Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Prodef-Algorithm: Interfaz para el modelado de meta-heurísticas

Prodef-Algorithm: Interface for meta-heuristics modelling

Daniel del Castillo de la Rosa

La Laguna, 14 de junio de 2022

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. **Andrés Calimero García Pérez**, con N.I.F. 51.149.384-Y, Graduado en Ingeniería Informática, como cotutor

C E R T I F I C A N

Que la presente memoria titulada:

"*Prodef-Algorithm: Interfaz para el modelado de meta-heurísticas*"

ha sido realizada bajo su dirección por D. **Daniel del Castillo de la Rosa**, con N.I.F. 51.154.908-X.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 14 de junio de 2022

Agradecimientos

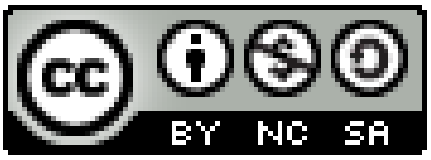
En primer lugar, quiero agradecer a mi tutora, Gara, y a mi cotutor, Andrés, por haberme acompañado, ayudado y apoyado durante el transcurso de este Trabajo de Fin de Grado.

También quiero agradecer a todos esos profesores que, a lo largo de la carrera, me han demostrado que les importa lo que hacen y me han motivado para seguir adelante.

Quiero agradecer a mi familia, por estar siempre ahí y por darme los medios y el apoyo necesarios a la hora de estudiar lo que quiero.

Por último, quiero agradecer a mis amigos, especialmente a los que conocí en la carrera, por hacer que mi paso por el grado no sea únicamente una cuestión académica, sino también una etapa de mi vida que he disfrutado, con sus más y sus menos, y que me ha hecho crecer como persona.

Licencia



Resumen

Si bien en la actualidad existe una gran cantidad de investigación respecto a algoritmos de optimización bioinspirados, estos todavía no gozan de un uso extendido fuera del mundo académico. Su complejidad intrínseca y la dificultad para implementarlos y adaptarlos al problema en cuestión dificultan su uso. Prodef es una herramienta que trata de cambiar eso, permitiendo al usuario definir un problema con una interfaz gráfica basada en bloques desarrollada mediante la librería Blockly. Cada problema está compuesto por cuatro elementos: una serie de parámetros, las variables que conforman una solución, los objetivos y las restricciones que existen para que una solución sea factible. Una vez se ha definido el problema se puede obtener una solución. Para ello se transforma la definición basada en bloques en una basada en ProdefLang, un lenguaje de dominio específico que ayuda en la especificación de objetivos y restricciones. Finalmente, un componente llamado resolutor es capaz de recibir esta definición y computar una solución para el problema. Actualmente, existen dos resolutores cada uno basado en un framework distinto: JMetal y METCO. Sin embargo, en un principio Prodef no permitía al usuario influir en la ejecución más allá de decidir con qué resolutor resolver el problema.

En este trabajo se han implementado los cambios necesarios para que Prodef permita al usuario definir sus propios algoritmos de forma sencilla. Para añadir esta característica ha sido necesario hacer cambios a lo largo de toda la estructura de Prodef. En primer lugar, se ha añadido la capacidad de definir y guardar algoritmos desde la interfaz gráfica. También ha sido necesario crear un nuevo resolutor. Este resolutor es fundamentalmente distinto a aquellos que ya existían, ya que permite especificar qué algoritmo se quiere usar para resolver el problema en cuestión. Asimismo, para implementar este nuevo resolutor, ha sido necesario desarrollar un nuevo compilador de ProdefLang (el lenguaje de dominio específico que usa Prodef) para Rust, ya que este es el lenguaje que se ha elegido para la implementación del nuevo resolutor. Para permitir el modelado de algoritmos, este nuevo resolutor hace uso de una abstracción basada en componentes que se ha diseñado durante este trabajo. Asimismo, se han creado varios componentes para ilustrar el funcionamiento del sistema. Una característica de gran relevancia de este diseño es que los algoritmos que se definen son independientes de cualquier problema. Esto permite aplicar el mismo algoritmo a distintos problemas, si bien es necesario elegir ciertos parámetros y componentes específicos que sí son dependientes del problema.

Estos cambios permiten a Prodef erigirse como una herramienta única. Los usuarios que no necesiten definir sus propios algoritmos podrán usar algoritmos predefinidos e, incluso, comparar distintos algoritmos para encontrar los de mayor rendimiento para su problema. Sin embargo, aquellos usuarios que quieran tener un mayor control sobre la ejecución pueden definir los suyos propios. Además, esto también permite a usuarios que quieran aprender sobre metaheurísticas una plataforma para dar sus primeros pasos. Estas ventajas son de interés para cualquier futuro usuario de la herramienta. Por tanto, este Trabajo de Fin de Grado acerca Prodef a su objetivo de facilitar a empresas e individuos el modelado y la resolución de problemas de optimización combinatoria, sin tener que conocer los detalles de implementación de los mismos.

Palabras clave: Optimización combinatoria, metaheurísticas, algoritmos evolutivos, modelado de algoritmos, Prodef.

Abstract

While there is currently a large amount of research on bio-inspired optimisation algorithms, they are not yet widely used outside academia. Their intrinsic complexity and the difficulty of implementing and adapting them to the problem at hand hinder their use. Prodef is a tool that tries to change that, allowing the user to define a problem with a block-based graphical interface implemented using the Blockly library. Each problem is composed of four elements: a set of parameters, the variables that make up a solution, the objectives and the constraints that exist for a solution to be feasible. Once the problem has been defined, a solution can be obtained. This is done by transforming the block-based definition into one based on ProdefLang, a domain-specific language that helps in the specification of objectives and constraints. Finally, a component called solver is able to receive this definition and compute a solution for the problem. Currently, two solvers exist, each based on a different framework: JMetal and METCO. However, initially Prodef did not allow the user to influence the execution beyond deciding which solver to use to solve the problem.

In this project , the necessary changes have been implemented to allow Prodef users to define their own algorithms in a simple way. To add this feature it has been necessary to make changes throughout the entire structure of Prodef. Firstly, the ability to define and save algorithms from the graphical interface has been added. It has also been necessary to create a new solver. This solver is fundamentally different from those that already exist as it allows the users to specify which algorithms they want to use to solve the problem in question. Likewise, to implement this new solver, it has been necessary to develop a new ProdefLang compiler (the domain-specific language used by Prodef) for Rust, as this is the language that has been chosen for the implementation of the new solver. To enable the modeling of algorithms, this new solver makes use of a component-based abstraction that has been designed during this work. In addition, several components have been created to illustrate the operation of the system. A major feature of this design is that the defined algorithms are independent of the problem. This allows the same algorithm to be applied to different problems, although it is necessary to choose specific parameters and components that depend on the problem.

These changes allow Prodef to establish itself as a unique tool. Users who do not need to define their own algorithms will be able to use predefined algorithms and even compare different algorithms to find the best performing ones for their problem. However, users who want to have more control over the performance can define their own. In addition, this also provides users who want to learn about metaheuristics with a platform to take their first steps. These advantages contribute to solving a number of use cases that could be of interest to a future user of the tool. Therefore, this Final Degree Project brings Prodef closer to its goal of making it easier for companies and individuals to model and solve combinatorial optimisation problems, without having to know the implementation details.

Keywords: Combinatorial optimisation, metaheuristics, evolutionary algorithms, modeling algorithms, Prodef.

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

El diseño y la experimentación con algoritmos evolutivos y otros tipos de meta-heurísticas bio inspiradas es un campo de investigación muy amplio y activo en la actualidad. Se ha demostrado que este tipo de técnicas son capaces, en general, de ofrecer soluciones de alta calidad para problemas de optimización complejos y que difícilmente serían abordables mediante técnicas exactas. En la actualidad, si un usuario sin conocimientos específicos sobre meta-heurísticas y/o programación desea aplicar este tipo de técnicas para resolver un problema concreto (sobre el cual sí que tiene conocimiento y/o experiencia profunda) podrá evitar implementar desde cero este tipo de algoritmos. Esto se debe a que ya existen algunas herramientas que permiten usar ciertas técnicas meta-heurísticas fácilmente para la optimización de problemas.

Estas herramientas suponen una gran ventaja respecto a la implementación directa del algoritmo desde el principio, no solo en términos de reutilización de código, sino también en cuanto a metodología y claridad de conceptos. Algunos ejemplos de estas herramientas son jMetal y METCO. jMetal [6] es un *framework* desarrollado en Java que ofrece una estructura o jerarquía de clases para: definir la codificación de las soluciones (`jmetal-solution`), proporcionar implementaciones de algoritmos de optimización con meta-heurísticas (`jmetal-algorithm`), hacer pruebas con problemas (`jmetal-problem`) o diseñar experimentos y analizar los resultados obtenidos (`jmetal-lab`). Por su parte, METCO [2] es una herramienta basada en *plugins* que permite a los usuarios definir nuevos problemas y/o nuevos algoritmos sobreescribiendo algunos métodos concretos en C++.

En ambos casos, los usuarios pueden beneficiarse de toda la infraestructura disponible en estos frameworks para utilizar múltiples resolutores, basados en las meta-heurísticas más ampliamente consolidadas en la literatura, para ejecutar dichos algoritmos con diferentes parámetros o incluso para experimentar con modelos paralelos o con esquemas que automatizan el proceso de sintonizado de los algoritmos a utilizar. Por tanto, gracias a estas herramientas los usuarios pueden abstraerse de todo lo relativo a la implementación de los algoritmos y del núcleo central que hace posible toda la experimentación. Sin embargo, para utilizar estas herramientas será necesario conocer el lenguaje de programación base (Java o C++ en este ejemplo) así como la estructura interna de la herramienta en cuestión (jMetal o METCO). Sin estos conocimientos básicos,s un usuario no podría implementar todo el código relativo a la especificación del problema de optimización.

1.1. Antecedentes y estado actual

Prodef es una herramienta surgida en este contexto y que nace con el objetivo de extender el uso de estas técnicas meta-heurísticas fuera del ámbito de la investigación convencional, tratando de acercar el uso de estas técnicas a usuarios con poco o inexistente conocimiento sobre metaheurística y/o programación. Para ello, trata de establecer una clara separación entre la definición del problema y la determinación del método de optimización a aplicar y, al mismo tiempo, proporciona una interfaz gráfica para el modelado del problema, lo cual supone un salto cualitativo en lo que respecta a sencillez de uso, dejando atrás complejas implementaciones a nivel de código o configuraciones a bajo nivel.

La versión actual de Prodef es fruto de varios Trabajos de Fin de Grado:

- Prodef: meta-modelado de problemas de optimización combinatoria, 2020. Andrés Calimero García Pérez [9].
- Prodef-GUI: Interfaz gráfica para el modelado de problemas, 2021. Daniel González Expósito [11].
- Prodef: Diseño, implementación y experimentación con nuevos resolutores, 2022. Miguel Angel Ordoñez Morales [15].

Inicialmente, se definió un meta-modelo que permite especificar, en términos abstractos, las características esenciales de un problema de optimización. A partir de este modelo, o especificación genérica de un problema, se ofrece una interfaz para traducción directa a diferentes *frameworks* de optimización (como por ejemplo, jMetal y METCO). En Prodef el modelado del problema se realiza a través de una especificación realizada en lenguaje ProdefLang y recogida a través de un fichero JSON. Este lenguaje de dominio específico se creó con el objetivo de posibilitar la definición de funciones objetivo y restricciones en los modelos de forma sencilla. El lenguaje diseñado es una mezcla entre formulación matemática y lenguaje de programación orientado a objetos. ProdefLang permite expresar cualquier tipo de función objetivo y restricción, manteniendo a su vez la simplicidad, pues el objetivo es que cualquier persona sin conocimientos previos de programación pueda usarlo. En una segunda fase, se dotó a Prodef de una interfaz gráfica para poder definir y gestionar diferentes problemas, así como sus correspondientes instancias. De esta forma, en lugar de trabajar directamente con archivos JSON para la especificación de los problemas en lenguaje ProdefLang, los usuarios pueden utilizar una interfaz basada en bloques. A través de la interfaz basada en bloques es posible modelar aspectos del problema como son:

- Las variables del problema, es decir, las incógnitas para las cuales queremos obtener su valor (solución al problema).
- Las funciones objetivo a maximizar o minimizar.
- Las restricciones que deben cumplir las soluciones factibles.
- Los datos de entrada que determinan generalmente las características de las instancias del problema.

A modo de ejemplo, en la Figura 1.1 se presenta el modelado en la interfaz web de Prodef de un problema de planificación de menús.

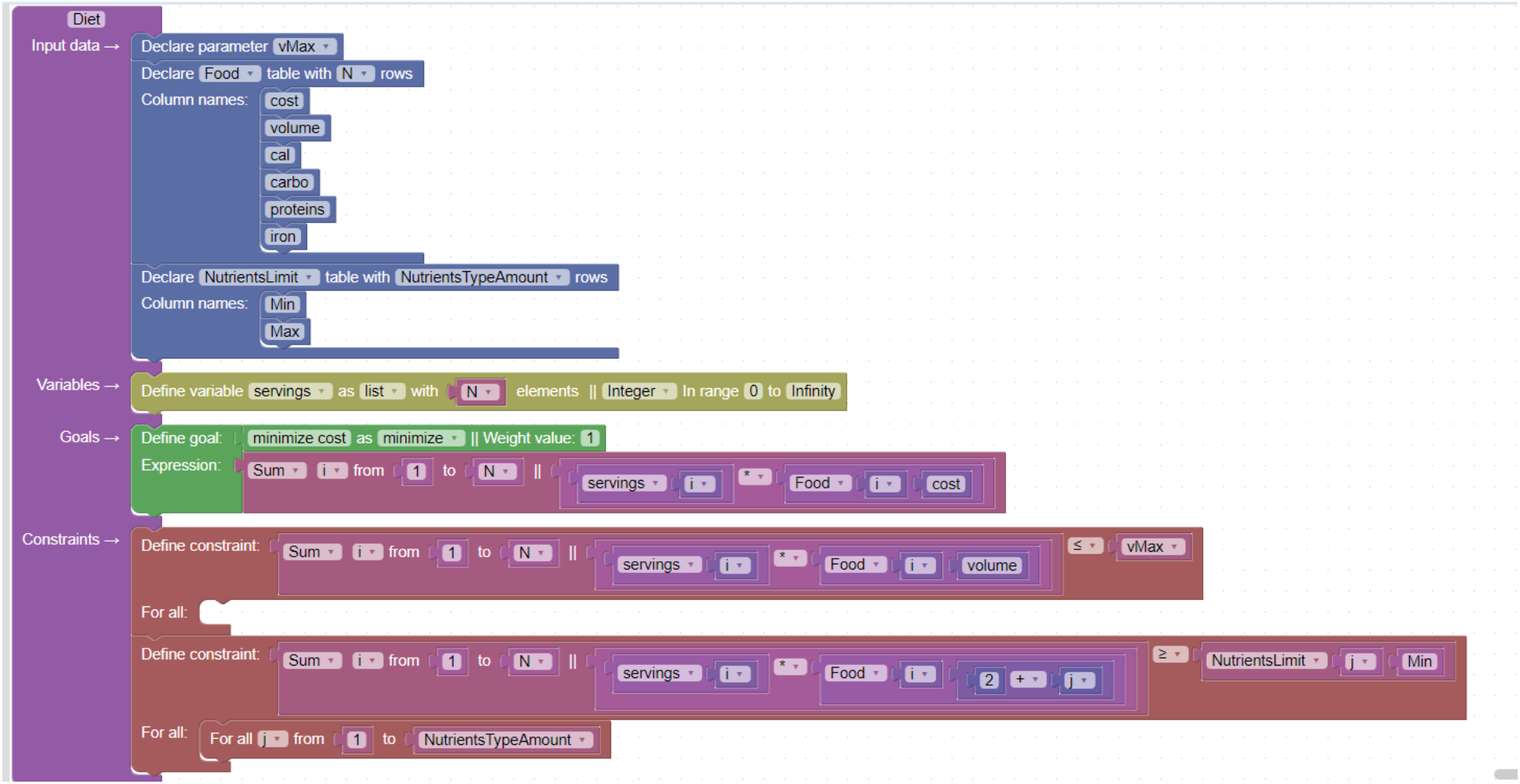


Figura 1.1: Un problema de planificación de menús representado usando Prodef

Por lo tanto, en la actualidad Prodef permite a los usuarios definir un problema de optimización mediante bloques básicos y, a continuación, obtener soluciones para las instancias definidas también a través de la propia interfaz o precargadas mediante archivos CSV. Para la resolución de los problemas, Prodef realizará una traducción a lenguajes de programación como Java o C++ para poder hacer uso directo de herramientas como jMetal o METCO. Sin embargo, la resolución llevada a cabo desde Prodef preestablece, por defecto, un algoritmo evolutivo genérico, sin posibilidad de configurar parámetros como el tamaño de la población o las probabilidades de cruce y/o mutación. Tampoco se ofrecen mecanismos para configurar aspectos concretos de la evolución, como pueden ser los operadores genéticos y los mecanismos de selección y/o reemplazamiento.

1.2. Objetivos

Prodef en su estado actual es un proyecto único y que en el futuro puede ser de utilidad, sin embargo, no puede competir con una implementación experta y a medida de cualquier metaheurística. Esto se debe a que el resolutor de Prodef recibe una información limitada sobre el problema, mientras que una persona que implementa manualmente un algoritmo metaheurístico conoce bien el problema y, por lo tanto, puede decidir correctamente qué operadores genéticos o mecanismos de selección usar. Como consecuencia, el código que el resolutor es capaz de generar es genérico y, a menudo, poco eficaz. El objetivo de este trabajo es extender la funcionalidad de Prodef de forma que también se permita especificar el algoritmo que se quiere aplicar. Los algoritmos se podrán especificar mediante el uso de una serie de componentes predefinidos.

La dificultad radica en diseñar una especificación para estos componentes algorítmicos que sea lo suficientemente flexible para permitir representar la mayor cantidad posible de metaheurísticas, pero que también permita que estos componentes se puedan reutilizar en distintos algoritmos. Todo esto se debe hacer teniendo en cuenta el público para el que este proyecto está dirigido, es decir, personas con poca o ninguna experiencia en la programación y/o en el diseño de algoritmos metaheurísticos.

Teniendo en cuenta esto se han definido los siguientes objetivos:

Tabla 1.1: Descripción de los objetivos

Objetivo	**Descripción**
Análisis de Prodef en su estado actual.	Abordar el estudio de la herramienta Prodef, comprendiendo su estructura modular y su funcionamiento interno.
Análisis de meta-heurísticas e identificación de sus principales componentes.	Analizar las meta-heurísticas más ampliamente utilizadas para comprender su estructura interna y sus principales componentes, detectando similitudes y diferencias entre ellas. Durante este proceso podrá servir de ayuda el estudio de algunos *frameworks* para optimización con meta-heurísticas.
Diseño de un lenguaje o formato específico para el modelado de meta-heurísticas.	Analizar posibles alternativas para la especificación, configuración y/o parametrización de meta-heurísticas. La especificación de las mismas se podría realizar a través de un DSL y/o un fichero JSON de forma análoga a como se hace para el modelado de problemas en Prodef.
Diseño e implementación de traductores de modelos de meta-heurísticas a código ejecutable de las mismas.	Obtener algún mecanismo que, a partir de una especificación abstracta de una determinada meta-heurística, genere un resolutor capaz de simular y ejecutar el proceso de búsqueda especificado por el método en cuestión.
Diseño e implementación de una interfaz gráfica para el modelado de meta-heurísticas.	Diseñar e implementar una interfaz visual a través de la cual se puedan modelar, configurar y/o parametrizar distintos aspectos relativos a las meta-heurísticas o algoritmos de resolución.
Documentación.	Elaborar el material de soporte para el uso de la herramienta, así como la memoria del TFG y demás documentación requerida durante la realización de la asignatura de TFG.

Capítulo 2

Optimización con metaheurísticas

En este capítulo se explorará la base teórica de Prodef, esto es, los principios básicos de la optimización con metaheurísticas, especialmente aquellas que son bioinspiradas. Además, se investigará la posibilidad de dividir metaheurísticas en distintas partes o componentes reutilizables.

2.1. Optimización combinatoria

Prodef trata de resolver problemas de optimización combinatoria. Solucionar un problema de optimización se puede ver como elegir una solución factible de entre el espacio de soluciones. Se desea que esta solución que se elige tenga una calidad razonable respecto a una determinada función de evaluación. Sabiendo esto, un problema de optimización se puede definir haciendo uso de dos elementos:

- **Conjunto de soluciones**. La forma de las soluciones y qué condiciones deben cumplirse para que una solución sea factible. A la hora de describir el conjunto de soluciones mediante código, se debe elegir qué estructuras de datos usar para representar cada solución. La optimización combinatoria se caracteriza por el hecho de que el conjunto de soluciones factibles es un conjunto finito de soluciones discretas.
- **Función de evaluación**. Esta función debe ser capaz de valorar numéricamente cada solución, lo cual permite comparar las soluciones entre ellas.

2.2. Metaheurísticas

A menudo no es factible resolver problemas de optimización de forma exacta. Dependiendo de la complejidad del problema, es posible que se necesite una gran cantidad de tiempo para resolverlo con algoritmos exactos. En estos casos, se tiende a usar métodos no exactos que puedan ofrecer una solución en un tiempo asumible, aunque esa solución no sea la óptima. Entre estos métodos no exactos podemos encontrar las metaheurísticas. Las metaheurísticas no aseguran que la solución obtenida sea la óptima, pero a cambio nos permiten obtener soluciones de cierta calidad en un tiempo de cómputo razonable. Otra ventaja de las metaheurísticas reside en que, a diferencia de los algoritmos exactos que son específicos para cada problema, su estructura se puede reutilizar en varios

problemas, si bien es necesario cambiar algunos aspectos como la representación del problema y la función de evaluación.

2.2.1. Tipos de metaheurísticas

Existen varias formas de catalogar las metaheurísticas. En la Figura 2.1 se pueden observar algunas de las clasificaciones existentes:

- **Búsqueda local o global**. Una búsqueda local explora en cada momento su alrededor y se dirige a aquellas partes del espacio de soluciones más favorables. De esta forma se consigue encontrar soluciones que representan óptimos locales en el espacio de soluciones. Sin embargo, este método tiende a atascarse en óptimos locales, sin llegar a encontrar el óptimo global. Para ello, muchas metaheurísticas como la búsqueda tabú o GRASP añaden otras técnicas para no evitar explorar únicamente óptimos locales.
- **Basadas en trayectoria o en poblaciones**. Las metaheurísticas basadas en trayectoria se centran en una única solución que van mejorando con el tiempo, mientras que las basadas en poblaciones mantienen y mejoran varias soluciones candidatas.
- **Bioinspiradas**. Muchas metaheurísticas están basadas en comportamientos de animales como las hormigas, las aves o las abejas o incluso en los procesos de evolución natural, como es el caso de los algoritmos evolutivos.

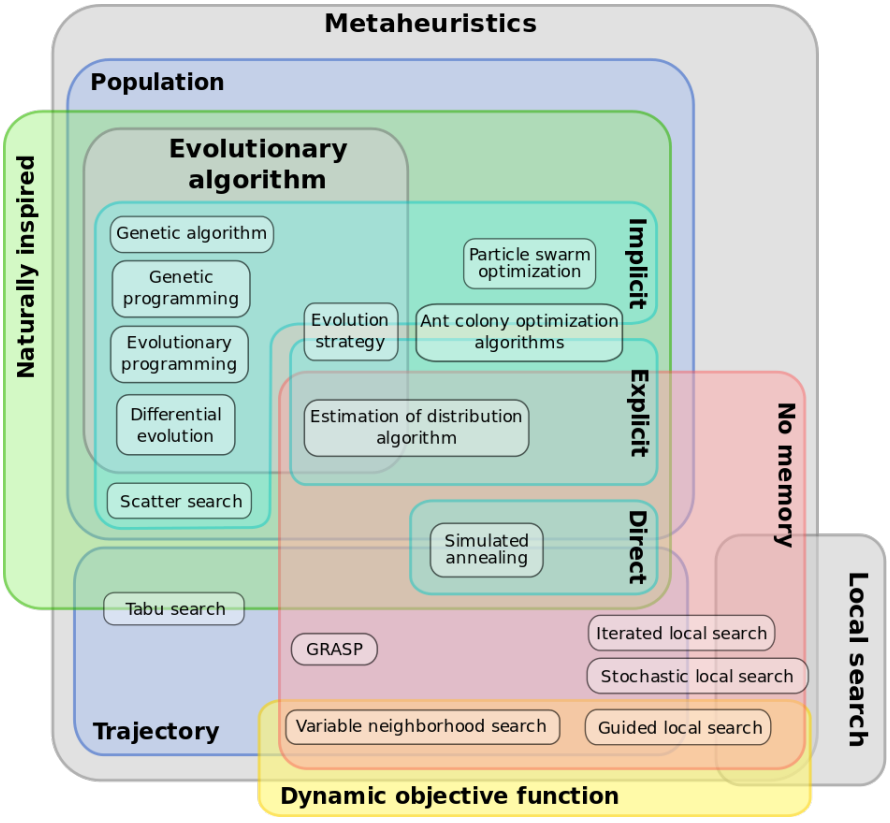


Figura 2.1: Clasificación de metaheurísticas. Fuente: Wikipedia

A continuación estudiaremos varias metaheurísticas y veremos qué necesitamos, además de definir el conjunto de soluciones y la función de evaluación, para implementarlas.

2.2.2. Búsqueda tabú

La búsqueda tabú (*tabu search*) [10] se basa en una búsqueda que empieza en una solución arbitraria factible y luego se va desplazando entre vecinos. Para decidir a qué vecino moverse los evalúa a todos con una función heurística y elige la solución que puntúe mejor, incluso si esa nueva solución es peor que la actual. Con este funcionamiento es fácil ver que esta metaheurística se quedaría atrapada fácilmente en cualquier óptimo local del espacio de soluciones. Para evitar esto se le añade la característica que le da el nombre de búsqueda tabú: una lista de elementos tabú. Cada vez que se explora una nueva solución se coloca la solución, o el cambio que se hizo para llegar a ella, en la lista tabú. Esta solución o cambio no se puede volver a visitar o realizar hasta que deje de ser tabú, lo cual usualmente ocurre después de un número predeterminado de iteraciones. Este método evita que la búsqueda se quede atrapada en una zona del espacio de soluciones. Otra cuestión a destacar es que típicamente también se guarda la mejor solución encontrada hasta el momento, ya que es posible que no se vuelva a ella.

Al analizar esta heurística podemos ver qué es necesario para implementarla:

- Una solución inicial.
- Una definición de vecindad. Se debe saber qué soluciones pueden considerarse vecinas de otras.
- Un valor para la duración del tabú. Es un valor específico para esta metaheurística y que representa el periodo, ya se mida en tiempo o en número de iteraciones, durante el cual está activo el tabú.

2.2.3. GRASP

GRASP (*Greedy Randomized Adaptive Search Procedure*) es una técnica introducida por primera vez en 1989 [7]. La idea principal de esta metaheurística es generar una solución con un algoritmo constructivo randomizado y luego aplicarle una búsqueda local. Este proceso se repite varias veces para realizar búsquedas locales que empiecen en distintos puntos del espacio de soluciones. Por ello, es importante que el algoritmo constructivo que genera las soluciones no sea determinista, ya que es crucial que la solución inicial varíe. La búsqueda local explora el entorno de la solución y, si encuentra una solución con mejor valor para la función objetivo, selecciona esa y repite el proceso hasta encontrar un óptimo local.

Para implementarlo se necesita:

- Un método para generar soluciones aleatorias.
- Una definición de vecindad. Se debe saber qué soluciones pueden considerarse vecinas de otras para aplicar correctamente la búsqueda local.

2.2.4. Recocido simulado

En el recocido simulado (*simulated annealing*) [13] también es necesario empezar desde una solución factible arbitraria. A partir de esa solución se elige un vecino de

forma aleatoria y se decide si se acepta ese vecino como siguiente solución. Esto se hace mediante una probabilidad que se calcula haciendo uso de la siguiente fórmula:

$$p = min(1, e^{-(\Delta E/T)})$$

ΔE representa la diferencia de calidad entre la nueva solución y la anterior. Se puede deducir que, si la nueva solución es mejor, siempre se acepta. La T representa la temperatura, que es un valor que se debe elegir previo a la ejecución. Una temperatura más alta hace que sea más posible que peores soluciones se acepten. Es por esto que se hace que la temperatura decrezca durante la ejecución, haciendo que la búsqueda se vuelva cada vez menos propensa a escoger soluciones que empeoren la actual, hasta que finalmente se decante por un óptimo local. Para implementar esta metaheurística es necesario:

- Una solución inicial.
- Una definición de vecindad. Se debe saber qué soluciones pueden considerarse vecinas de otras.
- El valor inicial de la temperatura. Es un valor específico para esta metaheurística que representa cómo de fácilmente se van a aceptar soluciones peores.
- El proceso de descenso de la temperatura. Típicamente, se definen etapas con distintas velocidades de descenso.

2.2.5. Colonia de hormigas

La metaheurística de la colonia de hormigas (*ant colony*) [4] se inspira en la entomología, concretamente en el método que utilizan las hormigas para encontrar el camino más corto a una fuente de alimento u otro objetivo, mediante el uso de feromonas. Para ello se divide la solución en distintas elecciones que se tienen que tomar y en cada iteración se calculan *m* soluciones. Para calcular cada una de estas soluciones se tienen que tomar decisiones. Al principio de la ejecución, todos los caminos son equiprobables, pero después de cada iteración se les añade una cantidad de feromonas determinada por el valor de la función de evaluación. La probabilidad de cada decisión se calcula de la siguiente manera:

$$P_{rama} = \frac{T_{rama}}{T_{total}}$$

Siendo T una cantidad de feromonas. En concreto, T_{rama} hace referencia a la cantidad de feromonas en una determinada rama de una decisión y T_{total} el total de feromonas en esa decisión. Para evitar que las probabilidades sean iguales a 0 en la primera iteración se elige una cantidad de feromonas inicial. Para implementar esta metaheurística es necesario:

- Un conjunto de decisiones que pueden usarse para generar cualquier solución dentro del conjunto de soluciones.
- El valor inicial de las feromonas. Es un valor específico para esta metaheurística. La velocidad con la que converja el algoritmo dependerá de este valor.
- El número de *hormigas* (soluciones) que se calculan en cada iteración.

2.2.6. Optimización por enjambre de partículas

La optimización por enjambre de partículas (*particle swarm optimization*) [12] se basa en la simulación de un número determinado de partículas que se mueven por el espacio de soluciones de acuerdo a su velocidad. Por lo tanto, es necesario que el espacio de soluciones permita este tipo de movimiento. La velocidad de cada partícula se actualiza en cada iteración. Cada partícula trata de mantener su inercia, pero también se ve atraída hacia la mejor solución encontrada por cualquier partícula y la mejor que haya encontrado esa determinada partícula. Para implementar esta metaheurística es necesario:

- Un espacio de soluciones que permita aplicar una velocidad a una solución para obtener otra.
- Dos constantes que determinan cuánto afecta la atracción al máximo global y al máximo de cada partícula.
- El número de partículas que se usan.
- Una forma de generar una cierta cantidad de soluciones iniciales.

2.3. Frameworks para optimización con metaheurísticas

En la actualidad existe una gran cantidad de *frameworks* de optimización basados en metaheurísticas. Algunos ejemplos pueden ser ECJ [14], JMetal [6], METCO [2], ParadisEO [1] o EasyLocal [3]. Algunos se han especializado en búsquedas locales, mientras que otros en algoritmos evolutivos, dando lugar a una gran variedad de opciones. Estas herramientas ayudan a su usuario, evitando que este tenga que realizar una implementación desde cero y promoviendo la reutilización de código. Sin embargo, siguen existiendo obstáculos que impiden que el uso de este tipo de algoritmos se popularice fuera del ámbito académico.

- No es trivial comparar este tipo de herramientas. Puede ser difícil incluso para un usuario experimentado saber cuál será la técnica más eficiente para resolver un problema. Es por ello que resulta complejo elegir entre distintas herramientas que se especializan en distintos tipos de algoritmos. A menudo, no existe una manera directa de comparar entre dos *frameworks*. Para un usuario inexperto esto supone un gran problema.
- Separar la definición del problema de la forma de resolverlo sigue siendo un desafío. Esto no es casualidad, ya que aunque las metaheurísticas son, en principio, independientes del problema, se sigue necesitando información específica para su desarrollo. Es por estos motivos que es común en los *frameworks* definir junto con el propio problema los operadores de mutación o de *crossover* que se quieren usar y la representación.

Respecto al primer punto, se han realizado varios estudios sobre el estado del ecosistema [5] [16]. En uno de estos estudios [5] se recogió información de más de 50 *frameworks*, pero solo 8 cumplieron con tres criterios básicos:

- Ser código abierto
- Tener actividad en los últimos 5 años (el estudio es de 2021)
- Tener más de 15 contribuidores.

Los *frameworks* que cumplieron estos requisitos aparecen en la Figura 2.2 (los lenguajes de implementación más rápidos se marcan en verde, mientras que el rojo se usa para los más lentos; la columna *kloc* representa los miles de líneas de código).

Name	Language	Update	License	Contributors	kloc	Evol.	EDAs	PSO	Local Search	Cluster	Multicore	GPGPU	Multiobjective	Landscapes	States	Auto. Design
ParadisEO	C++	2021	LGPLv2	33	82	Y	Y	Y	Y	Y	Y	~	Y	Y	Y	Y
jMetal	Java	2021	MIT	29	60	Y	N	Y	N	Y	N	N	Y	N	?	N
ECF	C++	2017	MIT	19	15	Y	N	Y	N	Y	N	N	N	N	Y	N
ECJ	Java	2021	AFLv3	33	54	Y	Y	Y	N	Y	Y	Y	Y	N	Y	N
DEAP	Python	2020	LGPLv3	45	9	Y	N	N	N	Y	Y	N	Y	N	Y	N
CIlib	Scala	2021	Apachev2	17	4	Y	N	N	N	N	N	N	N	N	?	N
HeuristicLab	C#	2021	GPLv3	20	150	Y	N	Y	Y	Y	Y	N	Y	~	Y	N
Clojush	Clojure	2020	EPLv1	17	19	Y	N	N	N	N	N	N	N	N	N	N

Figura 2.2: Principales *frameworks* para computación evolutiva y metaheurísticas. Fuente: [5]

Si bien estudios como estos son útiles a la hora de ayudar a elegir entre distintos *frameworks* queda patente al mirar la tabla que un usuario inexperto no podría hacer uso de esta información. Es difícil saber si se debería resolver un determinado problema usando computación evolutiva u optimización por enjambre de partículas, por ejemplo. Además, tampoco es trivial elegir entre dos *frameworks* que sí implementan el tipo de algoritmo que se necesita, en caso de que se tenga claro.

Respecto al segundo punto, también cabe destacar que (como se ve en la Figura 2.2) se necesita tener conocimientos de algún lenguaje de programación para usar estos *frameworks*.

Prodef trata de solucionar estos problemas, permitiendo al usuario definir sus problemas de forma independiente de la implementación y sin necesidad de usar lenguajes de programación. Si bien esta es una meta compleja, es necesario para que Prodef pueda cumplir su objetivo: acercar este tipo de técnicas a otro público, uno que no necesariamente tiene experiencia programando o conocimientos sobre metaheurísticas.

Durante este trabajo se ha tenido especialmente en cuenta la existencia de HeuristicLab [18], ya que este *framework* posee una potente interfaz gráfica que permite visualizar los problemas y sus soluciones. Sin embargo, se ha decidido que HeuristicLab tampoco compite por el mismo usuario que Prodef, ya que no permite definir problemas desde la interfaz, sino usar aquellos que ya se han definido.

Otra herramienta que se ha investigado es `EC-KitY` [17]. Aunque está centrado en algoritmos evolutivos y actualmente solo soporta programación genética basada en árboles, esta librería destaca por su sencillez de uso.

Capítulo 3

Prodef

En este capítulo se describirán más profundamente los distintos componentes que forman Prodef, así como su funcionamiento y distintas mejoras que se han añadido de manera tangencial durante este Trabajo de Fin de Grado.

3.1. Arquitectura

En el momento en el que este trabajo comenzó, Prodef está compuesto principalmente por nueve componentes, cada uno de ellos alojado en su propio repositorio, que entre ellos suman decenas de miles de líneas de código. La gran mayoría de ellos están implementados en el lenguaje de programación TypeScript, mientras que algunos están implementados en JavaScript o Java. Estos nueve componentes son los siguientes:

- `prodef-gui-frontend`. El front-end web que permite modelar problemas gráficamente.
- `prodef-gui-backend`. El backend que permite guardar en una base de datos no relacional los distintos problemas, instancias y ejecuciones.
- `prodef-common`. Una pequeña librería que define los tipos principales de Prodef.
- `prodef-io`. Contiene distintas herramientas para validar que la entrada y salida de otros componentes de Prodef son correctas.
- `prodef-lang`. La definición del lenguaje de dominio específico de Prodef: ProdefLang.
- `prodef-compiler`. La definición de un compilador de ProdefLang a otro lenguaje. Se ha desarrollado de forma genérica para que se pueda implementar para distintos lenguajes.
- `prodef-compiler-java`. La implementación original de `prodef-compiler`, al lenguaje de programación Java.
- `prodef-solver`. Contiene la definición de un resolutor, así como un servidor que puede ser usado con cualquier implementación de un resolutor.
- `prodef-solver-jmetal`. La implementación original del resolutor de Prodef, apoyándose en la librería JMetal.

Cabe destacar que también existe otro compilador y resolutor alternativos desarrollado por Miguel Ordóñez [15]. En el trabajo original de Andrés García [9] se presentaba la arquitectura de Prodef con la imagen que aparece en la Figura 3.1. Sin embargo, Prodef ha sufrido varios cambios y ahora su arquitectura se puede representar de forma más precisa con esquema de la Figura 3.2. En ambas figuras el símbolo (*) representa que el módulo en cuestión es una interfaz y se necesita una implementación. Por ejemplo, en la versión original, se crearon `prodef-solver-jmetal` y `prodef-compiler-java`.

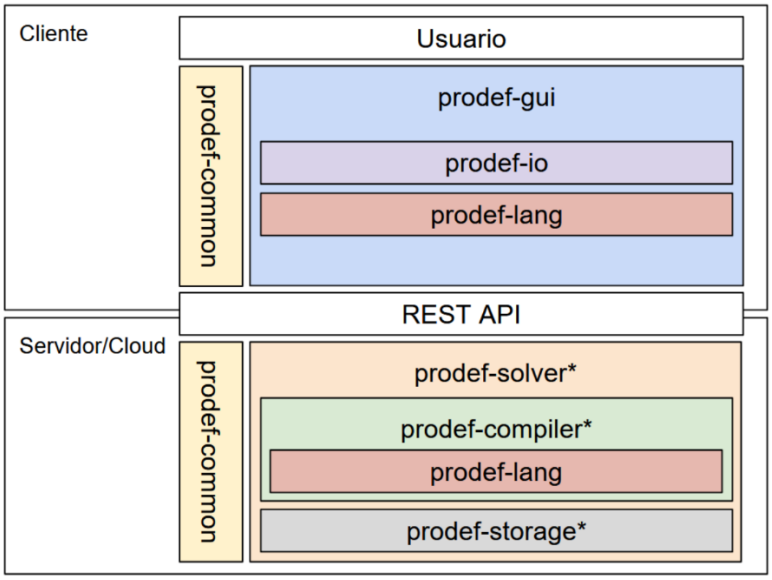


Figura 3.1: Arquitectura original de Prodef

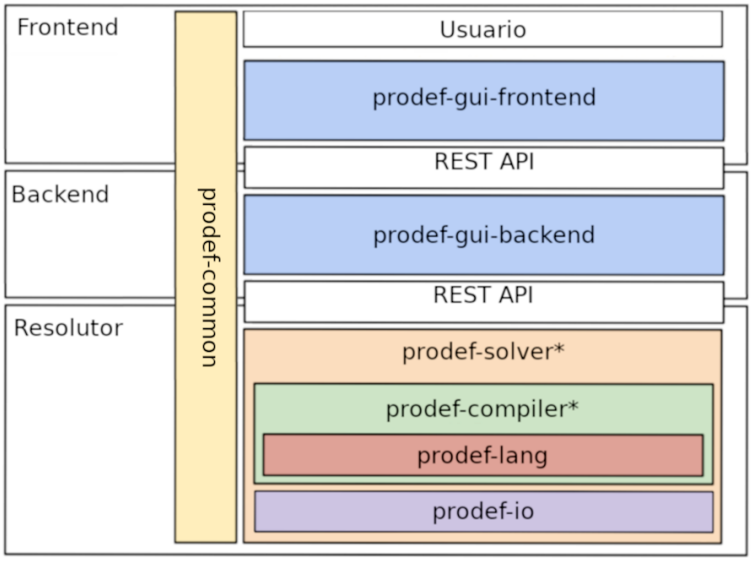


Figura 3.2: Arquitectura actual de Prodef

Aunque puede parecer que han ocurrido una gran cantidad de cambios, en realidad todo se debe a un único motivo. `prodef-storage` solo era capaz de guardar ejecuciones completas con su propio problema, incluyendo los datos necesarios del problema. Sin embargo, al desarrollar el frontend gráfico Daniel González [11] se dio cuenta de que no era posible guardar únicamente cada problema en su totalidad con su ejecución. Por

el contrario, para poder representar problema independientemente de los valores de una instancia concreta, era necesario dividir la definición de un problema de una forma que `prodef-storage` no contemplaba. Para ello surgió, `prodef-gui-backend` que usa una base de datos para suplir esta necesidad y consigue que el resolutor se dedique únicamente a resolver los problemas.

3.2. Funcionamiento

En esta sección se explicará el funcionamiento de Prodef **previo** a la realización de este Trabajo de Fin de Grado. Para ello se seguirá el proceso que un problema debe recorrer para ser resuelto usando Prodef.

3.2.1. Definición del problema

Se debe definir el problema usando una interfaz gráfica basada en Blockly [8]. En la Figura 3.3 se puede ver el problema de la mochila representado mediante esta interfaz. La definición está separada en cuatro secciones:

- Parámetros. Datos constantes que se definen para cada una ejecución.
- Variables. Las distintas variables que conforman una solución. Las variables se pueden agrupar en matrices, listas, listas permutadas o un único valor. Por ejemplo, en el problema del viajante de comercio que apareció en la Figura 5.1 se usa una lista permutada para representar las ciudades. Además, con la excepción de la lista permutada, todos los tipos de variables permiten usar números reales o enteros y añadir restricciones para que estos números estén dentro de un determinado rango.
- Objetivos. Las funciones objetivo del problema. Especifican cómo calcular el valor objetivo a partir de los valores de las variables y los parámetros.
- Restricciones. Definen si una solución es factible o no. Una solución únicamente es factible si cumple todas las restricciones. Por ejemplo, en el problema de la mochila existe la restricción de que la suma de los pesos de los elementos elegidos no puede superar el peso máximo. Esto se puede apreciar fácilmente en la Figura 3.3.

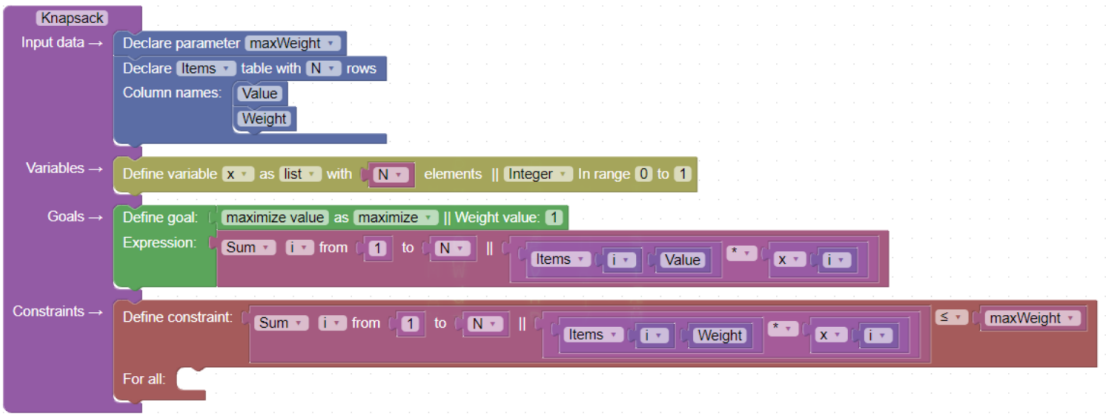


Figura 3.3: Problema de la mochila representado usando Prodef

3.2.2. Definición de una instancia

Después de definir el problema es necesario crear una instancia válida. Esto también se puede hacer desde la interfaz gráfica, ya sea escribiendo manualmente los valores o subiendo un fichero en formato *csv* (*comma separated values*) que los contenga. Un ejemplo de ello se puede ver en la Figura 3.4.

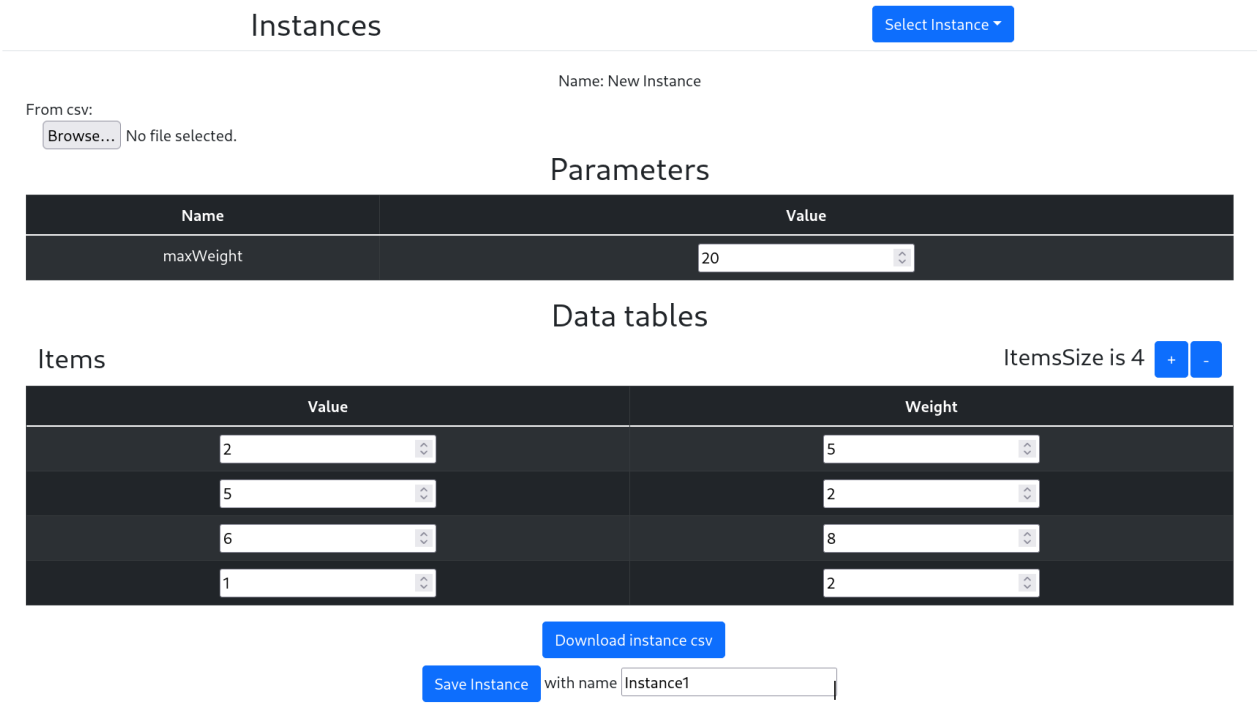


Figura 3.4: Ejemplo de cómo crear una instancia usando Prodef

Cabe destacar que, tanto el problema como la instancia, se pueden guardar por separado (un problema puede tener varias instancias asociadas).

3.2.3. Compilación del problema

Una vez se ha definido un problema y aportado los datos para crear una instancia de ese problema, se puede enviar al resolutor. Aunque el problema se define usando la interfaz gráfica, se guarda y se envía al resolutor en un formato JSON concreto que contiene ciertos campos con expresiones en ProdefLang. En el Apéndice A se puede encontrar un ejemplo del problema de la mochila completo de este formato. Cuando el problema llega al resolutor primero debe compilarse. Esto hace que las expresiones escritas en ProdefLang se transformen en expresiones adecuadas para el lenguaje del resolutor. A continuación se muestra un ejemplo de restricción expresada en ProdefLang. El ejemplo, en concreto, es la restricción del problema de la mochila que ya se ha podido ver en la interfaz gráfica en la Figura 3.3.

```
1 sum x[i]*item[i].weight over i=(1:N) <= MaxWeight
```

En esta pequeña expresión se puede apreciar qué tipo de lenguaje es ProdefLang y qué tipo de operaciones permite. La traducción de este fragmento a Java se puede realizar gracias a `prodef-compiler-java` dando el siguiente resultado:

```
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
```

Destaca que el acceso a los valores de `__item` no es demasiado legible. Esto se debe a que en la interfaz gráfica y ProdefLang se hace uso de clases que luego desaparecen durante la compilación. En la definición del problema de la mochila cada ítem contenía un valor y un peso, sin embargo, durante la compilación estos nombres se convierten en índices, lo que hace el código resultante menos legible. En el Apéndice B se puede encontrar un ejemplo de un problema compilado en su totalidad.

Es importante que para cada restricción no solo se genera una expresión que permite comprobar si se cumple o no la restricción, sino que también se genera una expresión que evalúa el grado de violación de la restricción. Esta expresión permite obtener un valor numérico que representa cómo de lejos se encuentra esa solución de ser factible.

3.2.4. Resolución

Una vez el problema ya ha sido compilado, el resolutor puede usar estas expresiones e introducirlas en una determinada *template*. Esta *template* es la que luego se ejecuta y obtiene el resultado del problema. Cuando ya se ha obtenido el resultado se guarda en el *backend* y se puede mostrar al usuario en el frontend. Los pasos completos para obtener el resultado de un problema se pueden ver en la Figura 3.5.

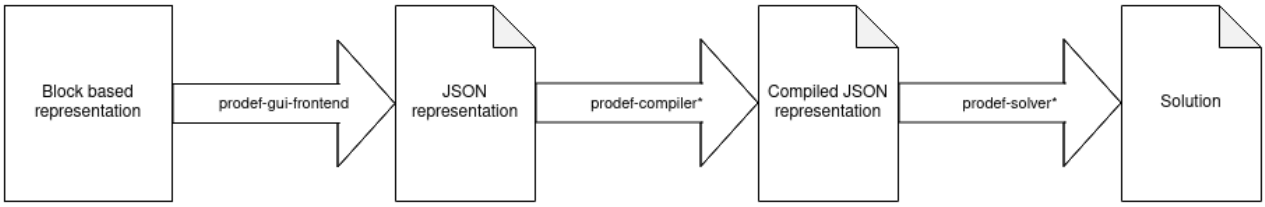


Figura 3.5: Diagrama que muestra el proceso de resolución de un problema

3.3. Tecnologías utilizadas

En los distintos módulos de Prodef se usa una gran variedad de tecnologías. Aquí se hace una pequeña enumeración de las más relevantes:

- *TypeScript*. Es el lenguaje que se usa en la mayor parte de Prodef.
- *Java*. Se usa en el resolutor original, junto con *JMetal*.
- *React*. Es el *framework* que se eligió para desarrollar el frontend. También se hace uso de *React-Redux* y *Redux-Saga*.

- *MongoDB*. La base de datos de *prodeg-gui* usa *MongoDB*. Para interactuar con la base de datos se usa la librería *Mongoose*.
- *Blockly*. La librería que ha permitido desarollar una interfaz gráfica basada en bloques.
- *Handlebars*. Un lenguaje de plantillas que se usan en los resolutores.
- *Koa*. El *framework web* que se emplea para el servidor del resolutor.
- *antlr4ts*. antlr4ts permite a *antlr* generar código *TypeScript* a partir de una gramática. El código generado es capaz de leer el lenguaje definido por la gramática. Esta herramienta se usa en *prodef-compiler* para definir y leer ProdefLang.
- *Github Actions*. La herramienta que se ha usado para conseguir integración continua.

Se ha considerado de utilidad este listado debido a que para poder comprender la magnitud del proyecto es necesario tener en cuenta la cantidad de herramientas que se emplean.

3.4. Mejoras introducidas

Las primeras semanas de este trabajo de fin grado se emplearon para familiarizarse con Prodef, entender cómo funcionaba y qué componentes dependían de qué componentes. Durante este tiempo también se trató de empezar a hacer pequeñas modificaciones al código. Actualizando dependencias y arreglando los problemas que, en ocasiones, venían con las actualizaciones. Cabe destacar en este punto que se ha dedicado una gran cantidad de esfuerzo a entender el código que ya existía para poder modificarlo de forma correcta. A lo largo de estas primeras semanas también se realizaron varias mejoras y correcciones. Dichas modificaciones se relacionan a continuación:

- Evitar que un usuario pueda hacer privados problemas que no son suyos.
- Convertir el frontend de JavaScript a *TypeScript*, consiguiendo una mayor homogeneidad en los lenguajes utilizados en Prodef, además de conseguir errores de tipado, que serían más difíciles de detectar en JavaScript.
- Añadir la opción de borrar problemas. Antes solo se podían añadir problemas, pero no existía la opción de borrarlos.
- No permitir guardar problemas no válidos. Aunque definir los problemas con bloques hace imposible cometer errores sintácticos, sí que era posible definir problemas con errores semánticos. Por ejemplo, problemas que usasen un parámetro no definido. Durante este trabajo se ha implementado una comprobación que impide que problemas semánticamente incorrectos se puedan guardar.
- Refactorizar parte del código.

3.5. Limitaciones de Prodef

Las limitaciones actuales de Prodef están intrínsecamente relacionadas con el objetivo de este Trabajo de Fin de Grado. Tal y como ya se comentó con anterioridad, Prodef utiliza un único algoritmo para resolver todos los problemas con los que se encuentra. Según el teorema No Free Lunch [19], no existe un algoritmo que funcione de manera óptima para cualquier problema, lo que significa que Prodef usando un único algoritmo difícilmente podrá igualar a una metaheurística programada *ad hoc*. Esto se ve claramente en los resultados de la experimentación realizada por Miguel Ordóñez en su Trabajo de Fin de Grado [15].

Capítulo 4

Modelado de algoritmos en Prodef

Para resolver las limitaciones de Prodef en este trabajo se propone añadir la posibilidad de modelar algoritmos. De esta forma, no sería necesario usar siempre el mismo algoritmo para resolver un problema e incluso se podrían comparar fácilmente distintos algoritmos y distintos problemas. En este capítulo se explica el trabajo realizado y las decisiones tomadas durante el proceso.

4.1. Diseño

Para poder comparar distintos algoritmos y problemas, los algoritmos deben ser independientes del problema, es decir, cada algoritmo debe poder aplicarse a distintos problemas. Sin embargo, no es trivial definir un algoritmo de forma que sea independiente del problema que trata de resolver. En este trabajo se optó por un sistema basado en componentes. El algoritmo principal está compuesto de una serie de componentes, que son, por necesidad, independientes del problema. No obstante, cada uno de estos componentes puede definir una serie de dependencias que sí que son dependientes del problema y/o de la ejecución. Estas dependencias pueden ser parámetros numéricos o componentes específicos.

Los componentes específicos sí que dependen directamente del problema que se trata de resolver. Como tampoco es factible definir estos componentes específicos para cualquier problema que el usuario pueda especificar (aunque es una línea que quizás podría investigarse), se ha decidido que estos componentes dependan del tipo de variable del problema. Como ya se comentó en el capítulo anterior, Prodef soporta en la actualidad cuatro tipos de variables distintas: matrices, listas, listas permutadas y valores. La decisión de que estos componentes dependan del tipo de variable trata de alcanzar un punto medio entre crear componentes a medida para cada problema y crear componentes que sean útiles para todo tipo de problemas. Lo primero es relativamente fácil y permite crear componentes eficientes, pero es difícil, si no imposible, abstraer a un usuario de la implementación, ya que sería necesario que el usuario desarrollase estos componentes. Lo segundo sería ideal de cara al usuario, pero es, por desgracia, prácticamente imposible, ya que el mismo operador de mutación no puede funcionar con dos variables distintas, por ejemplo, con una matriz y con una lista permutada.

Respecto a cómo crear los distintos componentes, se decidió que fuesen componentes predefinidos en código de un lenguaje de programación estándar. Si bien se planteó que el propio usuario pudiese definirlos mediante bloques o algún lenguaje especial, se llegó a la conclusión de que poner esta carga en el usuario haría más difícil el uso de la herramienta.

Además, el lenguaje o interfaz que se usase para la programación de los componentes tendría que ser, necesariamente, muy potente, tanto como un lenguaje de programación tradicional si se quiere evitar la pérdida de expresión. Sin embargo, añadir este tipo de lenguaje o interfaz no es trivial y, cómo ya se mencionó, dañaría la intención de abstraer al usuario de parte de la complejidad, permitiéndole operar a un nivel más alto.

Otro inconveniente de esta aproximación es que limita el número de agrupaciones de variables que se pueden usar en un problema. Se permite usar uno de los tipos de variables que existen en Prodef, pero no se permite más de uno. Es decir, una solución se debe representar haciendo uso de una matriz, una lista, una lista permutada o un único valor. Se decidió que, en principio, esta era una desventaja aceptable, ya que, si bien Prodef, por diseño, puede soportar problemas con más de un tipo de variables, ninguno de los resolutores actuales implementa esta capacidad. Por tanto, esta restricción ya existía *de facto*. En principio, incluso con las decisiones de diseño que se han tomado, sería posible usar más de un tipo de variable, pero se deberían diseñar componentes específicos que tratasen con una determinada combinación de variables, lo cual puede probar complejo.

Un último problema a resolver es el manejo de restricciones. Después de evaluar si era más deseable permitir que los algoritmos pudiesen trabajar con soluciones no factibles durante la ejecución o que únicamente pudiesen trabajar con soluciones factibles, se llegó a la conclusión de era mejor permitir ambas según la decisión del usuario y del algoritmo en cuestión. La primera aproximación consistía en una opción que se elegía al definir el algoritmo y que causaba que se permitiese o no el manejo de soluciones no factibles. Sin embargo, esta idea fue evolucionando, hasta llegar a la que se ha empleado finalmente. En la implementación final cada componente cumple uno de los siguientes casos:

- “seguro”. No permiten que en el flujo principal del algoritmo existan soluciones no factibles.
- “no seguro”. Permiten soluciones no factibles.
- Válido para ambos casos. Pueden usarse junto con los dos otros tipos de componente, mientras que los otros dos componentes no se pueden usar juntos en un mismo algoritmo.

Cabe resaltar que existe una diferencia importante entre los componentes seguros y los que siempre son válidos. Los segundos nunca añadirán una solución no factible, los componentes seguros tampoco lo harán, pero además asumen que las soluciones que ya existen son factibles. Es decir, si un componente seguro recibiese como argumento soluciones no factible, no podría reaccionar correctamente, mientras que uno válido para todos los casos sí. Se han dividido de esta forma los componentes para que distintos componentes puedan tratar las restricciones de distinta manera. El tercer tipo de componente, el válido para ambos casos, existe para que ciertos componentes no tengan que definirse dos veces, una como componentes seguros y otra como no seguros. Un ejemplo de esto son componentes como la repetición de un proceso, que operan con las soluciones de manera muy limitada y, por lo tanto, no influyen en si el algoritmo que los usa viola las restricciones o no.

4.2. Cambios en prodef-gui

Después de las primeras ideas de diseño se hicieron algunas pruebas de concepto sobre la creación de bloques. Para ello fue necesario familiarizarse con la librería Blockly. Aunque la documentación de Blockly es deficiente en algunos casos, fue de gran ayuda tener de referencia los ejemplos del trabajo ya desarrollado por Daniel González [11]. Más tarde, se hizo obvia la necesidad de separar la definición de problemas y algoritmos en el frontend, así como su almacenamiento en el *backend*. La separación en el *backend* resultó sencilla después de familiarizarse con el uso MongoDB y Mongoose. Se crearon nuevos esquemas y *endpoints* en la base de datos para algoritmos, de la misma forma que ya existían para los problemas. No obstante, separar la definición en el *frontend* probó ser más complicado de lo que parecía, debido al acoplamiento que existía. El código que permitía ver problemas se repetía en dos componentes de *React*: `ViewProblem` y `NewProblem`. Para poder extraer el código y reutilizarlo se creó un nuevo componente llamada `BlocklyWorkspace` que encapsula el visionado y modificación del problema. `ViewProblem` y `NewProblem` se cambiaron por `ViewElement` y `NewElement` que soportan tanto algoritmos como problemas y hacen uso del nuevo componente `BlocklyWorskpace` para evitar duplicar el código. Una representación de este cambio se puede encontrar en la Figura 4.1. Además, fue necesario conocimiento de React y React-Redux.

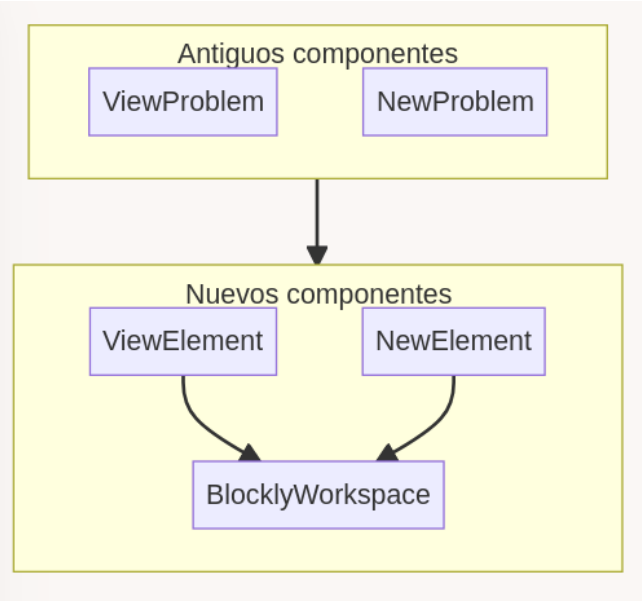


Figura 4.1: Cambios en los componentes de `prodef-gui-frontend` que permitían ver problemas

4.3. Nuevo resolutor

Después de diseñar los primeros esquemas para el modelado de algoritmos, fue evidente que para permitir al usuario de Prodef especificar el algoritmo que quería usar a base de componentes no era recomendable hacer uso de un *framework* existente. En primer lugar, se pudo comprobar que no existen interfaces visuales de alto nivel para el diseño o especificación de algoritmos. Pero además, tampoco fue posible usar *frameworks* existentes para implementar cada uno de los componentes por separado. Hay dos razones de peso para esta afirmación:

- Los *frameworks* suelen definir algoritmos concretos. Para aplicar uno de estos algoritmos concretos es necesario implementar una serie de requisitos en el problema que se quiere resolver. Por ejemplo, para aplicar un determinado algoritmo genético podría ser obligatoria definir un operador de reparación específicos para el problema a resolver. De este modo se consigue que la carga de trabajo para utilizar el *framework* sea mínima. Sin embargo, esto es justo lo contrario de lo que Prodef necesita, ya que Prodef trata de permitir definir el algoritmo con base en pequeños componentes. Por este motivo, tratar de utilizar uno de los *frameworks* existentes para crear estos, idealmente pequeños, componentes parece traer consigo un beneficio mucho menor de lo que se podría esperar inicialmente.
- Como ya se ha comentado anteriormente, los *frameworks* existentes tienden a especializarse en un tipo concreto de metaheurística. Tratar de usar un *framework* podría significar limitar el tipo de componentes que se pueden crear en Prodef.

Además, la principal meta de este Trabajo de Fin de Grado es asentar las bases del modelado de algoritmos de Prodef, de forma que en el futuro se pueda definir la mayor y más variada cantidad de componentes posibles. Esta flexibilidad bien puede valer el coste inicial en eficiencia que puede suponer alejarse de *frameworks* ya trabajados y optimizados.

4.4. Elección del lenguaje para el resolutor

Una de las decisiones más importantes que se tuvieron que tomar durante el proyecto fue el lenguaje de programación en el que se realizaría el nuevo resolutor. Se consideraron varias alternativas:

- Java
- C++
- Haxe
- Rust
- Go

Para elegir se han considerado los siguientes criterios:

- **Adopción**. En este apartado C++ y Java consiguen ventaja, con Go y Rust en segundo lugar y por último Haxe. Este es un punto importante, ya que queremos facilitar la extensión del resolutor.
- **Rendimiento**. Aunque en principio, cualquiera de estos lenguajes podría tener el rendimiento necesario para este proyecto, C++ y Rust tienen un rendimiento mayor al de Haxe, Go y Java, en gran parte debido a los recolectores de basura de estos últimos.
- **Seguridad de memoria**. Java, Haxe y Go ganan en este campo gracias a sus recolectores de basura, con Rust por detrás con su modelo de propiedad y C++ en último lugar.

- **Características modernas**. Ciertas características (*patttern matching*, tipos de datos algebraicos, herramientas integradas para el manejo de dependencias, etc.) pueden hacer que un lenguaje sea más sencillo y más seguro de usar. En este punto destacan Rust y Haxe, mientras que Java, Go y C++ se quedan atrás.
- **Generación de código**. Una ventaja de Haxe es que se puede transpilar a varios lenguajes, como C++, JavaScript o Java. El código resultante, aunque al ser generado automáticamente no es completamente legible, puede llegar a resultar útil para conseguir un punto de partida en varios lenguajes usando Prodef y luego realizar retoques.

Finalmente, se ha decidido usar Rust para el resolutor. Su menor adopción respecto a C++ y Java se compensan con otras de sus características. Haxe fue especialmente considerado por la capacidad de generar código en otros lenguajes, pero tras examinar algunos ejemplos de generación de código se pudo comprobar que partir de ese código para luego refinarlo parece ser una tarea ardua. Dependiendo del lenguaje, el código generado tiene distintos niveles de legibilidad. Además, la potencia del sistema de tipos de Rust y su capacidad de captar errores en el momento de la compilación son de gran ayuda para conseguir una implementación robusta del resolutor.

4.5. Implementación de un nuevo compilador

Prodef está organizado de forma modular, para permitir reutilizar parte de sus componentes si se cambian otros. Para ello, existen varias interfaces, como `prodef-compiler`, que deben ser implementadas. En el momento del inicio de este trabajo existían dos compiladores: `prodef-compiler-java` y `prodef-compiler-cpp`. Para el desarrollo de este trabajo fue necesario crear un nuevo compilador para Rust. Andrés García, durante su Trabajo de Fin de Grado [9], creó una clase llamada `CBasedCompiler` que simplificaba la implementación de compiladores que siguiesen ciertos convenios procedentes de C. Por desgracia, la sintaxis de Rust es sustancialmente diferente a la de C y no se pudo hacer uso de esta simplificación, teniendo, por el contrario, que implementar el compilador completo. Este compilador está alojado ahora en la organización ULL-prodef bajo el nombre de `prodef-compiler-rust`.

4.6. Composición del nuevo resolutor

El nuevo resolutor está dividido en varias partes. La primera y, en cierto sentido, la más importante está escrita en TypeScript. Es la que implementa la interfaz de resolutor que existe en `prodef-solver` y maneja el flujo del programa. Esta parte hace uso de la librería Handlebars para transformar los datos que le llegan en un fichero Rust a ejecutar. La plantilla que se usa se puede encontrar en el Apéndice E.

El resto del nuevo resolutor está escrito en Rust y consiste en dos librerías, que contienen el resto del código Rust que permite que el fichero se pueda ejecutar correctamente. Esta separación es de vital importancia, ya que permite que estas librerías no necesiten recompilarse para cada nuevo problema. La primera de estas librerías, `prodef-componentsdefinition` se encarga de definir los distintos conceptos que se presentarán a continuación. La segunda, `prodef-default-components`, contiene los componentes implementados hasta el momento, de forma que sea trivial usar únicamente las definiciones

y definir componentes propios. También se verán ejemplos de distintos componentes más adelante.

Las dependencias entre estas librerías y el código generado se pueden ver en la Figura 4.2. `prodefdefaultcomponents` depende de los tipos definidos en `prodefcomponents-definition` para poder definir los componentes. El código generado hace uso de ambas librerías.

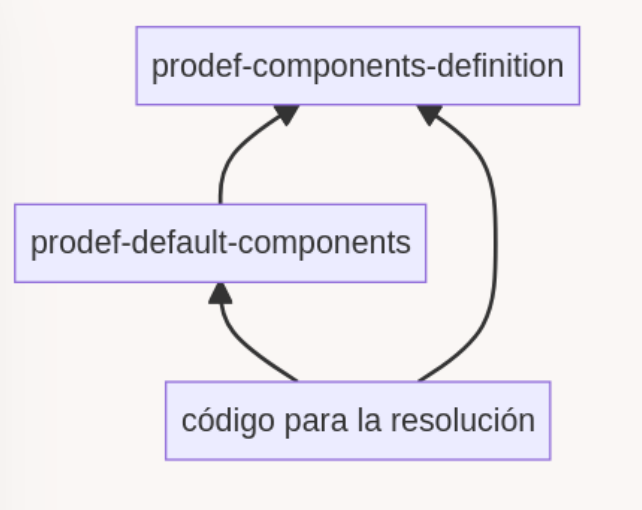


Figura 4.2: Dependencias en el código Rust

4.6.1. Definición de problema

A continuación se puede ver la definición del *trait* (el equivalente a una interfaz en Rust) que un problema debe implementar:

```rust
pub trait Problem<S> {
    fn evaluate_solution(&self, solution: &S) -> f64 {
        self.evaluate_goals(solution).into_iter().sum()
    }

    fn evaluate_goals(&self, solution: &S) -> Vec<f64> {
        (0..self.number_of_goals())
            .map(|i| self.evaluate_goal(i, solution).unwrap())
            .collect()
    }

    fn compare_solutions(&self, a: &S, b: &S) -> Ordering {
        self.evaluate_solution(a)
            .partial_cmp(&self.evaluate_solution(b))
            .unwrap_or(Ordering::Greater)
    }

    fn evaluate_goal(&self, index: usize, solution: &S) -> Option<f64>;
    fn number_of_goals(&self) -> usize;

    fn check_feasibility(&self, solution: &S) -> bool {
        (0..self.number_of_constraints()).all(|i| self.check_constraint(i, solution).
    unwrap())
    }
```

```
24
25      fn check_total_constraint_violation(&self, solution: &S) -> f64 {
26          (0..self.number_of_constraints())
27              .map(|i| self.check_constraint_violation(i, solution).unwrap())
28              .sum()
29      }
30
31      fn check_constraint(&self, _index: usize, _solution: &S) -> Option<bool> {
32          None
33      }
34
35      fn check_constraint_violation(&self, _index: usize, _solution: &S) -> Option<f64> {
36          None
37      }
38
39      fn number_of_constraints(&self) -> usize {
40          0
41      }
42
43      fn get_best_solution(&mut self) -> &S;
44      fn get_best_solution_value(&self) -> f64;
45      fn update_best_solution(&mut self, solution: S);
46  }
```

Como se puede ver, se hace uso de *blanket implementations*, de forma que no sea necesario implementar todos los métodos, pero que sea posible sobreescribirlos si fuese necesario. Las partes de obligatoria implementación se dividen en dos: la definición de los objetivos y la implementación de algún método para guardar la mejor solución hasta el momento. De los objetivos se debe definir cuántas son y la forma de evaluar cada una de ellas. Si el problema tiene restricciones, también se debe definir el número de restricciones, la forma de comprobar si cada restricción se cumple o no y una forma de evaluar el grado de violación de cada restricción. Esta última parte se ha incluido porque ya estaba presente en Prodef. Si el valor resultante es negativo significa que la restricción no se cumple, a menor sea el valor más se aleja el problema de cumplir la restricción. Otra particularidad es que se permite evaluar tanto los objetivos como las restricciones uno a uno, en caso de que un componente necesitase hacerlo. Sin embargo, también se proveen métodos para evaluar un valor global de los objetivos y si la solución es factible, de forma que no siempre sea necesario comprobar cada objetivo o restricción individualmente.

Cabe destacar que el *trait* `Problem` no limita qué puede ser una solución. Siempre que se puedan implementar los métodos necesarios, cualquier cosa podría ser una solución. Sin embargo, el tipo de la solución sí que restringirá el tipo de componentes específicos que se pueden usar. Además, `prodefcomponentsdefinitions` ya define de variables que se usan en Prodef y los componentes que se han desarrollado en este trabajo hacen uso de esas variables. No obstante, haciendo uso de esta definición, se podría definir un problema para una nueva variable y unos componentes hechos por un usuario avanzado.

4.6.2. Definición de componente principal

La definición de un componente principal es sencilla una vez se ha definido un problema.

```rust
pub trait MainComponent<P: Problem<S>, S> {
    fn apply(&mut self, problem: &mut P, solutions: Vec<S>) -> Vec<S>;
}
```

Un componente principal debe implementar un método que permita aplicar el componente a un vector de soluciones. También recibe el problema como argumento, ya que es este el que permite evaluar las distintas soluciones, además de permitir acceder a la mejor solución encontrada y, posiblemente, actualizarla. Cabe destacar que el componente puede modificarse a sí mismo y también a las soluciones. Además, es importante resaltar que, al ser esto un *trait* (equivalente de interfaz en Rust), no limita qué puede guardar un componente, permitiendo, por ejemplo, que guarde el mejor valor presente en esa iteración o cualquier valor que pueda resultar útil al componente. Además, tampoco define el constructor, de forma que cada componente puede definir que parámetros quiere libremente, aunque, es cierto, que para su uso con Prodef estos argumentos deben ser numéricos, componentes específicos o algoritmos.

4.6.3. Definición de algoritmo

La definición de algoritmo también es sencilla, una vez ya se tiene la definición de componente principal.

```rust
pub struct Algorithm<P: Problem<S>, S> {
    components: Vec<Box<dyn MainComponent<P, S>>>,
}
```

Un algoritmo no es un *trait* como las definiciones que ya hemos visto, sino que, al contrario, es una estructura ya definida. En concreto, contiene un vector de componentes principales que se aplican uno tras otro. Ha sido necesario usar *Box*, un puntero inteligente, para aplicar una capa de indirección, ya que los componentes principales que se guardan en ese vector pueden tener distintos tamaños. Lo único necesario es que implementen la interfaz de componente principal.

4.7. Tipos de componentes

Se ha comentado anteriormente que cada componente principal puede tener componentes específicos como dependencias. Estos componentes dependen del tipo de la variable del problema. Sin embargo, además de eso, también están clasificados en cuatro categorías distintas, de forma que un componente principal pueda especificar qué tipo de componente específico necesita.

4.7.1. Generadores

Los generadores son capaces de generar una solución. Es importante destacar que esta solución no tiene por qué ser factible. El *trait* que un generador debe implementar es el siguiente:

```
pub trait Generator {
    type Solution;
    fn generate_solution(&mut self) -> Self::Solution;
}
```

Un generador debe ser capaz de producir soluciones del tipo concreto que se busca. En este caso, en vez de usar un tipo genérico como en las otras ocasiones (la sintaxis <S>) se ha usado un tipo asociado (`type Solution`), lo que expresa que cada generador solo puede implementar un único tipo concreto. Es decir, un generador **nunca** podrá ser genérico sobre el tipo que devuelve. Una vez más, es notable que cada generador puede definir qué datos contiene y qué necesita para inicializarse.

4.7.2. Modificadores

Los modificadores son capaces de recibir una solución y devolver una solución modificada. De nuevo, al igual que ocurre con los generadores, esta solución no tiene por qué ser factible. El *trait* que define el comportamiento de un modificador es el siguiente:

```
pub trait Modifier {
    type Solution;
    fn modify_solution(&mut self, solution: Self::Solution) -> Self::Solution;
}
```

4.7.3. Reproductores

Los reproductores son capaces de recibir dos soluciones y devolver una nueva solución a partir de esas dos. Es el componente principal que haga uso de cada reproductor el que debe elegir qué soluciones mezclar con cuáles. Los reproductores se definen de la siguiente forma:

```
pub trait Breeder {
    type Solution;
    fn breed_solutions(&mut self, first: Self::Solution, second: Self::Solution) -> Self:
    :Solution;
}
```

4.7.4. Vecindades

Las vecindades representan estructuras de entorno; son capaces de recibir una solución y devolver un *array* de soluciones "vecinas". Estos componentes se definen de la siguiente forma:

```
pub trait Neighborhood {
    type Solution;
    fn get_neighborhood(&mut self, solution: Self::Solution) -> Vec<Self::Solution>;
}
```

Hay que destacar que las mismas ideas que permiten definir un modificador pueden servir para definir reproductores o vecindades. En cierto sentido, son distintas formas de expresar la misma relación entre las soluciones. Es decir, si definimos una vecindad, en una lista permutada, como aquellas soluciones que se pueden alcanzar realizando un intercambio dentro de la lista, también podemos definir un modificador que realice un intercambio aleatorio. Este modificador podría dar como nueva solución cualquiera de las que están en la vecindad de la solución inicial.

4.8. Formato JSON

Todos estos componentes tienen una representación JSON. Esta representación se define en el resolutor y es la que se usa fuera del mismo. Es decir, describe a los componentes para que la interfaz gráfica pueda entender qué bloques existen y cuáles son sus características. Esta representación también se usa para informar al resolutor del algoritmo que se quiere usar para resolver un problema.

4.8.1. Algoritmo

```
export interface Algorithm {
  name: String;
  components: MainComponent[];
}
```

Un algoritmo es simplemente, al igual que en su representación en Rust, una lista de componentes principales. En este caso también tiene un nombre.

4.8.2. Componente principal

```
export type SafetyLevel = 'safe' | 'both' | 'unsafe';

export interface MainComponent {
  name: string;
  title: string;
  description?: string;
  safety: SafetyLevel;
  dependencies: (NumberDependency | SpecificComponent | AlgorithmDependency)[];
}
```

Un componente principal tiene varios campos:

- El nombre del componente.
- Un pequeño título. Es una descripción en una frase que explica qué hace.
- Una descripción más elaborada. Es opcional.
- Cómo de "seguro" es usar este componente. Esto hace referencia a si permite o no manejar soluciones no factibles.
- Las distintas dependencias que pueda tener.

4.8.3. Dependencias numéricas

```
export interface NumberDependency {
  type: 'integer' | 'float';
  value: number;
  title: string;
}
```

Las dependencias numéricas son las más sencillas. Su tipo define si el número debe ser entero o de punto flotante y el valor guarda el número en cuestión. Todas las dependencias tienen su tipo y además un pequeño título al igual que cada componente.

4.8.4. Componentes específicos

```
export type VariableShape = 'permutation' | 'vector' | 'matrix' | 'value';

export interface SpecificComponent {
  type: 'modifier' | 'breeder' | 'neighborhood' | 'generator';
  name: string;
  variableShape: VariableShape;
  arguments: NumberDependency[];
  description?: string;
  title: string;
}
```

Los componentes específicos pueden ser de cualquiera de los cuatro tipos que ya hemos mencionado. Además, tienen cada uno su propio nombre, descripción y título. También contienen `variableShape` que hace referencia al tipo de variable con el que pueden ser usados. Por último, los componentes específicos pueden tener sus propios argumentos, que solo pueden ser numéricos.

4.8.5. Dependencias de algoritmos

```
export interface AlgorithmDependency {
  type: 'algorithm';
  algorithm: MainComponent[];
  title: string;
}
```

En ocasiones un componente puede tener como dependencia un algoritmo. Es el caso, por ejemplo, de aquellos componentes que repiten otro proceso varias veces. Este tipo de dependencias contienen una lista de componentes principales. Por ejemplo, un componente que repita un determinado proceso un número de veces utilizaría este tipo de dependencia para recibir el proceso a repetir.

4.9. Generación automática de bloques

Una de las decisiones de diseño más importantes que se han tomado fue la de crear un nuevo *endpoint* en el resolutor para obtener los componentes disponibles. De esta forma, el `frontend` puede preguntarle al resolutor qué componentes están disponibles y usar únicamente esos. Esto permite que para añadir un componente nuevo solo haya que añadirlo al resolutor. Si no existiese este *endpoint* los componentes que se quisiesen usar tendrían que definirse tanto en el resolutor como en el *frontend*. Después de que surgiese esta idea fue necesario comprobar que esto efectivamente se podía realizar, ya que el *frontend* necesitaba ser capaz de representar cualquier componente posible. Después de realizar una prueba de concepto, se comprobó que era posible generar automáticamente los bloques (usando `Blockly`) necesarios a partir del formato JSON que se presentó anteriormente en la sección 4.8. El título de cada componente y dependencia aparece en el bloque de cada componente y la descripción aparece como *tooltip* si dejas el puntero unos segundos encima del bloque en cuestión. Para crear los bloques solo se tienen en cuenta las dependencias de algoritmos, ya que las otras dependencias son parámetros numéricos o componentes específicos, los cuales se eligen en el momento de solicitar una ejecución.

También se intentó que las definiciones JSON de los componentes se crease automáticamente a partir de su implementación en Rust. Si se hubiese conseguido esto, bastaría con implementar cada componente en `prodef-default-components` para que estuviesen disponibles en la interfaz, ya que el resolutor, podría usar el *endpoint* del que se ha hablado, para enviar estas definiciones generadas automáticamente a `prodef-gui`. Sin embargo, esto probó ser complejo. Finalmente, se optó por definir para cada componente su formato JSON en el resolutor. En resumen, para añadir un componente al resolutor es necesario implementarlo en código Rust y luego añadir su definición JSON al resolutor. Solucionar esta cuestión podría convertirse en una nueva línea de trabajo en el futuro.

Para la implementación del *endpoint* también fue necesario modificar `prodef-solver`. El objetivo de esta modificación era hacer que cada resolutor implemente una función que devuelve una lista con los componentes que están implementados. Esto rompe la compatibilidad con los anteriores resolutores, lo que era inevitable, ya que los anteriores resolutores no soportan especificar el algoritmo a usar. Sin embargo, si se quisiese, sería posible adaptar `prodef-gui` para soportar ambas versiones de resolutor. Esto podría ser beneficioso, por ejemplo, para dar más opciones a usuarios que no quieran definir su propio algoritmo, ni elegir uno de los ya definidos.

Capítulo 5

Ejemplos de algoritmos y ejecuciones

En este capítulo se va a afianzar y demostrar cómo funciona realmente todo lo explicado anteriormente. Para ello se va a mostrar cómo se podría resolver el problema del viajante de comercio.

5.1. Definición de problema

El primer paso para resolver un problema es definirlo tal y como se explicó en la sección 3.2.1. Este algoritmo representado mediante Prodef se puede encontrar en la Figura 5.1.

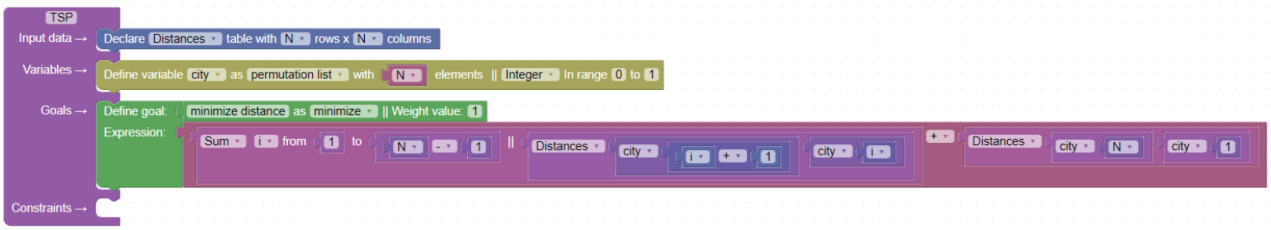


Figura 5.1: El problema del viajante de comercio representado usando Prodef

Se puede observar que se han definido los cuatro componentes de la definición:

- **Parámetros**. En este caso solo existe uno, una matriz de tamaño NxN con las distancias entre ciudades.
- **Variables**. Una lista permutada que contiene la solución. Un cambio en el orden de esta lista permutada supone un cambio en el orden en el que se recorren las ciudades.
- **Objetivos**. Solamente se define un objetivo. Se debe minimizar y es la distancia entre ciudades. Para calcular esta distancia total es necesario usar un sumatorio, así como las variables y los parámetros.
- **Restricciones**. Con la representación actual del problema no es necesario añadir restricciones.

5.2. Definición de algoritmo

Para este ejemplo se ha utilizado un algoritmo GRASP (*Greedy Randomized Adaptive Search Procedure*) sencillo. Este metaheurística ya se estudió previamente en la sección 2.2.3. La definición mediante bloques de una versión de este algoritmo se puede apreciar en la Figura 5.2.

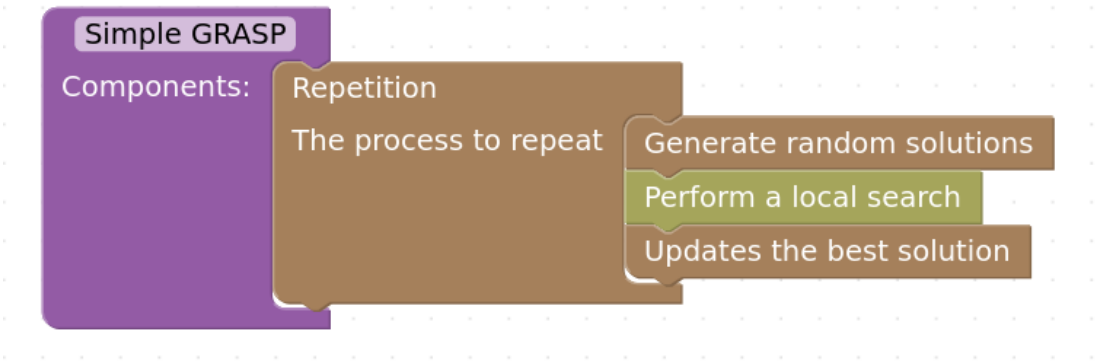


Figura 5.2: Algoritmo GRASP definido mediante Prodef

Puede resultar extraño que no se especifiquen detalles como el número de repeticiones o la forma concreta de generar soluciones. Sin embargo, es necesario recordar que esto es únicamente el esquema del algoritmo. Los parámetros y los componentes específicos se elegirán cuando se vaya a ejecutar el algoritmo con un problema concreto. A continuación, se explorará la definición y la implementación de los componentes que han sido necesarios para definir este algoritmo, con la intención de que sirvan como ejemplo para ilustrar la creación de componentes.

Cabe destacar que las diferencias de colores entre los distintos componentes se deben a los diferentes niveles de seguridad. Los bloques amarillos solo se pueden usar en algoritmos que no permiten violar las restricciones, los rojos únicamente en algoritmos que sí que son capaces de manejar soluciones no factibles y los naranjas se pueden usar en ambos.

5.2.1. Componente de repetición

El primer componente que se ha usado es el componente `FixedRepetition`. Es un componente sencillo que toma como argumento un algoritmo y lo aplica un determinado número de veces. La definición de este componente es la siguiente:

```
{
    "name": "FixedRepetition",
    "title": "Repetition",
    "description": "Repeats a process for a fixed number of times",
    "safety": "both",
    "dependencies": [
        {
            "type": "integer",
            "value": 1,
            "title": "The number of times to repeat the process"
        },
        {
```

```
            "type": "algorithm",
            "algorithm": {
                "components": []
            },
            "title": "The process to repeat"
        }
    ]
}
```

En la definición se pueden apreciar varios detalles que merece la pena destacar. El primero de ellos es que este componente tiene como nivel de seguridad "*both*", lo que significa que se puede usar para construir algoritmos que trabajan o no con soluciones no factibles. Esto resulta lógico, ya que la repetición de un proceso no genera soluciones no factibles si el proceso es incapaz de hacerlo. También se puede observar que el título aparece en el propio bloque, tal y como se ve en la Figura 5.2 y como se comentó anteriormente. También se puede ver que cada dependencia tiene un valor por defecto, 1 en el caso del número de repeticiones y un algoritmo vacío en el caso del proceso a repetir. No ocurre lo mismo en el caso de las dependencias de componentes específicos, ya que no es posible, al menos de la forma en la que están estructuradas las dependencias, elegir uno por defecto.

A continuación se puede encontrar el código en Rust que implementa este componente:

```rust
pub struct FixedRepetition<P: Problem<S>, S> {
    times: i64,
    algorithm: Algorithm<P, S>,
}

impl<P: Problem<S>, S: Clone> MainComponent<P, S> for FixedRepetition<P, S> {
    fn apply(&mut self, problem: &mut P, mut solutions: Vec<S>) -> Vec<S> {
        for _ in 0..self.times {
            solutions = self.algorithm.run_components(problem, solutions);
        }
        solutions
    }
}

impl<P: Problem<S>, S> FixedRepetition<P, S> {
    pub fn new(times: i64, algorithm: Algorithm<P, S>) -> Self {
        Self { times, algorithm }
    }
}
```

Se puede comprobar que, como se explicó anteriormente, cada componente puede contener los datos que se crean necesarios. En este caso, `FixedRepetition` guarda el número de veces que se repetirá el proceso (dependencia numérica) y el proceso a repetir en sí (dependencia de algoritmo).

5.2.2. Generador básico

El segundo componente principal que aparece en el algoritmo que se ha presentado es el generador básico.

```
{
    "name": "BasicGenerator",
    "title": "Generate random solutions",
    "description": "Uses an specific generator to generate random solutions until they
    are feasible",
    "safety": "both",
    "dependencies": [
        {
            "type": "generator",
            "name": "",
            "variableShape": "permutation",
            "arguments": [],
            "title": "The specific generator to use"
        },
        {
            "type": "integer",
            "value": 1,
            "title": "The number of solutions to generate"
        }
    ]
}
```

En su definición se puede ver que `BasicGenerator` permite su uso en ambos tipos de algoritmos, aquellos que manejan soluciones no factibles y aquellos que únicamente manejan soluciones factibles. Esto puede resultar sorpresivo, ya que, según su descripción, genera soluciones hasta que alcanza soluciones factibles. No obstante, hay que recordar que lo que hace que un componente no se pueda usar en entornos con soluciones no factibles no es el hecho de que no las produzca, sino que asuma que todas las soluciones que llegan a él son factibles. En este caso, al ser un generador, no se asume la factibilidad de ninguna solución previa, si bien es verdad que sí se producen soluciones que son únicamente factibles. Cabe destacar que es posible definir generadores que, en vez de generar una nueva solución, apliquen un operador de reparación o que simplemente generen soluciones que puedan ser no factibles (en este último caso el componente no podría usarse en algoritmos que no violan las restricciones). Por otro lado, cabe destacar que, aunque aparezca el campo *variableShape* en la definición y tenga un valor concreto (*permutation*), es solo un ejemplo, ya que la forma real de la variable se decidirá, como ya se ha comentado, al solicitar una ejecución.

```
pub struct BasicGenerator<G: Generator> {
    specific_generator: G,
    number_of_solutions: i64,
}

impl<G: Generator> BasicGenerator<G> {
    pub fn new(specific_generator: G, number_of_solutions: i64) -> Self {
        Self {
            specific_generator,
            number_of_solutions,
        }
    }
}

```

```rust
impl<P, G> MainComponent<P, G::Solution> for BasicGenerator<G>
where
    P: Problem<G::Solution>,
    G: Generator,
{
    fn apply(&mut self, problem: &mut P, _: Vec<G::Solution>) -> Vec<G::Solution> {
        (0..self.number_of_solutions)
            .map(|_| self.generate_until_feasible(problem))
            .collect()
    }
}

impl<G: Generator> BasicGenerator<G> {
    fn generate_until_feasible<P: Problem<G::Solution>>(&mut self, problem: &mut P) -> G::Solution {
        loop {
            let solution = self.specific_generator.generate_solution();
            if problem.check_feasibility(&solution) {
                break solution;
            }
        }
    }
}
```

La implementación de `BasicGenerator` no debería causar sorpresas después de haber visto su definición. Define los mismos argumentos que en su definición JSON y se implementa de una manera simple, con un bucle que genera soluciones hasta que encuentra una solución factible.

5.2.3. Búsqueda local

```json
{
    "name": "LocalSearch",
    "title": "Perform a local search",
    "description": "Performs a local search in the active solution using a provided neighborhood",
    "safety": "safe",
    "dependencies": [
        {
            "type": "neighborhood",
            "name": "",
            "variableShape": "permutation",
            "arguments": [],
            "title": "The specific neighborhood to use"
        }
    ]
}
```

Es posible que alguien se preguntase por qué la búsqueda local sí que restringe su uso a algoritmos que manejan únicamente soluciones factibles. Para entender esto es necesario ver su implementación.

```rust
pub struct LocalSearch<N: Neighborhood> {
    neighborhood: N,
}

impl<P, N> MainComponent<P, N::Solution> for LocalSearch<N>
where
    P: Problem<N::Solution>,
    N: Neighborhood,
    N::Solution: Clone,
{
    fn apply(&mut self, problem: &mut P, solutions: Vec<N::Solution>) -> Vec<N::Solution> {
        solutions
            .into_iter()
            .map(|solution| self.apply_local_search(problem, solution))
            .collect()
    }
}

impl<N: Neighborhood> LocalSearch<N>
where
    N::Solution: Clone,
{
    pub fn new(neighborhood: N) -> Self {
        Self { neighborhood }
    }
    fn apply_local_search<P>(&mut self, problem: &mut P, mut solution: N::Solution) -> N::Solution
    where
        P: Problem<N::Solution>,
    {
        let mut solution_value = problem.evaluate_solution(&solution);
        loop {
            let new_solution = self.apply_local_search_step(problem, solution);
            let new_solution_value = problem.evaluate_solution(&new_solution);
            if new_solution_value == solution_value {
                break new_solution;
            }
            solution = new_solution;
            solution_value = new_solution_value;
        }
    }

    fn apply_local_search_step<P>(&mut self, problem: &mut P, solution: N::Solution) -> N::Solution
    where
        P: Problem<N::Solution>,
    {
        let mut neighbors = self.neighborhood.get_neighborhood(solution.clone());
        neighbors.push(solution);
        neighbors
            .into_iter()
            .filter(|solution| problem.check_feasibility(solution))
            .max_by(|a, b| problem.compare_solutions(a, b))
            .unwrap()
    }
}
```

Si se centra la atención en la función `apply_local_search_step` se puede ver que solo elige la siguiente solución entre las soluciones factibles de la vecindad (línea 50). Para asegurar que siempre se encuentra una solución factible (lo contrario detendría la ejecución debido a la llamada a `unwrap()`), la propia solución se usa como resultado en caso de que ninguna soluciones de la vecindad sea factible (línea 47). Además, tener en cuenta la solución actual también es útil en el caso en que la solución actual es mejor que las demás de su vecindad y ya se ha encontrado un óptimo local. Para que esto funcione correctamente, la solución inicial debe ser factible, por lo tanto, este componente asume la factibilidad de las soluciones que recibe. Esto tiene como consecuencia que este componente solo se pueda usar en algoritmos que no violan las restricciones. No obstante, sí que sería posible definir otra búsqueda local que sí aceptase soluciones no factibles o incluso que las aceptase o no de acuerdo a ciertos parámetros, como ocurre en el *simulated annealing*.

5.2.4. Actualizar la mejor solución

La solución que se devuelve al final de la ejecución se guarda dentro de la clase que representa el problema. Esta se puede actualizar desde cualquier componente. Sin embargo, excepto que haya argumentos de peso para no hacerlo, es mejor actualizarlo desde componentes especializados, de forma que quede constancia en la estructura del algoritmo de los momentos en los que ocurren las actualizaciones. Por ello existe el siguiente componente:

```
{
    "name": "UpdateBestSolution",
    "title": "Updates the best solution",
    "description": "Explores the active solutions and updates the actual best solution if
     needed. Only updates the best solution if there are feasible solutions",
    "safety": "both",
    "dependencies": []
}
```

Lo más destacable de esta definición es que el componente funciona con y sin violación de restricciones. Además, está explicado en la descripción. El componente no asume que las soluciones sean factibles, sin embargo, solo actualiza la mejor solución si una alternativa es factible. Es sencillo apreciar esto en el código (línea 32).

```
pub struct UpdateBestSolution {}

impl<P: Problem<S>, S: Clone> MainComponent<P, S> for UpdateBestSolution {
    fn apply(&mut self, problem: &mut P, solutions: Vec<S>) -> Vec<S> {
        let current_best = self.get_best(problem, &solutions);
        if let Some(solution) = current_best {
            let all_time_best_value = problem.get_best_solution_value();
            let current_best_value = problem.evaluate_solution(&solution);
            if current_best_value > all_time_best_value {
                problem.update_best_solution(solution.clone())
            }
        }
        solutions
```

```rust
    }
}

impl UpdateBestSolution {
    pub fn new() -> Self {
        Self {}
    }

    fn get_best<'a, P: Problem<S>, S>(
        &self,
        problem: &mut P,
        solutions: &'a Vec<S>,
    ) -> Option<&'a S>
    where
        P: Problem<S>,
    {
        solutions
            .iter()
            .filter(|solution| problem.check_feasibility(solution))
            .max_by(|a, b| problem.compare_solutions(a, b))
    }
}
```

5.3. Especificar los parámetros y componentes específicos

Una vez se ha definido un algoritmo, un problema y una instancia de ese problema (se ha podido ver cómo en la Figura 3.4), se puede empezar el proceso de solicitar una ejecución. El primer paso es elegir el algoritmo que se desea emplear para la resolución. Esto se puede apreciar en la Figura 5.3. En la actualidad se pueden elegir tanto algoritmos propios, como algoritmos públicos de otros usuarios.

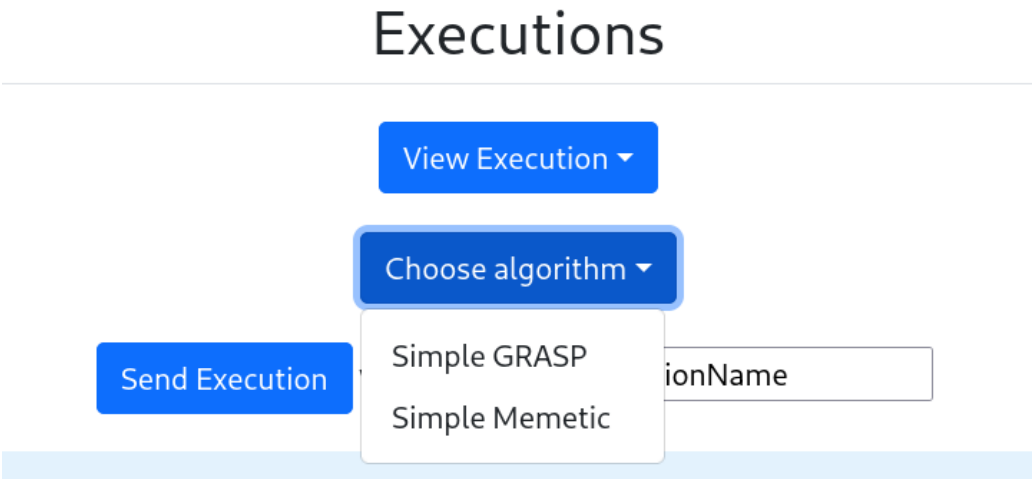


Figura 5.3: Interfaz de Prodef ofreciendo dos algoritmos a elegir

Cuando se elige un algoritmo aparece una nueva sección mostrando una representación de la estructura del algoritmo y que permite especificar los parámetros numéricos y los

componentes específicos para la ejecución. Esta interfaz se genera automáticamente a partir del algoritmo elegido. Se puede observar en la Figura 5.4. Se pueden elegir los componentes específicos que se quiera entre aquellos que soportan la variable del tipo del problema, ya que la interfaz solo muestra los componentes válidos para el problema en cuestión.

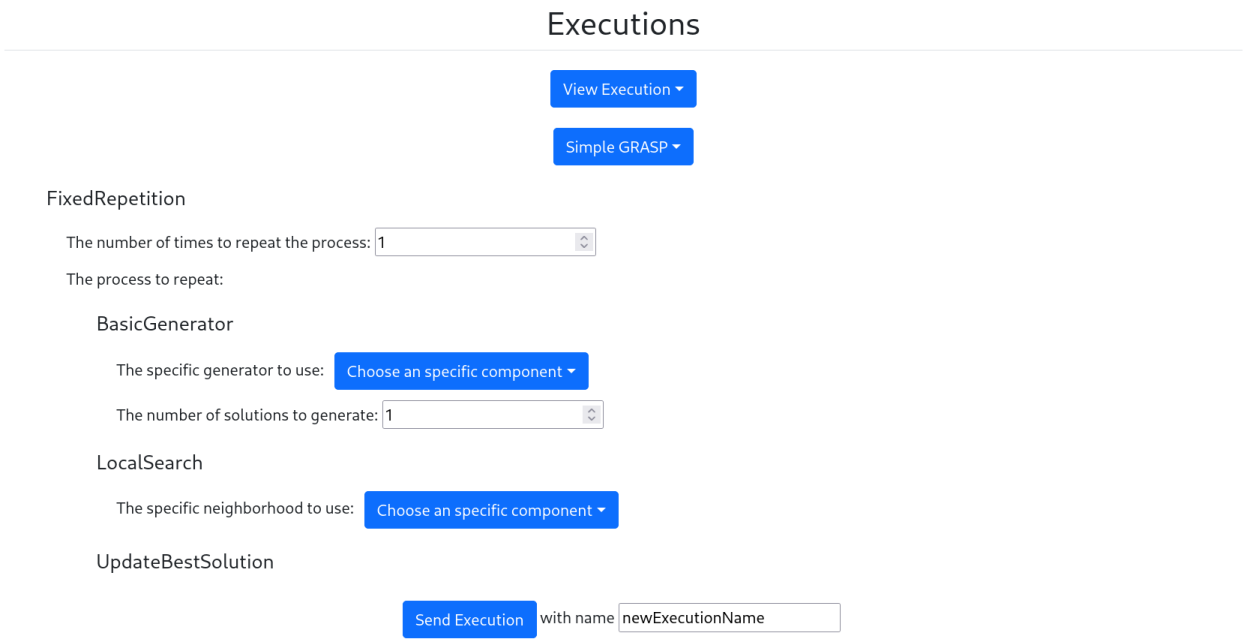


Figura 5.4: Interfaz para la especificación de parámetros numéricos y componentes específicos

En este caso, se usará el algoritmo GRASP que se ha visto anteriormente. Para este algoritmo, tal y como se puede ver en la Figura 5.4 se necesita un generador y una vecindad. En concreto, se elegirá un generador completamente aleatorio y una vecindad basada en *swaps* (intercambios).

5.3.1. Generador específico

A continuación se puede encontrar la definición de un componente específico simple que permite generar permutaciones aleatorias.

```
{
    "type": "generator",
    "name": "SimplePermutationGenerator",
    "variableShape": "permutation",
    "arguments": [],
    "description": "Generates a new permutation randomly",
    "title": "Generates a new permutation"
}
```

Lo más importante es apreciar que, al ser un componente específico, contiene el campo `variableShape`. Este campo restringe en qué tipo de problemas se puede usar este componente. En este caso se puede usar en problemas que usen una lista permutada, como la definición del problema del viajante de comercio que se ha propuesto.

```rust
pub struct SimplePermutationGenerator<const SIZE: i64> {
    rng: ThreadRng,
}

impl<const SIZE: i64> Generator for SimplePermutationGenerator<SIZE> {
    type Solution = Permutation<SIZE>;

    fn generate_solution(&mut self) -> Permutation<SIZE> {
        let mut permutation = Permutation::new();
        permutation.inner.shuffle(&mut self.rng);
        permutation
    }
}

impl<const SIZE: i64> SimplePermutationGenerator<SIZE> {
    pub fn new() -> Self {
        Self { rng: thread_rng() }
    }
}
```

También es sencillo ver, a nivel de código, que este es un componente específico, ya que se define para el tipo `Permutation<SIZE>`, mientras que los componentes principales eran agnósticos respecto el tipo de variable. En este componente específico en concreto, lo único destacable es el uso de `ThreadRng`, que permite realizar un *shuffle* a la permutación inicial. Esta clase no existe directamente en Rust, sino que proviene del paquete (o *crate*, como se llaman en Rust) `rand`.

5.3.2. Vecindad

La vecindad que se usa en este ejemplo está basada en intercambios. Esta vecindad resulta muy fácil de definir para una lista permutada.

```json
{
    "type": "neighborhood",
    "name": "PermutationSwapNeighborhood",
    "variableShape": "permutation",
    "arguments": [],
    "description": "This neighborhood is defined by whether a solution can be reached
    with one swap",
    "title": "Neighbors are solutions that are one swap away"
}
```

No debería haber ninguna sorpresa en la definición. Cabe destacar que se podría implementar una vecindad similar a esta, pero que definiese un algoritmo para el número de intercambios a realizar. De esta manera, un mismo componente específico podría representar varias vecindades basadas en la misma idea, pero de distinto tamaño.

```rust
pub struct PermutationSwapNeighborhood<const SIZE: i64> {}

impl<const SIZE: i64> Neighborhood for PermutationSwapNeighborhood<SIZE> {
    type Solution = Permutation<SIZE>;

    fn get_neighborhood(&mut self, solution: Self::Solution) -> Vec<Self::Solution> {
        (0..SIZE - 1)
            .flat_map(|i| (i + 1..SIZE).map(move |j| (i, j)))
            .map(|swap| {
                let mut inner = solution.inner.clone();
                inner.swap(swap.0, swap.1);
                Self::Solution { inner }
            })
            .collect()
    }
}

impl<const SIZE: i64> PermutationSwapNeighborhood<SIZE> {
    pub fn new() -> Self {
        Self {}
    }
}
```

La implementación es bastante directa. Se calculan todas las soluciones que se puedan obtener realizando un único intercambio.

5.4. Obtener un resultado

Una vez se ha configurado correctamente el algoritmo se puede enviar una solicitud de ejecución. En la solicitud se envía tanto el problema como el algoritmo. Un ejemplo del problema del viajante de comercio definido en JSON usando ProdefLang se puede encontrar en el Apéndice C. Asimismo, la definición JSON completa del algoritmo utilizado se puede encontrar en el Apéndice F. Es importante resaltar que el usuario final no tiene por qué conocer siquiera que esta definición existe, ya que usando la interfaz gráfica nunca debería necesitar interactuar con ella. En el momento en que el *backend* recibe la solicitud, crea una entrada para la nueva ejecución y pasa los datos al resolutor.

Cuando los datos llegan al resolutor, este se encarga de usar el compilador adecuado y compilar el problema. En el caso del ejemplo del problema del viajante de comercio se puede encontrar dicho problema compilado en el Apéndice D. Respecto al algoritmo, se usa la definición que se ha visto anteriormente para generar código Rust que hace uso de esos componentes. El código en cuestión se puede encontrar a continuación:

```
private compileMainComponentList(components: MainComponent[]) {
  const compiledComponents = components
    .map((component) => this.compileMainComponent(component))
    .map((component) => `.add_component(${component})`)
    .join('');
  return 'Algorithm::new()' + compiledComponents;
}

private compileMainComponent(component: MainComponent) {
  const dependencies = component.dependencies
    .map((dependency) => this.compileDependency(dependency))
    .join(', ');
  return `Box::new(${component.name}::new(${dependencies}))`;
}

private compileDependency(
  dependency: NumberDependency | SpecificComponent | AlgorithmDependency
): string {
  switch (dependency.type) {
    case 'algorithm':
      return this.compileMainComponentList(dependency.algorithm);

    case 'float':
      return dependency.value + ' as f64';
    case 'integer':
      return dependency.value + ' as i64';

    default:
      return this.compileSpecificComponent(dependency);
  }
}

private compileSpecificComponent(component: SpecificComponent) {
  const args = component.arguments
    .map((arg) => arg.value + ` as ${arg.type === 'float' ? 'f64' : 'i64'}`)
    .join(', ');
  return `${component.name}::${this.genericArguments}::new(${args})`;
}
```

El código recorre recursivamente todos los componentes creando el código Rust que llama a los componentes especificados. El código Rust resultante es el siguiente:

```
Algorithm::new().add_component(
    Box::new(FixedRepetition::new(
        10 as i64,
        Algorithm::new()
            .add_component(Box::new(BasicGenerator::new(
                SimplePermutationGenerator::<5>::new(),
                5 as i64,
            )))
            .add_component(Box::new(LocalSearch::new(
                PermutationSwapNeighborhood::<5>::new(),
            )))
            .add_component(Box::new(UpdateBestSolution::new()))),
)));
```

De esta forma se construye el algoritmo que luego se usará para resolver el problema. En el Apéndice G se puede encontrar todo el código Rust generado para resolver este problema. Una vez se ha generado el código necesario, este se ejecuta e imprime los resultados en la salida estándar. La salida estándar de la ejecución se lee desde el programa principal y se transforma en una respuesta a la solicitud de ejecución. Cuando el usuario trate de ver el resultado de su ejecución, se mandará una petición al resolutor y, si el problema ya ha sido resuelto, se enviará la respuesta necesaria al *backend*. El *backend*, por su parte, guardará el resultado de la ejecución, además de enviarlo al *frontend*.

5.5. Visualizar el resultado

El resultado de una ejecución se puede ver directamente desde el frontend una vez que el problema ha sido resuelto. Esta interfaz se puede ver en la Figura 5.5. Si bien esta interfaz ya existía anteriormente en Prodef, se ha añadido una nueva sección que permite comprobar el algoritmo con el que se realizó la ejecución y qué parámetros se utilizaron para obtener esos resultados.

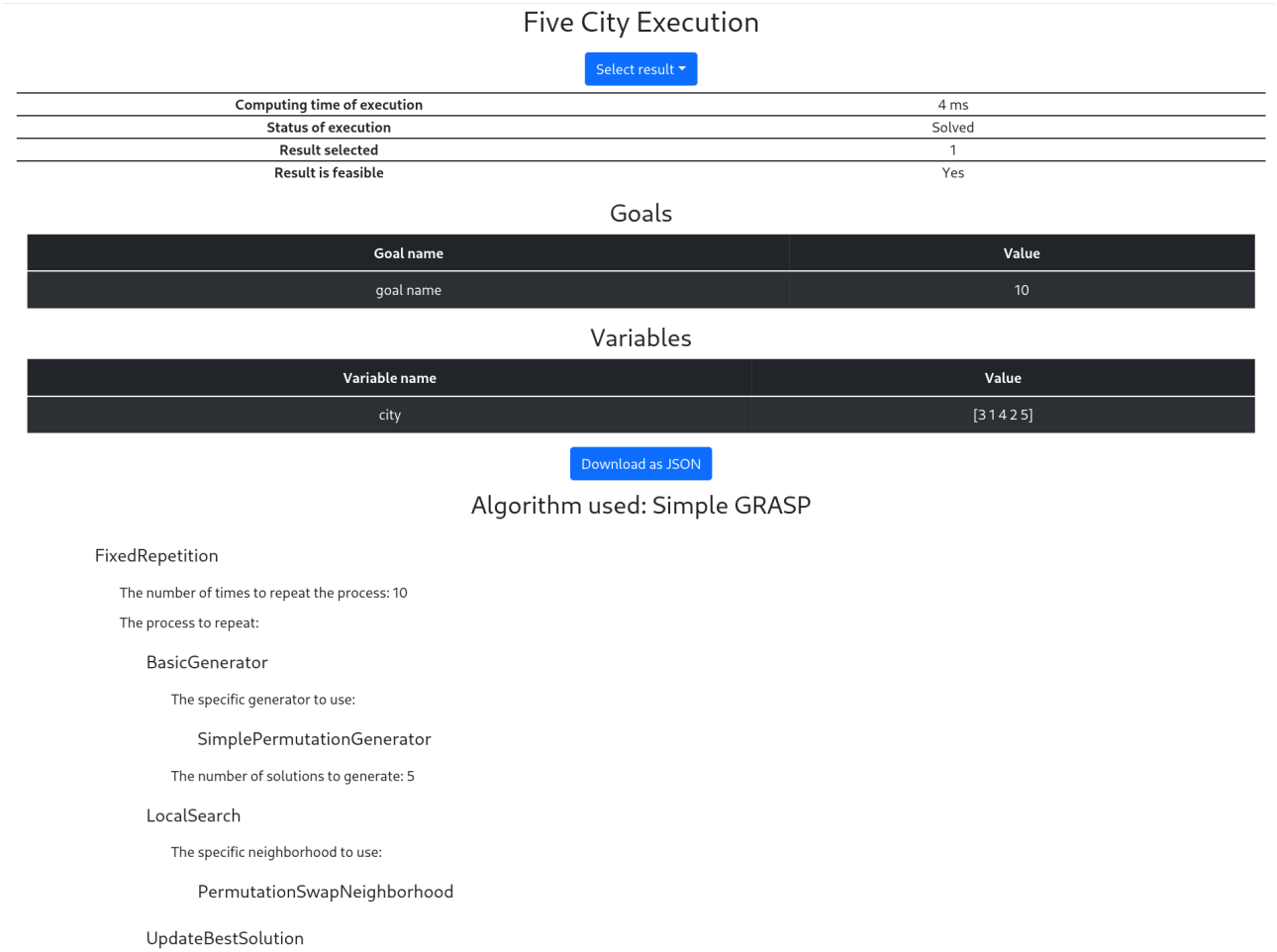


Figura 5.5: Interfaz para la visualización de ejecuciones

5.6. Manejo de restricciones

Hasta el momento se ha usado como ejemplo el problema del viajante de comercio. Este problema tiene la particularidad de que no tiene restricciones con la representación elegida (lista permutada). Sin embargo, una de las mayores dificultades a la hora de desarrollar Prodef ha sido el manejo de las restricciones. Para mostrar un ejemplo de cómo el modelo de algoritmos afronta esta dificultad se propone el problema de la mochila. Este problema ya ha aparecido anteriormente en la Figura 3.3. Para hacer evidentes las diferencias entre algoritmos que permiten violar las restricciones y aquellos que no, se resolverá este problema con dos algoritmos similares. Estos dos algoritmos se pueden ver en las Figuras 5.6 y 5.7.

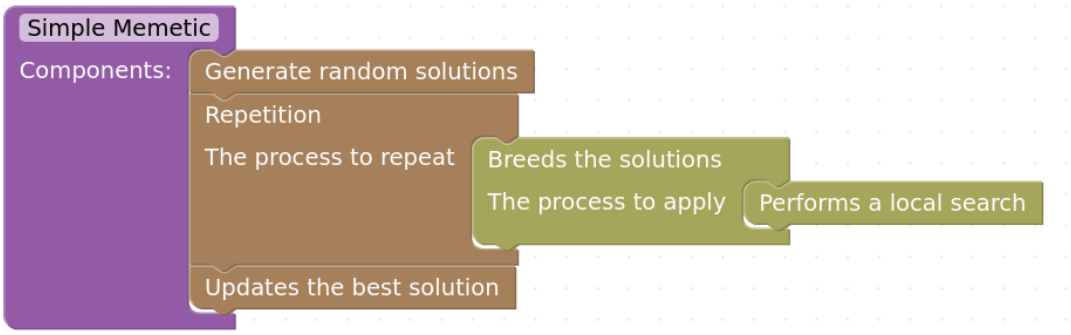


Figura 5.6: Algoritmo memético definido con Prodef

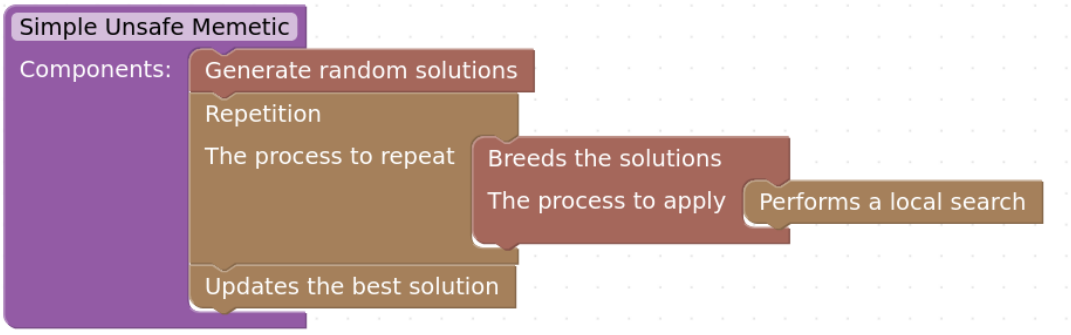


Figura 5.7: Algoritmo memético definido con Prodef con violación de restricciones

Ambos son algoritmos meméticos. Empiezan generando un determinado número de soluciones y luego repiten varias veces el proceso de crear nuevas soluciones a partir de las existentes y aplicarles una búsqueda local. No obstante, algunos de los componentes que usan son diferentes. A continuación, se presentarán los componentes que no se han visto hasta ahora.

- Generador simple que permite violaciones. El generador que se había visto anteriormente generaba soluciones hasta encontrar una factible, este las genera sin tener en cuenta si son o no factibles.
- Reproductor. Este reproductor crea nuevas soluciones a partir de las existentes, filtra las no factibles y les aplica un proceso al resto, para luego juntarlas con las que ya existían y elegir las mejores.

- Reproductor que permite violaciones. Funciona igual que el otro reproductor, excepto que no filtra las soluciones no factibles. Para elegir cuáles son las mejores, prioriza la factibilidad por encima de valor de cada solución.
- Búsqueda local flexible. Esta búsqueda local permite violar las restricciones. Para elegir qué solución es mejor, prioriza la factibilidad por encima del valor objetivo y, además, entre las no factibles elige aquellas que violan en menor medida las restricciones. Cabe destacar que en el algoritmo esta búsqueda se aplica únicamente a las soluciones nuevas que se producen a partir de la reproducción y es por ello que el componente de la búsqueda local aparece en el interior del componente reproductor. Si la búsqueda local se aplicase después de la reproducción, como se hacía en el GRASP, se estaría aplicando a soluciones que ya pasaron por la búsqueda local, ya que en este caos, a diferencia de GRASP, la reproducción también dejar como resultado algunas soluciones que ya existían, ya que elige las mejores.

Estas pequeñas diferencias hacen que ambos algoritmos se comporten de manera distinta. Cabe destacar que dependiendo del problema es posible que cualquiera de ellos sea mejor que el otro. Para poder compararlos también es necesario tener en cuenta los componentes específicos que se usan. En este caso se usan los mismos componentes para ambos algoritmos.

- Generador aleatorio. Se usa un generador que genera un vector con valores aleatorios dentro de un rango. En este caso, el rango es entre 0 y 1.
- Reproductor. El reproductor que se usa mezcla las dos soluciones que recibe de manera aleatoria. Para cada posición se elige el valor correspondiente en uno de las soluciones originales. Cuál de estas soluciones seleccionar se elige de forma aleatoria.
- Vecindad. La vecindad que se usa para la búsqueda local consiste en sumar o restar una determinada cantidad a cualquier variable de la solución. En este caso se ha elegido el valor 1, ya que la solución se representa de forma binaria.

5.6.1. Resultados

En este caso, tratando de resolver la instancia del problema de la mochila que se puede encontrar en la Figura 5.10. Si comparamos los resultados que aparecen en las Figuras 5.8 y 5.9 podemos comprobar que ambos algoritmos han encontrado la solución óptima. Sin embargo, el algoritmo inseguro ha tardado mucho más tiempo. Seguramente esta diferencia de tiempo se deba a que perdió tiempo explorando soluciones no factibles.

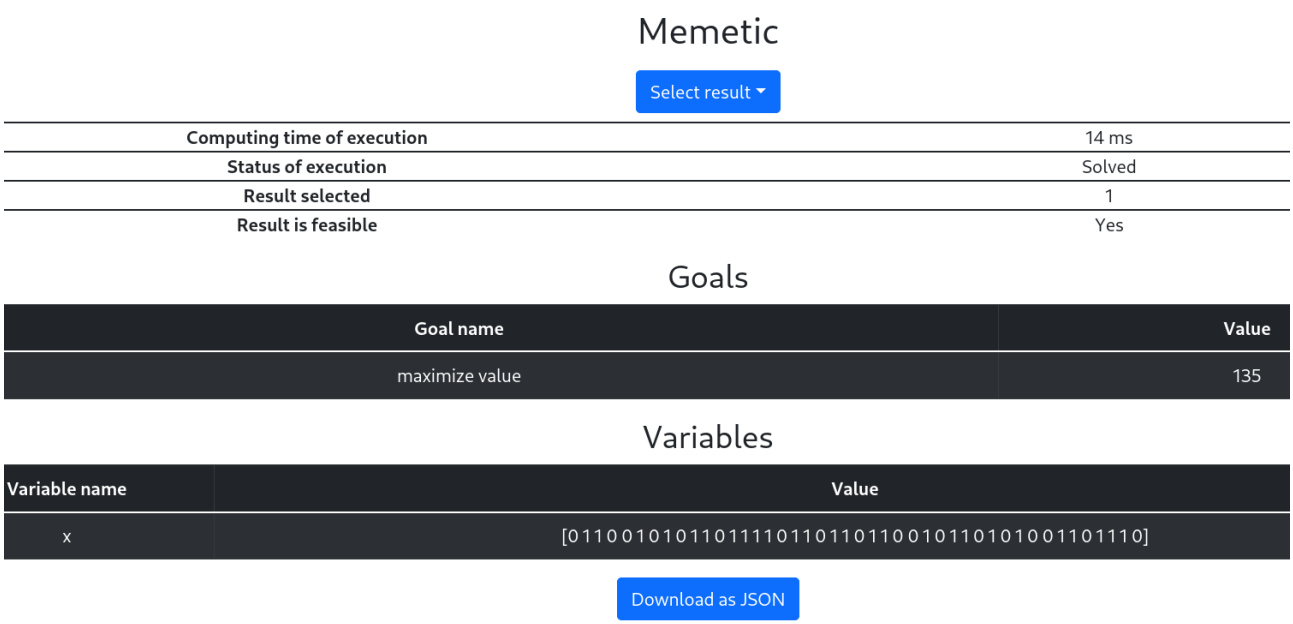


Figura 5.8: Resultado de usar el algoritmo Simple Memetic con la instancia de la mochila visible en la Figura 5.10

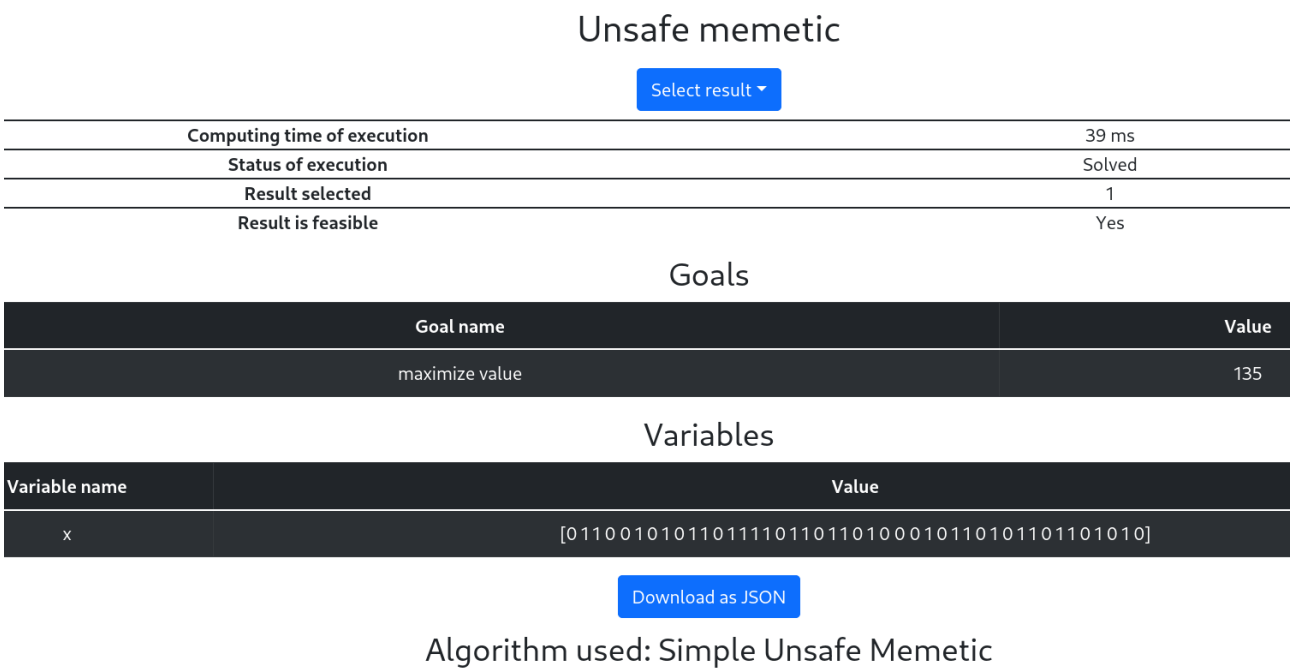


Figura 5.9: Resultado de usar el algoritmo Simple Unsafe Memetic con la instancia de la mochila visible en la Figura 5.10

5.7. Observaciones para la implementación de otros algoritmos

A lo largo de este capítulo se han mostrado las definiciones de un algoritmo GRASP y de un algoritmo memético. En esta sección se pretende mostrar cómo se podrían modelar otros algoritmos, en concreto, aquellos que aparecen en la sección 2.2. Cabe destacar que estos algoritmos se eligieron por sus diferencias entre sí y por contener

Name: 43 element instance

Parameters

Name	Value
maxWeight	84

Data tables

Items

Value	Weight
3	8
4	1
8	9
1	4
4	7
9	2
3	8
7	3
1	4
2	1
5	1
3	8
4	1
8	3
3	2
4	1
3	8
4	3
3	3
1	4
5	4
9	2
2	5
7	7
3	4
2	9
1	7
8	3
2	4
4	1
9	2
3	8
3	2
1	4
2	1
5	8
1	4
5	6
9	9
2	8
7	7
2	4
1	1
1	7

Figura 5.10: Instancia de la mochila con 43 objetos

algunas cuestiones que contribuyen a que sea difícil definirlos fácilmente para cualquier problema. Es importante destacar que poder modelar GRASP implica que se pueden modelar muchos tipos de búsquedas como ILS (*Iterated local search)* y VNS (*Variable Neighborhood Search*). De la misma manera, poder modelar un algoritmo memético implica que se pueden modelar algoritmos evolutivos.

5.7.1. Búsqueda tabú

Para la búsqueda tabú, tal y como se comentó anteriormente, es necesaria una definición de vecindad. Por lo tanto, sería necesario un componente específico de vecindad para el tipo de variable del problema que se quiera resolver. Para la búsqueda en sí se podría diseñar un componente principal que recibiese esta vecindad y un parámetro numérico para especificar la duración del tabú. Este componente podría guardar en cada iteración las soluciones tabú, usándolas después para evitar que la búsqueda se quede atrapada.

5.7.2. Recocido simulado

El recocido simulado también se podría implementar usando Prodef. Es necesaria una definición de vecindad. El valor inicial de la temperatura y la velocidad de descenso se podrían especificar como parámetros del componente en cuestión. En el caso de que se quisiesen varias etapas de temperatura, se podrían hacer un componente con un número fijo de etapas o incluso con un número máximo. En el futuro se podría añadir la posibilidad de parámetros de longitud variable para lidiar con estos casos. Sin embargo, una versión básica del recocido simulado con una única etapa o un número fijo de ellas se podría implementar sin grandes problemas.

5.7.3. Colonia de hormigas

La metaheurística de la colonia de hormigas sí resulta muy complicada de modelar con el diseño actual. Esto no ocurre por casualidad. Este algoritmo hace uso de soluciones incompletas que va completando paso a paso hasta alcanzar la solución final. Lidiar con soluciones incompletas es difícil, si no imposible, con la forma en la que `prodef-compiler` funciona, ya que las expresiones que produce están pensadas para evaluar soluciones completas. Si bien no se descarta que se puedan introducir cambios para lidiar con este tipo de algoritmos en el futuro, en este trabajo no se ha encontrado la forma de hacerlo.

5.7.4. Optimización por enjambre de partículas

Otro algoritmo que en la actualidad no puede representarse es el de la optimización por enjambre de partículas. Para representar este algoritmo sería necesaria una coordinación entre el componente principal y el específico que ahora mismo no es posible. Esto se debe a que cada solución se modifica en un grado distinto dependiendo de su inercia y de la distancia a la que se encuentra de la mejor solución y de la mejor solución encontrada por esa partícula. Al variar este valor entre iteraciones y entre soluciones se vuelve imposible representarlo con el diseño actual. Es posible que en el futuro esto pueda cambiar para diseñar modificadores que puedan tener en cuenta valores dinámicos como estos.

Capítulo 6

Conclusiones y líneas futuras

Durante este trabajo se ha extendido y adaptado Prodef para que permita al usuario definir sus propios algoritmos. Prodef es una herramienta para la resolución de problemas de optimización combinatoria. Esta herramienta permite a un usuario inexperto en el campo de la optimización, o que está tratando de aprender, definir sus propios problemas y obtener una solución. Después del trabajo realizado en este Trabajo de Fin de Grado, también se permite al usuario definir sus propios algoritmos. Para añadir esta característica ha sido necesario hacer cambios a lo largo de toda la estructura de Prodef. Se ha añadido la capacidad de definir y guardar algoritmos desde la interfaz gráfica. También ha sido necesario crear un nuevo resolutor. Este resolutor es fundamentalmente distinto a aquellos que ya existían, ya que permite especificar qué algoritmo se quiere usar para resolver el problema en cuestión. Asimismo, para implementar este nuevo resolutor, ha sido necesario desarrollar un nuevo compilador de ProdefLang (el lenguaje de dominio específico que usa Prodef) para Rust, ya que este es el lenguaje que se ha elegido para la implementación del nuevo resolutor. Para permitir el modelado de algoritmos, este nuevo resolutor hace uso de una abstracción basada en componentes que se ha diseñado durante este trabajo. Asimismo, se han creado varios componentes para ilustrar el funcionamiento del sistema. Una característica de gran relevancia de este diseño es que los algoritmos que se definen son independientes de cualquier problema. Esto permite aplicar el mismo algoritmo a distintos problemas, si bien es necesario elegir ciertos parámetros y componentes específicos que sí son dependientes del problema.

6.1. Conclusiones

Como conclusión, es destacable que los cambios realizados permiten a Prodef erigirse como una herramienta única. Los usuarios que no necesiten definir sus propios algoritmos podrán usar algoritmos predefinidos e, incluso, comparar distintos algoritmos para encontrar los de mayor rendimiento. Sin embargo, aquellos usuarios que quieran tener un mayor control sobre la ejecución pueden definir los suyos propios. Además, esto también permite a usuarios que quieran aprender sobre metaheurísticas una plataforma para dar sus primeros pasos. Estas ventajas contribuyen a resolver una serie de casos de uso que podrían ser de interés para un futuro usuario de la herramienta.

Por tanto, este Trabajo de Fin de Grado acerca Prodef a su objetivo de facilitar a empresas e individuos el modelado y la resolución de problemas de optimización combinatoria, sin tener que conocer los detalles de implementación de los mismos. Un punto a destacar, es que, aunque pueda parecer que permitir al usuario definir su propio algoritmo elimina

la abstracción sobre la resolución que ofrecía Prodef anteriormente, se cumplen dos condiciones:

- Definir un algoritmo propio es opcional. Un usuario inexperto puede usar Prodef sin verse obligado a definir su propio algoritmo.
- La definición de algoritmo se realiza con un cierto nivel de abstracción. Definir un algoritmo usando Prodef continúa siendo más sencillo que desarrollar un algoritmo *ad hoc*.

6.2. Líneas de trabajo futuro

Durante el trabajo se han ido descubriendo diversos problemas y se han experimentado distintas dificultades. La mayor dificultad encontrada ha sido, sin duda, trabajar en un proyecto ya existente, con sus propios defectos y fortalezas. Para completar este trabajo ha sido necesario entender una gran cantidad de código repartido en distintos repositorios y desarrollados en distintos lenguajes de programación, con distintas herramientas, e, incluso, por distintas personas. Sin embargo, también se han encontrado otras grandes dificultades en el diseño. No ha sido trivial encontrar una abstracción útil para desarrollar metaheurísticas a un nivel más alto del habitual. La abstracción que se presenta en este trabajo tiene sus propias limitaciones e inconvenientes, pero es capaz de ofrecer algo nuevo y tiene un gran potencial.

Además, en algunos puntos durante el desarrollo se ha hecho evidente que existen partes de Prodef que requieren más trabajo. En esta sección se enumeran estas líneas de trabajo, describiendo cada una en profundidad.

El área de Prodef que más dista de estar listo para su uso profesional es la experiencia de usuario. Aunque la interfaz de usuario es funcional en su estado actual, dista mucho de tener una calidad satisfactoria. En este sentido, hay varias cuestiones que se podrían mejorar:

- Ver los problemas y algoritmos existentes. En la actualidad, los problemas y los algoritmos simplemente aparecen en la página principal de la interfaz web. De esta forma, al entrar a la web puedes ver tus algoritmos y problemas junto con los de otras personas que sean públicos. Por ahora esto funciona, no obstante, si en el futuro la herramienta empieza a usarse y se crean numerosos problemas y algoritmos, la interfaz dejará de ser funcional, ya que será muy difícil encontrar el algoritmo/problema que se busca. Además, no hay ninguna herramienta que permita descubrir fácilmente otros elementos.
- Una mejor forma de especificar los componentes del algoritmo. En este caso ocurre lo mismo que en el anterior. En el caso de que llegue a haber una gran cantidad de componentes específicos, la interfaz actual dejará de ser funcional. Lo ideal sería que los componentes específicos se puedan añadir también con una interfaz basada en bloques, viendo el algoritmo igual que se definió.
- Separar la especificación del algoritmo de la ejecución de forma más marcada. Si se separase se podría conseguir guardar la especificación de forma separada y permitir ejecutar más fácilmente el mismo algoritmo con distintos parámetros.

- Permitir comparar el resultado de varias ejecuciones desde la interfaz. Aunque ahora es posible ver el resultado de cada ejecución, no es cómodo comparar los resultados.

A continuación se pueden encontrar otras posibles líneas de trabajo futuro que no están relacionadas directamente con la experiencia de usuario:

- Componentes. Durante este trabajo se ha diseñado un sistema de componentes y se ha modificado Prodef para soportarlo. Sin embargo, actualmente no existe una gran variedad de componentes. Por lo tanto, se necesita dedicar trabajo a añadir un mayor número de ellos.

- Permitir un mayor grado de configuración de los componentes. Con la implementación actual, un componente puede definir tres tipos de dependencias: algoritmos, componentes específicos y parámetros numéricos. Sin embargo, debería ser posible añadir argumentos que sean específicos del propio componente de forma que se puedan configurar aspectos del propio componente. Por ejemplo, una búsqueda local podría ser ansiosa o no. En la actualidad para crear esta diferencia es necesario crear dos componentes distintos, pero esto no resulta ideal ni para el usuario ni para la persona que implementa los componentes.

- Dar más información al *frontend*. Aunque se han hecho mejoras en este aspecto, como no permitir guardar problemas no válidos semánticamente, todavía se puede mejorar más en este aspecto. Esto no solo incluye información para poder rechazar problemas o algoritmos no válidos, sino también para poder ofrecer mejores errores al usuario e incluso clasificar mejor los problemas, algoritmos o componentes.

- Añadir nuevos tipos de variables. En la actualidad, Prodef maneja matrices, listas, listas permutadas y valores. Es posible que sea útil añadir tipos de variables adicionales. Un ejemplo podrían ser representaciones binarias. Si bien ya es posible representarlas mediante vectores de enteros cuyos valores están restringidos a 1 o a 0, tener un tipo de variable propio permitiría representar las soluciones de manera especial en el código generado para una mayor eficiencia y crear componentes específicos de mayor calidad, que posean más información sobre la representación.

- Cambiar la implementación de las vecindades. La implementación actual define que las vecindades devuelven un vector con los vecinos de la solución. Sin embargo, se podrían conseguir mejoras de rendimiento en algunos, por ejemplo, en búsquedas locales ansiosas, si no se devolviese un vector, sino un iterador. De forma que no se calculen obligatoriamente todas las soluciones vecinas si el componente que quiere hacer uso de la vecindad no lo necesita.

- Facilitar el uso de la herramienta mediante documentación. Sería muy útil para un usuario tener documentación de cada componente y guías sobre cómo usar Prodef. Se podrían guardar datos de uso de cada componente ofreciendo estadísticas al usuario sobre qué componentes son los más usados e, incluso, recomendarle componentes específicos de acuerdo con el algoritmo y problema que quiere resolver.

- Facilitar la creación de nuevos componentes. Una idea para esto es generar la definición JSON de cada componente automáticamente a partir del código, como se comentó en la sección 4.9

Capítulo 7

Summary and Conclusions

During this project, Prodef has been extended and adapted to allow users to define their own algorithms. Prodef is a tool for solving combinatorial optimisation problems. This tool allows users who are inexperienced in the field of optimisation, or who are trying to learn, to define their own problems and obtain a solution. After the work done in this Final Degree Project, Prodef also allows users to define their own algorithms. To add this feature, it has been necessary to make changes throughout the entire structure of Prodef. Firstly, the ability to define and save algorithms from the graphical interface was added. It has also been necessary to create a new solver. This solver is fundamentally different from those that already exist, as it allows users to specify which algorithms they want to use to solve the problem in question. Likewise, to implement this new solver, it has been necessary to develop a new ProdefLang compiler (the domain-specific language used by Prodef) for Rust, as this is the language that has been chosen for the implementation of the new solver. To enable the modeling of algorithms, this new solver makes use of a component-based abstraction that has been designed during this work. In addition, several components have been created to illustrate the operation of the system. A major feature of this design is that the defined algorithms are independent of the problem. This allows the same algorithm to be applied to different problems, although it is necessary to choose specific parameters and components that depend on the problem.

7.1. Conclusions

In conclusion, it is noteworthy that the changes made during this project, allow Prodef to establish itself as a unique tool. Users who do not need to define their own algorithms will be able to use predefined algorithms and even compare different algorithms to find the best performing ones. However, users who want to have more control over performance can define their own. In addition, this also provides users who want to learn about metaheuristics with a platform to take their first steps. These advantages contribute to solving a number of use cases that could be of interest to a future user of the tool. Therefore, this Final Degree Project brings Prodef closer to its objective of making it easier for companies and individuals to model and solve combinatorial optimisation problems, without having to know the implementation details. One point to note is that, although it may seem that allowing the user to define his or her own algorithm eliminates the abstraction on resolution that Prodef previously offered, two conditions are met:

- Defining algorithms is optional. An inexperienced user can use Prodef without being

forced to define his own algorithm.

- The algorithm definition benefits from a certain level of abstraction. Defining an algorithm using Prodef is still simpler than developing an *ad hoc* algorithm.

7.2. Lines for future work

During the course of the project, a number of problems have been discovered, and several difficulties have been experienced. The greatest difficulty encountered has undoubtedly been working on an existing project, with its own flaws and strengths. To complete this work it has been necessary to understand a large amount of code distributed in different repositories and developed in different programming languages, with different tools, and even by different people. However, other major design challenges have also been faced. It has not been trivial to find a useful abstraction to develop metaheuristics at a higher level than usual. The abstraction presented in this report has its own limitations and drawbacks, but is capable of offering something new and has great potential.

Additionally, at some points during development, it has become apparent that there are parts of Prodef that require more work. This section lists these lines of work, describing each in depth.

The area of Prodef that is furthest from being ready for professional use is the user experience. Although the user interface is functional in its current state, it is far from being of satisfactory quality. In this respect, there are a number of issues that could be improved:

- View existing problems and algorithms. Currently, problems and algorithms are simply displayed on the main page of the web interface. This way, when you log in, you can see your algorithms and problems along with those of other people that are public. For now this works, however, if in the future the tool starts to be used and numerous problems and algorithms are created, the interface will no longer be functional, as it will be very difficult to find the algorithm/problem being searched for. In addition, there is no tool that allows easy discovery of other elements.

- A better way to specify the components of the algorithm. If there are too many components, the current interface will no longer be functional. Ideally, specific components can also be added with a block-based interface, viewing the algorithm as it was defined.

- Separate the algorithm specification from execution more noticeably. Separating this would allow the specification to be stored separately and allow the same algorithm to be run more easily with different parameters.

- Allow the results of different runs to be compared from the interface. Although now it is possible to see the result of each run, it is not convenient to compare the results.

Other possible lines of future work that are not directly related to user experience can be found in the following:

- Components. During this work, a component system has been designed and Prodef has been modified to support it. However, there is currently not a large variety of

components. Therefore, work needs to be dedicated to adding a larger number of them.

- Allow a greater degree of component configuration. With the current implementation, a component can define three types of dependencies: algorithms, specific components and numerical parameters. However, it should be possible to add arguments that are specific to the component itself so that aspects of the component itself can be configured. For example, a local search could be anxious or not. Currently, to create this difference it is necessary to create two separate components, but this is not ideal for neither the user nor the person implementing the components.

- Give more information to the frontend. Although improvements have been made in this area, such as not allowing semantically invalid problems to be stored, there is still room for improvement in this area. This not only includes information to be able to reject invalid problems or algorithms, but also to be able to provide better errors to the user and even better classify problems, algorithms or components.

- Adding new types of variables. Prodef currently handles matrix, lists, permuted lists and values. It may be useful to add additional variable types. An example might be binary representations. While it is already possible to represent these using integer vectors whose values are restricted to 1 or 0, having your own variable type would allow you to represent solutions in a special way in the generated code for greater efficiency and to create specific, higher quality components that have more information about the representation.

- Change the implementation of neighborhoods. The current implementation defines that neighborhoods return a vector with the neighbors of the solution. However, performance improvements could be achieved in some componentes, e.g., eager local searches, by not returning a vector, but an iterator. So that all neighboring solutions are not necessarily calculated if the component that wants to make use of the neighborhood does not need it.

- Facilitate the use of the tool through documentation. It would be very useful for a user to have documentation for each component and guides on how to use Prodef. Usage data could be stored for each component, providing statistics to the user on which components are the most used and even recommending specific components according to the algorithm and problem they want to solve.

- Facilitate the creation of new components. One idea for this is to generate the JSON definition of each component automatically from the code, as discussed in Section 4.9

Capítulo 8

Presupuesto

En este capítulo se presenta el presupuesto del trabajo realizado. El coste de este proyecto proviene del tiempo que se ha empleado en su desarrollo. Para estimar el coste por hora se han consultado varias plataformas de comparación de sueldos y se ha elegido la cifra de 12€ por hora.

Tabla 8.1: Presupuesto

Nombre	**Horas**	**Coste (€)**
Investigar el funcionamiento de Prodef	25	300
Leer literatura sobre metaheurísticas	25	300
Diseñar componentes	50	600
Adaptar prodef-gui	80	960
Desarrollar el nuevo compilador	20	240
Desarrollar el nuevo resolutor	100	1200
Total	300	3600

Apéndice A

Problema de la mochila definido con ProdefLang

```
{
  "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
      }
    ]
  }
]
```

```
156 }
```

Apéndice B

Problema de la mochila tras ser compilado


```
{
  "metadata": {
    "numberOfConstraints": 1,
    "numberOfGoals": 1,
    "numberOfVariables": 1,
    "numberOfAtomicVariables": 5,
    "problemName": "Basic binary knapsack problem",
    "problemType": "binary"
  },
  "parameters": [
    {
      "index": 0,
      "expression": "int __N = 5;",
      "symbol": "__N",
      "originalSymbol": "N",
      "dimensions": 0,
      "rows": 1,
      "columns": 1,
      "source": "parameter",
      "within": "integers"
    },
    {
      "index": 1,
      "expression": "int __MaxWeight = 80;",
      "symbol": "__MaxWeight",
      "originalSymbol": "MaxWeight",
      "dimensions": 0,
      "rows": 1,
      "columns": 1,
      "source": "parameter",
      "within": "integers"
    }
  ],
  "objects": [
    {
      "index": 0,
      "expression": "double[][] __item = {{0, 33, 15},{0, 24, 20},{0, 36, 17},{0, 37, 8},
    {0, 12, 31}};",
      "symbol": "__item",
```

```
      "originalSymbol": "item",
      "dimensions": 2,
      "rows": 5,
      "columns": 3,
      "source": "class",
      "within": "reals"
    }
  ],
  "variables": [
    {
      "index": 0,
      "name": "Items in the knapsack",
      "shape": {
        "type": "vector",
        "isPermutation": false,
        "size": {
          "fixed": false,
          "value": "N"
        }
      },
      "expression": "int[] __x = new int[(int) (5)];",
      "range": {
        "lowerBound": "(int) (0)",
        "upperBound": "(int) (1)"
      },
      "originalRange": {
        "lowerBound": 0,
        "upperBound": 1
      },
      "symbol": "__x",
      "originalSymbol": "x",
      "dimensions": 1,
      "rows": 1,
      "columns": 5,
      "size": 5,
      "source": "variable",
      "within": "integers"
    }
  ],
  "functions": [
    {
      "index": 0,
      "expression": "double cos(double number) { return Math.cos(number); }",
      "name": "cos",
      "arguments": [
        0
      ]
    },
    {
      "index": 1,
      "expression": "double sin(double number) { return Math.sin(number); }",
      "name": "sin",
      "arguments": [
        0
      ]
    },
    {
      "index": 2,
```

```
      "expression": "double sqrt(double number) { return Math.sqrt(number); }",
      "name": "sqrt",
      "arguments": [
        0
      ]
    }
  ],
  "goals": [
    {
      "index": 0,
      "name": "Maximize the value of the items",
      "originalExpression": "sum x[i]*item[i].value over i=(1:N)",
      "expression": "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)",
      "sense": "maximize",
      "weight": 1
    }
  ],
  "constraints": [
    {
      "index": 0,
      "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) <=
    __MaxWeight",
      "violationExpression": "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)"
    }
  ]
}
```

Apéndice C

Problema del viajante de comercio definido con ProdefLang

```
{
  "name": "TSP",
  "description": "Optional description (optimal: 130)",
  "parameters": [
    {
      "name": "N",
      "symbol": "N",
      "value": 5
    }
  ],
  "classes": [
    {
      "name": "Distances",
      "symbol": "Distances",
      "attributes": [
        {
          "name": "col1",
          "symbol": "col1"
        },
        {
          "name": "col2",
          "symbol": "col2"
        },
        {
          "name": "col3",
          "symbol": "col3"
        },
        {
          "name": "col4",
          "symbol": "col4"
        },
        {
          "name": "col5",
          "symbol": "col5"
        }
      ]
    }
  ],
  "variables": [
```

```
    {
      "name": "city",
      "symbol": "city",
      "within": "integers",
      "shape": {
        "type": "vector",
        "isPermutation": true,
        "size": {
          "fixed": false,
          "value": "N"
        }
      }
    }
  ],
  "goals": [
    {
      "name": "goal name",
      "sense": "minimize",
      "weight": 1,
      "expression": "(sum Distances[city[i]][city[(i) + (1)]] over i = (1:(N) - (1))) + (
   Distances[city[N]][city[1]])"
    }
  ],
  "constraints": [],
  "objects": [
    {
      "class": "Distances",
      "attributes": [
        {
          "attribute": "col1",
          "value": 0
        },
        {
          "attribute": "col2",
          "value": 1
        },
        {
          "attribute": "col3",
          "value": 6
        },
        {
          "attribute": "col4",
          "value": 2
        },
        {
          "attribute": "col5",
          "value": 1
        }
      ]
    },
    {
      "class": "Distances",
      "attributes": [
        {
          "attribute": "col1",
          "value": 2
        },
        {
```

```
          "attribute": "col2",
          "value": 0
        },
        {
          "attribute": "col3",
          "value": 5
        },
        {
          "attribute": "col4",
          "value": 1
        },
        {
          "attribute": "col5",
          "value": 1
        }
      ]
    },
    {
      "class": "Distances",
      "attributes": [
        {
          "attribute": "col1",
          "value": 3
        },
        {
          "attribute": "col2",
          "value": 2
        },
        {
          "attribute": "col3",
          "value": 0
        },
        {
          "attribute": "col4",
          "value": 5
        },
        {
          "attribute": "col5",
          "value": 6
        }
      ]
    },
    {
      "class": "Distances",
      "attributes": [
        {
          "attribute": "col1",
          "value": 7
        },
        {
          "attribute": "col2",
          "value": 3
        },
        {
          "attribute": "col3",
          "value": 8
        },
        {
```

```
          "attribute": "col4",
          "value": 0
        },
        {
          "attribute": "col5",
          "value": 9
        }
      ]
    },
    {
      "class": "Distances",
      "attributes": [
        {
          "attribute": "col1",
          "value": 5
        },
        {
          "attribute": "col2",
          "value": 7
        },
        {
          "attribute": "col3",
          "value": 1
        },
        {
          "attribute": "col4",
          "value": 3
        },
        {
          "attribute": "col5",
          "value": 0
        }
      ]
    }
  ]
}
```

Apéndice D

Problema del viajante de comercio tras ser compilado


```
{
  "metadata": {
    "numberOfConstraints": 0,
    "numberOfGoals": 1,
    "numberOfVariables": 1,
    "numberOfAtomicVariables": 4,
    "problemName": "Basic TSP problem",
    "problemType": "integerPermutation"
  },
  "parameters": [
    {
      "index": 0,
      "expression": "const N___: i64 = 4;",
      "symbol": "N___",
      "originalSymbol": "N",
      "dimensions": 0,
      "rows": 1,
      "columns": 1,
      "source": "parameter",
      "within": "integers"
    }
  ],
  "objects": [
    {
      "index": 0,
      "expression": "const DISTANCE___: [[f64; 4]; 4] = [[0 as f64, 12 as f64, 25 as f64,
     17 as f64], [12 as f64, 0 as f64, 35 as f64, 8 as f64], [25 as f64, 35 as f64, 0 as
    f64, 11 as f64], [17 as f64, 8 as f64, 11 as f64, 0 as f64]];",
      "symbol": "distance___",
      "originalSymbol": "distance",
      "dimensions": 2,
      "rows": 4,
      "columns": 4,
      "source": "class",
      "within": "reals"
    }
  ],
  "variables": [
    {
```

```
      "index": 0,
      "name": "Visited cities",
      "shape": {
        "type": "vector",
        "isPermutation": true,
        "size": {
          "fixed": false,
          "value": "N"
        }
      },
      "expression": "let mut city___: [i64; 4] = [0; 4];",
      "range": {
        "lowerBound": "i64::MIN",
        "upperBound": "i64::MAX"
      },
      "originalRange": {
        "lowerBound": "-Infinity",
        "upperBound": "Infinity"
      },
      "symbol": "city___",
      "originalSymbol": "city",
      "dimensions": 1,
      "rows": 1,
      "columns": 4,
      "size": 4,
      "source": "variable",
      "within": "integers"
    }
  ],
  "functions": [
    {
      "index": 0,
      "expression": "fn cos(number: f64) -> f64 { number.cos() }",
      "name": "cos",
      "arguments": [
        0
      ]
    },
    {
      "index": 1,
      "expression": "fn sin(number: f64) -> f64 { number.sin() }",
      "name": "sin",
      "arguments": [
        0
      ]
    },
    {
      "index": 2,
      "expression": "fn sqrt(number: f64) -> f64 { number.sqrt() }",
      "name": "sqrt",
      "arguments": [
        0
      ]
    }
  ],
  "goals": [
    {
      "index": 0,
```

```
      "name": "Minimize the distance traveled",
      "originalExpression": "sum distance[city[i], city[i+1]] over i=(1:N-1) + distance[
    city[N], city[1]]",
      "expression": "((std::cmp::min((1) as usize, ((N___ as f64) - (1 as f64)) as usize)
    ..=std::cmp::max((1) as usize, ((N___ as f64) - (1 as f64)) as usize)).map(|i___| (
    DISTANCE___[(city___[(i___) as usize - 1]) as usize - 1][(city___[((i___ as f64) + (1
     as f64)) as usize - 1]) as usize - 1]) as f64).sum::<f64>() as f64) + (DISTANCE___[(
    city___[(N___) as usize - 1]) as usize - 1][(city___[(1) as usize - 1]) as usize - 1]
     as f64)",
      "sense": "minimize",
      "weight": 1
    }
  ],
  "constraints": []
}
```

Apéndice E

Plantilla para generar código Rust

```
use serde_json::json;
use std::time::Instant;

use prodef_components_definition::*;
use prodef_default_components::{
    main_components::*, specific_components::matrix::*, specific_components::permutation::*,
    specific_components::value::*, specific_components::vector::*,
};

{{#each parameters}}
{{{this}}}
{{/each }}

{{#each objects}}
{{{this}}}
{{/each }}

{{#each functions}}
{{{this}}}
{{/each }}

struct {{problemName}} {
    best_solution: {{{type}}}{{{genericArguments}}},
}

impl Problem<{{{type}}}{{{genericArguments}}}> for {{problemName}} {
    fn evaluate_goal(&self, index: usize, solution: &{{{type}}}{{{genericArguments}}}) -> Option<f64> {
        let {{{variableSymbol}}} = solution.inner;
        match index {
            {{#each goals}}
            {{index}} => Some({{{expression}}},),
            {{/each }}
            _ => None,
        }
    }

    fn number_of_goals(&self) -> usize {
        {{numberOfGoals}}
    }
```

```
40
41      {{#if constraints}}
42      fn check_constraint(&self, index: usize, solution: &{{{type}}}{{{genericArguments}}})
         -> Option<bool> {
43          let {{{variableSymbol}}} = solution.inner;
44          match index {
45              {{#each constraints}}
46                  {{index}} => Some({{{expression}}},),
47              {{/each}}
48              _ => None,
49          }
50      }
51
52      fn check_constraint_violation(&self, index: usize, solution: &{{{type}}}{{{
        genericArguments}}}) -> Option<f64> {
53          let {{{variableSymbol}}} = solution.inner;
54          match index {
55              {{#each constraints}}
56                  {{index}} => Some({{{violationExpression}}},),
57              {{/each}}
58              _ => None,
59          }
60      }
61
62      fn number_of_constraints(&self) -> usize {
63          {{numberOfConstraints}}
64      }
65      {{/if}}
66
67      fn get_best_solution(&mut self) -> &{{{type}}}{{{genericArguments}}} {
68          &self.best_solution
69      }
70
71      fn get_best_solution_value(&self) -> f64 {
72          self.evaluate_solution(&self.best_solution)
73      }
74
75      fn update_best_solution(&mut self, solution: {{{type}}}{{{genericArguments}}}) {
76          self.best_solution = solution;
77      }
78  }
79
80  fn main() {
81      let mut algorithm = {{{algorithm}}};
82      let problem = {{problemName}} {
83          best_solution: {{type}}::new(),
84      };
85      let start = Instant::now();
86      let problem = algorithm.execute(problem);
87      let time = start.elapsed();
88      let json = json!({
89          "computingTime": time.as_millis() as usize,
90          "isFeasible": problem.check_feasibility(&problem.best_solution),
91          "goalValues": (0..problem.number_of_goals()).map(|i| problem.evaluate_goal(i, &
        problem.best_solution)).collect::<Vec<_>>(),
92          "variableValue": problem.best_solution.inner
93      });
94      print!("{}", json);
```

```
95 }
```

Apéndice F

Definición de un algoritmo GRASP sencillo

```
{
  "name": "Simple GRASP",
  "components": [
    {
      "name": "FixedRepetition",
      "title": "Repetition",
      "description": "Repeats a process for a fixed number of times",
      "safety": "both",
      "dependencies": [
        {
          "type": "integer",
          "value": 1,
          "title": "The number of times to repeat the process"
        },
        {
          "type": "algorithm",
          "algorithm": [
            {
              "name": "BasicGenerator",
              "title": "Generate random solutions",
              "description": "Uses an specific generator to generate random solutions
    until they are feasible",
              "safety": "safe",
              "dependencies": [
                {
                  "type": "generator",
                  "name": "SimplePermutationGenerator",
                  "variableShape": "permutation",
                  "arguments": [],
                  "title": "The specific generator to use"
                },
                {
                  "type": "integer",
                  "value": 1,
                  "title": "The number of solutions to generate"
                }
              ]
            },
            {
```

```
              "name": "LocalSearch",
              "title": "Perform a local search",
              "description": "Performs a local search in the active solution using a provided neighborhood",
              "safety": "safe",
              "dependencies": [
                {
                  "type": "neighborhood",
                  "name": "PermutationSwapNeighborhood",
                  "variableShape": "permutation",
                  "arguments": [],
                  "title": "The specific neighborhood to use"
                }
              ]
            },
            {
              "name": "UpdateBestSolution",
              "title": "Updates the best solution",
              "description": "Explores the active solutions and updates the actual best solution if needed. Only uses feasible solutions",
              "safety": "both",
              "dependencies": []
            }
          ],
          "title": "The process to repeat"
        }
      ]
    }
  ]
}
```

Apéndice G

Ejemplo de código generado para la resolución de un problema

```rust
use serde_json::json;
use std::time::Instant;

use prodef_components_definition::*;
use prodef_default_components::{
    main_components::*, specific_components::matrix::*, specific_components::permutation::*,
    specific_components::value::*, specific_components::vector::*,
};

const N___: i64 = 5;

const DISTANCES___: [[f64; 5]; 5] = [
    [0 as f64, 1 as f64, 6 as f64, 2 as f64, 1 as f64],
    [2 as f64, 0 as f64, 5 as f64, 1 as f64, 1 as f64],
    [3 as f64, 2 as f64, 0 as f64, 5 as f64, 6 as f64],
    [7 as f64, 3 as f64, 8 as f64, 0 as f64, 9 as f64],
    [5 as f64, 7 as f64, 1 as f64, 3 as f64, 0 as f64],
];

fn cos(number: f64) -> f64 {
    number.cos()
}
fn sin(number: f64) -> f64 {
    number.sin()
}
fn sqrt(number: f64) -> f64 {
    number.sqrt()
}

struct TSP___ {
    best_solution: Permutation<5>,
}

impl Problem<Permutation<5>> for TSP___ {
    fn evaluate_goal(&self, index: usize, solution: &Permutation<5>) -> Option<f64> {
        let city___ = solution.inner;
        match index {
            0 => Some(
```

```
                -(1 as f64
                    * (((std::cmp::min((1) as usize, (((N___) as f64) - ((1) as f64)) as usize)
                        ..=std::cmp::max((1) as usize, (((N___) as f64) - ((1) as f64)) as usize))
                        .map(|i___| {
                            (DISTANCES___[(city___[(i___) as usize - 1]) as usize - 1][(city___
                                [(((i___) as f64) + ((1) as f64)) as usize - 1])
                                as usize
                                - 1]) as f64
                        })
                        .sum::<f64>()) as f64)
                    + ((DISTANCES___[(city___[(N___) as usize - 1]) as usize - 1]
                        [(city___[(1) as usize - 1]) as usize - 1]) as f64)),
            ),
            _ => None,
        }
    }

    fn number_of_goals(&self) -> usize {
        1
    }

    fn get_best_solution(&mut self) -> &Permutation<5> {
        &self.best_solution
    }

    fn get_best_solution_value(&self) -> f64 {
        self.evaluate_solution(&self.best_solution)
    }

    fn update_best_solution(&mut self, solution: Permutation<5>) {
        self.best_solution = solution;
    }
}

fn main() {
    let mut algorithm = Algorithm::new().add_component(Box::new(FixedRepetition::new(
        1 as i64,
        Algorithm::new()
            .add_component(Box::new(BasicGenerator::new(
                SimplePermutationGenerator::<5>::new(),
                1 as i64,
            )))
            .add_component(Box::new(LocalSearch::new(
                PermutationSwapNeighborhood::<5>::new(),
            )))
            .add_component(Box::new(UpdateBestSolution::new())),
    )));
    let problem = TSP___ {
        best_solution: Permutation::new(),
    };
    let start = Instant::now();
    let problem = algorithm.execute(problem);
    let time = start.elapsed();
    let json = json!({
        "computingTime": time.as_millis() as usize,
```

```
        "isFeasible": problem.check_feasibility(&problem.best_solution),
        "goalValues": (0..problem.number_of_goals()).map(|i| problem.evaluate_goal(i, &
    problem.best_solution)).collect::<Vec<_>>(),
        "variableValue": problem.best_solution.inner
    });
    print!("{}", json);
}
```

Bibliografía

[1] Sébastien Cahon, Nordine Melab, and E-G Talbi. Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of heuristics*, 10(3):357–380, 2004.

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

[3] Luca Di Gaspero and Andrea Schaerf. Easylocal++: an object-oriented framework for the flexible design of local-search algorithms. *Software: Practice and Experience*, 33(8):733–765, 2003.

[4] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, 1996.

[5] Johann Dreo, Arnaud Liefooghe, Sébastien Verel, Marc Schoenauer, Juan J Merelo, Alexandre Quemy, Benjamin Bouvier, and Jan Gmys. Paradiseo: from a modular framework for evolutionary computation to the automated design of metaheuristics: 22 years of paradiseo. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1522–1530, 2021.

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

[7] Thomas A Feo and Mauricio GC Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations research letters*, 8(2):67–71, 1989.

[8] Neil Fraser. Ten things we've learned from blockly. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, pages 49–50, 2015.

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

[10] Fred Glover. Tabu search—part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

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

[12] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, volume 4, pages 1942–1948. IEEE, 1995.

[13] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[14] Sean Luke, Liviu Panait, Gabriel Balan, Sean Paus, Zbigniew Skolicki, Jeff Bassett, Robert Hubley, and A Chircop. Ecj: A java-based evolutionary computation research system. *Downloadable versions and documentation can be found at the following url: http://cs. gmu. edu/eclab/projects/ecj*, 880, 2006.

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

[16] José Antonio Parejo, Antonio Ruiz-Cortés, Sebastián Lozano, and Pablo Fernandez. Metaheuristic optimization frameworks: a survey and benchmarking. *Soft Computing*, 16(3):527–561, 2012.

[17] Moshe Sipper, Tomer Halperin, Itai Tzruia, and Achiya Elyasaf. EC-KitY: Evolutionary computation tool kit in Python. https://www.eckity.org/, 2022.

[18] Stefan Wagner and Michael Affenzeller. Heuristiclab: A generic and extensible optimization environment. In *Adaptive and Natural Computing Algorithms*, pages 538–541. Springer, 2005.

[19] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.