Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Othimi-Algorithm: Interfaz para modelado de metaheurísticas compatibles con jMetal

Othimi-Algorithm: Interface for modeling metaheuristics compatible with jMetal.

Daniel Hernández Fajardo

La Laguna, 14 de julio de 2023

Dª. **Gara Miranda Valladares**, con N.I.F. 78.563.584-T profesora Titular de Universidad adscrita al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutora.

D. **Daniel del Castillo de la Rosa**, con N.I.F. 51.154.908-X graduado en Ingeniería Informática, como cotutor.

C E R T I F I C A (N)

Que la presente memoria titulada:

"Othimi-Algorithm: Interfaz para modelado de metaheurísticas compatibles con jMetal"

ha sido realizada bajo su dirección por D. **Daniel Hernández Fajardo**, con N.I.F. 51.148.249-K.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 14 de julio de 2023

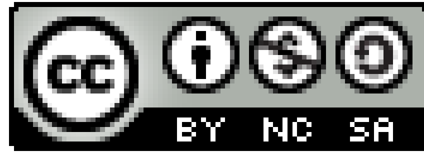
// Agradecimientos

En primer lugar, me gustaría expresar mi profundo agradecimiento a mi tutora, Gara, y a mi cotutor Daniel del Castillo de la Rosa, por guiarme y brindarme apoyo durante todo el desarrollo del Trabajo de Fin de Grado.

También quiero agradecer a todos los profesores que me han hecho disfrutar de la carrera y me han motivado para seguir aprendiendo en el futuro. No me quiero olvidar de agradecer a todos los amigos que he conocido en la carrera, los cuales me han apoyado durante estos años y me han hecho pasar muy buenos momentos.

Por último quiero agradecer a mi familia, por estar en los buenos y malos momentos y creer en mí desde el principio.

Licencia



© Esta obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional.

Resumen

Actualmente, existe una gran cantidad de investigaciones respecto a los algoritmos de optimización bioinspirados, sin embargo, estos algoritmos son conocidos por ser de implementar y de adaptar al problema en cuestión, lo que limita su uso fuera del ámbito académico. Por esto nació originalmente Prodef, una solución web con el objetivo de que su interfaz, permita a cualquier usuario modelar algoritmos y problemas de optimización de manera sencilla y agradable visualmente, sin programar o realizar configuraciones complejas. Esta interfaz ha sido desarrollada a partir de la biblioteca Blockly, que implementa una interfaz gráfica basada en bloques para la creación de problemas y algoritmos, pudiendo mover y unir los bloques como si fuera un puzzle.

Prodef presentaba varios problemas en la propia herramienta e interfaz, lo que ocasionaba que no cumpliese con los estándares de calidad esperados. Con el fin de tener una herramienta con mejor usabilidad, se creó Othimi, una nueva solución web creada desde cero para solucionar los problemas existentes en Prodef. Othimi actualmente tiene implementado el modelado de problemas, sin embargo, carece de la sección de modelado de algoritmos.

En este trabajo, se ha implementado en Othimi la capacidad para que los usuarios definan sus propios algoritmos, utilizando bloques basados en componentes de jMetal. Esto se realiza basándose en la idea del diseño de algoritmos planteada en Prodef, en la que los algoritmos están compuestos por componentes que tienen dependencias llamadas parámetros. Se quiere aprovechar las funcionalidades del paquete jMetal-auto, que permite el desarrollo de algoritmos a partir de componentes y la configuración automática mediante una cadena de caracteres. Para ello se ha realizado un estudio exhaustivo de los componentes y parámetros de jMetal, que son necesarios en los algoritmos de jMetal-auto. Posteriormente, se crearon los bloques, los cuales están basados en componentes de jMetal. Estos bloques permiten al usuario especificar los parámetros y componentes de los algoritmos, siguiendo plantillas predefinidas con la misma estructura de los algoritmos implementados en jMetal-auto. Sin embargo, el uso de la plantilla solo es necesario si se quiere configurar un algoritmo y obtener un archivo para jMetal-auto, ya que Othimi permite a los usuarios poder crear sus propios algoritmos de manera manual con los nuevos bloques, sin la necesidad de utilizar una estructura estricta como la de los algoritmos de jMetal, debido al carácter general de la mayoría de bloques. Para las plantillas, la interfaz de Othimi ofrece la opción de exportar un archivo ejecutable para jMetal-auto, que contiene la configuración del algoritmo especificada por el usuario. A diferencia de Prodef, en Othimi la especificación de parámetros se realiza al mismo tiempo que el modelado del algoritmo, añadiendo a los distintos bloques, campos con sus parámetros necesarios.

Con los cambios e implementaciones realizadas en Othimi, se ofrece al usuario la posibilidad de modelar sus propios algoritmos de manera manual o usando plantillas predefinidas, las cuales es posible exportar para su ejecución con jMetal-auto, ofreciendo un mayor abanico de posibilidades para trabajar con la optimización combinatoria. Todo esto acerca a Othimi a su objetivo de poder facilitar al público general y a las empresas el uso de algoritmos de optimización bioinspirados, llevándolos fuera del ámbito académico.

Palabras clave: Algoritmos bioinspirados, optimización combinatoria, modelado de algoritmos, Othimi, Prodef, jMetal, jMetal-auto, Blockly

Abstract

Currently, there is a considerable amount of research on bioinspired optimization algorithms, however, these algorithms are known for their complexity and difficulty in implementation and adaptation to specific problems, which limits their usage outside the academic field. This is why Prodef was originally created as a web solution with the objective of providing users with an interface that allows for easy and visually appealing modeling of optimization problems and algorithms without the need for programming or complex configurations. The interface is developed using the Blockly library, which implements a block-based graphical interface for the creation of problems and algorithms, enabling users to move and connect blocks as if solving a puzzle.

Prodef had several issues with its tool and interface, which prevented it from meeting the expected quality standards. In order to improve usability, Othimi was created as a completely new web solution to address the problems encountered in Prodef. Currently, Othimi includes the functionality for problem modeling but lacks the algorithm modeling section.

In this work, the capability for users to define their own algorithms has been implemented in Othimi using blocks based on jMetal components. This implementation builds upon the algorithm design concept introduced in Prodef, where algorithms are composed of components that have dependencies known as parameters. The aim is to use the functionalities provided by the jMetal-auto package, which enables algorithm development using components and automatic configuration through character strings. To achieve this, an extensive study of jMetal components and parameters required for jMetal-auto algorithms was made. Then, blocks were created based on jMetal components, allowing users to specify the parameters and components of their algorithms using predefined templates that follow the structure of jMetal-auto algorithms.However, the use of templates is only necessary if users want to configure an algorithm and obtain a file for jMetal-auto, as Othimi allows users to manually create their own algorithms using the new blocks without adhering to the strict structure of jMetal algorithms. This flexibility is made possible due to the general nature of most blocks. For templates, the Othimi interface offers the option to export an executable file for jMetal-auto, containing the user-specified algorithm configuration. Unlike Prodef, in Othimi, parameter specification is done simultaneously with algorithm modeling by adding fields for the necessary parameters to the respective blocks.

With the changes and implementations made in Othimi, users now have the option to model their own algorithms manually or use predefined templates that can be exported for execution with jMetal-auto, thereby providing a wider range of possibilities for working with combinatorial optimization. All of these advancements bring Othimi closer to its objective of facilitating the use of bioinspired optimization algorithms by the general public and businesses, extending their application beyond the academic field.

Keywords: Bio-inspired algorithms, combinatorial optimization, algorithm modeling, Othimi, Prodef, jMetal, jMetal-auto, Blockly

Índice general

1. Introducción **1**
1.1. Antecedentes . . . 1
1.2. Objetivos a realizar . . . 3

2. Introducción a las metaheurísticas **4**
2.1. Metaheurísticas . . . 4
2.2. Tipos de metaheurísticas . . . 4
2.3. Algoritmos evolutivos . . . 5
2.3.1. Elementos de un algoritmo evolutivo . . . 5
2.4. Optimización por enjambre de partículas (PSO) . . . 7
2.4.1. Elementos del PSO . . . 8
2.5. Algoritmos multiobjetivo . . . 8

3. Othimi **10**
3.1. Instalación y puesta en marcha . . . 10
3.2. Estado actual del *front-end* . . . 11
3.2.1. Autenticación de usuario . . . 11
3.2.2. Modelado de problemas . . . 13
3.2.3. Creación de instancias . . . 14
3.3. Estado actual del *back-end* . . . 14
3.4. Estado del resolutor de Rust- . . . 15

4. jMetal y componentes para jMetal-auto **16**
4.1. Algoritmos basados en componentes . . . 16
4.2. jMetal-auto . . . 17
4.2.1. Componentes de jMetal-auto y sus parámetros . . . 18
4.3. Descripción de los componentes y sus parámetros para el NSGAII y el MOPSO 21
4.3.1. Componentes de inicialización . . . 21
4.3.2. Componentes de evaluación y terminación . . . 22
4.3.3. Componentes de selección . . . 22
4.3.4. Componentes de variación . . . 24
4.3.5. Componentes de posición y velocidad en el MOPSO . . . 26

5. Modelado de algorimos con Blockly **27**
5.1. Blockly . . . 27
5.2. Implementación de los bloques en la interfaz gráfica . . . 28
5.2.1. Clasificación de los bloques . . . 29
5.2.2. Modificación dinámica y limitación de parámetros . . . 29
5.2.3. Bloques desarrollados . . . 32

5.3. Plantillas predefinidas 35
5.4. Modelado manual de algoritmos 37

6. Ejecución de un algoritmo con jMetal-auto **39**
6.1. Obtener un archivo ejecutable con jMetal a partir de Prodef 39
6.2. Ejecución con jMetal 41
6.2.1. Instalación de jMetal 41
6.2.2. Ejecución del archivo 42

7. Conclusiones y líneas futuras **45**
7.1. Conclusiones 45
7.2. Líneas futuras 46

8. Summary and Conclusions **47**
8.1. Conclusions 47
8.2. Future lines of work 48

9. Presupuesto **49**

Índice de Figuras

2.1. Proceso algoritmo evolutivo 5
2.2. Diagrama PSO 7

3.1. Marca Othimi 10
3.2. Zona de trabajo de Othimi 11
3.3. Interfaz de autentificación 12
3.4. Pantalla de registro con email 12
3.5. Pantalla de inicio de sesión con email 13
3.6. Plantilla por defecto del problema Knapsack 13
3.7. Filtro de problemas 14
3.8. Pantalla de creación de instancias 14

4.1. Aclaración sobre los colores 19

5.1. Campos desplegables y numéricos 28
5.2. Enter Caption 31
5.3. Bloque de terminación 32
5.4. Bloques de inicialización 32
5.5. Bloque de evaluación 33
5.6. Bloques de selección 33
5.7. Bloques de mutación y cruce 34
5.8. Bloques de actualización de posición, velocidad y mejores soluciones 35
5.9. Selección de la plantilla de algoritmo 36
5.10 Plantilla algoritmo evolutivo 36
5.11 Algoritmo memético en Prodef 37
5.12 Algoritmo memético en Othimi 38

6.1. Plantilla algoritmo evolutivo 40
6.2. Botón para exportar el fichero 41
6.3. Extensión para Java 42
6.4. Botón para ejecutar 43
6.5. Ejecución del algoritmo 43
6.6. Soluciones al problema 44

Índice de Tablas

1.1. Objetivos del proyecto 3

4.1. Estructura del proyecto Maven de jMetal 16
4.2. Componentes de inicialización para NSGAII. 21
4.3. Parámetros de los componentes de inicialización para NSGAII 21
4.4. Componentes de inicialización para MOPSO 21
4.5. Parámetros de los componentes de inicialización para MOPSO 22
4.6. Componentes de evaluación y terminación. 22
4.7. Parámetros de los componentes de evaluación y terminación 22
4.8. Componentes de selección para NSGAII 23
4.9. Parámetros de los componentes de selección para NSGAII 23
4.10. Componentes de selección para MOPSO 23
4.11. Parámetros de los componentes de selección para MOPSO 24
4.12. Componentes de variación para NSGAII. 24
4.13. Parámetros de los componentes de variación para NSGAII 25
4.14. Componentes de variación para MOPSO. 25
4.15. Parámetros de los componentes de variación para MOPSO 25
4.16. Componentes de posición y velocidad. 26
4.17. Parámetros de los componentes de posición y velocidad 26

9.1. Presupuesto 49

Capítulo 1

Introducción

En la investigación actual, el diseño e implementación de algoritmos evolutivos y otras metaheurísticas bioinspiradas son muy utilizados. Estas técnicas son valiosas para resolver problemas complejos de optimización. Para las personas sin muchos conocimientos en este campo y que deseen utilizar estos algoritmos, no es necesario implementarlos desde cero, ya que existen diversas herramientas que facilitan su uso. Un ejemplo de este tipo de herramientas sería jMetal [16], el cual es un *framework* desarrollado en Java y que es utilizado para la experimentación con algoritmos de optimización multiobjetivos.

No obstante, estas herramientas suelen estar diseñadas para adaptarse a un tipo específico de técnica, lo que puede limitar su flexibilidad. Además, si un usuario que no es experto quiere utilizar estas herramientas, deberá aplicar distintos tipos de técnicas dependiendo del problema a resolver. También hay que tener en cuenta que el usuario debe de tener un mínimo de conocimientos sobre programación y comprender la herramienta para utilizarla y configurar los parámetros necesarios para la ejecución.

Ante esta situación, surge la necesidad de crear una herramienta con el objetivo de ayudar a los usuarios inexpertos a utilizar diversas técnicas de optimización de manera más sencilla. Es necesario una interfaz amigable y sencilla que permita al usuario definir tanto los problemas como los algoritmos a utilizar. Para lograr esto se pretende adoptar un enfoque de separación entre el problema y el método que se quiere aplicar para resolverlo.

1.1. Antecedentes

Prodef [9] es una herramienta que nace con el objetivo de extender el uso de la metaheurística fuera del ámbito de la investigación convencional, permitiendo al usuario hacer uso de estas técnicas sin necesidad de conocimientos previos sobre metaheurísticas y/o programación.

La versión actual de la herramienta es fruto de varios Trabajos de Fin de Grado a lo largo del tiempo:

- *Prodef: meta-modelado de problemas de optimización combinatoria (2020).* Andrés Calimero García Pérez [18].

- *Prodef-GUI: Interfaz gráfica para el modelado de problemas* (2021). Daniel González Expósito [19].

- *Prodef: Diseño, implementación y experimentación con nuevos resolutores* (2022). Miguel Angel Ordoñez Morales [24].

- *Prodef-Algorithm: Interfaz para el modelado de meta-heurísticas* (2022). Daniel del Castillo de la Rosa [15].

- *Prodef-SaaS: Despliegue y puesta en marcha de un servicio para la resolución de problemas de optimización* (2022). Ángel Tornero Hernández [26].

- *Prodef-Solution: Interfaz para la representación y visualización de soluciones* (2022). Yeixon Reinaldo Morales Gonzalez [22].

- *Prodef: unificación e integración de módulos* (2023). Alejandro Lugo Fumero [21].

Inicialmente, se definió un meta-modelo que permite especificar, en términos abstractos, las características esenciales de un problema de optimización. A partir de este modelo se ofrecía una interfaz gráfica para la traducción directa a distintos frameworks de optimización (jMetal y METCO [14]). De esta forma el usuario puede definir un problema de manera abstracta y que un resolutor reciba posteriormente dicho problema y compute una solución para el mismo.

Cabe destacar que el modelado del problema se realiza a través de una especificación realizada en lenguaje ProdefLang y recogida a través de un fichero JSON. Este lenguaje de dominio específico se creó con el objetivo de posibilitar la definición de funciones objetivo y restricciones en los modelos de forma sencilla. El lenguaje diseñado es una mezcla entre formulación matemática y lenguaje de programación orientado a objetos. ProdefLang permite expresar cualquier tipo de función objetivo y restricción, manteniendo a su vez la simplicidad, pues el objetivo es que cualquier persona sin conocimientos previos de programación pueda usarlo. Posteriormente, se desarrolló una interfaz gráfica para la definición de los problemas y sus instancias. En esta interfaz se hace uso de bloques a partir de Blockly [2] en vez de trabajar directamente con un archivo JSON. De esta manera, el usuario puede definir un problema de manera más sencilla y amigable con bloques, que luego se formatean a archivos JSON para la computación del problema.

En una segunda fase, se diseñó una interfaz gráfica para el modelado de algoritmos. Esta permite al usuario diseñar y definir sus propios algoritmos de optimización de la misma forma que los problemas, a partir de bloques. Para ello ha sido necesario la creación de un nuevo resolutor en lenguaje Rust y su propio compilador de ProdefLang a Rust. El nuevo resolutor ofrece la especificación a alto nivel de los algoritmos y flexibilidad a la hora de la definición de componentes. Una característica de gran relevancia del resolutor es que los algoritmos que se definen son independientes del problema, permitiendo aplicar el algoritmo para diferentes casos. Sin embargo, es necesario elegir componentes y parámetros específicos dependiendo del problema en cuestión.

En una tercera fase más actual se ha creado una nueva versión de Prodef llamada Othimi con el objetivo de mejorar la herramienta anterior y solucionar varios problemas que hacían que Prodef no cumpliese con los estándares de calidad esperados. Othimi ofrece una interfaz gráfica mejorada en comparación a la anterior y un rediseño de la creación de las instancias con el fin de que sea más sencillo para el usuario.

1.2. Objetivos a realizar

Actualmente, Othimi dispone de la capacidad de definir un problema y crear las instancias de dicho problema, no obstante, en estos momentos no se pueden realizar el modelado de los algoritmos para resolver problemas de optimización. Al igual que sucedía en Prodef, el apartado de definición de problemas y el de algoritmos eran independientes el uno del otro y en Othimi no está implementado el apartado de modelado de algoritmos. Por lo tanto, lo que este Trabajo de Fin de Grado pretende es la implementación, del modelado de algoritmos en esta nueva interfaz, realizando varios cambios en comparación con la versión anterior. El objetivo principal es trasladar componentes de jMetal a bloques de Blockly para poder modelar algoritmos con la misma estructura y parámetros que los expuestos en el nuevo subproyecto de jMetal, llamado jMetal-auto [4] en el cual los algoritmos se configuran a partir de cadenas de caracteres. La idea es poder obtener modelando un algoritmo desde la interfaz gráfica, un archivo que contenga la configuración de dicho algoritmo, la cual debe ser igual a las de jMetal-auto para poder ejecutarlo con esta herramienta. Además, se busca que los nuevos bloques diseñados puedan ser lo suficientemente generales para poder diseñar algoritmos personalizados, pero sin poder ser ejecutados en jMetal, ya que no cumplirían con su estructura.

Por lo tanto, los objetivos a realizar durante este Trabajo de Fin de Grado son los siguientes:

Objetivo	**Descripción**
Análisis de Prodef y Othimi.	Realizar un estudio de Prodef y de su nueva versión Othimi.
Análisis de jMetal y sus algoritmos.	Analizar el funcionamiento de jMetal, principalmente la funcionalidad de configuración y definición de algoritmos a partir de una cadena de parámetros, identificando los componentes y parámetros necesarios.
Implementación de la definición de algoritmos en la nueva interfaz gráfica.	A partir de la nueva versión de la interfaz, se implementará la definición de algoritmos haciendo uso de la librería Blockly, creando los bloques necesarios a partir de los componentes extraídos de jMetal-auto.
Obtención de un archivo compatible con jMetal-auto.	Implementar una funcionalidad que permita exportar un archivo con la configuración de un algoritmo definido en la interfaz de Othimi que sea compatible con jMetal-auto, para ejecutarlo usando la herramienta jMetal.
Documentación	Elaboración de la memoria del TFG así como la documentación necesaria durante el curso de la asignatura y los materiales de ayuda para el uso de Prodef.

Tabla 1.1: Objetivos del proyecto

Capítulo 2

Introducción a las metaheurísticas

2.1. Metaheurísticas

Las metaheurísticas son técnicas de optimización y búsqueda para la resolución de problemas, que nos permiten abordar situaciones complejas mediante procedimientos aproximados en vez de utilizar técnicas exactas. Estas se centran en la eficiencia de la búsqueda, utilizando información relevante del problema para encontrar soluciones óptimas sin garantía de alcanzar el óptimo global.

Una de las principales ventajas de las metaheurísticas es la capacidad para solucionar problemas complejos, en los que métodos tradicionales de optimización son ineficientes o no tienen posibilidad de aplicarse. Por lo tanto, las metaheurísticas se caracterizan por ofrecer una forma flexible y potente de búsqueda que puede adaptarse a distintos escenarios y condiciones.

Sin embargo, hay que tener en cuenta que implementar y aplicar metaheurísticas puede ser un desafío debido a su complejidad, ya que la selección de los parámetros adecuados, la definición de los operadores de búsqueda y la configuración correcta de los criterios de parada son aspectos cruciales que requieren experiencia y bastantes conocimientos específicos. A pesar de esta desventaja, las metaheurísticas son extensamente utilizadas en campos tecnológicos como ciencia de datos, ingeniería y logística.

2.2. Tipos de metaheurísticas

Existen diferentes tipos de metaheurísticas para resolver los problemas de optimización, algunos de ellos son los siguientes:

- **Algoritmos evolutivos.** Se basan en los principios de la evolución biológica, usando operadores de selección, cruce y mutación con los que crear nuevas soluciones.

- **Optimización por enjambre de partículas.** Método que se basa en el comportamiento de un enjambre, en el cual las "partículas" representan las soluciones, las cuales se mueven en el espacio de búsqueda.

- **Búsqueda Tabú.** Se basa en una búsqueda de óptimos locales desplazándose entre los vecinos aunque sean peores, con la peculiaridad de que se crea una lista de movimientos prohibidos(tabú) para evitar quedarse atrapada de manera repetitiva en una solución.

- **Algoritmos meméticos.** Consiste en la realización de la búsqueda global junto con la mejora local. Para encontrar regiones prometedoras se realiza la búsqueda global, tras la cual a las soluciones obtenidas se le aplican técnicas de búsqueda local para mejorarla.

Aunque como se ha visto, hay bastantes tipos de metaheurísticas, cada una con sus peculiaridades. Este Trabajo de Fin de Grado se centrará principalmente en los algoritmos evolutivos y la optimización por enjambre de partículas, ya que estos son los tipos que de momento están definidos en jMetal-auto [4] para la configuración de algoritmos mediante una cadena de parámetros.

2.3. Algoritmos evolutivos

Hay muchas variantes de algoritmos evolutivos, no obstante la idea detrás de estos es la misma: dada una población de individuos con recursos limitados, se producirá una competencia por dichos recursos, provocando la supervivencia del más apto. Este proceso producirá un incremento en las aptitudes de la población.

A partir de un conjunto de soluciones iniciales creadas de manera aleatoria, se seleccionan los individuos con mayores valores de aptitud para que produzcan la próxima generación. Esto se realiza aplicando la recombinación y/o mutación. La recombinación es un operador el cual se aplica a los padres para que produzcan una descendencia. La mutación se aplica de manera individual a un candidato, resultando en un posible candidato con una variación al original. Tras la creación de la descendencia a partir de las operaciones de recombinación y mutación, los nuevos candidatos compiten con los antiguos por un lugar en la nueva generación. Todo este proceso se repite hasta que se cumpla el criterio de terminación, como por ejemplo el límite computacional o evaluaciones máximas.

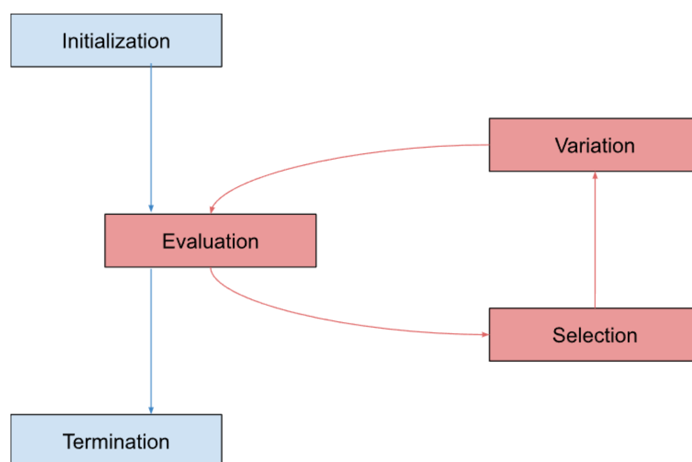


Figura 2.1: Proceso algoritmo evolutivo

2.3.1. Elementos de un algoritmo evolutivo

Los elementos principales que conforman un algoritmo evolutivo son los siguientes [17]:

- **Población.** La población es la representación de las posibles soluciones. Definir la población es un proceso simple que requiere establecer el tamaño de la población y se suele inicializar de forma aleatoria o a través de algún criterio específico. Normalmente, el tamaño de la población es constante y no varía, lo que genera la escasez de recursos por los cuales se produce la competencia entre los individuos. El operador de selección trabaja directamente con la población actual, realizando la selección de los mejores individuos para producir la próxima generación o los peores individuos para su reemplazo.
- **Función de evaluación.** El objetivo de la función de evaluación es evaluar la calidad de los individuos de la población. Además, representa los requisitos que atienden al objetivo del problema y evalúa la calidad en función de estos requisitos.
- **Operador de selección.** Se trata del proceso en el cual se eligen los individuos que se convertirán en los padres de la siguiente generación. Por lo general se tiene la intención de elegir a los individuos más aptos, ya que producirán una mejor descendencia, no obstante se les proporciona alguna posibilidad a los peores individuos. Si un individuo es padre, quiere decir que ha sido seleccionado para someterse al proceso de variación con el objetivo de producir descendencia. La selección es la principal responsable del incremento de la calidad dentro de la población, ya que los individuos más aptos tienen más posibilidades de convertirse en padres.
- **Operadores de variación**. Estos operadores son los encargados de generar la diversidad de la población. Los operadores de variación se dividen entre la recombinación y la mutación.
    - **Recombinación.** La recombinación o cruce, se encarga de fusionar los genotipos de dos individuos diferentes con los que producir una descendencia con las características de ambos combinados. La decisión de qué características se combinan y de qué forma, se realizan de manera aleatoria. De esta forma es posible que aparezcan tanto individuos con características no deseadas e individuos con características mejores, aunque la mayoría no son ni mejores ni peores que sus padres. Existe la posibilidad de operadores de recombinación en los que se hagan usos de más de dos padres, no obstante no tienen un equivalente biológico en la realidad.
    - **Mutación.** La mutación consiste en realizar cambios aleatorios al genotipo de un individuo, dando como resultado un individuo con características ligeramente diferentes al original y una variación en la población. La mutación permite la exploración del espacio de búsqueda en busca de mejores soluciones. Los cambios realizados por la mutación se realizan a partir de una probabilidad de mutación que dependiendo de su valor puede llegar a generar cambios drásticos o limitarla.
- **Terminación.** Determina cuál es el criterio de parada para detener el algoritmo. Técnicamente el criterio es alcanzar una solución óptima, no obstante es posible que nunca se satisfaga esta condición y por lo tanto se aplica una condición que garantice la parada del algoritmo. Comúnmente encontramos opciones como el tiempo máximo de CPU, número total de evaluaciones, mejora mínima de aptitud durante un periodo o la disminución de la diversidad de la población.

2.4. Optimización por enjambre de partículas (PSO)

La Optimización por Enjambre de partículas es un algoritmo metaheurístico inspirado en el comportamiento de una bandada de aves, un enjambre de insectos o un banco de peces. Este algoritmo fue presentado en 1995 por Eberhart y Kennedy.

El algoritmo simula un enjambre en el que cada partícula es una posible solución y estas son representadas mediante un vector en un espacio de búsqueda multidimensional. Además, cada partícula tiene asociado un vector de velocidad, el cual determina su próximo movimiento. El PSO actualiza la velocidad de las partículas en función de su velocidad actual, su mejor posición personal y la mejor posición global encontrada de entre las de todo el enjambre. Para poder encontrar la mejor solución, se produce una colaboración entre las partículas, de tal forma que comparten la mejor posición que han podido encontrar hasta el momento. Esto influye en los movimientos de las demás partículas, que se irán convergiendo a la mejor solución encontrada en cada iteración [25].

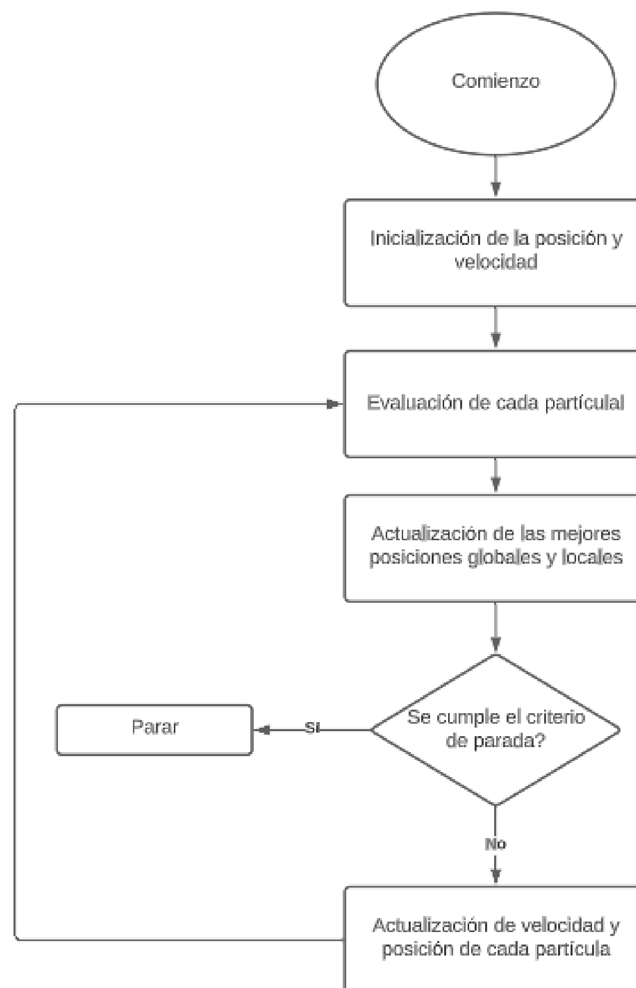


Figura 2.2: Diagrama PSO

El PSO es efectivo en problemas de optimización en espacios tanto de valores reales como discretos. Su simplicidad y eficiencia computacional lo hacen una buena opción para tratar problemas complejos cuando no hay suficiente información del espacio de

búsqueda.

2.4.1. Elementos del PSO

Los elementos principales del PSO son:

- **Enjambre.** Es el conjunto formado por todas las partículas. Las partículas interactúan entre sí y comparten información sobre las mejores soluciones encontradas, permitiendo la exploración conjunta del espacio de búsqueda.

- **Partículas.** Individuos del enjambre que representan posibles soluciones al problema de optimización. Cada partícula tiene una posición y una velocidad que determinan su movimiento en cada iteración.

- **Función de evaluación.** Se utiliza para medir la calidad de una solución en el espacio de búsqueda, calculando el rendimiento de cada partícula en su posición actual.

- **Posición.** Representa una posible solución al problema. Cada partícula mantiene un registro de su mejor posición personal, es decir, la mejor solución encontrada hasta el momento.

- **Velocidad.** Vector que indica la dirección y magnitud del movimiento de una partícula en el espacio de búsqueda. La velocidad se actualiza en cada iteración en función de la mejor posición personal y la mejor posición global encontrada por el enjambre.

- **Mejor posición personal.** Mejor solución encontrada hasta el momento por cada partícula individualmente.

- **Mejor posición global.** Mejor solución encontrada entre todas las partículas del enjambre.

- **Mutación.** Hay versiones del PSO, que incorporan al proceso el operador de mutación. Como ya se describió anteriormente en el algoritmo evolutivo, la mutación afecta al algoritmo proporcionando variabilidad al espacio de búsqueda. El operador de mutación modifica las características de las partículas en cada iteración tras la actualización de la velocidad y posición. y utiliza la velocidad, la posición junto a otros parámetros como posibles opciones a las que se puede aplicar la mutación. Este proceso genera aleatoriedad en el algoritmo, permitiendo que las partículas no se lleguen a quedar atrapadas en alguna solución cuando podría existir una más óptima.

2.5. Algoritmos multiobjetivo

Modelar un problema de optimización con solamente una función objetivo puede resultar en ciertos casos una tarea complicada, por lo que es muy común observar problemas con múltiples objetivos a optimizar, lo que se llama optimización multiobjetivo. Para resolver estos problemas de múltiples objetivos es necesario modificar o crear algoritmos

que sean capaces de encontrar un conjunto de soluciones óptimas que satisfagan los objetivos. Este conjunto de soluciones finales se denomina Pareto.

A diferencia de la optimización mono-objetivo, los algoritmos multiobjetivo deben ser capaces de contemplar múltiples objetivos, para ellos ciertos elementos, aunque siendo los mismos, se deben comportar de manera diferente que en un algoritmo mono-objetivo. Ejemplos de esto serían la selección, la cual usaría técnicas de selección basadas en dominancia, que priorizan soluciones que son dominantes sobre otras en función a los objetivos y los operadores de variación que deben mantener la diversidad de las soluciones y el equilibrio entre las funciones objetivo [13].

Capítulo 3

Othimi

Como ya se dijo anteriormente, Othimi es la nueva versión mejorada del antiguo Prodef [9], una herramienta web con una interfaz gráfica para modelar problemas y algoritmos de optimización sin que el usuario necesite programar. La interfaz está creada a partir de la biblioteca Blockly [2], que sirve para crear interfaces gráficas de programación basadas en bloques que los usuarios pueden arrastrar y soltar bloques predefinidos para, en el caso de Othimi realizar el modelado de bloques y algoritmos. El uso de esta manera de Blockly hace que Othimi sea una herramienta única en el ámbito de la metaheurística queriendo proporcionar a los usuarios la posibilidad de modelar sus propios problemas, algoritmos e instancias de la manera más amigable posible, sin la necesidad de grandes conocimientos sobre metaheurística.

Othimi se divide en el *front-end* y el *back-end* los cuales están creados desde cero utilizando tecnologías actuales como Vite.js [10] y React [8] para el *front-end* y los *frameworks* GraphQl y Koa [5] para *back-end* con la finalidad de mejorar la calidad en comparación a Prodef. Tanto el cambio de marca como la renovación del código establece la base necesaria para poder desarrollar una herramienta de gran calidad y sencilla de desarrollar en el futuro.

Figura 3.1: Marca Othimi

3.1. Instalación y puesta en marcha

Tanto la instalación como la puesta en marcha de Othimi es muy sencilla, una característica de gran utilidad para los usuarios que quieran utilizar la herramienta. Los pasos a seguir son los siguientes:

1. Clonar los repositorios del *front-end* y del *back-end*:

- https://github.com/ULL-prodef/prodef-gui-frontendV2.git
- https://github.com/ULL-prodef/prodef-gui-backendV2.git

2. Instalar Node.js [7]

3. Entrar en los directorios que hemos clonado y ejecutar en cada uno de ellos `npm install` para la instalación de todos los paquetes necesarios y luego ejecutar `npm run dev` que los pondrá en marcha. A la hora de hacerlo, el *front-end* se redirigirá automáticamente al usuario a la interfaz gráfica.

3.2. Estado actual del *front-end*

El *front-end* es la parte de la herramienta con la que interactúan los usuarios directamente. A través de esta interfaz gráfica, intuitiva y amigable, el usuario puede modelar problemas de optimización y sus instancias. El principal objetivo de esta interfaz es ofrecer una experiencia de usuario accesible y de calidad, sin necesidad de tener conocimientos avanzados de programación.

El *front-end* está desarrollado utilizando el *framework* Vite.js y tecnologías como React, caracterizándose por un gran rendimiento y capacidad para brindar una experiencia fluida al usuario en comparación a la versión anterior. Blockly se utiliza para poder implementar una interfaz gráfica basada en bloques, donde los usuarios pueden modelar problemas moviendo bloques y uniéndolos entre sí. Actualmente, las funcionalidades implementadas son crear problemas personalizados, definir parámetros y tablas, ofreciendo opciones de filtrado, búsqueda y paginación para facilitar la gestión de problemas e instancias. Tiene elementos generales como la barra de navegación para proporcionar una navegación coherente y elementos específicos para la autenticación y la gestión de problemas e instancias.

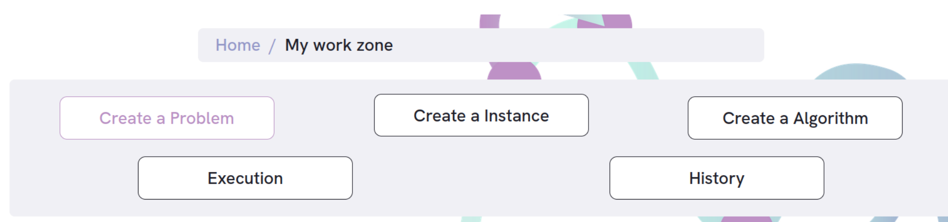


Figura 3.2: Zona de trabajo de Othimi

3.2.1. Autenticación de usuario

Othimi presenta una proceso de autenticación de usuario, el cual es necesario realizar si se quiere acceder a la zona de trabajo. Los usuarios pueden iniciar sesión utilizando cuentas de GitHub o a través de un correo electrónico, como se puede observar en la Figura 3.3.

La autenticación mediante GitHub es la más sencilla, ya que el usuario solo necesita otorgar permisos a la `OAuth` de Prodef. El *back-end* se encarga de manejar la información y generar el token correspondiente. Por otro lado, la autenticación por correo electrónico se divide en dos procesos: inicio de sesión y registro. En el registro, se solicita un nombre de usuario único, un correo electrónico único y una contraseña que cumpla con

ciertos requisitos. En el inicio de sesión, el usuario puede ingresar su dirección de correo electrónico o su nombre de usuario, junto con la contraseña correspondiente. Tanto en la figura 3.4 como en la 3.5 se muestra los datos que hay que proporcionar.

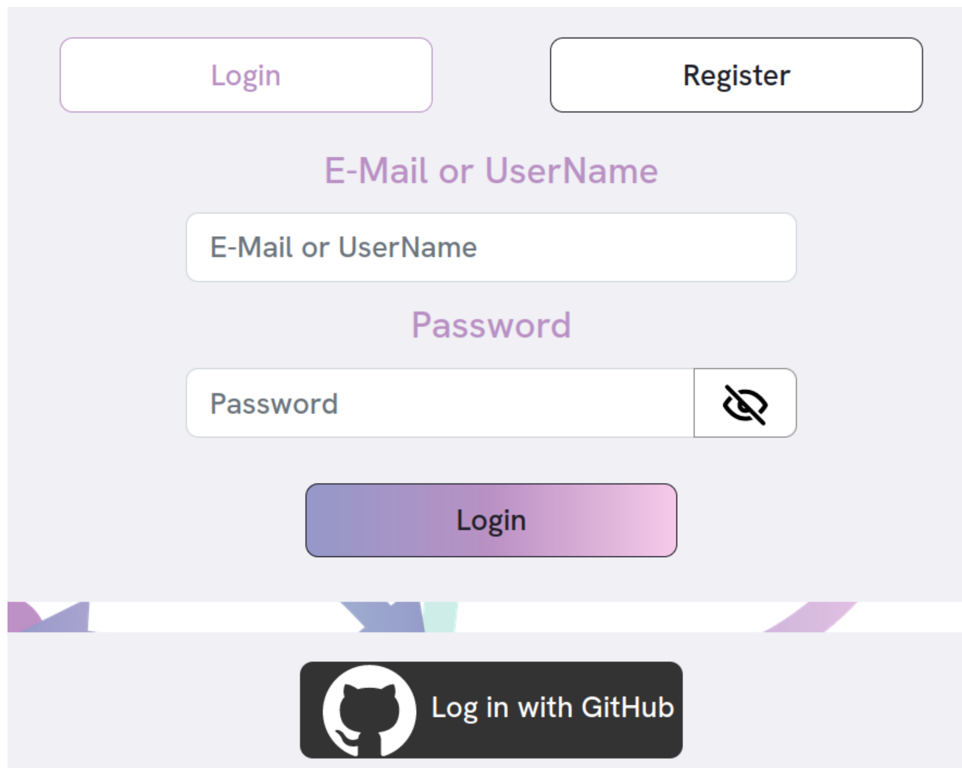


Figura 3.3: Interfaz de autentificación

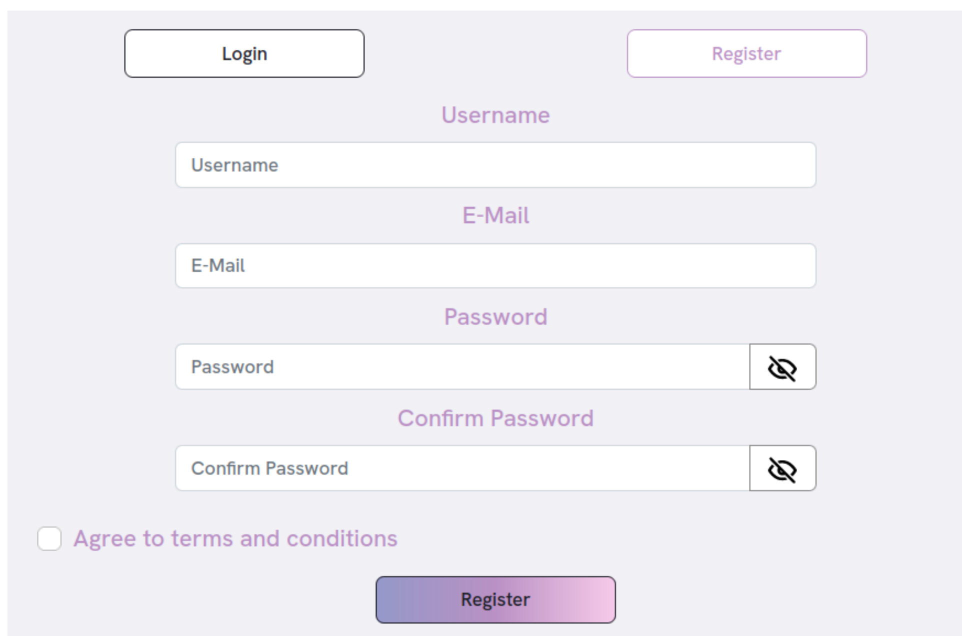


Figura 3.4: Pantalla de registro con email

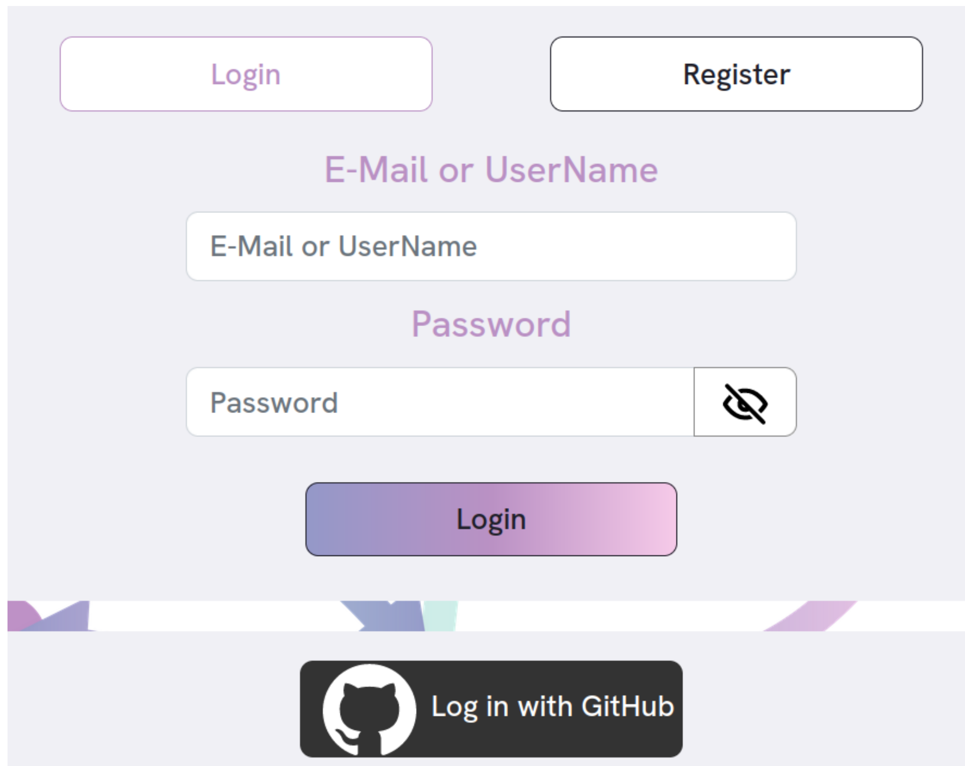


Figura 3.5: Pantalla de inicio de sesión con email

3.2.2. Modelado de problemas

Una de las principales funcionalidades implementadas hasta el momento es el modelado de problemas a partir de bloques. Esto permite al usuario definir un problema de optimización de manera sencilla y sin necesidad de tener grandes conocimientos sobre metaheurística y programación.

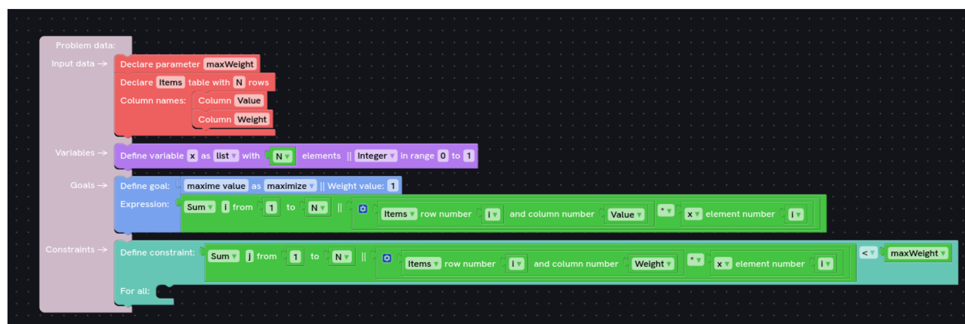


Figura 3.6: Plantilla por defecto del problema Knapsack

Para crear un problema, el usuario lo puede hacer manera manual utilizando los bloques definidos o también existe la opción de visualizar y duplicar plantillas hechas por defecto como la mostrada en la Figura 3.6. Que haya plantillas por defecto ya creadas proporciona una total simplificación a la hora del modelado. La herramienta permite guardar los problemas que el usuario cree y ofrece una función de filtrado para mostrar los problemas existentes en función del creador del problema, la familia del problema y el número de tablas y parámetros.

Figura 3.7: Filtro de problemas

3.2.3. Creación de instancias

Othimi permite crear instancias en las que especificar parámetros y tablas que se utilizarán en un problema. Una instancia puede tener uno o más parámetros y tablas. En la zona de administración de parámetros, se pueden crear y eliminar parámetros, cambiar sus nombres y valores. En la zona de administración de tablas, se sigue un esquema similar al de los parámetros, donde se pueden crear tablas, cambiar sus nombres, filas y columnas, y agregar valores para cada celda. Además, hay una zona dedicada a cargar tablas desde archivos .csv, donde se notificará al usuario en caso de errores. Al igual que con los problemas se pueden guardar las instancias creadas y buscar a partir de un filtro.

Figura 3.8: Pantalla de creación de instancias

3.3. Estado actual del *back-end*

El *back-end* de Othimi usa el framework Apollo [1] para GraphQL. Como *framework*, se usa Koa en lugar de Express para simplificar las tecnologías utilizadas en Othimi. y para la base de datos, se utiliza MongoDB, una base de datos NoSQL.

En cuanto a la autenticación, Othimi permite la autenticación mediante correo electrónico, a diferencia del antiguo Prodef que solo admitía autenticación a través de GitHub. Para cifrar las contraseñas, se utiliza la librería Bcrypt. Además, tiene implementado el

uso de JSON Web Tokens (JWT) en el *back-end* para asegurar la comunicación segura entre el *front-end* y el *back-end*.

En la base de datos existen nuevos modelos de datos utilizando Mongoose de los cuales actualmente se están utilizando `Problems`, `Instances`, `UsersEmail` y `UsersGitHub`. Pero existen otros modelos planteados como `Algorithm`, `SharedProblems`, o `Historial` para utilizar en futuros desarrollos.

3.4. Estado del resolutor de Rust-

Por último destacar que Daniel del Castillo de la Rosa en su TFG [15] creó un resolutor para el antiguo Prodef en el lenguaje de programación Rust. Este resolutor permite especificar qué algoritmo se quiere usar para resolver el problema en cuestión y permitir el modelado de algoritmos, haciendo uso de una abstracción basada en componentes.

La idea del diseño de algoritmos a partir de componentes, consiste en que con el fin de realizar comparaciones entre diferentes algoritmos y problemas, es fundamental que los algoritmos sean independientes del problema en sí. Esto significa que cada algoritmo debe ser capaz de aplicarse a diferentes problemas, sin estar limitado a uno específico. Esta independencia permite evaluar y analizar el desempeño de los algoritmos en diversas situaciones y determinar su eficacia de manera más objetiva y generalizada. El algoritmo principal consiste en una serie de componentes que son independientes del problema en sí. Sin embargo, cada uno de estos componentes puede tener dependencias específicas del problema y/o de la ejecución. Estas dependencias pueden ser parámetros numéricos o componentes particulares que varían en función del problema que se esté abordando.

Capítulo 4

jMetal y componentes para jMetal-auto

Como se ha comentado, jMetal [16] es un framework basado en Java para la optimización multiobjetivo con metaheurísticas. Actualmente, en la optimización multiobjetivo se tiende a utilizar herramientas de diseño automático y configuración automática de parámetros para encontrar configuraciones de metaheurísticas que resuelvan de manera efectiva diversos problemas. La versión más reciente, jMetal 6.0, es una extensión de jMetal 5.0, que agrega un paquete para desarrollar algoritmos a partir de plantillas basadas en componentes y otro paquete para la configuración y diseño automático de algoritmos. Aprovechando esta nueva funcionalidad, se quieren crear bloques en base a los componentes necesarios para la configuración de algoritmos en jMetal-auto [4] para poder obtener un archivo que contenga una configuración de un algoritmo capaz de ejecutarse en jMetal.

El proyecto de jMetal es un proyecto Maven que sigue la siguiente estructura:

Subproyecto	**Contenido**
`jmetal-core`	Clases principales
`jmetal-solution`	Codificaciones de soluciones
`jmetal-algorithm`	Implementaciones de algoritmos
`jmetal-problem`	Problemas de referencia
`jmetal-lab`	Experimentación y visualización
`jmetal-parallel`	Extensiones para ejecución paralela
`jmetal-auto`	Auto-diseño y configuración
`jmetal-component`	Algoritmos basados en componentes

Tabla 4.1: Estructura del proyecto Maven de jMetal

4.1. Algoritmos basados en componentes

El subproyecto “jmetal-component” en jMetal se centra en la implementación de algoritmos basados en componentes. Este enfoque se basa en la idea de que los algoritmos evolutivos pueden ser construidos y configurados mediante la combinación de diferentes componentes o módulos funcionales.

En jMetal, los algoritmos basados en componentes permiten a los desarrolladores crear algoritmos personalizados al seleccionar y combinar diferentes componentes según sus necesidades, la misma idea que pretende implementar Othimi pero con la diferencia

de usar una interfaz gráfica y Blockly [2]. Estos componentes incluyen operadores de cruce, operadores de mutación, selección y criterios de parada, entre otros. Esto ofrece una mayor flexibilidad, ya que permite la reutilización de componentes existentes y su combinación para construir nuevos algoritmos, facilitando el desarrollo de algoritmos evolutivos adaptados a problemas específicos.

Los algoritmos basados en componentes fomentan la escalabilidad, ya que los módulos individuales pueden ser fácilmente modificados o reemplazados para mejorar el rendimiento del algoritmo en diferentes situaciones, Permitiendo una mayor capacidad de adaptación de los algoritmos a medida que evolucionan las necesidades o se presentan nuevos desafíos.

4.2. jMetal-auto

jMetal-auto es un subproyecto de jMetal que se enfoca en el diseño automático y la configuración de metaheurísticas multiobjetivo. Este diseño se logra a través de técnicas de aprendizaje automático y optimización automática. Estas técnicas permiten ajustar automáticamente los parámetros y configuraciones de los algoritmos para adaptarse a diferentes problemas y mejorar su rendimiento.

El objetivo de esta funcionalidad es eliminar o reducir la necesidad de una intervención por parte del usuario en el proceso de desarrollo y configuración de los algoritmos, facilitando a los investigadores y desarrolladores el uso de jMetal para resolver problemas multiobjetivo. Además, el enfoque de diseño automático y configuración en jMetal permite explorar de manera más eficiente el espacio de búsqueda de algoritmos y encontrar soluciones óptimas o cercanas a óptimas para problemas multiobjetivo, dando como resultado una mejora de la calidad de los resultados y acelerar el proceso de desarrollo de algoritmos. Actualmente, a fechas de desarrollo de este TFG, los algoritmos disponibles son el NSGAII [20] y el MOPSO [12], aunque como jMetal es una herramienta en constante desarrollo, se está implementado nuevos algoritmos como el MOEAD.

A continuación se muestra un ejemplo de un archivo de configuración para la ejecución del algoritmo NSGAII, el cual toma como parámetros una cadena de caracteres que contiene la especificación de los componentes que tiene dicho algoritmo y los valores de los parámetros que corresponden a esos componentes.

```
1  public class NSGAIIConfiguredFromAParameterString {
2  
3    public static void main(String[] args) {
4      String referenceFrontFileName = "resources/referenceFrontsCSV/ZDT1.csv";
5  
6      String[] parameters =
7          ("--problemName org.uma.jmetal.problem.multiobjective.zdt.ZDT1 "
8              + "--randomGeneratorSeed 12 "
9              + "--referenceFrontFileName " + referenceFrontFileName + " "
10             + "--maximumNumberOfEvaluations 25000 "
11             + "--algorithmResult population "
12             + "--populationSize 100 "
13             + "--offspringPopulationSize 100 "
14             + "--createInitialSolutions random "
15             + "--variation crossoverAndMutationVariation "
16             + "--selection tournament "
17             + "--selectionTournamentSize 2 "
```

```
18              + "--rankingForSelection dominanceRanking "
19              + "--densityEstimatorForSelection crowdingDistance "
20              + "--crossover SBX "
21              + "--crossoverProbability 0.9 "
22              + "--crossoverRepairStrategy bounds "
23              + "--sbxDistributionIndex 20.0 "
24              + "--mutation polynomial "
25              + "--mutationProbabilityFactor 1.0 "
26              + "--mutationRepairStrategy bounds "
27              + "--polynomialMutationDistributionIndex 20.0 ")
28              .split("\\s+");
29
30      AutoNSGAII autoNSGAII = new AutoNSGAII();
31      autoNSGAII.parse(parameters);
32
33      AutoNSGAII.print(autoNSGAII.fixedParameterList());
34      AutoNSGAII.print(autoNSGAII.configurableParameterList());
35
36      EvolutionaryAlgorithm<DoubleSolution> nsgaII = autoNSGAII.create();
37
38      EvaluationObserver evaluationObserver = new EvaluationObserver(1000);
39      RunTimeChartObserver<DoubleSolution> runTimeChartObserver =
40          new RunTimeChartObserver<>(
41              "NSGA-II", 80, 100,
42              referenceFrontFileName, "F1", "F2");
43
44      nsgaII.observable().register(evaluationObserver);
45      nsgaII.observable().register(runTimeChartObserver);
46
47      nsgaII.run();
48
49      JMetalLogger.logger.info("Total computing time: " + nsgaII.totalComputingTime()); ;
50
51      new SolutionListOutput(nsgaII.result())
52          .setVarFileOutputContext(new DefaultFileOutputContext("VAR.csv", ","))
53          .setFunFileOutputContext(new DefaultFileOutputContext("FUN.csv", ","))
54          .print();
55    }
56  }
```

4.2.1. Componentes de jMetal-auto y sus parámetros

Para conocer los tipos de bloques y parámetros que habrá que implementarse en la interfaz, el primer paso es analizar la documentación publicada acerca del funcionamiento de jMetal-auto y el estudio de los componentes y los parámetros asociados a estos componentes. Para ello, se ha hecho un estudio y lectura exhaustiva del código de jMetal, analizando los componentes de configuración necesarios para los algoritmos implementados hasta el momento en jMetal-auto que serían el NSGAII y MOPSO. Posteriormente, se estudió el código de todos esos componentes ubicados en el subproyecto "jmetal-component" entendiendo su función y los parámetros que necesita. Tras obtener el listado de componentes, se realizó el mismo proceso para los parámetros, obteniendo la información de que representan, el tipo de dato y los intervalos asignables para sus valores en caso de que sean numéricos.

Tras estudiar a fondo el código de jMetal, se han representado los componentes im-

plementados en el código de jMetal a partir de unos esquemas. En estos esquemas se han utilizado colores para diferenciar cuáles son los parámetros y cuáles los tipos de los componentes, como viene explicado en la Figura 4.1. Estarían los componentes cuyos tipos son los coloreados en azul, que pueden tener como parámetros, los campos coloreados en morado o componentes que ejercen como parámetros de dichi componente, coloreados de verde. Un ejemplo de esto sería el componente de selección mostrado en el algoritmo evolutivo, donde sus posibles tipos son `tournament`, `random`, `populationAndNeighborhoodMatingPoolSelection` coloreados en azul y para cada tipo sus parámetros aparecen en morado como por ejemplo el parámetro `selectionTournamentSize` de tipo `TournamentSelection`.

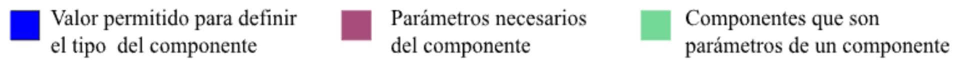
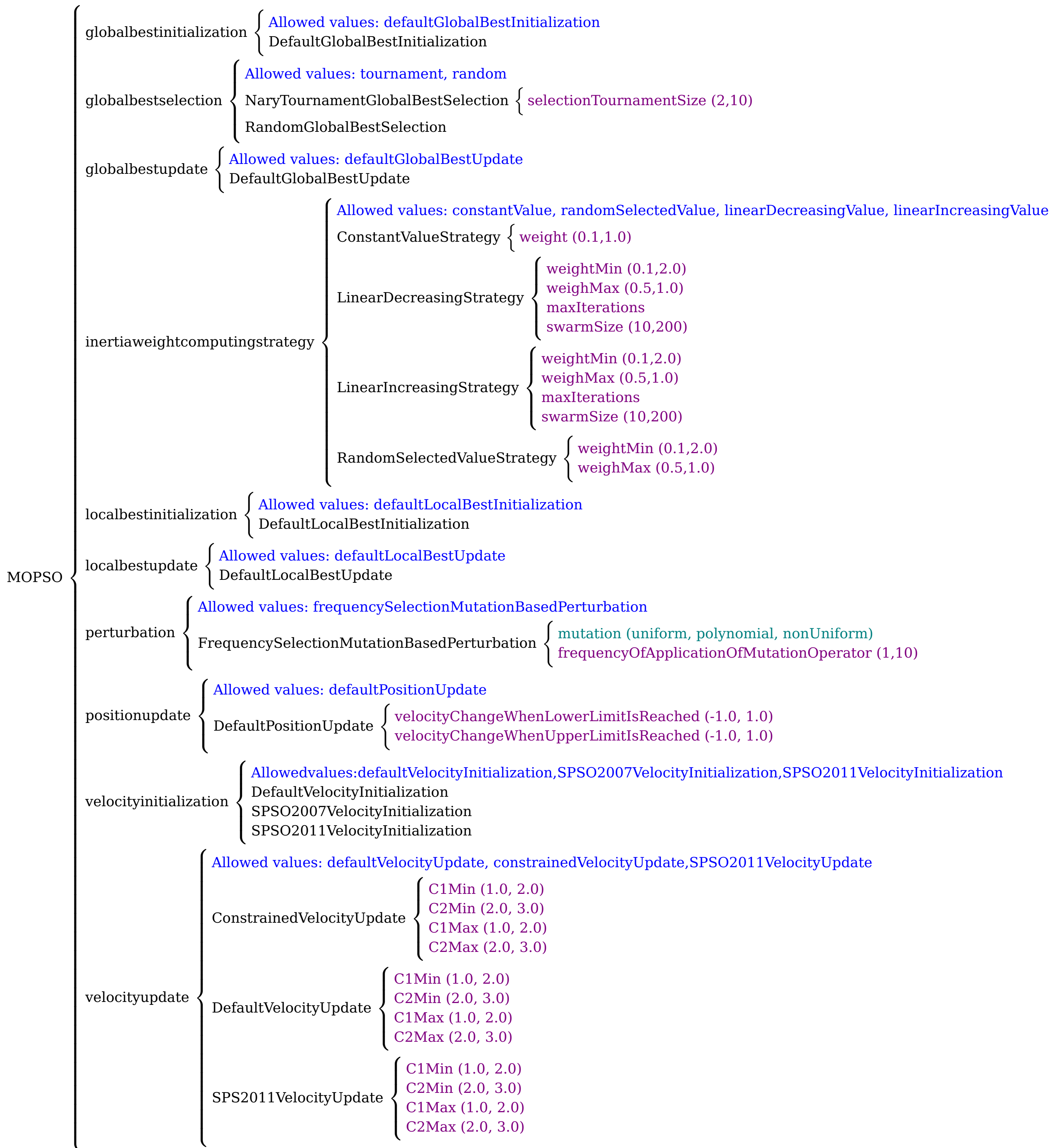


Figura 4.1: Aclaración sobre los colores

- Common
  - evaluation
    - MultiThreadedEvaluation
    - SequentialEvaluation
    - SequentialEvaluationWithArchive
  - solutionscreation
    - Allowed values: random, latinHypercubeSampling, scatterSearch
    - LatinHypercubeSamplingSolutionsCreation
    - RandomSolutionsCreation
    - ScatterSearchSolutionsCreation
  - termination
    - TerminationByComputingTime
    - TerminationByEvaluations
    - TerminationByKeyboard
    - TerminationByQualityIndicator

- Evolutionary-algorithm
  - selection
    - Allowed values: tournament, random,populationAndNeighborhoodMatingPoolSelection
    - TournamentSelection
      - selectionTournamentSize (2,10)
    - PopulationAndNeighborhoodSelection
      - neighborhoodSelectionProbability
      - neighborhoodSelectionSize
      - replacedSolutions
    - RandomSelection
  - replacement
    - Allowed values: rankingAndDensityEstimatorReplacement
    - RankingAndDensityEstimatorReplacement
      - removalPolicy
      - rankingForReplacement
      - densityEstimatorForReplacement
  - variation
    - Allowed values: crossoverAndMutationVariation, differentialEvolutionVariation
    - CrossoverAndMutationVariation
      - offspringPopulationSize
      - crossover (SBX, BLX_ALPHA)
      - mutation (uniform, polynomial, nonUniform)
    - DifferentialEvolutionCrossoverVariation
      - differentialEvolutionCrossover
      - mutation (uniform, polynomial, nonUniform)


MOPSO
globalbestinitialization
Allowed values: defaultGlobalBestInitialization
DefaultGlobalBestInitialization
globalbestselection
Allowed values: tournament, random
NaryTournamentGlobalBestSelection
selectionTournamentSize (2,10)
RandomGlobalBestSelection
globalbestupdate
Allowed values: defaultGlobalBestUpdate
DefaultGlobalBestUpdate
inertiaweightcomputingstrategy
Allowed values: constantValue, randomSelectedValue, linearDecreasingValue, linearIncreasingValue
ConstantValueStrategy
weight (0.1,1.0)
LinearDecreasingStrategy
weightMin (0.1,2.0)
weighMax (0.5,1.0)
maxIterations
swarmSize (10,200)
LinearIncreasingStrategy
weightMin (0.1,2.0)
weighMax (0.5,1.0)
maxIterations
swarmSize (10,200)
RandomSelectedValueStrategy
weightMin (0.1,2.0)
weighMax (0.5,1.0)
localbestinitialization
Allowed values: defaultLocalBestInitialization
DefaultLocalBestInitialization
localbestupdate
Allowed values: defaultLocalBestUpdate
DefaultLocalBestUpdate
perturbation
Allowed values: frequencySelectionMutationBasedPerturbation
FrequencySelectionMutationBasedPerturbation
mutation (uniform, polynomial, nonUniform)
frequencyOfApplicationOfMutationOperator (1,10)
positionupdate
Allowed values: defaultPositionUpdate
DefaultPositionUpdate
velocityChangeWhenLowerLimitIsReached (-1.0, 1.0)
velocityChangeWhenUpperLimitIsReached (-1.0, 1.0)
velocityinitialization
Allowedvalues:defaultVelocityInitialization,SPSO2007VelocityInitialization,SPSO2011VelocityInitialization
DefaultVelocityInitialization
SPSO2007VelocityInitialization
SPSO2011VelocityInitialization
velocityupdate
Allowed values: defaultVelocityUpdate, constrainedVelocityUpdate,SPSO2011VelocityUpdate
ConstrainedVelocityUpdate
C1Min (1.0, 2.0)
C2Min (2.0, 3.0)
C1Max (1.0, 2.0)
C2Max (2.0, 3.0)
DefaultVelocityUpdate
C1Min (1.0, 2.0)
C2Min (2.0, 3.0)
C1Max (1.0, 2.0)
C2Max (2.0, 3.0)
SPS2011VelocityUpdate
C1Min (1.0, 2.0)
C2Min (2.0, 3.0)
C1Max (1.0, 2.0)
C2Max (2.0, 3.0)

4.3. Descripción de los componentes y sus parámetros para el NSGAII y el MOPSO

Tras obtener una imagen general de qué componentes son necesarios para la definición y configuración de un algoritmo en jMetal-auto, se tienen que describir los componentes y sus parámetros de los algoritmos NSGAII y MOPSO a modo de aproximación en la que estarán basados los nuevos bloques de la interfaz para su creación.

4.3.1. Componentes de inicialización

Los componentes de inicialización en un algoritmo de optimización son responsables de generar la población inicial de soluciones que se utilizará como punto de partida para la búsqueda de soluciones óptimas. A continuación se describirán este tipo de componentes y los parámetros.

NSGAII

Componente	**Descripción**	**Parámetros**
InitialSolutionsCreation	Este componente genera las soluciones iniciales para el algoritmo	type, population

Tabla 4.2: Componentes de inicialización para NSGAII.

Parámetros	**Descripción**
populationSize	Tamaño de la población inicial

Tabla 4.3: Parámetros de los componentes de inicialización para NSGAII

MOPSO

Componente	**Descripción**	**Parámetros**
InitialSwarmsCreation	Este componente genera el enjambre inicial, en el MOPSO	type, swarmSize
GlobalBestInitialization y LocalBestInitialization	Estos componentes se encargan de inicializar las mejores posiciones individuales y globales en el MOPSO.	type
Velocityinitialization	Inicializa la velocidad de las partículas al comienzo del MOPSO.	type

Tabla 4.4: Componentes de inicialización para MOPSO

Parámetros	**Descripción**
swarmSize	Tamaño del enjambre inicial

Tabla 4.5: Parámetros de los componentes de inicialización para MOPSO

4.3.2. Componentes de evaluación y terminación

Los componentes de evaluación son los responsables de medir la calidad de las soluciones generadas durante el proceso de búsqueda. Estos componentes desempeñan un papel crucial en la toma de decisiones del algoritmo, ya que proporcionan información sobre la calidad relativa de las soluciones y guían hacia soluciones óptimas. Por otro lado, los componentes de terminación son responsables de establecer las condiciones bajo las cuales se detiene la ejecución del algoritmo para poder evitar una ejecución infinita.

NSGAII y MOPSO

Componente	**Descripción**	**Parámetros**
Evaluation	En este componente, las soluciones generadas soneva-luadas y se les asigna un valor de aptitud.	type
TerminationByEvaluations	Componente que realiza la parada del proceso si llegar a un límite de evaluaciones	maximumNumberOf Evaluations

Tabla 4.6: Componentes de evaluación y terminación.

Parámetros	**Descripción**
maximumNumberOfEvaluations	Número máximo de iteraciones que se ejecuta el proceso

Tabla 4.7: Parámetros de los componentes de evaluación y terminación

4.3.3. Componentes de selección

Los componentes de selección son responsables de elegir las soluciones o individuos más aptos para la siguiente generación o iteración del algoritmo. Desempeñan un papel importante en la evolución y mejora de las soluciones. La selección se basa en la evaluación de la calidad de las soluciones de acuerdo con una función objetivo o criterios específicos.

NSGAII

Componente	**Descripción**	**Parámetros**
Random	En este componente, selecciona las soluciones de manera aleatoria.	
Tournament	Componente que selecciona al azar un número determinado de soluciones de la población y luego elige la mejor solución de ese grupo.	tournamentSize
PopulationAndNeighborhood Selection	Componente que realiza una selección basada en la vecindad de cada solución.	probability, neighborhoodSize, replacedSolutions

Tabla 4.8: Componentes de selección para NSGAII

Parámetros	**Descripción**
tournamentSize	Tamaño del torneo.
neighborhoodSize	Tamaño del vecindario.
replacedSolutions	Número de soluciones que se reemplazarán.
probability	Posibilidad de que una solución dentro del vecindario sea seleccionada.

Tabla 4.9: Parámetros de los componentes de selección para NSGAII

MOPSO

Componente	**Descripción**	**Parámetros**
FrecuencySelection	En este componente, selecciona las soluciones de manera aleatoria basándose en la frecuencia de mutación. Es específico para la mutación en el MOPSO.	frecuency

Tabla 4.10: Componentes de selección para MOPSO

Parámetros	Descripción
frecuency	Frecuencia de mutacion.

Tabla 4.11: Parámetros de los componentes de selección para MOPSO

4.3.4. Componentes de variación

Los componentes de variación realizan las operaciones sobre las soluciones existentes con el fin de generar nuevas soluciones en cada iteración. El objetivo de estos generar diversidad entre la población con el fin de evitar la convergencia en soluciones subóptimas y la creación de soluciones mejores. Estos componentes predominan en los algoritmos evolutivos.

NSGAII

Componente	Descripción	Parámetros
Crossover	Componente que realiza el cruce entre 2 individuos de la población.	type, offspringPopulationSize, probability, repairStrategy
Mutation	Componente que genera cambios aleatorios en los individuos.	type, probability, repairStrategy, perturbation, distributionIndex
DifferentialEvolution Crossover	Componente que utiliza una mutación diferencial para combinar un individuo objetivo con un vector de prueba generado por 3 individuos aleatorios.	offspringPopulationSize, probability

Tabla 4.12: Componentes de variación para NSGAII.

Parámetros	Descripción
offspringPopulationSize	Tamaño de la población de descendientes generados.
probability	Posibilidad de efectuar la mutación o el cruce sobre un individuo.
repairStrategy	Si una solución generada viola ciertas restriccio- nes, se aplica una estrategia de reparación para modificar la solución y hacerla válida o factible.
perturbation	Introduce cambios aleatorios en los valores de los genes.
distributionIndex	Determina la tasa de mezcla entre los valores de los padres durante el cruce.

Tabla 4.13: Parámetros de los componentes de variación para NSGAII

MOPSO

Componente	Descripción	Parámetros
Mutation	Componente que genera cambios aleatorios en las partículas.	type, probability, repairStrategy, perturbation, distributionIndex

Tabla 4.14: Componentes de variación para MOPSO.

Parámetros	Descripción
probability	Posibilidad de efectuar la mutación o el cruce sobre una partícula.
repairStrategy	Si una solución generada viola ciertas restriccio- nes, se aplica una estrategia de reparación para modificar la solución y hacerla válida o factible.
perturbation	Introduce los cambios aleatorios en las características de una partícula.
distributionIndex	Determina la tasa de mezcla entre los valores de los padres durante el cruce.

Tabla 4.15: Parámetros de los componentes de variación para MOPSO

4.3.5. Componentes de posición y velocidad en el MOPSO

En el algoritmo MOPSO para controlar el movimiento se utilizan los componentes de posición y de velocidad. Cada partícula tiene asociada una posición que representa una solución, la cual tendrá que actualizarse en cada iteración. Además, al igual que los componentes de posición con la posición de una partícula, los componentes de velocidad tendrá que actualizar su velocidad, que está influenciada por la inercia de la misma partícula.

Componente	**Descripción**	**Parámetros**
VelocityUpdate	Este componente, actualiza las velocidades de las partículas utilizando la mejor posición individual y la mejor posición global.	type, C1Min, C2Min, C1Max, C2Max
PositionUpdate	Este componente se encarga de actualizar las posiciones de las partículas a partir de la velocidad.	upperLimit, lowerLimir
GlobalBestUpdate	Este componente, actualiza la mejor posición global del enjambre.	
LocalBestUpdate	Este componente se encarga de actualizar las me- jores posiciones individuales de cada partícula.	
InertiaWeightStrategy	Este componente define la estrategia para calcular la inercia, determinando la influencia de la velocidad antigua al actualizar la velocidad.	type, weight

Tabla 4.16: Componentes de posición y velocidad.

Parámetros	**Descripción**
C1Min, C2Min, C1Max, C2Max	Límites de los coeficientes de aprendizaje que se utilizanpara actualizar la velocidad de una partícula.
upperLimit y lowerLimit	Indican la medida en la que se debe modificar la velocidad, al acercarse a los límites.
weight	Afecta la magnitud y dirección de la velocidad de la partícula en cada iteración.

Tabla 4.17: Parámetros de los componentes de posición y velocidad

Capítulo 5

Modelado de algorimos con Blockly

Continuando con la idea de definir los bloques mediante componentes desarrollada por Daniel Del Castillo de La Rosa [15] y aprovechando la nueva versión de jMetal [16] que permite el desarrollo de algoritmos a partir de componentes, se han definido bloques para representar, manera general, componentes extraídos de jMetal. Esto introduce la posibilidad de modelar algoritmos que tengan la misma configuración requerida para jMetal-auto [4], pero también de poder crear algoritmos personalizados debido al carácter general de la mayoría de los bloques.

5.1. Blockly

Blockly [2] es una biblioteca desarrollada por Google que permite la creación de editores visuales de programación mediante bloques que se pueden arrastrar y soltar. Esta biblioteca ofrece una interfaz gráfica intuitiva y visualmente agradable. La principal característica de Blockly es su capacidad para crear programas utilizando bloques gráficos interconectados. Los bloques se pueden arrastrar y soltar en un área de trabajo, para luego enlazarlos entre sí para formar un programa completo.

Blockly es altamente personalizable y se puede adaptar a diferentes lenguajes de programación y dominios específicos. Permite crear editores visuales para una amplia variedad de aplicaciones, como la programación de robots, el desarrollo de juegos, la creación de simulaciones y más. En Blockly existe la posibilidad de generar código en diferentes lenguajes de programación, como JavaScript, Python, PHP, entre otros. Blockly se utiliza ampliamente en entornos educativos, plataformas de aprendizaje en línea y herramientas de desarrollo de software. Su enfoque visual y su facilidad de uso lo convierten en una herramienta poderosa para enseñar y aprender los conceptos de programación.

En Othimi, Blockly se utiliza para su interfaz de creación de problemas. Blockly se integra en su entorno para permitir a los usuarios diseñar problemas utilizando una interfaz gráfica basada en bloques, de manera que no tengan que programar en ningún momento. En este TFG, Blockly se utilizará para el modelado de algoritmos, aprovechando la nueva interfaz creada por Alejandro Lugo Fumero [21] que rehizo desde cero la interfaz para mejorarla en comparación a la existente en Prodef.

5.2. Implementación de los bloques en la interfaz gráfica

Lo primero a destacar son los cambios introducidos con el objetivo de mejorar la interfaz de modelado de problemas:

- **Ubicación de la definición de los bloques.** En Prodef los bloques para el modelado de algoritmos se encontraban implementados en el resolutor y para que aparecieran en el área de trabajo de Blockly, se enviaba una petición desde el resolutor al *front-end* para obtenerlos y cargarlos. Esto no era una práctica eficiente, ya que se estaba almacenando contenido estático en el resolutor y enviándolo a la hora de cargar la página. Además, obligaba al usuario a tener en funcionamiento el resolutor para poder acceder al modelado de algoritmos cuando en realidad no son funcionalidades que deberían ser totalmente dependientes la una de la otra. Por este motivo se ha tomado la decisión de definir los bloques en el Front-end de manera que el resolutor y el apartado de modelado de algoritmos sean independientes.

- **Parametrización.** Se ha cambiado totalmente la especificación de parámetros, dejando de ser independiente del proceso de modelado de un algoritmo para formar parte de este. En Othimi se posibilita que el usuario realice este proceso durante el modelado del algoritmo, para que a la vez que se van uniendo los bloques se le permita al usuario obtener información acerca de los distintos tipos y parámetros de estos y asignar sus valores deseados. Esto se ha realizado añadiendo a los bloques campos desplegables donde seleccionar las opciones mostradas y campos numéricos para introducir los valores como se puede observar en la Figura 5.1.

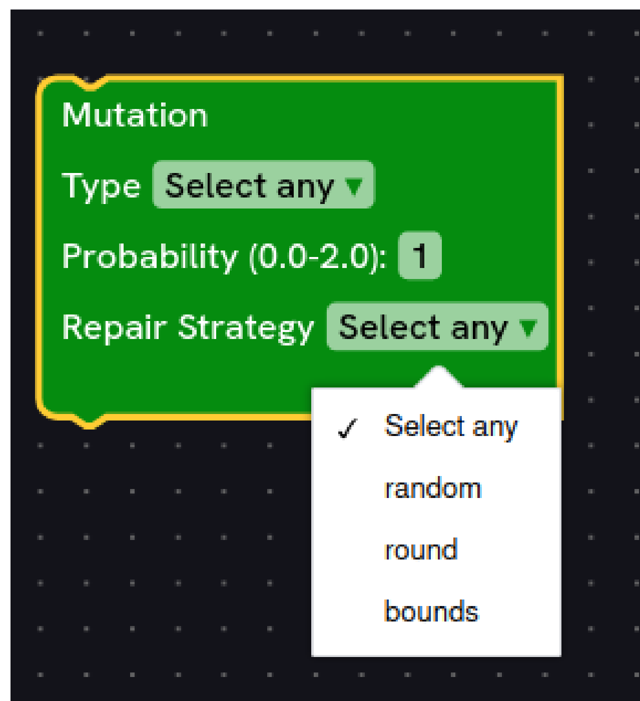


Figura 5.1: Campos desplegables y numéricos

5.2.1. Clasificación de los bloques

Tras definir una aproximación en el capítulo 4 de los componentes y parámetros que hay que incluir para poder ejecutar un algoritmo con jMetal, es el momento de empezar con la creación de los bloques. La idea principal es que un usuario pueda crear un algoritmo de manera autónoma, pero que también tenga la posibilidad de usar una plantilla por defecto en la que ya estén unidos los bloques necesarios para una correcta ejecución en jMetal, de manera que solamente habría que realizar la parametrización como se muestra en la Figura 5.10.

Se ha decidido clasificar los bloques en clases para que el usuario los pueda encontrar dentro del área de trabajo en función del proceso que realiza cada bloque. Obviamente, con el desarrollo de Othimi se necesitará crear nuevas clases o incluso crear agrupaciones más generales en función de las necesidades de la herramienta:

- **Termination.** Contiene los bloques que dictarán el criterio de parada de un algoritmo.

- **Initialization.** Se encuentran los bloques para generar las soluciones o características iniciales. Este tipo de bloques es indispensable a la hora de definir un algoritmo. En estos momentos contiene un bloque para la inicialización de una población y la de un enjambre de partículas.

- **Evaluation.** Contiene los bloques que tendrán como función medir la calidad y el rendimiento de cada solución.

- **Selection.** Este tipo de bloques se encarga de elegir las mejores soluciones en cada iteración. Encontramos bloques de selección, que se utilizan para la selección de individuos para realizar las operaciones de mutación y cruce, pero también hay existe uno para realizar una selección global entre las soluciones.

- **Variation.** Contiene los bloques que realizan modificaciones y alteraciones de la población, generando diversidad entre las soluciones. Estos bloques serían los bloques de mutación y de cruce.

- **Update.** Este tipo de bloques realizan funciones de actualización dentro del algoritmo que pueden ser sobre las soluciones u otros parámetros del proceso. Actualmente, se encuentran los bloques como los de actualización de velocidad y posición para el algoritmo PSO.

5.2.2. Modificación dinámica y limitación de parámetros

En jMetal hay componentes como por ejemplo los de selección mostrados, que dependiendo del tipo de selección que se quiera aplicar tendrá unos parámetros específicos que en general no son comunes con otros tipos de selección. Además, jMetal solo permite un intervalo de valores para los parámetros, por lo que si introducimos valores incorrectos fallará a la hora de ejecutar el algoritmo, lo que estropearía la experiencia del usuario a la hora de usar Othimi.

Limitación de valores

En cuanto a la limitación de los valores de los parámetros, es bastante sencillo de implementar. Las medidas que se han tomado son las siguientes:

- Mostrar visualmente en los bloques el intervalo permitido para cada uno de sus parámetros.
- En la definición de los bloques se ha programado cuáles deben ser los valores mínimos, máximos y su valor por defecto. Si el usuario intenta insertar un valor incorrecto, el campo volverá a su valor por defecto.

A continuación tenemos al parámetro de la población, el cual debe ser mayor a 10 e inferior a 200, siendo su valor por defecto 100.

```
{
    type: "field_number",
    name: "population",
    value: 100,
    min: 10,
    max: 200
}
```

Modificación dinámica

Para la modificación de parámetros de un bloque de manera dinámica, se ha realizado las siguientes implementaciones.

- Se han creado funciones llamadas `applyOnChangeTo`, las que dependiendo del valor del campo `type` eliminan y añaden los parámetros necesarios para un bloque determinado.
- Se ha implementado un `Listener` que actúa sobre el *workspace* de la interfaz, el cual aplica las funciones `applyOnChangeTo` a sus bloques correspondientes cuando se produce la creación de un bloque. De esta manera, la modificación de la estructura de los bloques se realiza de manera dinámica al interactuar con la interfaz.

Como resultado, obtenemos diferentes configuraciones de bloques como se muestra en la Figura5.2 con lo bloques de selección, evitando que el usuario utilice parámetros erróneos, resultando un error de ejecución en jMetal.

```
workspace.addChangeListener((event: { type: string; blockId: string; }) => {
      if (event.type === Blockly.Events.BLOCK_CREATE) {
        const newBlock = workspace.getBlockById(event.blockId);
        if (newBlock?.type === 'mutation_definition') {
          applyOnChangeToMutationBlock(newBlock);
        } else if (newBlock?.type === 'crossover_definition') {
          applyOnChangeToCrossoverBlock(newBlock);
        } else if (newBlock?.type === 'velocity_definition') {
          applyOnChangeToVelocityBlock(newBlock);
        } else if (newBlock?.type === 'global_selection_definition') {
          applyOnChangeToGlobalSelectionBlock(newBlock);
        } else if (newBlock?.type === 'selection_definition') {
          applyOnChangeToSelectionBlock(newBlock);
        }
      }
    });
```

```
function applyOnChangeToSelectionBlock(block: Blockly.BlockSvg | Blockly.Block | null) {
      block?.setOnChange(() => {
        const typeField = block.getField('type');
        const input = block.getInput('dummy2');
        if (typeField?.getValue() === 'random') {
          if (input?.fieldRow) {
            while (input?.fieldRow.length > 0) {
              const field = input?.fieldRow[0];
              if (field.name) {
                input.removeField(field.name);
              }
            }
          }
        } else if (typeField?.getValue() === 'tournament'){
          if (!block.getField('TSize')) {
            if (input?.fieldRow) {
              while (input?.fieldRow.length > 0) {
                const field = input?.fieldRow[0];
                if (field.name) {
                  input.removeField(field.name);
                }
              }
            }
            input?.appendField('Size (2-10):','TSize').appendField(new Blockly.
    FieldNumber(2, 2, 10), 'size');
          }
        } else if (typeField?.getValue() === 'neighborhood'){
          if (!block.getField('Probability')) {
            if (input?.fieldRow) {
              while (input?.fieldRow.length > 0) {
                const field = input?.fieldRow[0];
                if (field.name) {
                  input.removeField(field.name);
                }
              }
            }
          input?.appendField('Probability (0.0-1.0):','Probability').appendField(new
    Blockly.FieldNumber(1, 0, 1), 'probability');
\subsubsection{          input?.appendField('Size (5-50):','NSize').appendField(new
    Blockly.FieldNumber(5, 5, 50), 'size');}
          input?.appendField('Replaced Solutions (1-5):','Replaced').appendField(new
    Blockly.FieldNumber(3, 1, 5), 'solutions');
          }
        }
      });
    }

```

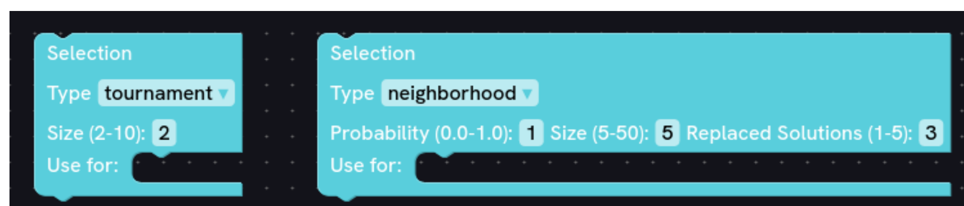


Figura 5.2: Enter Caption

5.2.3. Bloques desarrollados

A continuación se mostrarán y describirán los bloques desarrollados, así como el planteamiento a la hora de crearlos:

Bloque de terminación por repetición

Este bloque toma como base el componente de `TerminationByEvaluations` el cual tiene como único parámetro el número de evaluaciones y un `input` en el que unir los bloques que realizarán el proceso.

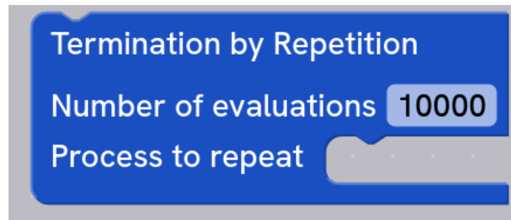


Figura 5.3: Bloque de terminación

Bloques de inicialización

En cuanto a los bloques de inicialización, encontramos dos bloques distintos. El primero, `Create initial solutions`, es uno general que está destinado a crear las soluciones iniciales, teniendo que seleccionar el método y el tamaño. El bloque `Swarm initialization`, por otro lado, está destinado a efectuar la inicialización en el PSO, ya que como se puede apreciar en la Figura 5.4 a diferencia del bloque anterior, se inicializa la velocidad y las mejores soluciones locales y globales.

Figura 5.4: Bloques de inicialización

Bloque de evaluación

Para realizar el proceso de evaluación de las soluciones se ha definido un bloque muy simple, en el cual lo único que hay que realizar es la selección del tipo de evaluación que se hará.

Figura 5.5: Bloque de evaluación

Bloques de selección

En cuanto a los bloques de selección, se han tenido que crear 3 bloques distintos:

- **Selection.** Se trata del bloque de selección general, en el que dependiendo del tipo de selección que se requiera, sus parámetros cambiarán, como ya pudo ver en la figura 5.2. Este incluye un `input` en el que se deberán conectar los bloques que harán uso de las soluciones seleccionadas.

- **Frecuency mutation selection.** Este bloque realiza lo mismo que el bloque general descrito antes, sin embargo, para realizar la configuración del PSO, jMetal necesita una selección totalmente específica para la mutación. Como es específico para jMetal y dependiente de que exista mutación, se ha optado por crear un bloque individual para representar este tipo de selección.

- **Global solution selection.** A diferencia del los dos anteriores, este bloque está destinado a obtener soluciones de manera global. Un ejemplo de esto sería la selección en el PSO de las mejores posiciones globales para incluirlas en el Pareto.

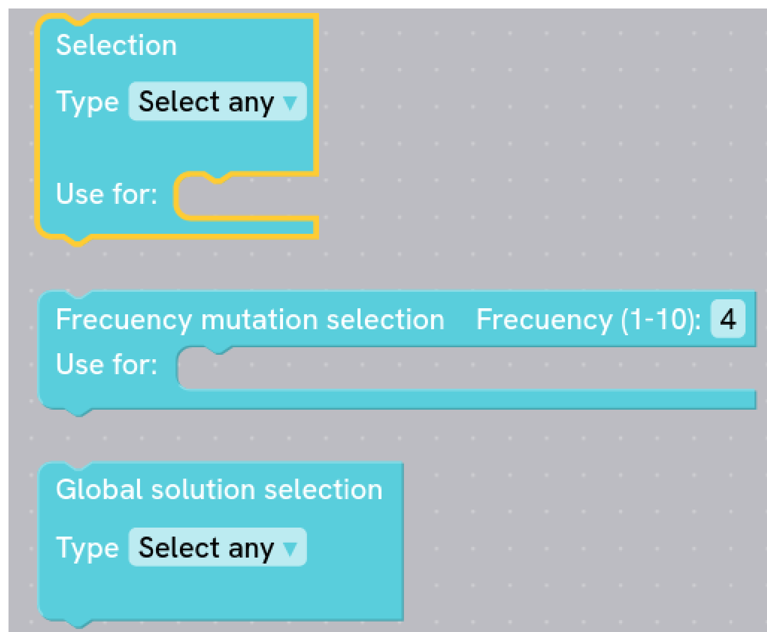


Figura 5.6: Bloques de selección

Bloques de variación

En los componentes de variación de jMetal, los operadores de mutación y cruce, se muestran como operadores dependientes el uno del otro. Sin embargo, en los algoritmos evolutivos la mutación y el cruce son operadores que no tienen la obligación de ser

dependientes entre sí. Aunque generalmente el cruce se realice antes que la mutación, también se pueden dar casos que el orden cambie. Debido a esto se han creado bloques independientes para la mutación y el cruce. Es necesario conectar estos bloques a un bloque de selección, ya que no se entendería a qué individuos se les está realizando la operación. Al igual que con otros bloques, sus parámetros cambian dependiendo del tipo de cruce o mutación seleccionado.

- **Crossover.** Este bloque realiza el proceso de cruce de la población seleccionada y obtiene como resultado una nueva población que será del tamaño especificado.
- **Diferential Evolution Crossover.** Este bloque realiza la misma función que el anterior, no obstante se trata de un cruce para un tipo de variación diferente, la diferencial.
- **Mutation.** Se trata del bloque general para el operador de mutación, que realiza cambios en las características de los individuos seleccionados.

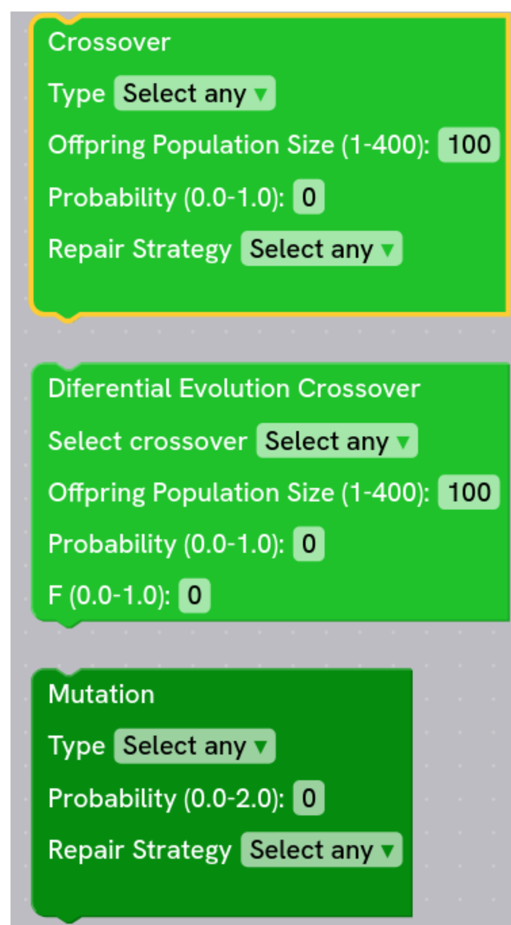


Figura 5.7: Bloques de mutación y cruce

Bloques de actualización

Estos bloques se encargan de realizar las actualizaciones pertinentes a los parámetros de un algoritmo. Principalmente, estos bloques han sido desarrollados para su uso en el PSO.

- **Velocity update.** Es un bloque que unifica el componente `InertiaWeightStrategy` con `VelocityUpdate` de jMetal. Esto se debe a que la inercia, se utiliza específicamente para calcular la velocidad, por lo tanto, no es necesario definirla como un bloque independiente y se ha terminado añadiendo como un parámetro al bloque de velocidad.

- **Position update.** Este bloque tiene como objetivo la actualización de la posición de las partículas en el PSO. En este, se debe especificar en que medida se produce el cambio de la velocidad, pudiendo insertar valores negativos y positivos.

- **Global y Local best update.** El objetivo de estos bloques es actualizar las soluciones globales o locales que tiene el algoritmo.

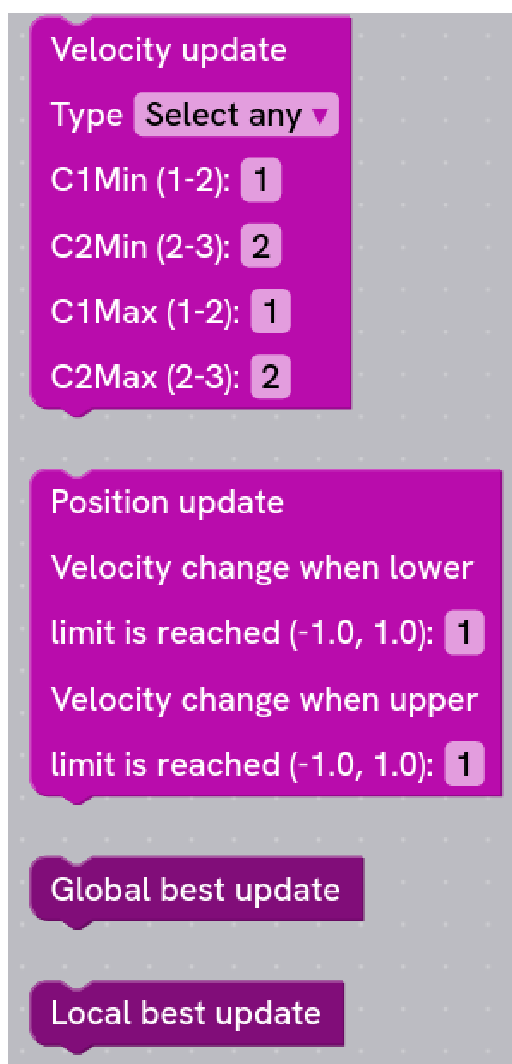


Figura 5.8: Bloques de actualización de posición, velocidad y mejores soluciones

5.3. Plantillas predefinidas

Para facilitar la configuración de algoritmos para su ejecución en jMetal, se han creado unas plantillas predefinidas. Como se muestra en las Figuras 5.9 y 5.10, al seleccionar el algoritmo en el desplegable aparecerá en la interfaz un algoritmo evolutivo por defecto, en

el cual el usuario puede cambiar algunos bloques, como el de selección y el de *crossover*, dependiendo de la configuración del algoritmo que pretenda utilizar. No obstante, hay bloques, como por ejemplo el de creación de soluciones iniciales, distinguible en la Figura 5.10 por su color rojo, que ha sido modificado para que no se pueda mover ni eliminar de la plantilla. Esto es debido a que se trata de un proceso indispensable para los algoritmos evolutivos, además de que resultaría en un error de ejecución a la hora de intentar ejecutarlo con jMetal.

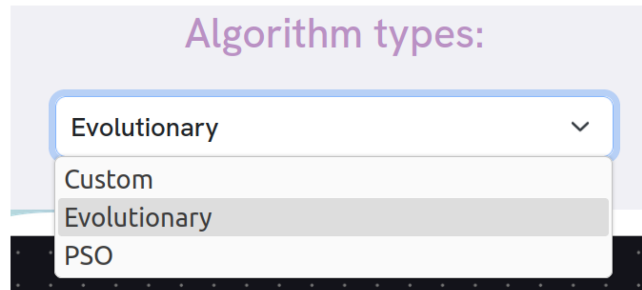


Figura 5.9: Selección de la plantilla de algoritmo

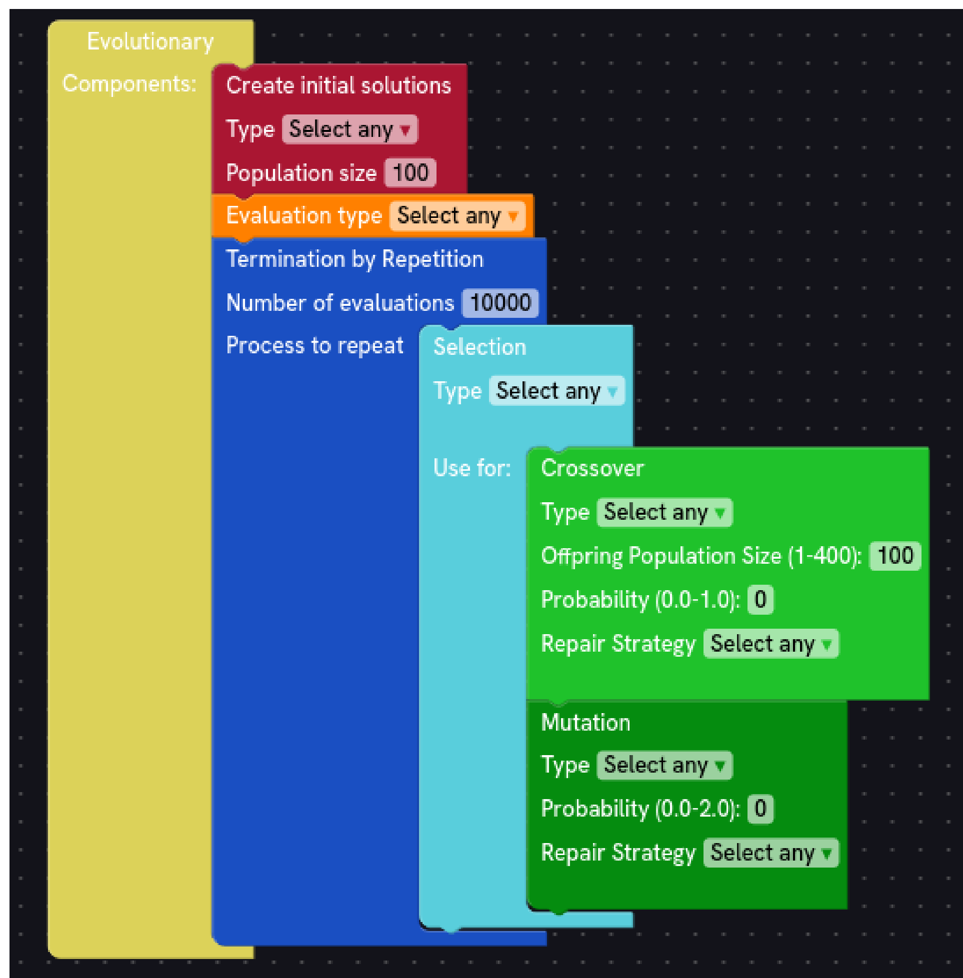


Figura 5.10: Plantilla algoritmo evolutivo

5.4. Modelado manual de algoritmos

Este TFG se ha centrado principalmente en crear bloques a partir de los componentes definidos en jMetal para que sean compatibles con los algoritmos definidos en jMetal-auto. No obstante, la idea era definir bloques de manera generaliza para que también puedan usarse para crear algoritmos de manera manual e independientes a jMetal. Como se comentó anteriormente, la interfaz permite a los usuarios modelar algoritmos a partir de plantillas predefinidas, sin embargo, los usuarios tienen posibilidad en Othimi de modelar su propio algoritmo, al igual que en la versión antigua Prodef.

En la Figura 5.11 se muestra el modelo de un algoritmo memético de Prodef diseñado por Daniel Del Castillo de La Rosa [15]. El algoritmo empieza generando un determinado número de soluciones y luego repite varias veces el proceso, creando nuevas soluciones a partir de las existentes y aplicando una búsqueda local a estas. Si nos fijamos en la Figura 5.12, se ha modelado un algoritmo casi idéntico con los nuevos bloques creados en Othimi. En este se crean las soluciones iniciales de forma aleatoria para posteriormente realizar un cruce entre soluciones existentes para generar nuevas soluciones y buscar la mejor solución local.

En conclusión, tenemos una interfaz en la que el usuario es capaz de modelar en Othimi algoritmos compatibles para su ejecución en jMetal, como modelos del antiguo Prodef. Resultando en una mayor cantidad de opciones a la hora del modelado de algoritmos.

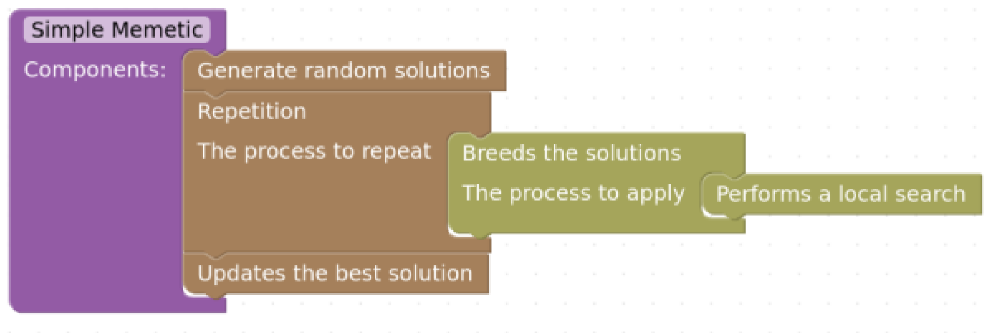


Figura 5.11: Algoritmo memético en Prodef

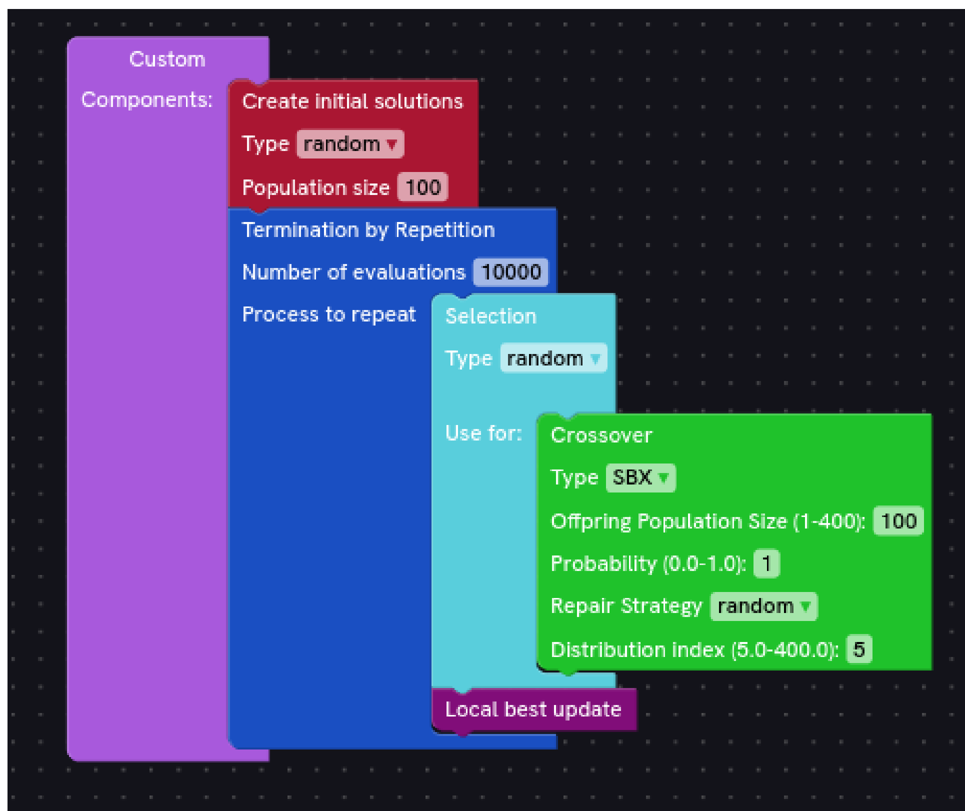


Figura 5.12: Algoritmo memético en Othimi

Capítulo 6

Ejecución de un algoritmo con jMetal-auto

El objetivo principal de este Trabajo de Fin de Grado era poder obtener un archivo a partir de la interfaz de modelado de problemas, el cual contenga la configuración necesaria para su ejecución con jMetal-auto [4]. La configuración del algoritmo vendría dada por los bloques usados por el usuario y los valores de sus parámetros. Para ello se hizo un estudio de los componentes y parámetros necesarios, además de un traslado de estos componentes y parámetros de forma coherente y generalizada a bloques de Blockly [2], ya que los bloques pueden definir algoritmos exactamente iguales a jMetal [16], pero también tienen que poder usarse para definir algoritmos con estructuras diferentes a los de jMetal.

6.1. Obtener un archivo ejecutable con jMetal a partir de Prodef

El primer paso es realizar la instalación y puesta en marcha de Othimi como se explica en el capítulo 3. Tras esto tendremos que iniciar sesión con el método preferido para poder acceder a la zona de trabajo donde poder modelar el algoritmo.

Para poder obtener el archivo, se debe crear un algoritmo y especificar los parámetros correspondientes durante su modelado. En este caso se partirá a partir de un algoritmo evolutivo mostrado en la Figura 6.1.

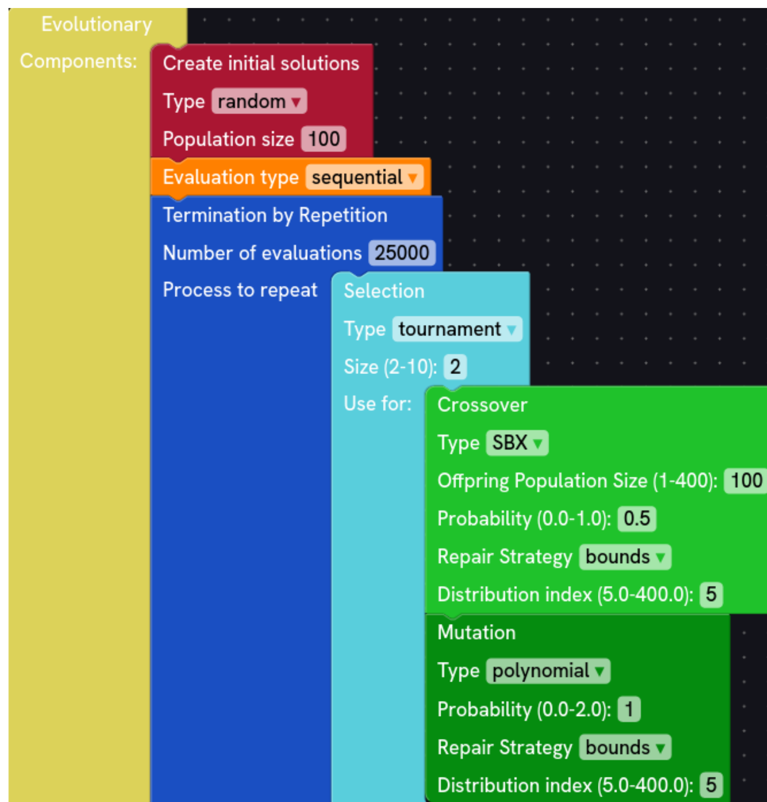


Figura 6.1: Plantilla algoritmo evolutivo

Tras definir el algoritmo usando los mismos bloques y especificando con parámetros los valores observables en la Figura 6.2, si se hace clic en el botón de denominado "`Export jMetal file`" se descarga automáticamente un fichero con extensión .java con la configuración realizada en la interfaz gráfica. El contenido del fichero debería ser el siguiente:

```
public class NSGAIIConfiguredFromAParameterString {

  public static void main(String[] args) {
    String referenceFrontFileName = "resources/referenceFrontsCSV/ZDT1.csv";

    String[] parameters =
        ("--problemName org.uma.jmetal.problem.multiobjective.zdt.ZDT1 "
            + "--randomGeneratorSeed 12 "
            + "--referenceFrontFileName " + referenceFrontFileName + " "
            + "--maximumNumberOfEvaluations 25000 "
            + "--algorithmResult population "
            + "--populationSize 100 "
            + "--offspringPopulationSize 100 "
            + "--createInitialSolutions random "
            + "--variation crossoverAndMutationVariation "
            + "--selection tournament "
            + "--selectionTournamentSize 2 "
            + "--rankingForSelection dominanceRanking "
            + "--densityEstimatorForSelection crowdingDistance "
            + "--crossover SBX "
            + "--crossoverProbability 0.5 "
            + "--crossoverRepairStrategy bounds "
            + "--sbxDistributionIndex 5 "
            + "--mutation polynomial "
            + "--mutationProbabilityFactor 1.0 "
```

```java
26              + "--mutationRepairStrategy bounds "
27              + "--polynomialMutationDistributionIndex 5 ")
28              .split("\\s+");
29
30      AutoNSGAII autoNSGAII = new AutoNSGAII();
31      autoNSGAII.parse(parameters);
32
33      AutoNSGAII.print(autoNSGAII.fixedParameterList());
34      AutoNSGAII.print(autoNSGAII.configurableParameterList());
35
36      EvolutionaryAlgorithm<DoubleSolution> nsgaII = autoNSGAII.create();
37
38      EvaluationObserver evaluationObserver = new EvaluationObserver(1000);
39      RunTimeChartObserver<DoubleSolution> runTimeChartObserver =
40          new RunTimeChartObserver<>(
41              "NSGA-II", 80, 100,
42              referenceFrontFileName, "F1", "F2");
43
44      nsgaII.observable().register(evaluationObserver);
45      nsgaII.observable().register(runTimeChartObserver);
46
47      nsgaII.run();
48
49      JMetalLogger.logger.info("Total computing time: " + nsgaII.totalComputingTime()); ;
50
51      new SolutionListOutput(nsgaII.result())
52          .setVarFileOutputContext(new DefaultFileOutputContext("VAR.csv", ","))
53          .setFunFileOutputContext(new DefaultFileOutputContext("FUN.csv", ","))
54          .print();
55    }
56  }
```

Como se pudo observar se debería obtener un algoritmo configurado a partir de una cadena de parámetros en la cual están incluidos los componentes y los valores de sus parámetros para ejecutar la clase `NSGAIIConfiguredFromAParameterString`.

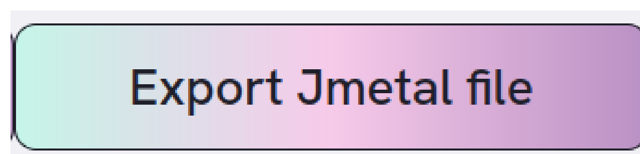


Figura 6.2: Botón para exportar el fichero

6.2. Ejecución con jMetal

6.2.1. Instalación de jMetal

Se necesita descargar el framework de jMetal, para ello clonamos el repositorio de jMetal [23]. Ejecutando en la terminal el siguiente comando en la ubicación donde se quiera realizar la clonación:

- git clone git@github.com:jMetal/jMetal.git

Se debería descargar la herramienta correctamente. Para continuar la instalación se debe compilar el código del proyecto, ya que se trata de un proyecto Maven. No obstante, se deben cumplir los siguientes requerimientos para poder usar jMetal.

- Java 14+ JDK [3]

- Maven [6]

Para trabajar con jMetal, se recomienda utilizar un entorno de desarrollo integrado (IDE), ya que simplifica en gran medida el proceso. Solamente es necesario importar el proyecto de jMetal. En mi caso, he utilizado Visual Studio Code [11], pero jMetal también es compatible con otros IDE como IntelliJ Idea, Netbeans o Eclipse, ofreciendo así diversas opciones para los desarrolladores.

Si se quiere utilizar Maven en Visual Studio Code es fundamental descargar la extensión llamada *Extension Pack for Java*. Esta extensión instalará todas las extensiones necesarias para trabajar con Java de manera sencilla. De esta forma, con solo abrir el proyecto se realizará todo el proceso de compilación de manera automática y sin errores.

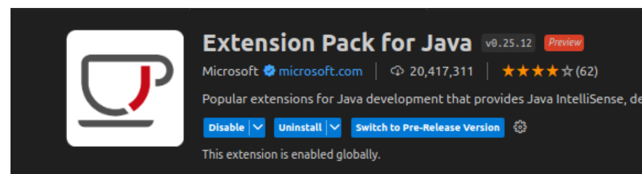


Figura 6.3: Extensión para Java

6.2.2. Ejecución del archivo

Para poder realizar la ejecución, lo primero que se debe hacer es colocar el archivo en el directorio correcto dentro del proyecto de jMetal, en este caso el directorio `examples` de jMetal-auto. No es totalmente necesario que se sitúe en este mismo directorio, sin embargo, habría que cambiar manualmente las direcciones de los *import*s del archivo. La ruta para llegar al directorio `examples` es la siguiente:

- jMetal/jmetal-auto/src/main/java/org/uma/jmetal/auto/autoconfigurablealgorithm/examples

Una vez que el archivo esté ubicado correctamente, se puede proceder con la ejecución. Para ello, se abre el archivo y luego se hace clic en el botón con forma de “play” ubicado en la esquina superior derecha de la interfaz de VSCode como señala la Figura 6.4. Esto ejecutará directamente el algoritmo, siempre y cuando no se haya cometido ningún error al definirlo en la interfaz de Othimi.

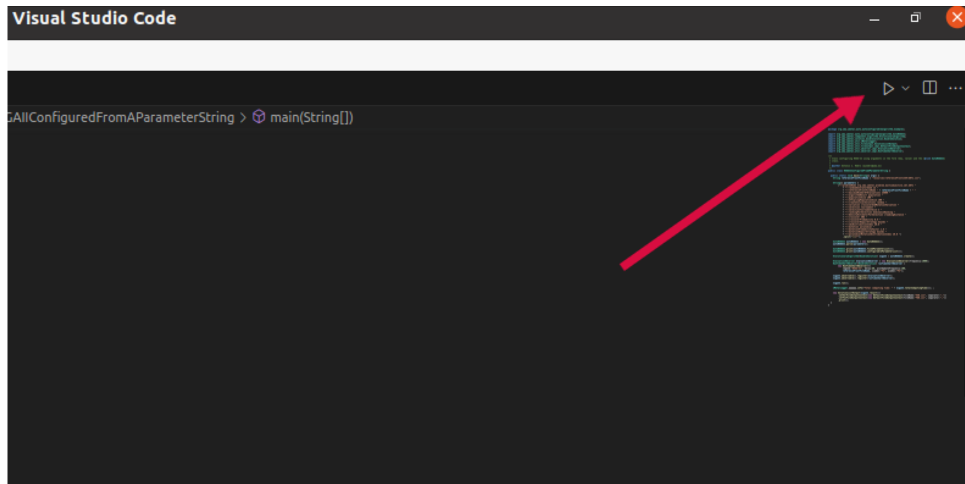


Figura 6.4: Botón para ejecutar

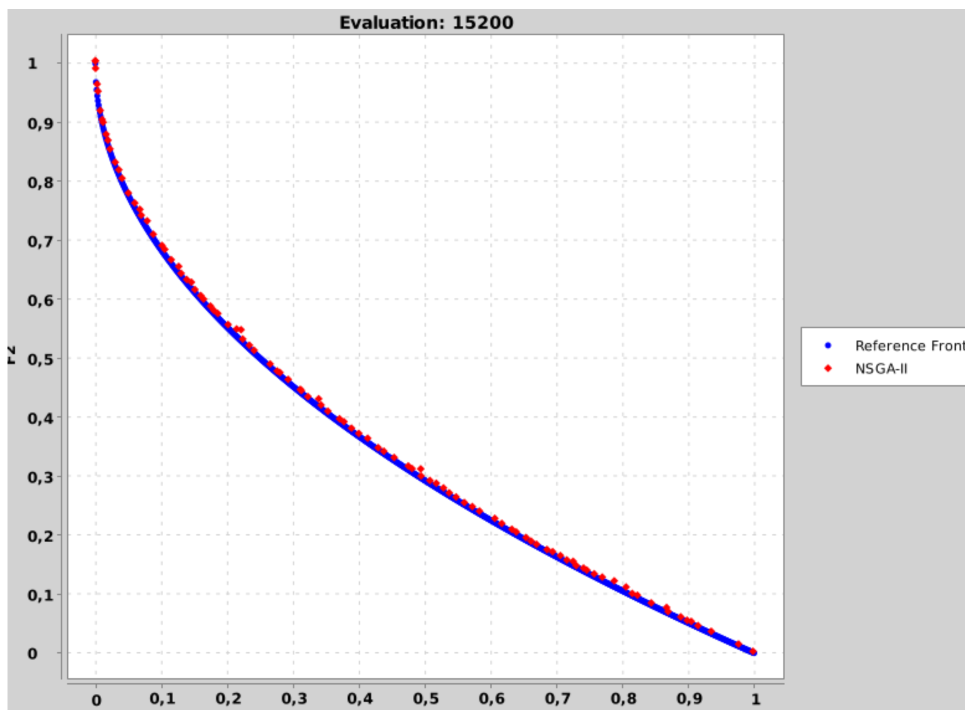


Figura 6.5: Ejecución del algoritmo

En la Figura 6.5 se puede observar la demostración gráfica que realiza jMetal durante la ejecución del algoritmo, finalizándose en el momento en el que se alcanza el criterio de terminación. Este gráfico, en algoritmos de optimización multiobjetivo, representa el espacio de objetivos, donde cada punto en el gráfico corresponde a una solución encontrada. Por otro lado, la línea, el Pareto, que es el conjunto de soluciones óptimas.

El problema de optimización con el que se está resolviendo el algoritmo es el ZDT1 [27], acrónimo de "Zitzler-Deb-Thiele Function 1", es un problema clásico utilizado en la evaluación de algoritmos de optimización multiobjetivo. Tiene como objetivo encontrar un conjunto de soluciones que optimicen dos funciones objetivo: la minimización de la función f1(x) y la maximización de la función f2(x). Una mejora a futuro podría ser el análisis de los problemas de optimización de jMetal y añadir la funcionalidad de seleccionar el problema que se quiere resolver, ya que de momento simplemente se ejecuta el algoritmo

sobre un problema por defecto y la manera de probar otro sería editar el archivo de forma manual.

Finalmente, las soluciones del problema quedarían guardadas en un archivo con formato csv, ubicado en la raíz del proyecto. Para el problema ZDT1, podemos observar en la Figura 6.6 que el archivo contiene 2 columnas, las cuales son las soluciones para las funciones f1(x) y f2(x) del problema.

	Predeterminado	Predeterminado
1	0.9999978126774433	4.0811390869785756E-4
2	1.662854465234176E-9	1.0012864818534775
3	0.7527085708805038	0.13375115684322958
4	0.4060082851414054	0.36497725584386487
5	0.0015623671098280029	0.9620637740714628
6	0.556426840466525	0.25503518197885117
7	0.7220456573164497	0.15087337988999439
8	0.7845012893466383	0.11502133919971408
9	0.02226478675185839	0.8523404618701454
10	0.4257200320774685	0.34918656792017444
11	0.9258563192236938	0.038435841308094645
12	0.22682596789121928	0.5249167483168253
13	0.3792092074805792	0.3945396183041339
14	0.003984725974666425	0.9379880452712877
15	0.6365650025592331	0.20317651947161466
16	0.6532800409952613	0.19272270896444055
17	0.21627250791442343	0.5384515792426411
18	0.6878232557527277	0.17122143167956302
19	0.0416510094141434	0.7967792671055125
20	0.016145194024803557	0.8763922332695298
21	0.0497614943327071	0.7782208705715895
22	0.8055363639405781	0.10311747838430868
23	0.704684888629079	0.1617640107466018

Figura 6.6: Soluciones al problema

Capítulo 7

Conclusiones y líneas futuras

Este Trabajo de Fin de Grado tenía como objetivo continuar con la remodelación de la interfaz gráfica de Prodef a su nueva versión Othimi. Los cambios implementados al apartado de modelado de problemas, no solo abren la posibilidad de poder definir algoritmos multi-objetivos, algo que no se podía hacer anteriormente, sino que también es compatible para la resolución de problemas de optimización con jMetal. Esto otorga una mayor variedad de opciones a los usuarios, ya que pueden definir sus propios algoritmos usando los nuevos bloques desarrollados a partir de Blockly o hacer uso de las plantillas predeterminadas en las que solamente es necesario especificar los parámetros. Todo esto contribuye a que las personas puedan aprender sobre la metaheurística y captar el interés de nuevos usuarios.

7.1. Conclusiones

Para completar este Trabajo de Fin de Grado ha sido necesario realizar un análisis y comprensión exhaustiva de la herramienta jMetal, ya que era necesario conocer los componentes y parámetros que utiliza jMetal-auto para la configuración de sus algoritmos. Al mismo tiempo, se estudiaron los algoritmos evolutivos como el PSO para conocer su funcionamiento y sus características. Los conocimientos adquiridos se utilizaron a la hora del diseño de los bloques, creando bloques basados en componentes de jMetal, de manera que fueran compatibles con las configuraciones de algoritmos de jMetal-auto, pero con las suficientes características generales para poder utilizarlos a la hora de modelar una gran variedad de algoritmos con estructuras diferentes a los de jMetal-auto. Se han arreglado problemas de eficiencia de la herramienta en comparación a su versión anterior, ya que los bloques ahora están definidos en el *front-end* evitando su carga desde el resolutor, siendo una práctica ineficiente. También se han creado plantillas predefinidas de algoritmos para facilitar a los usuarios el uso de la herramienta y su nueva funcionalidad para jMetal. Otro añadido es que ahora el proceso de parametrización se realiza durante el modelado del algoritmo, proporcionando al usuario información útil sobre los tipos de cada bloque y sus parámetros. Sin embargo, hay funcionalidades que habrían sido muy útiles de implementar, como el almacenamiento de algoritmos generados por el usuario.

Las principales dificultades detectadas durante la realización del Trabajo de Fin de Grado son las siguientes:

- **Analizar todos los componentes y parámetros de jMetal-auto**. Para obtener la información de cuáles eran los componentes necesarios junto con sus parámetros

y los valores permitidos de estos, se tuvo que ir mirando archivo por archivo de código en el repositorio de jMetal lo que supuso mucho esfuerzo y tiempo.

- **Programación con Blockly**. Blockly es una biblioteca que, por un lado, te permite realizar cosas muy originales, pero, sin embargo, en algunos aspectos está muy limitada, sobre todo para realizar modificaciones de la estructura de los bloques, además de la poca información que se puede encontrar en *Internet*.

7.2. Líneas futuras

El Trabajo de Fin de Grado se ha centrado en la remodelación del apartado de modelado de algoritmos, además de añadir funcionalidades como obtener un archivo ejecutable para jMetal. Sin embargo, todo el tiempo se ha invertido en el desarrollo únicamente de la interfaz gráfica, por lo que el resolutor que existía, de momento, no es compatible con esta nueva versión. Durante el desarrollo se han detectado varias mejoras posibles para esta nueva versión de la definición de algoritmos que proporcionarían una calidad más satisfactoria:

- **Sincronizar la nueva interfaz con el resolutor.** Actualmente, no existe una conexión entre la interfaz de modelado de algoritmos y el resolutor, por lo que habría que realizar las modificaciones necesarias para poder enviar los algoritmos creados por un usuario al resolutor y que este comprenda los nuevos bloques.

- **Permitir la selección de los problemas de jMetal.** Dentro del archivo de configuración del algoritmo en formato jMetal se tiene que señalar que problema se quiere resolver. No obstante, en la versión actual se utiliza uno por defecto dependiendo de que algoritmo se trate. Por lo que una buena opción es estudiar que problemas de optimización tiene jMetal y permitir al usuario seleccionar el que quiera.

- **Guardar algoritmos.** Actualmente, no se ha implementado la función de guardar los algoritmos creados por el usuario, que es algo que sería de gran utilidad.

- **Selección de poblaciones.** Uno de los problemas a la hora de implementar la definición de algoritmos con Blockly fue que desde el punto de vista con el que se abordó el desarrollo, el bloque de selección no señala explícitamente a qué población está haciendo dicha selección. Sería interesante hallar una manera de poder hacerlo sin que resulte incompatible con jMetal.

Capítulo 8

Summary and Conclusions

This project aimed to continue the remodeling of the graphical interface of Prodef to its new version, Othimi. The implemented changes in the problem modeling section not only open the possibility of defining multi-objective algorithms, which was not possible before, but also make it compatible with solving optimization problems using jMetal. This provides users with a greater variety of options, as they can define their own algorithms using the new blocks developed with Blockly or make use of pre-defined templates where they only need to specify the parameters. All of this contributes to helping people learn about metaheuristics and attract the interest of new users.

8.1. Conclusions

To complete this project, a thorough analysis and understanding of the jMetal tool was necessary, as it was required to understand how algorithm configurations and their specific components and parameters are defined. At the same time, evolutionary algorithms such as PSO were studied to understand their functionality and characteristics. The acquired knowledge was then applied in the design of the blocks, creating blocks based on jMetal components that are compatible with jMetal-auto algorithm configurations, but also general enough to model a wide variety of algorithms with different structures than those in jMetal-auto. Efficiency issues present in the previous version were addressed by defining the blocks in the front-end, eliminating the need for loading them from the solver, which was an inefficient practice. Predefined algorithm templates were also created to facilitate the use of the tool and its new functionality for jMetal. Additionally, parameter specification is now done during the algorithm modeling process, providing the user with useful information about the types and parameters of each block. However, there are certain functionalities that would have been useful to implement, such as the ability to save algorithms.

The main difficulties encountered during the development of this project are as follows:

- **Analyzing all the components and parameters of jMetal-auto.** To obtain the necessary information about the required components, their parameters, and the allowed parameter values, it was necessary to examine each code file in the jMetal repository, which required considerable effort and time.
- **Programming with Blockly.** Blockly is a library that allows for creative implementations, but in some aspects, it is limited, especially when it comes to modifying the

structure of the blocks. Additionally, there is a lack of available information on the internet.

8.2. Future lines of work

This project has focused on the remodeling of the algorithm modeling section, as well as the addition of features such as generating executable files for jMetal. However, all the development efforts have been focused solely on the graphical interface, so the solver that existed previously is not currently compatible with this new version. During the development process, several possible improvements for this new version of algorithm definition have been identified, which would provide a more satisfactory user experience:

- **Synchronize the new interface with the solver.** Currently, there is no connection between the algorithm modeling interface and the solver, so the necessary modifications need to be made to send the algorithms created by users to the solver and ensure it understands the new blocks.

- **Allow the selection of jMetal problems.** In the jMetal algorithm configuration file, it is necessary to specify the problem to be solved. However, the current version uses a default problem depending on the algorithm. Therefore, it would be beneficial to study the optimization problems available in jMetal and allow users to select the desired problem.

- **Implement algorithm saving.** Currently, the function to save user-created algorithms has not been implemented, but it would be highly useful to have.

- **Population selection.** One issue encountered during the implementation of algorithm definition with Blockly was that the selection block does not explicitly indicate which population it refers to. Finding a way to address this without causing incompatibilities with jMetal would be interesting.

Capítulo 9

Presupuesto

Este capítulo recoge el presupuesto estimado del trabajo. El coste del trabajo viene dado por la cantidad de tiempo empleado en su desarrollo. Para realizar una estimación del precio de la hora de trabajo, se han consultado varias plataformas de comparación de sueldos, dando como resultado un precio de 14€la hora.

Nombre	**Horas**	**Coste(€)**
Análisis de Prodef y Othimi	15	210
Análisis de jMetal y estudio de componentes y parámetros de jMetal-auto	70	980
Estudio de algoritmos metaheurísticos	35	490
Estudio de Blockly	30	420
Desarrollo de los bloques y algoritmos	80	1120
Implementación de la obtención de un archivo ejecutable con jMetal	50	700
Pruebas de ejecuciones con jMetal	20	280
Total	300	4200

Tabla 9.1: Presupuesto

Bibliografía

[1] Apollo graphql. https://www.apollographql.com/. [Accessed 12-July-2023].

[2] Blockly. https://developers.google.com/blockly. [Accessed 28-May-2023].

[3] Java. https://www.oracle.com/es/java/technologies/downloads/. [Accessed 2-July-2023].

[4] jmetal-auto. https://jmetal.readthedocs.io/en/latest/autoconfiguration.html. [Accessed 9-July-2023].

[5] Koa. https://www.npmjs.com/package/koa. [Accessed 12-July-2023].

[6] Maven. https://maven.apache.org. [Accessed 2-July-2023].

[7] Node.js. https://nodejs.org/es. [Accessed 12-July-2023].

[8] React. https://vitejs.dev. [Accessed 12-July-2023].

[9] Ull-prodef. https://github.com/ULL-prodef. [Accessed 11-July-2023].

[10] Vite. https://vitejs.dev. [Accessed 12-July-2023].

[11] Vscode. https://code.visualstudio.com. [Accessed 13-July-2023].

[12] C.A.C. Coello, G.T. Pulido, and M.S. Lechuga. Handling multiple objectives with particle swarm optimization. *IEEE transactions on evolutionary computation*, 8(3):256–279, 2004.

[13] Yann Collette and Patrick Siarry. *Multiobjective optimization: principles and case studies*. Decision Engineering. Springer, 2013.

[14] Gara Miranda Coromoto León and Carlos Segura. Metco: A parallel plugin-based framework for multi-objective optimization. *International Journal on Artificial Intelligence Tools*, 18(4):569–558, 2009.

[15] Daniel del Castillo de la Rosa. Prodef-algorithm: Interfaz para el modelado de meta-heurísticas. Technical report, Universidad de La Laguna, 2022.

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

[17] A.E Eiben. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer Nature, Berlin, Heidelberg, 2015.

[18] Andrés Calimero García Pérez. Prodef: meta-modelado de problemas de optimización combinatoria. Technical report, Universidad de La Laguna, 2020.

[19] Daniel González Expósito. Prodef-gui: Interfaz gráfica para el modelado de problemas. Technical report, Universidad de La Laguna, 2021.

[20] Yufeng Guo, Edward C. Keedwell, Godfrey A. Walters, and Soon-Thiam Khu. Hybridizing cellular automata principles and nsgaii for multi-objective design of urban water networks. In *Evolutionary Multi-Criterion Optimization*, volume 4403 of *Lecture Notes in Computer Science*, pages 546–559, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[21] Alejandro Lugo Fumero. Prodef: unificación e integración de módulos. Technical report, Universidad de La Laguna, 2023.

[22] Yeixon Reinaldo Morales Gonzalez. Prodef-solution: Interfaz para la representación y visualización de soluciones. Technical report, Universidad de La Laguna, 2022.

[23] Antonio J. Nebro. jMetal Repository. `https://github.com/jMetal/jMetal`, 2023.

[24] Miguel Angel Ordoñez Morales. Prodef: Diseño, implementación y experimentación con nuevos resolutores. Technical report, Universidad de La Laguna, 2022.

[25] Jun Sun, Choi-Hong Lai, and Xiao-Jun Wu. *Particle swarm optimisation: classical and quantum perspectives*. Chapman Hall/CRC numerical analysis and scientific computing. CRC Press, Boca Raton [Fla.], 2012.

[26] Ángel Tornero Hernández. Prodef-saas: Despliegue y puesta en marcha de un servicio para la resolución de problemas de optimización. Technical report, Universidad de La Laguna, 2022.

[27] Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary computation*, 8(2):173–195, 2000.