cambio, hay ocasiones en las que peca de ineficiente. Debido a ello, se propone en este TFG diseñar, desarrollar y evaluar un nuevo tipo de *Solution* que mejore a la ya existente.

6.3. Proceso de desarrollo

Para realizar este objetivo, los pasos seguidos han sido los siguientes:

1. Análisis de cómo se implementan las soluciones, los problemas y los operadores de cruce y mutación en *jMetal*.

2. Elección del enfoque que se iba a seguir. Se escogió una *Solution* basada en árboles, tal y como viene descrito en el artículo *Parameter Meta-optimization of Metaheuristic Optimization Algorithms*.

3. Elaboración de los distintos paquetes.

4. Prototipo del *Problem* y *Solution* para comentar las distintas opciones de diseño con el tutor.

5. Diseño e implementación del nuevo *Problem* y *Solution* (*YamlProblem* y *YamlSolution*).

6. Prototipo de generación de soluciones iniciales en forma de YAML.

7. Diseño e implementación de soluciones iniciales en forma de YAML.

8. Prototipo de los operadores de *mutation* y *crossover* para esta nueva *Solution* (*YamlMutation* y *YamlCrossover*). Se empezará con la *mutation* y, una vez funcione, se procederá a incorporar el *crossover*.
9. Diseño e implementación de la mutación.
10. Modificación de las clases necesarias de *REvolver* para permitir autoconfigurar soluciones de la codificación nueva en desarrollo.
11. Probar la autoconfiguración del algoritmo *NSGA-II*, empleando la nueva codificación (únicamente contando con la mutación).
12. Valoración del tutor.
13. Desarrollo del *crossover*.
14. Probar la autoconfiguración del algoritmo *NSGA-II*, empleando la nueva codificación (contando con *mutation* y *crossover*), con la herramienta *Evolver*.

6.4. Arquitectura

A modo de introducción y puesta en contexto de los siguientes apartados, se va a mostrar la arquitectura de la nueva solución ideada:

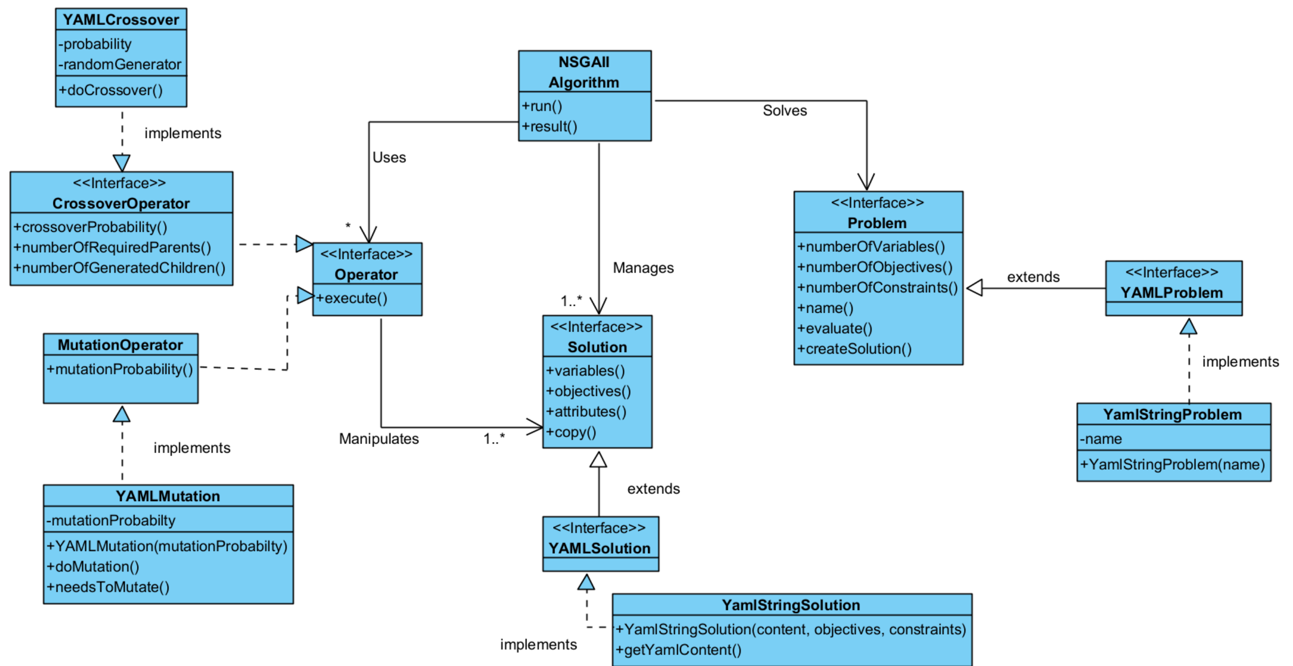

Figura 14: Diagrama de clases con las clases implementadas en el contexto de la arquitecture de jMetal.

Como se puede ver, todo lo creado sigue y se amolda a la estructura de jMetal.

6.5. Implementación de la clase YAMLProblem

YamlProblem es un nuevo tipo de problema personalizado diseñado para trabajar con representaciones en formato *YAML* dentro del marco de algoritmos evolutivos de *jMetal*. En lugar de definir un problema matemático clásico (como minimizar una función), este componente permite evolucionar configuraciones definidas en ficheros YAML, lo que abre la puerta a metaoptimizar configuraciones de algoritmos o sistemas complejos.

El paquete *yamlproblem* se encuentra estructurado de la siguiente manera:

- *org.uma.evolver.jmetal.core.problem.yamlproblem*: contiene la definición general de problemas basados en YAML.
- *org.uma.evolver.jmetal.core.problem.yamlproblem.impl*: contiene las implementaciones concretas, como *YamlStringProblem*.

Esta separación entre interfaz e implementación favorece la extensibilidad, permitiendo definir múltiples tipos de problemas basados en *YAML* con distintas características (por

ejemplo, con múltiples objetivos, restricciones o estructuras de datos más complejas).

El propósito de *YamlProblem* es permitir el uso de algoritmos evolutivos para explorar espacios de soluciones no tradicionales, como lo son los espacios de configuración. En concreto:

- Permite codificar una configuración *YAML* como solución candidata.
- Permite que los operadores evolutivos trabajen sobre configuraciones *YAML* válidas.
- Facilita la integración con herramientas que usan *YAML* como formato de entrada (por ejemplo, configuradores automáticos de algoritmos).

A la hora de integrar este nuevo problema en el sistema, hay que tener en cuenta varias cosas:

1. *Problema (YamlProblem)*: define la estructura del problema. Se espera una solución representada en formato YAML.
2. *Solución (YamlStringSolution)*: encapsula el contenido *YAML* que representa una posible configuración candidata.
3. *Generador inicial*: se parte de plantillas *YAML* que sirven como base para generar soluciones iniciales válidas completamente aleatorias.
4. *Evaluación*: el problema no evalúa directamente las soluciones, sino que deja ese papel a componentes externos (por ejemplo, evaluadores que ejecuten el *YAML* en un sistema real o simulado).

6.6. Implementación de la clase YamlSolution

La nueva *Solution* implementada, denominada *YamlSolution*, presenta novedades. En lugar de vectores numéricos, cada individuo es una cadena *YAML* que define una configuración completa. Esta solución permite aplicar operadores evolutivos directamente sobre configuraciones estructuradas. Se compone de una interfaz *YamlSolution* y una implementación concreta *YamlStringSolution*, totalmente integradas en *jMetal*.

La nueva solución se encuentra en el paquete *org.uma.evolver.jmetal.core.solution.yamlsolution*. La estructura de la *YamlSolution*, denominada *YamlStringSolution*, está compuesta por:

- El contenido del YAML, representado como un único string.
- El número de objetivos.
- El número de restricciones.

6.7. Definición de los nuevos operadores: mutation

La clase *YamlMutation* representa un operador de mutación específico para soluciones que codifican configuraciones en formato YAML. Hereda de *MutationOperator¡YamlSolution¿*, y su propósito es aplicar cambios (mutaciones) sobre fragmentos seleccionados de una solución YAML, simulando así pequeñas variaciones en la configuración de un algoritmo.

Este operador no trabaja sobre genes numéricos tradicionales, sino sobre bloques textuales del documento YAML. Para ello:

- Divide el *YAML* en secciones determinadas por los parámetros de primer nivel (*algorithmResult*, *variation*, *selection*, etc.).
- Decide, de manera probabilística, si cada sección debe ser mutada.
- En caso afirmativo, delega la mutación a submódulos especializados que mutarán lo relativo a cada parámetro de primer nivel.

El funcionamiento de la mutación describe un enfoque muy sencillo:

- Cada parámetro de primer nivel tiene la misma probabilidad de ser mutado.
- Dicha probabilidad es $1/n$ parámetros de primer nivel, 1/5, es decir, 0,2.
- Si el parámetro de primer nivel cumple la probabilidad de ser mutado, se muta; sino, se queda tal cual.
- Se aplica este enfoque al resto de parámetros y, con ello, se construye la solución mutada, el cual es un *YAML* mutado.

Cada parámetro de primer nivel tiene su propia mutación, orientada a ser lo más eficaz, eficiente e inteligente posible.

Gracias a este diseño modular y adaptable, el sistema permite modificar dinámicamente configuraciones complejas sin perder estructura, lo que resulta especialmente útil en contextos de metaoptimización o diseño automático de algoritmos.

6.8. Definición de los nuevos operadores: crossover

La clase *YamlCrossover* implementa un operador de cruce diseñado para soluciones que representan configuraciones en formato YAML. Hereda de *CrossoverOperator¡YamlSolution¿*, y su función principal es combinar dos configuraciones parentales para generar nuevas soluciones que mezclan características de ambos, simulando así una recombinación genética aplicada a configuraciones textuales.

En lugar de operar sobre valores numéricos convencionales, este operador:

- Trabaja con bloques de texto que corresponden a secciones clave del *YAML* (*algorithmResult*, *variation*, *selection*, etc.).
- Para cada sección, y condicionado por una probabilidad global, decide si realizar una combinación entre los padres o conservarlas tal cual.
- Delega la tarea de mezclar cada bloque a componentes especializados que aseguran mantener la coherencia y validez estructural del YAML.

El mecanismo de cruce es sencillo, pero efectivo:

- Si la probabilidad determina que se debe cruzar, se generan dos nuevos *YAML* resultantes de mezclar los bloques correspondientes de ambos padres.
- Si no, se retornan copias idénticas.
- Esta estrategia garantiza que las soluciones combinadas mantengan su integridad.

Al tener cada bloque una lógica de cruce propia, el diseño modular permite adaptarse y evolucionar de forma eficiente en el contexto de la metaoptimización o el diseño automático de algoritmos, facilitando la generación de nuevas configuraciones con características heredadas de distintas soluciones previas.

6.9. Adaptación de Evolver a la codificación en árbol

Tras realizar la nueva codificación en árbol, no se podía directamente incluir a Evolver, debido a que el proyecto se encuentra estrictamente ligado a la codificación de arrays de dobles. Implementar la codificación en árbol no es sólo implementar la solución, el problema y los operadores de cruce y mutación.

Para poder soportar la codificación en árbol, se han tenido que crear nuevas versiones de las clases:

1. *MetaOptimizationProblem*
2. *NSGAIIOptimizingNSGAIIForProblemZDT4*
3. *OutputResults*
4. *WriteExecutionDataToFilesObserver*
5. *ParameterManagement*

Las clases mencionadas dieron lugar a las siguientes:

1. *MetaOptimizationYAMLProblem*: se trata de la case en dónde se optimiza se hizo que se implementase la interfaz Problem de tipo YamlSolution. Esta clase es la más importante, ya que hay que instanciarla a la hora de optimizar cualquier algoritmo en cualquier problema. Lo que se ha hecho es adaptar la clase *MetaOptimization-Problem* a la codificación en árbol.
2. *NSGAIIOptimizingYAMLNSGAIIForProblemZDT4*: se ha empleado esta clase par comprobar el funcionamiento correcto del meta-optimizador. Se define un problema real, en este caso, el problema *ZTD4*
3. *OutputYamlResults*: aquí se muestran los resultados en una interfaz al optimizar un problema, por ejemplo, el *NSGAIIOptimizingYAMLNSGAIIForProblemZDT4*. Los resultados que se van obteniendo al optimizar se van mostrando. De este modo, se puede observar en tiempo real cómo las soluciones van mejorando, gracias al meta-optimizador. Además, esta clase se encarga de guardar en la carpeta *RESULTS* todos los resultados obtenidos así como la evolución de las configuraciones obtenidas en archivos CSV. Esto ya existía en Evolver, pero solo servía para la codificación de arrays de dobles, por tanto, ha habido que adaptar una versión para archivos YAML.

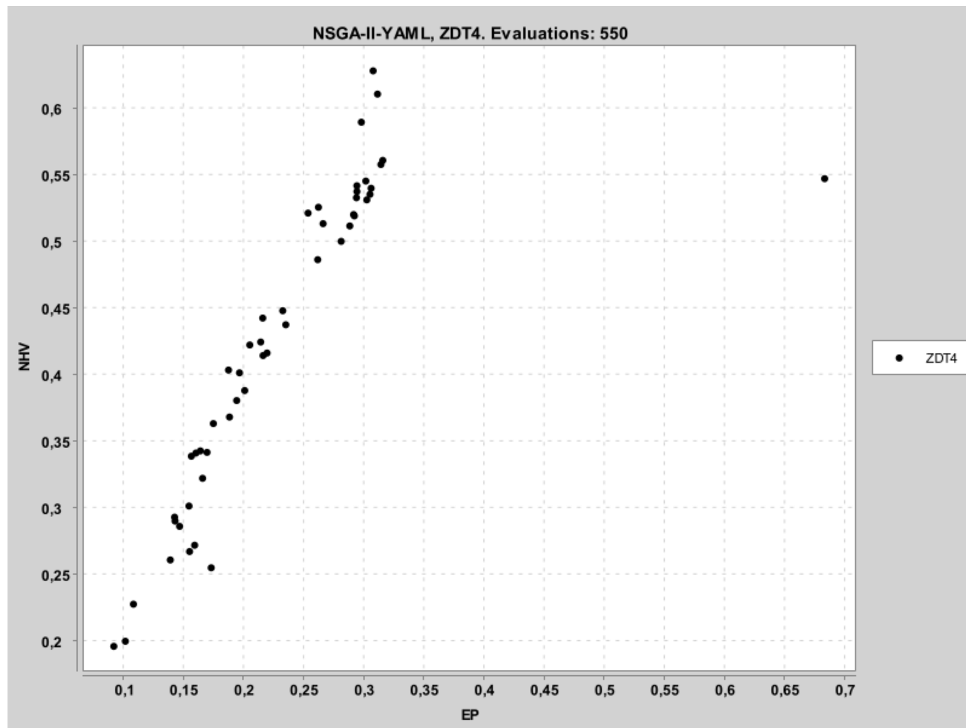

Figura 15: Interfaz generada por OutputYAMLResults para el la solución YAML.

4. *WriteExecutionDataToFilesObserverYAML*: esta clase trabaja de la mano de *OutputYamlResults*. Mientars esta guarda y muestra los resultados gráficamente, *WriteExecutionDataToFilesObserverYAML* muestra en terminal las evaluaciones realizada hasta el momento, en tiempo real.

```
jun 07, 2025 12:10:50 P. M. org.uma.jmetal.util.observable.impl.DefaultObservable register
INFO: DefaultObservable Evolutionary Algorithm: org.uma.jmetal.util.observer.impl.EvaluationObserver@5afa04c registered
jun 07, 2025 12:10:50 P. M. org.uma.jmetal.util.observable.impl.DefaultObservable register
INFO: DefaultObservable Evolutionary Algorithm: org.uma.jmetal.util.observer.impl.FrontPlotObserver@1e88b3c registered
jun 07, 2025 12:10:50 P. M. org.uma.jmetal.util.observable.impl.DefaultObservable register
INFO: DefaultObservable Evolutionary Algorithm: Observer that writes output files from a list of numbers representing iterations where the outputs are required registered
SLF4J(W): No SLF4J providers were found.
SLF4J(W): Defaulting to no-operation (NOP) logger implementation
SLF4J(W): See https://www.slf4j.org/codes.html#noProviders for further details.
jun 07, 2025 12:11:15 P. M. org.uma.evolver.util.WriteExecutionDataToFilesObserverYaml update
INFO: EVAlS -> 100
jun 07, 2025 12:11:15 P. M. org.uma.jmetal.util.observer.impl.EvaluationObserver update
INFO: Evaluations: 100
jun 07, 2025 12:11:29 P. M. org.uma.evolver.util.WriteExecutionDataToFilesObserverYaml update
INFO: EVAlS -> 150
jun 07, 2025 12:11:29 P. M. org.uma.jmetal.util.observer.impl.EvaluationObserver update
INFO: Evaluations: 150
jun 07, 2025 12:11:39 P. M. org.uma.evolver.util.WriteExecutionDataToFilesObserverYaml update
INFO: EVAlS -> 200
jun 07, 2025 12:11:39 P. M. org.uma.jmetal.util.observer.impl.EvaluationObserver update
INFO: Evaluations: 200
```

Figura 16: Información en tiempo real de las evaluaciones del metaoptimizador

5. *ParameterManagement*: Esta clase se emplea en En este caso, no se creo una nueva clase, sino que se añadieron una serie de métodos a la hora de manejar los parámetros en soluciones YAML. Esto ha sido fundamental, dado que la gestión de parámetros

en las soluciones solo estaba hecho para aquellas de soluciones de arrays de dobles, por ende, se ha tenido que adaptar la funcionalidad de la clase para que soporte los mismos parámetros pero expresados de una manera distinta, en una codificación en árbol.

Además, se han creado nuevas clases para obtener una funcionalidad idéntica a la de Evolver con codificación de arrays de dobles que se mencionarán a continuación.

Algo indispensable es poder crear soluciones iniciales de manera aleatoria, es decir, la generación de un archivo YAML, de manera aleatoria, que representará la solución inicial sobre la cual se empezará a mutar y cruzar para obtener una solución mejor. El funcionamiento del generador inicial de soluciones *YAML* es muy sencillo, lo único que hay que pasarle es un archivo *YAML* que contenga todas las posibilidades de configuración de cada parámetro de primer nivel. Con ello, se genera una solución inicial totalmente válida.

```
1 public YamlSolution createSolution() {
2     String initial = InitialSolutionYamlGenerator.generate("NSGAII-DOUBLE.yml");
3     return new YamlStringSolution(initial, numberOfObjectives(), numberOfConstraints());
4 }
```

La clase YamlStringParser decodifica el *YAML* en formato concreto para que Evolver lo evalúe. Véamoslo con un ejemplo. Si esta fuese una solución YAML:

```
1  algorithmResult:
2    type: categorical
3    values:
4      population:
5  createInitialSolutions:
6    type: categorical
7    values:
8      scatterSearch:
9  offspringPopulationSize:
10   type: categorical
11   values: 5
```

Listing 5: Ejemplo YAML a ser decodificado

Se decodificaría de la siguiente manera el *YAML* 6 para que Evolver pueda interpretar la solución obtenida.

```
1 --algorithmResult population
2 --createInitialSolutions scatterSearch
3 --offspringPopulationSize 5
4 --selection random
```

Listing 6: Ejemplo YAML a ser decodificado

Con todo lo mencionado en esta subsección, se ha conseguido incorporar la codificación en árbol a *Evolver*.

En resumen, la integración de esta nueva solución basada en configuraciones *YAML* dentro de *jMetal* representa un avance significativo para la flexibilidad y adaptabilidad de los algoritmos evolutivos. Al tratar las soluciones como documentos estructurados y legibles, se facilita:

- La manipulación y evolución de parámetros complejos de forma modular y transparente.
- La exploración eficaz del espacio de configuraciones mediante operadores especializados.
- El mantenimiento de la coherencia y validez de los datos.

Esta aproximación abre nuevas posibilidades para:

- La metaoptimización.
- El diseño automático.
- La mejora de *jMetal* como herramienta versátil en optimización basada en algoritmos evolutivos.

7. Evaluación de la nueva codificación en árbol

Una vez implementada la nueva representación basada en *YAML*, o árbol, para el problema de meta-optmización, el siguiente paso es realizar una comparativa rigurosa de dicha representación respecto a la original basada en arrays para determinar si hay mejoras significativas tanto cuantitativa (¿se acelera la ejecución con la nueva codificación?) como cualitativamente (¿se obtienen mejores soluciones?).

Responder a estas preguntas requeriría hacer una experimentación siguiendo la metodología científica que se aplica en artículos de optimización multi-objetivo con metaheurísticas, usando un amplio abanico de familias de problemas multi-objetivo e como conjuntos de entrenamiento. Llevar a cabo esa experimentación está fuera del ámbito del presente TFG, por lo que se va a aplicar la metodología pero usando un escenario simplificado, con una única familia de problemas.

7.1. Metodología experimental

Para comparar algoritmos evolutivos multi-objetivo la metodología que se sigue es:

1. Seleccionar los algoritmos a comparar.
2. Seleccionar los problemas a optimizar.
3. Seleccionar los indicadores de calidad que miden la bondad de las soluciones que generan los algoritmos sobre los problemas.
4. Configurar los parámetros de los algoritmos para que la comparativa sea justa.
5. Ejecutar cada par (algoritmo, problema) un número determinado de veces (10 o más)
6. Obtener medidas estadísticas de los resultados (medianas, rangos intercuartílicos) y aplicar test estadísticos.

A continuación se detalla cómo se han abordado estas etapas en el TFG. Dado que REvolver es un software de meta-optimización multi-objetivo, hay dos algoritmos involucrados: el algoritmo base y el meta-optimizador. Ambos son NSGA-II, pero configurar el primero de forma adecuada es lo que busca el segundo.

Para el meta-optimizador se ha optado por la configuración estándar para problemas continuos en el caso de usar la representación en array, y se ha hecho una adaptación para la codificación en árbol:

- Tamaño de población: 50.
- Condición de parada: 1000 evaluaciones.
- Cruce: SBX (índice de distribución = 20.0) / cruce *YamlCrossover*. En ambos casos, la probabilidad de cruce es 0.9.
- Mutación: polynomial (índice de distribución = 20.0) / *YAMLMutation*. En ambos casos, la probabilidad de mutación es 1.0/5 (5 es el número de parámetros de primer nivel del espacio de parámetros de NSGA-II).

El resultado de la meta-optimización son las mejores configuraciones encontradas según los indicadores de calidad seleccionados, que han sido EP (epsilon) y NHV (hipervolumen normalizado). En ambos casos, cuanto menores son sus valores, mejores son las soluciones.

De esta etapa se obtendrán varias configuraciones para NSGA-II: la obtenida usando la codificación de array de Evolver (EvNSGAII), la resultante de la codificación con *YAML* usando *random crossover* (Y-NSGAII-RC) y la obtenida con *YAML* y cruce no aleatorio (Y-NSGAII-SBC). En la segunda, cuando se cruzan parámetros enteros o reales, el cruce de los mismos se hace de forma aleatoria a partir de los padres; en la tercera, cuando el valor de cruce de ambos padres es de tipo SBX, se realiza el cruce propio de SBX (*SBXCrossover*), con su correspondiente sbxDistributionIndex con el valor de 20.0. A su vez, cuando los dos padres tienen un valor de cruce de BLX ALPHA, se va a proceder a realizar el cruce propio de BLX ALPHA (BLXALPHACrossover).

Por tanto, la comparativa será entre estas tres versiones de NSGA-II más la original, que usa la configuración por defecto. De ese modo se podrá determinar, por un lado, las mejoras del enfoque de meta-optimización respecto del algoritmo NSGA-II de referencia y, por otro, si la codificación en árbol da mejores resultados que la de array. En esta comparativa, todas estas versiones de NSGA-II se configuran con un tamaño de población de 100 individuos y la condición de parada es hacer 30.000 evaluaciones. Para cada combinación (problema, algoritmo) se han hecho 25 ejecuciones independientes.

Como problemas de entrenamiento, se ha usado la familia DTLZ Deb et al. [2005],

compuesta por siete problemas continuos de tres objetivos. Por tanto, la meta-optimización buscará configuraciones de NSGA-II para estos siete problemas en conjunto.

7.2. Métricas consideradas para la comparación de los algoritmos

En el contexto de optimización multiobjetivo, las métricas de calidad son herramientas esenciales para evaluar el desempeño de los algoritmos. Dos de las métricas más relevantes implementadas en jMetal son el *Épsilon Aditivo (EP)* y el *Hipervolumen Normalizado (NHV)*, cada una con características distintivas que complementan el análisis del frente de Pareto obtenido.

El Épsilon Aditivo, también conocido como *Additive Epsilon Indicator*, es una métrica que cuantifica la distancia mínima necesaria para que una aproximación del frente de Pareto domine a otra. Matemáticamente, se define como:

$$I_{\epsilon+}(A, B) = \inf\{\epsilon \in \mathbf{R} \mid \forall \mathbf{z}^2 \in B, \exists \mathbf{z}^1 \in A \mid z_i^1 \leq z_i^2 + \epsilon, \forall i\} \tag{3}$$

donde:

- A representa el frente de aproximación.
- B es el frente de referencia.
- $\mathbf{z}^1$ y $\mathbf{z}^2$ son vectores de objetivos.

En cuanto a sus características clave, encontramos:

- *Interpretación intuitiva*: Valores más pequeños indican mejor calidad.
- *Comparación directa*: Permite comparar dos conjuntos de soluciones.
- Propósito: mide el grado de convergencia de un conjunto de soluciones respecto al frene de referencia.
- *Eficiencia computacional*: Relativamente rápido de calcular.

El Hipervolumen Normalizado es una variante del clásico hipervolumen (HV) que opera en un espacio normalizado $[0, 1]^n$, donde n es el número de objetivos. Se calcula como:

$$NHV(A) = 1 - \frac{HV(A)}{HV(R)} \tag{4}$$

donde:

- $HV(A)$ es el hipervolumen del frente de aproximación.
- $HV(R)$ es el hipervolumen del frente de referencia.
- El punto de referencia se ajusta a $(1{,}1, \ldots, 1{,}1)$ en espacio normalizado.

Las ventajas por emplear NHV son:

- *Comparabilidad*: Al normalizar, permite comparar resultados entre diferentes problemas
- *Balance*: Considera tanto convergencia como diversidad
- *Universalidad*: Aplicable a problemas con cualquier número de objetivos
- *Sensibilidad a dominancia*: Sólo contribuyen soluciones no dominadas

Un NHV de 0.0 indica optimalidad, mientras que valores cercanos a 1 muestran bajo desempeño. En jMetal, esta métrica es particularmente útil para:

- Validar nuevos algoritmos
- Comparar variantes de metaheurísticas
- Ajustar parámetros automáticamente

7.3. Comparativa gráfica de los algoritmos

Al ejecutar el proceso de meta-optimización (usar NSGA-II para optimizar NSGA-II para los problemas DTLZ) se almacenan los frentes que se van obteniendo con una periodicidad, que por defecto es el tamaño de la población del meta-optimizador. Estos frentes se pueden además visualizar durante la ejecución. A continuación, muestran ejemplos de esta visualización.

En las siguientes gráficas, cada punto representa una configuración y su posición representa la media de los siete problemas para cada indicador. Recordemos que el objetivo es minimizar el NHV y EP, es decir, que se encuentren lo más próximos a 0.

En la figura 17 se muestra el frente tras 1000 evaluaciones del algoritmo EvNSGAII, versión de arrays de dobles. Se puede apreciar que las soluciones son diversas (no están agrupadas en una pequeña región) y que hay una única solución no dominada con los valores EP=0.057 y NHV = 0.0817.

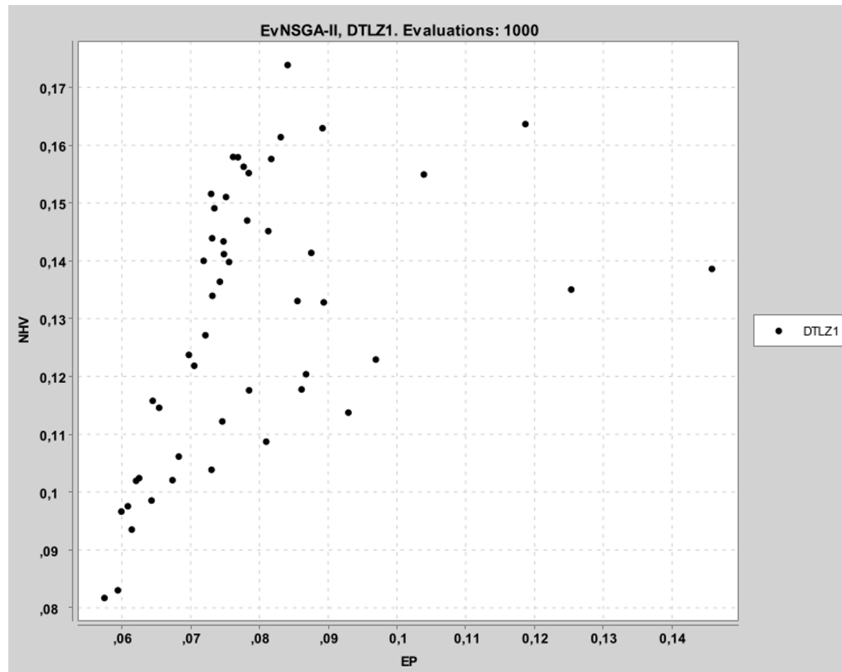

Figura 17: Evolución de las soluciones para los problemas DTLZ tras 1000 evaluaciones empleando EvNSGAII.

El frente obtenido en las mismas condiciones por el algoritmo Y-NSGAII-RC se incluye en la figura 18. En este caso, hay una solución destacada sobre el resto, con valores EP = 0.057 y NHV = 0.0818.

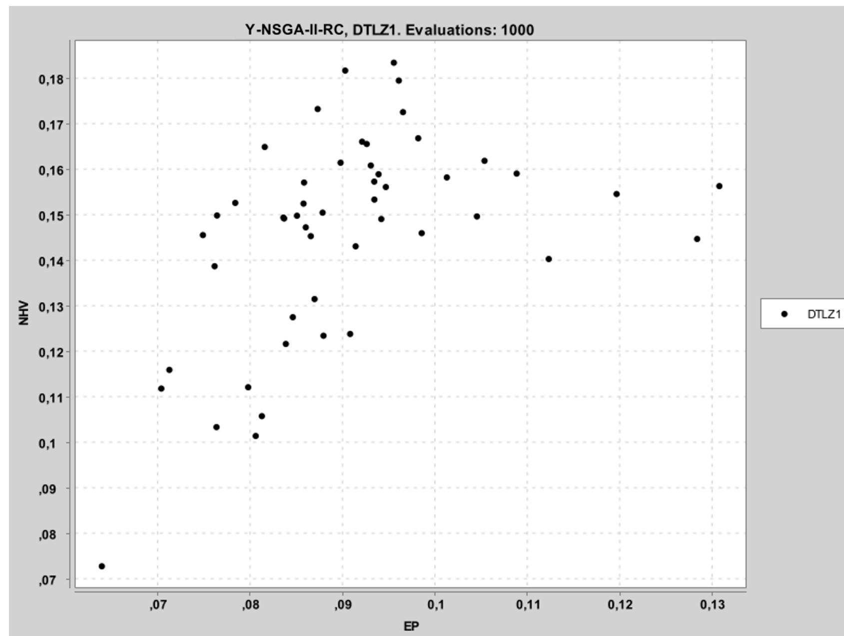

Figura 18: Evolución de las soluciones para el problema DTLZ1 a lo largo de las 1000 evaluaciones empleando Y-NSGAII-RC.

Por último, en la figura 19, en la que el algoritmo es la variante Y-NSGAII-SBC, se observa que hay dos soluciones no dominadas (EP,NHV):(0.060, 0.0785) y (0.054, 0.0855).

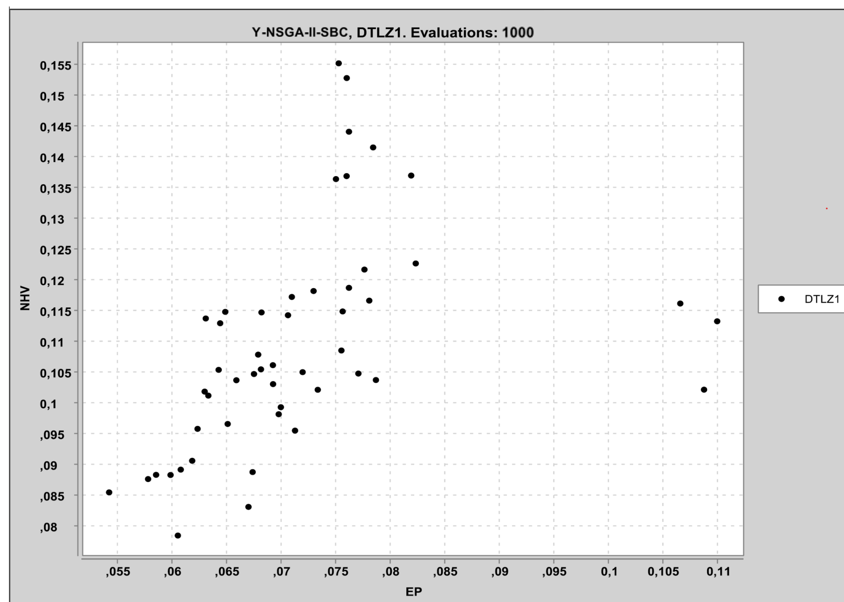

Figura 19: Evolución de las soluciones para el problema DTLZ1 a lo largo de las 1000 evaluaciones empleando Y-NSGAII-SBC.

Ahora vamos a estudiar cómo evolucionan los algoritmos a lo largo de las 1000 evaluaciones. Para ello, se han cogido cinco muestras:

- *CSV1* representa la primera muestra de todas, a las 100 evaluaciones. Nos sirve para estudiar la calidad de la primera solución obtenida.
- *CSV2* a las 250 evaluaciones.
- *CSV3* a las 500 evaluaciones.
- *CSV4* a las 750 evaluaciones.
- *CSV5* la última muestra es a las 1000 evaluaciones. Coincide con el último valor de las figuras 17, 18 y 19.

Cada CSV representa la mejor, o mejores soluciones, obtenidas en cada una de las evaluaciones. Si alguno de los puntos no aparece, no es que no exista, sino que el CSV inmediatamente siguiente se encuentra en el mismo punto exactamente. Eso nos indica que no se ha mejorado la solución de una evaluación a otra. Esto se puede ver claramente en la figura 20. El CSV 3, de color verde, a las 500 evaluaciones no aparece. Eso nos indica que el CSV inmediatamente siguiente, es decir, el CSV 4, de color naranja, a las 750 evaluaciones posee el mismo valor que a las 500 evaluaciones. Poder observar la evolución de las soluciones es de gran utilidad, ya que nos aporta información acerca de cuáles de los algoritmos converge antes, cuál obtiene inicialmente mejores soluciones o en dónde no se mejora la solución respecto a la evaluación, o archivo CSV, anterior.

Las gráficas mostradas a continuación han sido generadas añadiendo lo correspondiente al proyecto SAES, el cual es tratado en la sección 7.5. Se añadió el código correspondiente, como Jupyter Notebook, para poder representar de manera gráfica la información contenida en los respectivos CSV. Dichos CSV fueron generados con Evolver, a través de la clase *OutputYamlResults*, explicada en la sección 6.9.

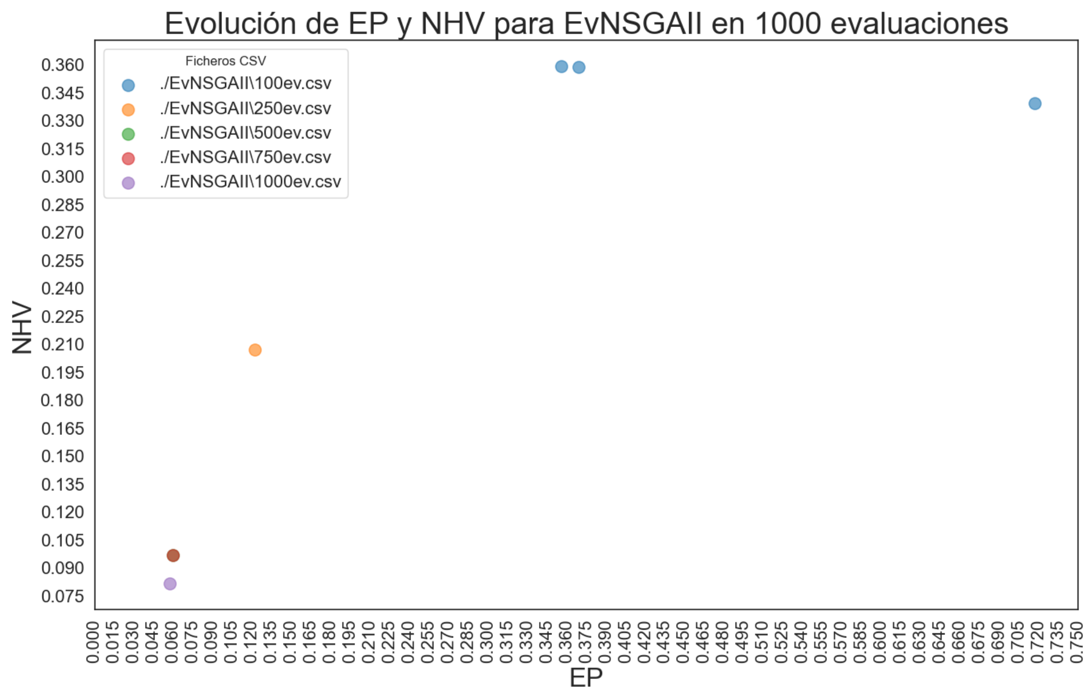

Figura 20: Problema DTLZ1 en la evaluación 1000 con EvNSGAII

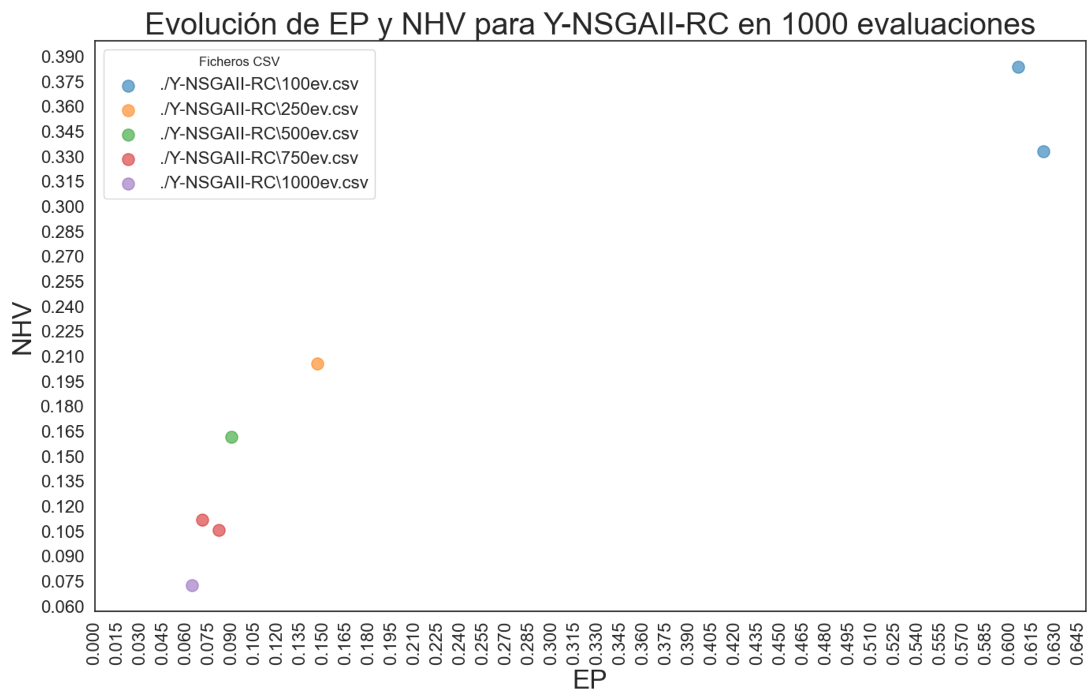

Figura 21: Problema DTLZ1 en la evaluación 1000 con Y-NSGAII-RC

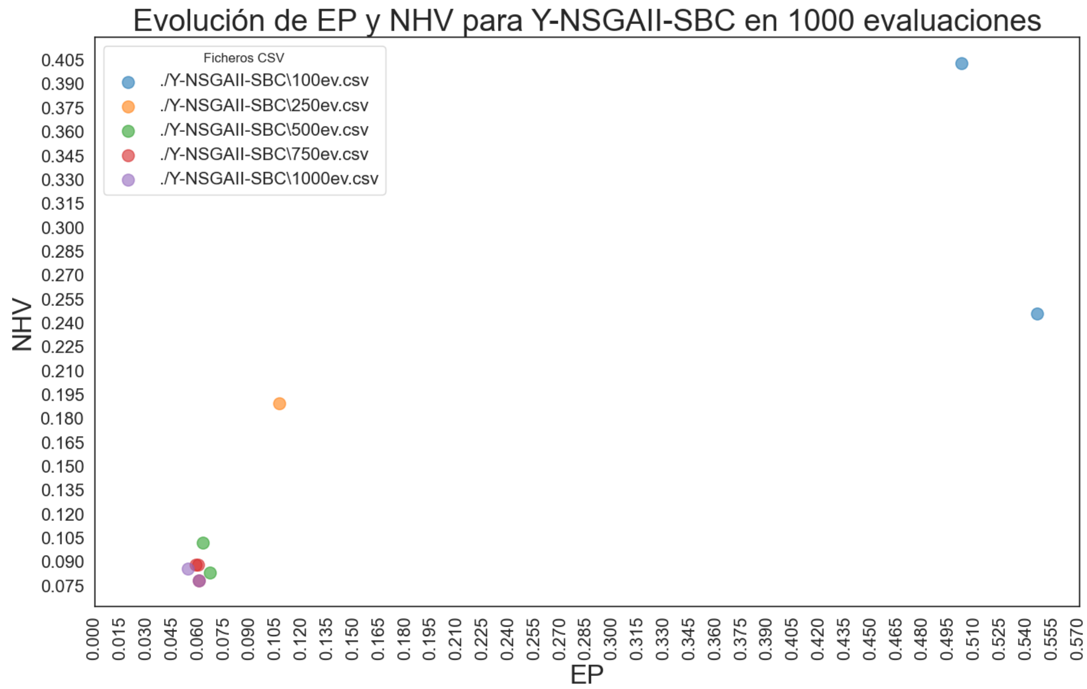

Figura 22: Problema DTLZ1 en la evaluación 1000 con Y-NSGAII-SBC

Para poder ver fácilmente qué algoritmo obtiene mejores soluciones iniciales, vamos a fusionar en un único gráfico las soluciones obtenidas a las 100 evaluaciones de las figuras 20, 21 y 22:

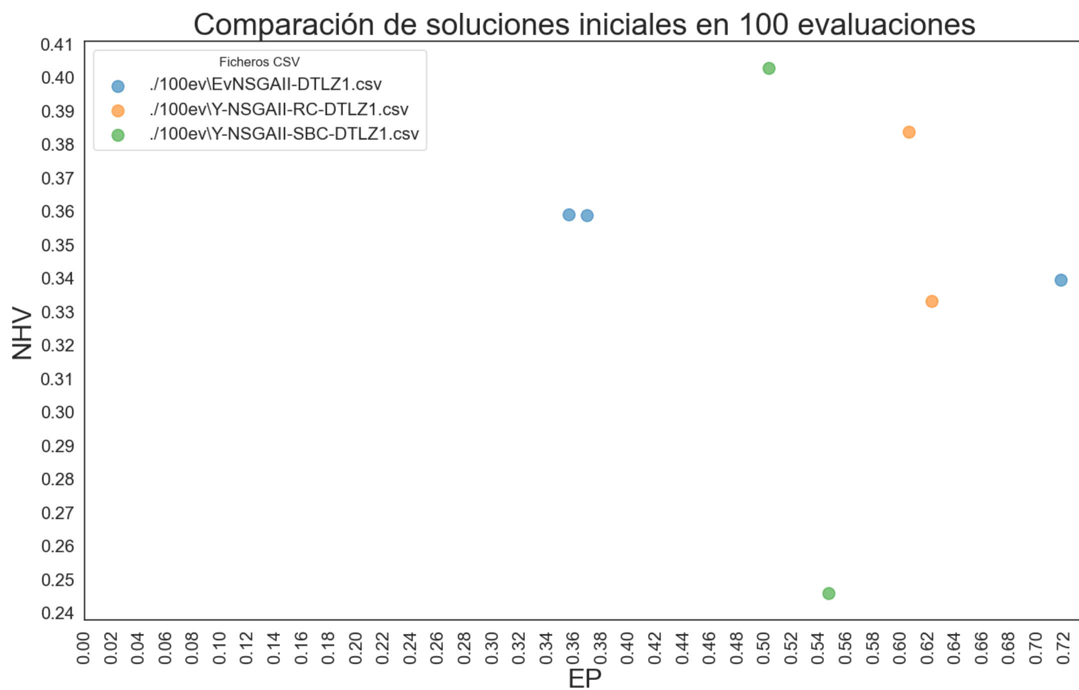

Figura 23: Comparación de las soluciones iniciales de EVNGSAII, Y-NSGAII-RC y Y-NSGAII.SBC.

Podemos ver claramente cómo en NHV gana el algoritmo Y-NSGAII-SBC con una ventaja de alrededor de 0.1 unidades respecto a Y-NSGAII-RC, el segundo mejor. En cuanto a EP, hay un claro ganador, el algoritmo EvNSGAII, siendo Y-NSGAII-SBC el segundo mejor.

A su vez, vamos a representar qué algoritmo obtiene la mejor solución a las 1000 evaluaciones. Se va a seguir el mismo enfoque que en la figura 23.

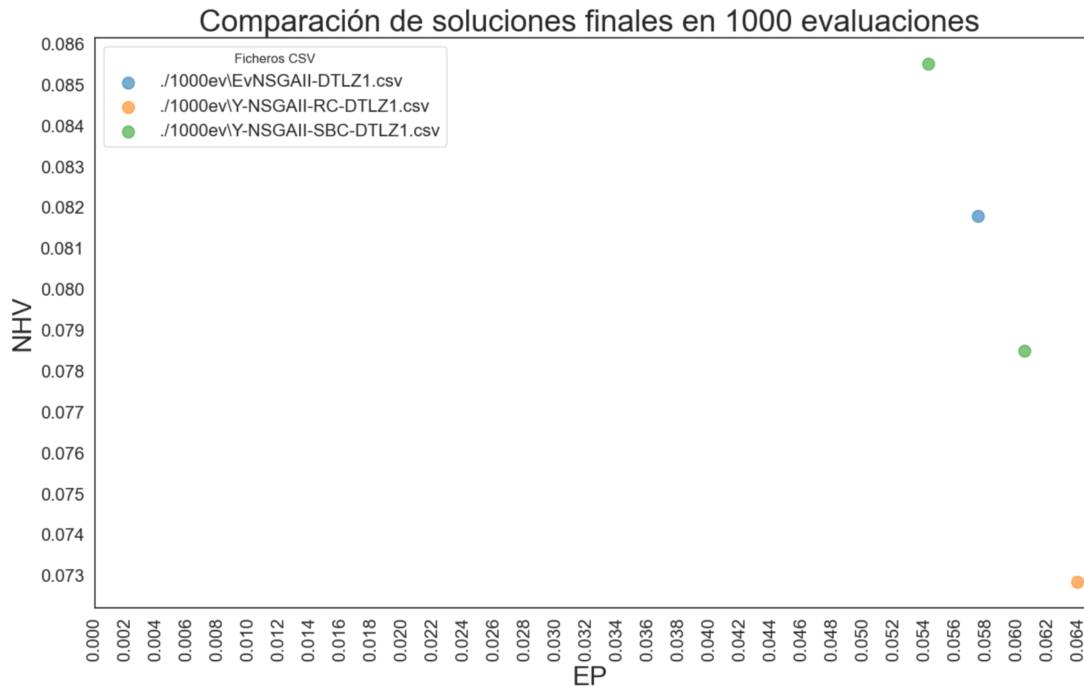

Figura 24: Comparación de las soluciones finales de EVNGSAII, Y-NSGAII-RC y Y-NSGAII.SBC.

En este caso, en la figura 24. correspondiente a la evaluación 1000, el algoritmo que mejores resultados ha obtenido en cuanto a NHV es Y-NSGAII-RC, seguido de Y-NSGAII-SBC. En EP, el mejor resultado ha sido obtenido por el algoritmo Y-NSGAII-SBC.

7.4. Comparativa de las mejores configuraciones encontradas

El resultado de ejecutar un algoritmo de meta-optimización son configuraciones del algoritmo base que deberían resolver todos los problemas del conjunto de entrenamiento (los DTLZ) de forma eficaz.

Dado que se parte del algoritmo NSGA-II en su configuración estándar como referencia, a continuación se va a configurar NSGA-II con las configuraciones obtenidas por EvNSGA-

II, Y-NSGA-RC e Y-NSGAII-SBC. Para ello se ha llevado a cabo un estudio experimental en que cada par (algoritmo, problema) se ha ejecutado 25 veces y se han calculado los valores de EP y NHV para cada frente obtenido.

En las siguientes tablas, las filas corresponden a cada uno de los siete problemas y las columnas a los algoritmos. Por cada fila, la celda que tiene un color más oscuro corresponde al mejor valor y la que se encuentra con un color algo menos oscuro es el segundo mejor valor.

Tabla 1: EP. Median and Interquartile Range

	NSGAII	EvNSGAII	Y-NSGAII-RC	Y-NSGAII-SBC
DTLZ1	$1{,}37e-01_{4{,}2e-02}$	$7{,}52e-02_{1{,}7e-02}$	$8{,}00e-02_{1{,}5e-02}$	$7{,}04e-02_{9{,}9e-03}$
DTLZ2	$1{,}17e-01_{3{,}2e-02}$	$7{,}88e-02_{1{,}2e-02}$	$7{,}63e-02_{1{,}7e-02}$	$7{,}66e-02_{1{,}1e-02}$
DTLZ3	$1{,}78e+00_{3{,}1e+00}$	$1{,}22e-01_{7{,}1e-02}$	$1{,}75e-01_{1{,}1e-01}$	$1{,}38e-01_{4{,}2e-02}$
DTLZ4	$1{,}09e-01_{2{,}4e-02}$	$7{,}27e-02_{1{,}1e-02}$	$7{,}42e-02_{9{,}3e-01}$	$7{,}17e-02_{8{,}7e-03}$
DTLZ5	$1{,}33e-02_{3{,}2e-03}$	$8{,}19e-03_{4{,}9e-04}$	$8{,}69e-03_{4{,}3e-04}$	$8{,}29e-03_{5{,}8e-04}$
DTLZ6	$7{,}82e-01_{9{,}0e-02}$	$7{,}31e-03_{8{,}2e-04}$	$8{,}06e-03_{1{,}0e-03}$	$7{,}17e-03_{1{,}1e-03}$
DTLZ7	$8{,}87e-02_{1{,}6e-02}$	$5{,}62e-02_{1{,}4e-02}$	$3{,}75e-01_{1{,}9e-01}$	$5{,}18e-02_{1{,}1e-02}$

La tabla 1 representa la mediana y el rango intercuartílico obtenidos por cada uno de los problemas resueltos por los distintos algoritmos para el indicador EP. Al observar la tabla, el algoritmo Y-NSGAII-SBC ocupa el primer puesto en cuatro de los siete problemas y el segundo puesto en tres. Por otro lado, EvNSGAII gana en dos de los siete problemas y ocupa el segundo lugar en dos de los casos. A su vez, el algoritmo NSGAII tradicional no desempeña muy bien en este aspecto.

Si tuviésemos que quedarnos con uno de los cuatro algoritmos, según los resultados del indicador EP, sería el Y-NSGAII-SBC, con codificación *YAML* y cruce real (SBX y BLX ALPHA) ya que es el que más veces ha ocupado el primer puesto y, en el peor de los casos, ha quedado segundo. Sin embargo, para determinar qué algoritmo es el que mejor rendimiento ofrece hay que analizar también los resultados del indicador NHV y, además, aplicar herramientas y tests estadísticos sobre los resultados.

Tabla 2: NHV. Median and Interquartile Range

	NSGAII	EvNSGAII	Y-NSGAII-RC	Y-NSGAII-SBC
DTLZ1	$1{,}02e-01_{3{,}2e-02}$	$6{,}92e-02_{3{,}0e-02}$	$6{,}92e-02_{2{,}2e-02}$	$5{,}96e-02_{1{,}2e-02}$
DTLZ2	$1{,}98e-01_{1{,}5e-02}$	$1{,}16e-01_{3{,}6e-03}$	$1{,}12e-01_{3{,}5e-03}$	$1{,}19e-01_{4{,}3e-03}$
DTLZ3	$1{,}00e+00_{0{,}0e+00}$	$3{,}20e-01_{3{,}0e-01}$	$4{,}47e-01_{4{,}0e-01}$	$4{,}28e-01_{2{,}1e-01}$
DTLZ4	$1{,}16e-01_{1{,}4e-02}$	$3{,}31e-02_{2{,}9e-03}$	$3{,}18e-02_{9{,}7e-01}$	$3{,}58e-02_{3{,}4e-03}$
DTLZ5	$2{,}83e-02_{3{,}4e-03}$	$2{,}23e-02_{8{,}3e-04}$	$2{,}17e-02_{1{,}3e-03}$	$2{,}29e-02_{7{,}4e-04}$
DTLZ6	$1{,}00e+00_{0{,}0e+00}$	$1{,}02e-02_{7{,}8e-04}$	$8{,}29e-03_{7{,}5e-04}$	$9{,}90e-03_{4{,}8e-04}$
DTLZ7	$1{,}35e-01_{1{,}5e-02}$	$6{,}03e-02_{5{,}8e-03}$	$1{,}88e-01_{8{,}1e-02}$	$6{,}09e-02_{3{,}9e-03}$

Los resultados obtenidos del NHV en la tabla 2 varían respecto a la tabla 1. En este caso, el algoritmo Y-NSGAII-RC, con cruce aleatorio, gana en cuatro ocasiones. El EvNSGAII gana en dos ocasiones y queda cuatro veces segundo, resultando ser el más constante de los cuatro con diferencia.

A la hora de determinar cuál de los dos algoritmos es el mejor en este aspecto, es más complicado, ya que, por un lado, hay uno que ha ganado cuatro veces, pero el otro ha quedado primero y/o segundo en seis de los siete problemas. Por ello, el mejor algoritmo se decidiría en función de nuestros intereses propios; es decir, hemos de tener claro si valoramos más que un algoritmo gane o que sea más constante y regular.

7.5. Comparativa de algoritmos empleando SAES

Como se ha comentado en la sección anterior, para determinar qué algoritmo es el que mejor resultado ha proporcionado en promedio en los siete problemas DTLZ, además de usar tablas de medianas y rangos intercuartílicos, hay que usar tests estadísticos que permitan un análisis más preciso de los resultados. Para usar estos análisis se ha hecho uso de la herramienta SAES[1], desarrollada en la Universidad de Málaga. En los siguientes tests se van a comparar los distintos algoritmos en base a su NHV.

La tabla 3 muestra los resultados del test de suma de rangos de Wilcoxon, que compara pares de algoritmos aplicados a un determinado problema. Si el valor p es menor que 0,05, se puede afirmar que las diferencias entre los algoritmos comparados son estadísticamente significativas.

[1]SAES: https://github.com/jMetal/SAES

Tabla 3: Wilcoxon rank sum test table

	EvNSGAII	Y-NSGAII-RC	Y-NSGAII-SBC
NSGAII	-------	---=--+	-------
EvNSGAII		=-=--+	=+=++==
Y-NSGAII-RC			=+==++-

En la tabla 3 los resultados se presentan de la siguiente forma. En cada celda encontraremos siete signos (-, + y =), uno por cada problema. El primero sería el correspondiente al problema DTLZ1 y el último al DTLZ7. Los resultados se interpretan de la siguiente manera:

- - : el algoritmo de la columna es mejor que el de la fila.
- + : el algoritmo de la fila es mejor que el de la columna.
- = : no se puede determinar cuál de los dos algoritmos es mejor.

Los resultados son claros en el caso de NSGAII, es el peor que el resto en prácticamente todos los casos (sólo hay dos excepciones). Y-NSGAII-RC prácticamente gana en todos sus enfrentamientos, por tanto, resulta ser el claramente mejor. En segundo puesto quedaría el EvSNSGAII. Si comparamos EvNSGAII e Y-NSGAII-RC observamos que en dos problemas las diferencias no son significativas, en tres lo son a favor de Y-NSGAII-RC y en uno es mejor EVNSGA-II. En el caso de Y-NSGA-II-RC e Y-NSGAII-SBC, en tres problemas no hay diferencias, en tres Y-NSGAII-SBC es mejor según el test y sólo en un problema Y-NSGAII-SBC es mejor estadísticamente.

El uso del test de Wilcoxon nos permite analizar en detalle las diferencias entre los algoritmos, pero puede no ser el adecuado si queremos saber en promedio qué algoritmo es el que obtiene mejor rendimiento. Para ello se puede usar el gráfico de distancia crítica, incluido en SAES.

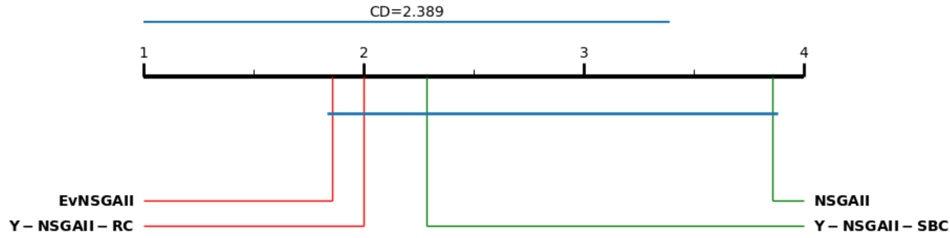

Figura 25: Critical distance ranking

El gráfico 25 muestra el resultado del ranking de distancia crítica, en el que los algoritmos que obtengan mejores resultados se situarán orientados a números más bajos (hacia la izquierda) Según el gráfico, el mejor es el EvNSGAII, seguido de Y-NSGAII. Sin embargo, no se puede determinar cuál de los dos es mejor estadísticamente, ya que la distancia que los separa es menor que la distancia crítica, que el test determina que es 2,389; es decir, debido a que los dos algoritmos se encuentran demasiado próximos, estadísticamente no se puede determinar cuál es el mejor. Según el ranking, los dos peores algoritmos son el Y-NSGAII-SBC y el NSGAII tradicional.

Por último, para determinar la distribución de los valores de los indicadores de calidad en cada una de las 25 ejecuciones independientes, se van a usar gráficos de caja (boxplot) generados por la herramienta SAES. Cada caja incluye la mediana y está determinada por los valores del primer y tercer cuartil, apareciendo los valores anormalmente altos o bajos (outliers) como círculos.

En los gráficos que se van a mostrar se incluyen, para cada problema, los boxplots de cada algoritmo. En general, si las cajas de dos algoritmos no se solapan, se puede afirmar que, estadísticamente, las diferencias entre los algoritmos son significativas. En este sentido, se podría decir que los boxplots permiten ver de forma visual lo que el test de Wilcoxon devuelve como un valor.

La figura 26 permite ver que los resultados de todos los algoritmos están muy agrupados. Si nos centramos en el solape de las cajas, se aprecia que la de NSGA-II está por encima del resto, lo que confirma que es peor de forma significativa. Si se observa la tabla 3, en la primera fila, correspondiente a NSGA-II, el primer símbolo es - en todos los casos. Por otro lado, las cajas de los otros algoritmos aparecen muy solapadas, por lo que las diferencias entre ellos no deben ser estadísticamente significativas, lo que se verifica en la tabla del test de Wilcoxon, en la que se puede ver que las posiciones correspondientes

al problema DTLZ1 el símbolo es =.

Se comentará el gráfico para el problema ZDT2 (figura 27) y ZDT5 (figura 30):

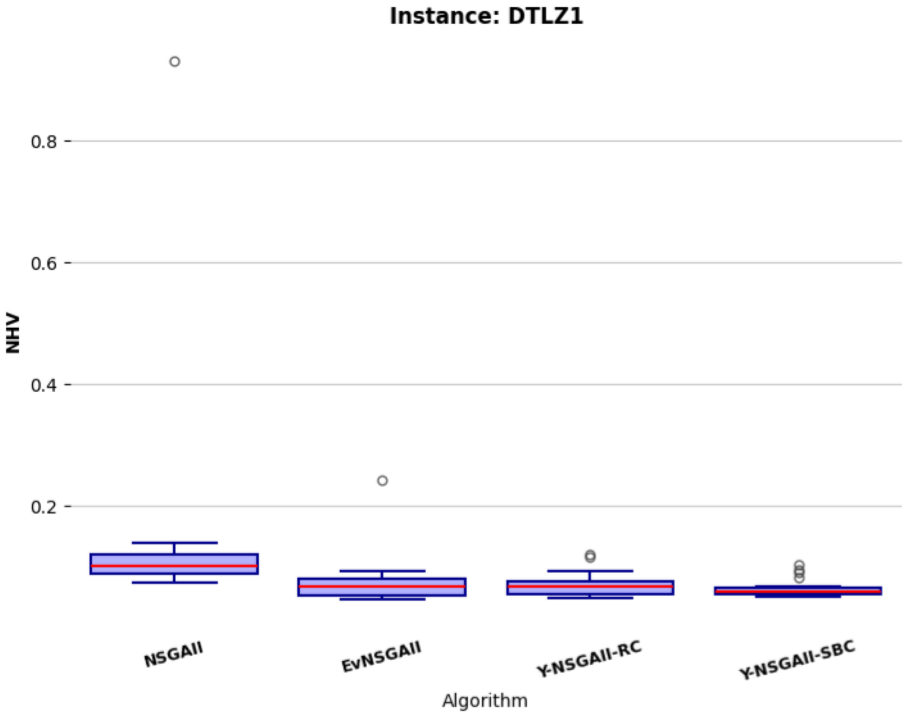

Figura 26: Boxplot para DTLZ1

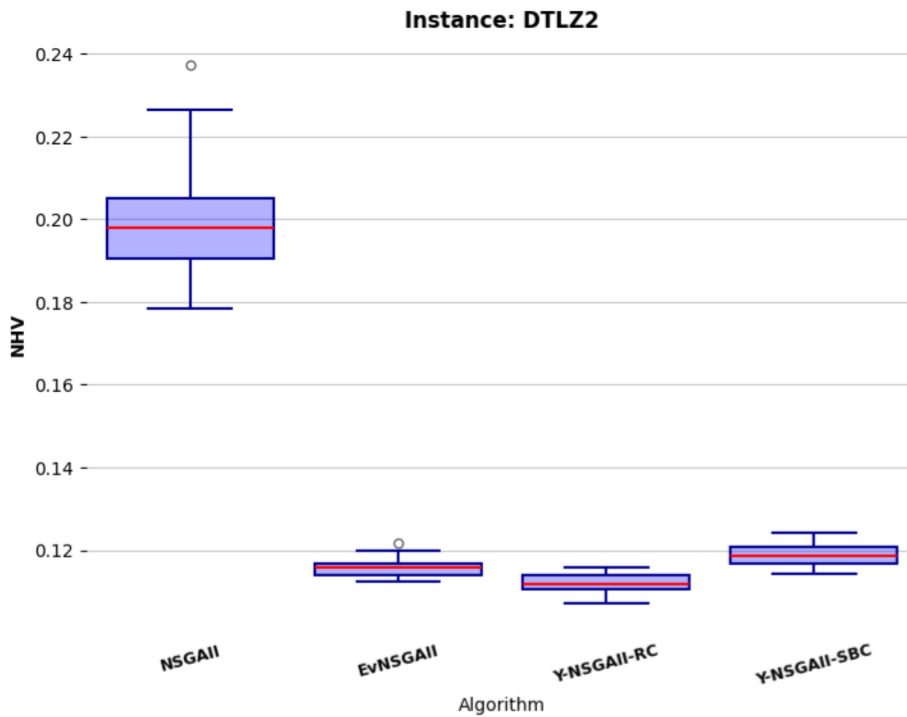

Figura 27: Boxplot para DTLZ2

Al igual que en el resto de ejemplos, el objetivo es minimizar el NHV, es decir, que su valor sea el más cercano posible a cero. En la figura 27 podemos observar cómo el peor algoritmo, al igual que en muchos de los otros casos, es el NSGAII tradicional. Su rango de NHV se encuentra muy por encima del resto y su mediana (línea horizontal roja) se encuentra en torno a 0.2. Los otros tres algoritmos restantes se encuentran mucho más igualados. Podríamos considerar que el mejor resultado es el algoritmo Y-NSGAII-RC, ya que se encuentra ligeramente por debajo del resto. No obstante, al solaparse los resultados (la figura morada) no podemos determinar con certeza cuál es la mejor solución.

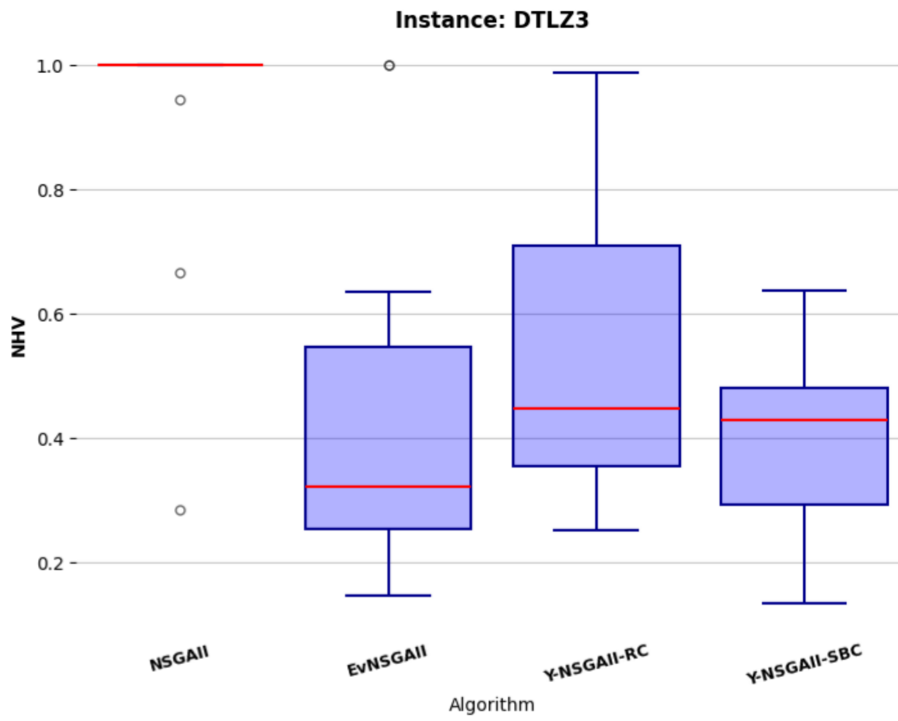

Figura 28: Boxplot para DTLZ3

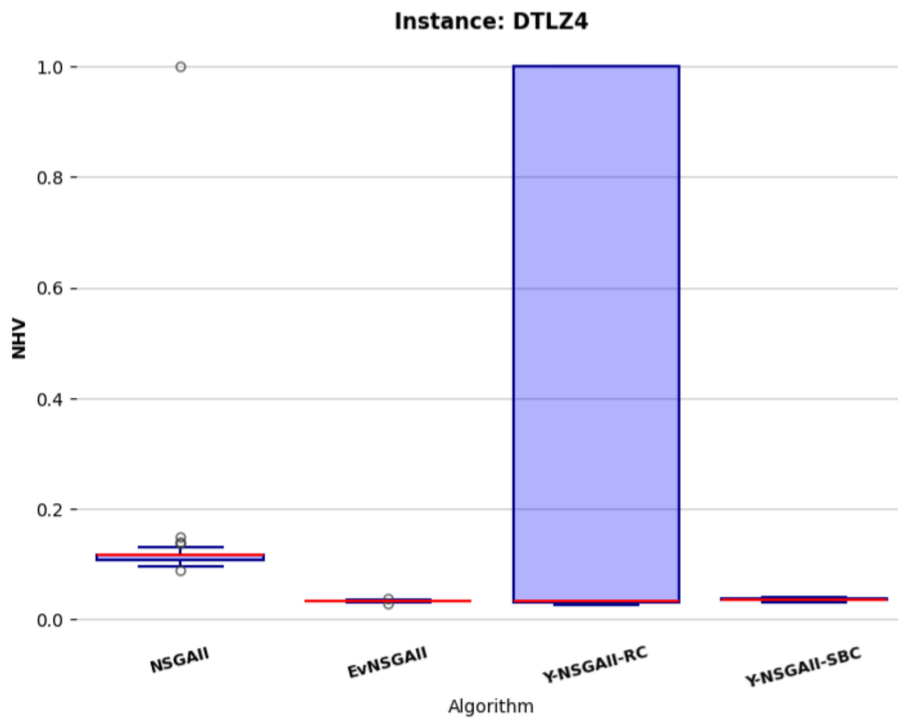

Figura 29: Boxplot para DTLZ4

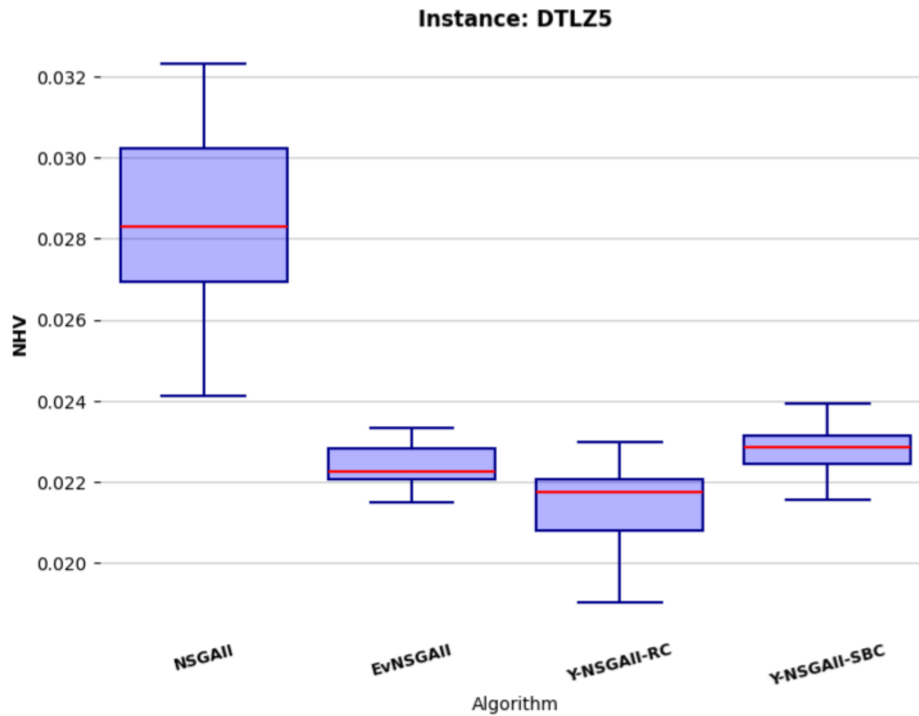

Figura 30: Boxplot para DTLZ5

Este gráfico refleja un caso claro en el que sí encontramos un algoritmo mejor que el resto. En la figura 30 se puede ver cómo el peor resultado es, igual que en la figura 27, el NSGAII original, ya que su área y mediana se encuentran más hacia arriba. No obstante, si nos fijamos en los valores de NHV (eje de coordenadas) del EvNSGAII e Y-NSGAII-SBC, veremos que la diferencia es bastante pequeña, son muy similares. Por ello, no podemos determinar cuál de los dos es mejor. En este caso, sí podemos garantizar y determinar la existencia de un algoritmo mejor, el Y-NSGAII-RC. Su área se encuentra por debajo y sin solapar respecto al resto. Lo mismo ocurre con su mediana.

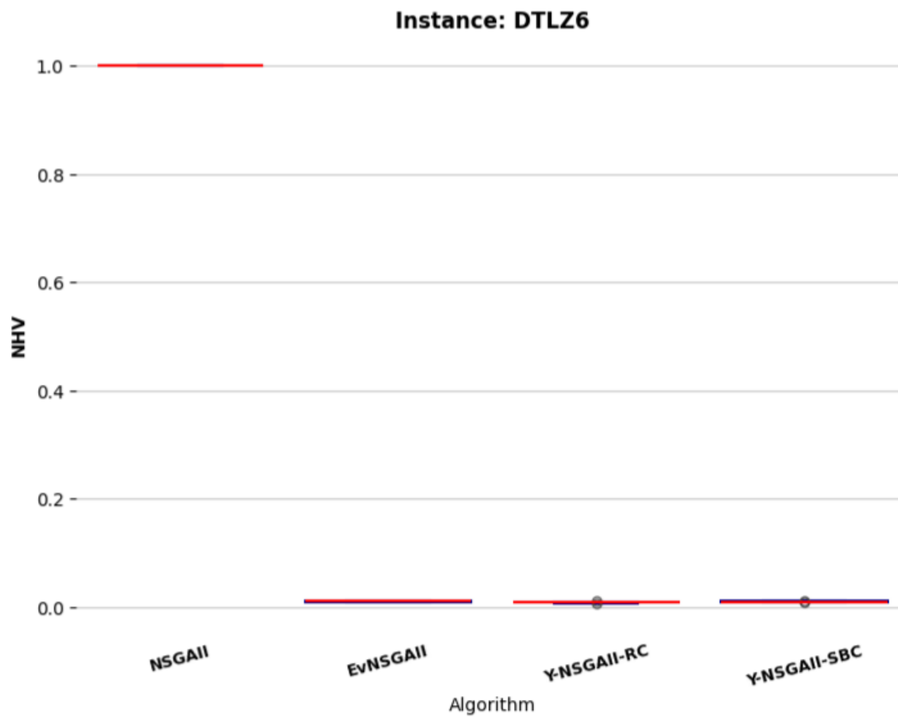

Figura 31: Boxplot para DTLZ6

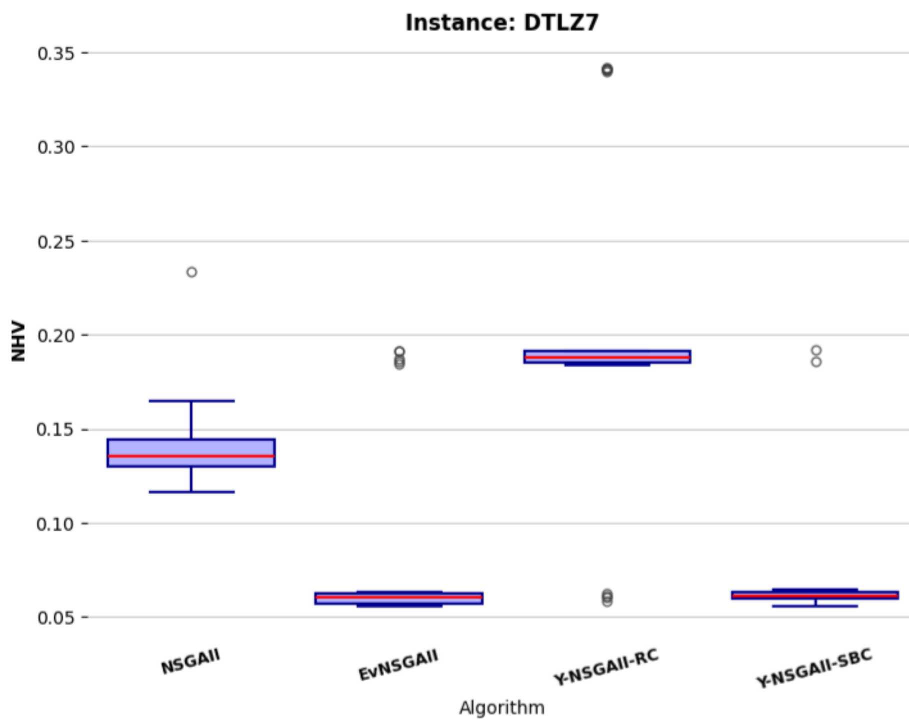

Figura 32: Boxplot para DTLZ7

8. Conclusiones y Líneas Futuras

8.1. Conclusiones

Los objetivos de este Trabajo de Fin de Grado han sido cumplidos. Se ha conseguido mejorar y ampliar la herramienta *Evolver*, que ha dado lugar a *REvolver*. Se ha conseguido generar automáticamente los códigos de las clases *ConfigurableNSGAII*, *EvNSGAIIDoubleProblem* y *EvNSGAIIPermutationProblem* en función de un archivo *YAML*, en donde queda recogida la configuración del algoritmo NSGAII.

Por otro lado, se ha conseguido implantar en Evolver una nueva codificación, alternativa a la actual, de arrays de valores reales, basada en una codificación en árbol de los parámetros. Posteriormente, se ha podido adaptar Evolver a esta nueva codificación y evaluar la misma, siendo enfrentada con la codificación ya existente.

Además, se ha comprobado que los objetivos son perfectamente compatibles entre sí; se puede emplear la clase generada *GeneratedEvNSGAIIDoubleProblem* con la codificación en árbol.

8.2. Líneas Futuras

Pese a que se han cumplido los objetivos planteados, la temática del proyecto realizado se inscribe en una línea de investigación que tiene un recorrido cuyo alcance queda fuera de un TFG. Entre los aspectos a abordar, habría que seleccionar y experimentar con una gran cantidad de familias de problemas para realmente evaluar la calidad de la codificación creada. Además, las evaluaciones se deberían hacer de una manera más exhaustiva, aumentando notablemente el número de ellas. Por otro lado, sería recomendable modificar valores del espacio de parámetros para comprobar la posibilidad de obtener soluciones de mayor calidad. En el trabajo se ha trabajado con el algoritmo NSGA-II, pero en el futuro se podrían incluir otros algoritmos de optimización multi-objetivo representativos, como MOEA/D o SMS-EMOA.

Una línea de trabajo que surge del proyecto tiene que ver con la generación de una gran cantidad de información sobre configuraciones que se generan al ejecutar los algoritmos de meta-optimización. Analizar dichas configuraciones podría dar lugar a un estudio analítico enfocado en determinar qué parámetros son los más influyentes en determinados casos, lo que permitiría diseñar algoritmos más eficientes.

Referencias

José F. Aldana-Martín, Juan J. Durillo, and Antonio J. Nebro. Evolver: Meta-optimizing multi-objective metaheuristics. *Software X*, 1(1):1–8, 2023.

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

Gabriel Kronberger Christoph Neumüller, Stefan Wagner and Michael Affenzeller. Parameter Meta-optimization of Metaheuristic Optimization Algorithms. *Heuristic and Evolutionary Algorithms Laboratory*, 1(1):367–374, 2012.

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.

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.

K. Deb, L. Thiele, M. Laumanns, and E. Zitzler. Scalable test problems for evolutionary multiobjective optimization. In Ajith Abraham, Lakhmi Jain, and Robert Goldberg, editors, *Evolutionary Multiobjective Optimization. Theoretical Advances and Applications*, pages 105–145. Springer, USA, 2005.

Kalyanmoy Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley Sons, Inc., USA, 2001. ISBN 047187339X.

J.J. Durillo and A.J. Nebro. jmetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760 – 771, 2011. ISSN 0965-9978. doi: 10.1016/j.advengsoft.2011.05.014.

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. ISBN 978-1-4503-3488-4.

Aimin Zhou, Bo-Yang Qu, Hui Li, Shi-Zheng Zhao, Ponnuthurai Nagaratnam Suganthan,

and Qingfu Zhang. Multiobjective evolutionary algorithms: A survey of the state of the art. *Swarm and Evolutionary Computation*, 1(1):32 – 49, 2011. ISSN 2210-6502.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

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