Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Prodef: Diseño, implementación y experimentación con nuevos resolutores

Prodef: Design, implementation and experimentation on new solvers

Miguel Angel Ordoñez Morales

La Laguna, 11 de Marzo de 2022

D. **Casiano Rodríguez León**, con N.I.F. 42.020.072-S, Catedrático de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

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 cotutora

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

Que la presente memoria titulada:

"*Prodef: Diseño, implementación y experimentación con nuevos resolutores*"

ha sido realizada bajo su dirección por D. **Miguel Angel Ordoñez Morales**, con N.I.F. 54.949.286-D.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 11 de Marzo de 2022

Agradecimientos

A los profesores que me guiaron a lo largo de la carrera y en especial mis tutores por su ayuda, paciencia y dedicación, compartiéndome sus conocimientos.

A mis padres por confiar en mi, por los valores y principios que me han inculcado.

A mi hermana por apoyarme y darme ánimo para siempre seguir adelante.

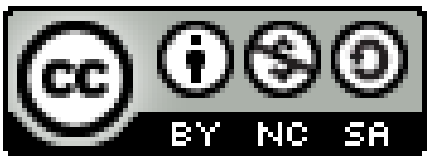
A mis abuelos por el cariño, alegrías y enseñanzas que me han dado.

A mis amigos que me ha regalado la vida por el apoyo diario.

A la Universidad de La Laguna (España) por enseñarme otra perspectiva de la carrera.

A la Universidad Católica Andrés Bello (Venezuela) por ser el punto de partida en el que comencé mi carrera universitaria.

Licencia



Resumen

A lo largo de la historia han surgido cambios que tienen su origen en el deseo de obtener un mayor beneficio frente a los costos de los procesos y recursos utilizados. Un ejemplo de ello fue la revolución industrial, que marcó el inicio de un nuevo sistema económico y supuso un gran avance tecnológico, aumentando la capacidad de producción, disminuyendo el tiempo de fabricación y simplificando tareas complejas en varias operaciones simples.

Esto no solo define épocas, sino que ocurre en nuestro día a día cuando se busca otra alternativa para "obtener más por menos", ya que el ser humano "es perezoso por naturaleza". Esto no es del todo malo si se tiene en cuenta que gracias a ello, se ha logrado optimizar las actividades para gastar la menor cantidad de energía posible. De esta necesidad surge la herramienta Prodef, la cual permite la resolución de problemas para los que conocemos su formulación pero no cómo resolverlos.

Prodef es una herramienta que provee una arquitectura extensible que posibilita la traducción desde una definición matemática del problema (pro-blem def-inition) hasta potencialmente cualquiera de los frameworks de optimización basados en heurísticas de computación evolutiva. Actualmente, Prodef cuenta con la integración con jMetal (Metaheuristic Algorithms in Java) como framework de optimización basado en heurísticas, utilizando el lenguaje de programación Java. A partir de jMetal, Prodef crea una abstracción para resolver los problemas de optimización siguiendo una serie de pasos.

Pero cambiar nuestra perspectiva al enfrentarnos a un mismo problema, puede marcar la diferencia entre resolverlo o no, entre eficiencia o ineficiencia e incluso en la obtención del valor óptimo. A partir de esta idea, se ha desarrollado este trabajo con la finalidad de añadir otra manera de resolver los problemas en Prodef. Para ello, se utiliza la herramienta METCO (Metaheuristics-based Extensible Tool Cooperative Optimization), que es otro framework de optimización basados en heurísticas. METCO en vez de utilizar Java, utiliza C++ y en consecuencia, se necesita la traducción desde el lenguaje usado para la definición matemática a C++, con el objetivo de crear una abstracción capaz de resolver cualquier problema de optimización combinatoria.

Sin embargo, una abstracción para los problemas de optimización combinatoria limita su capacidad de resolución, pero incrementa su alcance a distintas áreas. Es una herramienta con un gran potencial que aún puede seguir creciendo, ya que permite a personas sin conocimientos específicos de programación o pequeñas y medianas empresas mejorar procesos productivos.

Palabras clave: Optimización combinatoria, meta-heurísticas, algoritmos evolutivos, herramientas para optimización, problema de la mochila (Knapsack), problema del viajante de comercio (TSP).

Abstract

Throughout history, there have been changes in the desire to obtain greater profit and lower cost of processes and resources used. For example the industrial revolution, which started a new economic system and technological advance, increasing production capacity, decreasing manufacturing time and simplifying complex tasks into several simple operations.

This doesn't only define times, but occurs in our daily lives when we look for another alternative to "get more for less", since the human being "is lazy by nature". This isn't all bad, because we have been able to optimize the activities to spend as little energy as possible. Prodef was created for this reason, which allows the resolution of problems for which we know the formulation but not how to solve them.

Prodef is a tool that provides an extensible architecture that makes the translation from a mathematical definition of the problem (pro-blem def-inition) to any of the optimization frameworks based on evolutionary computation heuristics.Currently, Prodef is integrated with jMetal (Metaheuristic Algorithms in Java) as an optimization framework based on heuristics, using the Java programming language. Prodef creates an abstraction with jMetal to solve optimization problems following a series of steps.

But changing our perspective when studying the same problem makes the difference between solving it or not, between efficiency or inefficiency, and even obtaining the optimal value. From this idea, I have added another way to solve the problems in Prodef, using METCO (Metaheuristics-based Extensible Tool Cooperative Optimization) tool. METCO is another optimization framework based on heuristics using C++. Therefore, translating the language used for the mathematical definition into C++ is necessary to create an abstraction and solve any combinatorial optimization problem.

However, an abstraction for combinatorial optimization problems limits its solution, but increases its scope to different areas. It's a tool with great potential that can continue to grow, since it allows people without specific programming knowledge or small and medium-sized companies to improve production processes.

Keywords: Combinatorial optimization, meta-heuristics, evolutionary algorithms, optimization tools, Knapsack problem, traveling salesman problem (TSP).

Índice general

1. Introducción **1**

1.1. Antecedentes y estado actual del tema . . . 3

1.2. Objetivos . . . 4

1.3. Fases y desarrollo del proyecto . . . 5

2. Metco **7**

2.1. Instalación . . . 7

2.2. Definición de un problema . . . 8

2.2.1. Inicialización . . . 9

2.2.2. Evaluación . . . 10

2.2.3. Rangos de búsqueda . . . 10

2.2.4. Dirección de la optimización . . . 11

2.2.5. Clonación . . . 11

2.2.6. Adicional . . . 11

2.3. Ejecución de un problema . . . 11

2.4. Consideraciones . . . 13

2.4.1. Diseño modular . . . 13

2.4.2. Documentación . . . 13

2.4.3. CMake . . . 13

3. Prodef **15**

3.1. Instalación . . . 17

3.2. Ejecución . . . 18

3.3. Estructura común de un módulo . . . 19

3.3.1. Repositorio . . . 19

3.3.2. Dependencias . . . 20

3.3.3. TypeScript . . . 20

3.3.4. Empaquetador . . . 22

3.3.5. Permisos . . . 23
3.4. Elementos adicionales comunes de un módulo . . . 24
3.4.1. Pruebas . . . 24
3.4.2. Formatear código . . . 24
3.5. Publicación de un módulo . . . 25
3.5.1. Archivos incluidos . . . 25
3.5.2. Scripts . . . 26
3.5.3. Tipos . . . 26
3.5.4. Publicar . . . 26
3.6. Depuración . . . 26
3.6.1. Google Chrome . . . 27
3.6.2. Visual Studio Code . . . 27
4. Compilador C++ 29
4.1. prodef-compiler-cpp . . . 29
4.2. Dependencias . . . 30
4.3. Funciones de prodef-lang . . . 30
4.4. Compilador . . . 31
4.4.1. Constructor . . . 31
4.4.2. Valor absoluto . . . 33
4.4.3. Valor mínimo . . . 33
4.4.4. Potencia . . . 34
4.4.5. Límites máximos y mínimos . . . 34
4.4.6. Renombrar variables . . . 35
4.5. Ciclos . . . 35
4.5.1. Bucle for . . . 36
4.5.2. Extender el lenguaje prodef-lang . . . 36
4.5.3. Función wrapper . . . 37
4.5.4. Paradigma funcional . . . 38
4.6. Exportaciones . . . 39
4.7. Flujo de trabajo . . . 40
5. Resolutor de METCO 42
5.1. prodef-solver-metco . . . 42
5.2. Dependencias . . . 43
5.3. Plantilla . . . 43

5.3.1. ProdefProblemInstance.hpp.handlebars . . . 44
5.3.2. ProdefProblemInstance.cpp.handlebars . . . 46
5.4. Resolutor . . . 47
5.4.1. Constructor . . . 47
5.4.2. Funciones auxiliares . . . 47
5.4.3. Solve . . . 48
5.5. Variables de entorno . . . 49
5.6. Iniciar API . . . 50
5.7. Flujo de trabajo . . . 51
6. Experimentación 52
6.1. Módulos . . . 52
6.1.1. prodef-problem-formatter . . . 52
6.1.2. prodef-evaluator . . . 53
6.2. Configuración . . . 56
6.2.1. Algoritmo . . . 57
6.2.2. Ejecución . . . 57
6.2.3. Componentes del ordenador . . . 58
6.3. Resultados . . . 58
6.3.1. Nomenclatura . . . 58
6.3.2. Knapsack . . . 59
6.3.3. TSP . . . 66
6.4. Análisis . . . 73
7. Conclusiones y líneas futuras 75
7.1. Conclusiones . . . 75
7.2. Líneas futuras . . . 76
8. Summary and Conclusions 78
8.1. Summary . . . 78
8.2. Conclusion . . . 78
9. Presupuesto 79

Índice de Figuras

1.1. Estructura de módulos en Prodef 4
1.2. Cronograma de trabajo 6

2.1. Fichero de ejemplo para una instancia del problema Knapsack 10
2.2. Proceso para generar el Makefile 14

3.1. Arquitectura de Prodef 15
3.2. Depurar con Google Chrome 27
3.3. Depurar con Visual Studio Code 28

4.1. Problema antes de compilar 40
4.2. Problema compilado 41

5.1. Proceso de compilación en C/C++ 44
5.2. Funcionamiento de Prodef 49

6.1. Instancia 20-1 al inicio de las ejecuciones con 1600 evaluaciones 61
6.2. Instancia 20-1 al final de la ejecuciones con 80000 evaluaciones 61
6.3. Instancia 50-1 al inicio de las ejecuciones con 1600 evaluaciones 62
6.4. Instancia 50-1 al final de la ejecuciones con 80000 evaluaciones 62
6.5. Instancia 100-1 al inicio de las ejecuciones con 1600 evaluaciones 63
6.6. Instancia 100-1 al final de las ejecuciones con 80000 evaluaciones 63
6.7. Valor máximo de la instancia 100-1 durante 80000 evaluaciones 65
6.8. Valor mínimo de la instancia 100-1 durante 80000 evaluaciones 65
6.9. Media entre máximos y mínimos de la instancia 100-1 durante 80000 evaluaciones 66
6.10 Instancia ulysses16 al inicio de las ejecuciones con 1600 evaluaciones 68
6.11 Instancia ulysses16 al final de la ejecuciones con 80000 evaluaciones 69
6.12 Instancia att48 al inicio de las ejecuciones con 1600 evaluaciones 69
6.13 Instancia att48 al final de la ejecuciones con 80000 evaluaciones 70
6.14 Instancia rat-99 al inicio de las ejecuciones con 1600 evaluaciones 70

6.15. Instancia rat-99 al final de las ejecuciones con 80000 evaluaciones 71
6.16. Valor máximo de la instancia att-48 durante 80000 evaluaciones 72
6.17. Valor mínimo de la instancia att-48 durante 80000 evaluaciones 72
6.18. Media entre máximos y mínimos de la instancia att-48 durante 80000 evaluaciones 73

Índice de Tablas

6.1. Instancias del problema Knapsack 59
6.2. Resultados del problema Knapsack 59
6.3. Instancias del problema TSP 67
6.4. Resultados del problema TSP 67

9.1. Costes de infraestructura 79
9.2. Costes de desarrollo 79

Capítulo 1

Introducción

Los procesos productivos han evolucionado a un ritmo acelerado en las últimas décadas, ya que cada día estamos en un mundo más industrializado, donde cada detalle es fundamental. Por ello, se debe tener un control de los procesos para reducir los costes y aumentar la productividad, donde el campo de la optimización, una rama de las Matemáticas Aplicadas y de las Ciencias de la Computación se ha convertido en una herramienta muy utilizada para encontrar las *"mejores"* soluciones a los problemas planteados. Sin embargo, la calidad de estas soluciones dependerá de diversos factores como la técnica algorítmica aplicada, la eficiencia de la implementación de la técnica en cuestión, su parametrización y, por supuesto, del conocimiento específico del problema que tiene el usuario.

Los **problemas** de optimización pueden ser simples o más complejos dependiendo del número de objetivos, el espacio de la solución, el número de variables de decisión, las restricciones especificadas, etc. Algunos problemas de optimización que satisfacen determinadas propiedades son relativamente fáciles de resolver [14]. Sin embargo, muchos problemas de optimización son bastante difíciles de resolver; de hecho, muchos de ellos encajan en la familia de *problemas NP-hard* [1].

De la misma manera, los **algoritmos** de optimización pueden ser muy simples o complejos dependiendo de si son capaces de dar o soluciones aproximadas, ya sean menos o más eficientes o incluso si necesitan ser diseñados específicamente para un problema particular o no. Considerando lo anterior, a la hora de decidir qué técnica algorítmica aplicar hay que tener en cuenta las propiedades del problema a resolver, pero también qué resultados esperamos obtener: calidad de la solución, eficiencia del enfoque, flexibilidad y generalidad del método, etc.

Dado que se han propuesto muchos algoritmos de optimización, con diferentes propósitos y desde diferentes perspectivas, no existe un único criterio que se puede utilizar para clasificarlos. Podemos encontrar diferentes categorías de algoritmos en la literatura en función de las características relacionadas con su implementación interna, su esquema general de comportamiento, el campo de aplicación, complejidad, etc.

Los enfoques exactos generalmente se basan en una búsqueda enumerativa, exhaustiva o de fuerza bruta que arroja la solución óptima al problema. Sin embargo, en la mayoría de los problemas relacionados con instancias reales o grandes, el espacio de búsqueda puede ser demasiado grande o puede que no haya, incluso no existe una forma conveniente de enumerarlo. En tales situaciones, es común recurrir a algún tipo de *método aproximado*.

Estos enfoques generalmente no garantizan soluciones óptimas, pero algunos de ellos al menos aseguran un cierto nivel de calidad en sus soluciones, por ejemplo, los *algoritmos de aproximación*.

Las *heurísticas ad-hoc* son otro tipo de método aproximado que incorpora información sobre el problema en cuestión para decidir qué solución candidata debe probarse a continuación o cómo se puede producir la próxima candidata [18]. Dado que se basan en un conocimiento específico del problema, tales métodos ad-hoc tienen el inconveniente de depender del problema. Es decir, una amplia variedad de heurísticas puede diseñarse de manera específica y exitosa para optimizar un problema dado, pero desafortunadamente, no se pueden extrapolar directamente a otros problemas.

Una *meta-heurística* [7, 17] es un método aproximado para resolver una clase muy general de problemas. Combina funciones objetivos y heurísticas de una manera abstracta y, con suerte, eficiente, generalmente sin requerir una comprensión más profunda de su estructura. Las meta-heurísticas se han definido como estrategias maestras para guiar y modificar otras heurísticas para producir soluciones más allá de las normalmente identificado por heurísticas ad-hoc. En comparación con los métodos de búsqueda exactos, las meta-heurísticas no pueden garantizar una exploración sistemática de todo el espacio de la solución por lo que no pueden garantizar que se logren soluciones óptimas. En cambio, con el objetivo de aumentar la eficiencia, buscan examinar solo las partes donde las soluciones parecen *"buenas"* o *"prometedoras"*. Las meta-heurísticas pueden ser basadas en trayectoria o poblacionales en función de si mantienen una sola solución activa, o un conjunto de ellas, respectivamente. Por otro lado, las nuevas soluciones candidatas se construyen principalmente de dos formas: desde cero o como una evolución de las soluciones actuales.

Existe en la literatura una amplia variedad de meta-heurísticas [9]. Entre ellas, los algoritmos evolutivos [4] han demostrado ser capaces de obtener resultados muy competitivos para una amplia variedad de problemas. Estos algoritmos se basan en el concepto natural de evolución para gestionar poblaciones de individuos (soluciones) que mutan y se combinan entre sí para generar nuevas generaciones en las que los más aptos (soluciones de mayor calidad o más prometedoras) son quienes sobreviven y alcanzan la siguiente generación [2]. Sin embargo, la curva de aprendizaje en el ámbito de las meta-heurísticas así como la dificultad a la hora de utilizar la mayor parte de las librerías y frameworks existentes no ha propiciado la expansión de estas técnicas en sectores empresariales no vinculados con la investigación, ya que hay que tener en cuenta que para poder utilizar este tipo de herramientas, generalmente es necesario tener conocimientos sólidos sobre programación (el lenguaje concreto dependerá de la herramienta en cuestión), sobre la propia formulación del problema (variables, objetivos y restricciones), y, finalmente, sobre la técnica algorítmica a utilizar (parametrización general y configuración interna). Por ello, aunque en el sector empresarial se dispone de mucha información sobre el problema que se necesita resolver, no necesariamente se va a disponer de personal con conocimientos de programación y/o formación específica para utilizar o configurar meta-heurísticas. Además de este problema, nos encontramos con la cohesión o vínculo que puede tener la definición de un problema con el tipo de técnica a aplicar, lo cual depende de factores tales como:

- La representación del problema influye en el tamaño y otras características del espacio de soluciones.

- Cada problema es distinto y no existe una regla que se pueda aplicar de manera genérica.
- La definición de los movimientos óptimos de una solución a otra, depende de muchos factores, que dificultan que se pueda establecer una regla general.

Por lo tanto, si el usuario desea resolver el problema mediante la aplicación de técnicas diferentes, puede resultar una tarea bastante complicada, ya que tendrá que definir el problema de distintas formas o bien implementar funciones adaptadas a cada una de las estrategias a utilizar. Como ya se ha mencionado, esto es debido al fuerte vínculo entre la definición del problema y la técnica utilizada. Esto hace necesario que el usuario entienda el diseño interno de la herramienta que vaya a utilizar, pues solo así podrá seleccionar el tipo de técnicas, operadores y resto de configuración adecuada para el problema que necesita resolver.

Teniendo lo anterior en cuenta, surge la necesidad de diseñar e implementar nuevas herramientas o interfaces que ayuden a mejorar la aplicabilidad de estas técnicas de optimización avanzada a sectores externos al de la investigación formal. El objetivo es proporcionar implementaciones eficientes y flexibles de técnicas algorítmicas, que permita a los usuarios centrarse únicamente en el propio problema, evitándoles tener que introducirse en áreas distintas a las de su dominio. Por lo tanto, es interesante disponer de herramientas de más alto nivel (con mecanismos de mayor abstracción) para la resolución de problemas mediante la aplicación de este tipo de técnicas. De hecho, el objetivo ideal sería permitir que, mediante un modelado sencillo e intuitivo (sin usar lenguajes de programación concretos) de un problema de optimización, se obtuviera automáticamente una implementación de una meta-heurística (un algoritmo evolutivo simple, por ejemplo) para resolver dicho problema. Con este objetivo en mente, y gracias a un Trabajo de Fin de Grado desarrollado durante el pasado curso académico [6], se ha comenzado a trabajar en una herramienta denominada *Prodef* (acrónimo para "*Problem Definition*").

1.1. Antecedentes y estado actual del tema

Prodef [6] es una herramienta que proporciona una capa de abstracción entre el modelo del problema y las técnicas de resolución, con el fin de facilitar al usuario sin conocimientos de programación ni de métodos de optimización bio-inspirados, la tarea de modelar y resolver problemas de optimización combinatoria, ya que permite la definición de problemas de optimización combinatoria a un nivel de descripción abstracto y su posterior resolución usando librerías y frameworks externos. Además, *Prodef* fue diseñado para ser una herramienta fácilmente extensible, ya que cuenta con un bloque de módulos que conforman su base como son `prodef-storage`, `prodef-compiler` y `prodef-solver`, a partir de los cuales se puede implementar módulos específicos dependiendo del resolutor incorporado como lo son `prodef-storage-redis`, `prodef-compiler-java` y `prodef-solver-jmetal`, conformando una estructura en módulos como se muestran en la Figura 1.1.

La herramienta gira en torno a un meta-modelo que permite capturar, en términos abstractos, las características esenciales de un problema de optimización. El meta-modelo puede servir como forma de almacenamiento o modelado genérico para problemas de optimización, a partir de los cuales se pueden generar problemas de optimización reales,

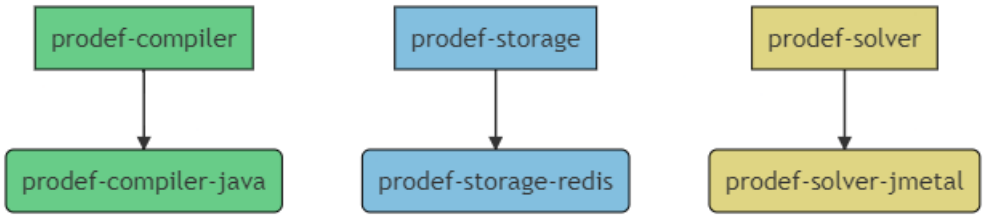


Figura 1.1: Estructura de módulos en Prodef

ejecutables. A partir de este modelo o especificación genérica de un problema se pretende obtener una traducción directa a al menos algún framework o herramienta de optimización basada en meta-heurísticas bio-inspiradas.

En la actualidad *Prodef* ofrece conexión directa con uno de los frameworks más extendidos en el ámbito de la optimización con meta-heurísticas: *jMetal* [3, 13]. Este framework está escrito en Java y requiere que el usuario codifique en ese mismo lenguaje. Sin embargo, *Prodef* proporciona un módulo que es capaz de traducir el modelado del problema a Java (`prodef-compiler-java`), y a su vez, a la estructura de clases de *jMetal* que hará posible la ejecución de un algoritmo evolutivo "simple" para resolver el problema definido. En este contexto, `prodef-solver` es el módulo encargado de definir la API REST y de proporcionar la base para todos los resolutores que se implementen en *Prodef*. Este módulo implementa un servidor web con los *endpoints* de la API REST para los resolutores: el servidor se encarga de gestionar el ciclo de vida de las ejecuciones solicitadas y de llamar al resolutor correspondiente cuando sea requerido. Por su parte, `prodef-solver-jmetal` es el módulo que implementa un resolutor basado en el framework *jMetal*. Los resolutores heredan de la clase `Solver` definida en `prodef-solver`. La estrategia que sigue este resolutor consiste en generar un archivo de código fuente Java a partir del problema compilado. El archivo se genera mediante una plantilla de *Handlebars* y, posteriormente, se ejecuta la clase `Main` desarrollada, pasándole como argumentos el nombre de la clase generada con la plantilla (`ProdefProblemInstance.java`) y el tipo de problema.

Cuando finaliza la ejecución se obtiene como resultado (de la salida estándar del proceso de Java) un documento JSON que contiene las soluciones encontradas al problema. Este archivo es procesado por el resolutor para crear el objeto `Solution` que será devuelto por el método `solve` y que constituirá la solución final de la ejecución del problema.

1.2. Objetivos

Teniendo en cuenta el diseño y la estructura interna de *Prodef*, para incluir nuevos resolutores en la herramienta, habrá que implementar los correspondientes módulos de compilación y de resolución asociados al nuevo framework. En este trabajo se plantea la extensión de *Prodef* para incluir como resolutor adicional el framework *METCO* (*Metaheuristics-based Extensible Tool Cooperative Optimization*). *METCO* [11] es un framework para optimización combinatoria desarrollado en C++ por investigadores de la Universidad de La Laguna, el cual será explicado con mayor profundidad en el Capítulo 2.

Con el fin de ofrecer desde *Prodef* la posibilidad de utilizar *METCO* como resolutor de problemas, será necesario abordar los objetivos que se relacionan a continuación:

1. **Análisis de Prodef y del resolutor para jMetal (ACT1):** El objetivo de esta tarea es abordar el estudio de la herramienta *Prodef*, comprendiendo su estructura modular y su funcionamiento interno. Además, habrá que analizar con detenimiento la implementación del actual resolutor (*jMetal*) ofrecido por la herramienta, y que servirá como punto de partida para diseñar el nuevo resolutor.

2. **Análisis de METCO e implementación de problemas (ACT2):** Analizar *METCO* como framework de optimización con meta-heurísticas. Con el fin de entender mejor el funcionamiento de este framework se implementarán tres problemas de optimización bastante comunes en el ámbito académico: el problema de la mochila (*Knapsack*), el problema del viajante de comercio (*TSP*) y un problema de planificación.

3. **Diseño e implementación del compilador de C++ (ACT3):** Diseñar e implementar el módulo que permita traducir las especificaciones de los problemas al lenguaje de programación C++, pues es el lenguaje en el que está implementado el framework *METCO*.

4. **Diseño e implementación del resolutor para METCO (ACT4):** Diseñar e implementar el módulo que dé soporte para resolver, mediante la interfaz de *Prodef*, los problemas a través de las técnicas algorítmicas ofrecidas por *METCO*.

5. **Documentación (ACT5):** Elaboración del material de soporte para el uso de la herramienta así como la memoria del Trabajo de Fin de Grado y demás documentación requerida durante la realización del mismo.

A lo largo del desarrollo de este trabajo se valoró la importancia de la calidad de las soluciones fruto del resolutor para *METCO*, mediante la interfaz de *Prodef*, en comparación con las soluciones calculadas directamente con *METCO* sin utilizar *Prodef* y por lo cual se propuso el siguiente objetivo:

Experimentación y análisis de resultados: Proponer y definir un formato estándar de ejecución y recogida de resultados para resolutores basados en algoritmos evolutivos, de manera que se pueda especificar las tareas de múltiples ejecuciones con diferentes algoritmos, instancias y variantes. Este estándar facilita que puedan ser utilizados por programas de análisis de datos y visualización para obtener conclusiones sobre la calidad y la eficiencia de las soluciones y de los algoritmos.

Inicialmente se tenía como objetivo (ACT2) implementar tres problemas de optimización combinatoria, entre ellos *Knapsack*, *TSP* y un problema de planificación, pero se optó por solo implementar los dos primeros para darle más importancia a este nuevo objetivo, ya que es de vital importancia para mejorar la calidad del resolutor de *METCO* y obtener mejores soluciones.

1.3. Fases y desarrollo del proyecto

Atendiendo al conjunto de objetivos principales que conforman este Trabajo de Fin de Grado, la temporalización estimada de los mismos se recoge en el plan de trabajo de la Figura 1.2. En la tabla, se representa cada uno de los objetivos en la primera

columna y cada semana en la primera fila, de esta manera se puede ver la estimación de tiempo asignada a cada una de las tareas, salvo la de documentación, ya que se hace de manera paralela a las demás tareas. Sin embargo, la planificación de este cronograma se extendió para la finalización de todas las tareas y el anexo del nuevo objetivo explicado anteriormente.

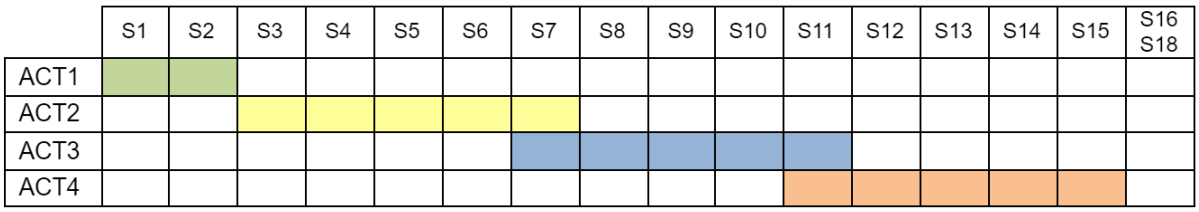


Figura 1.2: Cronograma de trabajo

Capítulo 2

Metco

METCO [11] (*Metaheuristics-based Extensible Tool Cooperative Optimization*) es un framework para optimización combinatoria, que proporciona implementaciones de algunos de los algoritmos evolutivos y otras meta-heurísticas más conocidas, incluyendo entre ellas algunos métodos para optimización multi-objetivo (`NSGAII`, `SPEA2`, `IBEA`, entre otros). Está desarrollado en `C++` y se basa en plugins para la definición de distintos tipos de algoritmos, operadores, problemas, etc.

Este framework permite al usuario establecer características específicas del problema que necesita solucionar (representación de la solución, número y tipo de objetivos, operadores genéticos, etc.), pudiendo especificar una amplia gama de opciones de configuración para adaptar el comportamiento del software a varios modelos diferentes. De hecho, una de las principales ventajas de *METCO* con respecto a otras herramientas es la posibilidad de usar hiper-heurísticas para proporcionar más tiempo de cómputo a aquellas configuraciones de algoritmos que van presentando un mejor comportamiento al resolver un problema dado. Para ello, la herramienta se basa en modelos paralelos basados en islas [16].

El repositorio está alojado actualmente en la organización de GitHub `PAL-ULL`, pero para tener una mayor comodidad de trabajo, se creó un fork en la organización `ULL-prodef`, con la finalidad de desarrollar el trabajo en una rama y posteriormente hacer una *pull request*.

2.1. Instalación

Se debe seguir un proceso para instalar y configurar *METCO*, que consiste en los siguientes pasos:

1. Clonar el repositorio de *METCO* que se encuentra en la nueva organización.

```
~$ git clone git@github.com:ULL-prodef/software-metco.git
```

2. Instalar las siguientes dependencias necesarias:
   - gcc/g++ compiler
   - Autotools
   - Bison
   - Flex

- MPICH
- GNU GSL

En el caso de un entorno Linux (Debian) se instalan de la siguiente manera:

```
~$ sudo apt-get install build-essential
~$ sudo apt-get install autotools-dev automake
~$ sudo apt-get install bison
~$ sudo apt-get install flex
~$ sudo apt-get install mpich mpich-doc
~$ sudo apt-get install libgsl0-dev
```

3. Ir a la ruta donde se encuentran los scripts necesarios para la instalación de la herramienta y ejecutarlos.

```
~$ cd software-metco/oplink/algorithms/team/src/skeleton/main
~$/software-metco/oplink/algorithms/team/src/skeleton/main$ ./gen.sh
~$/software-metco/oplink/algorithms/team/src/skeleton/main$ ./configure
```

4. Ejecutar la compilación de todos los algoritmos con el comando make.

```
~$/software-metco/oplink/algorithms/team/src/skeleton/main$ make
```

2.2. Definición de un problema

Hay que tomar varios aspectos en cuenta para definir un problema. Se utilizará como ejemplo el problema de la mochila o por su nombre en ingles *Knapsack*. Para ello, se definen los siguientes pasos:

1. Crear una carpeta en la "*ruta específica*" de los problemas con el "*nombre del problema*".

```
~$ cd software-metco/oplink/algorithms/team/src/plugins/problems
~/software-metco/oplink/algorithms/team/src/plugins/problems$ mkdir Knapsack
```

2. Añadir en la nueva carpeta un archivo de encabezado (.h) y archivo fuente (.cpp) con el mismo nombre de la carpeta.

```
~/software-metco/oplink/algorithms/team/src/plugins/problems$ cd Knapsack
~/software-metco/oplink/algorithms/team/src/plugins/problems/Knapsack$ touch
    Knapsack.h
~/software-metco/oplink/algorithms/team/src/plugins/problems/Knapsack$ touch
    Knapsack.cpp
```

3. Incluir la librería `Individual.h` en el archivo de encabezado `Knapsack.h`.

```
#include "Individual.h"
```

La clase `Individual` representa un problema genérico; similar a una clase abstracta que tiene que ser heredada por todos los problemas para sobrescribir ciertos métodos, que se explican más adelante.

4. Definir la nueva clase en el encabezado `Knapsack.h` y hacer que herede de la clase `Individual`.

```
#ifndef __KNAPSACK_H__
#define __KNAPSACK_H__

#include "Individual.h"

class Knapsack : public Individual {
    public:
        Knapsack() {}
        virtual ~Knapsack() {}
}

#endif
```

5. Especificar los siguientes comportamientos del problema (los cuales se detallan en las siguientes subsecciones para el ejemplo concreto del `Knapsack`):

    - Inicialización
    - Evaluación
    - Rangos de búsqueda
    - Dirección de la optimización
    - Clonación

2.2.1. Inicialización

Se sutiliza el método `init`, el cual se encarga de inicializar el problema con los parámetros dados a través de la línea de comandos o un fichero de configuración. Dentro de este método, se debe definir el número de **objetivos** y **variables** del problema. Para realizar tal especificación se utilizan los siguientes métodos:

- `setNumberOfVar`: Define el número de variables del problema. Es una buena práctica tener dos vectores que definan el valor mínimo y máximo de cada variable si no tienen valores constantes, por ejemplo:

```
static vector<double> minVar;
static vector<double> maxVar;
```

- `setNumberOfObj`: Define el número de objetivos del problema. En la optimización mono objetivo se asigna el número 1, ya que solo hay una función objetivo.

Teniendo esto en cuenta el método `init` queda tal y como se muestra a continuación:

```
bool Knapsack::init(const vector<string>& params) {
    // Check for the problem parameters
    if (params.size() != ARGS) {
        cout << "Error Knapsack: Incorrect parameters " << endl;
        cout << "<instance> <type_crossover>" << endl;
        return false;
    }

    string filename = params[0];
    setNumberOfObj(1);
```

```
5         // Número de objetos
80        // Peso máximo de la mochila
15  33    // Peso y beneficio del primer objeto
20  24    // Peso y beneficio del segundo objeto
17  36    // Peso y beneficio del tercer objeto
8   37    // Peso y beneficio del cuarto objeto
31  12    // Peso y beneficio del quinto objeto
```

Figura 2.1: Fichero de ejemplo para una instancia del problema Knapsack

```
    type_crossover = atoi(params[1].c_str());

    // Reads a problem from file
    readFile(filename);
    ratios();
    return true;
}
```

Se puede ver que la lectura de los parámetros se hace desde un método auxiliar, que se encarga de definir el número de variables y no se utiliza los vectores de valores mínimos y máximos, ya que las variables tienen valores constantes, ya sea 0 (fuera de la mochila) o 1 (dentro de la mochila). Para este problema el fichero de configuración debe tener el formato del fichero de la Figura 2.1.

2.2.2. Evaluación

Se utiliza el método `evaluate`, que contiene la función objetivo del problema y para cada objetivo se debe llamar a la función `setObj(dimension, value)` con la actualización del valor del objetivo.

```
void Knapsack::evaluate(void) {
    float objective = 0;
    for (int j = 0; j < nItems; j++)
        objective += profits[j] * getVar(j);
    setObj(0, objective);
}
```

2.2.3. Rangos de búsqueda

Son dos métodos que definen el valor máximo y mínimo de cada variable.

```
double inline getMaximum(const int i) const { return 1; }
double inline getMinimum(const int i) const { return 0; }
```

En este ejemplo se retorna siempre un valor constante, pero en caso que fueran variables, se utiliza un índice para devolver el valor correspondiente.

```
double getMaximum(const int i) const { return maxVar[i]; };
double getMinimum(const int i) const { return minVar[i]; };
```

2.2.4. Dirección de la optimización

Se debe definir en el método `getOptDirection` con las variables `MINIMIZE` o `MAXIMIZE` si el problema es de minimización o maximización respectivamente.

```
unsigned int inline getOptDirection(const int i) const { return MAXIMIZE; }
```

En el caso del *Knapsack* es un problema de maximización, ya que se quiere buscar el "mayor" beneficio posible con los objetos dentro de la mochila.

2.2.5. Clonación

Se define el método `clone`, el cual retorna una copia "vacía" del objeto actual.

```
Individual* Knapsack::clone(void) const {
    return new Knapsack();
}
```

Este método no hace una copia completa de las variables de la instancia, sino que crea una instancia vacía, esto es, crea un individuo con la misma estructura en cuanto a variables y tipo de soluciones pero sin los valores concretos de la solución original. Posteriormente en el algoritmo, durante la ejecución de la instancia se hace la copia de los valores correspondientes.

2.2.6. Adicional

Hay métodos auxiliares que se puedan necesitar, pero que no son obligatorios de sobrescribir. En el caso de este problema algunos métodos extra son:

- `checkCapacity`: Comprueba que la capacidad de la mochila no se haya superado.
- `repair`: Restablece la factibilidad del problema.
- `restart`: Generación aleatoria de una solución.

2.3. Ejecución de un problema

Para ejecutar un problema de manera secuencial, es decir sin paralelismo y sin búsqueda local se hace a través de la línea de comandos, especificando algunos parámetros como:

- `outputPath`: ruta del directorio de la ejecución.
- `pluginPath`: ruta del plugin que se utiliza para el formato de la salida.
- `outputPrinterModule`: plugin para el formato de la salida, por ejemplo `PlainText` o `JsonPrinter`.
- `algorithm`: nombre del algoritmo, por ejemplo, `MonoGA`.

- `problem`: nombre del problema, por ejemplo: `Knapsack`.
- `critStop`: criterio para finalizar las evaluaciones del algoritmo, por ejemplo, por número de evaluaciones o por tiempo.
- `critStopValue`: valor numérico del criterio de parada utilizado.
- `printPeriod`: indica cada cuánto se muestra el estado del algoritmo.
- `useExternalArchive`: indica si se utiliza un archivo externo (almacenamiento de soluciones "prometedoras") para la ejecución.
- `algorithmParameters`: conjunto de parámetros internos del algoritmo.
- `ProblemParameters`: conjunto de parámetros específicos del problema.
- `LocalSearch`: indica si se usa búsqueda local.

El comando sigue el siguiente orden de parámetros:

```
~$ ./metcoSeq outputPath pluginPath outputPrinterModule outputFile algoritmhm problem
    critStop critStopValue printPeriod useExternalArchive [algorithmParameters] ! [
    ProblemParameters] $ LocalSearch ;
```

Es importante mantener el orden de los siguientes símbolos especiales, pues cada uno de ellos tiene un significado concreto:

- `!`: indica el inicio de los parámetros de un problema.
- `$`: indica el inicio de la búsqueda local.
- `;`: indica el final del comando.

Sustituyendo los parámetros por valores de ejemplo quedaría el siguiente comando para el problema `Knapsack`. Cabe mencionar que se utiliza la ruta de ejemplo `/home/<username>/`

```
~$ ./metcoSeq /home/<username>/solution . JsonPrinter execution.json MonoGA Knapsack
    EVALUATIONS 50000 1000 0 80 0.1 0.9 2 ! /home/<username>/problem/input.text 0 $ NoOp
    ;
```

Para el ejemplo `Knapsack` se utiliza el fichero `input.text` como entrada de los parámetros, el cual debe tener el formato de problema antes mostrado.

Finalmente, es importante tener en cuenta que para ejecutar este comando es necesario estar ubicado en la ruta donde específicamente se encuentre el ejecutable `metcoSeq`. Esta ruta se puede ver a continuación:

```
~$ cd software-metco/oplink/algorithms/team/src/skeleton/main
~$/software-metco/oplink/algorithms/team/src/skeleton/main$ ./metcoSeq [Resto del comando
    ]
```

2.4. Consideraciones

A la hora de trabajar con *METCO* se tomaron algunas decisiones importantes. Estas decisiones así como la motivación de las mismas se describen brevemente en los siguientes apartados.

2.4.1. Diseño modular

METCO está diseñado como una herramienta modular en la que se cargan de manera dinámica distintas partes del programa como si fuesen plugins: se utilizan librerías dinámicas con extensión `.so`. Un problema en *METCO* se representa como un plugin y esto se refleja en la existencia de ficheros con extensión `.library.cpp` como, por ejemplo, `Knapsack.library.cpp`, cuyo contenido es el siguiente:

```
#include "KnapsackWithViolationDegree.h"

extern "C" {
    Individual *maker(){
    return new Knapsack();
  }
}
```

Esto permite crear una instancia del problema sabiendo su nombre como si se tratase del patrón de diseño *Factory* [5], pero sin la necesidad de tener una clase que administre todas las instancias de problemas creadas, sino que carga de manera dinámica el archivo `Knapsack.so`.

Para seguir con la misma línea de trabajo con el diseño y no afectar la estructura interna de *METCO*, las nuevas funcionalidades se crearon las siguientes librerías externas:

- `prodef-common`: contiene la integración de *METCO* con otra herramienta utilizada en este trabajo llamada *Prodef*, la cual será explicada en el Capítulo 3.
- `metco-core`: implementa la capacidad para tratar problemas que pueden violar momentáneamente sus restricciones con el objetivo de conseguir soluciones más variadas en el espacio.

2.4.2. Documentación

Para que la curva de aprendizaje sea más rápida y no se comentan los mismos errores, se ha desarrollado una documentación de *METCO* en el siguiente enlace, donde se abarca el diseño de clases que contiene, funcionamiento de las librerías, proceso de compilación y ejecución, entre otros detalles para tener en cuenta para todo programador que comienza un desarrollo en *METCO*.

2.4.3. CMake

Las nuevas librerías utilizan *CMake* [12] para su compilación y se añaden durante la compilación de toda la aplicación *METCO* dinámicamente utilizando un fichero `Make` [8]

que ya estaba presente en el proyecto, pero ¿Por qué no modificar directamente este fichero?

METCO contiene una estructura de `Makefiles`, uno por módulo y otro general donde se utilizan todos esos `Makefiles` particulares para crear este último a través de un proceso (véase la Figura 2.2), utilizando los ficheros `configure.ac` y `Makefile.am`. Los distintos niveles de direccionamiento indirecto resuelven la complejidad de mantener ficheros como `configure`, `Makefile.in` y el `Makefile` final, pero añaden el problema de demasiadas capas de direccionamiento indirecto.

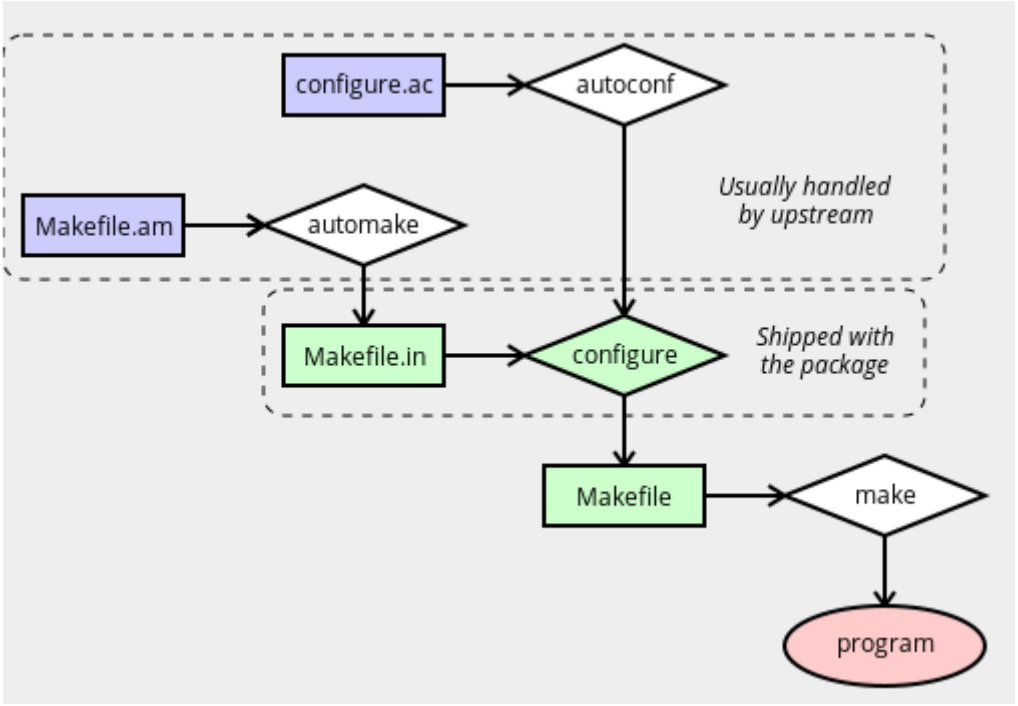


Figura 2.2: Proceso para generar el Makefile

Cmake utiliza ficheros llamados `CMakeLists.txt` que describen el origen y destino de los ficheros y a diferencia de un `Makefile` complejo, un `CMakeLists.txt` utiliza un lenguaje de más alto nivel similar a lenguajes de programación modernos, por lo que son más fácil de escribir y sobre todo mantener.

Capítulo 3

Prodef

Prodef [6] es una herramienta que proporciona una capa de abstracción entre el problema y las técnicas de resolución, como ya se explicó en el Capítulo 1. Por lo tanto, permite modelar problemas de optimización combinatoria, sin la necesidad de tener conocimientos específicos en la materia, y resolverlos utilizando algoritmos bio-inspirados pero mediante otras herramientas como *jMetal* o *METCO*.

Con la idea de ser un proyecto extensible, la herramienta está conformada por distintos módulos, agrupados en: los módulos que forman parte de su núcleo (véase la Figura 3.1) y los módulos que expanden sus capacidades y funcionalidades (véase la Figura 1.1).

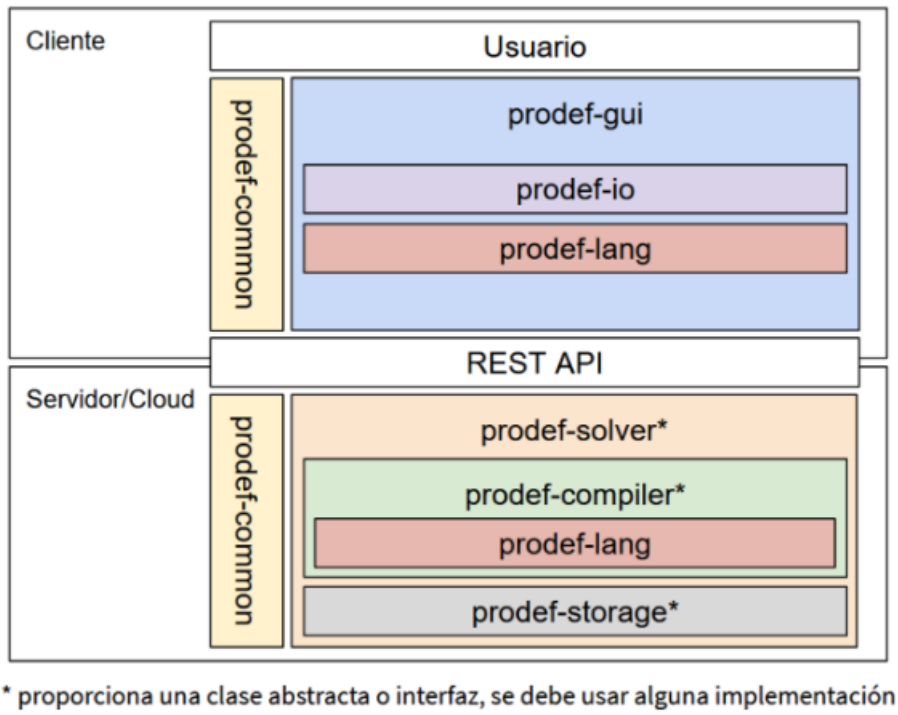


Figura 3.1: Arquitectura de Prodef

Los módulos que forman parte de su base son fijos, por ejemplo un compilador, ya que define la base a partir de la cual se pueden crear nuevos módulos que añadan nuevas funcionalidades, por ejemplo un compilador en C++. Entre los módulos del núcleo se encuentran:

- `prodef-common`: Proporciona las interfaces y tipo de datos comunes para que el resto de módulos puedan comunicarse entre sí sin problema. Además, estos datos comúnes forman la definición del meta-modelo para los problemas.
- `prodef-gui`: Proporciona una interfaz gráfica, donde el usuario puede diseñar el problema, utilizando lenguajes de programación visuales con la librería Blockly.
- `prodef-io`: Contiene funciones para deserializar y validar problemas y soluciones en formato JSON, a partir de los cuales se obtiene el problema definido con las interfaces comunes del módulo `prodef-common`.
- `prodef-lang`: Define el Lenguaje de Dominio Específico (DSL) utilizado por el usuario para definir el problema, siendo una mezcla entre un lenguaje matemático y orientado a objetos.
- `prodef-compiler`: Contiene la implementación común de todos los compiladores. Además, tiene una implementación específica que facilita en gran medida la implementación de compiladores basados en lenguajes cuya sintaxis sea similar a `C`.
- `prodef-solver`: Define la API-REST y proporciona la base para definir nuevos resolutores. Este servidor se encarga de gestionar el ciclo de vida de las ejecuciones y de llamar al resolutor cuando sea necesario.
- `prodef-storage`: Es un *driver* que define la base para cualquier otro *driver* para el almacenamiento no persistente.

A partir de los módulos del núcleo se definieron los siguientes módulos:

- `prodef-compiler-java`: Implementa un compilador específico de *ProdefLang* al lenguaje de programación Java. Utiliza la implementación de compiladores basados en el lenguaje `C`.
- `prodef-solver-jmetal`: Es una implementación de un resolutor para *Prodef* basado en el framework *jMetal*. Consiste en generar un archivo en java a partir del problema compilado, utilizando la plantilla Handlebars.
- `prodef-storage-redis`: Es un *driver* de almacenamiento persistente, que utiliza la base de datos Redis, usando la librería Tedis.

Prodef fue inicialmente desarrollada por Andrés Calimero García en su Trabajo de Fin de Grado [6] y los repositorios relacionados están alojados en la organización `PAL-ULL`. Sin embargo, con la idea de seguir creciendo el proyecto, se creó una organización exclusiva para *Prodef* llamada `ULL-prodef`, donde se migró todos los repositorios y se configuraron como meta-repositorios para facilitar su instalación.

3.1. Instalación

A lo largo de esta sección y la siguiente se explica cómo instalar y ejecutar *Prodef* con el problema de la mochila, utilizando la línea de comandos.

Antes de comenzar, se debe tener instalado las siguientes herramientas:

- Clonar repositorios con SSH.
- Node.
- Java.

Una vez instaladas, se realizan los siguientes pasos:

1. Clonar el repositorio.

```
~$ git clone git@github.com:ULL-prodef/prodef.git
```

2. Instalar las dependencias.

```
~$ cd prodef
~/prodef$ npm i
```

3. Descargar los subrepositorios utilizando *meta*.

```
~/prodef$ npm run update
```

   Meta es una herramienta para administrar varios sistemas o librerías, que permite clonar una estructura de muchos proyectos independientemente del repositorio en que se encuentren y facilita construir todos los repositorios a la vez con *npm*. Esta herramienta promueve las arquitecturas de muchos repositorios, que beneficia la implementación y la reutilización del código.

4. Crear el fichero `.npmrc`.

```
//npm.pkg.github.com/:_authToken=TOKEN
@pal-ull:registry=https://npm.pkg.github.com/
@ull-prodef:registry=https://npm.pkg.github.com/
```

   Este fichero se debe crear en la raíz de cada repositorio, remplazando el valor `TOKEN` con un token personal de GitHub adecuado.

5. Instalar, compilar y enlazar los subrepositorios haciendo uso de *meta*.

```
~/prodef$ npm run install:all
~/prodef$ npm run build:all
~/prodef$ npm run link:all
```

   Enlazar los subrepositorios con `npm link` permite trabajar en una dependencia y que sus cambios sean reflejados donde esté instalada, sin necesidad de publicar la dependencia como un paquete. Por ejemplo, el módulo o compilador específico `prodef-compiler-java` depende del módulo o compilador base `prodef-compiler`. Con los enlaces simbólicos, cualquier cambio en la dependencia se verá reflejado en el módulo `prodef-compiler-java`, sin necesidad de volver a instalarla.

   Nota: Al ejecutar el último (`npm run link:all`) puede llegar a fallar con el siguiente error:

```
~/prodef/node_modules/meta-npm/bin/meta-npm-link:39
    metaPackageJson.dependencies[childPackageJson.name] = null;
                                                        ^

TypeError: Cannot set property '@ull-prodef/prodef-common' of undefined
```

Esto se debe a que el archivo `package.json` contiene una etiqueta llamada "*dependencies*" vacía y al instalar el meta-repositorio puede llegar a eliminarse. Si este fuera el caso, solo se tiene que agregar esa etiqueta al `package.json`.

```
{
  "name": "@ull-prodef/prodef",
  "version": "1.0.0",
  "description": "Meta repository of prodef",
    .
    .
  "dependencies": {}
    .
    .
}
```

3.2. Ejecución

Para poder resolver el problema de la mochila (`Knapsack`), se deben seguir los siguientes pasos:

1. Ejecutar el servidor.

```
~/prodef$ cd prodef-solver-jmetal
~/prodef/prodef-solver-jmetal$ npm start
```

2. Realizar una petición POST con la definición del problema.

   Se puede utilizar la herramienta curl para ejecutar el problema, que se encuentra definido en este enlace. Otra alternativa es utilizar la herramienta Postman para ejecutar el problema anterior.

   Esta petición retorna un identificador y una clave para obtener la respuesta, tal que:

```
{
  "executionId": "39ac8cd0-2b36-11eb-9c47-5b57c8e2758a",
  "secretKey": "82f5ccaa-ff9a-498c-a402-14f0f4968cf3"
}
```

3. Hacer una petición GET con el identificador y la clave obtenidos en el paso anterior.

```
curl http://localhost:8046/v1/executions/39ac8cd0-2b36-11eb-9c47-5b57c8e2758a \
  -H 'Secret-Key: 82f5ccaa-ff9a-498c-a402-14f0f4968cf3'
```

   Al ejecutar la petición. se retorna la solución del problema en formato JSON.

3.3. Estructura común de un módulo

Cada proyecto en *Prodef* tiene una estructura similar con el objetivo de mantener la consistencia y que sea más fácil a la hora de extenderlo. Dicha estructura común será explicada a lo largo de esta sección.

Por ejemplo, para un módulo con el nombre `prodef-compiler-cpp` la estructura sería la siguiente:

```
prodef-compiler-cpp
├── dist
├── node_modules
├── src
│   └── index.ts
├── .gitignore
├── .npmrc
├── package-lock.json
├── package.json
├── README.md
├── rollup.config.js
└── tsconfig.json
```

3.3.1. Repositorio

Lo primero es configurar el repositorio de trabajo, el cual se crea en la organización Prodef, con el respectivo prefijo `prodef` y posteriormente se indica el nombre específico. Esto es, si se crea un nuevo módulo sobre un compilador en C++, su nombre sería `prodef-compiler-cpp`. Donde la palabra `compiler` indica que utiliza la base del módulo `prodef-compiler` y la palabra *cpp* indica que es una implementación en el lenguaje C++.

Para comenzar a trabajar con el repositorio se clona de la siguiente manera:

```
~$ git clone git@github.com:ULL-prodef/prodef-compiler-cpp.git
```

Una vez clonado, se accede a la carpeta y se crea un archivo `README.md` inicial.

```
~$ cd prodef-compiler-cpp
~/prodef-compiler-cpp$ touch README.md
```

Como es un repositorio de git y esta bajo control de versiones, se añade el fichero `README`.

```
~/prodef-compiler-cpp$ git add README.md
~/prodef-compiler-cpp$ git commit -m "chore: add README file"
~/prodef-compiler-cpp$ git push origin main
```

Además, se crea el fichero `.gitignore`, que contiene los ficheros o directorios que no se incluyen en el control del versiones. Para saber su contenido, se parte de la plantilla generada en gitignore.io con las etiquetas `node`, `windows`, `linux` y `visual studio code`.

```
~/prodef-compiler-cpp$ git add .gitignore
~/prodef-compiler-cpp$ git commit -m "chore: add gitignore file"
~/prodef-compiler-cpp$ git push origin main
```

```
prodef-compiler-cpp
├── .gitignore
└── README.md
```

3.3.2. Dependencias

Se deben configurar las dependencias y para ello se utiliza el fichero `package.json`, donde se encuentra la configuración de cada una de las herramientas que se utilizará en el proyecto. Para generarlo se hace lo siguiente:

```
~/prodef-compiler-cpp$ npm init -y
```

Para definir una forma estándar para facilitar las confirmaciones, se utiliza la herramienta commitizen.

```
~/prodef-compiler-cpp$ npm i commitizen --save-dev
```

Esto modifica el fichero `package.json` añadiendo un apartado de `devDependencies`, donde se puede ver la herramienta o paquete, seguida de la versión utilizada. Además, se generará automáticamente un archivo y un directorio. El primero es el `package-lock.json`, donde se encuentra todo el árbol de dependencias y el directorio es el `node_modules`, que no está bajo control de versiones y es donde se encuentran los paquetes instalados, para no depender de una ruta externa al proyecto.

Con este paquete se facilitan las confirmaciones más descriptivas y estándar. Para simplificar el uso del mismo, se puede instalar de manera global con la bandera `-g` y ejecutarlo de la siguiente manera:

```
~/prodef-compiler-cpp$ npm i -g commitizen
~/prodef-compiler-cpp$ git add --all
~/prodef-compiler-cpp$ cz
```

Una vez ejecutado el comando `cz`, aparece una interfaz para realizar la confirmación, siguiendo un estándar.

```
prodef-compiler-cpp
├── node_modules
├── .gitignore
├── package-lock.json
├── package.json
└── README.md
```

3.3.3. TypeScript

TypeScript, el lenguaje utilizado en *Prodef*, es un súper conjunto de JavaScript puesto que añade muchas posibilidades a este último. Sin embargo, no es un lenguaje que los navegadores o la consola entiendan y es necesario compilar el código del módulo a JavaScript para posteriormente ejecutarlo.

El archivo `tsconfig.json` es el que indica que un proyecto utiliza TypeScript. Este archivo se coloca en la raíz del proyecto y, aunque puede tener decenas de configuraciones, se utilizará la plantilla disponible en otros módulos de *Prodef*.

```
{
  "compilerOptions": {
    "moduleResolution": "node",
    "target": "es6",
    "module": "es6",
        .
        .
        .
    "outDir": "./dist",
    "declarationDir": "./dist"
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules"]
}
```

Lo más importante, es que la etiqueta `include` indica que el directorio `src` va a contener el código fuente del módulo en TypeScript y que la etiqueta `declarationDir` indica que la salida del código compilado en JavaScript estará en el directorio `dist`.

Para probarlo se instala el compilador de TypeScript, tanto en el proyecto como globalmente para facilitar su uso. Además, se puede crear la carpeta `src` con un archivo de ejemplo `index.ts`, que luego se eliminará.

```
~/prodef-compiler-cpp$ npm i typescript --save-dev
~/prodef-compiler-cpp$ npm i -g typescript

~/prodef-compiler-cpp$ mkdir src
~/prodef-compiler-cpp$ touch src/index.ts
```

En caso de tener un archivo con extensión `.ts` en el directorio `src`, al ejecutar el comando `tsc` se generará el código JavaScript con la configuración del archivo `tsconfig.json`.

Nota: No ejecutar el comando `tsc` con un fichero específico, tal que `tsc index.ts`, ya que se toma una configuración por defecto y no la puesta en el archivo `tsconfig.json`.

```
~/prodef-compiler-cpp$ tsc
```

Nota: En caso de no existir ningún fichero `.ts` en el directorio `src` aparecerá el error "*No inputs were found...*".

```
prodef-compiler-cpp
├── dist
├── node_modules
├── src
│   └── index.ts
├── .gitignore
├── package-lock.json
├── package.json
├── README.md
└── tsconfig.json
```

3.3.4. Empaquetador

A medida que crece la aplicación y se dividen los archivos en módulos, la importación de cada uno no es la manera óptima de trabajar con TypeScript, ya que hace que el servidor ejecute una solicitud HTTP por cada importación y en consecuencia el programa sea menos eficiente. Los empaquetadores mejoraran significativamente el rendimiento de la aplicación con un paquete integrado, es decir, generan un solo fichero que contiene la información realmente necesaria y el servidor solo hace una solicitud con un solo archivo, siendo más eficiente.

En este caso, en el proyecto se implementa el empaquetador *rollup*, el cual cuenta con un archivo de configuración llamado `rollup.config.js`,que está en la raíz del proyecto. Para crearlo se toma la plantilla de otro módulo de *Prodef*. Para que funcione se deben instalar las siguientes librerías:

```
~/prodef-compiler-cpp$ npm i rimraf --save-dev
~/prodef-compiler-cpp$ npm i rollup --save-dev

~/prodef-compiler-cpp$ npm i @rollup/plugin-node-resolve --save-dev
~/prodef-compiler-cpp$ npm i @rollup/plugin-commonjs --save-dev
~/prodef-compiler-cpp$ npm i @rollup/plugin-json --save-dev

~/prodef-compiler-cpp$ npm i rollup-plugin-sourcemaps --save-dev
~/prodef-compiler-cpp$ npm i rollup-plugin-typescript2 --save-dev
~/prodef-compiler-cpp$ npm i rollup-plugin-node-builtins --save-dev

~/prodef-compiler-cpp$ npm i lodash.camelcase --save-dev
```

Una vez instaladas, se pueden ver desde el `package.json`, en el cual se debe modificar la etiqueta `scripts`, tal que:

```
"scripts": {
    "prebuild": "rimraf dist",
    "build": "rollup -c"
},
```

En el `package.json` también se añaden las variables `main`, `module` y `browser`, que serán usadas para definir los nombres de los archivos de salida del `rollup.config.js`.

```
{
  "name": "@ull-prodef/prodef-compiler-cpp",
  "version": "1.0.0",
  "description": "Implementation of a C++ compiler for Prodef language",
  "main": "dist/index.cjs.js",
  "module": "dist/index.esm.js",
  "browser": "dist/index.umd.js",
  "scripts": {
    "prebuild": "rimraf dist",
    "build": "rollup -c"
  },
  .
  .
}
```

Nota: Es importante definir el nombre del proyecto con el prefijo `@ull-prodef/`, ya que más adelante se va a utilizar para crear un módulo privado, en este momento es útil para la configuración de *rollup*, ya que el archivo de configuración tiene el nombre en cuenta.

Finalmente, para ejecutar el empaquetador se ejecuta el comando `build`, que a su vez va a llamar el comando `prebuild` encargado de eliminar el directorio, para luego hacer el empaquetado desde cero con el comando `build`.

```
~/prodef-compiler-cpp$ npm run build
```

Para no ejecutar este comando en cada momento, se añade la etiqueta build:watch dentro de los scripts del `package.json`. De esta manera, el empaquetador detecta cambios en los archivos de TypeScript y se ejecute automáticamente con cualquier cambio.

```
"scripts": {
    .
    .
    "build:watch": "rollup -cw"
},
```

```
~/prodef-compiler-cpp$ npm run build:watch
```

En caso de que no se detecten los cambios al guardar los ficheros TypeScripts que se crearán en la carpeta `src` más delante, se añade la opción `chokidar` en la configuración de *rollup* y se vuelve a ejecutar el comando anterior.

```
watch: {
  include: 'src/**',
  chokidar: {
    usePolling: true
  }
},
```

```
prodef-compiler-cpp
├── dist
├── node_modules
├── src
│   └── index.ts
├── .gitignore
├── package-lock.json
├── package.json
├── README.md
├── rollup.config.js
└── tsconfig.json
```

3.3.5. Permisos

Para tener acceso a librerías privadas de la organización se debe crear el archivo `.npmrc` en la raíz del proyecto. El proceso es el mismo que se hizo en la sección de instalación con el token de GitHub. Es importante añadir este archivo en el `.gitignore`, ya que contiene datos sensibles como el token.

```
//npm.pkg.github.com/:_authToken=TOKEN
@pal-ull:registry=https://npm.pkg.github.com/
@ull-prodef:registry=https://npm.pkg.github.com/
```

```
prodef-compiler-cpp
├── dist
├── node_modules
├── src
│   └── index.ts
├── .gitignore
├── .npmrc
├── package-lock.json
├── package.json
├── README.md
├── rollup.config.js
└── tsconfig.json
```

3.4. Elementos adicionales comunes de un módulo

En esa sección se describirán algunos elementos que son útiles para desarrollar un módulo en *Prodef*, que aunque no son obligatorios para crear un nuevo módulo, son buenas prácticas de programación.

3.4.1. Pruebas

Es recomendado añadir pruebas durante el desarrollo del módulo para asegurarse de que el código funciona como uno piensa que debería funcionar. En el desarrollo de las pruebas se utiliza el marco de pruebas jest.

Las pruebas son añadidas en el directorio `__tests__` en la raíz de `src`, además de crear el fichero de configuración de las pruebas `jest.config.js`.

3.4.2. Formatear código

Establecer un estándar de como se escribe el código es importante para lograr un "*código limpio*", sobre todo en un proyecto donde trabajan distintos programadores. Existen dos tipos de reglas:

- Formato: longitud de línea máxima, tipo de indentación, tamaño de la indentación, entre otras.
- Calidad del código: no permitir variables que no se usen, no permitir declaraciones de variables globales, requerir el uso de objetos de la clase error como argumento del rechazo de una promesa, entre otras reglas.

ESLint es una linter que permite aplicar un conjunto de estándares de estilo, formato y codificación para un código, es decir examina y detecta en que parte del código

no se cumple el estándar. Para utilizarlo, se crea en la raíz del proyecto el fichero `.eslintrc` para la configuración del linter y `.eslintignore` para indicar los ficheros o directorios donde no se quiere aplicar las reglas definidas como, por ejemplo, los directorios `node_modules` o `dist`.

Por otro lado Prettier es un formateador de código, que no solo detecta problemas de estilo de codificación, sino también reescribe el código cada vez que se guarda un archivo. Para ello, se crea el fichero `.prettierrc`, donde se coloca la configuración deseada, la cual no afecta a los ficheros que ya se encuentran en `.eslintignore`.

Ambas herramientas se complementan bastante bien, ya que Prettier disminuye la necesidad de añadir un gran conjunto de reglas de reglas en ESLint, ya que se acopla a un estilo de codificación y ejecuta los cambios directamente en el momento de guardar el fichero, por otro lado ESLint indica los errores relacionados con mañas prácticas de programación. Por lo tanto, la idea es que se use Prettier para formatear (regla de formato) y ESLint para detectar errores (regla de calidad del código).

3.5. Publicación de un módulo

Para publicar un módulo se utiliza GitHub Registry, que es un servicio de alojamiento de paquetes de software, similar a npmjs.org, donde se puede publicar o descargar módulos tanto privados como públicos.

3.5.1. Archivos incluidos

Por defecto la carpeta `dist` que se genera al compilar el programa, no se incluye en el repositorio, ya que está presente en el fichero `.gitignore` como salida por defecto del empaquetador (*rollup*). Pero ocurre lo contrario con un paquete publicado, donde no se quiere el código fuente, pero si los archivos de compilación.

Para ello, se debe indicar en el `package.json` la lista de archivos/carpetas que se desean publicar. Esto se hace modificando la propiedad `files`.

```
{
    "name": "@ull-prodef/prodef-compiler-cpp",
    "version": "1.0.0",
    "description": "Implementation of a C++ compiler for Prodef language",
    "main": "dist/index.cjs.js",
    "module": "dist/index.esm.js",
    "browser": "dist/index.umd.js",
    "files": [
        "dist"
    ],
    .
    .
    .
}
```

Nota: Los ficheros `README.md` y `package.json` se añaden automáticamente.

3.5.2. Scripts

`prepare` se ejecuta antes de empaquetar y publicar el paquete y en local (`npm install`).

```
"prepare": "npm run build"
```

3.5.3. Tipos

Para que TypeScript sepa las definiciones de tipo de cada variable, se modifica el `package.json` en la propiedad `types`.

```
{
    "name": "@ull-prodef/prodef-compiler-cpp",
    "version": "0.0.2",
    "description": "Implementation of a C++ compiler for Prodef language",
    "main": "dist/index.cjs.js",
    "module": "dist/index.esm.js",
    "browser": "dist/index.umd.js",
    "types": "dist/index.d.ts",
    "files": [
        "dist"
    ],
    .
    .
    .
}
```

Nota: En caso de no hacer este paso, al importar el módulo no se detectaran las definiciones de tipo, aunque sean importados en la instalación del módulo y como consecuencia aparecerá un error similar al siguiente:

```
Could not find a declaration file for module '@ull-prodef/prodef-compiler-cpp'. '/home/
    j2mc/Personal/Proyectos/TFG/prodef/prodef-solver-jmetal/node_modules/@ull-prodef/
    prodef-compiler-cpp/dist/index.cjs.js' implicitly has an 'any' type.
```

3.5.4. Publicar

Al estar presente el archivo `.npmrc` en la raíz del proyecto, solo hace falta ejecutar el comando `npm publish`, ya que la información necesaria para publicar el paquete se encuentra en ese fichero.

```
~/prodef-compiler-cpp$ npm publish
```

3.6. Depuración

Para entender el desarrollo del programa y cada uno de sus módulos es necesario la depuración. A continuación, se explica un ejemplo con el módulo `prodef-solver-jmetal`.

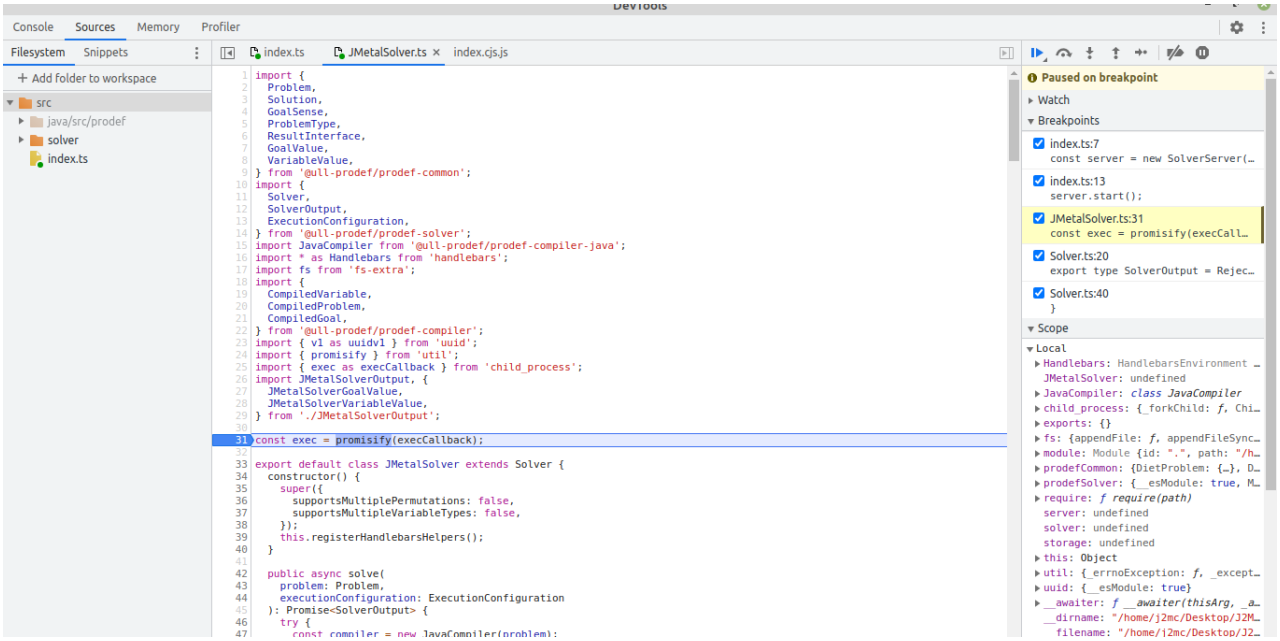


Figura 3.2: Depurar con Google Chrome

3.6.1. Google Chrome

Una opción es colocar dentro de los scripts del `package.json` una opción que nos ofrece depurar en Google Chrome.

```
"scripts": {
    .
    .
    "start:dev": "node --inspect-brk dist/index.cjs.js",
    .
    .
},
```

Se toma cualquier fichero `index` generado por el empaquetador y, en este caso, se opta por `index.cjs.js`. No importa el fichero elegido, ya que fueron generados con *source map*, lo que significa que al ejecutar el script añadido, se va a depurar el código que se desarrolla en TypeScript y no el código empaquetado en JavaScript.

```
~/prodef/prodef-solver-jmetal$ npm run start:dev
```

Una vez ejecutado, se abre el navegador Google Chrome y se escribe la ruta `chrome://inspect`. Gracias a las herramientas integradas del navegador, para el desarrollo web, se podrá ver el fichero para su depuración, tal y como se muestra en la Figura 3.2.

3.6.2. Visual Studio Code

Sin embargo, la opción anterior al usar el subsistema Linux en Windows (wsl) puede dar error, ya que las rutas no se traducen entre `/mnt/c` (Linux) y `/c` (Windows), lo que da error al cargar los *source maps* y por lo tanto, se termina depurando con los módulos generados en JavaScript y no en TypeScript.

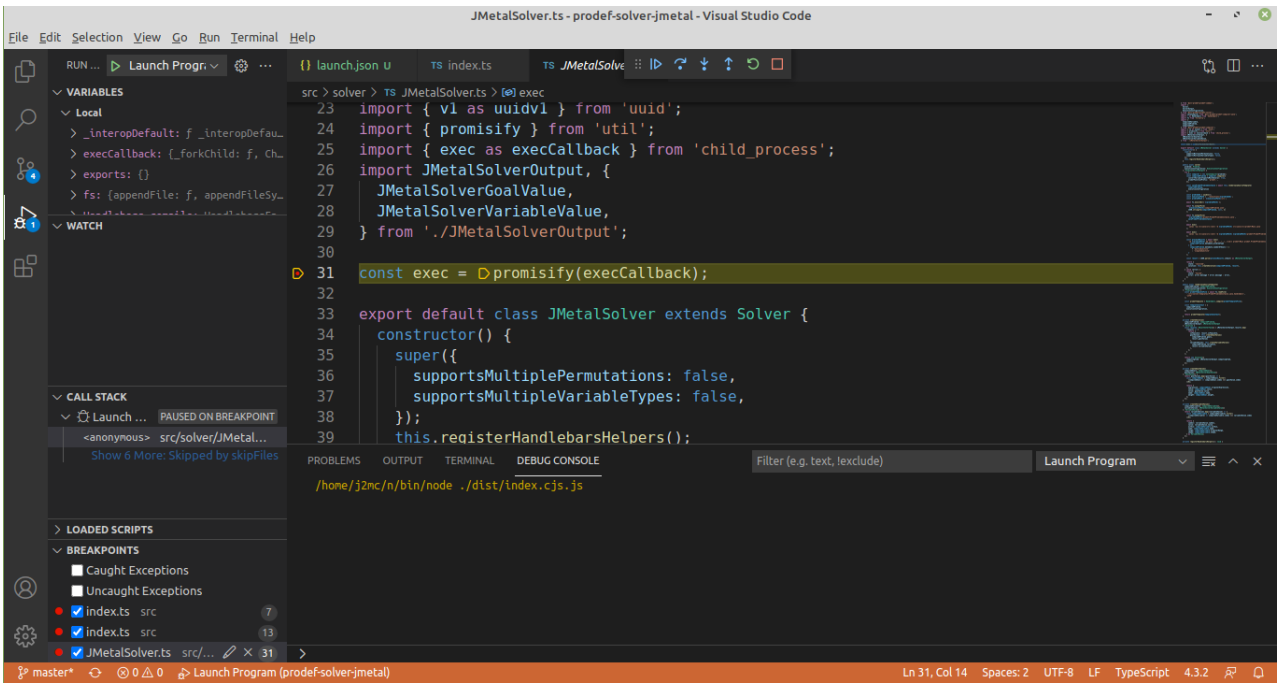


Figura 3.3: Depurar con Visual Studio Code

Para solucionar esto, se puede depurar desde Visual Studio Code, lo cual consiste en generar un archivo de configuración dentro de `.vscode` llamado `launch.json` y ejecutar el programa desde la interfaz de Visual Studio Code, como se ve en la Figura 3.3. Para una información más detallada se puede consultar la guía de depuración.

La plantilla utilizada como depurar en el fichero `launch.json` es la siguiente:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Launch Prodef",
      "program": "${workspaceFolder}/src/index.ts",
      "request": "launch",
      "skipFiles": [
        "<node_internals>/**"
      ],
      "type": "pwa-node"
    }
  ]
}
```

Capítulo 4

Compilador C++

Debido a que *METCO* está desarrollado en C++, se debe crear un compilador que traduzca el problema definido en el lenguaje *prodef-lang* al lenguaje de programación C++. Para ello, se ha creado el módulo que se describe a continuación.

4.1. prodef-compiler-cpp

Partiendo del código fuente con la estructura común de un módulo vista en el Capítulo 3, se puede continuar con el desarrollo del módulo específico *prodef-compiler-cpp*, el cual se alojará en un repositorio de la organización y tendrá la siguiente estructura de directorios.

```
prodef-compiler-cpp
├── dist
├── node_modules
├── src
│   ├── functions
│   │   ├── CosFunctionCompiler.ts
│   │   ├── SinFunctionCompiler.ts
│   │   └── SqrtFunctionCompiler.ts
│   ├── CppCompiler.ts
│   └── index.ts
├── .gitignore
├── .npmrc
├── package-lock.json
├── package.json
├── README.md
├── rollup.config.js
└── tsconfig.json
```

Para construir módulos específicos, solo será necesario desarrollar la estructura interna de la carpeta `src` (que en este punto debe estar vacía). En caso de que exista algún fichero de prueba dentro de `src` debe ser eliminado.

4.2. Dependencias

El primer paso consiste en instalar dos dependencias de *Prodef* que contienen las interfaces comunes y la base del compilador, respectivamente.

```
~/prodef-compiler-cpp$ npm i @ull-prodef/prodef-common
~/prodef-compiler-cpp$ npm i @ull-prodef/prodef-compiler
```

Nota: Es importante que exista el archivo `.npmrc` con su respectivo *token* para tener los permisos necesarios para la instalación, de lo contrario la instalación fallará.

En este punto la carpeta `src` sigue vacía, ya que se modificó únicamente el `package.json` con la instalación de las dependencias.

4.3. Funciones de prodef-lang

El objetivo perseguido a la hora de diseñar *prodef-lang* era proporcionar un lenguaje de especificación de problemas, sencillo e intuitivo, para que cualquier persona sin conocimientos previos de programación pudiera utilizarlo. Por eso, no se debe caer en el error de añadirle tantas funcionalidades que convierta a *prodef-lang* en un lenguaje equivalente a un lenguaje de programación de alto nivel como Java o C++, ya que perdería su razón de ser.

Para añadir funciones se crea la carpeta `functions`, en la cual se van a definir las funciones básicas coseno, seno y raíz cuadrada, las cuales deben ser definidas en el lenguaje objetivo. Por ejemplo, para la función coseno se crea el fichero con el nombre de la clase `CosFunctionCompiler` tal que:

```
import { FunctionCompiler } from '@ull-prodef/prodef-compiler';

export default class CosFunctionCompiler extends FunctionCompiler {
  public compile(): string {
    return 'double cos_(double number) { return std::cos(number); }'
  }
}
```

Para definir la función coseno, se puede utilizar como plantilla su implementación en otro compilador, ya que lo único que cambia entre un compilador y otro es lo que retorna la función como string. En este caso, se retorna la función definida en el lenguaje objetivo C++.

```
double cos_(double number) {
    return std::cos(number);
}
```

La función coseno en C++ no funciona totalmente, ya que depende de la librería `cmath` pero en la definición de estas funciones no se incluye la importación de las librerías necesarias, tal y como se explicará en el Capítulo 5.

Nota: La idea es que todas las funciones se definan con los mismos nombres de `cos`, `sin` y `sqrt` para mantener una consistencia y que el código sea más fácil de entender para otro programador. Pero en este caso, no se puede mantener los nombres, debido a que la librería `cmath` declara las funciones en el espacio de nombres global y ocurre un

conflicto por ambigüedad. Por tal razón, se agrega el símbolo "_" al final del nombre de la función y queda `cos_` en vez de `cos`.

A partir de la definición de las funciones coseno, seno y raíz cuadrada se tiene la siguiente estructura dentro de `src`:

```
src
└─ functions
   ├─ CosFunctionCompiler.ts
   ├─ SinFunctionCompiler.ts
   └─ SqrtFunctionCompiler.ts
```

4.4. Compilador

Para la creación del núcleo del compilador se crea el fichero `CppCompiler.ts` en la raíz de `src`. Este fichero va a contener las llamadas a las funciones definidas anteriormente, así como el resto de traducciones al lenguaje objetivo C++ desde *prodef-lang*.

Se define la clase `CppCompiler` como extensión de la clase abstracta `CBasedCompiler`, que es un compilador que facilita la implementación de nuevos compiladores para lenguajes basados en el sintaxis del lenguaje de programación C. De esta forma, se puede reaprovechar la estructura general y simplemente implementar o sobrescribir algunos métodos específicos.

```
export default class CppCompiler extends CBasedCompiler {

}
```

4.4.1. Constructor

En este método se debe especificar tanto las funciones del lenguaje *prodef-lang* como la configuración del compilador:

```
const defaultCBasedOptions = {
    types: new Map<WithinType, string>([
      ['integers', 'int'],
      ['reals', 'double'],
    ]),
    bracketsPosInDeclarations: 'afterName',
    explicitySetDimensionsInDeclarations: true,
    useNewKeywordInVariableDeclarations: false,
};
```

Cuenta con la anterior configuración por defecto, donde cada uno de estos valores modifica la sintaxis del lenguaje objetivo. A continuación se muestra cómo varían fragmentos de código con distintas configuraciones, las cuales se colocan como comentario.

- `types`: Indica los tipos de datos válidos en el lenguaje objetivo, utilizando un `hash` que asigna el valor para los números enteros y reales con la clave `integers`.

```
private int[] __x = new int[(int) (5)];      // ['integers', 'int']
private Integer[] __x = new int[(int) (5)];  // ['integers', 'Integer']
```

- `bracketsPosInDeclarations`: Indica la posición de los corchetes al definir un `array`.

```
private int[] __x = new int[(int) (5)];  // afterType
private int __x[] = new int[(int) (5)];  // afterName
```

- `explicitySetDimensionsInDeclarations`: Indicar si añade el tamaño del `array` al momento de la declaración.

```
private int[] __x = new int[(int) (5)];   // false
private int[5] __x = new int[(int) (5)];  // true
```

- `useNewKeywordInVariableDeclarations`: Indica si se utiliza la palabra reservada `new` para crear un `array`.

```
private int[] __x = new int[(int) (5)];  // true
private int[] __x;                       // false
```

A partir de estos parámetros se tiene que elegir una combinación específica que se adapte al sintaxis del lenguaje de programación C++, como se ve a continuación:

- Los tipos de variables reconocidos en C++ son `int` y `double` para los números enteros y reales respectivamente, por lo cual el atributo `types` tiene el siguiente valor:

```
types: new Map<WithinType, string>([
  ['integers', 'int'],
  ['reals', 'double'],
]),
```

- La declaración de los *array* en C++ establece que los corchetes se fijan después del nombre de la variable a diferencia de otros lenguajes de programación como Java que los colocan antes, en consecuencia el atributo `bracketsPosInDeclarations` tiene el valor `afterName`.

```
dataType vectorName[size];
```

El resto de la configuración no depende tanto de la sintaxis del lenguaje sino con la definición de los problemas.

- El tamaño de los *array* es estático durante la ejecución de *METCO*, por ejemplo el problema Knapsack utiliza un vector para representar al conjunto de objetos que se pueden meter o no en la mochila, si hay 100 objetos el *array* tiene 100 posiciones y ese tamaño no se modificará a lo largo de la ejecución. Por lo tanto, los *arrays* se definen con un tamaño fijo y en consecuencia `explicitySetDimensionsInDeclarations` tendrá el valor *true*.
- Continuando la idea anterior, tampoco se utilizará la palabra reservada *new* para inicializar los *arrays*, declarando el atributo `useNewKeywordInVariableDeclarations` con el valor `false`, ya que se utilizaran tamaños estáticos, evitando así la complejidad del manejo de memoria dinámica y punteros en C++.

Una vez definidos los valores ideales para cada atributo de la configuración del compilador según la **sintaxis de C++** y **necesidades del problema**, se observa que concuerda con la configuración por defecto mostrada inicialmente, con la cual se obtiene el siguiente resultado:

```
private int __x[5];
```

Finalmente el constructor queda tal y como se muestra a continuación:

```
export default class CppCompiler extends CBasedCompiler {
    constructor(problem: Problem) {
        super(problem);

        this.functions.getFunctions().forEach((functionItem) => {
          switch (functionItem.name) {
            case 'sin':
                this.registerFunctionCompiler(
                    functionItem.name,
                    new SinFunctionCompiler(functionItem)
                );
                break;
            case 'cos':
                this.registerFunctionCompiler(
                    functionItem.name,
                    new CosFunctionCompiler(functionItem)
                );
                break;
            case 'sqrt':
                this.registerFunctionCompiler(
                    functionItem.name,
                    new SqrtFunctionCompiler(functionItem)
                );
                break;
            }
        });
    }
}
```

4.4.2. Valor absoluto

Es un método que dado una expresión escrita en el lenguaje objetivo C++, retorna otra expresión resultado de aplicar la operación matemática valor absoluto a la expresión original.

```
compileAbsoluteValue(type: WithinType, expression: string): string {
    return this.compileCastExpression(type, `std::abs(${expression})`);
}
```

4.4.3. Valor mínimo

Es un método que dada dos expresiones escritas en el lenguaje objetivo, retorna una expresión que devuelve el valor mínimo entre las dos expresiones iniciales.

```
compileMinimumValue(type: WithinType, expression1: string, expression2: string): string {
```

```
    return this.compileCastExpression(type, `std::min(${expression1}, ${expression2})`);
}
```

4.4.4. Potencia

Es un método que dada una expresión escrita en el lenguaje objetivo que representa la base de la operación potencia y otra que simboliza el exponente, devuelve la expresión producto de aplicar la operación matemática potencia.

```
compileExponentialOperation(
    type: WithinType,
    baseExpression: string,
    exponentExpression: string
): string {
    return this.compileCastExpression(
        type,
        `std::pow(${baseExpression}, ${exponentExpression})`
    );
}
```

4.4.5. Límites máximos y mínimos

Es un método que dado un tipo de variable (entero o real) y un signo (positivo o negativo), retorna una expresión escrita en el lenguaje objetivo que representa el valor "*infinito*", es decir, el máximo valor positivo o negativo del conjunto de números enteros o reales del lenguaje de programación C++.

```
compileInfinity(type: WithinType, positiveInfinity = true): string {
    let result = '';
    switch (type) {
        case 'integers':
            if (positiveInfinity) {
                result = 'INT_MAX';
            } else {
                result = 'INT_MIN';
            }
            break;
        case 'reals':
            if (positiveInfinity) {
                result = 'DBL_MAX';
            } else {
                result = 'DBL_MIN';
            }
            break;
    }
    return result;
}
```

Nota: Los nombres como `INT_MAX` o `DBL_MIN` son constantes de las librerías de C++ `climits` y `cfloat`, respectivamente.

4.4.6. Renombrar variables

Es un método que dado un símbolo, retorna otro símbolo válido en el lenguaje objetivo con un prefijo para evitar colisiones en los nombres de otros símbolos.

```
transformSymbol(symbol: string): string {
    return `__${symbol}`;
}
```

4.5. Ciclos

Es un método que define como se van a generar en el lenguaje objetivo los bucles para las operaciones como sumatorio y producto entre otras. Estas operaciones se utilizarán generalmente para el cálculo de las funciones objetivo y para la comprobación del cumplimiento de las restricciones. A continuación se puede ver parte del código implementado para *prodef-solver-jmetal* para el lenguaje Java con la implementación del sumatorio.

```
compileLoop(
    symbol: string,
    expression: string,
    reducer: ReducerType,
    start?: string | null,
    end?: string | null,
    vector?: string | null
): string {

    ...
    const rangeExpression = `IntStream.range((int) Math.min(${start}, ${end}), (int) Math
    .max(${start}, ${end}) + 1)`;
    ...
    return `${rangeExpression}.mapToDouble(${symbol} -> ${expression}).reduce(0, Double::
    sum)`;
}
```

La función anterior recibe como parámetro el inicio y final del sumatorio, junto a la expresión que se evalúa en cada iteración, lo que es equivalente a la siguiente notación matemática:

$$\sum_{i=start}^{end} expression$$

A lo largo de esta sección será de utilidad trabajar con un caso específico junto a las palabras "*strart*", "*end*" y "*expression*" para explicar mejor el razonamiento del desarrollo de este método. Por ejemplo, el rango 1 y 5 con la expresión $x_i - 1$; sustituyendo estos valores tanto en la notación matemática como en el código de Java quedaría lo siguiente:

$$\sum_{i=1}^{5} x_i - 1$$

```
// Sustituyendo el rango
IntStream.range(1, 6).mapToDouble(${symbol} -> ${expression}).reduce(0, Double::sum)

// Sustituyendo el simbolo y expresion
IntStream.range(1, 6).mapToDouble(_i -> __x[__i] - 1).reduce(0, Double::sum)
```

4.5.1. Bucle for

La idea es representar las operaciones como **sumatorio** y **producto** en C++, utilizando como ejemplo el desarrollo hecho en *prodef-solver-jmetal* con Java. Siendo la primera aproximación el uso programación clásica con el ciclo `for`, ya que se repite constantemente una operación dentro de un rango especifico por cada uno de sus índices.

```
for (int __i = start; __i <= end; __i ++) {
    ${expression}
    ...
}

// Sustituyendo los valores
for (int __i = 1; __i <= 5; __i ++) {
    __x[__i] - 1;
    ...
}
```

Se espera que se retorne una expresión que se pueda evaluar como un **valor primitivo** en el lenguaje de programación objetivo C++, los valores primitivos son estructuras de datos integradas o definidas en el lenguaje de programación como `int`, `double`, `bool`, entre otros, a partir de los cuales se pueden crear otros tipos de datos derivados como los objetos.

```
double result = for (int __i = start; __i <= end; __i ++) {
    __x[__i] - 1;
    ...
}
```

Esta alternativa falla porque se espera un valor primitivo y el bucle `for` no se puede evaluar como un valor primitivo e igualar como el valor de una variable, por la definición de la sintaxis del lenguaje C++.

4.5.2. Extender el lenguaje prodef-lang

Añadir nuevas funcionalidades al lenguaje *prodef-lang* como las funciones coseno, seno y raíz cuadrada explicadas anteriormente evita el problema anterior, debido a que las funciones auxiliares sirven de envoltorio para cada bucle y son llamadas desde el método `compileLoop`.

```
// Definicion de la funcion
export default class SumFunctionCompiler extends FunctionCompiler {
  public compile(): string {
    return `double sum(int start, int end, std::string expression) {
        for (int __i = start; __i <= end; __i ++) {
            // Se debe evaluar la expresion
            expression
```

```
            ...
        }
    }‘
  }
}

// Llamada de la funcion
double obj = SumFunctionCompiler();
double result = obj.sum(start, end, expression);
```

Las funciones son una abstracción del ciclo `for` que retornan un valor primitivo, el cual puede ser igualado a un variable. Sin embargo, ocurre un problema relacionado con el **momento de compilación**, ya que estas funciones deben ser totalmente funcional en el lenguaje objetivo, es decir, este código se encuentra compilado antes de realizar el proceso de compilación del problema.

Por lo tanto, no se puede pasar como parámetro la expresión que representa código funcional en C++, a menos que en tiempo de ejecución dicha expresión sea evaluada con una función similar a eval, la cual es una función de JavaScript que permite evaluar código representado como un string, pero que no está presente en C++ y que por lo general no es recomendado su uso, porque permite inyectar código fuente malicioso y que sea ejecutado.

4.5.3. Función wrapper

Para evitar el problema relacionado con el momento de la compilación, se consideró una aproximación similar a la anterior con una función que envuelva al ciclo `for`, pero en vez de implementarla como extensión del lenguaje, es parte de la definición del problema.

```
// Definicion de la funcion
double sum(int start, int end) {
    for (int __i = start; __i <= end; __i ++) {
        __x[__i] - 1;
        ...
    }
}

// Llamada de la funcion
double result = sum(start, end);
```

Esta función se compila junto con el problema y en consecuencia no se tiene que pasar como parámetro la expresión que representa la operación interna $(x_i - 1)$ del sumatorio. En el código anterior de la función falta el acumulador, el cual se retorna al final de la misma con el resultado total, quedando:

```
double sum(int start, int end) {
    double result;
    for (int __i = start; __i <= end; __i ++) {
        result += __x[__i] - 1;
    }
    return result;
}
```

Sin embargo, esto trae consigo un problema porque se desconoce la **complejidad**

de la expresión, la cual puede tener comparaciones, declaraciones de variables, ciclos, entres otras expresiones.

```
result += {
    double aux = __x[__i];
    if (__x[__i] % 2 == 0) {
        return aux - 1:
    } else {
        return  aux + 1;
    }
}
```

Es como si la expresión se tratase de otro programa, que no puede igualarse como el valor primitivo de una variable, en consecuencia esta alternativa no termina de satisfacer las necesidades de la sintaxis de C++.

4.5.4. Paradigma funcional

Como se vio en la versión de Java, existe una serie de funciones como `range`, `reduce` y `mapToDouble` que están concatenadas unas después de otras, debido a que la idea es retornar una cadena que se evalúa como un valor primitivo y esto no es posible si el código se elabora en varios pasos.

```
IntStream.range(1, 6).mapToDouble(_i -> __x[__i] - 1).reduce(0, Double::sum)
```

Las funciones lambda se adaptan al sintaxis de las expresiones que vienen del compilador base, sin importar la complejidad de las mismas como se nombró en la alternativa anterior.

```
[this](int ${symbol}) -> double {
    return {expression};
};

// Sustituyendo el simbolo y expresion
[this](int __i) -> double {
    double aux = __x[__i];
    if (__x[__i] % 2 == 0) {
        return aux - 1:
    } else {
        return  aux + 1;
    }
};
```

Es importante incluir dentro de los corchetes la palabra reservada `this`, para que las variables que pertenecen a la instancia del problema como `__x` estén dentro del contexto adecuado; de lo contrario esas variables no se encontrarán.

A pesar de contar con las funciones *lambdas*, C++ no implementa la misma sintaxis para el paradigma funcional como en otros lenguajes de programación como JavaScript o Java, los cuales permiten **concatenar funciones** como:

```
objeto.map().filter().reduce()
```

Para evitar este problema se combina la alternativa de la función *wrapper* con programación funcional en C++, evitando así los dos problemas, la complejidad de las expresiones y la concatenación de funciones.

```cpp
double sumLoop(int start, int end, std::function<double(double)> lambda) {
    // Creacion de indices
    std::vector<int> indixes(end);
    std::iota(indixes.begin(), indixes.end(), start);

    // Operacion interna del sumatorio
    std::vector<double> output(end, 0);
    std::transform(indexes.begin(), indexes.end(), output.begin(), lambda);

    // Sumatorio de resultados parciales
    return std::accumulate(output.begin(), output.end(), 0);
}
```

Con esta idea se solucionan todos los inconvenientes planteados, donde cada ciclo se apoya en una función auxiliar, similar a la anterior, definida en el código de la clase del problema.

```
compileLoop(
    symbol: string,
    expression: string,
    reducer: ReducerType,
    start?: string | null,
    end?: string | null,
    vector?: string | null
): string {
    ...
    switch (reducer) {
        case 'sum':
            return `sumLoop(${start}, ${end}, ${lambdaFunc})`;
        case 'mult':
            return `multLoop(${start}, ${end}, ${lambdaFunc})`;
        case 'assert':
            return `assertLoop(${start}, ${end}, ${lambdaFunc})`;
        case 'min':
            return `minLoop(${start}, ${end}, ${lambdaFunc})`;
    }
}
```

4.6. Exportaciones

En este punto ya se ha terminado el compilador en el lenguaje objetivo C++, pero es una buena práctica crear un fichero que contenga todas la exportaciones, con la finalidad de que un consumidor del módulo realice las importaciones desde un mismo punto, por lo cual se creó el fichero `index.ts` en la raíz del directorio `src`.

```ts
import CppCompiler from './CppCompiler';
export default CppCompiler;
```

```
src
├── functions
│   ├── CosFunctionCompiler.ts
│   ├── SinFunctionCompiler.ts
│   └── SqrtFunctionCompiler.ts
├── CppCompiler.ts
└── index.ts
```

4.7. Flujo de trabajo

Para definir la instancia de un problema de manera visual, el usuario utiliza una aplicación web desarrollada a partir del módulo *prodef-gui* (ya mencionado en el Capítulo 3). Una vez que el usuario finaliza la definición del problema, este se traduce mediante el módulo *prodef-lang* en un objeto *json* que contiene toda la información necesaria del problema a partir de la cual trabaja el módulo `prodef-compiler-cpp`. A continuación se muestra un ejemplo simplificado del fichero *json* que recibe el compilador para el *TSP*.

```
{
    "executionConfiguration": {...},
    "problem": {
        "name": "Basic TSP problem",
        "description": "TSP problem using a permutation vector",
        "parameters": [...],
        "variables": [...],
        "goals": [...],
        "constraints": [...]
        "classes": [...],
        "objects": [...]
    }
}
```

Figura 4.1: Problema antes de compilar

Para explicar algunos de estos atributos, se tomará un problema de ejemplo, en este caso, será el *TSP*.

- `executionConfiguration`: Configuración de la ejecución, esto es, máximo tiempo de procesamiento y máximo número de soluciones.
- `problem`: Contiene a su vez otros atributos con la información siguiente:
  - `name`: Cadena con el nombre del problema.
  - `description`: Cadena con la descripción del problema.
  - `variables`: Conjunto de variables del problema, por ejemplo el *array* que contiene la permutación con el orden de las ciudades visitadas.
  - `goals`: Función objetivo para calcular la distancia de las ciudades visitadas en el orden definido previamente.
  - `constraints`: Conjunto de restricciones del problema. En este caso es un *array* vacío, debido a que por la manera en que está definido este problema, no tiene restricciones.
  - `classes`: Es la clase que representación "*algo*" del problema, es una abstracción del paradigma de programación orientado a objetos (*POO*). En este caso, se define una clase que representa una ciudad y sus distancias al resto de ciudades.

- `objects`: Al igual que el atributo anterior, es un concepto que viene del paradigma *POO*, donde se representa a cada instancia de la clase, en este caso sería un objeto por cada ciudad del problema.

El compilador recibe como entrada el problema de la Figura 4.1 en el lenguaje *prodef-lang* y retorna como salida el problema en el lenguaje C++ y en formato *Json*, tal y como se muestra en la Figura 4.2.

```
{
  "metadata": {...},
  "parameters": [...],
  "objects": [...],
  "variables": [...],
  "functions": [...],
  "goals": [...],
  "constraints": [...]
}
```

Figura 4.2: Problema compilado

A pesar de que ambos problemas tienen una estructura muy similar, las propiedades comunes varían su contenido según el lenguaje utilizado, ya sea C++ o *prodef-lang*.

Las nuevas propiedades representan lo siguiente:

- `metadata`: Nombre y descripción del problema.
- `functions`: Contiene las funciones definidas en el lenguaje (coseno, seno y raíz cuadrada).

Capítulo 5

Resolutor de METCO

Tras compilar el problema al lenguaje objetivo C++ (a través del módulo *prodef-compiler-cpp*) llega el momento de aplicar alguna herramienta para la resolución del problema. La herramienta de resolución (resolutor) calculará las soluciones al problema mediante algún algoritmo de optimización. En este caso, haremos uso de los algoritmos evolutivos implementados en la herramienta *METCO* [11]. Para ello, se necesita crear un módulo para comunicar *Prodef* con *Metco*. El proceso de desarrollo de dicho módulo se describirá a lo largo de este capítulo.

5.1. prodef-solver-metco

Al desarrollar un nuevo módulo como en el Capítulo 4, se parte de la estructura común de explicada en el Capítulo 3. El módulo se llamará *prodef-solver-metco*, que se desarrollará y alojará en un repositorio de la organización y tendrá la siguiente estructura de directorios.

```
prodef-solver-metco
├── dist
├── node_modules
├── executions
├── environment
│   └── environment.ts
├── src
│   ├── solver
│   │   ├── templates
│   │   │   ├── ProdefProblemInstance.cpp.handlebars
│   │   │   └── ProdefProblemInstance.h.handlebars
│   │   └── MetcoSolver.ts
│   └── index.ts
├── .gitignore
├── .npmrc
├── package-lock.json
├── package.json
├── README.md
├── rollup.config.js
└── tsconfig.json
```

5.2. Dependencias

Se instalan las librerías de *Prodef* necesarias como las interfaces comunes, tanto el compilador base como C++ y el código base de los resolutores.

```
~/prodef-solver-metco$ npm i @ull-prodef/prodef-common
~/prodef-solver-metco$ npm i @@ull-prodef/prodef-solver
~/prodef-solver-metco$ npm i @ull-prodef/prodef-compiler
~/prodef-solver-metco$ npm i @ull-prodef/prodef-compiler-cpp
```

5.3. Plantilla

Para el desarrollo del problema se utiliza handlebars, que es un lenguaje de plantillas con un enfoque útil, ya que permite separar los datos del diseño, esto significa que se puede **mantener una lógica** y cambiar los datos sin que uno afecte al otro. Esta capacidad resulta interesante para definir una estructura genérica de un problema (diseño/lógica) y a partir características específicas (datos) y un conjunto de reglas, se puede crear una instancia de ese problema.

```
~/prodef-solver-metco$ npm i handlebars
```

Utilizando este potencial se definió una plantilla que representa un problema genérico de *Prodef* en *METCO* y, junto al problema compilado de la Figura 4.2, se crea una instancia específica de ese problema en `METCO`. Para ello, se desarrollaron dos plantillas en el directorio `src/solver/templates`:

- `ProdefProblemInstance.hpp.handlebars`: Contiene la **definición** de los métodos del problema.
- `ProdefProblemInstance.cpp.handlebars`: Contiene la **implementación** de los método del problema

¿Por qué crear dos plantillas en vez de una? La separación del código en dos ficheros, uno para la definición (`hpp`) y otro para la implementación (`.cpp`), se debe a la manera en que funciona la compilación en C++, la cual se divide en diferentes etapas (véase la Figura 5.1) y conocer este proceso hará que la compilación y vinculación ahorren mucho tiempo.

Un programador desarrolla un programa en C++ que por lo general tiene múltiples archivos, entre los cuales se encuentran los archivos fuentes con la extensión `.cpp`, los cuales puede incluir otros archivos con la directiva `#include`, con la idea de dividir y reutilizar el código.

Mientras que el compilador en primera instancia llamará al **preprocesador** para remplazar la línea que contiene la directiva `#include` con todo el contenido del archivo incluido. Una vez que el preprocesador termina, el compilador inicia la fase de **compilación** con todos los archivos fuente con las directivas sustituidas por el contenido de los ficheros, produciendo los archivos objeto. Por último, la **vinculación** combina las unidades de traducción compiladas en un único programa para generar el ejecutable.

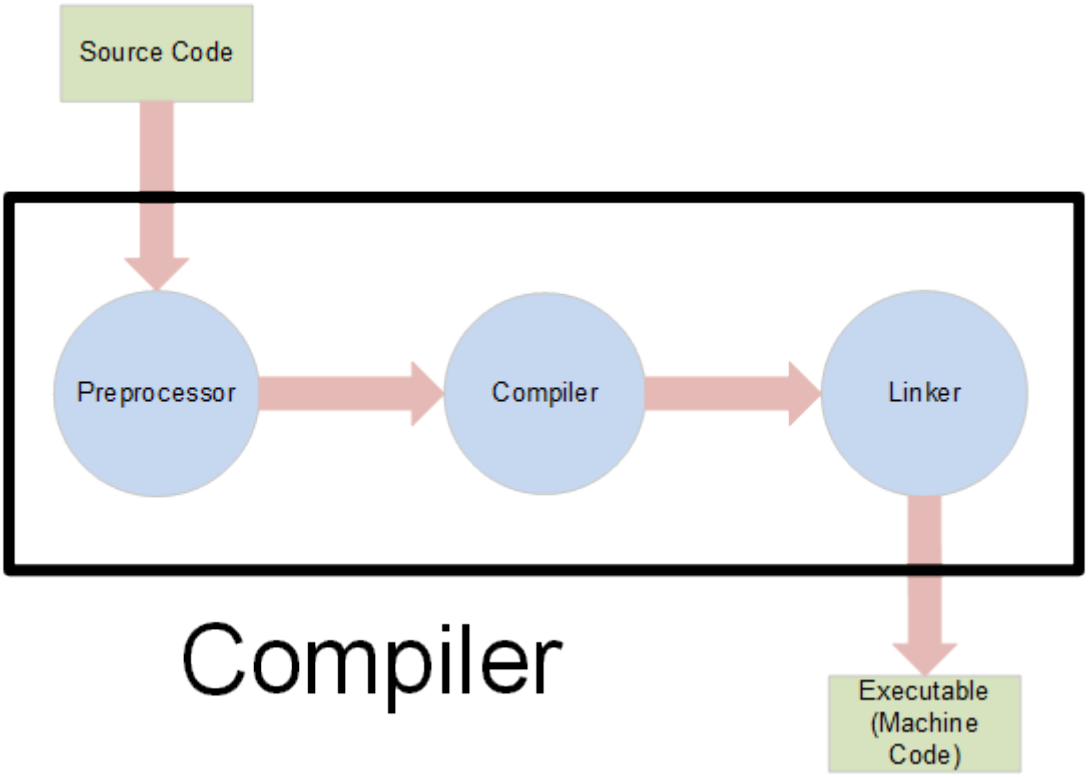


Figura 5.1: Proceso de compilación en C/C++

Si las directivas `#include` importan los ficheros `.cpp` en vez de los archivos de encabezado con la extensión `.hpp`, hace que los archivos objeto sean extremadamente grandes y como consecuencia todo el proceso de compilación sea más lento.

El compilador no necesita saber la implementación del código, ni siquiera la ubicación del archivo `.cpp`, todo lo que necesita se encuentra en los archivos de encabezado `.hpp`, que le dicen al compilador lo que puede esperar que esté disponible en el momento del enlace, más no como funciona, logrando con esta separación que los archivos objetos sean más pequeños y el proceso de compilación más eficiente.

5.3.1. ProdefProblemInstance.hpp.handlebars

Esta plantilla representa la clase que define un problema *Prodef* en *METCO*, siendo un diseño muy similar sea cual sea el problema, debido a que prácticamente todo el código es común para cualquier instancia del problema, menos un conjunto de atributos rodeados por llaves `{{expresión}}` como se puede ver a continuación con una simplificación de la plantilla.

```
#ifndef __PRODEF_PROBLEM_INSTANCE_H__
#define __PRODEF_PROBLEM_INSTANCE_H__

// Incluir librerias necesarias

class ProdefProblemInstance : public IndividualWithViolation {
  private:

    bool isBinary;
    bool isIntegerPermutation;
```

```
11
12      // Parametros
13      {{#each compiledProblem.parameters}}
14        {{{get_static_declaration expression}}}
15      {{/each}}
16
17      // Objetos
18      {{#each compiledProblem.objects}}
19        {{{get_static_declaration expression}}}
20      {{/each}}
21
22      // Variables
23      {{#each compiledProblem.variables}}
24        {{{expression}}}
25      {{/each}}
26
27    public:
28
29      ProdefProblemInstance();
30
31      bool init(const std::vector<std::string>& params);
32      void restart();
33
34      void dependentMutation(double mutationProbability);
35      void dependentCrossover(Individual* individual);
36
37      void updateBinaryVariables();
38      Individual* clone() const;
39
40      unsigned int getOptDirection(const int index) const;
41
42      ~ProdefProblemInstance();
43
44    private:
45
46      double evaluateGoal(int goalIndex) const;
47      bool evaluateConstraint(int constraintIndex) const;
48      double evaluateConstraintViolation(int constraintIndex) const;
49
50      double sumLoop(int start, int end, std::function<double(double)> lambda) const;
51      double multLoop(int start, int end, std::function<double(double)> lambda) const;
52      double assertLoop(int start, int end, std::function<double(double)> lambda) const;
53      double minLoop(int start, int end, std::function<double(double)> lambda) const;
54
55      double getMaximum(const int variableIndex) const;
56      double getMinimum(const int variableIndex) const;
57  };
58
59  #endif
```

En este punto se añaden las librerías `climits` y `cfloat`, tal y como se mencionó en el Capítulo 4. También se sobrescriben los métodos necesarios de la clase `Individual` de *METCO* tal y como se describió en el Capítulo 2.

Sin embargo, esta clase no hereda directamente de `Individual` sino de la clase `IndividualWithViolation` que a su vez sí hereda de esta. Esta abstracción se debe a la manera en que está diseñada la clase `Individual`, ya que está pensada para obtener solo

soluciones factibles utilizando el método `repair` de cada instancia.

La estrategia actual de *METCO* no es viable, porque desde la interfaz de *Prodef* no es posible especificar la manera de mejorar soluciones hasta lograr la factibilidad. Por ello, se ha adaptado *METCO* con esta nueva clase para tratar con problemas que violan momentáneamente sus restricciones, donde se asigna un **castigo** en función de su grado de violación, ordenando las soluciones según dos criterios.

1. **Grado de violación**: Se asigna cero si es una solución factible, de lo contrario un valor negativo, que se calcula según el valor en que las restricciones del problema no se cumplen. Por ejemplo, una instancia del problema *Knapsack* con una capacidad de 1000 y un peso 1200, tiene un grado de violación de -200, producto de la resta entre la capacidad y el peso de los objetos dentro de la mochila.
2. **Valor objetivo**: En caso de empate por el anterior criterio, se utiliza el valor de la función objetivo.

Las soluciones se ordenan de mayor a menor grado de violación, siendo cero el valor más alto, porque se utiliza el rango de enteros negativos; dentro de este criterio se organizan de mejor a peor valor de la función objetivo.

Lo ideal sería disponer de un algoritmo universal capaz de derivar desde la formulación del problema una heurística de búsqueda local capaz de producir soluciones factibles. Desafortunadamente, en el momento actual, la construcción de tales heurísticas es un proceso manual ligado a cada tipo de problema.

5.3.2. ProdefProblemInstance.cpp.handlebars

Esta plantilla contiene la implementación de cada uno de los métodos comunes del problema, pero a diferencia de la anterior plantilla, no se mantiene un diseño tan similar entre los distintos problemas, ya que cada método es dependiente del problema implementado como se ve a continuación con uno de sus métodos:

```
double ProdefProblemInstance::evaluateGoal(int goalIndex) const {
  switch (goalIndex) {
    {{#each compiledProblem.goals}}
      case {{{index}}}:
        return {{{expression}}};
    {{/each}}
  }
  throw std::out_of_range("goal for the required index doesn't exist");
}
```

Mientras que en la primera plantilla casi la totalidad del código es lenguaje C++, en la segunda cada método tiene un aspecto similar a `evaluateGoal`, donde se mezcla código C++ con textos que tienen el formato `{{expresión}}` o `{{{expresión}}}`. Dichos fragmentos representan el lenguaje *handlebars*, donde cada expresión que está dentro de las llaves puede ser cualquier cosa que pertenezca al objeto que representa al problema compilado de la Figura 4.2.

En el caso del método `evaluateGoal` se utiliza el conjunto de funciones objetivo definidas en el problema compilado y, con las palabras reservadas del lenguaje *handlebars* (como **each**), se recorre el `array` de funciones objetivo y se anexan al código C++.

Al finalizar el desarrollo de las plantillas se cuenta con la siguiente estructura:

```
src
└─ solver
   └─ templates
      ├─ ProdefProblemInstance.cpp.handlebars
      └─ ProdefProblemInstance.h.handlebars
```

5.4. Resolutor

Al igual que el módulo anterior se crea el núcleo del resolutor en `MetcoSolver.ts` en la raíz de `src/solver`, el cual va a contener la llamada al compilador C++, la creación de la plantilla y la ejecución del problema.

Se define la clase `MetcoSolver` como extensión de la clase abstracta `Solver`, que es un resolutor que implementa los métodos comunes de los distintos resolutores.

```
1 export default class MetcoSolver extends Solver {
2
3 }
```

5.4.1. Constructor

La primera función que se define es el constructor, donde se especifica tanto la configuración del resolutor como las funciones auxiliares que se verán en el siguiente apartado.

```
1 constructor() {
2     super({
3       supportsMultiplePermutations: false,
4       supportsMultipleVariableTypes: false,
5     });
6     this.registerHandlebarsHelpers();
7 }
```

La configuración tiene los siguientes atributos:

- **supportsMultiplePermutations:** Indica si se permiten multiples permutaciones.
- **supportsMultipleVariableTypes:** Indica si las permutaciones pueden ser de distinto tipo.

En el caso del resolutor de *METCO* ambos son falsos, ya que solo se resolverán problemas con un vector de permutación y C++ no permite vectores con valores de distinto tipo como otros lenguajes como Python o Ruby.

5.4.2. Funciones auxiliares

Handlebars también permite utilizar funciones auxiliares dentro de las plantillas, las cuales se añaden en el constructor con el método `registerHandlebarsHelpers`.

```
registerHandlebarsHelpers(): void {
    this.registerUpdateVariableHelper();
    this.registerIsMinimizeHelper();
    this.registerIsBinaryHelper();
    this.registerIsIntegerPermutationHelper();
    this.registerGetStaticDeclaration();
    this.registerGetStaticAssignment();
}
```

Se añaden distintas funciones con la idea de quitar complejidad de cálculo en las plantillas, ya que se adaptan los valores del problema compilado a las necesidades del compilador utilizado, ya sea C++, Java u otro lenguaje.

```
// Codigo en el resolutor
private registerIsBinaryHelper(): void {
    Handlebars.registerHelper('is_binary', (context: ProblemType) => {
        return context === 'binary';
    });
}
```

```
// Codigo en la plantilla
{{#with compiledProblem.metadata}}
  bool isBinary = {{{is_binary problemType}}};
{{/with}}
```

Por ejemplo, una de las funciones auxiliares es `registerIsBinaryHelper` que utiliza un atributo del problema compilado (veáse la Figura 4.2) para revisar si es un problema de tipo binario (como lo puede ser el problema *Knapsack*). Posteriormente, ese valor es asignado en la variable `isBinary` (de tipo `boolean`) evitando hacer ese cálculo en la plantilla.

5.4.3. Solve

Al crear un nuevo resolutor se debe sobrescribir el método `solve`, encargado de resolver el problema desde su compilación hasta obtener el resultado final. A continuación se puede ver el código resumido del método `solve` implementado.

```
async solve(problem: Problem): Promise<SolverOutput> {
    try {
      const problemPaths = await this.createProblem(problem);
      await this.moveToMetco(problemPaths);

      await this.compileMetco();
      await this.executeMetco();

      const result = this.getMetcoSolution()

      return this.createProdefSolution(result)
    } catch (error) {
      return this.createProdefError(error)
    }
}
```

Este método recibe el problema definido en el lenguaje `prodef-lang` (presentado en el Capítulo 3) generado desde la interfaz de usuario con el módulo `prodef-gui`, dando comienzo al flujo de funcionamiento de la herramienta *Prodef* (véase la Figura 5.2). A partir

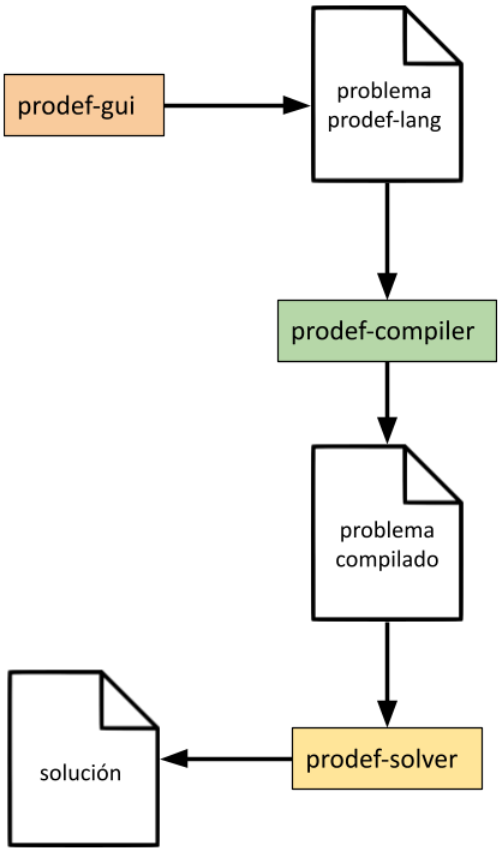


Figura 5.2: Funcionamiento de Prodef

de esa definición del problema, se crea el problema compilado (Figura 4.2) con la implementación específica de `prodef-compiler` con el módulo `prodef-compiler-cpp` para posteriormente ejecutarlo con la implementación particular del resolutor `prodef-solver` con *Metco* y el módulo `prodef-solver-metco`.

5.5. Variables de entorno

Se define un conjunto de parámetros que son utilizados para la ejecución del problema en *METCO* como ya se explicó en el Capítulo 2, así como la configuración de rutas y el puerto utilizado. Para ello, se definen las variables de entorno en el fichero `environment/environment.ts` con la idea de tener toda la información centralizada en un solo lugar.

```
export const environment = {
  ...
};
```

Se decidió utilizar el formato de un objeto `TypeScript` frente a otros como `yaml`, `config`, `json`, entre otros, por tres razones:

- Tipado de las variables
- Comentarios

- Integración con el resolutor en TypeScript

Una vez finalizado tanto el resolutor como las variables de entorno, el directorio del modulo debería tener la siguiente apariencia:

```
prodef-solver-metco
├── environment
│   └── environment.ts
├── src
│   └── solver
│       ├── templates
│       │   ├── ProdefProblemInstance.cpp.handlebars
│       │   └── ProdefProblemInstance.h.handlebars
│       └── MetcoSolver.ts
└── ...
```

5.6. Iniciar API

En este punto ya se ha terminado el módulo, pero se requiere de un lugar que invoque el resolutor e inicie la *API* en el puerto indicado. Por lo tanto, se creó el fichero `index.ts` en la raíz del directorio `src`.

```
import { SolverServer, MemoryStorage } from '@ull-prodef/prodef-solver';
import MetcoSolver from './solver/MetcoSolver';

import { environment } from '../environment/environment';

const solver = new MetcoSolver();
const storage = new MemoryStorage();

const server = new SolverServer(solver, storage, {
  enableLogger: environment.serverConfiguration.enableLogger,
  port: environment.serverConfiguration.port,
  maximumParallelExecutions: environment.serverConfiguration.maximumParallelExecutions,
});

server.start();
```

Es necesario llamar a la clase `SolverServer` con la información necesaria como el resolutor y el puerto para arrancar la *API* e interactuar con este módulo, ya sea desde *Postman* o la interfaz gráfica del módulo *prodef-gui*.

```
src
├── functions
│   ├── CosFunctionCompiler.ts
│   ├── SinFunctionCompiler.ts
│   └── SqrtFunctionCompiler.ts
├── CppCompiler.ts
└── index.ts
```

5.7. Flujo de trabajo

A continuación se explican los pasos que realiza el resolutor para resolver un problema:

1. Una vez que se inicia la *API* en el paso anterior, la aplicación está a la escucha de la siguiente *url* `http://localhost:8046/v1/executions`, la cual recibe el Problema 4.1 como un objeto *json*, a partir del cual genera el problema compilado (veáse la Figura 4.2) llamando al módulo *prodef-compiler-cpp*.

2. Posteriormente se crean los problemas en C++ usando ambas plantillas, tanto la definición como la implementación con el problema compilado y se hace uso de las funciones auxiliares de *handlebars*. Estos problemas se almacenan en una "*base de datos*" local, y para ello se utiliza un identificador único universal uuid para diferenciar cada problema, el cual es un número de 128 bits aleatorio generado en cada ejecución y que se almacenan en el directorio `executions`.

```
~/prodef-solver-metco$ npm i uuid
~/prodef-solver-metco$ npm i @types/uuid --save-dev
```

3. Se hace una copia del problema almacenado en el directorio `executions` con el identificador único en la ruta `oplink/algorithms/team/src/plugins/problems/ProdefProblemI` dentro de *METCO*

4. Se realiza el proceso de compilación y ejecución del problema de *METCO* en la nueva ruta. Tanto el anterior como este paso utilizan las variables de entorno con las rutas de las herramientas usadas. Es importante que las rutas dentro de las variables de entorno se configuren antes de iniciar el servidor.

5. Se retorna la respuesta en caso de éxito o el error en caso contrario. Para acceder a la respuesta del resolutor se debe realizar la petición *GET* a la ruta `http://localhost:8046/v1/executions/<identificador>`. Para más detalles, se puede consultar el Capítulo 3.

Capítulo 6

Experimentación

A lo largo de este capítulo se muestran los resultados obtenidos con distintos problemas y variantes tanto del problema *Knapsack* como *TSP*. Además, para mostrar los resultados de una manera visual, se hizo una colaboración con otro Trabajo de Fin de Grado que lleva como título "*Interfaz gráfica para el análisis de meta-heurísticas y para la visualización de soluciones a problemas de optimización combinatoria*" de Carolina Candelaria Álvarez Martín. Tras la realización de múltiples ejecuciones de las diferentes variantes de los problemas, se utilizó esta interfaz gráfica para generar las figuras que se incluyen en este capítulo.

6.1. Módulos

Para el análisis de los distintos problemas se desarrollaron dos módulos dentro de la organización *Prodef*: `prodef-problem-formatter` para la transformación de los datos de entrada para distintos problemas y variantes, y `prodef-evaluator` para la automatización de las ejecuciones de múltiples problemas, variantes e instancias. Estos módulos se describen en las secciones siguientes.

6.1.1. prodef-problem-formatter

Es un módulo que permite obtener los datos de entrada para distintos problemas y variantes con un formato específico. Estos datos contienen toda la información necesaria para la creación de la instancia como puede ser las funciones objetivo, restricciones, variables, entre otros. El módulo se encuentra alojado en este repositorio de la organización, que cuenta con un README con la explicación necesaria para su uso.

El módulo `prodef-problem-formatter` a partir de un fichero con la especificación de una instancia, genera los datos de entrada de una variante. Por lo general, el flujo de trabajo del módulo consiste en generar un fichero con el formato de datos *text*, a partir de los ficheros con las instancias de pruebas (véase la Tabla 6.1 y 6.3). El formato *text* es utilizado por las variantes definidas en *METCO* directamente. Luego se parte del formato *text* para crear los ficheros con formato *json*, utilizados por *Prodef* para generar el problema en `METCO`. El flujo de trabajo, varia según las instancias de prueba de cada problema:

- **Knapsack:** Las instancias de este problema ya vienen con un formato ideal para *METCO* y no se requiere generar el formato *text*. Las instancias de prueba presenta la lista las objetos que se pueden colocar en la mochila junto a su peso y beneficio, tal y como se observó en el fichero de la Figura 2.1.
- **TSP:** Al contrario que el anterior, estas instancias de prueba no presentan el formato necesario, por lo cual se tienen que generar ambos formatos: *text* y *json*. Todas variantes definidas utilizan la matriz de distancias entre las ciudades, en cambio las instancias de prueba contiene coordenadas de las ciudades en vez de la matriz. Por lo tanto, se tiene que hacer un calculo a priori de las matrices con diferentes funciones de distancias según la instancia, pero ¿Por qué no usar las coordenadas y evitar este calculo? Principalmente es por optimización, ya que este problema va a necesitar constantemente la distancia entre las ciudades y repetirá este cálculo.

Es importante mencionar que el último fichero con formato *json* es la entrada al módulo `prodef-compiler-cpp`, por lo cual debe tener la misma estructura que el problema de la Figura 4.1 con variaciones en los valores de las propiedades de este *json* según la instancia y tipo de problema.

Se utiliza el *script* para la ejecución este módulo. Este *script* se utiliza con comando que recibe el nombre del problema (*Knapsack* o *TSP*), el formato de los datos del fichero de salida (*text* o *json*) y la ruta absoluta o relativa de los datos del fichero de entrada. A partir del fichero de entrada se crea el nuevo fichero.

```
# Comando generico
~/prodef-problem-formatter$ npm run start -- -p <Problema> -f <Formato> -i <Fichero>

# Comando para un formato especifico
~/prodef-problem-formatter$ npm run start -- -p Knapsack -f text -i ../input.origin
```

6.1.2. prodef-evaluator

Es un módulo que permite realizar múltiples ejecuciones de diferentes variantes en la herramienta *METCO*. Este módulo se encuentra alojado en este repositorio de la organización, que al igual que el anterior módulo tiene un README con la información del mismo.

```
~/prodef-evaluator$ npm run start -- --run -p <Problema> -v [Variantes]
```

El comando anterior permite iniciar *prodef-evaluator*, donde se tiene que especificar un problema como puede ser el *Knapsack* o *TSP* y un conjunto de variantes, que indican si se utiliza la versión auto generada desde *Prodef*, la que está escrita directamente en *METCO* o ambas.

```
~/prodef-evaluator$ npm run start -- --run -p Knapsack -v Knapsack ProdefProblemInstance
```

Este módulo presenta ficheros de configuración para cada variante, los cuales obtienen la información de variables de entorno.

```
export const knapsackConfig: Config = {
  // Algorithm Configuration
  algorithm: {
    name: AlgorithmName.monoGa,
```

```
    parameters: [
      {
        name: AlgorithmParameterName.sizePopulation,
        value: environment.config.algorithm.sizePopulation,
      },
      {
        name: AlgorithmParameterName.mutationProbability,
        value: environment.config.algorithm.mutationProbability,
      },
      {
        name: AlgorithmParameterName.crossoverProbability,
        value: environment.config.algorithm.crossoverProbability,
      },
      {
        name: AlgorithmParameterName.survivalMethod,
        value: environment.config.algorithm.survivalMethod,
      },
    ],
  },

  // Problem Configuration
  problem: {
    name: ProblemName.knapsack,
    objectiveName: ProblemObjectName.profit,
    variant: ProblemVariantName.knapsack,
    categories: [
      {
        name: ProblemCategoryName.size20,
        numberOfInstances: environment.config.problem.numberOfInstances,
      },
      {
        name: ProblemCategoryName.size50,
        numberOfInstances: environment.config.problem.numberOfInstances,
      },
      {
        name: ProblemCategoryName.size100,
        numberOfInstances: environment.config.problem.numberOfInstances,
      },
    ],
    parameters: [
      {
        name: ProblemParameterName.inputFilePath,
      },
    ],
  },

  // Execution Configuration
  execution: {
    numberOfExecutions: environment.config.execution.numberOfExecutions,
    printPeriod: environment.config.execution.printPeriod,
    stopCriteria: {
      name: StopCriteriaName.evaluations,
      value: environment.config.execution.stopCriteria,
    },
  },

  // Execution Preferences Configuration
  executionPreferences: {
```

```
    build: false,
    extension: {
      input: ExtensionName.text,
      output: ExtensionName.json,
    },
  },
};
```

El anterior fichero es la configuración de una variante del problema *Knapsack*, la cual obtiene todos sus valores como contantes de las variables de entorno y presenta las especificaciones siguientes:

- **Algoritmo:** Nombre y parámetros del algoritmo.
- **Problema:** Nombre, variante y parámetros del problema, junto a la descripción del objetivo y la categoría de la instancia, que se refiere a su tamaño ya sea 20, 50, 100, entre otras categorías.
- **Ejecución:** Número de ejecuciones, criterio de parada (tiempo o número de evaluaciones) y `printPeriod`. Este último parámetro define indirectamente el número de puntos en la ejecución del problema, donde se puede ver el estado actual del problema y su comportamiento a lo largo de la ejecución.
- **Preferencias de ejecución:** Indica el tipo de entrada y salida del fichero que contiene la instancia, además tiene una bandera `build` para indicar si la instancia tiene que ser construida desde *Prodef*.

soluciones para diferentes puntos de la ejecución. No lo has dicho donde explicaste los parámetros de la ejecuión (el printPeriod por ejemplo)

Algoritmo 1 Proceso de prodef-evaluator

```
for each actualVariante ∈ variantes do
    for each actualInstancia ∈ instancias do
        for iteracion= 1,2,..., numeroDeEjecuciones do
            resolverInstancia(actualInstancia)
        end for
    end for
end for
```

Al ejecutar este módulo se genera un ciclo por cada variante e instancia, ya que al seleccionar una variante se ejecuta cada una de sus instancias un determinado número de veces antes de cambiar de variante, como se aprecia en el Algoritmo 1.

Una vez finalizada este proceso, se genera un conjunto de carpetas y ficheros con las soluciones encontradas. Por ejemplo, para el problema *Knapsack* con 2 variantes, 3 tipos de categorías o tamaño de instancia (20, 50 y 100) y 5 ejecuciones por cada instancia, se genera la siguiente estructura:

```
Knpasack
├── Variante-1
│   ├── size-20
│   ├── size-50
│   ├── size-100
│   └── local-config.json
├── Variante-2
│   ├── size-20
│   ├── size-50
│   ├── size-100
│   └── local-config.json
└── global-config.json
```

Dentro de la categoría se encuentra un fichero por cada ejecución y uno con el resumen de los resultados, que contienen las medias de los valores alcanzados como la función objetivo y tiempo.

```
size-20
├── execution-1.json
├── execution-2.json
├── execution-3.json
├── execution-4.json
├── execution-5.json
└── results.json
```

6.2. Configuración

Para la obtención de resultados se realizó un estudio previo para obtener los mejores parámetros y se tomaron en cuenta los siguientes aspectos:

- Las variantes que permiten violar las restricciones y las auto generadas desde *Prodef* pueden partir de soluciones no factibles y a lo largo del algoritmo van mejorando las soluciones hasta lograr la factibilidad. Por lo tanto, se optaron por parámetros que permitan a estas variantes converger con soluciones factibles.
- Las instancias de mayor tamaño por lo general requieren más tiempo para resolverse y en este caso están alrededor del tamaño 100. Por ello, se deben seleccionar no solo parámetros que permitan resolver instancias de distintos tamaños, sino también que permitan a las instancias de mayor tamaño obtener "*buenas*" soluciones.
- El tamaño de la población debe estar en equilibrio, ya que debe ser lo suficientemente grande para permitir realizar búsquedas en un espacio más grande de soluciones, pero sin superar un límite a partir del cual resulta ineficiente incrementar su tamaño puesto que se obtienen soluciones repetidas y el algoritmo será más lento.
- La probabilidad de cruce indica la frecuencia con que se producen cruces entre padres. Por ejemplo, si la probabilidad de cruce es 0, los hijos serán copias exactas de los padres, mientras que si es 100% el hijo se creará totalmente por cruce.

- La probabilidad de mutación indica la frecuencia con que cada individuo es mutado. Por ejemplo, si la mutación es de 100% el individuo cambia totalmente y en consecuencia la población degenera rápidamente.

Se utilizaron los mismos valores en cada una de las especificaciones para todas las instancias, indistintamente de su tamaño, los cuales se verán a continuación.

6.2.1. Algoritmo

Se utilizó un algoritmo evolutivo mono-objetivo con los siguientes parámetros:

- **Tamaño de la población**: 80
- **Probabilidad de mutación**: 0.1
- **Probabilidad de cruce**: 0.9
- **Método de selección**: El mejor conjunto entre padres e hijos. Por ejemplo, a partir de una generación de 80 individuos se genera un población hija de 80 individuos, teniendo en total 160 individuos; los cuales son clasificados de mejor a peor, sin distinguir quienes son padres o hijos, y se selecciona los 80 mejores individuos para continuar con el algoritmo, mientras que los 80 peores son descartados.

6.2.2. Ejecución

Para la ejecución de cada variante se utilizaron los siguientes parámetros:

- **Número de ejecuciones**: 30
- **Criterio de parada**: 80000 evaluaciones
- **PrintPeriod**: 1600

Las evaluaciones se refiere al número de individuos que se evalúan y es directamente proporcional al número de generaciones. Por ejemplo, si se tiene una población de 100 individuos y un número de evaluaciones de 1000, significa que habrán 10 generaciones.

```
Evaluaciones = Tamaño Población * Generaciones
```

Por otro lado, las ejecuciones indican el número de veces que se ejecuta una instancia, con el objetivo de obtener los valores medios de las soluciones y tiempos, junto a las desviaciones típicas.

Otro parámetro de la ejecución es el `printPeriod`, a partir del cual se calcula el número de puntos durante la ejecución del problema. Para un número de evaluaciones de 80000 y un `printPeriod` de 1600, habrán 50 puntos durante la ejecución.

```
Número de puntos = Evaluaciones / printPeriod
```

6.2.3. Componentes del ordenador

El ordenador utilizado para ejecutar los algoritmos cuenta con:

- Sistema operativo Debian(Linux) de 64 bits, ejecutada desde Windows con WSL2.
- Procesador AMD Ryzen 5 3600 6-Core.
- Memoria RAM de 8 GB.

6.3. Resultados

A lo largo de esta sección se puede observar los resultados obtenidos en cada uno de los problemas y sus variantes, entre las cuales están:

- **Metco sin violación de restricciones** (*Metco:Sin-Vio*): Es una variante que ha sido escrita directamente en código C++ y solo permite generar soluciones factibles, ya sea porque cumplen con las restricciones del problema o no tiene restricciones.
- **Metco con violación de restricciones** (*Metco:Con-Vio*): Es igualmente una variante escrita directamente en código C++, pero a diferencia de la anterior, esta sí permite soluciones no factibles, que violan en cierto grado las restricciones del problema.
- **Metco generado desde Prodef** (*Metco:Prodef*): Es una variante de código C++ auto-generado desde *Prodef*. Esta versión permite soluciones que violan las restricciones del problema.

Inicialmente solo existía la primera *Metco:Sin-Vio* y la última *Metco:Prodef* variante, pero al ser distintas maneras de resolver el problema, se desarrolló la alternativa *Metco:Con-Vio* para que exista un punto intermedio de comparación. Esta nueva variante *Metco:Con-Vio* combina las otras dos variantes, *Metco:Sin-Vio* porque está escrita directamente en C++ y *Metco:Prodef* porque permite la violación de restricciones.

6.3.1. Nomenclatura

En las tablas que se verán más adelante en este capítulo se utiliza la siguiente nomenclatura:

- **Instancias**: Representa los nombres de las instancias. En caso de instancias con el mismo tamaño se utiliza el formato `N-I`, donde `N` es el tamaño de la instancia y el símbolo `I` es el índice de la instancia.
- **N**: Tamaño de la instancia.
- **Fuentes**: Indica los nombres de los ficheros desde donde se obtuvieron las instancias.

- **Óptimos**: Representa la solución óptima conocida de la instancia.
- Luego, para cada par variante-instancia se presentan distintos valores, que son calculados a partir de las múltiples ejecuciones realizadas para cada configuración:
  - **Soluciones**: Es la media de los objetivos conseguidos (valor de fitness).
  - **Desv**: Desviación típica con respecto a los objetivos.
  - **Tiempos**: Media de tiempo en segundos utilizado por el ordenador para resolver una ejecución.

6.3.2. Knapsack

En esta sección se encuentran los resultados finales del problema *Knapsack*. En la Tabla 6.1 se relaciona el conjunto de instancias utilizadas para analizar las variantes del problema. Los archivos fuente de las instancias se encuentran en un repositorio de GitHub [10].

Instancias	N	Fuentes
20-1	20	`knapPI_13_20_1000.csv`
20-2	20	`knapPI_13_20_1000.csv`
20-3	20	`knapPI_13_20_1000.csv`
50-1	50	`knapPI_13_50_1000.csv`
50-2	50	`knapPI_13_50_1000.csv`
50-3	50	`knapPI_13_50_1000.csv`
100-1	100	`knapPI_13_100_1000.csv`
100-2	100	`knapPI_13_100_1000.csv`
100-3	100	`knapPI_13_100_1000.csv`

Tabla 6.1: Instancias del problema Knapsack

A continuación, en la Tabla 6.2 se presentan los resultados de las ejecuciones realizadas para cada configuración (variante-instancia).

		Metco:Sin-Vio			Metco:Con-Vio			Metco:Prodef		
Instancias	Óptimos	Soluciones	Desv	Tiempos	Soluciones	Desv	Tiempos	Soluciones	Desv	Tiempos
20-1	1716	**1716**	0	0.1	**1716**	0	0.24	**1716**	0	0.82
20-2	2501	**2501**	0	0.1	**2501**	0	0.24	**2501**	0	0.88
20-3	1200	**1200**	0	0.11	**1200**	0	0.24	**1200**	0	0.86
50-1	1989	**1989**	0	0.45	**1989**	0	0.54	**1989**	0	1.47
50-2	2806	**2806**	0	0.44	**2806**	0	0.54	**2806**	0	1.48
50-3	1200	**1200**	0	0.51	**1200**	0	0.55	**1200**	0	1.5
100-1	1989	**1989**	0	1.86	1942	58.2	1.17	1950	40.28	2.88
100-2	2806	**2806**	0	1.87	2800	18.3	1.17	2804	10.95	2.87
100-3	1760	**1736**	16.04	1.99	1730	25.66	1.17	1709	33.77	2.9

Tabla 6.2: Resultados del problema Knapsack

A primera vista en la Tabla 6.2 se puede hacer el siguiente análisis:

- Todas las variantes obtienen el óptimo conocido en pequeños y medianos problemas. Sin embargo, la variante *Metco:Sin-Vio* siempre obtiene las mejores soluciones de valor objetivo, sea cual sea el tamaño de la instancia. Esto demuestra la potencia de una búsqueda local en *Metco:Sin-Vio* para mejorar una solución.

- Con respecto a los tiempos, la variante *Metco:Sin-Vio* suele ser más rápida en pequeños y medianos problemas, mientras que la variante *Metco:Con-Vio* es más rápida EN grandes de problemas. Estas dos variantes tienen tiempos similares en los medianos problemas, significa que entre los medianos y grandes problemas se encuentra el punto en el que *Metco:Con-Vio* mejora en tiempo a *Metco:Sin-Vio*.
- *Metco:Prodef* es la versión más lenta como era de esperarse, debido a sus estatus de problema auto generado sin un diseño específico para el *Knapsack*.
- Las variantes que violan las restricciones (*Metco:Con-Vio* y *Metco:Prodef*) obtienen mayores valores de desviación típica en las instancias de mayor tamaño, que indica que se obtienen soluciones más dispersas. Este comportamiento posiblemente es porque se explora más el espacio de soluciones, ya que se tienen en cuenta tanto las soluciones factibles como no factibles.

Con el fin de hacer un análisis en mayor profundidad que el presentado anteriormente a partir de la Tabla 6.2, se ha seleccionado una instancia que represente a cada categoría o tamaño de problema: pequeños ≈ 20 (véase las Figuras 6.1 y 6.2), medianos ≈ 50 (véase las Figuras 6.3 y 6.4) y grandes ≈ 100 (véase las Figuras 6.5 y 6.6). A modo general, el comportamiento es similar entre las instancias del mismo tamaño.

A partir de estas categorías se representó de manera visual el comportamiento de estos representantes elegidos y para ello, se seleccionaron dos puntos a lo largo de la ejecución. Estos puntos son definidos por el atributo *PrintPeriod*, cuyo valor es 1600 y en consecuencia existen 50 puntos. Para cada categoría se tomó el primer (1) y último (50) punto, que representan 1600 y 80000 evaluaciones respectivamente.

Las gráficas de las Figuras 6.1, 6.2, 6.3, 6.4, 6.5 y 6.6 presentan una estructura similar. El eje de las abscisas representa el número de la ejecución, siendo un total de 30 ejecuciones. El eje de las ordenadas representa el valor del objetivo, que significa el beneficio de la mochila. Por lo tanto, estas gráficas representan 30 soluciones en un momento puntal de la ejecución para cada una de las variantes: *Metco:Sin-Vio*, *Metco:Con-Vio* y *Metco:Prodef*.

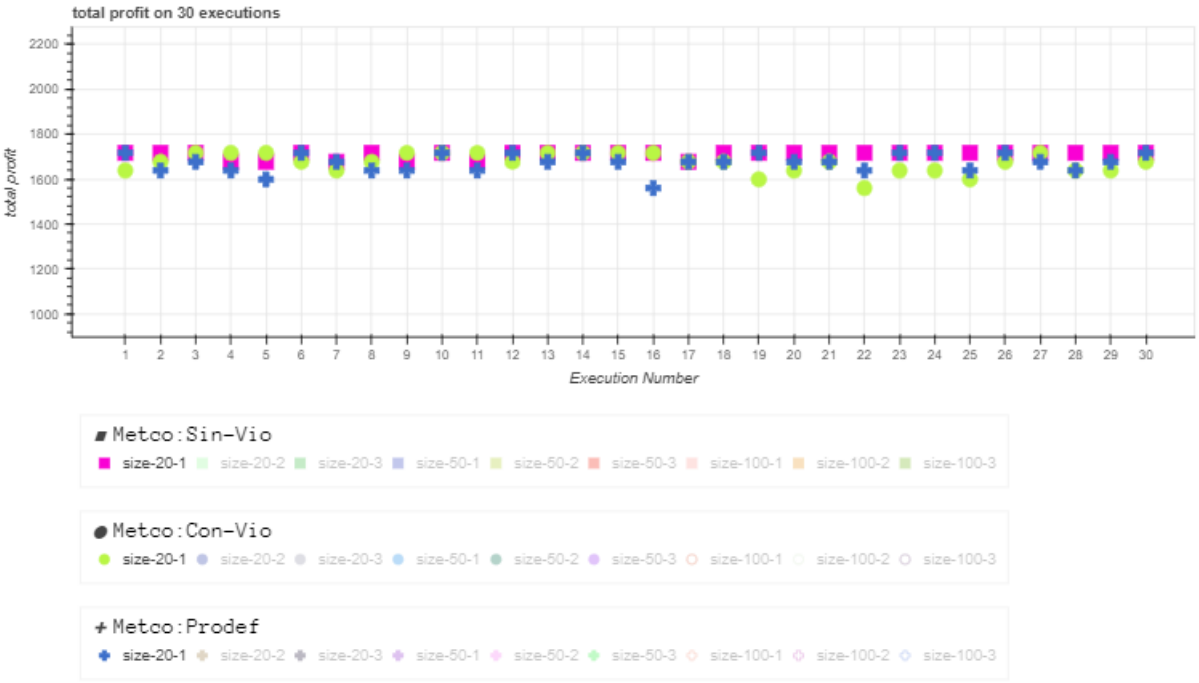


Figura 6.1: Instancia 20-1 al inicio de las ejecuciones con 1600 evaluaciones

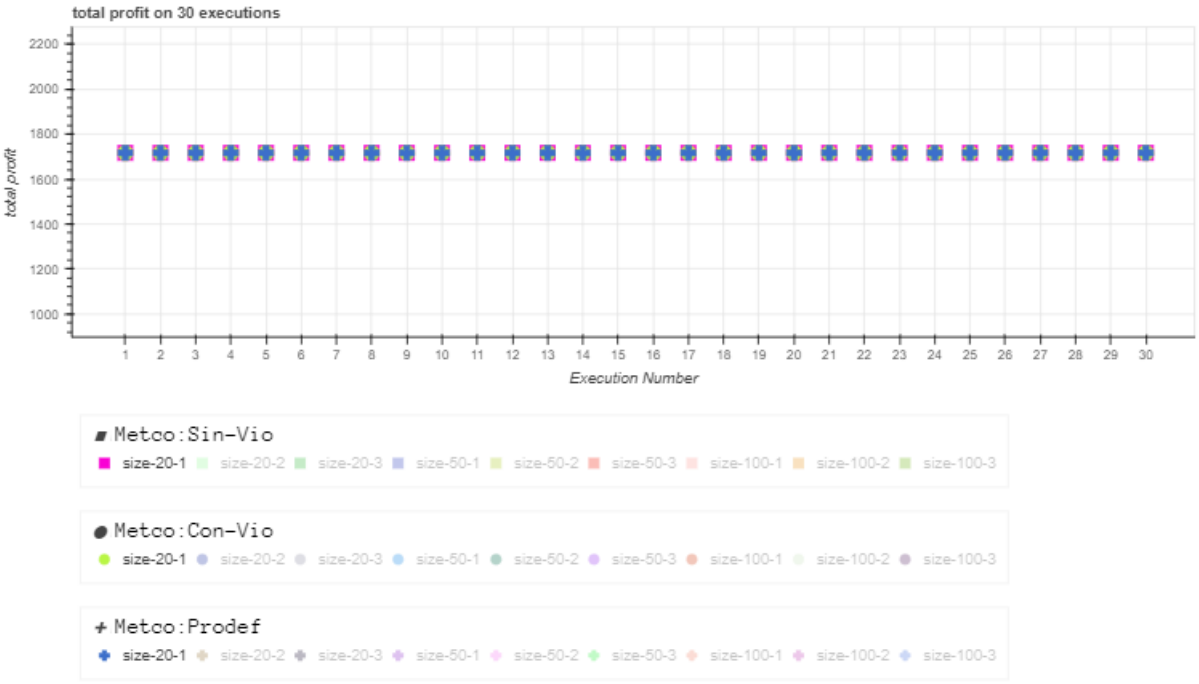


Figura 6.2: Instancia 20-1 al final de la ejecuciones con 80000 evaluaciones

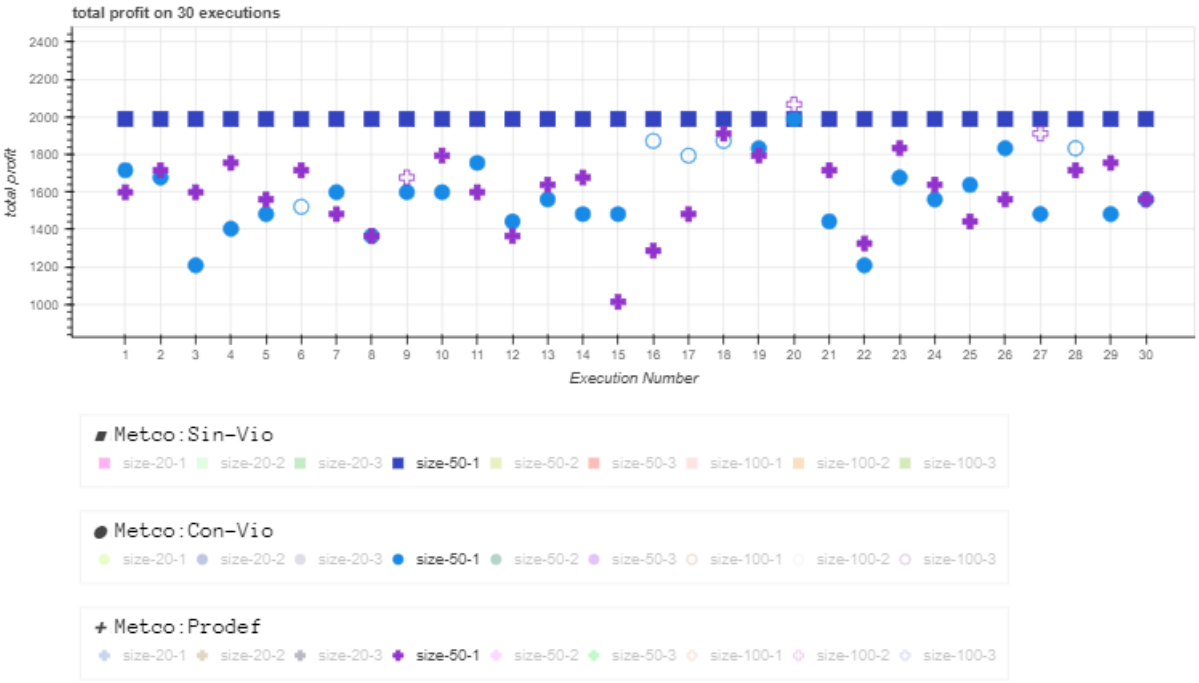


Figura 6.3: Instancia 50-1 al inicio de las ejecuciones con 1600 evaluaciones

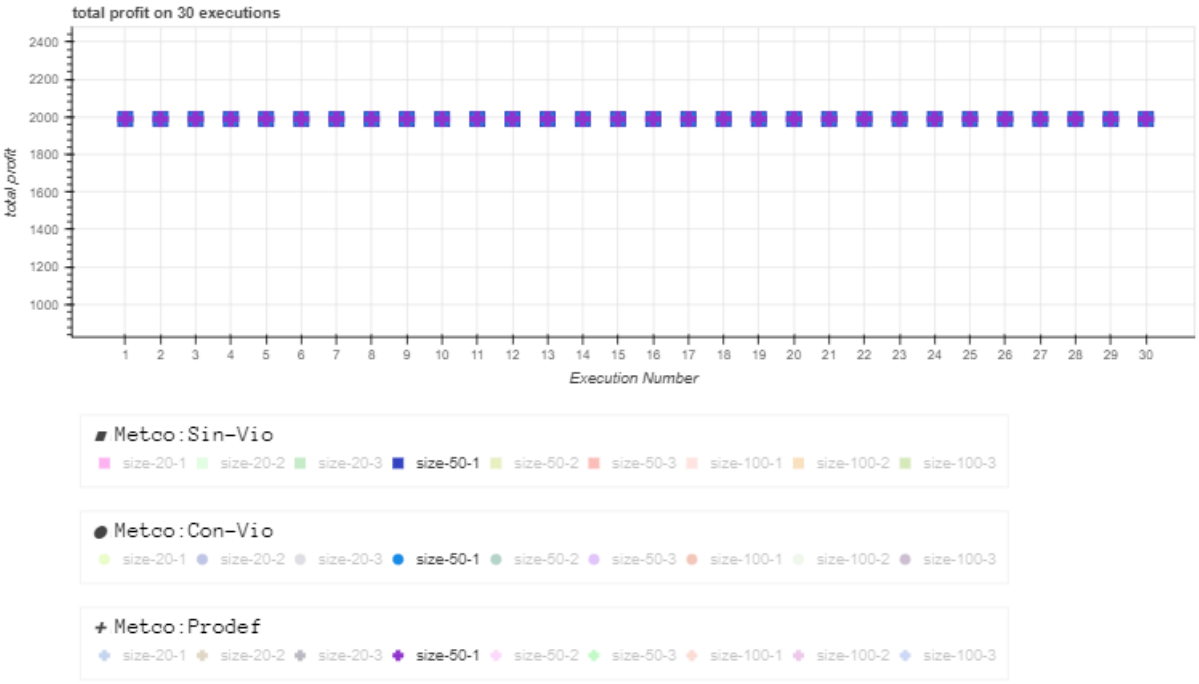


Figura 6.4: Instancia 50-1 al final de la ejecuciones con 80000 evaluaciones

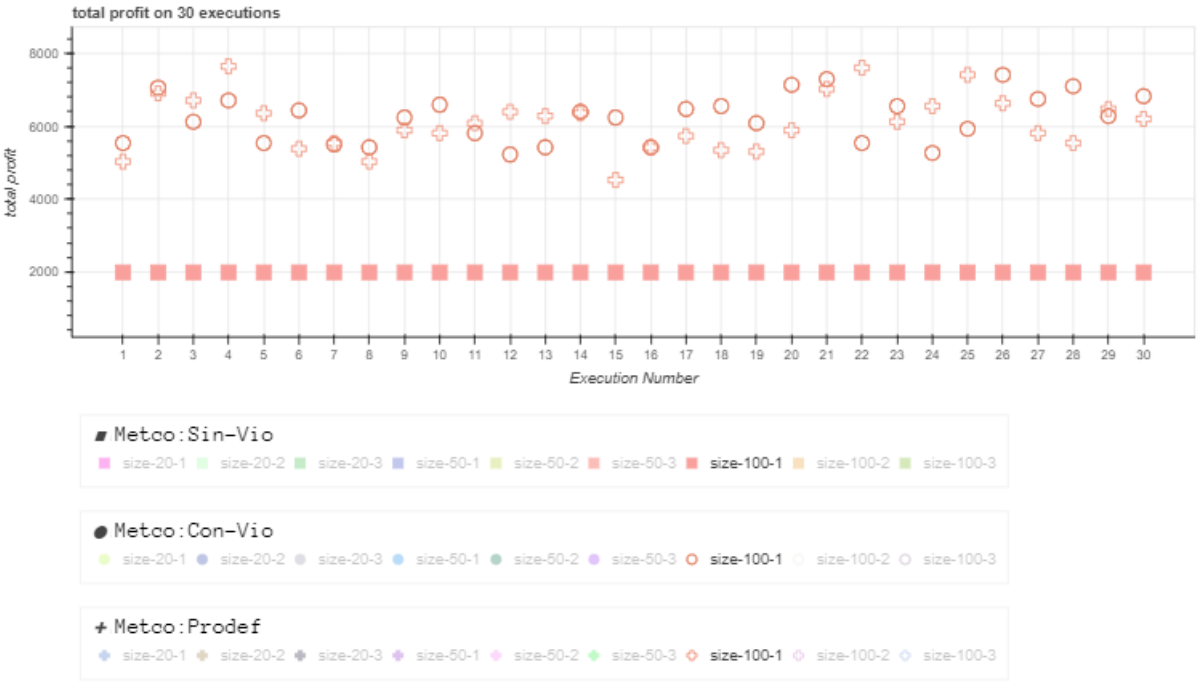


Figura 6.5: Instancia 100-1 al inicio de las ejecuciones con 1600 evaluaciones

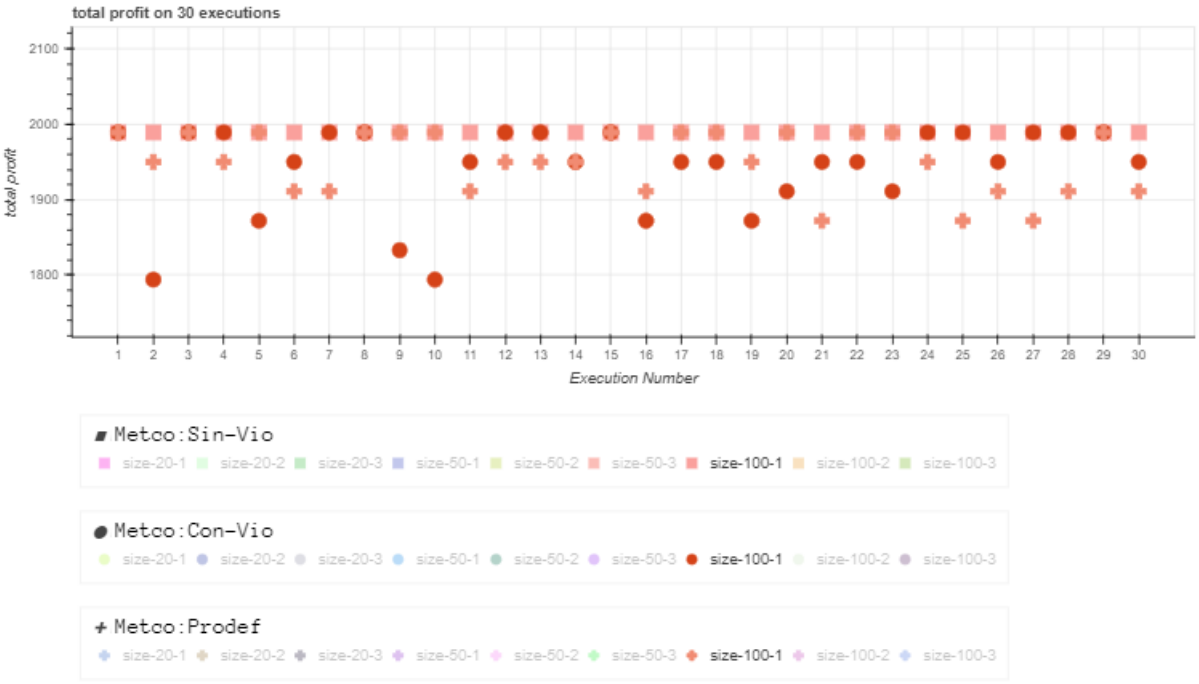


Figura 6.6: Instancia 100-1 al final de las ejecuciones con 80000 evaluaciones

En el diseño de las gráficas se puede observar los siguientes aspectos:

- Las leyendas de las gráficas contienen todas las instancias del problema, ya que están cargadas en la interfaz para elegir cualquier combinación de estas para su posterior estudio, pero las que se están comparando en un momento puntual

se pueden ver en un color más fuerte, mientras que el resto de instancias están desactivadas con un sombreado.

- Se utiliza el mismo símbolo relleno o no para diferenciar soluciones factibles o no respectivamente. Por ejemplo, en la Figura 6.5 que indica el inicio de la instancia de tamaño 100, se pueden ven todos los símbolos que pertenecen a variantes con violación de restricciones (*Metco:Con-Vio* y *Metco:Prodef*) sin relleno y que significa que todas soluciones son no factibles, posteriormente en Figura 6.6 ocurre lo contrario, debido a que las soluciones han logrado la factibilidad.

Con respecto al problema *Knapsack* se concluye lo siguiente:

- Las instancias de tamaño 20 y 50 sea la variante que sea generalmente alcanzan la solución óptima conocida como se ve en las Figuras 6.2 y 6.4. Esto se debe a que se usa un número de evaluaciones muy alto para instancias de este tamaño pero ideal para las de tamaño 100.
- Al comienzo de cualquier ejecución, la variante que utiliza una búsqueda local *Metco:Sin-Vio* obtiene valores muy cercanos al valor óptimo conocido como se ve en las Figuras 6.1, 6.3 y 6.5. Esto reafirma que un problema diseñado a medida va a obtener mejores soluciones.
- Las variantes que violan las restricciones con las instancias de tamaño 20 y 50 generalmente comienzan con soluciones factibles como se ve en las Figuras 6.1 y 6.3. Al igual que el punto anterior se utiliza un número de evaluaciones muy alto, el cual se divide para obtener 50 puntos a lo largo de la evolución, siendo el primer valor (1600 evaluaciones) todavía muy alto para ver ese proceso en el que las soluciones se vuelven factibles.
- Las variantes que admiten gestionar soluciones que no son factibles pueden comenzar con soluciones con "*mejores*" valores objetivo como se ve en la Figura 6.5, pero son soluciones no factibles y, por lo tanto, estos valores van empeorando hasta lograr la factibilidad y posteriormente mejorar como se ve en la Figura 6.6.

Para generar las gráficas de las Figuras 6.7, 6.8 y 6.9 se tomó como referencia la instancia de mayor tamaño, porque así se puede apreciar mejor la evolución del algoritmo. Estas gráficas presentan en el eje de las abscisas el número de evaluaciones, siendo un total de 80000 evaluaciones y el eje de las ordenadas el valor del objetivo. Por lo tanto, se representan la evolución de la instancia durante toda la ejecución para cada una de las variantes: *Metco:Sin-Vio*, *Metco:Con-Vio* y *Metco:Prodef*.

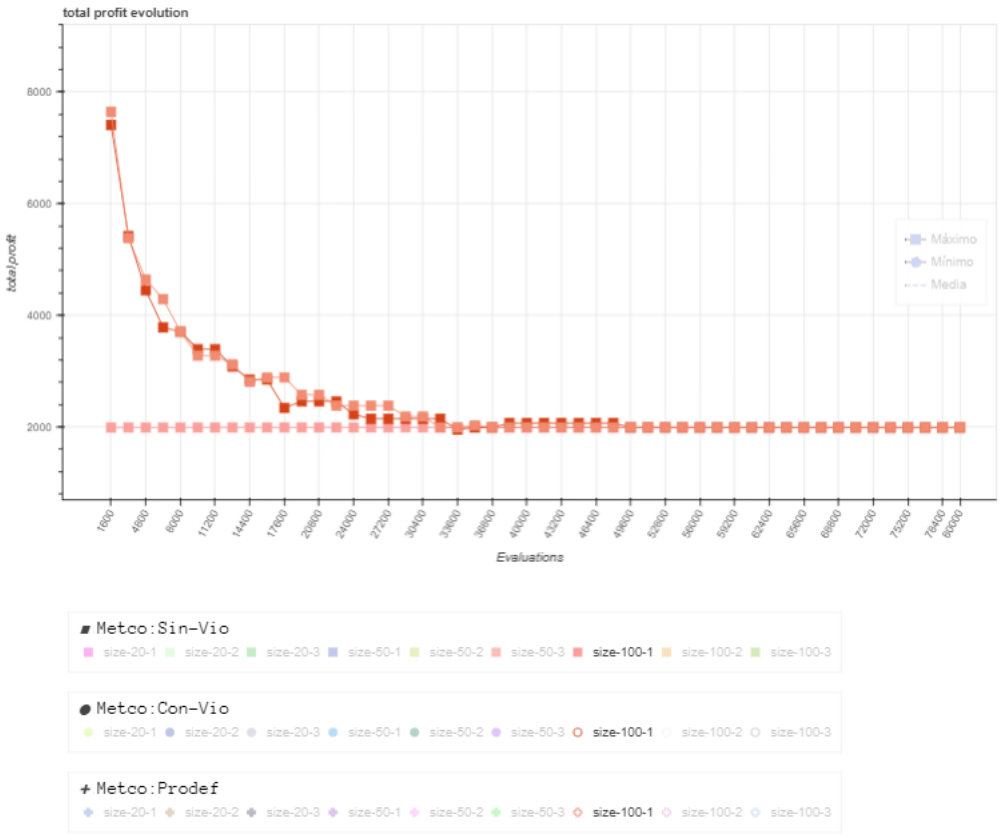


Figura 6.7: Valor máximo de la instancia 100-1 durante 80000 evaluaciones

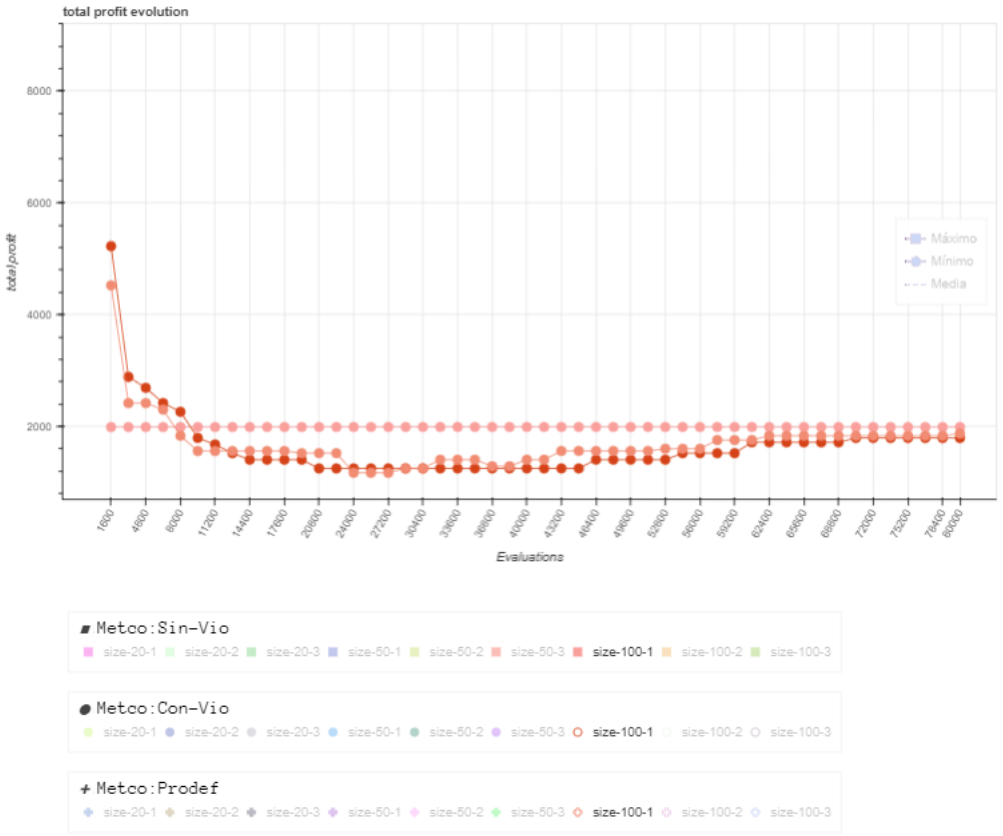


Figura 6.8: Valor mínimo de la instancia 100-1 durante 80000 evaluaciones

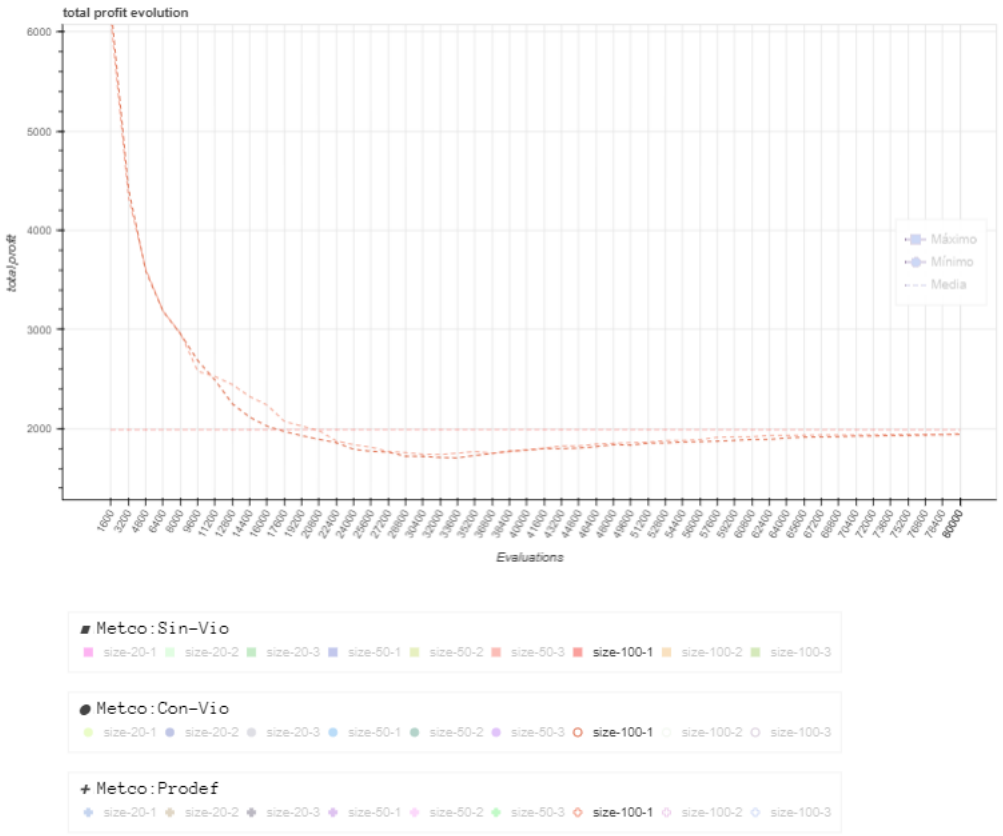


Figura 6.9: Media entre máximos y mínimos de la instancia 100-1 durante 80000 evaluaciones

Viendo las Figuras 6.7, 6.8 y 6.9, se puede concluir que la instancias sin violación de restricciones (*Metco:Sin-Vio*) consigue el óptimo conocido desde el comienzo, mientras que las otras variantes obtienen "*malos*" resultados al principio, pero a lo largo de la evolución van mejorando hasta conseguir la estabilidad y posteriormente el mejor valor conocido.

6.3.3. TSP

Al igual que el problema anterior, a continuación se encuentran los resultados finales del *TSP*, tanto los archivos fuente de cada instancia en la Tabla 6.3 que están alojadas en un repositorio académico [15] como los resultados de las ejecuciones en la Tabla 6.4. Sin embargo, en este problema solo se compara la versión *Metco:Sin-Vio* y *Prodef:Metco* sin utilizar *Metco:Con-Vio*. Esta simplificación en el número de variantes se debe a la no presencia de restricciones en este problema, porque se utiliza una codificación de permutación que hace que cualquier codificación represente a una solución factible.

Instancias	N	Fuentes
burma14	14	`burma14.tsp.gz`
ulysses16	16	`ulysses16.tsp.gz`
ulysses22	22	`ulysses22.tsp.gz`
swiss42	42	`swiss42.tsp.gz`
att48	48	`att48.tsp.gz`
berlin52	52	`berlin52.tsp.gz`
gr96	96	`gr96.tsp.gz`
rat99	99	`rat99.tsp.gz`
kroA100	100	`kroA100.tsp.gz`

Tabla 6.3: Instancias del problema TSP

		Metco:Sin-Vio			Metco:Prodef		
Instancias	Óptimos	Soluciones	Desv	Tiempos	Soluciones	Desv	Tiempos
burma14	3323	**3323**	2.33	0.03	3329	22.53	0.07
ulysses16	6859	6901	33.02	0.03	**6896**	30.49	0.08
ulysses22	7013	**7318**	298.12	0.03	7345	362.22	0.10
swiss42	1273	1573	105.66	0.04	**1552**	80.29	0.1
att48	10628	13592	1074.85	0.04	**13443**	884.26	0.1
berlin52	7542	**9749**	604.44	0.04	9766	501.08	0.1
gr96	55209	106638	6342.75	0.07	**106449**	6315.64	0.13
rat99	1211	2394	136.17	0.06	**2322**	131.46	0.14
kroA100	21282	46560	3162.76	0.06	**46103**	2780.52	0.14

Tabla 6.4: Resultados del problema TSP

A primera vista en la Tabla 6.4 se puede hacer el siguiente análisis:

- Existe un mayor equilibrio entre las distintas variantes en comparación al problema *Knapsack*, ya que el mejor valor no se centra en una sola variante. Esto se debe a que *Metco:Sin-Vio* no implementa una búsqueda local, diferenciándose de *Metco:Prodef* solamente en la manera en que se genera el problema: escrito directamente en C++ o auto generado desde Prodef.
- Aunque son diferencias mínimas, parece que la variante de *Metco:Prodef* obtiene finalmente mejores soluciones del valor objetivo con respecto a la variante *Metco:Sin-Vio*.
- Ambas variantes sea cual sea el tamaño del problema, tienen altos valores con respecto a las desviaciones típicas. Esto significa que se obtienen soluciones muy dispersas en el espacio de soluciones.
- La variante *Metco:Con-Vio* es más rápida que *Metco:Prodef* en todas las instancias. Esta última variante tiende a tardar el doble de tiempo con respecto a la primera. Nuevamente, esto se debe a que es un problema auto generado en comparación a una definición especifica para el *TSP*.
- Sin importar la variante, las soluciones tienen un comportamiento según el tamaño del problema. Mientras más grande es el problema, más se alejan las soluciones encontradas del mejor valor conocido. Esto es debido a la complejidad del *TSP*, ya que es un problema más difícil de resolver que el *Knapsack*.

¿Un número más alto de evaluaciones lograría mejores soluciones? La selección de parámetros para la resolución de problemas es un trabajo muy complejo, que depende de muchos factores como se mencionó en secciones anteriores. Por ello, se hicieron múltiples pruebas con distintos valores y aumentar el número de evaluaciones no siempre da como resultado mejores soluciones. En el caso del *TSP*, se utilizaron evaluaciones mucho más altas, donde se observó que el valor objetivo de las soluciones mejoraba mínimamente, mientras que el tiempo aumentaba significativamente. Por lo tanto, este problema llega a un punto que no vale la pena esa mejora con respecto a su coste.

Nuevamente, con el fin de hacer un análisis en mayor profundidad que el presentado a partir de la Tabla 6.4, se seleccionó una instancia que represente a cada categoría: pequeños ≈ 20 (véase las Figuras 6.10 y 6.11), medianos ≈ 50 (véase las Figuras 6.12 y 6.13) y grandes ≈ 100 (véase las Figuras 6.14 y 6.15).

Igualmente se seleccionaron dos puntos durante la ejecución, que representan 1600 y 80000 evaluaciones, respectivamente el primer (1) y último (50) punto.

Las gráficas de las Figuras 6.10, 6.11, 6.12, 6.13, 6.14 y 6.15 presentan una estructura similar. El eje de las abscisas representa el número de la ejecución, siendo un total de 30 ejecuciones. El eje de las ordenadas representa el valor del objetivo, que significa la distancia total recorrida a través del circuito de ciudades. Por lo tanto, estas gráficas representan 30 soluciones en un momento puntal de la ejecución para cada una de las variantes: *Metco:Sin-Vio* y *Metco:Prodef*.

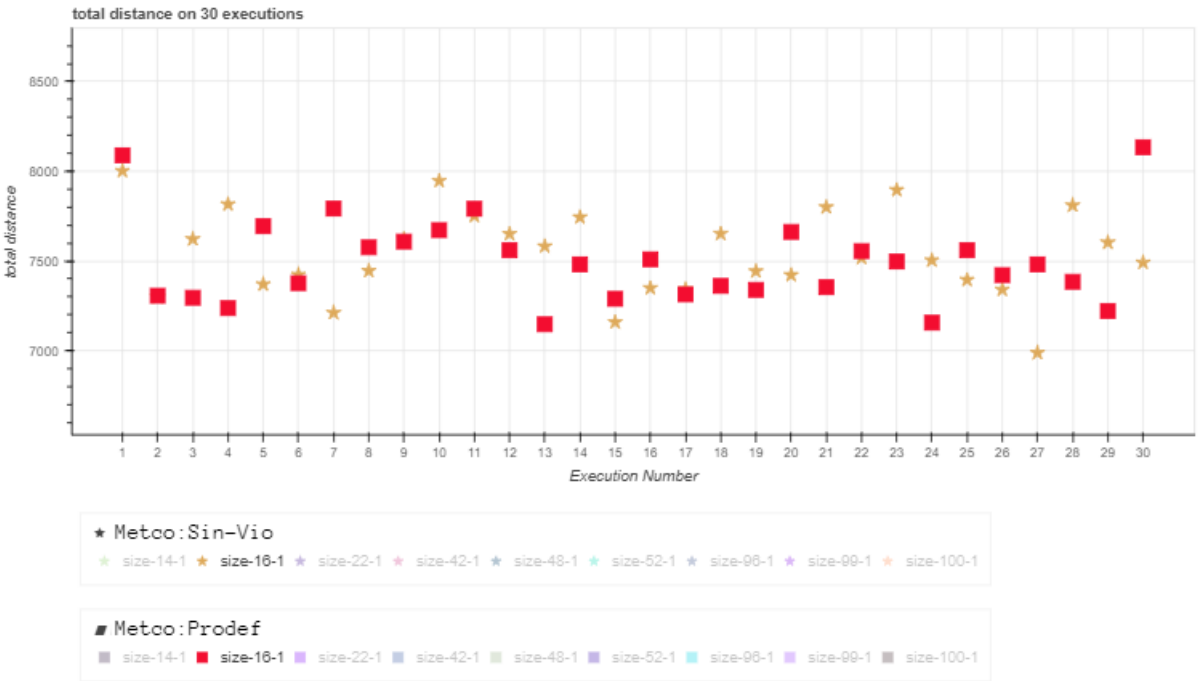


Figura 6.10: Instancia ulysses16 al inicio de las ejecuciones con 1600 evaluaciones

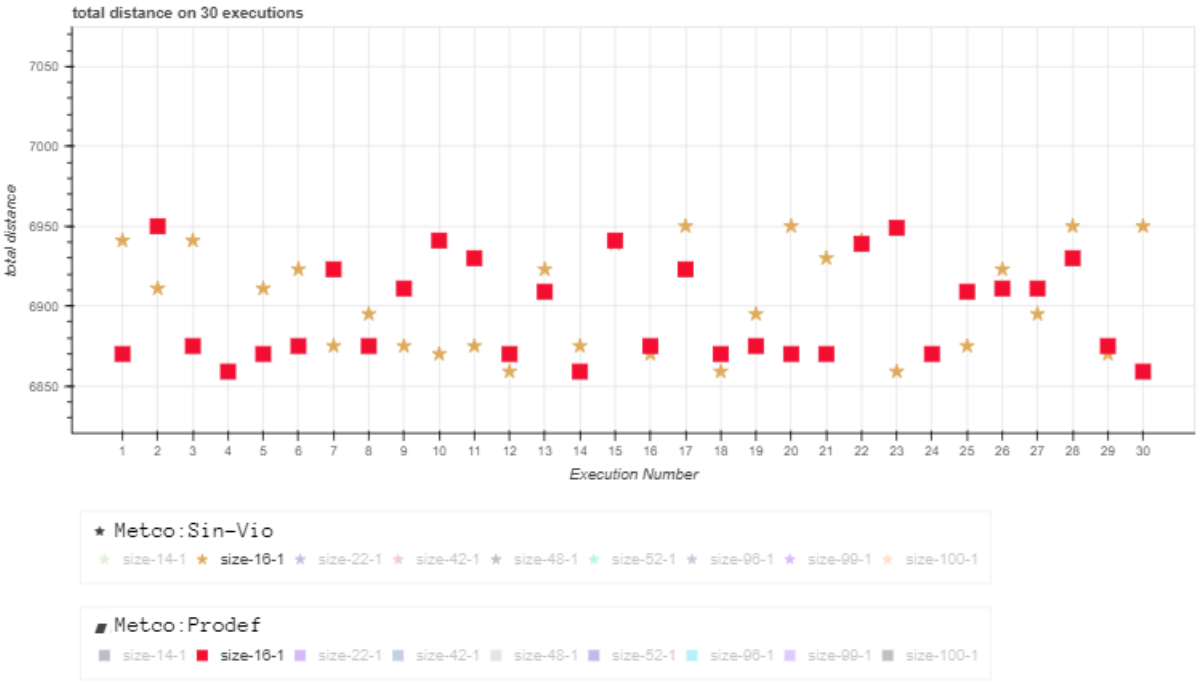


Figura 6.11: Instancia ulysses16 al final de la ejecuciones con 80000 evaluaciones

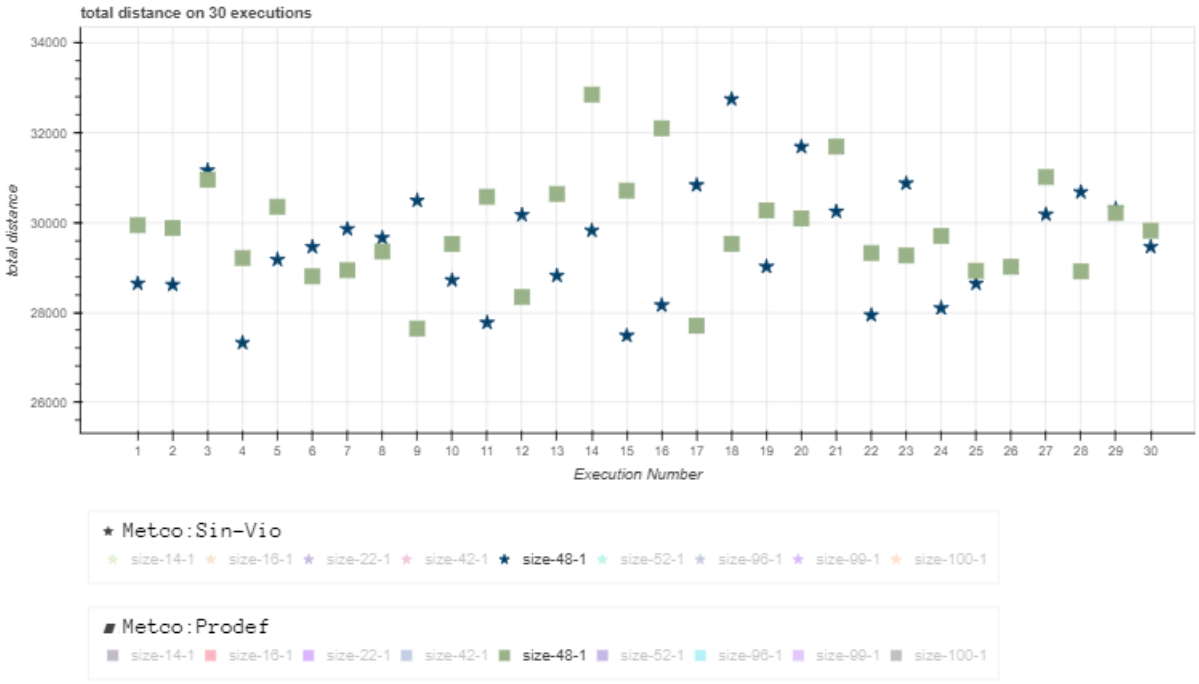


Figura 6.12: Instancia att48 al inicio de las ejecuciones con 1600 evaluaciones

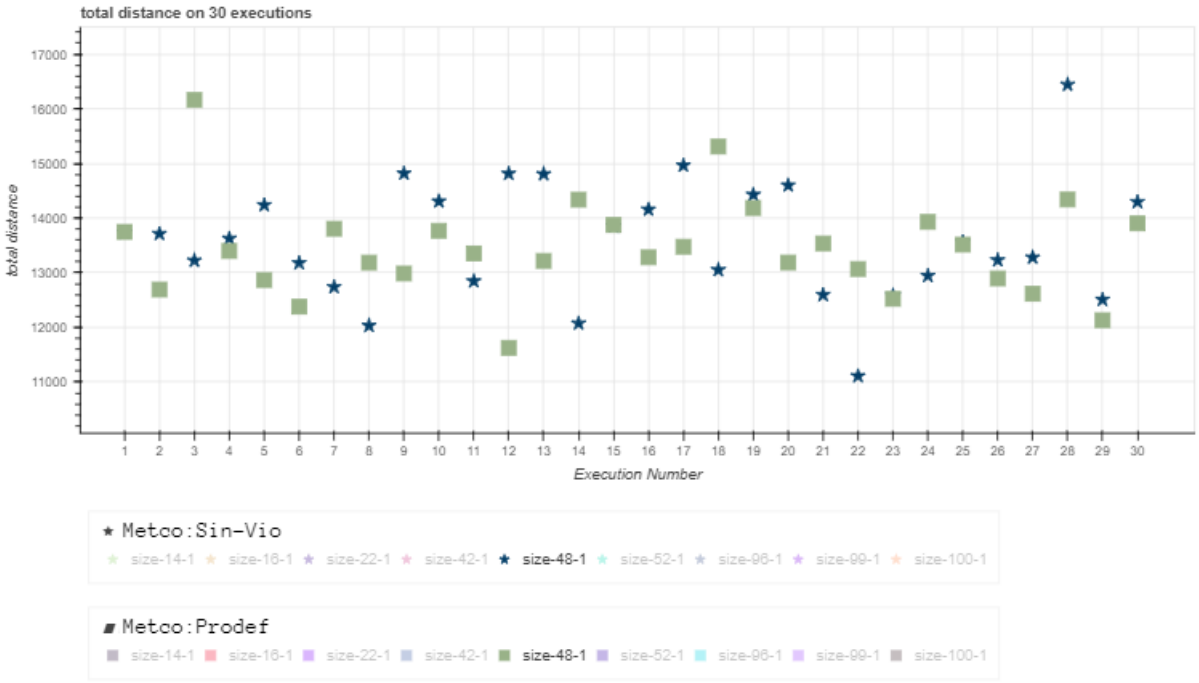


Figura 6.13: Instancia att48 al final de la ejecuciones con 80000 evaluaciones

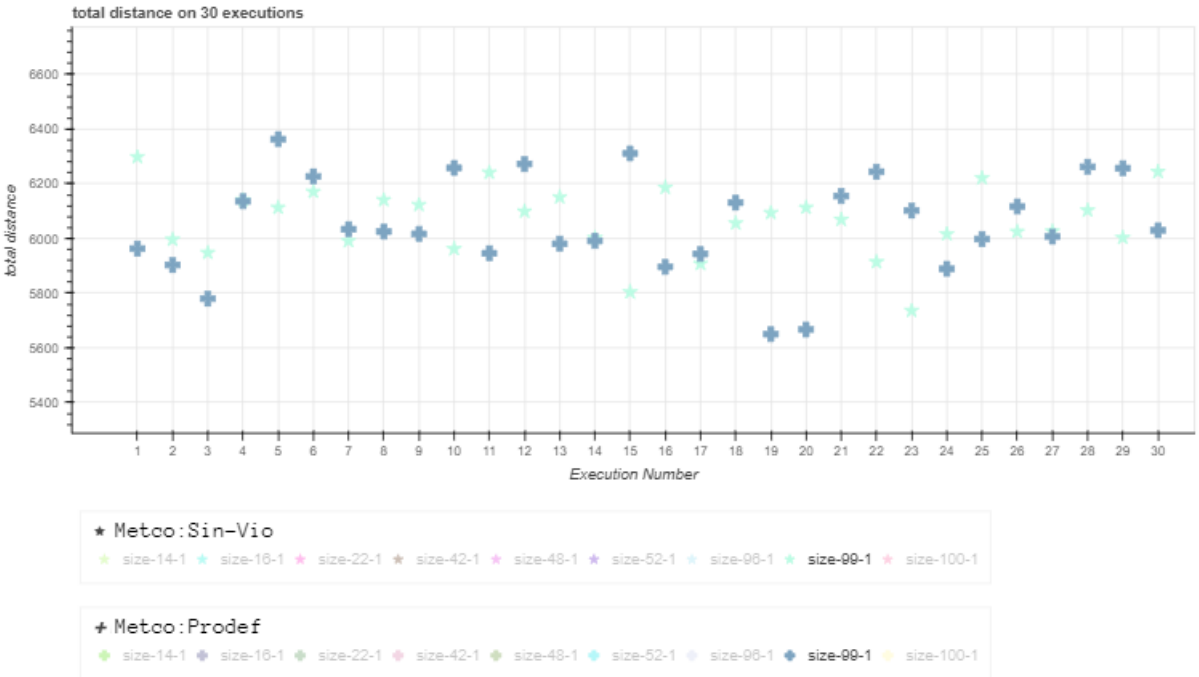


Figura 6.14: Instancia rat-99 al inicio de las ejecuciones con 1600 evaluaciones

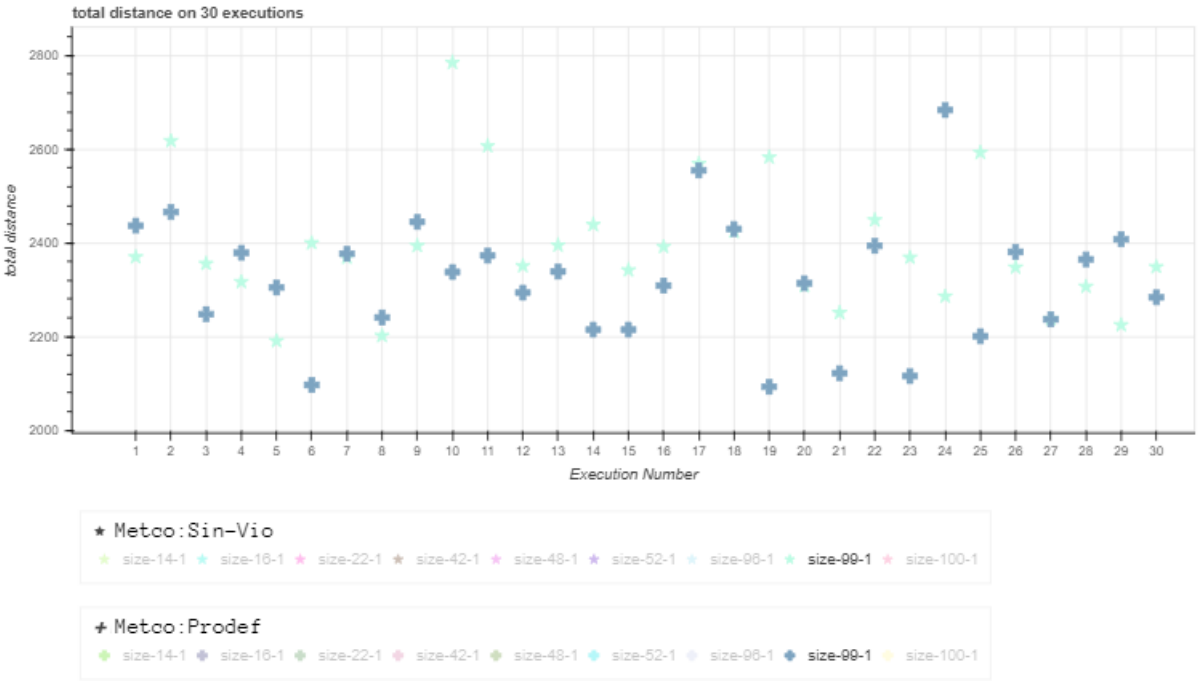


Figura 6.15: Instancia rat-99 al final de las ejecuciones con 80000 evaluaciones

Con respecto al problema *TSP* se concluye lo siguiente:

- Se reafirma que no hay soluciones "*no factibles*" por la manera en que está diseñado el problema, como se mencionó anteriormente. Esto se debe a la codificación de la solución como una permutación, que es un conjunto de valores **únicos** ordenados de determinada manera. Lo importante aquí, es que ningún número dentro de la permutación se repite y en consecuencia se evita que una ciudad sea visitada dos veces, a excepción de la inicial-final.
- Las instancias de distinto tamaño se comportan igual, no existe ese efecto que se observó en el problema *Knapsack*, que impedía ver la evolución de las instancias más pequeñas al utilizar un número muy alto de evaluaciones. Este efecto se ve al inicio de las ejecuciones en las Figuras 6.10, 6.12 y 6.14.
- Como se mencionó antes, las mejores soluciones conseguidas en la Tabla 6.4 pueden pertenecer a cualquier variante y tiene un ligero beneficio para la variante *Metco:Prodef*. Pero, como se puede ver al final de las ejecuciones en las Figuras 6.11, 6.13 y 6.15, ambas variantes terminan con soluciones muy similares, siendo resultado de la aleatoriedad. Por lo tanto, con respecto al valor objetivo no hay una variante mejor que otra.

Para generar las últimas gráficas de las Figuras 6.16, 6.17 y 6.18, se utilizó la instancia de tamaño 48, ya no sucede el mismo escenario que el problema *Knapsack* donde las evaluaciones eran tan grandes que no se apreciaba el resultado. Estas gráficas presentan en el eje de las abscisas el número de evaluaciones, siendo un total de 80000 evaluaciones y el eje de las ordenadas el valor del objetivo. Por lo tanto, se representan la evolución de la instancia durante toda la ejecución para cada una de las variantes: *Metco:Sin-Vio* y *Metco:Prodef*.

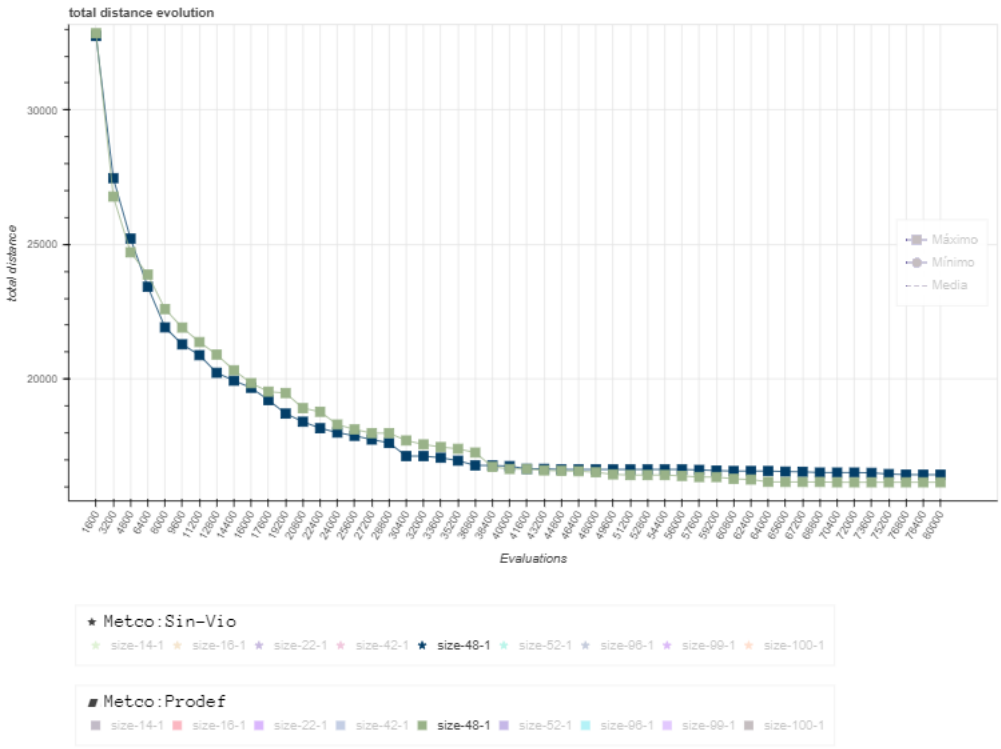


Figura 6.16: Valor máximo de la instancia att-48 durante 80000 evaluaciones

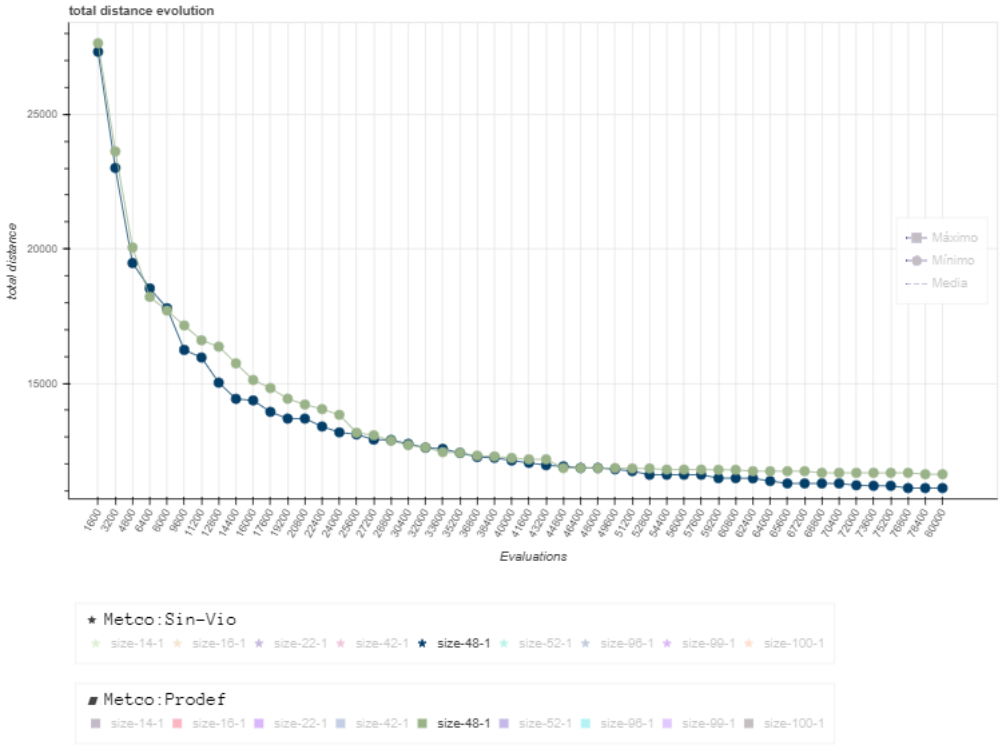


Figura 6.17: Valor mínimo de la instancia att-48 durante 80000 evaluaciones

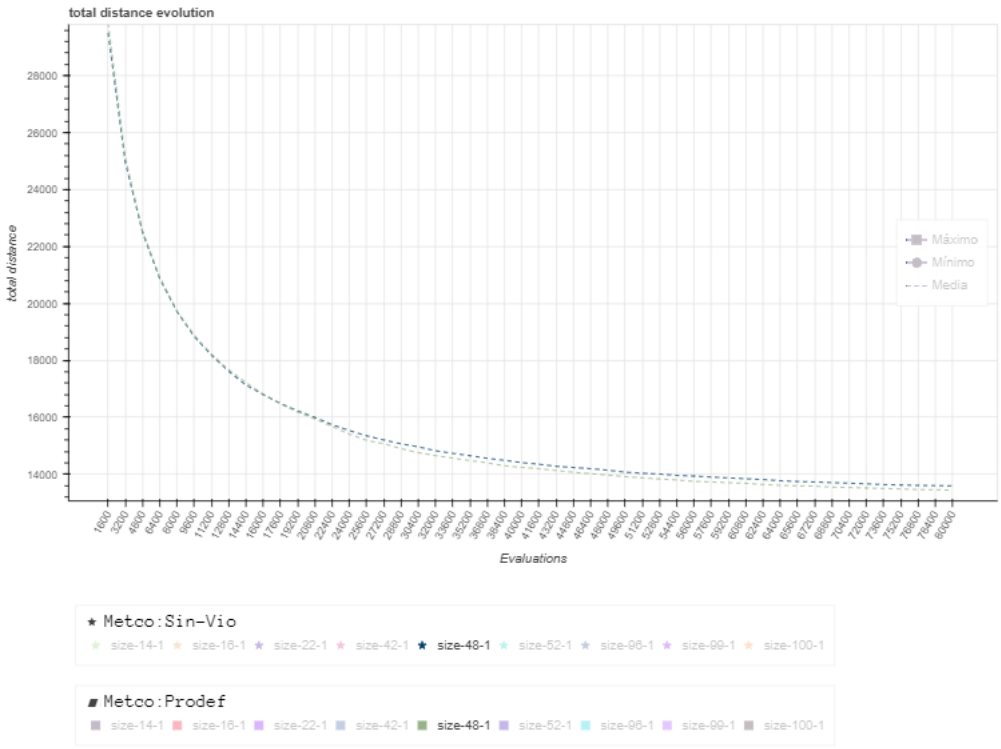


Figura 6.18: Media entre máximos y mínimos de la instancia att-48 durante 80000 evaluaciones

Viendo las gráficas de las Figuras 6.16, 6.17 y 6.18, se puede rectificar que la evolución de ambas variantes es muy similar.

6.4. Análisis

En las tablas 6.2 y 6.4 se puede observar el tiempo de cómputo medio asociado a cada uno de los tipos de ejecuciones, tanto con el problema *TSP* como *Knapsack* con distintas variantes e instancias. Atendiendo a los tiempos presentados en dichas tablas se puede hacer una evaluación del rendimiento del código auto-generado desde *Prodef* (variante *Metco:Prodef*) con respecto al resto de variantes que representan códigos escritos directamente en C++ por un usuario de la herramienta *METCO*.

Como se puede apreciar, la variante *Metco:Prodef* requiere más tiempo de cómputo para resolver las mismas instancias que las variantes escritas directamente en C++. La causa se encuentra en generación automática de código para las operaciones como **productorios** y **sumarios**, y en consecuencia en las funciones objetivo y/o en la comprobación de las restricciones del problema, ya que por lo general están compuestas por alguna de estas operaciones. Al traducir un sumatorio o productorio al lenguaje *prodef-lang* se requiere dos pasos para realizar dicha operación: primero se calcula un resultado parcial con la operación interna y posteriormente la suma o multiplicación de esos resultados parciales.

$$\sum_{i=1}^{5} x_i - 1$$

Para el sumatorio anterior se utiliza un rango que va del 1 al 5 y se aplica la operación interna $x_i - 1$ a cada elemento. Este sumatorio tiene dos representaciones según la variante utilizada, por un lado las escritas directamente en *METCO* tienen la estructura del Algoritmo 2, mientras que las auto generadas desde Prodef se corresponden con el Algoritmo 3.

Algoritmo 2 Proceso de sumatorio en variantes METCO

```
1: N ← 5
2: x[N]
3: resultado ← 0
4:
5: for i = 1; i <= N; i++ do
6:     resultado ← x[i] − 1 + resultado
7: end for
```

Algoritmo 3 Proceso de sumatorio en variantes Prodef

```
1: N ← 5
2: x[N]
3: solucionParcial[N]
4:
5: for i = 1; i <= N; i++ do
6:     solucionParcial[i] ← x[i] − 1
7: end for
8:
9: resultado ← 0
10: for i = 1; i <= N; i++ do
11:     resultado ← solucionParcial[i] + resultado
12: end for
```

En ambos pseudocódigos el resultado final del sumatorio se encuentra en la variable `resultado`, pero la manera de obtenerlo difiere, ya que el proceso de *Prodef* se añade un paso extra y por lo tanto se recorre el array de tamaño `N` dos veces, y no una sola vez como en las variantes escritas directamente en *METCO*. Este recorrido doble en las variantes de *Prodef* se debe a la forma en que está definido el lenguaje *prodef-lang* y como traduce estas operaciones, debido a que genera expresiones pensando en la programación funcional para utilizar métodos como `map` y `reduce` como se explicó en el Capítulo 4, por lo que impide utilizar una función que combine ambos pasos.

Esta diferencia de tiempo se nota aún más con el problema *Knapsack* que con el *TSP*, ya que este último no presenta restricciones y en consecuencia se realizan menos sumatorios que con respecto al *Knapsack*.

Capítulo 7

Conclusiones y líneas futuras

En este capítulo se expondrán las conclusiones del proyecto, así como posibles futuras líneas de trabajo.

7.1. Conclusiones

A partir de una fase análisis del estado previo de las herramientas *METCO* y *Prodef* se detectó que una de las primeras dificultades era la **curva de aprendizaje** de las herramientas utilizadas. Por ello, se documentó la instalación y proceso de construcción paso a paso de un problema en *METCO*, así como también la definición de la estructura común para un módulo en *Prodef*.

Después de esta fase de investigación, se desarrolló la integración entre *METCO* y *Prodef*, para resolver problemas de optimización combinatoria con otra perspectiva. Para tener una idea de los pasos a seguir, se utilizó los conocimientos de la integración entre *jMetal* y *Prodef* y se implementaron dos módulos, primero `prodef-compiler-cpp` para la traducción desde el lenguaje `prodef-lang` a C++, y luego el módulo `prodef-solver-metco` para generar el código C++ que define el problema, utilizando plantillas *handlebars* para su posterior resolución con *METCO*. La unión de ambas herramientas requirió de la adaptación de *METCO* para manejar problemas con un grado de violación en las restricciones, ya que de momento, *Prodef* no cuenta con la definición de heurísticas de búsqueda local capaz de producir soluciones factibles.

Para evaluar la calidad del resolutor se hizo un proceso de experimentación con instancias de los problemas *Knapsack* y *TSP* con dos principales categorías, los problemas escritos directamente en *METCO* y los auto generados desde *Prodef*. Para automatizar este proceso se crearon otros dos módulos, `prodef-problem-formatter` para adaptar formatos de instancias de distintas variantes y `prodef-evaluator` para generar múltiples ejecuciones de distintos problemas, variantes e instancias. El desarrollo de este análisis fue muy beneficioso para el resolutor en *METCO*, ya que con constantes ejecuciones se logró detectar y corregir fallos, logrando que las variantes generadas desde *Prodef* obtengan valores objetivos y tiempos muy cercanos a las variantes escritas directamente en el lenguaje objetivo, demostrando su **gran capacidad de resolución**.

Sin embargo, definir el problema directamente en *METCO* generalmente dará mejores resultados como se vio con la variante *Metco:Con-Vio*, pero es necesario conocer el

lenguaje objetivo y detalles específicos del problema a muy bajo nivel. Por ello, el objetivo de *Prodef* es **facilitar el modelado y resolución de problemas** sin la necesidad de conocer el lenguaje objetivo (C++ o Java) en el que se resuelve, lo que tiene un gran potencial para pequeñas y medianas empresas para optimizar procesos productivos sin la necesidad de contar con un programador y/o matemático para resolver el problema.

La clave está en tener conocimientos en el **porqué** o **qué** problema se necesita optimizar, más no en el **cómo**, siendo tanto *Prodef* como *METCO* fundamentales para lograr esta idea. Además de contar con un gran margen de crecimiento con ideas que se han detectado durante el desarrollo de este trabajo y resulta difícil pensar que exista un límite para estas herramientas, ya que el ser humano siempre va a tener la necesidad de optimizar procesos y eso es parte de su naturaleza.

7.2. Líneas futuras

A continuación se detalla una lista de posibles mejoras tanto para *METCO* como *Prodef*.

- **Automatizar instalación de *METCO***: Actualmente la instalación de esta herramienta tiene muchos pasos y algunos dependen del sistema operativo en el que se desarrolla. Por ello, se propone la utilización de Docker para automatizar la instalación y despliegue en cualquier entorno, aprovechando que los módulos que utilizan a *METCO* como *prodef-evaluator* y *prodef-solver-metco* están adaptados con variables de entorno y en consecuencia la comunicación con ellos es simple.
- **Analizar la calidad del código**: Una buena opción para evaluar el diseño de las distintas variantes es utilizar una herramienta para la evaluación del código fuente como SonarQube, ya sea de manera local o con una integración en el repositorio de la organización. La idea de utilizar esta plataforma es medir la diferencia entre el código escrito directamente en un lenguaje objetivo y el código auto generado desde un resolutor como *prodef-solver-metco* o *prodef-solver-jmetal*, hacer un análisis acerca de su legibilidad más que con su potencia resolutiva y mejorar el diseño del código.
- **Diseño o adaptación de un módulo para las evaluaciones de otros resolutores**: Actualmente se cuenta con el módulo *prodef-evaluator* para realizar múltiples ejecuciones de diferentes instancias y variantes en *METCO* junto a *prodef-solver-metco*. Por ello, se propone adaptar o diseñar un módulo genérico que permita hacer el mismo proceso con otros resolutores, para medir la calidad de soluciones y tiempo entre distintos resolutores.
- **Iniciar *METCO* con un fichero *json***: La manera de ejecutar *METCO* es a través de comandos en la terminal como se vio en el Capítulo 2, lo cual resulta complejo para la detección de errores en los parámetros que se pasan al problema. Por lo tanto, implementar otra manera de comunicación de más alto nivel reduciría este problema como lo es el formato *json*, el cual es simple de añadir porque *METCO* ya implementa este formato como la salida de la ejecución.
- **Implementar *cmake* en *METCO***: Actualmente se utiliza *Make* para el proceso de compilación de *METCO* a excepción de las nuevas librerías como se explicó en el

Capítulo 2. Se propone utilizar *CMake* en vez de *Make* como se hizo con las nuevas librerías y hacer de *METCO* una herramienta más fácil de extender y mantener, evitando así la complejidad del problema de múltiples capas de direccionamiento indirecto (véase la Figura 2.2).

- **Añadir más problemas de optimización**: Actualmente se ha implementado el resolutor con dos problemas de optimización (Knapsack y *TSP*) consiguiendo resultados satisfactorios, siendo problemas que se pueden clasificar según la representación de las soluciones como binarios y permutaciones respectivamente. Se propone implementar problemas con otro tipo de representación de variables como enteros/reales. Un ejemplo de ello, es el problema de planificación de dietas, cuyo objetivo es que la suma de cada nutriente de un conjunto de alimentos esté dentro de un rango específico, sin superar un volumen determinado y minimizando su coste.

Capítulo 8

Summary and Conclusions

This chapter has a summary and conclusions of the project.

8.1. Summary

During the analysis phase, I developed the documentation of the installation and construction process of a problem in METCO, as well as the definition of the common structure for a module in Prodef.

During the development phase, I developed specific modules for solving combinatorial optimization problems from Prodef with METCO. For the integration between both tools, I created prodef-compiler-cpp to translate from prodef-lang language to C++ and prodef-solver-metco to generate the C++ code that defines the problem, using template handlebars, and then solving with METCO.

In the evaluation phase, I made an experimentation with different instances of the Knapsack and TSP problems, and different variants such as problems written directly in METCO and autogenerated from Prodef. I created the modules prodef-problem-formatter and prodef-evaluator to optimize and generate multiple executions of different problems, variants and instances

8.2. Conclusion

Defining an optimization problem directly with code will generally give better results, but it’s necessary to know the target language and specific details of the problem at a very low level. Prodef’s goal is to simplify modeling and problem solving without the need to know the target language (C++ or Java), so it has great potential for small and medium-sized companies to optimize production processes without the need for a programmer and/or mathematician.

The idea is to know what needs to be optimized, not how to optimize it, and both Prodef and METCO are fundamental to achieve this idea. They also have a great potential for growth and it is difficult to think that there’s a limit to these tools, since human beings will always have the need to optimize processes and because is part of their nature.

Capítulo 9

Presupuesto

En este capítulo se presenta el presupuesto de este Trabajo de Fin de Grado. El presupuesto se ha dividido en dos apartados: costes de infraestructuras tecnológicas y costes de desarrollo (recursos humanos).

En la Tabla 9.1 se presentan los costes tecnológicos que corresponden a todos los materiales y/o equipamiento utilizado.

Tipos	Descripción	Coste
Amortización del equipo informático	Equipo informático utilizado para el desarrollo del trabajo como PC, ratón, pantalla y teclado	320€
	Total	320€

Tabla 9.1: Costes de infraestructura

En la Tabla 9.2 se relaciona la cantidad de horas empleadas en el desarrollo de este Trabajo de Fin de Grado. Como se puede apreciar, el total de horas invertidas en el desarrollo de este trabajo superan la cantidad de 300 horas, correspondientes al número de créditos **ECTS** de la asignatura "*Trabajo Fin de Grado*" (12 créditos). Para la estimación del precio por hora se consultaron distintas fuentes: Linkedin y InfoJobs.

Tipos	Coste/Hora	Horas	Total
Backend	13	230	2990€
Analista de datos	14	90	1260€
Arquitecto de software	19	90	1710€
	Total	410	5960€

Tabla 9.2: Costes de desarrollo

Bibliografía

[1] Gilles Brassard y Paul Bratley. *Fundamentals of algorithms.* Prentice Hall, 1996, págs. I-XIX, 1-524. isbn: 978-0-13-335068-5.

[2] Carlos A. Coello Coello, Carlos Segura y Gara Miranda. «History and Philosophy of Evolutionary Computation». En: *Handbook on Computational Intelligence*. World Scientific, 2016, págs. 509-545. doi: 10.1142/9789814675017_0014.

[3] Juan J. Durillo y Antonio J. Nebro. «jMetal: A Java framework for multi-objective optimization». En: *Advances in Engineering Software* 42.10 (2011), págs. 760-771. issn: 0965-9978. doi: https://doi.org/10.1016/j.advengsoft.2011.05.014. url: https://www.sciencedirect.com/science/article/pii/S0965997811001219.

[4] A. E. Eiben y J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003. isbn: 3-540-40184-9. url: http://www.cs.vu.nl/~gusz/ecbook/ecbook.html.

[5] Erich Gamma y col. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1.ª ed. Addison-Wesley Professional, 1994. isbn: 0201633612. url: http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1.

[6] Andrés Calimero García Pérez. *Prodef: meta-modelado de problemas de optimización combinatoria*. Inf. téc. Universidad de La Laguna, 2020. url: http://riull.ull.es/xmlui/handle/915/20601.

[7] Fred W. Glover y Gary A. Kochenberger. *Handbook of Metaheuristics*. Hardcover. 2003.

[8] *GNU make manual - GNU Project - Free Software Foundation*. 2020. url: https://www.gnu.org/software/make/manual/ (visitado 25-02-2022).

[9] K. Hussain y col. «Metaheuristic research: a comprehensive survey». En: *Artificial Intelligence Review* 52 (4 2019), págs. 2191-2233. issn: 1573-7462.

[10] Briukhnov Konstantin. *Knapsack Test Instances*. https://github.com/CostaBru/knapsac. 2009. (Visitado 25-02-2022).

[11] Coromoto León, Gara Miranda y Carlos Segura. «Metco: a Parallel Plugin-Based Framework for Multi-Objective Optimization». En: *International Journal on Artificial Intelligence Tools* 18.4 (2009), págs. 569-588.

[12] Ken Martin. *Cmake Documentation*. 2022. url: https://cmake.org/ (visitado 25-02-2022).

[13] Antonio J. Nebro, Juan J. Durillo y Matthieu Vergne. «Redesigning the JMetal Multi-Objective Optimization Framework». En: *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*. GECCO Companion '15. Madrid, Spain: Association for Computing Machinery, 2015, págs. 1093-1100. isbn: 9781450334884. doi: 10.1145/2739482.2768462. url: https://doi.org/10.1145/2739482.2768462.

[14] Jorge Nocedal y Stephen J. Wright. *Numerical optimization*. New York: Springer, 2006. isbn: 9780387400655 0387400656 0387303030 9780387303031 0387987932 9780387987934.

[15] Gerhard Reinelt. *Discrete and combinatorial optimization*. 1991. url: http://comopt.ifi.uni-heidelberg.de/.

[16] Zbigniew Maciej Skolicki. «An Analysis of Island Models in Evolutionary Computation». AAI3289714. Tesis doct. USA: George Mason University, 2007. isbn: 978-0-549-32961-9.

[17] El-Ghazali Talbi. *Metaheuristics - From Design to Implementation.* Wiley, 2009, págs. I-XXIX, 1-593. isbn: 978-0-470-27858-1.

[18] Thomas Weise. *Global Optimization Algorithms - Theory and Application*. en. Second. Self-Published, 2009. url: http://www.it-weise.de/.