Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Prodef: meta-modelado de problemas de optimización combinatoria

Prodef: meta-modeling of combinatorial optimization problems

Andrés Calimero García Pérez

La Laguna, 5 de julio de 2020

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

D. **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 cotutor

C E R T I F I C A N

Que la presente memoria titulada:

"*Prodef: meta-modelado de problemas de optimización combinatoria*"

ha sido realizada bajo su dirección por D. **Andrés Calimero García Pérez**, con N.I.F. 51.149.384-Y.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 5 de julio de 2020

Agradecimientos

Me gustaría agradecer a todos los profesores de la Universidad de La Laguna por su labor formando a profesionales y ciudadanos, y en especial a Gara Miranda Valladares y Casiano Rodríguez León, por tutorizar este trabajo y proveerme de todos los recursos necesarios para su realización.

También me gustaría agradecer a la comunidad *Open Source*, este trabajo no hubiese sido posible sin el trabajo altruista de todos los desarrolladores que han contribuido en proyectos como *ANTLR* y *Node.js*.

Licencia

Resumen

Los métodos de optimización bio-inspirados ofrecen grandes posibilidades para la resolución de problemas en el ámbito empresarial pero, debido a su complejidad intrínseca y también a sus particularidades a la hora de implementarlos y adaptarlos a cada uno de los problemas, no se ha conseguido extender su uso fuera del ámbito de la investigación convencional.

En este trabajo se ha desarrollado una herramienta que permite la definición de problemas de optimización combinatoria a un nivel de descripción abstracto y su posterior resolución usando librerías y frameworks externos, como jMetal. Esta herramienta actúa como capa de abstracción entre el modelo del problema y sus 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.

Palabras clave: Optimización combinatoria, algoritmos evolutivos, herramientas para optimización, meta-modelado de problemas, prodef.

Abstract

Bio-inspired optimization methods offer great possibilities for solving problems in the business environment but, due to their intrinsic complexity and also their particularities when implementing and adapting them to each of the problems, it has not been possible to extend its use outside the scope of conventional research.

In this work, a tool has been developed that allows the definition of combinatorial optimization problems at an abstract level of description and their subsequent resolution using libraries and external frameworks, such as jMetal. This tool acts as an abstraction layer between the problem model and its resolution techniques, in order to facilitate the user, without programming knowledge or bio-inspired optimization methods, the task of modeling and solving combinatorial optimization problems.

Keywords: Combinatorial optimization, evolutionary algorithms, optimization frameworks, problem meta-modeling, prodef.

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

En este capítulo se detalla la motivación y objetivos de este trabajo, así como los antecedentes y el estado actual del tema.

1.1. Motivación

Optimizar consiste en encontrar las mejores soluciones a los problemas que se nos plantean. A menudo existen formulaciones matemáticas que modelan estos problemas del mundo real. Prácticamente en todos los campos de la Ciencia y de la Ingeniería pueden encontrarse este tipo de problemas. El objetivo principal en el campo de la optimización es encontrar una solución óptima a un problema, de entre el conjunto de soluciones factibles al mismo, con un coste computacional razonable. En función del problema, la naturaleza de las soluciones varía y estarán definidas por una serie de restricciones, determinando así cuáles de las soluciones serán factibles y cuáles no. En el caso de un conjunto finito de soluciones discretas, se tratará de un problema de optimización combinatoria.

La optimización combinatoria es una rama de la optimización en Matemáticas aplicadas y en Ciencias de la Computación, relacionada con la investigación operativa, la teoría de algoritmos y la teoría de la complejidad computacional. El propósito de la optimización combinatoria es encontrar objetos matemáticos discretos que maximicen (o minimicen) una determinada función objetivo. Generalmente, estos objetos se denominan estados y al conjunto de todos los posibles estados candidatos se le denomina espacio de búsqueda. La resolución de un problema de optimización combinatoria consta de tres puntos clave:

- **Definir un conjunto de soluciones factibles**. Para ello es necesario especificar la forma en la que se representan las soluciones y también las condiciones que debe cumplir cualquier solución para ser válida. Esto requiere de un conocimiento específico del problema y de su aplicación de dominio.
- **Determinar una función de evaluación a optimizar**. La formulación de dicha función consiste en calificar como de buena es una solución. En este caso también es necesario un conocimiento específico y profundo del problema en sí mismo.
- **Elegir un método de optimización**. La elección del método de optimización depende de la complejidad del problema. En este caso, es necesario un conocimiento específico sobre algoritmia y técnicas de optimización (ya sean exactas o aproximadas).

Los problemas de optimización combinatoria [3, 5, 6, 7] se pueden categorizar en función de la representación de las soluciones: permutaciones, binarios, enteros, subgrafos, árboles, etc. En este trabajo sólo trataremos con problemas de optimización combinatoria cuyas soluciones se puedan codificar mediante permutaciones, variables binarias, enteras y/o reales (entendiendo por variables reales variables que pueden tomar valores decimales pero que son discretas, no continuas).

Formalmente, un problema de optimización combinatoria es una tupla (I, f, m, g) donde:

- I es un conjunto de instancias;
- dada una instancia $x \in I$, $f(x)$ es el conjunto finito de soluciones factibles;
- dada una instancia x y una solución factible y de x, $m(x, y)$ denota la medida de y (valor de la función objetivo), normalmente un número real positivo.
- g es la función objetivo, bien de minimización o de maximización.

El objetivo es, por tanto, encontrar para alguna instancia x una solución óptima, esto es, una solución factible y con:

$$m(x, y) = g\{m(x, y') \mid y' \in f(x)\}$$

Si, además, el problema cumple con las siguientes condiciones:

- el tamaño de cada solución factible $y \in f(x)$ está polinómicamente limitado en el tamaño de la instancia x dada,
- los lenguajes $\{x \mid x \in I\}$ y $\{(x, y) \mid y \in f(x)\}$ pueden ser reconocidos en tiempo polinomial, y
- m es computable en tiempo polinomial.

implica que el problema es un problema de optimización NP (NPO). Los problemas de optimización combinatoria interesantes usualmente son NPO, con este tipo de problemas no podemos garantizar encontrar la mejor solución en un tiempo razonable. Por este motivo se suele recurrir a algoritmos de aproximación para obtener una solución cercana a la óptima en un tiempo razonable; estos algoritmos suelen recurrir a heurísticas para encontrar soluciones.

"Heurística" deriva del griego *heuriskein*, que significa "encontrar" o "descubrir". Las heurísticas son técnicas que buscan soluciones de buena calidad (cercanas al óptimo) a un costo computacional razonable. Podemos clasificar las heurísticas en:

- Métodos constructivos: generan una solución (desde una solución vacía) agregando componentes hasta completar la solución. Son métodos rápidos, de baja calidad en los resultados.
- Métodos de búsqueda local: comienzan desde una solución e iterativamente van reemplazando la solución actual con alguna solución parecida de mejor calidad.

Un ejemplo sencillo de heurística podría ser la heurística *greedy* (o golosa). Esta heurística trata de construir una solución eligiendo de manera iterativa los elementos componentes de menor costo. La heurística *greedy* se podría aplicar, por ejemplo, en el problema del viajante de comercio (TSP), haciendo que en cada iteración se elija como próxima ciudad a visitar la ciudad más próxima (menor costo) entre las aún no visitadas.

Las heurísticas dependen en gran medida del problema concreto para el que se han diseñado. Las técnicas e ideas aplicadas a la resolución de un problema son específicas de éste, lo que hace difícil trasladar el aprendizaje a otros problemas. Por esta y otras limitaciones surgen las denominadas meta-heurísticas.

En definitiva, encontrar una solución óptima (mediante un método exacto) no suele ser una tarea fácil de resolver y en ocasiones podría llevar mucho tiempo de cómputo. En estos casos, una aproximación a la solución óptima puede ser suficiente para muchos problemas reales. Normalmente estas soluciones no son las mejores pero son suficientemente de buena calidad y pueden ser encontradas en un tiempo razonable para entornos reales y dinámicos. La resolución de estos problemas mediante meta-heurísticas es uno de los temas más investigados en el ámbito de las Ciencias de la Computación y observamos que, entre este tipo de técnicas, los algoritmos bio-inspirados han sido aplicados con éxito en áreas tan disímiles como la salud, el transporte, la economía y la industria, entre otras. Prueba de ello es que la actual literatura en el ámbito del diseño y la aplicación de este tipo de técnicas es extensa y continúa progresivamente en aumento año a año.

1.2. Meta-heurísticas

Las meta-heurísticas se pueden definir como heurísticas de nivel más alto. Son estrategias que guían el proceso de búsqueda de soluciones, incluyendo, en general, heurísticas subordinadas. Las meta-heurísticas admiten una descripción a nivel abstracto y son de uso genérico, es decir, no son específicas para un tipo de problema, por este motivo son especialmente interesantes para este trabajo. El objetivo de las meta-heurísticas sigue siendo encontrar soluciones factibles y de buena calidad y, para ello, tratan de recorrer el espacio de soluciones sin quedar “atrapados” en una zona. Para conseguirlo usan las nociones de exploración del espacio y de explotación de las soluciones obtenidas. En la Figura 1.1 se muestra una clasificación de meta-heurísticas según sus estrategias.

En este trabajo se implementa un resolutor basado en meta-heurísticas bio-inspiradas, más concretamente, meta-heurísticas inspiradas en la evolución de las especies, basadas en la combinación de soluciones y poblacionales. Estas meta-heurísticas han demostrado ofrecer buenos resultados y son ampliamente estudiadas en el ámbito de las Ciencias de la Computación. Cabe destacar que la herramienta no limita el uso de otros tipos de meta-heurísticas: las estrategias usadas por los resolutores no están limitadas ni predefinidas por la herramienta.

La computación evolutiva posee un amplio espectro de técnicas heurísticas que funcionan emulando mecanismos de evolución natural, trabajan sobre una población de individuos o conjunto de soluciones, que evoluciona utilizando mecanismos de selección y construcción de nuevas soluciones candidatas mediante recombinación de características de las soluciones seleccionadas. Las etapas del mecanismo evolutivo son:

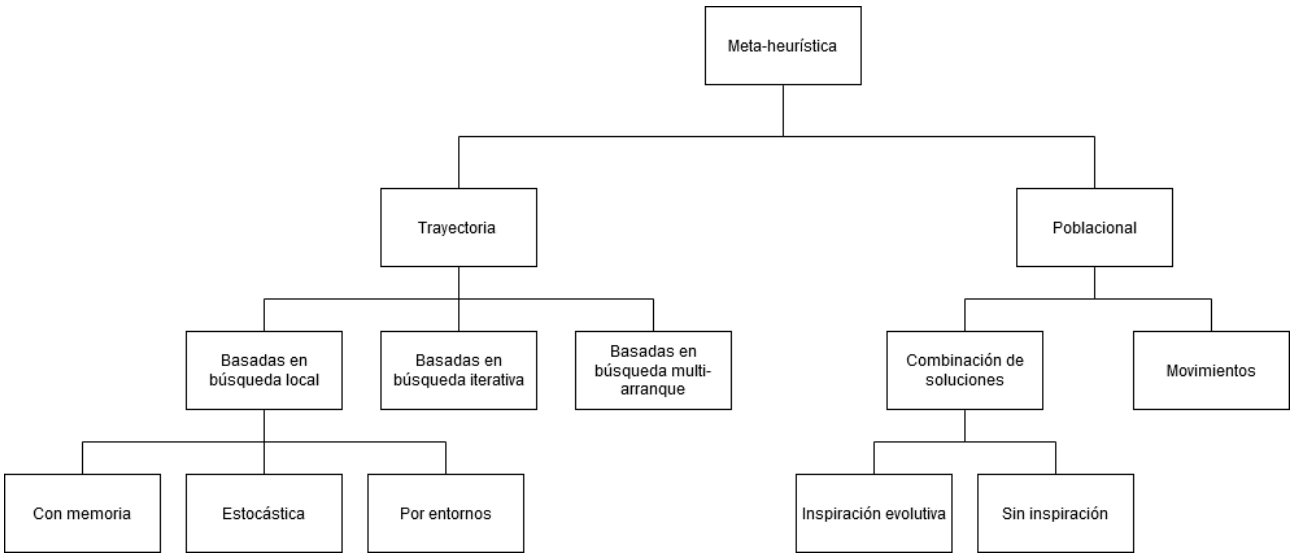

Figura 1.1: Clasificación de meta-heurísticas

- Evaluación de los individuos mediante la función de fitness (función objetivo).
- Selección de individuos adecuados (de acuerdo al fitness) para la aplicación de operadores evolutivos.
- Aplicación de operadores evolutivos.
- Reemplazo o recambio generacional.

Los operadores evolutivos determinan el modo en el que el algoritmo explora el espacio de búsqueda: el *cruce* es el operador principal, mientras que la *mutación* es un operador secundario, utilizado para agregar mayor diversidad al mecanismo de exploración.

El resolutor implementado en este trabajo hace uso de algoritmos evolutivos, que pertenecen al campo de la computación evolutiva. En el Algoritmo 1 se muestra el esquema de un algoritmo evolutivo.

Algoritmo 1: Esquema de un algoritmo evolutivo

```
1 P ← GenerarPoblacionInicial();
2 Evaluar(P);
3 while no se cumplen las condiciones de terminación do
4 |   P' ← Recombinar(P);
5 |   P'' ← Mutar(P');
6 |   Evaluar(P'');
7 |   P ← Seleccionar(P ∪ P'');
```

Un tipo de algoritmo evolutivo son los algoritmos genéticos (GA), estos algoritmos funcionan entre el conjunto de soluciones de un problema llamado fenotipo, y el conjunto de individuos de una población natural, codificando la información de cada solución en una cadena, generalmente binaria, llamada cromosoma. Los símbolos que forman la cadena son llamados genes. Cuando la representación de los cromosomas se hace con cadenas de dígitos binarios se le conoce como genotipo. Los cromosomas evolucionan

a través de iteraciones, llamadas generaciones. En cada generación, los cromosomas son evaluados usando alguna medida de aptitud (fitness). Las siguientes generaciones (nuevos cromosomas), son generadas aplicando los operadores genéticos repetidamente, siendo estos los operadores de selección, cruce, mutación y reemplazo.

En un algoritmo genético, una población de soluciones candidatas (llamadas individuos, criaturas o fenotipos) a un problema de optimización se desarrolla hacia mejores soluciones. Cada solución candidata tiene un conjunto de propiedades (sus cromosomas o genotipos) que pueden ser mutadas y alteradas. Tradicionalmente, las soluciones se representan en binario como cadenas de ceros y unos, pero también son posibles otras codificaciones.

El tamaño de la población depende de la naturaleza del problema, pero normalmente contiene varios cientos o miles de posibles soluciones. A menudo, la población inicial se genera aleatoriamente, creando individuos cuya codificación permite la representación de todas las posibles soluciones del problema (espacio de búsqueda). Ocasionalmente, las soluciones pueden ser "sembradas" en áreas donde es probable encontrar soluciones óptimas.

Durante cada generación sucesiva, una parte de la población existente se selecciona como progenitores de la nueva generación. Las soluciones individuales se seleccionan a través de un proceso basado en la aptitud, donde las soluciones "más aptas" (medido por una función de evaluación o *fitness*) son típicamente más probables de ser seleccionadas y, por tanto, tienen mayor probabilidad de sobrevivir. Ciertos métodos de selección evalúan la aptitud de cada solución y preferentemente seleccionan las mejores soluciones. Otros métodos califican solo a una muestra aleatoria de la población, ya que el proceso anterior puede llevar mucho tiempo.

La función de *fitness* se define sobre la representación genética y mide la calidad de la solución representada. Esta función depende siempre del problema y de la codificación utilizada. Por ejemplo, en el problema de la mochila se quiere maximizar el valor total de los objetos que se pueden poner en una mochila de alguna capacidad fija. Una representación de una solución puede ser un vector de bits, donde cada bit representa un objeto diferente, y el valor del bit (0 o 1) representa si el objeto está o no en la mochila. No todas las representaciones son válidas (*factibles*), ya que el tamaño de los objetos puede exceder la capacidad de la mochila. La aptitud de la solución es la suma de los valores de todos los objetos en la mochila si la representación es válida, o 0 de lo contrario.

Tras la selección del conjunto de individuos a partir de los cuales obtendremos descendencia, el siguiente paso consiste en generar la nueva generación de individuos (*offspring*) a través de una combinación de operadores genéticos: entrecruzamiento cromosómico (también llamado *crossover*, cruce o recombinación) y mutación. Cada nueva solución generada desciende directamente de un par de soluciones "padre". Al producir una solución descendiente usando los métodos de cruce y mutación anteriormente mencionados, se crea una nueva solución que típicamente comparte muchas de las características de sus "padres", tal y como sucede con la herencia genética. Este proceso de selección de padres, y de recombinación y mutación para obtener nuevos descendientes continúa hasta que se genere una nueva población (de soluciones) de tamaño apropiado.

Estos procesos finalmente resultan en la siguiente generación de población de cromosomas, que es diferente a la generación inicial. En general, la aptitud física promedio

se ha incrementado por este procedimiento para la población, ya que solo los mejores organismos de la primera generación son seleccionados para la cría, junto con una pequeña proporción de soluciones menos aptas. Estas soluciones menos aptas aseguran la diversidad genética dentro del grupo genético de los padres y, por lo tanto, aseguran la diversidad genética de la siguiente generación de hijos.

En estos contextos es común ajustar parámetros como la probabilidad de mutación, la probabilidad de entrecruzamiento cromosómico y el tamaño de la población para encontrar ajustes razonables para la clase de problema que se está trabajando. Una tasa de mutación muy pequeña puede conducir a la deriva genética, mientras que una tasa de mutación muy alta conlleva a una búsqueda aleatoria. Esto además provoca la pérdida de buenas soluciones, a menos que se utilice una selección elitista. Por su parte, una tasa de recombinación demasiado alta puede conducir a la convergencia prematura del algoritmo genético.

El proceso generacional se repite hasta que se alcanza una condición de terminación. Algunas de las condiciones de terminación más comunes son:

- Se encuentra una solución que satisface los criterios mínimos.
- Se alcanza un número fijado de generaciones.
- Se alcanza el presupuesto asignado (tiempo de cálculo / dinero).
- La aptitud de la solución de la clasificación más alta está alcanzando o ha alcanzado una meseta tal que las sucesivas iteraciones ya no producen mejores resultados.
- Inspección manual.
- Combinaciones de las anteriores.

El resolutor implementado en este trabajo usa un algoritmo genético genérico simple para resolver problemas de optimización combinatoria mono-objetivo (una única función objetivo), y para los problemas multi-objetivo (varias funciones objetivo) hace uso del algoritmo NSGA-II [2], los detalles de implementación y parámetros de configuración usados se exponen en el apartado 3.7.3, de esta memoria.

1.3. Antecedentes y estado actual del tema

Actualmente, existen diversas plataformas y entornos software para optimización mediante algoritmos evolutivos y otras meta-heurísticas bio-inspiradas (ver Subsección 1.3.1). Sin embargo, lograr una clara separación entre el problema y las implementaciones de los resolutores sigue siendo un desafío, debido a muchas dificultades tanto de representación del espacio de soluciones, como de búsqueda dentro de dicho espacio. Una de las principales dificultades a la hora de implementar meta-heurísticas es cómo definir una representación general para las soluciones de los problemas, de modo que el espacio de búsqueda para el problema correspondiente pueda estar completamente representado. Otra dificultad, una vez se ha definido el espacio de búsqueda que representa todas las posibles soluciones, es definir un mecanismo para explorar ese espacio. Este mecanismo

de exploración debería definir los movimientos o criterios para seleccionar las próximas soluciones a tener en cuenta.

En la mayoría de las estrategias de exploración, tanto la representación de la solución como la forma en que se pasa de una solución a otra depende en gran medida del propio algoritmo. De esta forma, el uso de un algoritmo o estrategia concreta indirectamente implica que el usuario debe proporcionar una implementación diferente y adecuada para cada caso, así como diferentes tipos de operadores de variación (por ejemplo, los operadores de cruce y mutación en los algoritmos evolutivos). Esto puede resultar una tarea bastante compleja y requiere de mucho tiempo y experiencia, pues no sólo hay que conocer las características del problema a resolver sino también los detalles inherentes a la propia técnica algorítmica a aplicar. Este hecho dificulta especialmente la aplicación de estas técnicas por parte de usuarios no expertos en este ámbito de investigación.

1.3.1. Frameworks para optimización con meta-heurísticas

A continuación se listan los frameworks y herramientas más extendidas para optimización con meta-heurísticas [8]:

- **EasyLocal**. EasyLocal es un framework escrito en C++ centrado en la resolución de problemas de optimización combinatoria mediante meta-heurísticas de búsqueda local, no implementa meta-heurísticas basadas en estrategias evolutivas y tampoco proporciona soporte para meta-heurísticas multiobjetivo. Destaca por su implementación de la meta-heurística *Tabu Search* y por su función de procesamiento por lotes.

- **ECJ**. ECJ es un framework de computación evolutiva escrito en Java, es el framework para optimización más popular de los expuestos en esta sección, posee una extensa documentación y una gran cantidad de *papers* publicados. ECJ destaca por su rendimiento, permite la ejecución paralela y distribuida de algoritmos, y por su implementación de algoritmos evolutivos. Se centra casi exclusivamente en estrategias evolutivas, apenas implementa otras meta-heurísticas.

- **ParadisEO**. ParadisEO es un framework escrito en C++ y basado en el framework EO (*Evolving Objects*), está centrado en el diseño reusable de las meta-heurísticas. Destaca por su documentación, implementación eficiente (soporte para ejecución pararela y distribuida) y soporte para la hibridación.

- **EvA2**. EvA2 es un framework escrito en Java centrado en técnicas meta-heurísticas basadas en población (entre ellas las evolutivas), aunque también implementa meta-heurísticas clásicas como *Hill Climbing* o *Simulated Annealing*. EvA2 soporta problemas multi-objetivo y ejecución distribuida de algoritmos.

- **FOM**. FOM es un framework escrito en Java que implementa un gran cantidad de meta-heurísticas, en comparación con el resto de frameworks mencionados en esta sección, FOM es el que más implementaciones realiza, aunque no implementa meta-heurísticas multi-objetivo. FOM también destaca por su soporte para el manejo de restricciones e hiper-heurísticas, por otro lado, su documentación es escasa, al igual que su popularidad.

- **HeuristicLab**. HeuristicLab es un framework para la resolución de problemas de optimización, implementa meta-heurísticas basadas en población, incluyendo las basadas en estrategias evolutivas. HeuristicLab difiere del resto de frameworks al proveer una interfaz gráfica que permite modelar problemas, e incluso implementar/modificar algoritmos. Su diseño basado en plugins y su API hace que sea altamente interoperable, destaca también por la cantidad de problemas y tutoriales disponibles.

- **JCLEC**. JCLEC es un framework escrito en Java de computación evolutiva, es parecido a ECJ en cuanto a funcionalidades, aunque carece de soporte para ejecución pararela o distribuida y, al contrario que ECJ, sí soporta diseño de experimentos y análisis estadístico. JCLEC destaca especialmente por su ingeniería de software, su código sigue las buenas practicas de desarrollo de software.

- **MALLBA**. MALLBA es un framework escrito en C++ que implementa un conjunto de meta-heurísticas para la resolución de problemas de optimización, también implementa algoritmos exactos y heurísticas. MALLBA hace especial énfasis en el soporte de ejecuciones paralelas y distribuidas, permitiendo que el usuario se beneficie de estas características sin tener que preocuparse por detalles de bajo nivel.

- **OAT**. OAT es un framework escrito en Java para la resolución de problemas de optimización, se centra especialmente en algoritmos bio-inspirados, como los algoritmos evolutivos, sistema inmunitario artificial o colonia de hormigas. OAT destaca por su soporte para el diseño de experimentos y análisis estadístico, proporciona una interfaz gráfica.

- **Opt4j**. Opt4j es un framework modular escrito en Java, se centra especialmente en la resolución de problemas multi-objetivo mediante meta-heurísticas bio-inspiradas. Opt4j proporciona una interfaz gráfica para la configuración de los parámetros de optimización y visualización del proceso, no implementa soporte para diseño de experimentos ni análisis estadístico.

- **jMetal**. jMetal es un framework escrito en Java que se centra en la implementación de meta-heurísticas multi-objetivo, aunque también implementa mono-objetivo. jMetal soporta, entre otras cosas, ejecución pararela en determinados algoritmos, obtención de indicadores de calidad (como hiper volumen o distancia generacional invertida), representaciones de variables binarias, reales, enteras y permutaciones y manejo de problemas con restricciones. Destaca la ingeniería de software de su código.

1.3.2. Lenguajes de modelado para optimización

Los lenguajes de modelado algebraico (*AML* por sus siglas en ingles) son lenguajes de alto nivel que permiten describir problemas matemáticos, entre ellos problemas de optimización, usando una sintaxis similar a la notación matemática de dichos problemas. Los *AML* solo se encargan del modelado del problema, para resolverlos hacen uso de resolutores externos [1].

A continuación se listan algunos de los *AML* más usados:

- **OPL**: *OPL* (*Optimization Programming Language*) es un *AML* propietario de *IBM* para su software de optimización *CPLEX*.
- **AMPL**: *AMPL* es un *AML* propietario de la empresa homónima *AMPL Optimization*, este lenguaje, al contrario que *OPL*, esta diseñado para poder ser usado con múltiples resolutores, como *CPLEX*, *GUROBI*, *XPRESS*, *LOQO* o *MINOS*.
- **GAMS**: *GAMS* es un *AML* propietario de la empresa homónima *GAMS*, al igual que *AMPL* también proporciona un amplio abanico de resolutores externos compatibles.
- **AIMMS**: *AIMMS* es una plataforma especializada en problemas de optimización de cadenas de suministro, proporciona su propio *AML* propietario de la empresa homónima *AIMMS*.

Todos los *AML* presentados anteriormente requieren que el usuario codifique su problema en un lenguaje de dominio específico con una sintaxis parecida a la de un lenguaje de programación, aunque simplificada, por lo que se requiere un aprendizaje. Además, todos los *AML* expuestos son propietarios y para su uso hay que adquirir costosas licencias. En este trabajo se pretende diseñar e implementar un *AML* que permita al usuario modelar su problema de forma más visual, evitando en la medida de lo posible que tenga que escribir código. El *AML* diseñado será publicado como software libre y no requerirá licencia para usarlo, podrá ser usado con resolutores propietarios o libres.

1.4. Objetivos

La hipótesis que se plantea se centra fundamentalmente en que los métodos de optimización bio-inspirados ofrecen grandes posibilidades para la resolución de problemas en el ámbito empresarial pero, debido a su complejidad intrínseca y también a sus particularidades a la hora de implementarlos y adaptarlos a cada uno de los problemas, no se ha conseguido extender su uso fuera del ámbito de la investigación convencional. Por ello, se plantea la necesidad de diseñar e implementar nuevas herramientas que ayuden a mejorar la aplicabilidad de estas técnicas de optimización. El gran reto es que no podemos diseñar una herramienta más a añadir a la lista de ya existentes, sino que debemos ser capaces de proporcionar una clara separación entre la definición del problema y la determinación del método de optimización a aplicar. También es necesario dar un salto cualitativo en lo que respecta a la sencillez de uso, tratando así de dejar atrás todo lo que implica complejas implementaciones o configuraciones a bajo nivel.

Es por ello que el principal objetivo de este trabajo es poder implementar una herramienta que permita una optimización a un nivel de descripción abstracto (no específico del problema) con el fin de aumentar la aplicabilidad de los algoritmos bio-inspirados y otras heurísticas. Para ello, es necesario definir una clara separación entre el problema y el método de solución a aplicar. De esta forma, se podría evitar la tarea de implementar código y diseñar representaciones de soluciones personalizadas a cada par *(problema, método)*, pues resulta bastante difícil y requiere de mucho tiempo y esfuerzo, especialmente para aquellos usuarios ajenos al área de conocimiento y que únicamente tienen interés por encontrar algún tipo de solución “admisible” para su problema.

En la fase de diseño es preciso partir de un meta-modelo. Este meta-modelo para problemas de optimización debe permitir capturar, en términos abstractos, las características

esenciales de un problema de optimización. El meta-modelo puede servir como forma de almacenamiento o modelado genérico de problemas de optimización, a partir de los cuales se pueden generar problemas de optimización reales, 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.

Para cumplir con los objetivos de este trabajo se han establecido las siguientes tareas:

- Toma de contacto: recopilación y estudio de problemas clásicos de optimización combinatoria, intentando resolverlos mediante frameworks y herramientas existentes con el fin de familiarizarse con estos últimos.
- Análisis de problemas: análisis de los problemas de la tarea anterior, identificando todas sus características (tipo de codificaciones, restricciones, funciones objetivo, etc.), con el fin de inferir un único meta-modelo que sea capaz de modelarlos completamente.
- Definición del DSL: diseño de un lenguaje especifico para la definición de funciones objetivo y restricciones en el meta-modelo.
- Desarrollo de la herramienta: creación de una herramienta que sea capaz de interpretar el meta-modelo y el DSL diseñado en la tarea anterior, la herramienta deberá proveer una base desde la cual poder implementar resolutores que puedan ejecutar algoritmos para resolver los problemas modelados con el meta-modelo.
- Desarrollo del resolutor de jMetal: desarrollo de un resolutor para la herramienta que use el framework jMetal.
- Creación de la documentación: creación de la documentación pertinente para el entendimiento de la arquitectura de la herramienta, DSL y meta-modelo, así como del resolutor implementado. Deberá proveer también de ejemplos.

Capítulo 2

Modelado de problemas de optimización combinatoria

En este capitulo se expone el meta-modelo propuesto para la definición de problemas de optimización combinatoria a un nivel de descripción abstracto.

Es importante aclarar que no se hace una distinción clara entre problema e instancia de problema, como sí se hace en la literatura científica. El meta-modelo propuesto incluye la definición del problema y los datos de la instancia a resolver en un mismo objeto, esto se ha hecho así para simplificar las interfaces y la gestión de las ejecuciones. Por lo tanto, cuando se habla de "problema" realmente se está haciendo referencia a la definición de un problema junto con los datos de la instancia a resolver.

2.1. Componentes del modelo

El meta-modelo consiste en 6 elementos: parámetros, variables, funciones objetivo, restricciones, clases y objetos.

```
interface ProblemInterface {
  /**
   * Name of the problem.
   */
  name: string;

  /**
   * Description of the problem (optional).
   */
  description?: string;

  /**
   * Parameters of the problem (optional).
   */
  parameters?: ParameterInterface[];

  /**
   * Variables of the problem.
   *
   * @minItems 1
   */
  variables: VariableInterface[];
```

```
23
24    /**
25     * Objective functions of the problem.
26     *
27     * @minItems 1
28     */
29    goals: GoalInterface[];
30
31    /**
32     * Constraints of the problem (optional).
33     */
34    constraints?: ConstraintInterface[];
35
36    /**
37     * Classes of the problem (optional).
38     */
39    classes?: ClassInterface[];
40
41    /**
42     * Instances of the classes of the problem (optional).
43     */
44    objects?: ObjectInterface[];
45  }
```

A continuación se explicará cada uno, usando para ello el problema de la mochila binaria como ejemplo. El problema de la mochila binaria, comúnmente abreviado por KP (del inglés Knapsack problem) es un problema de optimización combinatoria que modela una situación análoga a llenar una mochila, incapaz de soportar más de un peso determinado, con todo o parte de un conjunto de objetos, cada uno con un peso y valor específicos. Los objetos colocados en la mochila deben maximizar el valor total sin exceder el peso máximo.

Al final de este capitulo se expondrán otros ejemplos de modelos. La herramienta desarrollada usa el formato JSON para almacenar los modelos. Es por ello que los ejemplos que se presentan se muestran codificados en JSON.

2.1.1. Parámetros

Los parámetros no son más que constantes numéricas usadas para flexibilizar la definición del problema, estas constantes se podrán usar en la definición de variables, funciones objetivo, restricciones y clases. Los parámetros están definidos por un nombre opcional, el símbolo que se usará para referenciarlo y el valor numérico, que puede ser entero o real:

```
1  interface ParameterInterface {
2    /**
3     * Name of the parameter (optional).
4     */
5    name?: string;
6
7    /**
8     * Symbol of the parameter, must be unique.
```

```
   */
  symbol: SymbolType;

  /**
   * Value of the parameter.
   */
  value: number;
}
```

Por ejemplo, para el problema de la mochila binaria se podría definir como parámetros el número de elementos disponibles para colocar en la mochila, y el peso máximo permitido:

```
{
  "parameters": [
    {
      "name": "Number of items",
      "symbol": "N",
      "value": 5
    },
    {
      "name": "Maximum weight",
      "symbol": "MaxWeight",
      "value": 80
    }
  ]
}
```

De esta forma, se podrá modificar el problema fácilmente, sin tener que cambiar otras partes del modelo, solo los valores de los parámetros.

2.1.2. Variables

Las variables son lo que queremos calcular u optimizar durante el proceso de búsqueda. Según su tipo pueden ser enteras o reales, y según su forma (*shape*) pueden ser individuales, vectores, vectores permutados o matrices. Las variables están definidas por un nombre opcional, el símbolo que se usará para referenciarla, el tipo al que pertenece, su forma (`shape`) y el rango de valores que acepta:

```
interface VariableInterface {
  /**
   * Name of the variable (optional).
   */
  name?: string;

  /**
   * Symbol of the variable, must be unique.
   */
  symbol: SymbolType;

  /**
   * Number set to which the variable belongs.
   */
  within: WithinType;
```

```
  /**
   * Shape of the variable (single, vector or matrix).
   */
  shape: Shape;

  /**
   * Range of the value/s of the variable (by default unbounded).
   */
  range?: Range;
}
```

`WithinType` es un tipo que acepta como valor `integers` o `reals`, mientras que `Shape` se define como:

```
interface SingleShape {
  type: 'single';
}

interface VectorShape {
  type: 'vector';
  isPermutation: boolean;
  size: Size;
}

interface MatrixShape {
  type: 'matrix';
  size: {
    rows: Size;
    columns: Size;
  };
}

type Shape = SingleShape | VectorShape | MatrixShape;
```

`Size` representa un tamaño que depende de un parámetro previamente definido (`ParameterizedSize`), o bien un tamaño fijo (`FixedSize`):

```
interface ParameterizedSize {
  /**
   * Defines if the size is a fixed number or a parameterized value.
   */
  fixed: false;

  /**
   * Size, can be an integer or a parameter of the problem.
   */
  value: SymbolType;
}

interface FixedSize {
  /**
   * Defines if the size is a fixed number or a parameterized value.
   */
  fixed: true;

```

```
  /**
   * Size, can be an integer or a parameter of the problem.
   *
   * @asType integer
   */
  value: number;
}

type Size = ParameterizedSize | FixedSize;
```

En el problema de la mochila binaria lo que nos interesa es obtener la lista de elementos a meter en la mochila que maximice el valor de los elementos sin sobrepasar un peso máximo. Para ello se puede usar un vector binario cuyo tamaño sea el número de elementos disponibles. Si en la posición del vector que representa a un elemento hay un 0 significará que ese elemento no se encuentra en la mochila, en caso de que haya un 1 ese elemento sí estará dentro. Se puede definir ese vector de la siguiente forma:

```
{
  "variables": [
    {
      "name": "Items in the knapsack",
      "symbol": "x",
      "within": "integers",
      "range": {
        "lowerBound": 0,
        "upperBound": 1
      },
      "shape": {
        "type": "vector",
        "isPermutation": false,
        "size": {
          "fixed": false,
          "value": "N"
        }
      }
    }
  ]
}
```

Tal y como se puede apreciar, para definir una variable binaria, como este caso, se debe establecer el tipo de la variable a entera y el rango a [0, 1]. En este ejemplo también se observa el uso de un parámetro para definir el tamaño de una variable. De esta forma el tamaño del vector quedará vinculado al valor del parámetro que define el número de elementos disponibles.

2.1.3. Clases

El concepto de clase usado en el meta-modelo es similar al concepto de clase de los lenguajes de programación orientados a objetos: una clase define la estructura de los objetos que forman parte de la instancia del problema a resolver. Las clases están definidas por un nombre opcional, el símbolo que se usará para referenciarla y una lista de atributos:

```
interface ClassInterface {
  /**
   * Name of the class (optional).
   */
  name?: string;

  /**
   * Symbol of the class, must be unique.
   */
  symbol: SymbolType;

  /**
   * Attributes of the class.
   *
   * @minItems 1
   */
  attributes: ClassAttribute[];
}
```

`ClassAttribute` representa a un atributo de la clase, cuyo tipo siempre es un número real o una cadena de caracteres (no hace falta explicitarlo):

```
interface ClassAttribute {
  /**
   * Name of the attribute (optional).
   */
  name?: string;

  /**
   * Symbol of the attribute, must be unique.
   */
  symbol: SymbolType;
}
```

En el caso del problema de la mochila binaria se tiene, conceptualmente, una mochila y una serie de objetos. El único parámetro necesario para modelar la mochila es su peso máximo, para el cual se ha usado el parámetro `MaxWeight`. Para modelar cada objeto necesitamos su nombre, valor y peso, como se van a modelar varios objetos lo más conveniente es crear una clase que los represente y luego crear las instancias correspondientes, la clase se podría definir de la siguiente forma:

```
{
  "classes": [
    {
      "name": "Item",
      "symbol": "item",
      "attributes": [
        {
          "name": "Name",
          "symbol": "name"
        },
        {
          "name": "Value",
          "symbol": "value"
        },
```

```
        {
          "name": "Weight",
          "symbol": "weight"
        }
      ]
    }
  ]
}
```

2.1.4. Objetos

Los objetos, al igual que en los lenguajes de programación orientados a objetos, son instancias de clases. Un objeto debe especificar la clase de la que es instancia y darle valor a cada uno de los atributos definidos en ella:

```
interface ObjectInterface {
  /**
   * Class of the object.
   */
  class: string;

  /**
   * Attributes of the object.
   *
   * @minItems 1
   */
  attributes: ObjectAttribute[];
}
```

En este caso, `ObjectAttribute` es un atributo del objeto con su valor correspondiente:

```
interface ObjectAttribute {
  /**
   * Symbol of the attribute.
   */
  attribute: string;

  /**
   * Value of the attribute.
   */
  value: string | number;
}
```

Para el problema de la mochila binaria se define la clase `Item` con tres atributos: `name`, `value` y `weight`. Además, es necesario definir los objetos que implementan esa clase, es decir, los elementos que podrán ser seleccionados para ser introducidos en la mochila. En este ejemplo se tiene un total de 5 elementos distintos (como se definió con el parámetro `N`):

```
{
  "objects": [
    {
      "class": "item",
```

```
      "attributes": [
        {
          "attribute": "name",
          "value": "Item 1"
        },
        {
          "attribute": "value",
          "value": 33
        },
        {
          "attribute": "weight",
          "value": 15
        }
      ]
    },
    {
      "class": "item",
      "attributes": [
        {
          "attribute": "name",
          "value": "Item 2"
        },
        {
          "attribute": "value",
          "value": 24
        },
        {
          "attribute": "weight",
          "value": 20
        }
      ]
    },
    {
      "class": "item",
      "attributes": [
        {
          "attribute": "name",
          "value": "Item 3"
        },
        {
          "attribute": "value",
          "value": 36
        },
        {
          "attribute": "weight",
          "value": 17
        }
      ]
    },
    {
      "class": "item",
      "attributes": [
        {
          "attribute": "name",
          "value": "Item 4"
        },
        {
```

```
          "attribute": "value",
          "value": 37
        },
        {
          "attribute": "weight",
          "value": 8
        }
      ]
    },
    {
      "class": "item",
      "attributes": [
        {
          "attribute": "name",
          "value": "Item 5"
        },
        {
          "attribute": "value",
          "value": 12
        },
        {
          "attribute": "weight",
          "value": 31
        }
      ]
    }
  ]
}
```

2.1.5. Funciones objetivo

Las funciones objetivo representan aquello que queremos optimizar. Estas funciones están definidas por un nombre opcional, el sentido de la función objetivo (maximizar o minimizar), la expresión matemática escrita en *ProdefLang* (ver capítulo 4) y un peso opcional:

```
interface GoalInterface {
  /**
   * Name of the objective function (optional).
   */
  name?: string;

  /**
   * Sense of the objective function (minimize or maximize).
   */
  sense: GoalSense;

  /**
   * Expression of the objective function in prodef syntax.
   *
   * For example: sum x[i]*item[i].value over i=(1:N)
   */
  expression: string;

  /**
```

```
    * Weight of the objective function for single objective resolvers (optional).
    *
    * @minimum 0
    * @maximum 1
    */
  weight?: number;
}
```

En el problema de la mochila binaria, como ya se ha comentado, el objetivo es maximizar el valor de los objetos introducidos en la mochila. Por lo tanto, se puede definir la siguiente función objetivo:

```
{
  "goals": [
    {
      "name": "Maximize the value of the items",
      "sense": "maximize",
      "expression": "sum x[i]*item[i].value over i=(1:N)"
    }
  ]
}
```

Tal y como se puede observar, la expresión `sum x[i]*item[i].value over i=(1:N)` hace uso de los símbolos previamente definidos en las variables, clases y parámetros. En este caso se trata de maximizar el sumatorio con i tomando los valores [1, N] de la multiplicación del valor del vector x (al ser binario será 1 o 0) con el valor del objeto correspondiente (el que posee el mismo índice i). Esta expresión da la suma del valor de todos los objetos que están dentro de la mochila (tienen un 1 en la posición del vector x asociada a ese objeto). En la Subsección 3.4.1 se detalla la sintaxis del lenguaje *ProdefLang* usado para las expresiones.

2.1.6. Restricciones

Las restricciones son expresiones *booleanas* (pueden ser verdaderas o falsas) que determinan si una solución es factible o no, están definidas por un nombre opcional y la propia expresión de la restricción escrita en *ProdefLang*:

```
interface ConstraintInterface {
  /**
   * Name of the constraint (optional).
   */
  name?: string;

  /**
   * Expression of the constraint in prodef syntax.
   *
   * For example: sum x[i]*food[i].volume over i=1:N <= Vmax
   */
  expression: string;
}
```

Para que una solución sea factible en el problema de la mochila binaria se tiene que dar la condición de que la suma de los pesos de todos los objetos que están dentro de

la mochila no supere la capacidad máxima de la misma (`MaxWeight`). Para definir esta restricción se puede usar el siguiente fragmento:

```
{
  "constraints": [
    {
      "name": "Limit the total weight of the items in the knapsack",
      "expression": "sum x[i]*item[i].weight over i=(1:N) <= MaxWeight"
    }
  ]
}
```

Lo que se hace con la restricción anterior es calcular el peso total de los objetos dentro de la mochila de forma similar a como se calcula el valor total de los objetos en la función objetivo. A continuación, se añade el operador de comparación $<=$ para convertir la expresión en una expresión *booleana* que resultará verdadera (solución factible) si la suma de pesos es inferior o igual a la capacidad máxima de la mochila, y falsa en caso contrario (solución no factible).

2.2. Ejemplo: Modelo de una instancia de TSP sencilla

A continuación, se muestra un ejemplo para el problema del viajante de comercio (*Travelling Salesman Problem, TSP*). El problema consiste en minimizar la distancia recorrida por un vendedor ambulante que desea visitar todas las ciudades, pasando por cada una solo una vez y terminando en la misma ciudad desde la que empezó. Para este ejemplo se definirá solo 4 ciudades.

2.2.1. Parámetros

Como único parámetro se define `N`, que representa el número de ciudades de nuestro problema, en este caso 4:

```
{
  "parameters": [
    {
      "name": "Number of cities",
      "symbol": "N",
      "value": 4
    }
  ]
}
```

2.2.2. Variables

Puesto que el objetivo es obtener el orden de las ciudades a visitar, se define como variable un vector de enteros permutado. Cada ciudad es representada por un número que va del 1 al 4 (inclusive). Se puede usar el parámetro `N` para asignarle el tamaño al vector:

```
{
  "variables": [
    {
      "name": "Visited cities",
      "symbol": "city",
      "within": "integers",
      "shape": {
        "type": "vector",
        "isPermutation": true,
        "size": {
          "fixed": false,
          "value": "N"
        }
      }
    }
  ]
}
```

2.2.3. Clases

Para simplificar el ejemplo, en vez de definir cada ciudad con unas coordenadas y calcular posteriormente (en la función objetivo) la distancia a las otras ciudades, se va a definir una clase `distance` que contiene la distancia precalculada a cada una de las otras ciudades, de tal forma que cada ciudad tendrá su objeto `distance` asociado:

```
{
  "classes": [
    {
      "name": "Distances between cities",
      "symbol": "distance",
      "attributes": [
        {
          "name": "Distance with city 1",
          "symbol": "dist_c1"
        },
        {
          "name": "Distance with city 2",
          "symbol": "dist_c2"
        },
        {
          "name": "Distance with city 3",
          "symbol": "dist_c3"
        },
        {
          "name": "Distance with city 4",
          "symbol": "dist_c4"
        }
      ]
    }
  ]
}
```

2.2.4. Objetos

Con 4 ciudades se debería crear 4 instancias de la clase `distance`, especificando la distancia a cada una de las demás ciudades:

```
{
  "objects": [
    {
      "class": "distance",
      "attributes": [
        {
          "attribute": "dist_c1",
          "value": 0
        },
        {
          "attribute": "dist_c2",
          "value": 12
        },
        {
          "attribute": "dist_c3",
          "value": 25
        },
        {
          "attribute": "dist_c4",
          "value": 17
        }
      ]
    },
    {
      "class": "distance",
      "attributes": [
        {
          "attribute": "dist_c1",
          "value": 12
        },
        {
          "attribute": "dist_c2",
          "value": 0
        },
        {
          "attribute": "dist_c3",
          "value": 35
        },
        {
          "attribute": "dist_c4",
          "value": 8
        }
      ]
    },
    {
      "class": "distance",
      "attributes": [
        {
          "attribute": "dist_c1",
          "value": 25
        },
        {
```

```
          "attribute": "dist_c2",
          "value": 35
        },
        {
          "attribute": "dist_c3",
          "value": 0
        },
        {
          "attribute": "dist_c4",
          "value": 11
        }
      ]
    },
    {
      "class": "distance",
      "attributes": [
        {
          "attribute": "dist_c1",
          "value": 17
        },
        {
          "attribute": "dist_c2",
          "value": 8
        },
        {
          "attribute": "dist_c3",
          "value": 11
        },
        {
          "attribute": "dist_c4",
          "value": 0
        }
      ]
    }
  ]
}
```

2.2.5. Funciones objetivo

El objetivo es minimizar la distancia recorrida, dicha distancia se podría calcular haciendo el sumatorio de la distancia entre una ciudad y su siguiente en el vector `city`, y sumándole al final la distancia entre la última ciudad y la primera:

```
{
  "goals": [
    {
      "name": "Minimize the distance traveled",
      "sense": "minimize",
      "expression": "sum distance[city[i], city[i+1]] over i=(1:N-1) + distance[city[N],
      ↪  city[1]]",
      "weight": 1
    }
  ]
}
```

2.2.6. Restricciones

Al haber usado como variable un vector permutado no es necesario definir ninguna restricción, la condición de que se visiten todas las ciudades una sola vez ya se cumple al tratarse de una permutación.

2.3. Ejemplo: Modelo de una instancia del problema de dietas

A continuación, se muestra un ejemplo para un problema de dietas. El problema de dietas consiste en obtener un conjunto de alimentos con sus cantidades asociadas cuya suma de cada nutriente (calorías, vitamina A, proteínas, etc.) se encuentre entre un rango de valores aceptables y el volumen total de los alimentos no supere un volumen máximo determinado, el objetivo es minimizar el coste total de los alimentos seleccionados.

2.3.1. Parametros

Como parámetros se define la cantidad total de alimentos definidos (`N`) y el volumen máximo (`Vmax`):

```
{
  "parameters": [
    {
      "name": "Number of foods",
      "symbol": "N",
      "value": 5
    },
    {
      "name": "Maximum volume",
      "symbol": "Vmax",
      "value": 40
    }
  ]
}
```

2.3.2. Variables

Una lista de enteros cuyos elementos representen las cantidades de cada alimento que se tienen que consumir para minimizar el coste, cumpliendo con todas las restricciones, codifica la solución al problema. A continuación se define la variable `x` de tipo vector, y tamaño asociado al parámetro `N`, que representará la solución:

```
{
  "variables": [
    {
      "name": "Number of servings",
      "symbol": "x",
      "within": "integers",
      "range": {
        "lowerBound": 0,
```

```
        "upperBound": "Infinity"
      },
      "shape": {
        "type": "vector",
        "isPermutation": false,
        "size": {
          "fixed": false,
          "value": "N"
        }
      }
    }
  ]
}
```

2.3.3. Clases

En este problema tenemos, conceptualmente, una serie de alimentos, cada uno con sus propiedades (nombre, coste y valores nutricionales) y unos limites (inferiores y superiores) para cada valor nutricional. Estos objetos se pueden modelar con las clases `food` y `nutrientsLimit`:

```
{
  "classes": [
    {
      "name": "Food",
      "symbol": "food",
      "attributes": [
        {
          "name": "Name",
          "symbol": "name"
        },
        {
          "name": "Cost",
          "symbol": "cost"
        },
        {
          "name": "Volume",
          "symbol": "volume"
        },
        {
          "name": "Calories",
          "symbol": "cal"
        },
        {
          "name": "Carbohydrates",
          "symbol": "carbo"
        },
        {
          "name": "Proteins",
          "symbol": "proteins"
        },
        {
          "name": "Iron",
          "symbol": "iron"
        }
```

```
      ]
    },
    {
      "name": "Nutrients limit",
      "symbol": "nutrientsLimit",
      "attributes": [
        {
          "name": "Name",
          "symbol": "name"
        },
        {
          "name": "Minimum",
          "symbol": "minimum"
        },
        {
          "name": "Maximum",
          "symbol": "maximum"
        }
      ]
    }
  ]
}
```

2.3.4. Objetos

A continuación se definen las instancias de las clases declaradas anteriormente, en este caso se instancian 5 alimentos, y como se especifican 4 valores nutricionales (calorías, carbohidratos, proteínas y hierro) harán falta 4 instancias de `nutrientsLimit`:

```
{
  "objects": [
    {
      "class": "food",
      "attributes": [
        {
          "attribute": "name",
          "value": "Cheeseburger"
        },
        {
          "attribute": "cost",
          "value": 1.84
        },
        {
          "attribute": "volume",
          "value": 4
        },
        {
          "attribute": "cal",
          "value": 510
        },
        {
          "attribute": "carbo",
          "value": 34
        },
        {
```

```
          "attribute": "proteins",
          "value": 28
        },
        {
          "attribute": "iron",
          "value": 20
        }
      ]
    },
    {
      "class": "food",
      "attributes": [
        {
          "attribute": "name",
          "value": "Ham sandwich"
        },
        {
          "attribute": "cost",
          "value": 2.19
        },
        {
          "attribute": "volume",
          "value": 7.5
        },
        {
          "attribute": "cal",
          "value": 370
        },
        {
          "attribute": "carbo",
          "value": 35
        },
        {
          "attribute": "proteins",
          "value": 24
        },
        {
          "attribute": "iron",
          "value": 20
        }
      ]
    },
    {
      "class": "food",
      "attributes": [
        {
          "attribute": "name",
          "value": "Hamburger"
        },
        {
          "attribute": "cost",
          "value": 1.84
        },
        {
          "attribute": "volume",
          "value": 3.5
        },
```

```
        {
          "attribute": "cal",
          "value": 500
        },
        {
          "attribute": "carbo",
          "value": 42
        },
        {
          "attribute": "proteins",
          "value": 25
        },
        {
          "attribute": "iron",
          "value": 20
        }
      ]
    },
    {
      "class": "food",
      "attributes": [
        {
          "attribute": "name",
          "value": "Fish sandwich"
        },
        {
          "attribute": "cost",
          "value": 1.44
        },
        {
          "attribute": "volume",
          "value": 5
        },
        {
          "attribute": "cal",
          "value": 370
        },
        {
          "attribute": "carbo",
          "value": 38
        },
        {
          "attribute": "proteins",
          "value": 14
        },
        {
          "attribute": "iron",
          "value": 10
        }
      ]
    },
    {
      "class": "food",
      "attributes": [
        {
          "attribute": "name",
          "value": "Lowfat milk"
```

```
        },
        {
          "attribute": "cost",
          "value": 0.6
        },
        {
          "attribute": "volume",
          "value": 8
        },
        {
          "attribute": "cal",
          "value": 110
        },
        {
          "attribute": "carbo",
          "value": 12
        },
        {
          "attribute": "proteins",
          "value": 9
        },
        {
          "attribute": "iron",
          "value": 0
        }
      ]
    },
    {
      "class": "nutrientsLimit",
      "attributes": [
        {
          "attribute": "name",
          "value": "Calories"
        },
        {
          "attribute": "minimum",
          "value": 2000
        },
        {
          "attribute": "maximum",
          "value": "Infinity"
        }
      ]
    },
    {
      "class": "nutrientsLimit",
      "attributes": [
        {
          "attribute": "name",
          "value": "Carbohydrates"
        },
        {
          "attribute": "minimum",
          "value": 350
        },
        {
          "attribute": "maximum",
```

```
          "value": 375
        }
      ]
    },
    {
      "class": "nutrientsLimit",
      "attributes": [
        {
          "attribute": "name",
          "value": "Proteins"
        },
        {
          "attribute": "minimum",
          "value": 55
        },
        {
          "attribute": "maximum",
          "value": "Infinity"
        }
      ]
    },
    {
      "class": "nutrientsLimit",
      "attributes": [
        {
          "attribute": "name",
          "value": "Iron"
        },
        {
          "attribute": "minimum",
          "value": 100
        },
        {
          "attribute": "maximum",
          "value": "Infinity"
        }
      ]
    }
  ]
}
```

2.3.5. Funciones objetivo

El objetivo es minimizar el coste de los alimentos, para ello se podría hacer un sumatorio del coste del alimento (`food[i].cost`) multiplicado por su cantidad (`x[i]`):

```
{
  "goals": [
    {
      "name": "Minimize cost of servings",
      "sense": "minimize",
      "expression": "sum x[i]*food[i].cost over i=(1:N)"
    }
  ]
}
```

2.3.6. Restricciones

La primera restricción se encarga de que el volumen total de los alimentos no supere el valor máximo definido en el parámetro `Vmax`, para ello se realiza el sumatorio del volumen del alimento (`food[i].volume`) multiplicado por su cantidad (`x[i]`) y se compara con el valor de `Vmax`.

El resto de restricciones se ocupan de que los valores nutricionales de los alimentos seleccionados entren dentro del rango valido definido por las instancias de `nutrientsLimit`, para ello realiza el sumatorio del valor nutricional correspondiente del alimento con su cantidad y lo compara con el mínimo y máximo valor que puede tomar ese nutriente.

```
{
  "constraints": [
    {
      "name": "Limit the volume of food consumed",
      "expression": "sum x[i]*food[i].volume over i=(1:N) <= Vmax"
    },
    {
      "name": "Limit nutrient consumption of calories",
      "expression": "nutrientsLimit[1].minimum <= sum x[i]*food[i].cal over i=(1:N) <=
      ↪  nutrientsLimit[1].maximum"
    },
    {
      "name": "Limit nutrient consumption of carbohydrates",
      "expression": "nutrientsLimit[2].minimum <= sum x[i]*food[i].carbo over i=(1:N) <=
      ↪  nutrientsLimit[2].maximum"
    },
    {
      "name": "Limit nutrient consumption of proteins",
      "expression": "nutrientsLimit[3].minimum <= sum x[i]*food[i].proteins over i=(1:N)
      ↪  <= nutrientsLimit[3].maximum"
    },
    {
      "name": "Limit nutrient consumption of iron",
      "expression": "nutrientsLimit[4].minimum <= sum x[i]*food[i].iron over i=(1:N) <=
      ↪  nutrientsLimit[4].maximum"
    }
  ]
}
```

Es posible unir las restricciones de los valores nutricionales en una sola haciendo uso de una expresión `assert`:

```
{
  "name": "Limit nutrient consumption for each nutrient",
  "expression": "assert nutrientsLimit[j].minimum <= sum x[i]*food[i, 3+j] over i=(1:N)
  ↪  <= nutrientsLimit[j].maximum for all j=(1:4)"
}
```

Capítulo 3

Prodef: herramienta para el meta-modelado de problemas

En este capítulo se presenta el diseño y arquitectura de la herramienta desarrollada y se explican los módulos que la componen.

3.1. Diseño y arquitectura de la herramienta

La herramienta creada, a la que a partir de ahora nos referiremos a ella por su nombre, **Prodef**, ha sido diseñada basándose en el concepto de modularidad, en la Figura 3.1 se muestra el logo de la herramienta. *Prodef* está formada por un conjunto de módulos escritos en *TypeScript* que interactúan entre ellos. En la herramienta se distinguen dos tipos de módulos: los módulos que forman parte del núcleo de la herramienta, y los módulos que expanden las capacidades y funcionalidades de la herramienta.

Figura 3.1: Logo de Prodef

En la Figura 3.2 se puede observar la arquitectura de *Prodef*. La figura muestra cómo interactúan los distintos módulos, que se presentarán en las secciones posteriores.

Los módulos que forman parte del núcleo de *Prodef* son fijos, no están diseñados para ser fácilmente reemplazables y su función es proporcionar la base a partir de la cual poder crear módulos específicos que expandan *Prodef*. Los módulos del núcleo proporcionan:

- Interfaces comunes (usadas por más de un módulo) e interfaces que definen el meta-modelo para los problemas de optimización.

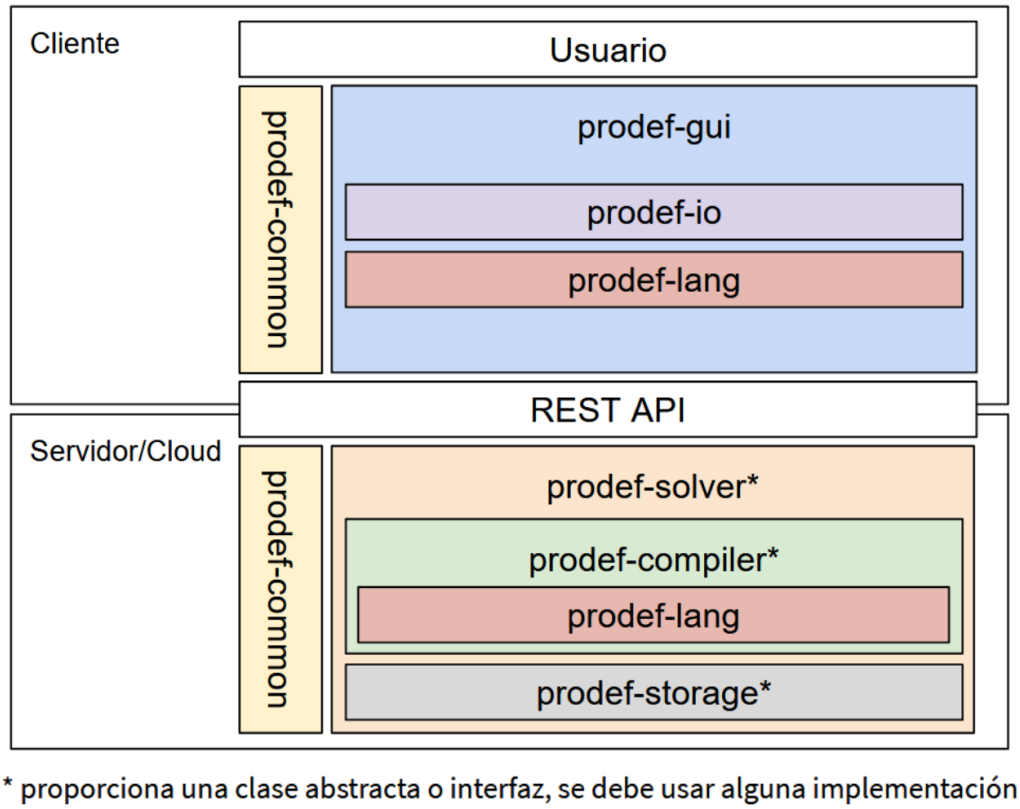

Figura 3.2: Arquitectura de Prodef

- Métodos para crear, modificar, consultar, serializar y deserializar problemas y soluciones.
- La definición e implementación del DSL *ProdefLang* usado para las funciones objetivo y restricciones, obteniendo un AST de las expresiones escritas en este lenguaje.
- Interfaces y clases abstractas para crear "compiladores" que sean capaces de traducir el AST de una expresión escrita en *ProdefLang* a código ejecutable escrito en algún lenguaje de programación.
- Interfaces para la implementación de "drivers" de almacenamiento (*storage*).
- Interfaces para la implementación de resolutores.
- Una implementación de servidor web que define la API REST que deberán tener todos los resolutores.

3.2. prodef-common

prodef-common es el módulo que proporciona las interfaces comunes, necesarias para que el resto de módulos se puedan comunicar entre sí sin problemas, además de las interfaces que definen el meta-modelo para los problemas y las soluciones, y de métodos útiles para interactuar con los modelos. Este módulo exporta tipos, interfaces e implementaciones.

Los tipos exportados más importantes son:

- `VariableType`: tipo de variable, puede tomar los valores: `binary`, `integer`, `real` y `integerPermutation`.
- `ProblemType`: tipo de problema, es un alias del tipo `VariableType`.
- `Shape`: es una unión de las interfaces `SingleShape`, `VectorShape` y `MatrixShape`.
- `Size`: es una unión de las interfaces `FixedSize` y `ParameterizedSize`.
- `SymbolSource`: puede tomar los valores: `parameter`, `variable` o `class`.
- `SymbolType`: alias del tipo `string`.
- `WithinType`: conjunto numérico, puede tomar los valores: `integers` o `reals`.

Las interfaces y los tipos exportados forman la definición del meta-modelo para los problemas de optimización combinatoria y sus soluciones. La interfaz `ProblemInterface` hace de punto de entrada para el meta-modelo de los problemas, mientras que la interfaz denominada `SolutionInterface` cumple el mismo rol pero para las soluciones. Cuando se construye el módulo (se ejecuta la tarea `build`) se crean automáticamente dos esquemas JSON (ver Apéndice A) a partir de las dos interfaces mencionadas anteriormente. Estos esquemas especifican la estructura que debe tener un archivo JSON para considerar que es un problema, o solución, válida. Estos esquemas también se exportan.

Las implementaciones son clases que implementan las interfaces relativas al meta-modelo del problema y de la solución, les añaden funcionalidades útiles como la validación del modelo, el cálculo del tipo de problema (binario, entero, real o permutación de enteros), la obtención del mapa de símbolos, etc. Las clases de implementación exportadas son:

- `Class`: implementación de `ClassInterface`.
- `Constraint`: implementación de `ContraintInterface`.
- `Goal`: implementación de `GoalInterface`.
- `ObjectImpl`: implementación de `ObjectInterface`.
- `Parameter`: implementación de `ParameterInterface`.
- `Problem`: implementación de `ProblemInterface`.
- `Solution`: implementación de `SolutionInterface`.
- `Variable`: implementación de `VariableInterface`.

Este módulo está alojado en el repositorio *tfg-andres-calimero-prodef-common* dentro de la organización de Parallel Algorithms and Languages de la ULL (`https://github.com/PAL-ULL/tfg-andres-calimero-prodef-common`).

Para una documentación técnica más detallada se puede recurrir a la documentación auto-generada en la página de GitHub pages desplegada asociada al repositorio (`https://pal-ull.github.io/tfg-andres-calimero-prodef-common/`).

3.3. prodef-io

prodef-io es un pequeño módulo que proporciona funciones para deserializar problemas y soluciones, hace uso de los esquemas JSON exportados por prodef-common para validar un texto codificado como JSON y proporcionar errores entendibles por humanos en caso de que el texto no sea un problema o solución válida. Para validar el texto contra los esquemas JSON se usa el validador ajv (`https://github.com/ajv-validator/ajv`).

A modo de ejemplo se muestra un fragmento de código que, a partir de un string con el problema codificado en JSON, obtiene el objeto `Problem` que lo representa:

```
const validationResult = ProdefIO.loadProblem("{...}");
if (validationResult.valid) {
    const problem: Problem = validationResult.result;
} else {
    console.log(errors.join());
}
```

Este módulo está alojado en el repositorio “tfg-andres-calimero-prodef-io” dentro de la organización de Parallel Algorithms and Languages de la ULL (`https://github.com/PAL-ULL/tfg-andres-calimero-prodef-io`).

Para una documentación técnica más detallada se puede recurrir a la documentación auto-generada en la página de GitHub pages desplegada asociada al repositorio (`https://pal-ull.github.io/tfg-andres-calimero-prodef-io/`).

3.4. prodef-lang

prodef-lang es el módulo que define e implementa el lenguaje específico de dominio *ProdefLang* (ver Subsección 3.4.1), este lenguaje es una mezcla entre notación matemática y lenguaje de programación orientado a objetos, es usado para definir las funciones objetivo y las restricciones de los problemas.

Para la definición del lenguaje se ha creado una gramática usando la implementación de notación Backus-Naur extendida (EBNF) que hace la herramienta ANTLR (`https://github.com/antlr/antlr4`), esta gramática se puede encontrar en el Apéndice B.

La herramienta ANTLR permite generar un parseador de lenguaje a partir de su gramática, en este módulo se usa ANTLR, junto con el paquete *antlr4ts*, para generar en tiempo de construcción (build) el código TypeScript del parseador del lenguaje *ProdefLang*, este parseador genera un árbol de parseo a partir de un texto escrito en *ProdefLang*. El árbol de parseo es recorrido siguiendo el patrón `visitor` y transformado en un árbol de sintaxis abstracta (AST). El AST será usado por los compiladores de *Prodef* para generar código ejecutable. Este módulo exporta la clase `Parser`, cuyo método `parse` es el encargado de generar el AST a partir de la expresión en texto. En la Figura 3.3 se muestra el diagrama de clases de los nodos del AST, el nodo `RootASTNode` es el nodo raíz del AST.

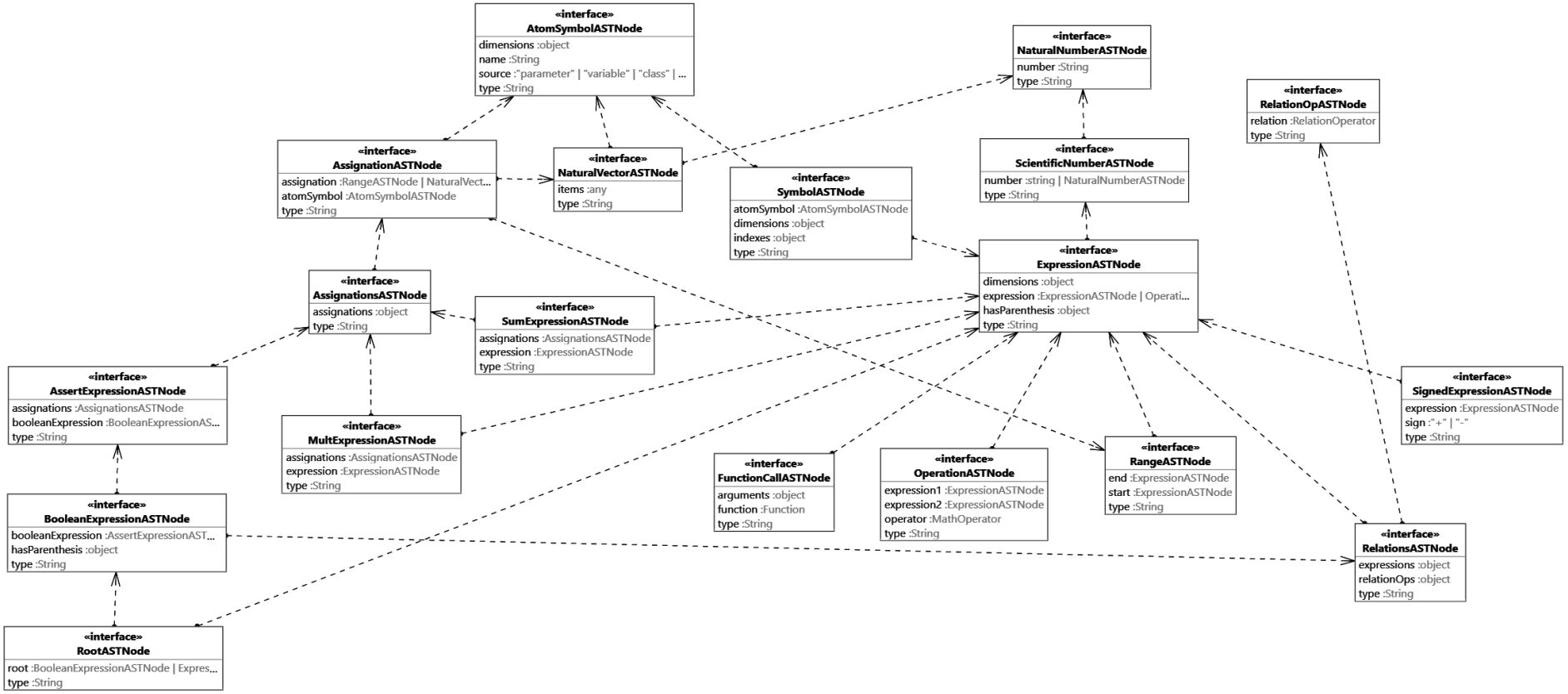

Figura 3.3: Diagrama de clases de los nodos del AST

ProdefLang implementa soporte para constantes y funciones. Mediante el objeto `constants` de la clase `Parser` se pueden consultar, crear y eliminar constantes en tiempo de ejecución, del mismo modo, mediante el objeto `functions` se puede consultar, crear y eliminar funciones, las funciones se definen mediante un nombre (que debe ser único) y una lista de argumentos con sus dimensiones. Por ejemplo, para añadir una nueva constante `pi` a un parseador se podría usar el siguiente código:

```
const parser = new Parser(problem);
parser.constants.addConstant({
      name: "pi",
      value: 3.14159265359,
});
```

y para añadir una función `cos` con un argumento de dimensión 0:

```
parser.functions.addFunction({
      name: "cos",
      arguments: [0],
});
```

De esta forma ya se podrá usar la constante `pi` y la función `cos` en las expresiones, es importante aclarar que la especificación de la implementación de la función `cos` es responsabilidad del compilador y no del parser:

```
const parsedExpression = parser.parse("cos(pi)");
if (parsedExpression.valid) {
    const ast: RootASTNode = parsedExpression.result;
}
```

`ProdefLang` implementa por defecto las constantes `pi`, `phi` y `e`, y las funciones `cos`, `sin` y `sqrt`, todas ellas con un solo argumento de dimensión 0.

A modo de ejemplo se muestra en la Figura 3.4 el árbol de parseo correspondiente a la expresión `sum x[i]^i over i=(1:N) + cos(PI + 0.5e-6)`, y en las figuras 3.5, 3.6 y 3.7 el AST generado a partir de ese árbol de parseo.

Este módulo está alojado en el repositorio "tfg-andres-calimero-prodef-lang" dentro de la organización de Parallel Algorithms and Languages de la ULL (`https://github.com/PAL-ULL/tfg-andres-calimero-prodef-lang`).

Para una documentación técnica más detallada se puede recurrir a la documentación auto-generada en la página de GitHub pages desplegada asociada al repositorio (`https://pal-ull.github.io/tfg-andres-calimero-prodef-lang/`).

3.4.1. Características del lenguaje ProdefLang

Junto a la herramienta se diseñó un lenguaje específico de dominio llamado *ProdefLang* cuya gramática se encuentra en el Apéndice B. Este lenguaje se creó con el objetivo de posibilitar la definición de funciones objetivo y restricciones en los modelos de forma sencilla. El lenguaje diseñado es una mezcla entre formulación matemática y lenguaje de programación orientado a objetos. *ProdefLang* debe permitir expresar cualquier tipo de función objetivo y restricción, manteniendo a su vez la simplicidad pues el objetivo es que cualquier persona sin conocimientos previos de programación pueda usarlo.

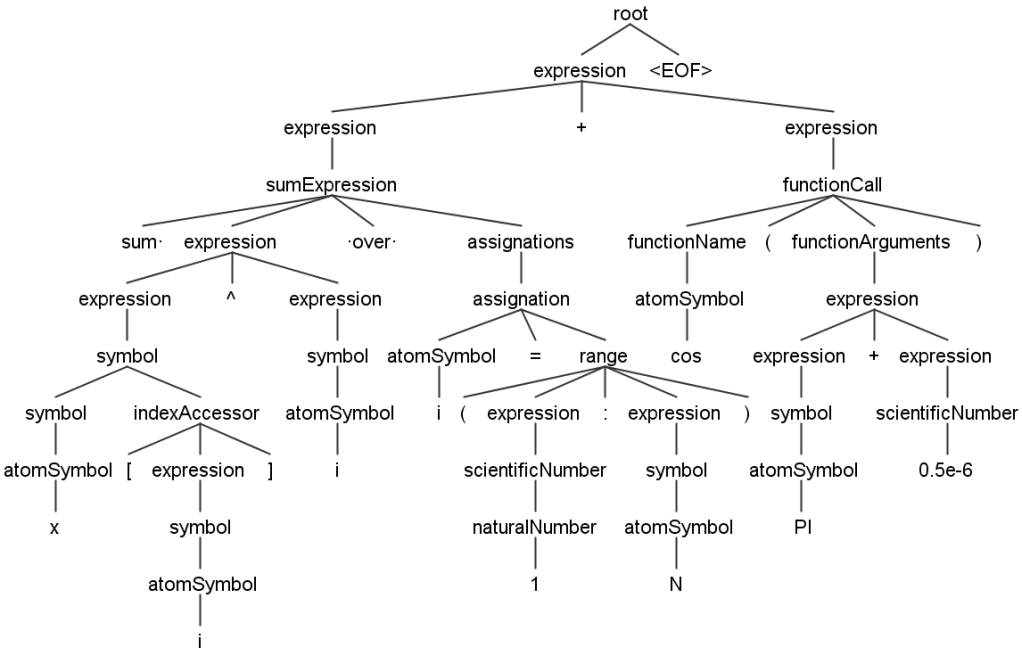

Figura 3.4: Árbol de parseo

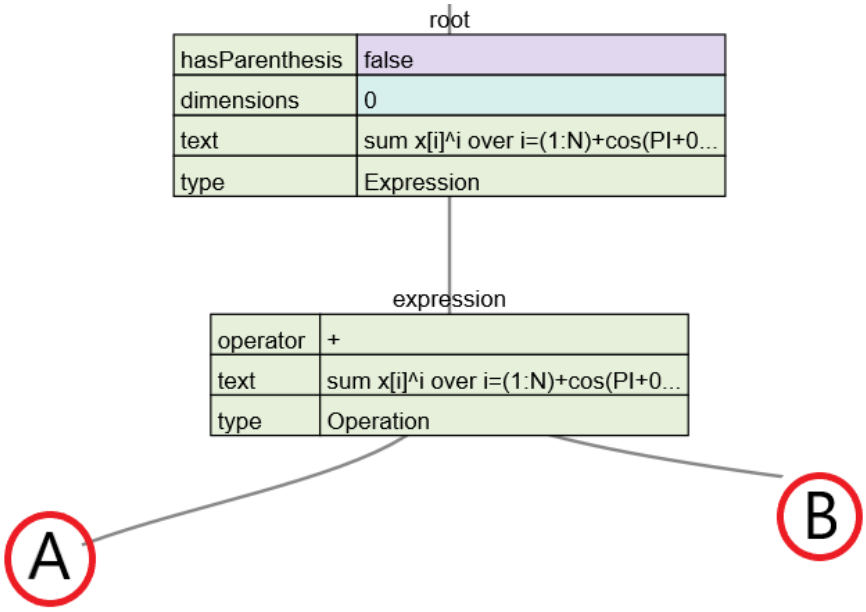

Figura 3.5: AST: raíz

Operadores aritméticos

ProdefLang tiene los operadores aritméticos básicos esperables en cualquier lenguaje de programación, incluyendo el operador $\wedge$ (potencia) que es menos común. En la Tabla 3.1 se muestran los operadores aritméticos disponibles.

Operadores de comparación

ProdefLang también dispone de los operadores de comparación usuales para poder especificar restricciones. En la Tabla 3.2 se muestran los operadores de comparación disponibles.

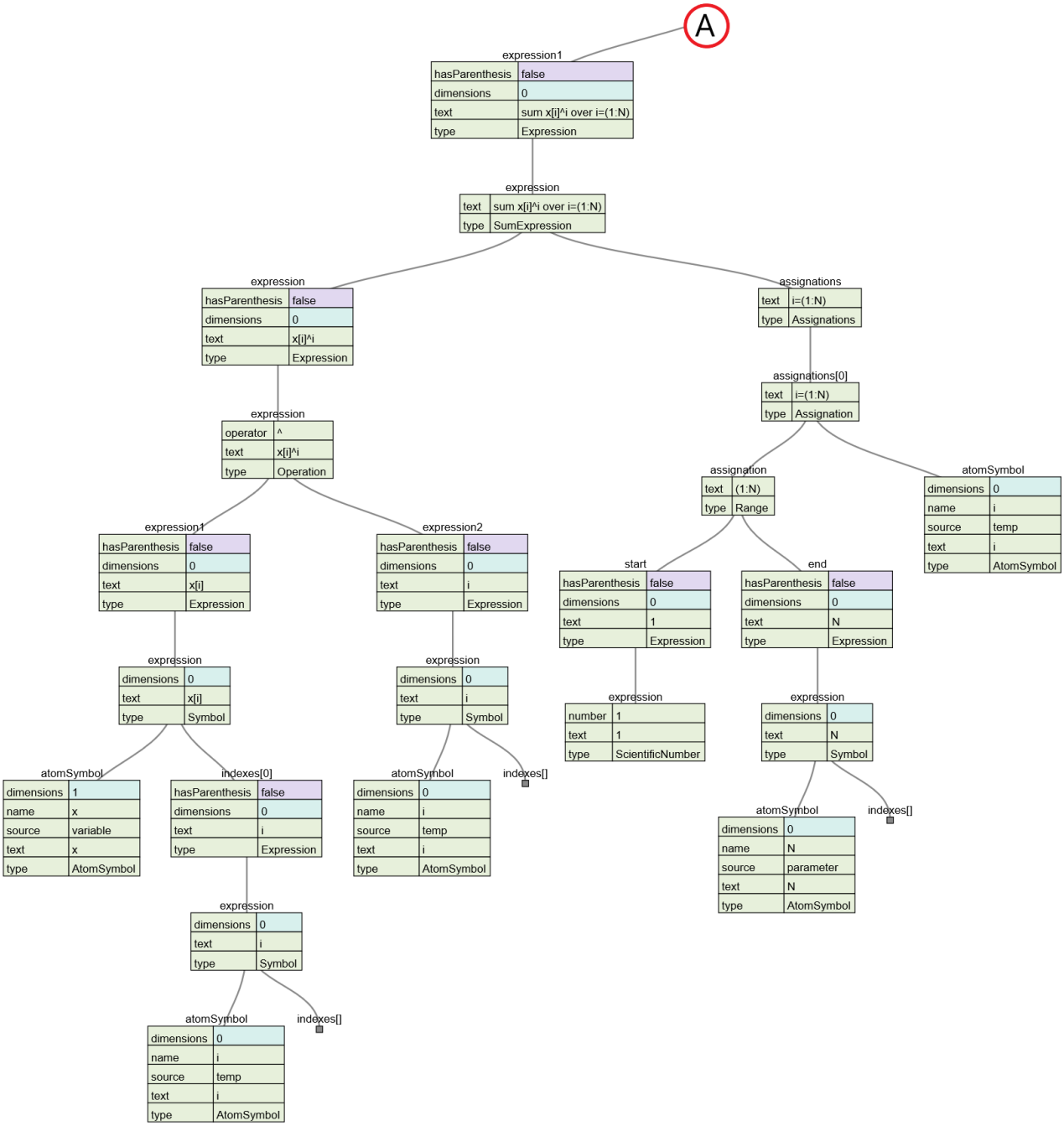

Figura 3.6: AST: rama A

Otros operadores

Además de los mencionados anteriormente, también tiene operadores de agrupación, asignación, acceso, vector y rango, como se muestra en la Tabla 3.3.

Accesores

Cada parámetro, variable y objeto definido en el modelo del problema tiene asociado un número que indica su dimensión. La dimensión de los parámetros siempre es 0 porque solo pueden ser valores numéricos simples. La dimensión de las variables depende del tipo de variable, las variables individuales, al igual que los parámetros, tienen dimensión 0, las variables de tipo vector tienen dimensión 1 y las de tipo matriz tienen dimensión 2.

Operador	Nombre	Sintaxis	Asociatividad	Ejemplo
+	Suma	a + b	Izquierda a derecha	4 + 5
-	Resta	a - b	Izquierda a derecha	10 - 6
*	Multiplicación	a * b	Izquierda a derecha	2 * 6
/	División	a / b	Izquierda a derecha	30 / 2
^	Potencia	a ^ b	Derecha a izquierda	10 ^ 3
+	Operación unaria más	+a	Derecha a izquierda	+7
-	Operación unaria menos	-a	Derecha a izquierda	-8

Tabla 3.1: Operadores aritméticos de *ProdefLang*

Operador	Nombre	Sintaxis	Asociatividad	Ejemplo
==	Igual	a == b	Izquierda a derecha	2 == 2
!=	Distinto	a != b	Izquierda a derecha	3 != 2
>	Mayor que	a > b	Izquierda a derecha	10 > 7
>=	Mayor o igual que	a >= b	Izquierda a derecha	8 >= 8
<	Menor que	a < b	Izquierda a derecha	3 < 15
<=	Menor o igual que	a <= b	Izquierda a derecha	6 <= 7

Tabla 3.2: Operadores de comparación de *ProdefLang*

Operador	Nombre	Sintaxis	Asociatividad	Ejemplo
( )	Agrupamiento	(x)	Ninguna	2 * (3 + 4)
[ ]	Vector, Índice	[...]	Ninguna	[2, 3, 1, 43] x[2]
:	Rango	(a:b)	Ninguna	(1:4)
=	Asignación	a = (rango | vector)	Derecha a izquierda	i = (1:5) i = [2, 3, 6]
.	Accesor de atributo	obj.atributo	Izquierda a derecha	x.peso

Tabla 3.3: Otros operadores de *ProdefLang*

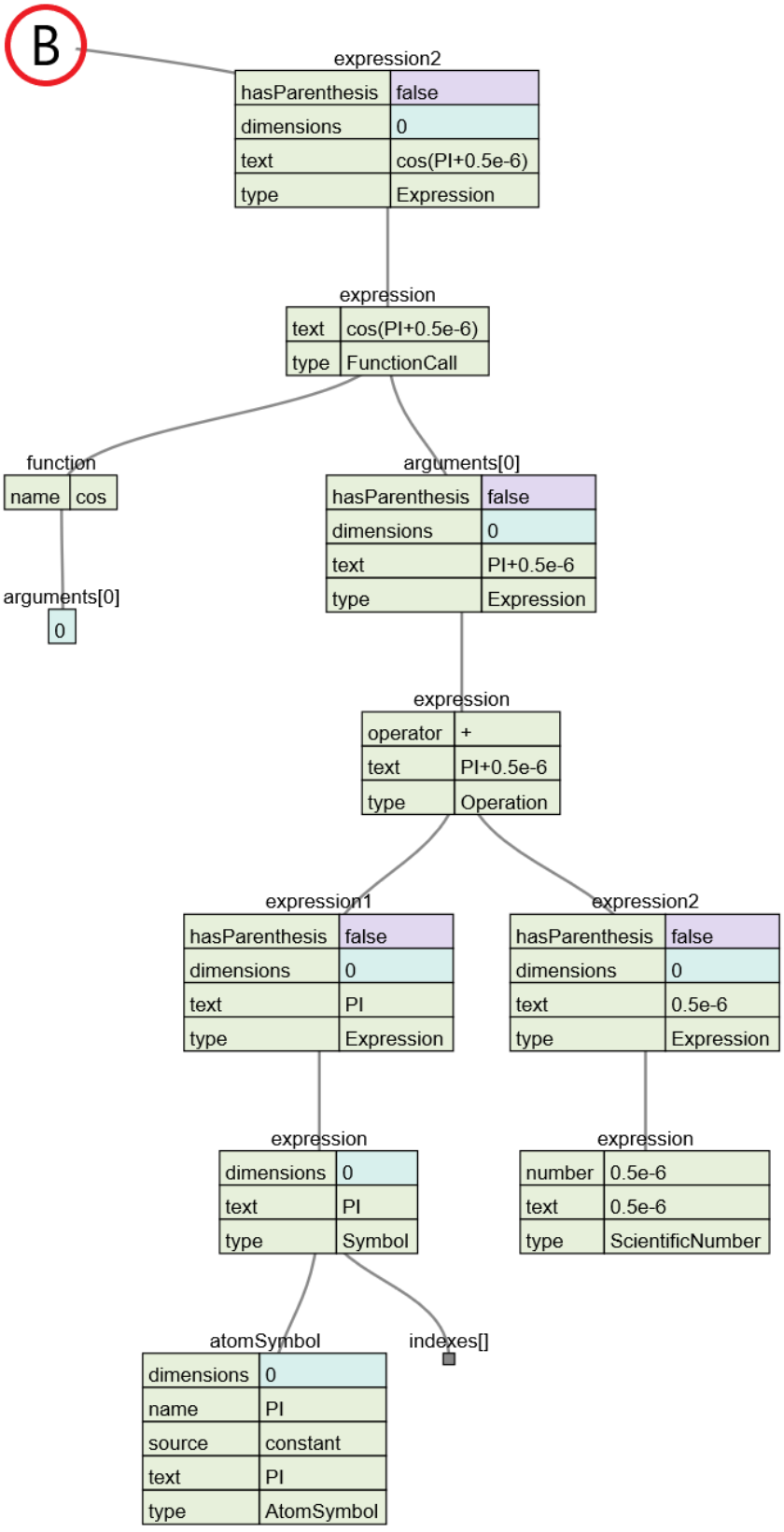

Figura 3.7: AST: rama B

Todos los objetos tienen dimensión 2, *ProdefLang* trata a los objetos como si fuesen una matriz de números reales, donde las filas representan las instancias y las columnas los atributos.

Para poder acceder en las funciones objetivo y restricciones a los valores de las variables y objetos, *ProdefLang* define el concepto de "Accesores", un *accesor* es un operador que permite acceder a un valor numérico a partir de un elemento con dimensión superior a 0. *ProdefLang* proporciona dos tipos de accesores: por índice y por nombre.

Accesores por índice

Los accesores por índice se pueden usar siempre, permiten acceder a un valor numérico a través de sus índices. Por ejemplo, si en el modelo se define una variable x de tipo vector, se podría acceder al primer elemento de la siguiente forma:

```
x[1]
```

Nótese que en *ProdefLang* el índice empieza en 1 en vez de en 0, como suele ser habitual en los lenguajes de programación.

Si se declarase una variable m de tipo matriz, se podría acceder al elemento en la segunda fila, tercera columna, de la siguiente forma:

```
m[2, 3]
```

En el caso de los objetos, si se declarase una clase `coche` con dos atributos: `peso` y `aceleracion`. Se podría acceder al atributo aceleración de la tercera instancia de la clase `coche` de la siguiente forma:

```
coche[3, 2]
```

En este caso también se podría usar un accesor por nombre. El orden de las instancias viene determinado por el orden de los objetos en el documento JSON del problema.

Accesores por nombre

En el caso de los objetos se puede usar el nombre del atributo para acceder a sus valores, por ejemplo, para la misma clase `coche` del ejemplo anterior, se podría acceder al atributo aceleración de la tercera instancia de la siguiente forma:

```
coche[3].aceleracion
```

Funciones

Con el objetivo de permitir modelar problemas complejos, *ProdefLang* soporta llamadas a funciones en sus expresiones, por defecto se declaran 3 funciones: cos (coseno), sin (seno) y sqrt (raíz cuadrada). Se pueden incorporar nuevas funciones al lenguaje en tiempo de ejecución tal y como se mostró en la Sección 3.4

Cada función acepta un número concreto de argumentos, cada argumento es de una dimensión concreta, y siempre devuelve un número real. Por ejemplo, se podría calcular el coseno de una variable individual x de la siguiente forma:

```
cos(x)
```

Al igual que en un lenguaje de programación, se puede pasar como argumento a una función una expresión que será resuelta antes de realizar la llamada, por ejemplo:

```
cos(3.14 + x)
```

Constantes

ProdefLang también soporta constantes, al igual que con las funciones hay 3 constantes definidas por defecto: pi (3.14159265359), phi (1.61803398874) y e (2.71828182845). También se pueden declarar nuevas constantes en tiempo de ejecución, tal y como se mostró en la Sección 3.4.

Para usar una constante en una expresión basta simplemente con referenciarla por su nombre:

```
cos(PI + x)
```

El nombre de las constantes es insensible a mayúsculas y minúsculas.

Estructuras de control

ProdefLang permite el uso de bucles a través de 3 tipos de sentencias: `sum`, `mult` y `assert`. Al contrario que la mayoría de lenguajes de programación, *ProdefLang* no implementa estructuras de control basadas en condicionales (if/else).

Sumatorio

La operación sumatorio es muy común en la definición de funciones objetivo y restricciones, en *ProdefLang* la sintaxis para realizar un sumatorio es la siguiente:

```
sum <expresión> over <variable>=(rango|vector)(,<variable>=(rango|vector))*
```

Por ejemplo, suponiendo una variable x de tipo vector y tamaño 5, se podría realizar el sumatorio de las siguientes formas:

```
sum x[i] over i=(1:5)
sum x[i] over i=[1,2,3,4,5]
```

En el caso de que x sea una matriz de tamaño 5 x 6, se podría realizar el sumatorio de las siguientes formas:

```
sum x[i,j] over i=(1:5), j=(1:6)
sum x[fila, columna] over columna=(1:6), fila=(1:5)
sum x[i,j] over i=(1:5), j=[1,2,3,4,5,6]
sum x[i,j] over i=[1,2,3,4,5], j=[1,2,3,4,5,6]
```

Multiplicatorio

ProdefLang implementa también la operación multiplicatorio, su sintaxis es idéntica al sumatorio, cambiando la palabra clave `sum` por `mult`:

```
mult <expresión> over <variable>=(rango|vector)(,<variable>=(rango|vector))*
```

Por ejemplo, suponiendo una variable x de tipo vector y tamaño 5, se podría realizar el multiplicatorio de las siguientes formas:

```
mult x[i] over i=(1:5)
mult x[i] over i=[1,2,3,4,5]
```

en el caso de que x sea una matriz de tamaño 5 x 6, se podría realizar el multiplicatorio de las siguientes formas:

```
mult x[i,j] over i=(1:5), j=(1:6)
mult x[fila, columna] over columna=(1:6), fila=(1:5)
mult x[i,j] over i=(1:5), j=[1,2,3,4,5,6]
mult x[i,j] over i=[1,2,3,4,5], j=[1,2,3,4,5,6]
```

Múltiples afirmaciones

A menudo las restricciones requieren que se comprueben condiciones sobre los elementos de un vector o matriz, para ello *ProdefLang* implementa la expresión de tipo `assert`. Su sintaxis es:

```
assert <expresión booleana> for all <variable>=(rango|vector)(,<variable>=(rango|vector))*
```

Por ejemplo, para definir una restricción que compruebe que todos los elementos de un vector x de tamaño 10 están en el rango [3, 7] se podría usar la siguiente expresión:

```
assert 3 <= x[i] <= 7 for all i=(1:10)
```

Esta restricción solo se validaría si se cumple la expresión booleana $3 \leq x[i] \leq 7$ para todo i en el rango [1:10].

3.5. prodef-compiler

prodef-compiler es el módulo que sirve de base para todas las implementaciones de compiladores de *Prodef*, exporta la clase abstracta `Compiler`, clase base de todos los compiladores. Además, también exporta la clase abstracta `CBasedCompiler`, que facilita en gran medida la implementación de compiladores para lenguajes cuya sintaxis está basada en el lenguaje C (como C++ o Java).

La clase `Compiler` recibe en su constructor el problema que se va a compilar y proporciona un método público `compile` que devuelve un objeto `CompiledProblem`, a este método se le pueden pasar una serie de opciones (para más información sobre las opciones se puede recurrir a la documentación técnica, el enlace se encuentra al final de esta sección). El objeto `CompiledProblem` contiene información sobre el problema (sus metadatos) y todas las expresiones de las funciones objetivo y restricciones compiladas al lenguaje correspondiente, así como las declaraciones de las variables, los parámetros, los objetos y el código de las funciones declaradas. En la figura 3.8 se muestra el diagrama de clases del problema compilado.

El resultado de compilar la expresión de una restricción son dos expresiones, la primera es la propia expresión booleana de la restricción, es decir, una expresión que devuelve un valor booleano en función de si la restricción se cumple o no, y la otra es la expresión de violación de la restricción (en *Prodef* a esta expresión se le llama `violationExpression`),

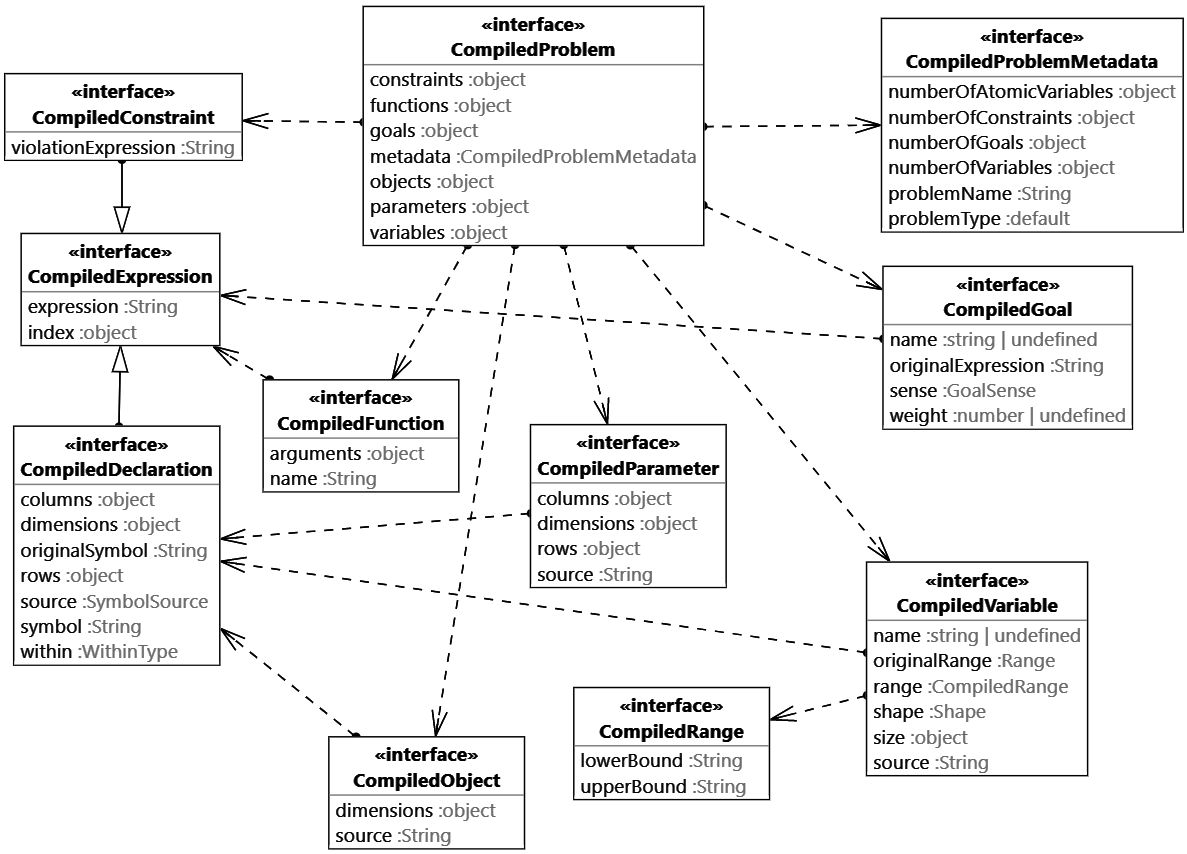

Figura 3.8: Diagrama de clases del problema compilado

esta expresión devuelve un número real que indica el grado de violación de la restricción, si el número es negativo indica que la restricción no se cumple, si es 0 o positivo indica que sí se cumple. Muchos algoritmos para la resolución de problemas de optimización se pueden beneficiar de conocer el grado de violación de cada restricción, es por eso que se ha incluido como parte de *Prodef*.

La clase `Compiler` crea y almacena una instancia de `Parser` para poder compilar las expresiones escritas en *ProdefLang*, se puede acceder de forma pública a la instancia con el getter `parser`, también se puede acceder directamente a los objetos `constants` y `functions` del parser, permitiendo consultar, eliminar y añadir constantes y funciones en tiempo de compilación. Como el compilador es el responsable de generar todo el código necesario para poder ejecutar las expresiones de las funciones objetivo y las restricciones, es también responsable de generar el código de todas las funciones usadas, y cómo se pueden declarar funciones en tiempo de ejecución `Compiler` proporciona el método `registerFunctionCompiler`, este método asocia la implementación de una función con su declaración, un ejemplo para un compilador de Java y la función `sin` sería el siguiente:

```
class SinFunctionCompiler extends FunctionCompiler {
    public compile(): string {
        return "double sin(double number) { return Math.sin(number); }";
    }
}

```

Método	Endpoint	Descripción	Resultados
GET	/capabilities	Devuelve las capacidades del resolutor	SolverCapabilities
POST	/executions	Crea una ejecución de un problema, el body de la petición debe contener un JSON que implemente la interfaz ExecutionRequest	ExecutionID
GET	/executions/:executionId	Devuelve el estado actual de una ejecución, se tiene que proporcionar la cabecera "Secret-Keyçon el valor de la clave secreta de la ejecución	ExecutionState
DELETE	/executions/:executionId	Elimina una ejecución, se tiene que proporcionar la cabecera "Secret-Keyçon el valor de la clave secreta de la ejecución	Vacío

Tabla 3.4: Definición de la API v1

```
const compiler = new JavaCompiler(problem);

// Declaración de la función
compiler.functions.addFunction({
    name: "sin",
    arguments: [0],
});

// Implementación de la función
compiler.registerFunctionCompiler(
    "sin",
    new SinFunctionCompiler()
);
```

Este módulo está alojado en el repositorio “tfg-andres-calimero-prodef-compiler” dentro de la organización de Parallel Algorithms and Languages de la ULL (`https://github.com/PAL-ULL/tfg-andres-calimero-prodef-compiler`).

Para una documentación técnica más detallada se puede recurrir a la documentación auto-generada en la página de GitHub pages desplegada asociada al repositorio (`https://pal-ull.github.io/tfg-andres-calimero-prodef-compiler/`).

3.6. prodef-solver

prodef-solver es el módulo encargado de definir la API REST y de proporcionar la base para todos los resolutores (`Solver`). Este módulo implementa un servidor web (`SolverSever`) 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 cuando sea requerido. En la Tabla 3.4 se muestra la definición de la API v1.

El servidor implementa un sistema de versionado semántico de la API, actualmente se implementa la versión v1, todos los endpoints tienen como prefijo la versión a la que

corresponden, por ejemplo, para obtener las capacidades del resolutor el endpoint final sería `/v1/capabilities`.

La interfaz `ExecutionRequest` define como debe ser el JSON proporcionado al endpoint `POST /executions` para solicitar al resolutor la ejecución del problema:

```
interface ExecutionConfiguration {
  maxComputingTime: number;
  maxNumberOfResults: number;
}

const defaultExecutionConfiguration: ExecutionConfiguration = {
  maxComputingTime: 10000,
  maxNumberOfResults: 10,
};

interface ExecutionRequest {
  problem: ProblemInterface;
  executionConfiguration: ExecutionConfiguration;
}
```

Se puede observar que, junto con el problema, se pueden especificar parámetros de configuración para la ejecución, como el máximo tiempo de computo (en milisegundos) y la cantidad máxima de soluciones ofrecidas.

El resolutor tiene asociadas unas capacidades que limitan el tipo de problemas que puede resolver, las capacidades se especifican mediante un objeto que implementa la interfaz `SolverCapabilities`:

```
interface SolverCapabilities {
  supportsMultipleVariableTypes?: boolean;
  supportsMultiplePermutations?: boolean;
}
```

Cuando se realiza una petición al endpoint `POST /executions` el servidor devuelve un objeto que implementa la interfaz `ExecutionID`:

```
interface ExecutionID {
  executionId: string;
  secretKey: string;
}
```

`executionId` es un UUID v1 que identifica de forma única al recurso de ejecución que se acaba de crear en el servidor, `secretKey` es un UUID v4 cuya función es proteger la información relativa a la ejecución, para poder recuperar información sobre el estado de la ejecución es necesario proveer tanto el `executionId` como la `secretKey`.

Los posible estados (`ExecutionState`) en los que puede estar una ejecución tras haber sido solicitada son los siguientes:

- Aceptado: se ha recibido la petición de ejecución correctamente, está a la espera de ser ejecutado.
- Rechazado: se ha rechazado la ejecución por criterio del resolutor, por ejemplo, por no soportar el tipo de problema proporcionado en la petición de ejecución.

- Ejecutándose: el resolutor está ejecutándose para obtener soluciones al problema proporcionado.
- Resuelto: el resolutor ha terminado su ejecución y se han obtenido soluciones.
- Fallido: ocurrió un error irrecuperable durante la ejecución del resolutor y no se han podido obtener soluciones.

La información relativa a las ejecuciones se almacena en el servidor a través de los *drivers* de almacenamiento, *prodef-solver* proporciona la interfaz `Storage`, que es la base para cualquier *driver*:

```
interface Storage {
  storeExecutionState: (
    executionID: ExecutionID,
    executionState: ExecutionState
  ) => Promise<void>;

  getExecutionState: (
    executionID: ExecutionID
  ) => Promise<ExecutionState | null>;

  removeExecutionState: (executionID: ExecutionID) => Promise<void>;
}
```

prodef-solver implementa un *driver* de almacenamiento basado en la memoria (no persistente), recomendado solo para realizar pruebas.

Este módulo está alojado en el repositorio “tfg-andres-calimero-prodef-solver” dentro de la organización de Parallel Algorithms and Languages de la ULL (`https://github.com/PAL-ULL/tfg-andres-calimero-prodef-solver`).

Para una documentación técnica más detallada se puede recurrir a la documentación auto-generada en la página de GitHub pages desplegada asociada al repositorio (`https://pal-ull.github.io/tfg-andres-calimero-prodef-solver/`).

3.7. Módulos específicos desarrollados

Los módulos específicos son módulos que expanden las capacidades y funcionalidades de *Prodef*, estos módulos se basan en las interfaces y clases abstractas proporcionadas por los módulos del núcleo, son dinámicos y fácilmente reemplazables.

Para poder tener un ejemplo completo de ejecución de la herramienta (desde la definición del problema hasta la obtención de soluciones) es necesario implementar una serie de módulos específicos. Se decidió usar el framework jMetal [4] para el resolutor de ejemplo (prodef-solver-jmetal). jMetal está escrito en Java y requiere que el usuario codifique en ese mismo lenguaje, por lo que también es necesario implementar un módulo que implemente un compilador de *Prodef* para Java (prodef-compiler-java), además, se implementó un driver de almacenamiento persistente que hace uso de bases de datos Redis para poder almacenar el estado de las ejecuciones (prodef-storage-redis).

En la Figura 3.9 se muestran los módulos específicos desarrollados junto con módulos del núcleo de los que extienden.

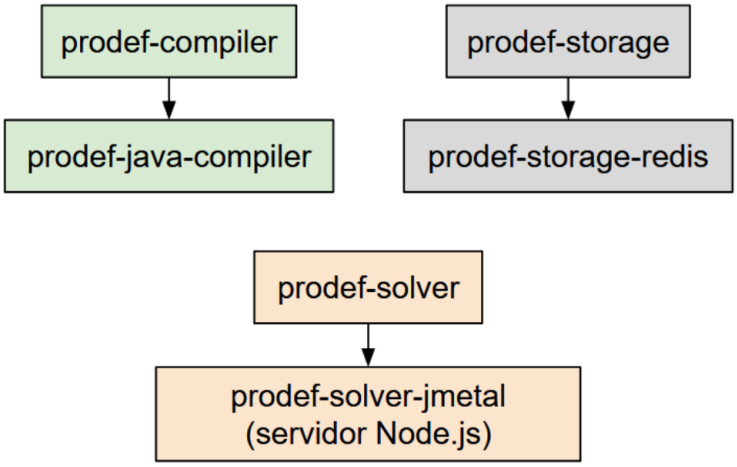

Figura 3.9: Módulos específicos desarrollados

3.7.1. prodef-compiler-java

prodef-compiler-java es un modulo especifico que implementa un compilador de *ProdefLang* para el lenguaje de programación *Java*. El compilador hereda de `CBasedCompiler` (definido en *prodef-compiler*) y, por tanto, solo implementa 6 métodos más el constructor. A continuación se muestra el código del compilador por partes.

El constructor llama al constructor padre con las opciones de configuración pertinentes y registra los compiladores de las funciones que se incluyen por defecto en *ProdefLang* (`sin`, `cos` y `sqrt`):

```
constructor(problem: Problem) {
  super(problem, {
    bracketsPosInDeclarations: 'afterType',
    explicitySetDimensionsInDeclarations: false,
    useNewKeywordInVariableDeclarations: true,
  });

  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;
```

```
    }
  });
}
```

`compileAbsoluteValue` es un método que, dada una expresión escrita en el lenguaje objetivo (*Java* en este caso), devuelve una expresión que resulta de aplicar la operación matemática de valor absoluto a la expresión original:

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

`compileMinimumValue` es un método que, dadas dos expresiones escritas en el lenguaje objetivo, devuelve una expresión que retorna el valor mínimo de ambas expresiones:

```
protected compileMinimumValue(
  type: WithinType,
  expression1: string,
  expression2: string
): string {
  return this.compileCastExpression(
    type,
    `Math.min(${expression1},${expression2})`
  );
}
```

`compileExponentialOperation` es un método que, dada una expresión escrita en el lenguaje objetivo que representa la base de una operación de potencia, y otra expresión que representa el exponente, devuelve una expresión que resulta de aplicar la operación matemática de potencia:

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

`compileInfinity` es un método que, dado un tipo de variable (entera o real) y un signo (positivo o negativo), devuelve una expresión que representa el valor “Infinito” en el lenguaje objetivo:

```
protected compileInfinity(type: WithinType, positiveInfinity = true): string {
  let result = '';
  switch (type) {
    case 'integers':
      if (positiveInfinity) {
        result = 'Integer.MAX_VALUE';
      } else {
        result = 'Integer.MIN_VALUE';
```

```
      }
      break;
    case 'reals':
      if (positiveInfinity) {
        result = 'Double.MAX_VALUE';
      } else {
        result = 'Double.MIN_VALUE';
      }
      break;
  }
  return result;
}
```

`transformSymbol` es un método que, dado un símbolo, devuelve otro símbolo valido en el lenguaje objetivo que evite colisiones con otros posibles símbolos:

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

Por último, `compileLoop` es un método que especifica como se va a generar el código en el lenguaje objetivo para ejecutar los bucles de las expresiones `sum`, `mult` y `assert`. En este caso se escogió una estrategia basada en el paradigma de programación funcional, por lo tanto se hace uso de los `Streams` de *Java*:

```
protected compileLoop(
  symbol: string,
  expression: string,
  reducer: ReducerType,
  start?: string | null,
  end?: string | null,
  vector?: string | null
): string {
  let rangeOrVectorExpression = '';
  if (start && end) {
    rangeOrVectorExpression = `IntStream.range((int) Math.min(${start}, ${end}), (int)
    ↪  Math.max(${start}, ${end}) + 1)`;
  } else if (vector) {
    rangeOrVectorExpression = `Stream.of(${vector})`;
  }

  if (rangeOrVectorExpression === '')
    throw new Error('invalid arguments provided');

  switch (reducer) {
    case 'sum':
      return `${rangeOrVectorExpression}.mapToDouble(${symbol} ->
      ↪  ${expression}).reduce(0, Double::sum)`;
    case 'mult':
      return `${rangeOrVectorExpression}.mapToDouble(${symbol} ->
      ↪  ${expression}).reduce(1, (l, r) -> l * r)`;
    case 'assert':
      return `${rangeOrVectorExpression}.allMatch(${symbol} -> ${expression})`;
    case 'min':
      return `${rangeOrVectorExpression}.mapToDouble(${symbol} ->
      ↪  ${expression}).reduce(Math::min).getAsDouble()`;
```

```
  }
}
```

A continuación se muestra una serie de ejemplos para el problema de la mochila binaria, explicado en el Capítulo 2.

Para el parámetro `N` (número de objetos disponibles), el código resultante en *Java* es:

```
int __N = 5;
```

Para el parámetro `MaxWeight`, el código resultante es:

```
int __MaxWeight = 80;
```

La variable `x`, de tipo vector y tamaño 5, se transforma en el siguiente código *Java*:

```
int[] __x = new int[(int) (5)];
```

Los objetos de tipo `Item` se transforman en:

```
double[][] __item = {{0, 33, 15},{0, 24, 20},{0, 36, 17},{0, 37, 8},{0, 12, 31}};
```

La función objetivo `sum x[i]*item[i].value over i=(1:N)` se transforma en:

```
IntStream.range((int) Math.min(1, __N), (int) Math.max(1, __N) + 1).mapToDouble(__i ->
↪  __x[(__i)-1] * __item[(__i)-1][(2)-1]).reduce(0, Double::sum);
```

Y, por último, la restricción `sum x[i]*item[i].weight over i=(1:N) <= MaxWeight` se transforma en:

```
IntStream.range((int) Math.min(1, __N), (int) Math.max(1, __N) + 1).mapToDouble(__i ->
↪  __x[(__i)-1] * __item[(__i)-1][(3)-1]).reduce(0, Double::sum) <= __MaxWeight;
```

Su expresión de grado de violación asociado resulta:

```
IntStream.range((int) Math.min(0, 1), (int) Math.max(0, 1) + 1).mapToDouble(_i -> {
 double _constraintViolation1 = 0.0;
 double _expression = IntStream.range((int) Math.min(1, __N), (int) Math.max(1, __N) +
 ↪  1).mapToDouble(__i -> __x[(__i) - 1] * __item[(__i) - 1][(3) - 1]).reduce(0,
 ↪  Double::sum);
 double _target1 = __MaxWeight;
 if (_expression <= _target1) {
  _constraintViolation1 = (double)(Math.abs(_expression - _target1));
 } else {
  _constraintViolation1 = -(double)(Math.abs(_expression - _target1));
  if (_constraintViolation1 >= 0.0) _constraintViolation1 = -0.1;
 }
 return _constraintViolation1;
}).reduce(0, Double::sum);
```

Este módulo está alojado en el repositorio “tfg-andres-calimero-prodef-compiler-java” dentro de la organización de Parallel Algorithms and Languages de la ULL (`https://github.com/PAL-ULL/tfg-andres-calimero-prodef-compiler-java`).

Para una documentación técnica más detallada se puede recurrir a la documentación auto-generada en la página de GitHub pages desplegada asociada al repositorio (`https://pal-ull.github.io/tfg-andres-calimero-prodef-compiler-java/`).

3.7.2. prodef-storage-redis

prodef-storage-redis es un modulo que implementa un *driver* de almacenamiento persistente basado en la base de datos *Redis,* para ello implementa la interfaz `Storage` definida en *prodef-solver* y hace uso de la librería *Tedis* (`https://tedis.silkjs.org/`) para la interacción con la base de datos.

En la clase `RedisStorage` se implementa todo el código del modulo:

```
class RedisStorage implements Storage {
  private _storageInfo: RedisStorageInfo;
  private _tedis: Tedis;

  constructor(storageInfo: RedisStorageInfo) {
    this._storageInfo = storageInfo;
    this._tedis = new Tedis({
      host: storageInfo.host,
      port: storageInfo.port,
      password: storageInfo.password,
    });
  }

  protected get storageInfo(): RedisStorageInfo {
    return this._storageInfo;
  }

  public async storeExecutionState(
    executionID: ExecutionID,
    executionState: ExecutionState
  ): Promise<void> {
    await this._tedis.set(
      this.serializeExecutionID(executionID),
      JSON.stringify(executionState)
    );
  }

  public async getExecutionState(
    executionID: ExecutionID
  ): Promise<ExecutionState | null> {
    const result = await this._tedis.get(
      this.serializeExecutionID(executionID)
    );
    if (typeof result !== 'string') return null;

    return JSON.parse(result) as ExecutionState;
  }

  public async removeExecutionState(executionID: ExecutionID): Promise<void> {
    await this._tedis.del(this.serializeExecutionID(executionID));
  }

```

```
  private serializeExecutionID(executionID: ExecutionID): string {
    return `${executionID.executionId}+${executionID.secretKey}`;
  }
}
```

Este módulo está alojado en el repositorio “tfg-andres-calimero-prodef-storage-redis” dentro de la organización de Parallel Algorithms and Languages de la ULL (`https://github.com/PAL-ULL/tfg-andres-calimero-prodef-storage-redis`).

Para una documentación técnica más detallada se puede recurrir a la documentación auto-generada en la página de GitHub pages desplegada asociada al repositorio (`https://pal-ull.github.io/tfg-andres-calimero-prodef-storage-redis/`).

3.7.3. prodef-solver-jmetal

prodef-solver-jmetal es un módulo que implementa un resolutor para *Prodef* basado en el framework *jMetal*, como todos los resolutores hereda 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 (ver Apéndice C) y posteriormente se ejecuta la clase `Main` desarrollada (ver Apéndice D) pasándole como argumentos el nombre de la clase generada con la plantilla (`ProdefProblemInstance.java`) y el tipo de problema (por ejemplo: `binary singleobjective`). 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 constituirá la solución final de la ejecución del problema. A continuación se muestra los fragmentos de código más relevantes del resolutor.

El constructor llama al constructor padre pasándole las capacidades del resolutor, en este caso no acepta problemas con múltiples variables de tipo permutación ni de distintos tipos:

```
constructor() {
  super({
    supportsMultiplePermutations: false,
    supportsMultipleVariableTypes: false,
  });
  this.registerHandlebarsHelpers();
}
```

El método `solve` es el único método que tiene que implementar obligatoriamente un resolutor, se trata de un método asíncrono que devuelve un objeto `SolverOutput` con el resultado de la ejecución del problema:

```
public async solve(
  problem: Problem,
  executionConfiguration: ExecutionConfiguration
): Promise<SolverOutput> {
  try {
    const compiler = new JavaCompiler(problem);
```

```
    const compiledProblem = compiler.compile({
      convertVariableTypesToMostGeneral: true,
      prodefFunctionPrefix: 'prodef.',
    });

    const javaProdefProblemInstance = await this.renderJavaSourceTemplate(
      compiledProblem,
      executionConfiguration
    );

    const problemId = uuidv1();
    const executionPath = `./executions/${problemId}`;
    const problemPath = `${executionPath}/src`;

    await fs.ensureDir(`${problemPath}`);

    await fs.outputFile(
      `${executionPath}/compiledProblem.json`,
      JSON.stringify(compiledProblem, null, 2)
    );

    await fs.outputFile(
      `${problemPath}/prodef/ProdefProblemInstance.java`,
      javaProdefProblemInstance
    );

    await exec(
      `javac -cp src/java/src:lib/* -d ${problemPath} src/java/src/prodef/Main.java`
    );

    await exec(
      `javac -cp src/java/src:lib/* -d ${problemPath}
      ↪  ${problemPath}/prodef/ProdefProblemInstance.java`
    );

    const processResults = await exec(
      `cd ${problemPath} && java -cp .:../../../lib/* prodef/Main
      ↪  prodef.ProdefProblemInstance ${
        compiledProblem.metadata.problemType
      } ${
        compiledProblem.metadata.numberOfGoals > 1
          ? 'multiobjective'
          : 'singleobjective'
      }`
    );

    const result = JSON.parse(processResults.stdout) as JMetalSolverOutput;

    return {
      state: 'resolved',
      solution: this.createSolution(compiledProblem, result),
    };
  } catch (error) {
    return {
      state: 'failed',
      error: error.message ? error.message : error,
    };
```

```
  }
}
```

siendo `SolverOutput` un objeto `RejectedOutput` en caso de que la ejecución fuese rechazada, un objeto `ResolvedOutput` en caso de que la ejecución se completase de forma satisfactoria y un objeto `FailedOutput` en caso de que ocurriese algún error irrecuperable durante la ejecución:

```
interface RejectedOutput {
  state: 'rejected';
  cause: string;
}

interface ResolvedOutput {
  state: 'resolved';
  solution: Solution;
}

interface FailedOutput {
  state: 'failed';
  error: string;
}

type SolverOutput = RejectedOutput | ResolvedOutput | FailedOutput;
```

y siendo `JMetalSolverOutput` la interfaz que implementa el archivo JSON que genera el ejecutable de *Java* cuando se completa satisfactoriamente la ejecución:

```
interface JMetalSolverVariableValue {
  symbol: string;
  index: number;
  value: number | number[] | number[][];
}

interface JMetalSolverGoalValue {
  index: number;
  value: number;
}

interface JMetalSolverResult {
  variableValues: JMetalSolverVariableValue[];
  goalValues: JMetalSolverGoalValue[];
  isFeasible: boolean;
}

interface JMetalSolverOutput {
  computingTime: number;
  results: JMetalSolverResult[];
}
```

La clase `Main` que se ejecuta forma parte del conjunto de clases *Java* desarrolladas para crear la capa de traducción necesaria entre *Prodef* y *jMetal* (ver Apéndice D). En función del tipo de problema (cantidad de funciones objetivo y codificación de las soluciones) se ejecutará una estrategia concreta preestablecida, en la Tabla 3.5 se muestra la configuración del algoritmo genético genérico simple (en el caso de problemas mono-

Codificación	Operador crossover	Operador mutación
Binaria	SinglePointCrossover	BitFlipMutation
Real	SBXCrossover	PolynomialMutation
Entera	IntegerSBXCrossover	IntegerPolynomialMutation
Permutación entera	PMXCrossover	PermutationSwapMutation

Tabla 3.5: Configuración del algoritmo para problemas mono-objetivo y multi-objetivo

objetivo) y del algoritmo NSGA-II (en el caso de problemas multi-objetivo).

En el caso de los problemas mono-objetivo, se utiliza como operador de selección `BinaryTournamentSelection`, y como comparador de soluciones `DominanceComparator`, independientemente del tipo de codificación de las soluciones. En el caso de los multi-objetivo, se utiliza como ranking `FastNonDominatedSortRanking`, y como comparador `IntegerValueAttributeComparator`. En ambos tipos de problemas el tamaño de la población está fijado a 100, y la probabilidad de *crossover* a 0.9.

prodef-solver-jmetal es un módulo ejecutable, al contrario que el resto de módulos expuestos anteriormente. Se puede iniciar el servidor del resolutor para servir la API ejecutando la tarea `start` del paquete `npm`. Por defecto usa `MemoryStorage` como *driver* de almacenamiento, se recomienda modificar el código y usar otro *driver*, como `RedisStorage`, para entornos de producción. A continuación se muestra el código que configura e inicia el servidor:

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

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

const server = new SolverServer(solver, storage, {
  enableLogger: true,
  port: 8046,
  maximumParallelExecutions: 5,
});

server.start();
```

`JMetalSolver` es el resolutor que implementa este módulo, se puede apreciar que para iniciar el servidor (`SolverServer`) hace falta una instancia de un resolutor (`solver` en este caso) y una instancia de un *driver* de almacenamiento (`storage` en este caso), además, se le puede proporcionar un objeto de configuración de forma opcional. El diseño modular permite usar cualquier combinación de resolutor con *driver* de almacenamiento.

A modo de ejemplo se muestra el proceso para ejecutar un problema de la mochila binaria, suponiendo que el servidor está ejecutándose sobre el puerto 8046 en `localhost`. Para solicitar una ejecución al resolutor se tiene que usar el endpoint `POST http://localhost:8046/v1/executions`, para ello se podría usar la herramienta *curl*:

```
curl -X POST http://localhost:8046/v1/executions \
    -H 'Content-Type: application/json' \
```

```
    -d '{
    "executionConfiguration": {
        "maxComputingTime": 5000,
        "maxNumberOfResults": 1
    },
    "problem": {
        "name": "Basic binary knapsack problem",
        "description": "Optional description (optimal: 130)",
        "parameters": [
            {
                "name": "Number of items",
                "symbol": "N",
                "value": 5
            },
            {
                "name": "Maximum weight",
                "symbol": "MaxWeight",
                "value": 80
            }
        ],
        "variables": [
            {
                "name": "Items in the knapsack",
                "symbol": "x",
                "within": "integers",
                "range": {
                    "lowerBound": 0,
                    "upperBound": 1
                },
                "shape": {
                    "type": "vector",
                    "isPermutation": false,
                    "size": {
                        "fixed": false,
                        "value": "N"
                    }
                }
            }
        ],
        "goals": [
            {
                "name": "Maximize the value of the items",
                "sense": "maximize",
                "expression": "sum x[i]*item[i].value over i=(1:N)",
                "weight": 1
            }
        ],
        "constraints": [
            {
                "name": "Limit the total weight of the items in the knapsack",
                "expression": "sum x[i]*item[i].weight over i=(1:N) <= MaxWeight"
            }
        ],
        "classes": [
            {
                "name": "Item",
                "symbol": "item",
```

```
                "attributes": [
                    {
                        "name": "Name",
                        "symbol": "name"
                    },
                    {
                        "name": "Value",
                        "symbol": "value"
                    },
                    {
                        "name": "Weight",
                        "symbol": "weight"
                    }
                ]
            }
        ],
        "objects": [
            {
                "class": "item",
                "attributes": [
                    {
                        "attribute": "name",
                        "value": "Item 1"
                    },
                    {
                        "attribute": "value",
                        "value": 33
                    },
                    {
                        "attribute": "weight",
                        "value": 15
                    }
                ]
            },
            {
                "class": "item",
                "attributes": [
                    {
                        "attribute": "name",
                        "value": "Item 2"
                    },
                    {
                        "attribute": "value",
                        "value": 24
                    },
                    {
                        "attribute": "weight",
                        "value": 20
                    }
                ]
            },
            {
                "class": "item",
                "attributes": [
                    {
                        "attribute": "name",
                        "value": "Item 3"
```

```
                    },
                    {
                        "attribute": "value",
                        "value": 36
                    },
                    {
                        "attribute": "weight",
                        "value": 17
                    }
                ]
            },
            {
                "class": "item",
                "attributes": [
                    {
                        "attribute": "name",
                        "value": "Item 4"
                    },
                    {
                        "attribute": "value",
                        "value": 37
                    },
                    {
                        "attribute": "weight",
                        "value": 8
                    }
                ]
            },
            {
                "class": "item",
                "attributes": [
                    {
                        "attribute": "name",
                        "value": "Item 5"
                    },
                    {
                        "attribute": "value",
                        "value": 12
                    },
                    {
                        "attribute": "weight",
                        "value": 31
                    }
                ]
            }
        ]
    }
}'
```

La información requerida por el servidor es el objeto de configuración de la ejecución y el propio objeto del problema, se puede observar que se ha establecido el tiempo máximo de ejecución a 5 segundos, y el número máximo de soluciones a 1 (solo se devolverá una solución, la mejor; se descartarán otras soluciones distintas aunque sean igual de buenas).

Como resultado de la petición se obtendrá un objeto similar a:

```
{
    "executionId": "2d37b350-b195-11ea-809b-8d346d014eac",
    "secretKey": "3f534913-185c-4c6d-8c08-2bb2256a9f7e"
}
```

Los valores de `executionId` y de `secretKey` son necesarios para poder recuperar la información de la ejecución que se acaba de crear en el servidor. Para recuperar el estado de la ejecución se tiene que usar el endpoint `GET http://localhost:8046/v1/executions/2d37b350-`

```
curl http://localhost:8046/v1/executions/2d37b350-b195-11ea-809b-8d346d014eac \
    -H 'Secret-Key: 3f534913-185c-4c6d-8c08-2bb2256a9f7e'
```

Como resultado de la petición se obtendrá un objeto como el siguiente (suponiendo que han transcurrido más de 5 segundos y que la ejecución ha sido correcta):


```
{
  "state":"resolved",
  "solution":{
    "results":[
      {
        "isFeasible":true,
        "goalValues":[
          {
            "expression":"sum x[i]*item[i].value over i=(1:N)",
            "sense":"maximize",
            "value":130,
            "name":"Maximize the value of the items",
            "weight":1
          }
        ],
        "variableValues":[
          {
            "symbol":"x",
            "value":[
              1,
              1,
              1,
              1,
              0
            ],
            "within":"integers",
            "name":"Items in the knapsack",
            "range":{
              "lowerBound":0,
              "upperBound":1
            },
            "shape":{
              "type":"vector",
              "isPermutation":false,
              "size":{
                "fixed":false,
                "value":"N"
              }
            }
          }
        ]
```

```
      }
    ],
    "computingTime":5014
  },
  "stateMessage":"The problem was successfully solved",
  "problem":{
    "name":"Basic binary knapsack problem",
    "description":"Optional description (optimal: 130)",
    ...
  },
  "lastUpdate":1592521592719
}
```

Este objeto contiene los valores de las variables para todos los resultados, en este caso solo hay un resultado porque así se indicó previamente. Se puede llegar a la conclusión de que metiendo todos los objetos en la mochila, menos el último, se obtiene una solución factible y cercana a la óptima con un valor de 130.

Este módulo está alojado en el repositorio "tfg-andres-calimero-prodef-solver-jmetal" dentro de la organización de Parallel Algorithms and Languages de la ULL (`https://github.com/PAL-ULL/tfg-andres-calimero-prodef-solver-jmetal`).

Para una documentación técnica más detallada se puede recurrir a la documentación auto-generada en la página de GitHub pages desplegada asociada al repositorio (`https://pal-ull.github.io/tfg-andres-calimero-prodef-solver-jmetal/`).

Capítulo 4

Conclusiones y líneas futuras

En este capitulo se expondrán las conclusiones de este trabajo, así como algunos aspectos de la herramienta que se podrían mejorar en un futuro.

4.1. Conclusiones

Se ha desarrollado una herramienta modular (*Prodef*) para la resolución de problemas de optimización combinatoria. Esta herramienta define un meta-modelo para los problemas, junto con un lenguaje (*ProdefLang*) para la definición de funciones objetivo y restricciones. El diseño modular posibilita la implementación de nuevos resolutores, compiladores y *drivers* de almacenamiento de forma sencilla, actualmente se implementa un resolutor que usa el framework *jMetal*, un compilador de *ProdefLang* para *Java* (usado por el resolutor de *jMetal*) y un *driver* de almacenamiento basado en la base de datos *Redis*. Con los módulos implementados actualmente se ha demostrado que la herramienta es capaz de modelar, interpretar y resolver problemas de optimización combinatoria de corte académico, obteniendo resultados satisfactorios.

El objetivo de *Prodef* es facilitar a empresas e individuos el modelado y la resolución de problemas de optimización combinatoria, tanto mono-objetivo como multi-objetivo, permitiéndoles beneficiarse de los frameworks y librerías existentes que implementan algoritmos para la resolución de este tipo de problemas, sin tener que conocer los detalles de implementación de los mismos, por lo tanto, *Prodef* actúa como una capa de abstracción entre el modelo del problema y sus mecanismos de resolución. Una herramienta como *Prodef* podría resultar muy interesante desde el punto de vista comercial. En muchas ocasiones las pequeñas y medianas empresas no se pueden permitir contratar a un matemático y/o informático para optimizar los procesos productivos de la empresa con técnicas como las vistas en este trabajo. Sin embargo, con una herramienta como *Prodef* cualquier empleado de la empresa que conozca bien el proceso productivo que se quiera optimizar, podría modelarlo con la herramienta y obtener unos resultados que podrían implicar una reducción de costes y/o mejora en la productividad. Actualmente *Prodef* está en sus inicios y hay un gran margen de mejora para que su viabilidad comercial sea una realidad. En la siguiente sección se nombran algunas cuestiones de la herramienta que se podrían mejorar con el objetivo de hacerla más interesante de cara a una mejor acogida de la misma.

4.2. Trabajos futuros

A continuación, se listan algunos aspectos de *Prodef* que se podrían mejorar de cara a una mejor acogida de la herramienta:

- **Dotar de mayor inteligencia a los resolutores**. Actualmente, el único resolutor que se ha implementado (basado en *jMetal*) no extrae información del modelo del problema, más allá de si es mono-objetivo o multi-objetivo, para determinar la estrategia de resolución a seguir. Se podría dotar de más inteligencia al resolutor de tal forma que varíe los algoritmos a usar y sus configuraciones en función de la naturaleza del problema. Para lograrlo se podría usar, por ejemplo, la inteligencia artificial. Con esto se podría conseguir una mejora en la calidad de los resultados y/o una reducción en el tiempo de cómputo.

- **Implementación de una interfaz gráfica**. Por falta de tiempo no se ha podido implementar una interfaz gráfica para la herramienta en este trabajo, sin embargo, la interfaz gráfica es una parte fundamental de la misma. El objetivo de la herramienta es simplificar al máximo el modelado de problemas y su resolución, una interfaz gráfica permitiría al usuario usar la herramienta sin necesidad de manipular directamente documentos JSON ni realizar peticiones HTTP manualmente. Además, la interfaz gráfica podría incorporar un lenguaje visual para la definición de funciones objetivo y restricciones en *ProdefLang*, usando, por ejemplo, la librería *Blocky* (`https://developers.google.com/blockly`). Se ha diseñado *Prodef* teniendo en mente que, en un futuro, será usado a través de una interfaz gráfica web, es por ello que el lenguaje escogido para su desarrollo fue *TypeScript*, el formato usado para los problemas es *JSON* y los módulos generan *bundles* de *JavaScript* compatibles con los navegadores.

- **Implementación de más resolutores**. Como ya se ha mencionado, solo se ha implementado un resolutor en este trabajo, de cara a un futuro sería interesante implementar más resolutores basados en diferentes frameworks y librerías, como *METCO*, *ECJ* o *EasyLocal*. Implementar más resolutores no solo permitirá proveer al usuario de mayor diversidad, sino también ayudará a detectar patrones comunes en los distintos frameworks y comparar las distintas estrategias que han seguido. Todo ese conocimiento se podría usar para mejorar *Prodef*.

- **Definición e implementación de funciones usando *ProdefLang***. El lenguaje diseñado (*ProdefLang*) permite en sus expresiones la ejecución de funciones previamente definidas. Estas funciones deben ser implementadas en los módulos compiladores correspondientes, con lo cual, si un usuario quiere hacer uso de una función que no haya sido previamente implementada en el compilador que está usando, deberá implementarla él, usando el lenguaje objetivo (por ejemplo, *Java*). Esto va en contra del objetivo de permitir a los usuarios sin conocimientos de programación modelar problemas. Para solucionar esto se podría modificar el lenguaje para que permita la implementación de funciones usando el propio lenguaje, es decir, que el usuario pueda definir la implementación de su función usando también *ProdefLang*. El código en el lenguaje objetivo sería entonces generado por el compilador, de forma transparente al usuario. Es importante asegurarse de que *ProdefLang* siga siendo un lenguaje sencillo e intuitivo, no se debería caer en el error de añadirle

tantas funcionalidades que sea indistinguible de un lenguaje de programación de alto nivel como *Java*, pues entonces dejaría de tener razón de ser.

- **Permitir que las funciones en *ProdefLang* devuelvan valores con dimensión superior a 0**. Actualmente *ProdefLang* soporta la declaración de funciones en tiempo de compilación, para declarar una función es necesario proveer el nombre de la misma, junto con la cantidad de argumentos que recibe y sus dimensiones correspondientes. Sin embargo, no es posible especificar la dimensión del valor que devuelve la función: se da por hecho que una función siempre devuelve un valor numérico simple (de dimensión 0). Sería útil que se pudiesen definir funciones cuyo valor de retorno tuviese una dimensión superior a 0. Para ello habría que modificar, entre otras cosas, el transformador de *ProdefLang* que transforma el árbol de parseo en el AST: habría que asegurarse de que las expresiones son válidas teniendo en cuenta que las funciones ya no solo pueden devolver un número.

Capítulo 5

Summary and conclusions

5.1. Summary

Bio-inspired optimization methods offer great possibilities for solving problems in the business environment but, due to their intrinsic complexity and also their particularities when implementing and adapting them to each of the problems, it has not been possible to extend its use outside the scope of conventional research.

In this work, a tool has been developed that allows the definition of combinatorial optimization problems at an abstract level of description and their subsequent resolution using libraries and external frameworks, such as *jMetal*. This tool acts as an abstraction layer between the problem model and its resolution techniques, in order to facilitate the user, without programming knowledge or bio-inspired optimization methods, the task of modeling and solving combinatorial optimization problems.

5.2. Conclusions

A modular tool (*Prodef*) for solving combinatorial optimization problems has been developed. This tool defines a meta-model for the problems, together with a language (*ProdefLang*) for the definition of objective functions and restrictions. Modular design makes it easy to implement new solvers, compilers, and storage drivers. Currently is implemented: a solver that uses the *jMetal* framework, a compiler of *ProdefLang* for *Java* (used by the *jMetal* solver) and a storage driver based on the *Redis* database. With the modules currently implemented, it has been demonstrated that the tool is capable of modeling, interpreting and solving combinatorial optimization problems of an academic nature, obtaining satisfactory results.

The objective of *Prodef* is to make it easier for companies and individuals to model and solve combinatorial optimization problems, both mono-objective and multi-objective, allowing them to take advantage of existing frameworks and libraries that implement algorithms to solve this type of problems, without having to know the details of their implementation. Therefore, *Prodef* acts as an abstraction layer between the problem model and its resolution mechanisms. A tool like *Prodef* could be very interesting from a commercial point of view, on many occasions small and medium-sized companies cannot afford to hire a mathematician and/or computer scientist to optimize the company’s production processes with techniques such as those presented in this work, with a tool

like *Prodef* any employee of the company, who knows well the production process to be optimized, could model it with the tool and obtain results that could imply a reduction in costs and/or or improvement in productivity. Currently *Prodef* is in its infancy and there is room for improvement so that its commercial viability becomes a reality.

Capítulo 6

Presupuesto

En este capítulo se presenta el presupuesto del trabajo realizado. El presupuesto se divide en dos partes: los costes tecnológicos asociados al desarrollo y los costes de los recursos humanos.

Los costes tecnológicos se presentan en la Tabla 6.1.

Tipo	**Descripción**	**Coste**
Amortización de equipos informáticos	Amortización del equipo informático usado en el desarrollo (PC, pantallas, teclado y ratón).	250 €
Hosting	Hosting del resolutor en Azure para realización de pruebas.	40 €

Tabla 6.1: Costes tecnológicos

Los costes de recursos humanos se presentan en la Tabla 6.2, la cantidad de horas empleadas corresponde con el número de créditos ECTS de la asignatura “Trabajo Fin de Grado” (12 créditos), que a razón de 25 horas por crédito hace un total de 300 horas.

Tipo	**Descripción**	**Coste**
Desarrollo	Trabajo de desarrollo con un coste de 15 € la hora.	4500 €

Tabla 6.2: Costes de recursos humanos

En la Tabla 6.3 se presentan los costes totales de este proyecto.

Tipo	**Coste**
Costes tecnológicos	290 €
Costes de recursos humanos	4500 €
Total	4790 €

Tabla 6.3: Costes totales

Apéndice A

Esquemas JSON

Esquemas JSON para los problemas y las soluciones, siguiendo el draft-07 de JSON Schema (https://json-schema.org/).

A.1. Esquema JSON del problema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$ref": "#/definitions/ProblemInterface",
  "definitions": {
    "ProblemInterface": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "description": "Name of the problem."
        },
        "description": {
          "type": "string",
          "description": "Description of the problem (optional)."
        },
        "parameters": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/ParameterInterface"
          },
          "description": "Parameters of the problem (optional)."
        },
        "variables": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/VariableInterface"
          },
          "description": "Variables of the problem.",
          "minItems": 1
        },
        "goals": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/GoalInterface"
          },
```

```
          "description": "Objective functions of the problem.",
          "minItems": 1
        },
        "constraints": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/ConstraintInterface"
          },
          "description": "Constraints of the problem (optional)."
        },
        "classes": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/ClassInterface"
          },
          "description": "Classes of the problem (optional)."
        },
        "objects": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/ObjectInterface"
          },
          "description": "Instances of the classes of the problem (optional)."
        }
      },
      "required": [
        "name",
        "variables",
        "goals"
      ],
      "additionalProperties": false
    },
    "ParameterInterface": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "description": "Name of the parameter (optional)."
        },
        "symbol": {
          "$ref": "#/definitions/SymbolType",
          "description": "Symbol of the parameter, must be unique."
        },
        "value": {
          "type": "number",
          "description": "Value of the parameter."
        }
      },
      "required": [
        "symbol",
        "value"
      ],
      "additionalProperties": false
    },
    "SymbolType": {
      "type": "string"
    },
```

```
    "VariableInterface": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "description": "Name of the variable (optional)."
        },
        "symbol": {
          "$ref": "#/definitions/SymbolType",
          "description": "Symbol of the variable, must be unique."
        },
        "within": {
          "$ref": "#/definitions/WithinType",
          "description": "Number set to which the variable belongs."
        },
        "shape": {
          "$ref": "#/definitions/Shape",
          "description": "Shape of the variable (single, vector or matrix)."
        },
        "range": {
          "$ref": "#/definitions/Range",
          "description": "Range of the value/s of the variable (by default unbounded)."
        }
      },
      "required": [
        "symbol",
        "within",
        "shape"
      ],
      "additionalProperties": false
    },
    "WithinType": {
      "type": "string",
      "enum": [
        "integers",
        "reals"
      ]
    },
    "Shape": {
      "anyOf": [
        {
          "$ref": "#/definitions/SingleShape"
        },
        {
          "$ref": "#/definitions/VectorShape"
        },
        {
          "$ref": "#/definitions/MatrixShape"
        }
      ]
    },
    "SingleShape": {
      "type": "object",
      "properties": {
        "type": {
          "type": "string",
          "enum": [
```

```
            "single"
          ]
        }
      },
      "required": [
        "type"
      ],
      "additionalProperties": false
    },
    "VectorShape": {
      "type": "object",
      "properties": {
        "type": {
          "type": "string",
          "enum": [
            "vector"
          ]
        },
        "isPermutation": {
          "type": "boolean"
        },
        "size": {
          "$ref": "#/definitions/Size"
        }
      },
      "required": [
        "type",
        "isPermutation",
        "size"
      ],
      "additionalProperties": false
    },
    "Size": {
      "anyOf": [
        {
          "$ref": "#/definitions/ParameterizedSize"
        },
        {
          "$ref": "#/definitions/FixedSize"
        }
      ]
    },
    "ParameterizedSize": {
      "type": "object",
      "properties": {
        "fixed": {
          "type": "boolean",
          "enum": [
            false
          ],
          "description": "Defines if the size is a fixed number or a parameterized
          ↪  value."
        },
        "value": {
          "$ref": "#/definitions/SymbolType",
          "description": "Size, can be an integer or a parameter of the problem."
        }
```

```
    },
    "required": [
      "fixed",
      "value"
    ],
    "additionalProperties": false
  },
  "FixedSize": {
    "type": "object",
    "properties": {
      "fixed": {
        "type": "boolean",
        "enum": [
          true
        ],
        "description": "Defines if the size is a fixed number or a parameterized
        ↪  value."
      },
      "value": {
        "type": "integer",
        "description": "Size, can be an integer or a parameter of the problem."
      }
    },
    "required": [
      "fixed",
      "value"
    ],
    "additionalProperties": false
  },
  "MatrixShape": {
    "type": "object",
    "properties": {
      "type": {
        "type": "string",
        "enum": [
          "matrix"
        ]
      },
      "size": {
        "type": "object",
        "properties": {
          "rows": {
            "$ref": "#/definitions/Size"
          },
          "columns": {
            "$ref": "#/definitions/Size"
          }
        },
        "required": [
          "rows",
          "columns"
        ],
        "additionalProperties": false
      }
    },
    "required": [
      "type",
```

```
        "size"
      ],
      "additionalProperties": false
    },
    "Range": {
      "type": "object",
      "properties": {
        "lowerBound": {
          "anyOf": [
            {
              "type": "string",
              "enum": [
                "-Infinity"
              ]
            },
            {
              "type": "number"
            }
          ]
        },
        "upperBound": {
          "anyOf": [
            {
              "type": "string",
              "enum": [
                "Infinity"
              ]
            },
            {
              "type": "number"
            }
          ]
        }
      },
      "required": [
        "lowerBound",
        "upperBound"
      ],
      "additionalProperties": false
    },
    "GoalInterface": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "description": "Name of the objective function (optional)."
        },
        "sense": {
          "$ref": "#/definitions/GoalSense",
          "description": "Sense of the objective function (minimize or maximize)."
        },
        "expression": {
          "type": "string",
          "description": "Expression of the objective function in prodef syntax.\n\nFor
          ↪  example: sum(i=1:N, x[i]*food[i].cost)"
        },
        "weight": {
```

```
          "type": "number",
          "description": "Weight of the objective function for single objective resolvers (optional).",
          "minimum": 0,
          "maximum": 1
        }
      },
      "required": [
        "sense",
        "expression"
      ],
      "additionalProperties": false
    },
    "GoalSense": {
      "type": "string",
      "enum": [
        "minimize",
        "maximize"
      ]
    },
    "ConstraintInterface": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "description": "Name of the constraint (optional)."
        },
        "expression": {
          "type": "string",
          "description": "Expression of the constraint in prodef syntax.\n\nFor example: sum x[i]*food[i].volume over i=1:N <= Vmax"
        }
      },
      "required": [
        "expression"
      ],
      "additionalProperties": false
    },
    "ClassInterface": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "description": "Name of the class (optional)."
        },
        "symbol": {
          "$ref": "#/definitions/SymbolType",
          "description": "Symbol of the class, must be unique."
        },
        "attributes": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/ClassAttribute"
          },
          "description": "Attributes of the class.",
          "minItems": 1
        }
```

```
      },
      "required": [
        "symbol",
        "attributes"
      ],
      "additionalProperties": false
    },
    "ClassAttribute": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "description": "Name of the attribute (optional)."
        },
        "symbol": {
          "$ref": "#/definitions/SymbolType",
          "description": "Symbol of the attribute, must be unique."
        }
      },
      "required": [
        "symbol"
      ],
      "additionalProperties": false
    },
    "ObjectInterface": {
      "type": "object",
      "properties": {
        "class": {
          "type": "string",
          "description": "Class of the object."
        },
        "attributes": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/ObjectAttribute"
          },
          "description": "Attributes of the object.",
          "minItems": 1
        }
      },
      "required": [
        "class",
        "attributes"
      ],
      "additionalProperties": false
    },
    "ObjectAttribute": {
      "type": "object",
      "properties": {
        "attribute": {
          "type": "string",
          "description": "Symbol of the attribute."
        },
        "value": {
          "type": [
            "string",
            "number"
```

```
          ],
          "description": "Value of the attribute."
        }
      },
      "required": [
        "attribute",
        "value"
      ],
      "additionalProperties": false
    }
  }
}
```

A.2. Esquema JSON de la solución

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$ref": "#/definitions/SolutionInterface",
  "definitions": {
    "SolutionInterface": {
      "type": "object",
      "properties": {
        "results": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/ResultInterface"
          },
          "description": "List of results (solutions of the problem) ordered from best
          ↪  to worse."
        },
        "computingTime": {
          "type": "number",
          "description": "Computed time consumed to obtain the results."
        }
      },
      "required": [
        "results",
        "computingTime"
      ],
      "additionalProperties": false
    },
    "ResultInterface": {
      "type": "object",
      "properties": {
        "isFeasible": {
          "type": "boolean",
          "description": "True if the solution is feasible."
        },
        "variableValues": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/VariableValue"
```

```
        },
        "description": "Values of the variables of the problem.",
        "minItems": 1
      },
      "goalValues": {
        "type": "array",
        "items": {
          "$ref": "#/definitions/GoalValue"
        },
        "description": "Values (fitness) of the objective functions of the problem.",
        "minItems": 1
      }
    },
    "required": [
      "isFeasible",
      "variableValues",
      "goalValues"
    ],
    "additionalProperties": false
  },
  "VariableValue": {
    "anyOf": [
      {
        "$ref": "#/definitions/SingleVariableValue"
      },
      {
        "$ref": "#/definitions/VectorVariableValue"
      },
      {
        "$ref": "#/definitions/MatrixVariableValue"
      }
    ]
  },
  "SingleVariableValue": {
    "type": "object",
    "properties": {
      "name": {
        "type": "string",
        "description": "Name of the variable (optional)."
      },
      "symbol": {
        "$ref": "#/definitions/SymbolType",
        "description": "Symbol of the variable, must be unique."
      },
      "within": {
        "$ref": "#/definitions/WithinType",
        "description": "Number set to which the variable belongs."
      },
      "range": {
        "$ref": "#/definitions/Range",
        "description": "Range of the value/s of the variable (by default unbounded)."
      },
      "value": {
        "type": "number"
      },
      "shape": {
        "$ref": "#/definitions/SingleShape"
```

```
        }
      },
      "required": [
        "shape",
        "symbol",
        "value",
        "within"
      ],
      "additionalProperties": false
    },
    "SymbolType": {
      "type": "string"
    },
    "WithinType": {
      "type": "string",
      "enum": [
        "integers",
        "reals"
      ]
    },
    "Range": {
      "type": "object",
      "properties": {
        "lowerBound": {
          "anyOf": [
            {
              "type": "string",
              "enum": [
                "-Infinity"
              ]
            },
            {
              "type": "number"
            }
          ]
        },
        "upperBound": {
          "anyOf": [
            {
              "type": "string",
              "enum": [
                "Infinity"
              ]
            },
            {
              "type": "number"
            }
          ]
        }
      },
      "required": [
        "lowerBound",
        "upperBound"
      ],
      "additionalProperties": false
    },
    "SingleShape": {
```

```
      "type": "object",
      "properties": {
        "type": {
          "type": "string",
          "enum": [
            "single"
          ]
        }
      },
      "required": [
        "type"
      ],
      "additionalProperties": false
    },
    "VectorVariableValue": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "description": "Name of the variable (optional)."
        },
        "symbol": {
          "$ref": "#/definitions/SymbolType",
          "description": "Symbol of the variable, must be unique."
        },
        "within": {
          "$ref": "#/definitions/WithinType",
          "description": "Number set to which the variable belongs."
        },
        "range": {
          "$ref": "#/definitions/Range",
          "description": "Range of the value/s of the variable (by default unbounded)."
        },
        "value": {
          "type": "array",
          "items": {
            "type": "number"
          }
        },
        "shape": {
          "$ref": "#/definitions/VectorShape"
        }
      },
      "required": [
        "shape",
        "symbol",
        "value",
        "within"
      ],
      "additionalProperties": false
    },
    "VectorShape": {
      "type": "object",
      "properties": {
        "type": {
          "type": "string",
          "enum": [
```

```
            "vector"
          ]
        },
        "isPermutation": {
          "type": "boolean"
        },
        "size": {
          "$ref": "#/definitions/Size"
        }
      },
      "required": [
        "type",
        "isPermutation",
        "size"
      ],
      "additionalProperties": false
    },
    "Size": {
      "anyOf": [
        {
          "$ref": "#/definitions/ParameterizedSize"
        },
        {
          "$ref": "#/definitions/FixedSize"
        }
      ]
    },
    "ParameterizedSize": {
      "type": "object",
      "properties": {
        "fixed": {
          "type": "boolean",
          "enum": [
            false
          ],
          "description": "Defines if the size is a fixed number or a parameterized
          ↪  value."
        },
        "value": {
          "$ref": "#/definitions/SymbolType",
          "description": "Size, can be an integer or a parameter of the problem."
        }
      },
      "required": [
        "fixed",
        "value"
      ],
      "additionalProperties": false
    },
    "FixedSize": {
      "type": "object",
      "properties": {
        "fixed": {
          "type": "boolean",
          "enum": [
            true
          ],
```

```
        "description": "Defines if the size is a fixed number or a parameterized
        ↪  value."
      },
      "value": {
        "type": "integer",
        "description": "Size, can be an integer or a parameter of the problem."
      }
    },
    "required": [
      "fixed",
      "value"
    ],
    "additionalProperties": false
  },
  "MatrixVariableValue": {
    "type": "object",
    "properties": {
      "name": {
        "type": "string",
        "description": "Name of the variable (optional)."
      },
      "symbol": {
        "$ref": "#/definitions/SymbolType",
        "description": "Symbol of the variable, must be unique."
      },
      "within": {
        "$ref": "#/definitions/WithinType",
        "description": "Number set to which the variable belongs."
      },
      "range": {
        "$ref": "#/definitions/Range",
        "description": "Range of the value/s of the variable (by default unbounded)."
      },
      "value": {
        "type": "array",
        "items": {
          "type": "array",
          "items": {
            "type": "number"
          }
        }
      },
      "shape": {
        "$ref": "#/definitions/MatrixShape"
      }
    },
    "required": [
      "shape",
      "symbol",
      "value",
      "within"
    ],
    "additionalProperties": false
  },
  "MatrixShape": {
    "type": "object",
    "properties": {
```

```
        "type": {
          "type": "string",
          "enum": [
            "matrix"
          ]
        },
        "size": {
          "type": "object",
          "properties": {
            "rows": {
              "$ref": "#/definitions/Size"
            },
            "columns": {
              "$ref": "#/definitions/Size"
            }
          },
          "required": [
            "rows",
            "columns"
          ],
          "additionalProperties": false
        }
      },
      "required": [
        "type",
        "size"
      ],
      "additionalProperties": false
    },
    "GoalValue": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "description": "Name of the objective function (optional)."
        },
        "sense": {
          "$ref": "#/definitions/GoalSense",
          "description": "Sense of the objective function (minimize or maximize)."
        },
        "expression": {
          "type": "string",
          "description": "Expression of the objective function in prodef syntax.\n\nFor
          ↪  example: sum(i=1:N, x[i]*food[i].cost)"
        },
        "weight": {
          "type": "number",
          "description": "Weight of the objective function for single objective
          ↪  resolvers (optional).",
          "minimum": 0,
          "maximum": 1
        },
        "value": {
          "type": "number"
        }
      },
      "required": [
```

```
        "expression",
        "sense",
        "value"
      ],
      "additionalProperties": false
    },
    "GoalSense": {
      "type": "string",
      "enum": [
        "minimize",
        "maximize"
      ]
    }
  }
}
```

Apéndice B

Gramática de ProdefLang

```
grammar ProdefLang;

/*
* Parser rules
*/

root : (booleanExpression | expression) EOF;

booleanExpression
  : expression relationOp expression (relationOp expression)?
  | LPAREN booleanExpression RPAREN
  | assertExpression
  ;

expression
  : LPAREN expression RPAREN
  | (PLUS | MINUS) (PLUS | MINUS)? expression
  | <assoc=right> expression POW expression
  | expression (TIMES | DIV) expression
  | expression (PLUS | MINUS) expression
  | multExpression
  | sumExpression
  | functionCall
  | scientificNumber | symbol
  ;

assertExpression : KEYWORD_ASSERT booleanExpression KEYWORD_FORALL assignations ;

multExpression : KEYWORD_MULT expression KEYWORD_OVER assignations ;

sumExpression : KEYWORD_SUM expression KEYWORD_OVER assignations ;

functionCall : functionName LPAREN functionArguments RPAREN;

functionName : atomSymbol ;

functionArguments : expression (COMMA expression)* ;

assignation : atomSymbol ASSIGN_OP (range | naturalVector) ;

assignations : assignation (COMMA assignation)* ;

```

```
range : LPAREN expression RANGE_OP expression RPAREN ;

naturalVector : LSQUARE (atomSymbol | naturalNumber) (COMMA (atomSymbol |
↪  naturalNumber))* RSQUARE ;

// Maximum dimension is 2
indexAccessor : LSQUARE expression (COMMA expression)? RSQUARE ;
nameAccessor : NAME_ACCESSOR ;

symbol
  : symbol indexAccessor
  | symbol nameAccessor
  | atomSymbol
  ;

atomSymbol : SYMBOL ;

naturalNumber : NATURAL_NUMBER ;

scientificNumber : SCIENTIFIC_NUMBER | naturalNumber ;

relationOp
  : GT
  | GOET
  | LT
  | LOET
  | EQ
  | NEQ
  ;

/*
* Lexer rules, in order
*/

fragment NUMBER : ('0' | NATURAL_NUMBER) ('.' ('0' .. '9')+)? ;
fragment UNSIGNED_INTEGER : ('0' .. '9')+ ;

fragment E : 'E' | 'e' ;

fragment S: 'S' | 's' ;
fragment U: 'U' | 'u' ;
fragment M: 'M' | 'm' ;

fragment F: 'F' | 'f' ;
fragment O: 'O' | 'o' ;
fragment R: 'R' | 'r' ;
fragment A: 'A' | 'a' ;
fragment L: 'L' | 'l' ;

fragment T: 'T' | 't' ;

fragment V: 'V' | 'v' ;

fragment SIGN : ('+' | '-') ;

fragment VALID_ID_START : ('a' .. 'z') | ('A' .. 'Z') ;
fragment VALID_ID_CHAR : VALID_ID_START | ('0' .. '9') | '_' ;
```

```
99
100 // Name accessor as lexer rule to allow the use of the reserved words for attributes
    ↪  names.
101 NAME_ACCESSOR : POINT SYMBOL ;
102
103 KEYWORD_ASSERT : A S S E R T WS ;
104 KEYWORD_SUM : S U M WS ;
105 KEYWORD_MULT: M U L T WS ;
106 KEYWORD_FORALL : WS F O R (WS)? A L L WS ;
107 KEYWORD_OVER : WS O V E R WS ;
108
109 SYMBOL : VALID_ID_START VALID_ID_CHAR* ;
110
111 NATURAL_NUMBER : ('1' .. '9') ('0' .. '9')* ;
112
113 //The NUMBER part gets its potential sign from "(PLUS | MINUS)*" in the expression rule
114 SCIENTIFIC_NUMBER : NUMBER (E SIGN? UNSIGNED_INTEGER)? ;
115
116 LPAREN : '(' ;
117 RPAREN : ')' ;
118
119 LSQUARE : '[' ;
120 RSQUARE : ']' ;
121
122 PLUS : '+' ;
123 MINUS : '-' ;
124 TIMES : '*' ;
125 DIV : '/' ;
126 POW : '^' ;
127
128 GT : '>' ;
129 GOET : '>=' ;
130 LT : '<' ;
131 LOET : '<=' ;
132 EQ : '==' ;
133 NEQ : '!=' ;
134
135 ASSIGN_OP : '=' ;
136 RANGE_OP : ':' ;
137
138 POINT : '.' ;
139 COMMA : ',' ;
140
141 WS : ([ \r\n\t])+ -> skip ;
```

Apéndice C

Plantilla usada en el resolutor de jMetal

```
package prodef;

import prodef.AbstractProdefProblemInstance;
import prodef.utils.Variable;
import org.uma.jmetal.solution.Solution;
import org.uma.jmetal.solution.binarysolution.BinarySolution;

import java.util.*;
import java.util.stream.*;

public class ProdefProblemInstance<N extends Number> extends
↪  AbstractProdefProblemInstance<N> {
    // Parameters
    {{#each compiledProblem.parameters}}
    private final {{{expression}}}
    {{/each}}

    // Objects
    {{#each compiledProblem.objects}}
    private final {{{expression}}}
    {{/each}}

    // Variables
    {{#each compiledProblem.variables}}
    private {{{expression}}}
    {{/each}}

    public ProdefProblemInstance() {
        // ProdefProblemInstance info
        {{#with compiledProblem.metadata}}
        isBinary = {{{is_binary problemType}}};
        isIntegerPermutation = {{{is_integer_permutation problemType}}};
        numberOfVariables = {{{numberOfVariables}}};
        numberOfObjectives = {{{numberOfGoals}}};
        numberOfConstraints = {{{numberOfConstraints}}};
        numberOfAtomicVariables = {{{numberOfAtomicVariables}}};
        problemName = "{{{problemName}}}";
        {{/with}}

```

```
        // Execution configuration
        {{#with executionConfiguration}}
        maxComputingTime = {{{maxComputingTime}}};
        maxNumberOfResults = {{{maxNumberOfResults}}};
        {{/with}}

        // Functions
        prodef = new Functions();

        // Variables
        List<Variable> variables = new ArrayList<>();

        {{#each compiledProblem.variables}}
        Variable {{{symbol}}}_v = new Variable({{{index}}}, "{{{symbol}}}",
↪   "{{{originalSymbol}}}", {{{size}}}, {{{dimensions}}}, {{{rows}}}, {{{columns}}},
↪   {{{range.lowerBound}}}, {{{range.upperBound}}});
        variables.add({{{symbol}}}_v);
        {{/each}}

        buildVariablesMaps(variables);
    }

    public void updateBinaryVariables(final BinarySolution solution) {
        int variableIndex = 0;
        int bitIndex = 0;
        BitSet bitset;

        {{#each compiledProblem.variables}}
        {{{update_binary_variable .}}}
        {{/each}}
    }

    protected void updateCompensatedVariables(final Solution<N> solution) {
        int index = 0;

        {{#each compiledProblem.variables}}
        {{{update_variable .}}}
        {{/each}}
    }

    public double evaluateGoal(int goalIndex) throws Exception {
        switch (goalIndex) {
            {{#each compiledProblem.goals}}
            case {{{index}}}:
                return {{{expression}}};
            {{/each}}
        }

        throw new Exception("invalid goal index");
    }

    public boolean isMinimize(int goalIndex) {
        switch (goalIndex) {
            {{#each compiledProblem.goals}}
            case {{{index}}}:
                return {{{is_minimize sense}}};
            {{/each}}
```

```
        }

        return false;
    }

    public boolean evaluateConstraint(int constraintIndex) throws Exception {
        switch (constraintIndex) {
            {{#each compiledProblem.constraints}}
            case {{{index}}}:
                return {{{expression}}};
            {{/each}}
        }

        throw new Exception("invalid constraint index");
    }

    public double evaluateConstraintViolation(int constraintIndex) throws Exception {
        switch (constraintIndex) {
            {{#each compiledProblem.constraints}}
            case {{{index}}}:
                return {{{violationExpression}}};
            {{/each}}
        }

        throw new Exception("invalid constraint index");
    }

    private class Functions extends AbstractFunctions {
        {{#each compiledProblem.functions}}
        {{{expression}}}

        {{/each}}
    }
}
```

Apéndice D

Clases desarrolladas para el resolutor de jMetal

D.1. Main

```java
package prodef;

import prodef.multiobjective.BinaryNSGAIIProblemRunner;
import prodef.multiobjective.DoubleNSGAIIProblemRunner;
import prodef.multiobjective.IntegerNSGAIIProblemRunner;
import prodef.multiobjective.IntegerPermutationNSGAIIProblemRunner;
import prodef.singleobjective.BinaryGeneticProblemRunner;
import prodef.singleobjective.DoubleGeneticProblemRunner;
import prodef.singleobjective.IntegerGeneticProblemRunner;
import prodef.singleobjective.IntegerPermutationGeneticProblemRunner;
import prodef.utils.Results;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.lang.reflect.Constructor;

public class Main {
    // Arguments example: prodef.ProdefProblemInstance binary singleobjective
    public static void main(String[] args) {
        try {
            String className = args[0];
            String problemType = args[1];
            boolean isSingleObjective = args[2].equalsIgnoreCase("singleobjective");

            Results results = null;
            AbstractProdefProblemInstance<Integer> integerProblemInstance;
            AbstractProdefProblemInstance<Double> realProblemInstance;

            switch (problemType) {
                case "binary":
                    integerProblemInstance = (AbstractProdefProblemInstance<Integer>)
                    ↪  createInstanceFromClassName(className);
                    results = executeBinaryProblem(integerProblemInstance,
                    ↪  isSingleObjective);
                    break;
                case "integer":
```

```
35                  integerProblemInstance = (AbstractProdefProblemInstance<Integer>)
                    ↪  createInstanceFromClassName(className);
36                  results = executeIntegerProblem(integerProblemInstance,
                    ↪  isSingleObjective);
37                  break;
38              case "real":
39                  realProblemInstance = (AbstractProdefProblemInstance<Double>)
                    ↪  createInstanceFromClassName(className);
40                  results = executeRealProblem(realProblemInstance,
                    ↪  isSingleObjective);
41                  break;
42              case "integerPermutation":
43                  integerProblemInstance = (AbstractProdefProblemInstance<Integer>)
                    ↪  createInstanceFromClassName(className);
44                  results = executeIntegerPermutationProblem(integerProblemInstance,
                    ↪  isSingleObjective);
45                  break;
46          }
47
48          if (results != null) {
49              BufferedWriter writer = new BufferedWriter(new
                ↪  FileWriter("results.json"));
50              results.generateJSON().write(writer, 2, 0);
51              writer.close();
52              System.out.println(results.generateJSON().toString(2));
53          }
54
55      } catch (Exception e) {
56          System.exit(10);
57      }
58  }
59
60  private static Object createInstanceFromClassName(String className) throws
    ↪  Exception {
61      Class<?> clazz = Class.forName(className);
62      Constructor<?> constructor = clazz.getConstructor();
63      return constructor.newInstance();
64  }
65
66  private static Results executeBinaryProblem(AbstractProdefProblemInstance<Integer>
    ↪  problemInstance, boolean isSingleObjective) {
67      if (isSingleObjective) {
68          BinaryGeneticProblemRunner binaryGeneticProblemRunner = new
            ↪  BinaryGeneticProblemRunner();
69          return binaryGeneticProblemRunner.run(problemInstance);
70      } else {
71          BinaryNSGAIIProblemRunner binaryNSGAIIProblemRunner = new
            ↪  BinaryNSGAIIProblemRunner();
72          return binaryNSGAIIProblemRunner.run(problemInstance);
73      }
74  }
75
76  private static Results executeIntegerProblem(AbstractProdefProblemInstance<Integer>
    ↪  problemInstance, boolean isSingleObjective) {
77      if (isSingleObjective) {
78          IntegerGeneticProblemRunner integerGeneticProblemRunner = new
            ↪  IntegerGeneticProblemRunner();
```

```
            return integerGeneticProblemRunner.run(problemInstance);
        } else {
            IntegerNSGAIIProblemRunner integerNSGAIIProblemRunner = new
            ↪  IntegerNSGAIIProblemRunner();
            return integerNSGAIIProblemRunner.run(problemInstance);
        }
    }

    private static Results executeRealProblem(AbstractProdefProblemInstance<Double>
    ↪  problemInstance, boolean isSingleObjective) {
        if (isSingleObjective) {
            DoubleGeneticProblemRunner doubleGeneticProblemRunner = new
            ↪  DoubleGeneticProblemRunner();
            return doubleGeneticProblemRunner.run(problemInstance);
        } else {
            DoubleNSGAIIProblemRunner doubleNSGAIIProblemRunner = new
            ↪  DoubleNSGAIIProblemRunner();
            return doubleNSGAIIProblemRunner.run(problemInstance);
        }
    }

    private static Results
    ↪  executeIntegerPermutationProblem(AbstractProdefProblemInstance<Integer>
    ↪  problemInstance, boolean isSingleObjective) {
        if (isSingleObjective) {
            IntegerPermutationGeneticProblemRunner
            ↪  integerPermutationGeneticProblemRunner = new
            ↪  IntegerPermutationGeneticProblemRunner();
            return integerPermutationGeneticProblemRunner.run(problemInstance);
        } else {
            IntegerPermutationNSGAIIProblemRunner integerPermutationNSGAIIProblemRunner
            ↪  = new IntegerPermutationNSGAIIProblemRunner();
            return integerPermutationNSGAIIProblemRunner.run(problemInstance);
        }
    }
}
```

D.2. AbstractProdefProblemInstance

```
package prodef;

import org.uma.jmetal.solution.Solution;
import org.uma.jmetal.solution.binarysolution.BinarySolution;
import prodef.utils.Variable;
import prodef.utils.VariableIndex;

import java.util.*;

public abstract class AbstractProdefProblemInstance<N extends Number> {
    protected AbstractFunctions prodef;
    private final List<Variable> variables = new ArrayList<>();
    private final Map<Integer, Variable> variablesMap = new HashMap<>();
```

```java
    private final Map<Integer, VariableIndex> variablesIndexMap = new HashMap<>();

    // ProdefProblem info
    protected boolean isBinary;
    protected boolean isIntegerPermutation;
    protected int numberOfVariables;
    protected int numberOfObjectives;
    protected int numberOfConstraints;
    protected int numberOfAtomicVariables;
    protected String problemName;

    // Execution configuration
    protected long maxComputingTime;
    protected int maxNumberOfResults;

    protected void buildVariablesMaps(List<Variable> variables) {
        this.variables.clear();
        this.variables.addAll(variables);
        variablesMap.clear();
        variablesIndexMap.clear();

        int index = 0;
        for (Variable variable : variables) {
            for (int i = 0; i < variable.getSize(); ++i, ++index) {
                variablesMap.put(index, variable);
                variablesIndexMap.put(index, variable.getVariableIndex(i));
            }
        }
    }

    public Solution<N> getPermutationCompensatedSolution(final Solution<N> solution) {
        Solution<N> compensatedSolution = solution.copy();
        for (int i = 0; i < compensatedSolution.getVariables().size(); i++) {
            compensatedSolution.setVariable(i,
            ↪  getNValueOf(compensatedSolution.getVariables().get(i).intValue() + 1));
        }
        return compensatedSolution;
    }

    public void updateVariables(final Solution<N> solution) {
        if (isIntegerPermutation()) {
            updateCompensatedVariables(getPermutationCompensatedSolution(solution));
        } else {
            updateCompensatedVariables(solution);
        }
    }

    public int getNumberOfVariables() {
        return numberOfVariables;
    }

    public boolean isBinary() {
        return isBinary;
    }

    public boolean isIntegerPermutation() {
        return isIntegerPermutation;
```

```
70      }
71
72      public int getNumberOfObjectives() {
73          return numberOfObjectives;
74      }
75
76      public int getNumberOfConstraints() {
77          return numberOfConstraints;
78      }
79
80      public int getNumberOfAtomicVariables() {
81          return numberOfAtomicVariables;
82      }
83
84      public String getProblemName() {
85          return problemName;
86      }
87
88      public long getMaxComputingTime() {
89          return maxComputingTime;
90      }
91
92      public int getMaxNumberOfResults() {
93          return maxNumberOfResults;
94      }
95
96      public List<Integer> getListOfBitsPerVariable() {
97          List<Integer> result = new ArrayList<>();
98
99          for (Variable variable : variables) {
100             result.add(variable.getSize());
101         }
102
103         return result;
104     }
105
106     public N getLowerLimitOfVariable(int i) {
107         return getNValueOf(variablesMap.get(i).getLowerBound());
108     }
109
110     public N getUpperLimitOfVariable(int i) {
111         return getNValueOf(variablesMap.get(i).getFixedUpperBound());
112     }
113
114     public abstract void updateBinaryVariables(final BinarySolution solution);
115
116     protected abstract void updateCompensatedVariables(final Solution<N> solution);
117
118     public abstract double evaluateGoal(int goalIndex) throws Exception;
119
120     public abstract boolean isMinimize(int goalIndex);
121
122     public abstract boolean evaluateConstraint(int constraintIndex) throws Exception;
123
124     public abstract double evaluateConstraintViolation(int constraintIndex) throws
    ↪  Exception;
125
```

```
126     public List<Variable> getVariables() {
127         return variables;
128     }
129 
130     public Map<Integer, Variable> getVariablesMap() {
131         return variablesMap;
132     }
133 
134     public Map<Integer, VariableIndex> getVariablesIndexMap() {
135         return variablesIndexMap;
136     }
137 
138     private N getNValueOf(Number number) {
139         return (N) number;
140     }
141 
142     public abstract static class AbstractFunctions {
143     }
144 }
```

D.3. Problemas

D.3.1. AbstractConstrainedBinaryProblem

```
1  package prodef.problem;
2 
3  import org.uma.jmetal.problem.binaryproblem.impl.AbstractBinaryProblem;
4  import org.uma.jmetal.solution.binarysolution.BinarySolution;
5  import prodef.solution.ConstrainedBinarySolution;
6 
7  public abstract class AbstractConstrainedBinaryProblem extends AbstractBinaryProblem {
8      @Override
9      public BinarySolution createSolution() {
10         return new ConstrainedBinarySolution(getListOfBitsPerVariable(),
           ↪  getNumberOfObjectives(), getNumberOfConstraints());
11     }
12 }
```

D.3.2. AbstractConstrainedDoubleProblem

```
1  package prodef.problem;
2 
3  import org.uma.jmetal.problem.doubleproblem.impl.AbstractDoubleProblem;
4  import org.uma.jmetal.solution.doublesolution.DoubleSolution;
5  import org.uma.jmetal.solution.doublesolution.impl.DefaultDoubleSolution;
6 
7  public abstract class AbstractConstrainedDoubleProblem extends AbstractDoubleProblem {
8      @Override
```

```
    public DoubleSolution createSolution() {
        return new DefaultDoubleSolution(bounds, getNumberOfObjectives(),
        ↪  getNumberOfConstraints());
    }
}
```

D.3.3. AbstractConstrainedIntegerPermutationProblem

```
package prodef.problem;

import org.uma.jmetal.problem.permutationproblem.impl.AbstractIntegerPermutationProblem;
import org.uma.jmetal.solution.permutationsolution.PermutationSolution;
import prodef.solution.ConstrainedIntegerPermutationSolution;

public abstract class AbstractConstrainedIntegerPermutationProblem extends
↪  AbstractIntegerPermutationProblem {
    @Override
    public PermutationSolution<Integer> createSolution() {
        return new ConstrainedIntegerPermutationSolution(getLength(),
        ↪  getNumberOfObjectives(), getNumberOfConstraints());
    }
}
```

D.3.4. AbstractConstrainedIntegerProblem

```
package prodef.problem;

import org.uma.jmetal.problem.integerproblem.impl.AbstractIntegerProblem;
import org.uma.jmetal.solution.integersolution.IntegerSolution;
import org.uma.jmetal.solution.integersolution.impl.DefaultIntegerSolution;

public abstract class AbstractConstrainedIntegerProblem extends AbstractIntegerProblem {
    @Override
    public IntegerSolution createSolution() {
        return new DefaultIntegerSolution(getVariableBounds(), getNumberOfObjectives(),
        ↪  getNumberOfConstraints());
    }
}
```

D.3.5. BinaryProdefProblem

```
package prodef.problem;

import org.uma.jmetal.problem.AbstractGenericProblem;
import org.uma.jmetal.solution.binarysolution.BinarySolution;
```

```java
import prodef.AbstractProdefProblemInstance;

import java.util.List;

public class BinaryProdefProblem extends AbstractConstrainedBinaryProblem implements
    ProdefProblem<Integer, BinarySolution> {
    private final AbstractProdefProblemInstance<Integer> problemInstance;

    public BinaryProdefProblem(AbstractProdefProblemInstance<Integer> problemInstance) {
        this.problemInstance = problemInstance;

        setNumberOfVariables(problemInstance.getNumberOfVariables());
        setNumberOfObjectives(problemInstance.getNumberOfObjectives());
        setNumberOfConstraints(problemInstance.getNumberOfConstraints());
        setName(problemInstance.getProblemName());
    }

    @Override
    public AbstractProdefProblemInstance<Integer> getProblemInstance() {
        return problemInstance;
    }

    @Override
    public long getMaxComputingTime() {
        return problemInstance.getMaxComputingTime();
    }

    @Override
    public AbstractGenericProblem<BinarySolution> getProblem() {
        return this;
    }

    @Override
    public void evaluate(BinarySolution solution) {
        problemInstance.updateBinaryVariables(solution);

        try {
            for (int i = 0; i < getNumberOfObjectives(); ++i) {
                solution.setObjective(i, (problemInstance.isMinimize(i) ? 1.0 : -1.0) *
                    problemInstance.evaluateGoal(i));
            }

            evaluateConstraints(solution);
        } catch (Exception e) {
            System.err.println(e.getMessage());
            System.exit(20);
        }
    }

    public void evaluateConstraints(BinarySolution solution) throws Exception {
        for (int i = 0; i < getNumberOfConstraints(); ++i) {
            double constraintViolation = problemInstance.evaluateConstraintViolation(i);
            solution.setConstraint(i, constraintViolation);
        }
    }

    @Override
```

```
    public List<Integer> getListOfBitsPerVariable() {
        return problemInstance.getListOfBitsPerVariable();
    }
}
```

D.3.6. DoubleProdefProblem

```
package prodef.problem;

import org.uma.jmetal.problem.AbstractGenericProblem;
import prodef.AbstractProdefProblemInstance;
import org.uma.jmetal.solution.doublesolution.DoubleSolution;

import java.util.ArrayList;
import java.util.List;

public class DoubleProdefProblem extends AbstractConstrainedDoubleProblem implements
↪  ProdefProblem<Double, DoubleSolution> {
    private final AbstractProdefProblemInstance<Double> problemInstance;

    public DoubleProdefProblem(AbstractProdefProblemInstance<Double> problemInstance) {
        this.problemInstance = problemInstance;

        setNumberOfVariables(problemInstance.getNumberOfAtomicVariables());
        setNumberOfObjectives(problemInstance.getNumberOfObjectives());
        setNumberOfConstraints(problemInstance.getNumberOfConstraints());
        setName(problemInstance.getProblemName());

        List<Double> lowerLimit = new ArrayList<>();
        List<Double> upperLimit = new ArrayList<>();

        for (int i = 0; i < getNumberOfVariables(); ++i) {
            lowerLimit.add(problemInstance.getLowerLimitOfVariable(i));
            upperLimit.add(problemInstance.getUpperLimitOfVariable(i));
        }

        setVariableBounds(lowerLimit, upperLimit);
    }

    @Override
    public AbstractProdefProblemInstance<Double> getProblemInstance() {
        return problemInstance;
    }

    @Override
    public long getMaxComputingTime() {
        return problemInstance.getMaxComputingTime();
    }

    @Override
    public AbstractGenericProblem<DoubleSolution> getProblem() {
        return this;
    }
```

```
46
47      @Override
48      public void evaluate(DoubleSolution solution) {
49          problemInstance.updateVariables(solution);
50
51          try {
52              for (int i = 0; i < getNumberOfObjectives(); ++i) {
53                  solution.setObjective(i, (problemInstance.isMinimize(i) ? 1.0 : -1.0) *
                    ↪  problemInstance.evaluateGoal(i));
54              }
55
56              evaluateConstraints(solution);
57          } catch (Exception e) {
58              System.err.println(e.getMessage());
59              System.exit(20);
60          }
61      }
62
63      public void evaluateConstraints(DoubleSolution solution) throws Exception {
64          for (int i = 0; i < getNumberOfConstraints(); ++i) {
65              double constraintViolation = problemInstance.evaluateConstraintViolation(i);
66              solution.setConstraint(i, constraintViolation);
67          }
68      }
69  }
```

D.3.7. IntegerPermutationProdefProblem

```
1   package prodef.problem;
2
3   import org.uma.jmetal.problem.AbstractGenericProblem;
4   import org.uma.jmetal.solution.permutationsolution.PermutationSolution;
5   import prodef.AbstractProdefProblemInstance;
6
7   public class IntegerPermutationProdefProblem extends
    ↪  AbstractConstrainedIntegerPermutationProblem implements ProdefProblem<Integer,
    ↪  PermutationSolution<Integer>> {
8       private final AbstractProdefProblemInstance<Integer> problemInstance;
9
10      public IntegerPermutationProdefProblem(AbstractProdefProblemInstance<Integer>
        ↪  problemInstance) {
11          this.problemInstance = problemInstance;
12
13          setNumberOfVariables(problemInstance.getNumberOfAtomicVariables());
14          setNumberOfObjectives(problemInstance.getNumberOfObjectives());
15          setNumberOfConstraints(problemInstance.getNumberOfConstraints());
16          setName(problemInstance.getProblemName());
17      }
18
19      @Override
20      public AbstractProdefProblemInstance<Integer> getProblemInstance() {
21          return problemInstance;
22      }
```

```
23
24      @Override
25      public long getMaxComputingTime() {
26          return problemInstance.getMaxComputingTime();
27      }
28
29      @Override
30      public AbstractGenericProblem<PermutationSolution<Integer>> getProblem() {
31          return this;
32      }
33
34      @Override
35      public void evaluate(PermutationSolution<Integer> solution) {
36          problemInstance.updateVariables(solution);
37
38          try {
39              for (int i = 0; i < getNumberOfObjectives(); ++i) {
40                  solution.setObjective(i, (problemInstance.isMinimize(i) ? 1.0 : -1.0) *
                    ↪  problemInstance.evaluateGoal(i));
41              }
42
43              evaluateConstraints(solution);
44          } catch (Exception e) {
45              System.err.println(e.getMessage());
46              System.exit(20);
47          }
48      }
49
50      public void evaluateConstraints(PermutationSolution<Integer> solution) throws
        ↪  Exception {
51          for (int i = 0; i < getNumberOfConstraints(); ++i) {
52              double constraintViolation = problemInstance.evaluateConstraintViolation(i);
53              solution.setConstraint(i, constraintViolation);
54          }
55      }
56
57      @Override
58      public int getLength() {
59          return problemInstance.getNumberOfAtomicVariables();
60      }
61  }
```

D.3.8. IntegerProdefProblem

```
1   package prodef.problem;
2
3   import org.uma.jmetal.problem.AbstractGenericProblem;
4   import org.uma.jmetal.solution.integersolution.IntegerSolution;
5   import prodef.AbstractProdefProblemInstance;
6
7   import java.util.ArrayList;
8   import java.util.List;
9
```

```java
public class IntegerProdefProblem extends AbstractConstrainedIntegerProblem implements ProdefProblem<Integer, IntegerSolution> {
    private final AbstractProdefProblemInstance<Integer> problemInstance;

    public IntegerProdefProblem(AbstractProdefProblemInstance<Integer> problemInstance) {
        this.problemInstance = problemInstance;

        setNumberOfVariables(problemInstance.getNumberOfAtomicVariables());
        setNumberOfObjectives(problemInstance.getNumberOfObjectives());
        setNumberOfConstraints(problemInstance.getNumberOfConstraints());
        setName(problemInstance.getProblemName());

        List<Integer> lowerLimit = new ArrayList<>();
        List<Integer> upperLimit = new ArrayList<>();

        for (int i = 0; i < getNumberOfVariables(); ++i) {
            lowerLimit.add(problemInstance.getLowerLimitOfVariable(i));
            upperLimit.add(problemInstance.getUpperLimitOfVariable(i));
        }

        setVariableBounds(lowerLimit, upperLimit);
    }

    @Override
    public AbstractProdefProblemInstance<Integer> getProblemInstance() {
        return problemInstance;
    }

    @Override
    public long getMaxComputingTime() {
        return problemInstance.getMaxComputingTime();
    }

    @Override
    public AbstractGenericProblem<IntegerSolution> getProblem() {
        return this;
    }

    @Override
    public void evaluate(IntegerSolution solution) {
        problemInstance.updateVariables(solution);

        try {
            for (int i = 0; i < getNumberOfObjectives(); ++i) {
                solution.setObjective(i, (problemInstance.isMinimize(i) ? 1.0 : -1.0) * problemInstance.evaluateGoal(i));
            }

            evaluateConstraints(solution);
        } catch (Exception e) {
            System.err.println(e.getMessage());
            System.exit(20);
        }
    }

    public void evaluateConstraints(IntegerSolution solution) throws Exception {
```

```
        for (int i = 0; i < getNumberOfConstraints(); ++i) {
            double constraintViolation = problemInstance.evaluateConstraintViolation(i);
            solution.setConstraint(i, constraintViolation);
        }
    }
}
```

D.3.9. ProdefProblem

```
package prodef.problem;

import org.uma.jmetal.problem.AbstractGenericProblem;
import org.uma.jmetal.solution.Solution;
import prodef.AbstractProdefProblemInstance;

public interface ProdefProblem<N extends Number, S extends Solution<?>> {
    AbstractProdefProblemInstance<N> getProblemInstance();

    long getMaxComputingTime();

    AbstractGenericProblem<S> getProblem();
}
```

D.4. Soluciones

D.4.1. ConstrainedBinarySolution

```
package prodef.solution;

import org.uma.jmetal.solution.AbstractSolution;
import org.uma.jmetal.solution.binarysolution.BinarySolution;
import org.uma.jmetal.util.binarySet.BinarySet;
import org.uma.jmetal.util.pseudorandom.JMetalRandom;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class ConstrainedBinarySolution extends AbstractSolution<BinarySet> implements
↪  BinarySolution {
    protected List<Integer> bitsPerVariable;

    /**
     * Constructor
     */
    public ConstrainedBinarySolution(List<Integer> bitsPerVariable, int
    ↪  numberOfObjectives, int numberOfConstraints) {
        super(bitsPerVariable.size(), numberOfObjectives, numberOfConstraints);
```

```
20          this.bitsPerVariable = bitsPerVariable;
21
22          initializeBinaryVariables(JMetalRandom.getInstance());
23      }
24
25      /**
26       * Copy constructor
27       */
28      public ConstrainedBinarySolution(ConstrainedBinarySolution solution) {
29          super(solution.getNumberOfVariables(), solution.getNumberOfObjectives(),
            ↪  solution.getNumberOfConstraints());
30
31          this.bitsPerVariable = solution.bitsPerVariable;
32
33          for (int i = 0; i < getNumberOfVariables(); i++) {
34              setVariable(i, (BinarySet) solution.getVariable(i).clone());
35          }
36
37          for (int i = 0; i < getNumberOfObjectives(); i++) {
38              setObjective(i, solution.getObjective(i));
39          }
40
41          for (int i = 0; i < getNumberOfConstraints(); i++) {
42              setConstraint(i, solution.getConstraint(i));
43          }
44
45          attributes = new HashMap<Object, Object>(solution.attributes);
46      }
47
48      private static BinarySet createNewBitSet(int numberOfBits, JMetalRandom
        ↪  randomGenerator) {
49          BinarySet bitSet = new BinarySet(numberOfBits);
50
51          for (int i = 0; i < numberOfBits; i++) {
52              double rnd = randomGenerator.nextDouble();
53              if (rnd < 0.5) {
54                  bitSet.set(i);
55              } else {
56                  bitSet.clear(i);
57              }
58          }
59          return bitSet;
60      }
61
62      @Override
63      public int getNumberOfBits(int index) {
64          return getVariable(index).getBinarySetLength();
65      }
66
67      @Override
68      public ConstrainedBinarySolution copy() {
69          return new ConstrainedBinarySolution(this);
70      }
71
72      @Override
73      public int getTotalNumberOfBits() {
74          int sum = 0;
```

```
        for (int i = 0; i < getNumberOfVariables(); i++) {
            sum += getVariable(i).getBinarySetLength();
        }

        return sum;
    }

    private void initializeBinaryVariables(JMetalRandom randomGenerator) {
        for (int i = 0; i < getNumberOfVariables(); i++) {
            setVariable(i, createNewBitSet(bitsPerVariable.get(i), randomGenerator));
        }
    }

    @Override
    public Map<Object, Object> getAttributes() {
        return attributes;
    }
}
```

D.4.2. ConstrainedIntegerPermutationSolution

```
package prodef.solution;

import org.uma.jmetal.solution.AbstractSolution;
import org.uma.jmetal.solution.permutationsolution.PermutationSolution;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class ConstrainedIntegerPermutationSolution extends AbstractSolution<Integer>
        implements PermutationSolution<Integer> {

    /**
     * Constructor
     */
    public ConstrainedIntegerPermutationSolution(int permutationLength, int
    ↪  numberOfObjectives, int numberOfConstraints) {
        super(permutationLength, numberOfObjectives, numberOfConstraints);

        List<Integer> randomSequence = new ArrayList<>(permutationLength);

        for (int j = 0; j < permutationLength; j++) {
            randomSequence.add(j);
        }

        java.util.Collections.shuffle(randomSequence);

        for (int i = 0; i < permutationLength; i++) {
            setVariable(i, randomSequence.get(i));
        }
    }
```

```
32
33      /**
34       * Copy Constructor
35       */
36      public ConstrainedIntegerPermutationSolution(ConstrainedIntegerPermutationSolution
        ↪  solution) {
37          super(solution.getLength(), solution.getNumberOfObjectives(),
            ↪  solution.getNumberOfConstraints());
38
39          for (int i = 0; i < getNumberOfObjectives(); i++) {
40              setObjective(i, solution.getObjective(i));
41          }
42
43          for (int i = 0; i < getNumberOfVariables(); i++) {
44              setVariable(i, solution.getVariable(i));
45          }
46
47          for (int i = 0; i < getNumberOfConstraints(); i++) {
48              setConstraint(i, solution.getConstraint(i));
49          }
50
51          attributes = new HashMap<Object, Object>(solution.attributes);
52      }
53
54      @Override
55      public ConstrainedIntegerPermutationSolution copy() {
56          return new ConstrainedIntegerPermutationSolution(this);
57      }
58
59      @Override
60      public Map<Object, Object> getAttributes() {
61          return attributes;
62      }
63
64      @Override
65      public int getLength() {
66          return getNumberOfVariables();
67      }
68  }
```

D.5. Runners

D.5.1. AbstractProblemRunner

```
1   package prodef.runner;
2
3   import org.uma.jmetal.operator.crossover.CrossoverOperator;
4   import org.uma.jmetal.operator.mutation.MutationOperator;
5   import org.uma.jmetal.solution.Solution;
6   import org.uma.jmetal.util.AbstractAlgorithmRunner;
7   import prodef.AbstractProdefProblemInstance;
8   import prodef.problem.ProdefProblem;
```

```java
import prodef.utils.Results;

public abstract class AbstractProblemRunner<N extends Number, S extends Solution<?>>
    extends AbstractAlgorithmRunner implements Runner<N, S> {
    public abstract Results<N, S> run(AbstractProdefProblemInstance<N> problemInstance);

    protected abstract ProdefProblem<N, S> getProblem(AbstractProdefProblemInstance<N>
    problemInstance);

    protected abstract CrossoverOperator<S> getCrossoverOperator(ProdefProblem<N, S>
    prodefProblem);

    protected abstract MutationOperator<S> getMutationOperator(ProdefProblem<N, S>
    prodefProblem);
}
```

D.5.2. Runner

```java
package prodef.runner;

import org.uma.jmetal.solution.Solution;
import prodef.AbstractProdefProblemInstance;
import prodef.utils.Results;

public interface Runner<N extends Number, S extends Solution<?>> {
    Results<N, S> run(AbstractProdefProblemInstance<N> problemInstance);
}
```

D.5.3. RunnersConfiguration

```java
package prodef.runner;

public class RunnersConfiguration {
    public static final int POPULATION = 100;
    public static final double DISTRIBUTION_INDEX = 20.0;
    public static final double CROSSOVER_PROBABILITY = 0.9;
}
```

D.6. Runners para problemas multi-objetivo

D.6.1. AbstractNSGAIIProblemRunner

```java
package prodef.multiobjective;

```

```java
import org.uma.jmetal.algorithm.multiobjective.nsgaii.NSGAII;
import org.uma.jmetal.component.ranking.impl.FastNonDominatedSortRanking;
import org.uma.jmetal.component.termination.Termination;
import org.uma.jmetal.component.termination.impl.TerminationByComputingTime;
import org.uma.jmetal.solution.Solution;
import org.uma.jmetal.util.fileoutput.SolutionListOutput;
import org.uma.jmetal.util.fileoutput.impl.DefaultFileOutputContext;
import prodef.AbstractProdefProblemInstance;
import prodef.runner.RunnersConfiguration;
import prodef.problem.ProdefProblem;
import prodef.runner.AbstractProblemRunner;
import prodef.utils.Results;

import java.util.Comparator;
import java.util.List;

public abstract class AbstractNSGAIIProblemRunner<N extends Number, S extends
    Solution<?>> extends AbstractProblemRunner<N, S> {
    public Results<N, S> run(AbstractProdefProblemInstance<N> problemInstance) {
        ProdefProblem<N, S> prodefProblem = getProblem(problemInstance);

        Comparator<S> comparator = new
            FastNonDominatedSortRanking<S>().getSolutionComparator();

        Termination termination = new TerminationByComputingTime((int)
            prodefProblem.getMaxComputingTime());

        NSGAII<S> algorithm = new NSGAII<>(
                prodefProblem.getProblem(), RunnersConfiguration.POPULATION,
                    RunnersConfiguration.POPULATION,
                getCrossoverOperator(prodefProblem),
                    getMutationOperator(prodefProblem), termination
        );
        algorithm.run();

        List<S> population = algorithm.getResult();
        long computingTime = algorithm.getTotalComputingTime();

        new SolutionListOutput(population)
                .setVarFileOutputContext(new DefaultFileOutputContext("VAR.tsv"))
                .setFunFileOutputContext(new DefaultFileOutputContext("FUN.tsv"))
                .print();

        return new Results<>(prodefProblem.getProblemInstance(), population,
            computingTime, comparator);
    }
}
```

D.6.2. BinaryNSGAIIProblemRunner

```java
package prodef.multiobjective;

import org.uma.jmetal.operator.crossover.CrossoverOperator;
```

```java
import org.uma.jmetal.operator.crossover.impl.SinglePointCrossover;
import org.uma.jmetal.operator.mutation.MutationOperator;
import org.uma.jmetal.operator.mutation.impl.BitFlipMutation;
import org.uma.jmetal.solution.binarysolution.BinarySolution;
import prodef.AbstractProdefProblemInstance;
import prodef.runner.RunnersConfiguration;
import prodef.problem.BinaryProdefProblem;
import prodef.problem.ProdefProblem;

public class BinaryNSGAIIProblemRunner extends AbstractNSGAIIProblemRunner<Integer, BinarySolution> {
    @Override
    protected ProdefProblem<Integer, BinarySolution> getProblem(AbstractProdefProblemInstance<Integer> problemInstance) {
        return new BinaryProdefProblem(problemInstance);
    }

    @Override
    protected CrossoverOperator<BinarySolution> getCrossoverOperator(ProdefProblem<Integer, BinarySolution> prodefProblem) {
        return new SinglePointCrossover(RunnersConfiguration.CROSSOVER_PROBABILITY);
    }

    @Override
    protected MutationOperator<BinarySolution> getMutationOperator(ProdefProblem<Integer, BinarySolution> prodefProblem) {
        double mutationProbability = 1.0 / prodefProblem.getProblemInstance().getNumberOfAtomicVariables();
        return new BitFlipMutation(mutationProbability);
    }
}
```

D.6.3. DoubleNSGAIIProblemRunner

```java
package prodef.multiobjective;

import org.uma.jmetal.operator.crossover.CrossoverOperator;
import org.uma.jmetal.operator.crossover.impl.SBXCrossover;
import org.uma.jmetal.operator.mutation.MutationOperator;
import org.uma.jmetal.operator.mutation.impl.PolynomialMutation;
import org.uma.jmetal.solution.doublesolution.DoubleSolution;
import prodef.AbstractProdefProblemInstance;
import prodef.runner.RunnersConfiguration;
import prodef.problem.DoubleProdefProblem;
import prodef.problem.ProdefProblem;

public class DoubleNSGAIIProblemRunner extends AbstractNSGAIIProblemRunner<Double, DoubleSolution> {
    @Override
    protected ProdefProblem<Double, DoubleSolution> getProblem(AbstractProdefProblemInstance<Double> problemInstance) {
        return new DoubleProdefProblem(problemInstance);
    }
```

```
18
19      @Override
20      protected CrossoverOperator<DoubleSolution>
        ↪  getCrossoverOperator(ProdefProblem<Double, DoubleSolution> prodefProblem) {
21          return new SBXCrossover(RunnersConfiguration.CROSSOVER_PROBABILITY,
            ↪  RunnersConfiguration.DISTRIBUTION_INDEX);
22      }
23
24      @Override
25      protected MutationOperator<DoubleSolution>
        ↪  getMutationOperator(ProdefProblem<Double, DoubleSolution> prodefProblem) {
26          double mutationProbability = 1.0 /
            ↪  prodefProblem.getProblem().getNumberOfVariables();
27          return new PolynomialMutation(mutationProbability,
            ↪  RunnersConfiguration.DISTRIBUTION_INDEX);
28      }
29  }
```

D.6.4. IntegerNSGAIIProblemRunner

```
1   package prodef.multiobjective;
2
3   import org.uma.jmetal.operator.crossover.CrossoverOperator;
4   import org.uma.jmetal.operator.crossover.impl.IntegerSBXCrossover;
5   import org.uma.jmetal.operator.mutation.MutationOperator;
6   import org.uma.jmetal.operator.mutation.impl.IntegerPolynomialMutation;
7   import org.uma.jmetal.solution.integersolution.IntegerSolution;
8   import prodef.AbstractProdefProblemInstance;
9   import prodef.runner.RunnersConfiguration;
10  import prodef.problem.IntegerProdefProblem;
11  import prodef.problem.ProdefProblem;
12
13  public class IntegerNSGAIIProblemRunner extends AbstractNSGAIIProblemRunner<Integer,
    ↪  IntegerSolution> {
14      @Override
15      protected ProdefProblem<Integer, IntegerSolution>
        ↪  getProblem(AbstractProdefProblemInstance<Integer> problemInstance) {
16          return new IntegerProdefProblem(problemInstance);
17      }
18
19      @Override
20      protected CrossoverOperator<IntegerSolution>
        ↪  getCrossoverOperator(ProdefProblem<Integer, IntegerSolution> prodefProblem) {
21          return new IntegerSBXCrossover(RunnersConfiguration.CROSSOVER_PROBABILITY,
            ↪  RunnersConfiguration.DISTRIBUTION_INDEX);
22      }
23
24      @Override
25      protected MutationOperator<IntegerSolution>
        ↪  getMutationOperator(ProdefProblem<Integer, IntegerSolution> prodefProblem) {
26          double mutationProbability = 1.0 /
            ↪  prodefProblem.getProblem().getNumberOfVariables();
27          return new IntegerPolynomialMutation(mutationProbability,
            ↪  RunnersConfiguration.DISTRIBUTION_INDEX);
```

```
    }
}
```

D.6.5. IntegerPermutationNSGAIIProblemRunner

```
package prodef.multiobjective;

import org.uma.jmetal.operator.crossover.CrossoverOperator;
import org.uma.jmetal.operator.crossover.impl.PMXCrossover;
import org.uma.jmetal.operator.mutation.MutationOperator;
import org.uma.jmetal.operator.mutation.impl.PermutationSwapMutation;
import org.uma.jmetal.solution.permutationsolution.PermutationSolution;
import prodef.AbstractProdefProblemInstance;
import prodef.runner.RunnersConfiguration;
import prodef.problem.IntegerPermutationProdefProblem;
import prodef.problem.ProdefProblem;

public class IntegerPermutationNSGAIIProblemRunner extends
↪  AbstractNSGAIIProblemRunner<Integer, PermutationSolution<Integer>> {
    @Override
    protected ProdefProblem<Integer, PermutationSolution<Integer>>
    ↪  getProblem(AbstractProdefProblemInstance<Integer> problemInstance) {
        return new IntegerPermutationProdefProblem(problemInstance);
    }

    @Override
    protected CrossoverOperator<PermutationSolution<Integer>>
    ↪  getCrossoverOperator(ProdefProblem<Integer, PermutationSolution<Integer>>
    ↪  prodefProblem) {
        return new PMXCrossover(RunnersConfiguration.CROSSOVER_PROBABILITY);
    }

    @Override
    protected MutationOperator<PermutationSolution<Integer>>
    ↪  getMutationOperator(ProdefProblem<Integer, PermutationSolution<Integer>>
    ↪  prodefProblem) {
        double mutationProbability = 1.0 /
        ↪  prodefProblem.getProblem().getNumberOfVariables();
        return new PermutationSwapMutation<>(mutationProbability);
    }
}
```

D.7. Runners para problemas mono-objetivo

D.7.1. AbstractGeneticProblemRunner

```
package prodef.singleobjective;

```

```java
import org.uma.jmetal.algorithm.singleobjective.geneticalgorithm.GenerationalGeneticAlgorithm;
import org.uma.jmetal.algorithm.singleobjective.geneticalgorithm.GeneticAlgorithmBuilder;
import org.uma.jmetal.example.AlgorithmRunner;
import org.uma.jmetal.operator.selection.SelectionOperator;
import org.uma.jmetal.solution.Solution;
import org.uma.jmetal.util.comparator.DominanceComparator;
import org.uma.jmetal.util.fileoutput.SolutionListOutput;
import org.uma.jmetal.util.fileoutput.impl.DefaultFileOutputContext;
import prodef.AbstractProdefProblemInstance;
import prodef.runner.RunnersConfiguration;
import prodef.problem.ProdefProblem;
import prodef.runner.AbstractProblemRunner;
import prodef.utils.Results;

import java.util.Comparator;
import java.util.List;

public abstract class AbstractGeneticProblemRunner<N extends Number, S extends Solution<?>> extends AbstractProblemRunner<N, S> {
    public Results<N, S> run(AbstractProdefProblemInstance<N> problemInstance) {
        ProdefProblem<N, S> prodefProblem = getProblem(problemInstance);

        Comparator<S> comparator = new DominanceComparator<>();

        GeneticAlgorithmBuilder.GeneticAlgorithmVariant variant = GeneticAlgorithmBuilder.GeneticAlgorithmVariant.GENERATIONAL;

        GenerationalGeneticAlgorithm<S> algorithm =
                (GenerationalGeneticAlgorithm<S>) new ConstrainedGeneticAlgorithmBuilder<>(
                        prodefProblem.getProblem(), getCrossoverOperator(prodefProblem), getMutationOperator(prodefProblem))
                        .setPopulationSize(RunnersConfiguration.POPULATION)
                        .setMaxComputingTime(prodefProblem.getMaxComputingTime())
                        .setSelectionOperator(getSelectionOperator(prodefProblem))
                        .setComparator(comparator)
                        .setVariant(variant)
                        .build();

        AlgorithmRunner algorithmRunner = new AlgorithmRunner.Executor(algorithm).execute();
        List<S> population = algorithm.getPopulation();
        long computingTime = algorithmRunner.getComputingTime();

        new SolutionListOutput(population)
                .setVarFileOutputContext(new DefaultFileOutputContext("VAR.tsv"))
                .setFunFileOutputContext(new DefaultFileOutputContext("FUN.tsv"))
                .print();

        return new Results<>(prodefProblem.getProblemInstance(), population, computingTime, comparator);
    }

    protected abstract SelectionOperator<List<S>, S> getSelectionOperator(ProdefProblem<N, S> prodefProblem);
```

```
}
```

D.7.2. BinaryGeneticProblemRunner

```
package prodef.singleobjective;

import org.uma.jmetal.operator.crossover.CrossoverOperator;
import org.uma.jmetal.operator.crossover.impl.SinglePointCrossover;
import org.uma.jmetal.operator.mutation.MutationOperator;
import org.uma.jmetal.operator.mutation.impl.BitFlipMutation;
import org.uma.jmetal.operator.selection.SelectionOperator;
import org.uma.jmetal.operator.selection.impl.BinaryTournamentSelection;
import org.uma.jmetal.solution.binarysolution.BinarySolution;
import prodef.AbstractProdefProblemInstance;
import prodef.problem.BinaryProdefProblem;
import prodef.problem.ProdefProblem;
import prodef.runner.RunnersConfiguration;

import java.util.List;

public class BinaryGeneticProblemRunner extends AbstractGeneticProblemRunner<Integer,
↪  BinarySolution> {
    @Override
    protected ProdefProblem<Integer, BinarySolution>
    ↪  getProblem(AbstractProdefProblemInstance<Integer> problemInstance) {
        return new BinaryProdefProblem(problemInstance);
    }

    @Override
    protected CrossoverOperator<BinarySolution>
    ↪  getCrossoverOperator(ProdefProblem<Integer, BinarySolution> prodefProblem) {
        return new SinglePointCrossover(RunnersConfiguration.CROSSOVER_PROBABILITY);
    }

    @Override
    protected MutationOperator<BinarySolution>
    ↪  getMutationOperator(ProdefProblem<Integer, BinarySolution> prodefProblem) {
        double mutationProbability = 1.0 /
        ↪  prodefProblem.getProblemInstance().getNumberOfAtomicVariables();
        return new BitFlipMutation(mutationProbability);
    }

    @Override
    protected SelectionOperator<List<BinarySolution>, BinarySolution>
    ↪  getSelectionOperator(ProdefProblem<Integer, BinarySolution> prodefProblem) {
        return new BinaryTournamentSelection<>();
    }
}
```

D.7.3. ConstrainedGenerationalGeneticAlgorithm

```
1  package prodef.singleobjective;
2
3  import org.uma.jmetal.algorithm.singleobjective.geneticalgorithm.GenerationalGeneticAlg
   ↪  orithm;
4  import org.uma.jmetal.operator.crossover.CrossoverOperator;
5  import org.uma.jmetal.operator.mutation.MutationOperator;
6  import org.uma.jmetal.operator.selection.SelectionOperator;
7  import org.uma.jmetal.problem.Problem;
8  import org.uma.jmetal.solution.Solution;
9  import org.uma.jmetal.util.comparator.DominanceComparator;
10 import org.uma.jmetal.util.evaluator.SolutionListEvaluator;
11
12 import java.util.Comparator;
13 import java.util.List;
14
15 public class ConstrainedGenerationalGeneticAlgorithm<S extends Solution<?>> extends
   ↪  GenerationalGeneticAlgorithm<S> {
16     protected final Comparator<S> comparator;
17     protected final long maxComputingTime;
18     protected long initTime;
19     protected long computingTime;
20
21     public ConstrainedGenerationalGeneticAlgorithm(Problem<S> problem, long
       ↪  maxComputingTime, int populationSize,
22                                                    CrossoverOperator<S>
                                                      ↪  crossoverOperator,
                                                      ↪  MutationOperator<S>
                                                      ↪  mutationOperator,
23                                                    SelectionOperator<List<S>, S>
                                                      ↪  selectionOperator,
                                                      ↪  SolutionListEvaluator<S>
                                                      ↪  evaluator, Comparator<S>
                                                      ↪  comparator) {
24         super(problem, Integer.MAX_VALUE, populationSize, crossoverOperator,
           ↪  mutationOperator, selectionOperator, evaluator);
25
26         this.comparator = comparator;
27         this.maxComputingTime = maxComputingTime;
28     }
29
30     public ConstrainedGenerationalGeneticAlgorithm(Problem<S> problem, long
       ↪  maxComputingTime, int populationSize,
31                                                    CrossoverOperator<S>
                                                      ↪  crossoverOperator,
                                                      ↪  MutationOperator<S>
                                                      ↪  mutationOperator,
32                                                    SelectionOperator<List<S>, S>
                                                      ↪  selectionOperator,
                                                      ↪  SolutionListEvaluator<S>
                                                      ↪  evaluator) {
33         this(problem, maxComputingTime, populationSize, crossoverOperator,
           ↪  mutationOperator, selectionOperator, evaluator, new
           ↪  DominanceComparator<>());
34     }
35
```

```
    @Override
    protected List<S> replacement(List<S> population, List<S> offspringPopulation) {
        population.sort(comparator);
        offspringPopulation.add(population.get(0));
        offspringPopulation.add(population.get(1));
        offspringPopulation.sort(comparator);
        offspringPopulation.remove(offspringPopulation.size() - 1);
        offspringPopulation.remove(offspringPopulation.size() - 1);

        return offspringPopulation;
    }

    @Override
    protected boolean isStoppingConditionReached() {
        return (computingTime >= maxComputingTime);
    }

    @Override
    public void initProgress() {
        initTime = System.currentTimeMillis();
    }

    @Override
    public void updateProgress() {
        computingTime = System.currentTimeMillis() - initTime;
    }

    @Override
    public S getResult() {
        getPopulation().sort(comparator);
        return getPopulation().get(0);
    }
}
```

D.7.4. ConstrainedGeneticAlgorithmBuilder

```
package prodef.singleobjective;

import org.uma.jmetal.algorithm.Algorithm;
import
↪  org.uma.jmetal.algorithm.singleobjective.geneticalgorithm.GeneticAlgorithmBuilder;
import org.uma.jmetal.operator.crossover.CrossoverOperator;
import org.uma.jmetal.operator.mutation.MutationOperator;
import org.uma.jmetal.operator.selection.SelectionOperator;
import org.uma.jmetal.problem.Problem;
import org.uma.jmetal.solution.Solution;
import org.uma.jmetal.util.JMetalException;
import org.uma.jmetal.util.comparator.DominanceComparator;
import org.uma.jmetal.util.evaluator.SolutionListEvaluator;

import java.util.Comparator;
import java.util.List;

```

```java
public class ConstrainedGeneticAlgorithmBuilder<S extends Solution<?>> extends
    GeneticAlgorithmBuilder<S> {
    protected Comparator<S> comparator;
    protected long maxComputingTime;

    public ConstrainedGeneticAlgorithmBuilder(Problem<S> problem,
                                              CrossoverOperator<S> crossoverOperator,
                                              MutationOperator<S> mutationOperator) {
        super(problem, crossoverOperator, mutationOperator);
        this.comparator = new DominanceComparator<>();
    }

    public ConstrainedGeneticAlgorithmBuilder<S> setComparator(Comparator<S>
        comparator) {
        this.comparator = comparator;

        return this;
    }

    public ConstrainedGeneticAlgorithmBuilder<S> setMaxComputingTime(long
        maxComputingTime) {
        this.maxComputingTime = maxComputingTime;
        return this;
    }

    @Override
    public ConstrainedGeneticAlgorithmBuilder<S> setMaxEvaluations(int maxEvaluations) {
        super.setMaxEvaluations(maxEvaluations);
        return this;
    }

    @Override
    public ConstrainedGeneticAlgorithmBuilder<S> setPopulationSize(int populationSize) {
        super.setPopulationSize(populationSize);
        return this;
    }

    @Override
    public ConstrainedGeneticAlgorithmBuilder<S>
        setSelectionOperator(SelectionOperator<List<S>, S> selectionOperator) {
        super.setSelectionOperator(selectionOperator);
        return this;
    }

    @Override
    public ConstrainedGeneticAlgorithmBuilder<S>
        setSolutionListEvaluator(SolutionListEvaluator<S> evaluator) {
        super.setSolutionListEvaluator(evaluator);
        return this;
    }

    @Override
    public ConstrainedGeneticAlgorithmBuilder<S> setVariant(GeneticAlgorithmVariant
        variant) {
        super.setVariant(variant);
        return this;
    }
```

```
68
69      @Override
70      public Algorithm<S> build() {
71          if (getVariant() == GeneticAlgorithmVariant.GENERATIONAL) {
72              return new ConstrainedGenerationalGeneticAlgorithm<S>(getProblem(),
                ↪  getMaxComputingTime(), getPopulationSize(),
73                      getCrossoverOperator(), getMutationOperator(),
                        ↪  getSelectionOperator(), getEvaluator(), getComparator());
74          } else {
75              throw new JMetalException("Unknown or invalid variant: " + getVariant());
76          }
77      }
78
79      public Comparator<S> getComparator() {
80          return comparator;
81      }
82
83      public long getMaxComputingTime() {
84          return maxComputingTime;
85      }
86  }
```

D.7.5. DoubleGeneticProblemRunner

```
1   package prodef.singleobjective;
2
3   import org.uma.jmetal.operator.crossover.CrossoverOperator;
4   import org.uma.jmetal.operator.crossover.impl.SBXCrossover;
5   import org.uma.jmetal.operator.mutation.MutationOperator;
6   import org.uma.jmetal.operator.mutation.impl.PolynomialMutation;
7   import org.uma.jmetal.operator.selection.SelectionOperator;
8   import org.uma.jmetal.operator.selection.impl.BinaryTournamentSelection;
9   import org.uma.jmetal.solution.doublesolution.DoubleSolution;
10  import prodef.AbstractProdefProblemInstance;
11  import prodef.problem.DoubleProdefProblem;
12  import prodef.problem.ProdefProblem;
13  import prodef.runner.RunnersConfiguration;
14
15  import java.util.List;
16
17  public class DoubleGeneticProblemRunner extends AbstractGeneticProblemRunner<Double,
    ↪  DoubleSolution> {
18      @Override
19      protected ProdefProblem<Double, DoubleSolution>
        ↪  getProblem(AbstractProdefProblemInstance<Double> problemInstance) {
20          return new DoubleProdefProblem(problemInstance);
21      }
22
23      @Override
24      protected CrossoverOperator<DoubleSolution>
        ↪  getCrossoverOperator(ProdefProblem<Double, DoubleSolution> prodefProblem) {
25          return new SBXCrossover(RunnersConfiguration.CROSSOVER_PROBABILITY,
            ↪  RunnersConfiguration.DISTRIBUTION_INDEX);
```

```
    }

    @Override
    protected MutationOperator<DoubleSolution>
    ↪ getMutationOperator(ProdefProblem<Double, DoubleSolution> prodefProblem) {
        double mutationProbability = 1.0 /
        ↪ prodefProblem.getProblem().getNumberOfVariables();
        return new PolynomialMutation(mutationProbability,
        ↪ RunnersConfiguration.DISTRIBUTION_INDEX);
    }

    @Override
    protected SelectionOperator<List<DoubleSolution>, DoubleSolution>
    ↪ getSelectionOperator(ProdefProblem<Double, DoubleSolution> prodefProblem) {
        return new BinaryTournamentSelection<>();
    }
}
```

D.7.6. IntegerGeneticProblemRunner

```
package prodef.singleobjective;

import org.uma.jmetal.operator.crossover.CrossoverOperator;
import org.uma.jmetal.operator.crossover.impl.IntegerSBXCrossover;
import org.uma.jmetal.operator.mutation.MutationOperator;
import org.uma.jmetal.operator.mutation.impl.IntegerPolynomialMutation;
import org.uma.jmetal.operator.selection.SelectionOperator;
import org.uma.jmetal.operator.selection.impl.BinaryTournamentSelection;
import org.uma.jmetal.solution.integersolution.IntegerSolution;
import prodef.AbstractProdefProblemInstance;
import prodef.problem.IntegerProdefProblem;
import prodef.problem.ProdefProblem;
import prodef.runner.RunnersConfiguration;

import java.util.List;

public class IntegerGeneticProblemRunner extends AbstractGeneticProblemRunner<Integer,
↪ IntegerSolution> {
    @Override
    protected ProdefProblem<Integer, IntegerSolution>
    ↪ getProblem(AbstractProdefProblemInstance<Integer> problemInstance) {
        return new IntegerProdefProblem(problemInstance);
    }

    @Override
    protected CrossoverOperator<IntegerSolution>
    ↪ getCrossoverOperator(ProdefProblem<Integer, IntegerSolution> prodefProblem) {
        return new IntegerSBXCrossover(RunnersConfiguration.CROSSOVER_PROBABILITY,
        ↪ RunnersConfiguration.DISTRIBUTION_INDEX);
    }

    @Override
    protected MutationOperator<IntegerSolution>
    ↪ getMutationOperator(ProdefProblem<Integer, IntegerSolution> prodefProblem) {
```

```
        double mutationProbability = 1.0 /
        ↪  prodefProblem.getProblem().getNumberOfVariables();
        return new IntegerPolynomialMutation(mutationProbability,
        ↪  RunnersConfiguration.DISTRIBUTION_INDEX);
    }

    @Override
    protected SelectionOperator<List<IntegerSolution>, IntegerSolution>
    ↪  getSelectionOperator(ProdefProblem<Integer, IntegerSolution> prodefProblem) {
        return new BinaryTournamentSelection<>();
    }
}
```

D.7.7. IntegerPermutationGeneticProblemRunner

```
package prodef.singleobjective;

import org.uma.jmetal.operator.crossover.CrossoverOperator;
import org.uma.jmetal.operator.crossover.impl.PMXCrossover;
import org.uma.jmetal.operator.mutation.MutationOperator;
import org.uma.jmetal.operator.mutation.impl.PermutationSwapMutation;
import org.uma.jmetal.operator.selection.SelectionOperator;
import org.uma.jmetal.operator.selection.impl.BinaryTournamentSelection;
import org.uma.jmetal.solution.permutationsolution.PermutationSolution;
import org.uma.jmetal.util.comparator.RankingAndCrowdingDistanceComparator;
import prodef.AbstractProdefProblemInstance;
import prodef.problem.IntegerPermutationProdefProblem;
import prodef.problem.ProdefProblem;
import prodef.runner.RunnersConfiguration;

import java.util.List;

public class IntegerPermutationGeneticProblemRunner extends
↪  AbstractGeneticProblemRunner<Integer, PermutationSolution<Integer>> {
    @Override
    protected ProdefProblem<Integer, PermutationSolution<Integer>>
    ↪  getProblem(AbstractProdefProblemInstance<Integer> problemInstance) {
        return new IntegerPermutationProdefProblem(problemInstance);
    }

    @Override
    protected CrossoverOperator<PermutationSolution<Integer>>
    ↪  getCrossoverOperator(ProdefProblem<Integer, PermutationSolution<Integer>>
    ↪  prodefProblem) {
        return new PMXCrossover(RunnersConfiguration.CROSSOVER_PROBABILITY);
    }

    @Override
    protected MutationOperator<PermutationSolution<Integer>>
    ↪  getMutationOperator(ProdefProblem<Integer, PermutationSolution<Integer>>
    ↪  prodefProblem) {
        double mutationProbability = 1.0 /
        ↪  prodefProblem.getProblem().getNumberOfVariables();
```

```
        return new PermutationSwapMutation<>(mutationProbability);
    }

    @Override
    protected SelectionOperator<List<PermutationSolution<Integer>>,
    ↪  PermutationSolution<Integer>> getSelectionOperator(ProdefProblem<Integer,
    ↪  PermutationSolution<Integer>> prodefProblem) {
        return new BinaryTournamentSelection<>(new
        ↪  RankingAndCrowdingDistanceComparator<>());
    }
}
```

D.8. Utils

D.8.1. GoalValue

```
package prodef.utils;

import org.json.JSONObject;

public class GoalValue {
    private final int index;
    private double value;

    public GoalValue(int index, double value) {
        this.index = index;
        this.value = value;
    }

    public int getIndex() {
        return index;
    }

    public double getValue() {
        return value;
    }

    public void setValue(double value) {
        this.value = value;
    }

    public JSONObject generateJSON() {
        JSONObject outputJSON = new JSONObject();

        outputJSON.put("index", getIndex());
        outputJSON.put("value", getValue());

        return outputJSON;
    }
}
```

D.8.2. Result

```java
package prodef.utils;

import org.json.JSONArray;
import org.json.JSONObject;
import org.uma.jmetal.solution.Solution;
import org.uma.jmetal.solution.binarysolution.BinarySolution;
import org.uma.jmetal.util.ConstraintHandling;
import prodef.AbstractProdefProblemInstance;

import java.util.ArrayList;
import java.util.BitSet;
import java.util.List;

public class Result<N extends Number, S extends Solution<?>> {
    private final boolean isFeasible;
    private final List<VariableValue> variableValues = new ArrayList<>();
    private final List<GoalValue> goalValues = new ArrayList<>();
    private final AbstractProdefProblemInstance<N> problemInstance;
    private final S solution;

    public Result(AbstractProdefProblemInstance<N> problemInstance, S solution) {
        this.problemInstance = problemInstance;
        this.solution = solution;
        isFeasible = ConstraintHandling.isFeasible(solution);

        if (problemInstance.isBinary()) {
            parseBinaryVariableValues();
        } else {
            parseNumericVariableValues();
        }

        parseGoalValues();
    }

    public boolean isFeasible() {
        return isFeasible;
    }

    public List<VariableValue> getVariableValues() {
        return variableValues;
    }

    public List<GoalValue> getGoalValues() {
        return goalValues;
    }

    public S getSolution() {
        return solution;
    }

    public JSONObject generateJSON() {
        JSONObject outputJSON = new JSONObject();

        outputJSON.put("isFeasible", isFeasible);

```

```
56          JSONArray variableValuesJSON = new JSONArray();
57          for (VariableValue variableValue : variableValues) {
58              variableValuesJSON.put(variableValue.generateJSON());
59          }
60
61          JSONArray goalValuesJSON = new JSONArray();
62          for (GoalValue goalValue : goalValues) {
63              goalValuesJSON.put(goalValue.generateJSON());
64          }
65
66          outputJSON.put("variableValues", variableValuesJSON);
67          outputJSON.put("goalValues", goalValuesJSON);
68
69          return outputJSON;
70      }
71
72      private void parseNumericVariableValues() {
73          int index = 0;
74          for (Variable variable : problemInstance.getVariables()) {
75              Number[][] value = new Number[variable.getRows()][variable.getColumns()];
76
77              for (int i = 0; i < variable.getSize(); ++i) {
78                  VariableIndex variableIndex =
                    ↪  problemInstance.getVariablesIndexMap().get(index);
79                  Number solutionVar = (Number) solution.getVariable(index++);
80                  if (problemInstance.isIntegerPermutation()) {
81                      solutionVar = solutionVar.intValue() + 1;
82                  }
83                  value[variableIndex.getRow()][variableIndex.getColumn()] = solutionVar;
84              }
85
86              variableValues.add(new VariableValue(variable, value));
87          }
88      }
89
90      private void parseBinaryVariableValues() {
91          List<Variable> variables = problemInstance.getVariables();
92          BinarySolution binarySolution = (BinarySolution) solution;
93
94          int bitIndex = 0;
95          for (int index = 0; index < variables.size(); ++index) {
96              Variable variable = variables.get(index);
97              Number[][] value = new Number[variable.getRows()][variable.getColumns()];
98
99              BitSet bitset = binarySolution.getVariable(index);
100             for (int i = 0; i < variable.getSize(); ++i, ++bitIndex) {
101                 VariableIndex variableIndex =
                    ↪  problemInstance.getVariablesIndexMap().get(bitIndex);
102                 value[variableIndex.getRow()][variableIndex.getColumn()] =
                    ↪  bitset.get(i) ? 1 : 0;
103             }
104
105             variableValues.add(new VariableValue(variable, value));
106         }
107     }
108
109     private void parseGoalValues() {
```

```
        double[] objectives = solution.getObjectives();
        for (int i = 0; i < objectives.length; i++) {
            goalValues.add(new GoalValue(i, (problemInstance.isMinimize(i) ? 1.0 :
            ↪  -1.0) * objectives[i]));
        }
    }
}
```

D.8.3. Results

```
package prodef.utils;

import org.json.JSONArray;
import org.json.JSONObject;
import org.uma.jmetal.solution.Solution;
import prodef.AbstractProdefProblemInstance;

import java.util.Comparator;
import java.util.List;
import java.util.SortedSet;
import java.util.TreeSet;

public class Results<N extends Number, S extends Solution<?>> {
    private final AbstractProdefProblemInstance<N> problemInstance;
    private final List<S> population;
    private final long computingTime;
    private final int numberOfResults;
    Comparator<S> comparator;
    private final SortedSet<Result<N, S>> results = new TreeSet<>(new
    ↪  ResultComparator());

    public Results(AbstractProdefProblemInstance<N> problemInstance, List<S>
    ↪  population, long computingTime, Comparator<S> comparator) {
        this.problemInstance = problemInstance;
        this.population = population;
        this.numberOfResults = problemInstance.getMaxNumberOfResults();
        this.computingTime = computingTime;
        this.comparator = comparator;

        calculateResults();
    }

    public SortedSet<Result<N, S>> getResults() {
        return results;
    }

    public long getComputingTime() {
        return computingTime;
    }

    public JSONObject generateJSON() {
        JSONObject outputJSON = new JSONObject();

```

```
        JSONArray resultsJSON = new JSONArray();
        for (Result<N, S> result : results) {
            resultsJSON.put(result.generateJSON());
        }

        outputJSON.put("results", resultsJSON);
        outputJSON.put("computingTime", computingTime);

        return outputJSON;
    }

    private void calculateResults() {
        population.sort(comparator);
        for (int i = 0; i < numberOfResults && i < population.size(); ++i) {
            results.add(new Result<>(problemInstance, population.get(i)));
        }
    }

    private class ResultComparator implements Comparator<Result<N, S>> {
        @Override
        public int compare(Result<N, S> o1, Result<N, S> o2) {
            int result = comparator.compare(o1.getSolution(), o2.getSolution());
            if (result == 0) {
                if (!o1.getSolution().equals(o2.getSolution())) {
                    result = o1.getSolution().hashCode() - o2.getSolution().hashCode();
                }
            }
            return result;
        }
    }
}
```

D.8.4. Variable

```
package prodef.utils;

public class Variable {
    private final int index;
    private final String symbol;
    private final String originalSymbol;
    private final int size;
    private final int dimensions;
    private final int rows;
    private final int columns;
    private final Number lowerBound;
    private final Number upperBound;

    public Variable(int index, String symbol, String originalSymbol, int size, int
    ↪  dimensions, int rows, int columns, Number lowerBound, Number upperBound) {
        this.index = index;
        this.symbol = symbol;
        this.originalSymbol = originalSymbol;
        this.size = size;
```

```
19          this.dimensions = dimensions;
20          this.rows = rows;
21          this.columns = columns;
22          this.lowerBound = lowerBound;
23          this.upperBound = upperBound;
24      }
25
26      public int getIndex() {
27          return index;
28      }
29
30      public String getSymbol() {
31          return symbol;
32      }
33
34      public String getOriginalSymbol() {
35          return originalSymbol;
36      }
37
38      public int getSize() {
39          return size;
40      }
41
42      public int getDimensions() {
43          return dimensions;
44      }
45
46      public int getRows() {
47          return rows;
48      }
49
50      public int getColumns() {
51          return columns;
52      }
53
54      public Number getLowerBound() {
55          return lowerBound;
56      }
57
58      public Number getUpperBound() {
59          return upperBound;
60      }
61
62      public Number getFixedUpperBound() {
63          // Fix bug of jMetal
64          if (upperBound instanceof Integer) {
65              int upperBoundInt = (Integer) upperBound;
66              if (upperBoundInt == Integer.MAX_VALUE) return upperBoundInt - 1;
67          } else if (upperBound instanceof Double) {
68              double upperBoundInt = (Double) upperBound;
69              if (upperBoundInt == Double.MAX_VALUE) return upperBoundInt - 1;
70          }
71
72          return upperBound;
73      }
74
75      public VariableIndex getVariableIndex(int atomicVariableIndex) {
```

```
        int row = atomicVariableIndex / columns;
        int column = atomicVariableIndex % columns;
        return new VariableIndex(row, column);
    }
}
```

D.8.5. VariableIndex

```
package prodef.utils;

public class VariableIndex {
    private final int row;
    private final int column;

    public VariableIndex(int row, int column) {
        this.row = row;
        this.column = column;
    }

    public int getRow() {
        return row;
    }

    public int getColumn() {
        return column;
    }
}
```

D.8.6. VariableValue

```
package prodef.utils;

import org.json.JSONObject;

public class VariableValue extends Variable {
    private Number[][] value;

    public VariableValue(int index, String symbol, String originalSymbol, int size, int
    ↪  dimensions, int rows, int columns, Number lowerBound, Number upperBound,
    ↪  Number[][] value) {
        super(index, symbol, originalSymbol, size, dimensions, rows, columns,
        ↪  lowerBound, upperBound);
        this.value = value;
    }

    public VariableValue(Variable variable, Number[][] value) {
        this(variable.getIndex(), variable.getSymbol(), variable.getOriginalSymbol(),
        ↪  variable.getSize(), variable.getDimensions(), variable.getRows(),
        ↪  variable.getColumns(), variable.getLowerBound(), variable.getUpperBound(),
        ↪  value);
```

```
    }

    public VariableValue(Variable variable) {
        this(variable.getIndex(), variable.getSymbol(), variable.getOriginalSymbol(),
        ↪  variable.getSize(), variable.getDimensions(), variable.getRows(),
        ↪  variable.getColumns(), variable.getLowerBound(), variable.getUpperBound(),
        ↪  new Number[][]{});
    }

    public Number[][] getValue() {
        return value;
    }

    public void setValue(Number[][] value) {
        this.value = value;
    }

    public JSONObject generateJSON() {
        JSONObject outputJSON = new JSONObject();

        outputJSON.put("index", getIndex());
        outputJSON.put("symbol", getOriginalSymbol());

        if (getDimensions() == 0) {
            outputJSON.put("value", getValue()[0][0]);
        } else if (getDimensions() == 1) {
            outputJSON.put("value", getValue()[0]);
        } else {
            outputJSON.put("value", getValue());
        }

        return outputJSON;
    }
}
```

Bibliografía

[1] Ibrahim Assouroko and Peter O. Denno. A metamodel for optimization problems. *NIST*, Jan 2016.

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

[3] Kalyanmoy Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. Wiley, 2001.

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

[5] A. E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer-Verlag Berlin Heidelberg, 2015.

[6] Fred W. Glover and Gary A. Kochenberger, editors. *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*. Springer-Verlag US, 2003.

[7] Bernhard Korte and Jens Vygen. *Combinatorial Optimization. Theory and Algorithms*, volume 21 of *Algorithms and Combinatorics*. Springer-Verlag Berlin Heidelberg, 2000.

[8] José Parejo, Antonio Ruiz-Cortés, Sebastián Lozano, and Pablo Fernandez. Metaheuristic optimization frameworks: A survey and benchmarking. *Soft Comput.*, 16:527–561, 03 2012.