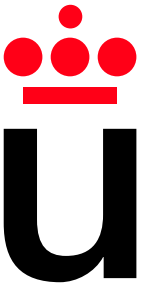


Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

Doble grado en Diseño y Desarrollo de Videojuegos e Ingeniería de Computadores

Curso 2022-2023

Trabajo Fin de Grado

METAHEURÍSTICAS PARA EL MINIMUM WEIGHTED DOMINATING SET PROBLEM

Autor: David González Rueda
Tutor: Jesús Sánchez-Oro Calvo
Cotutor: Alejandra Casado Ceballos

Agradecimientos

A mi madre Susana, por su inagotable paciencia, cariño y enseñanza; a Naia, por su amor y apoyo; a mi familia, por confiar en mí y darme fuerzas para seguir; y a mis tutores, Jesús y Alejandra, cuyo esfuerzo y guía han permitido que este trabajo haya sido posible. Gracias.

Resumen

El mínimo conjunto dominante de un grafo se compone del menor número de nodos que, o bien pertenecen a este subconjunto, o bien son adyacentes a alguno de los nodos que lo forma. Este Trabajo de Fin de Grado (TFG) propone un algoritmo para la resolución del problema *Minimum Weighted Dominating Set*, variante del problema anterior en el que a cada uno de los nodos le corresponde un peso determinado, y en el que se debe seleccionar el mínimo conjunto dominante de menor peso, aprovechando las ventajas de las metaheurísticas *Greedy Randomized Adaptive Search Procedure* (GRASP) e *Iterated Greedy*(IG).

Palabras clave:

- Dominio de Grafos
- Metaheurísticas
- Greedy Randomized Adaptive Search Procedure
- Iterated Greedy
- MWDSP

Índice de contenidos

Índice de tablas

Índice de figuras

Índice de códigos

1 Introducción

La finalidad de esta sección es contextualizar al lector sobre el problema que será resuelto, proporcionando las herramientas adecuadas para comprender la solución propuesta. Asimismo, se presenta una investigación y reflexión sobre las resoluciones precedentes, así como la estructura de los contenidos del presente documento.

1.1. Contexto y alcance

El dominio de grafos ha sido objeto de estudio por diversos ámbitos científicos durante un largo tiempo, adquiriendo cada vez más relevancia en la inminente realidad tecnológica. El objetivo principal se reduce a minimizar el número de nodos que dominan al resto, es decir, que cumplen que cualquier nodo del grafo se encuentra entre este subconjunto, o tiene una arista que lo conecta a alguno de los nodos seleccionados.

Este tipo de problemas tienen un gran número de variantes, dependiendo de las restricciones y reglas aplicadas sobre el grafo a resolver. Este Trabajo de Fin de Grado (TFG) se enfoca en la resolución del problema del conjunto dominante de mínimo peso (MWDSP, del inglés *Minimum Weighted Dominating Set Problem*), variante del problema presentado anteriormente. Dado un grafo $G = (V, A)$, en el que cada uno de los k elementos de V tienen un peso w, se busca un subconjunto $S \subseteq V$, del menor peso posible, que domine todos los nodos de G (ver la Sección 1.2 para la definición formal del problema).

El MWDSP es considerado $\mathcal{NP}$-completo [1] y, por lo tanto, no puede garan-

tizarse la construcción de la solución óptima con métodos exactos que puedan ser aplicadas a instancias de gran tamaño (para aplicaciones de gran escala, como el minado de datos), en un tiempo de CPU razonable.

Las aplicaciones de la dominación de grafos son ricas y variadas. Se incluyen entre ellas la creación de redes de influencia positiva [2, 3, 4], utilizadas para la creación de programas de intervención de problemas sociales (como la drogadicción, la bebida o el tabaquismo), y basadas en la premisa de evitar posibles recaídas destinando recursos al individuo y a un subconjunto de su red de contactos para que el programa se expanda a través del mayor número de personas afectadas posible.

También es utilizado en el diseño de redes inalámbricas ad-hoc [5], en las que se trata de ofrecer la máxima cobertura para prevenir fallos causados por nodos pertenecientes a la misma, modificando la topología de red de forma dinámica mediante la selección de un subconjunto de los mejores nodos (atendiendo a la conectividad y cobertura).

El dominio de grafos también tiene su lugar en el ámbito biológico [6, 7], donde ha sido utilizado para definir, de la red humana de interacción de proteínas, subconjuntos de éstas que, además de ofrecer el mayor control sobre la red, contienen genes esenciales, relacionados con el cáncer y orientados a virus.

Esta familia de problemas podría ser aplicada también en el ámbito de los videojuegos. Para que exista un buen diseño de niveles, al usuario se le deben de presentar unas mecánicas concretas, proporcionándole inmediatamente después un reto simple para que adopte dichas mecánicas en su estilo de juego. Se expone a continuación un pequeño problema ejemplo:

En el videojuego 'Ladrón por sorpresa', el jugador deberá evitar a toda cosa los guardias de seguridad del museo. Sin embargo, ya que el director del museo ha detectado que algunos objetos de su oficina han desaparecido, ha decidido instalar unas cámaras de seguridad. Como 'Game Designer', tu cometido es colocarlas en el escenario, de tal forma que eviten que el usuario se sienta abrumado por la dificultad, pero que, a su vez, se cubran todas las salas. El usuario se sentirá abrumado si hay un gran número de guardias en la sala y una cámara de seguridad.

En este problema, se puede considerar cada una de las salas como un nodo del grafo, las puertas y pasillos que conectan las salas como las aristas y los guardias de seguridad como los pesos, ya que se quiere evitar que se coloque una cámara en una sala que esté repleta de enemigos. De esta forma, se puede abstraer cualquier escenario compuesto por salas como una instancia de MWDSP.

1.2. Definición formal del problema

Un grafo no dirigido es definido como $G = (V, A)$, donde V representa el conjunto de vértices y A el conjunto de aristas. Una arista $(u, v) \in A$, con $u, v \in V$, representa la existencia de una relación entre los vértices u y v. Por otra parte, un conjunto dominante en un grafo $G = (V, A)$ consiste en un subconjunto $S \subseteq V$ que cumple que, para cada vértice $u \in V \backslash S$, existe un vértice $v \in S$ tal que $(u, v) \in A$.

Este trabajo se orienta a la resolución del problema MWDSP, donde cada uno de los vértices $v \in V$ del grafo tienen un peso $w \in \mathbb{N} \mid \forall w\ (w > 0)$. El objetivo de este problema es encontrar un conjunto dominante $S \subseteq V$ de mínimo peso (consultar ecuación 1.1). Además, para facilitar la explicación de los algoritmos utilizados en los capítulos posteriores, se representarán los nodos que se encuentran dominados con el conjunto CV, así como los nodos adyacentes de un nodo i como $N(i)$, tal que $N(i) = \{\ j \mid (i, j) \in A\ \}$.

Tomando el peso de cada vértice i como w_i, y el subconjunto dominante $S \subseteq V$ explicado anteriormente, el valor de la función objetivo puede ser calculada como:

$$\text{MWDSP}(S) = \sum_{i \in S} w_i \tag{1.1}$$

El objetivo del MWDSP es encontrar el subconjunto S^* con el mínimo valor de MWDSP(S^*). Formalmente,

$$S^* = \arg\min_{S \in \mathbb{S}} \text{MDWSP}(S) \tag{1.2}$$

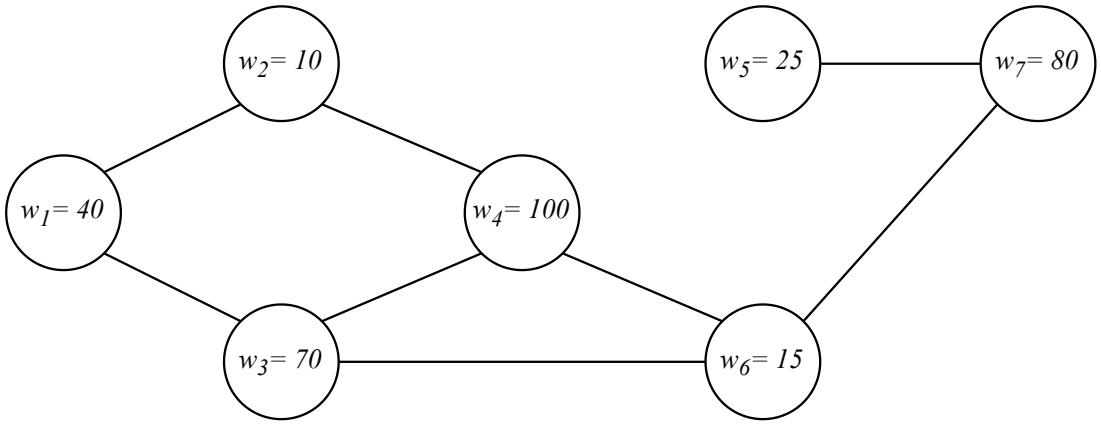
donde $\mathbb{S}$ representa todos los posibles subconjuntos dominantes de la instancia, es decir, todo el espacio de soluciones factibles.

La figura ?? muestra un grafo, posible instancia de un problema MWDSP, compuesto por 7 nodos, cada uno con un peso determinado, y 8 aristas, que los conecta entre sí. Se considerará que el peso w_1 corresponde al v_1 y así sucesivamente, evitando así posible información redundante en el gráfico.

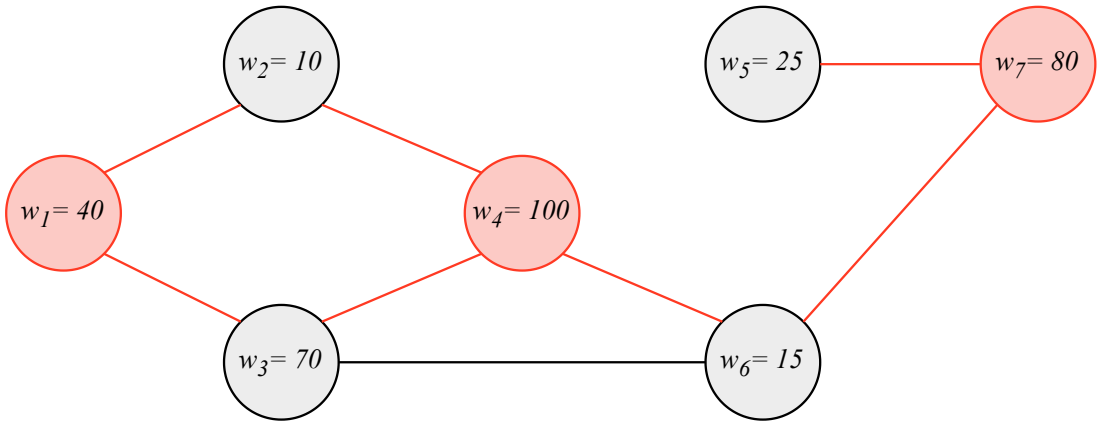
Asimismo, a la instancia le acompañan dos soluciones factibles, S_1 y S_2. En ambas, el fondo de los nodos pertenecientes a la solución han sido coloreados de color rojo, así como las aristas que hacen que dominen a sus nodos adyacentes. Aquellos de color gris claro son los nodos dominados.

Mientras que la solución S_1 está formada por los nodos $S_1 = \{v_1, v_4, v_7\}$, con un valor de la función objetivo de MWDSP(S_1) $= w_1 + w_4 + w_7 = 220$, la solución S_2 está formada por los nodos $S_2 = \{v_2, v_5, v_6\}$, con un valor de la función objetivo

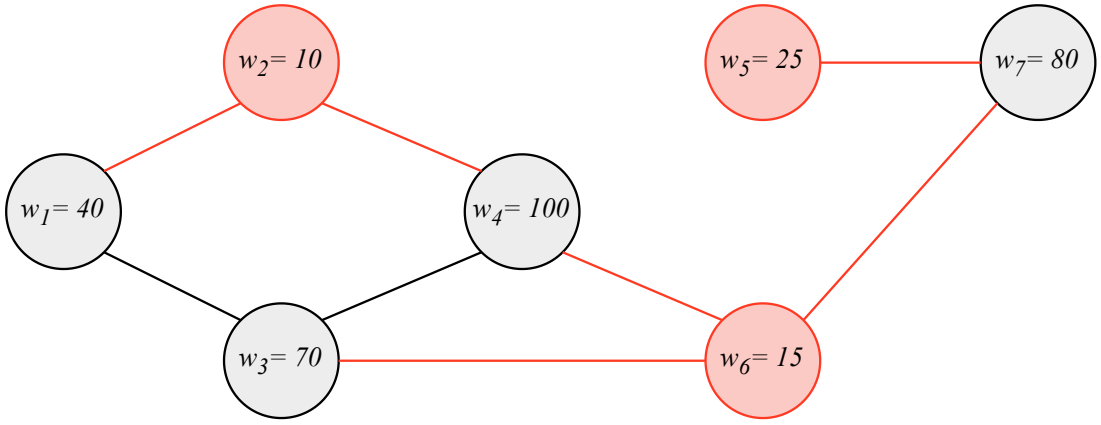
de MWDSP(S_2) $= w_2 + w_5 + w_6 = 50$. Buscando aquel subconjunto dominante de menor peso, y siendo un problema de minimización, la solución S_2 es mejor que S_1.



(a) Instancia compuesta de 7 nodos o vértices y 8 aristas.



(b) Solución $S_1 = \{v_1, v_4, v_7\}$.



(c) Solución $S_2 = \{v_2, v_5, v_6\}$.

Figura 1.1: Instancia del problema MWDSP compuesta por 7 nodos y 8 aristas 1.1(a), y dos soluciones posibles que la resuelven: S_1 1.1(b) y S_2 1.1(c)

Por otra parte, una solución no factible será aquella en la que los nodos seleccionados no dominen la totalidad de los nodos del grafo. Es decir, que $\exists i \in V \backslash S : \nexists j : (i,j) \in A$. Un posible ejemplo de una solución no factible, tomando la instancia de la figura 1.1(a), sería el siguiente:

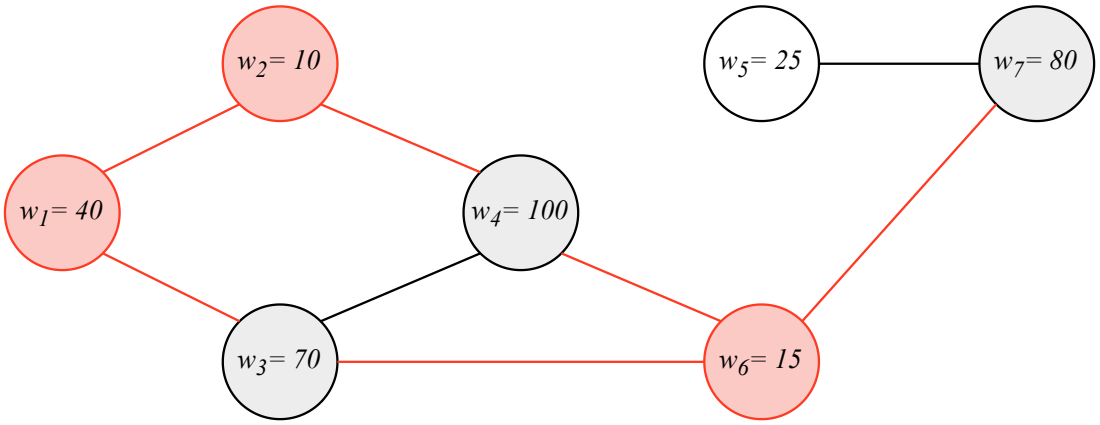


Figura 1.2: Ejemplo de solución no factible de una instancia MWDSP. El nodo v_5 no estaría dominado por ningún nodo seleccionado.

A continuación, se muestra un mapa de ejemplo 1.3 del videojuego *'Ladron por sorpresa'*, explicado en la sección 1.1, formado por el escenario del museo (compuesto por 5 salas interconectadas), donde los puntos de color representan a los guardias, y su abstracción a una instancia MWDSP:

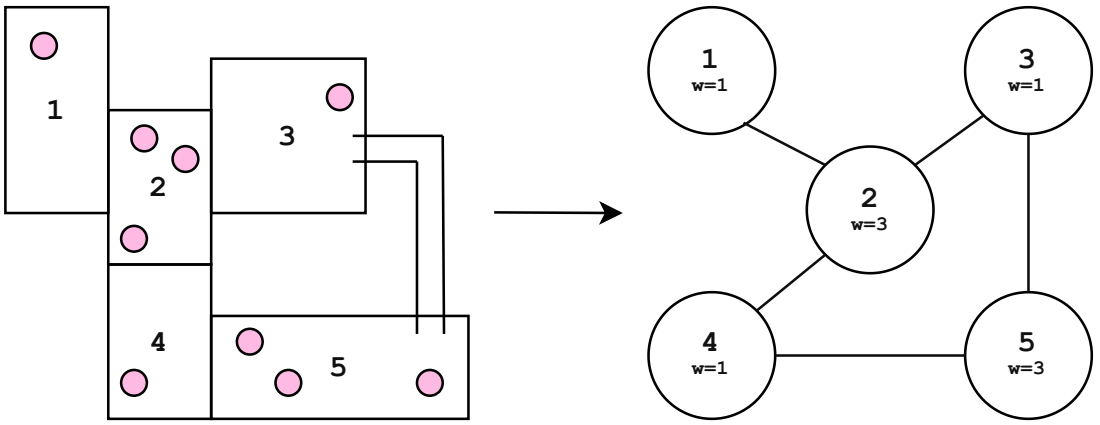


Figura 1.3: Mapa del videojuego 'Ladrón por sorpresa' y su abstracción en una instancia del problema MWDSP

Las cámaras deberían de colocarse en la sala 1, 3 y 4, de tal forma que todas las salas estuvieran vigiladas, pero el número de enemigos en las salas donde existe una cámara de seguridad fuese el mínimo posible.

1.3. Resumen del estado del arte

Como se ha descrito anteriormente, MWDSP puede ser visto como una variante del problema del Subconjunto Dominante, o Set Covering Problem (SC) [8]. Aunque se han desarrollado soluciones altamente eficientes para este tipo de problemas [9], no son aplicables al problema MWDSP, ya que las instancias suelen ser más extensas, con grafos que contienen un gran número de nodos y una alta densidad de aristas, dando lugar a matrices de adyacencia complejas.

Inicialmente, en investigaciones anteriores, se aplicaron algoritmos de aproximación para el MWDSP en un tipo específico de grafos conocidos como Unit Disk Graphs (UDG), relacionados estrechamente con la aplicación del diseño de redes inalámbricas. Un UDG [10] es un grafo de intersección que se construye considerando que cada uno de los nodos o vértices del grafo corresponden a un sensor en un plano euclidiano, y se unen mediante una arista solo si su distancia en el plano es menor a una unidad. Por otro lado, los algoritmos de aproximación, a diferencia de las heurísticas, se enfocan en encontrar soluciones de calidad y cuyos tiempos de cómputo están acotados por cotas conocidas. Para un mejor contexto, el ratio de aproximación de un algoritmo de aproximación se define como la razón entre la solución obtenida por el algoritmo y la solución óptima o el valor óptimo del problema. Se busca que este ratio sea lo más cercano posible a 1, lo que indica que el algoritmo produce soluciones muy cercanas a la óptima (si el ratio es 1, el algoritmo siempre encontraría las soluciones óptimas).

El primer algoritmo de aproximación de factor constante fue propuesto por Ambuhl et al. [11], logrando un ratio de 72. Sin embargo, este ratio ha sido progresivamente mejorado, hasta $(6+\epsilon)$ en [12] utilizando una estrategia de "doble partición" $(10 + \epsilon)$ sumando ambas fases de la partición). Posteriormente, en [13, 14] se consiguieron ratios de $(5 + \epsilon)$ y $(4 + \epsilon)$, respectivamente.

Seguidamente, el primer esquema de aproximación de tiempo polinomial (o PTAS en inglés) fue propuesto para el MWDSP [15], con un ratio de aproximación de $(1 + \epsilon)$, para el caso donde el ratio de los pesos de dos nodos adyacentes cualquiera estuviera acotado superiormente por una constante. Esta restricción fue resuelta en [16], cuando Li y Jin introdujeron un PTAS para cualquier instancia del problema.

No obstante, las metaheurísticas han adquirido una relevancia significativa en las investigaciones más recientes. Varias propuestas para abordar este problema se fundamentan en la utilización de búsquedas en poblaciones evolutivas. Un algoritmo hormiga o de optimización por colonia de hormigas (Ant Colony Optimization) con estrategia de corrección de feromonas (Raka-ACO) fue propuesto en [17]. Posteriormente, Potluri y Singh [18] introdujeron un algoritmo genético híbrido (HGA) y dos extensiones del algoritmo hormiga con una búsqueda local basada en la eliminación de nodos redundantes de la solución. En este último

algoritmo, se implementa un paso de preproceso después de la inicialización de las feromonas con el objetivo de fortalecer los valores de las feromonas asociados a 100 subconjuntos independientes generados de manera voraz (utilizando algoritmos *Greedy*).

Más tarde, Lin et al. [19] sugirieron un algoritmo memético que se basa en una técnica de construcción aleatoria voraz adaptable, junto a operaciones personalizadas de cruce y realinkado para resolver el problema. Por otro lado, en el artículo [20] se introdujo un algoritmo voraz iterativo conocido como R-PBIG. Este método mantiene una población de soluciones y utiliza pasos de deconstrucción y reconstrucción. Este enfoque se hibridó con un "solver" de ILP para mejorar las soluciones obtenidas.

Wang et al. [21] propusieron una variante de la búsqueda tabú llamada Algoritmo de Verificación de Configuración, que incluye un mecanismo que ajusta la función objetivo basándose en el historial de búsqueda. En [22] se presentó una búsqueda local aleatoria basada en el orden y con múltiples puntos de inicio, utilizando '*movimientos de salto*' para diversificar las soluciones.

Por último, se presentó un algoritmo de búsqueda tabú híbrida con un resolvedor IP (HTS-DS) por M. Albuquerque y T. Vidal en [23], que además hace uso de mecanismos de perturbación durante la búsqueda tabú para aumentar la exploración de posibles mejores soluciones.

Es importante destacar que cada uno de los métodos mencionados logró mejorar las soluciones en los conjuntos de instancias clásicas utilizados como referencia para el problema. Sin embargo, ninguno de estos métodos garantiza la obtención de las mejores soluciones conocidas u óptimas para todas las instancias, y su tiempo computacional tiende a ser elevado cuando se trabaja con grafos de gran tamaño.

1.4. Estructura del documento

El documento se estructura de la siguiente manera:

- En el Capítulo 2 se establecen los principales objetivos del proyecto. Se subdivide en dos secciones: la sección 2.1, en la que se define el objetivo principal, y la sección 2.2, que detalla los secundarios.
- En el Capítulo 3 se describe la propuesta algorítmica. Éste se divide en 3 secciones. La secciones 3.1, *Greedy Randomized Adaptive Search Procedure* y 3.2, *Iterated Greedy*, reflejan las metaheurísticas utilizadas en el algoritmo, mientras que en la sección 3.3 se presenta un esquema con la estructura final del mismo.

- En el Capítulo 4 se analiza la estructura, diseño e implementación de las clases principales del proyecto implementado.
- En el Capítulo 5 se muestran los resultados de los experimentos preliminares (en la sección 5.1), utilizados para establecer los mejores parámetros y otros aspectos del algoritmo, y el experimento final (sección 5.2), en que se realiza la comparativa del rendimiento del algoritmo frente al estado del arte.
- Por último, el Capítulo 6 resume las conclusiones derivadas de la investigación y el futuro trabajo que será realizado sobre el proyecto.

2

Objetivos

En esta sección, se presentan los objetivos a lograr durante el desarrollo del TFG. Mientras que la primera sección contiene el objetivo principal del TFG, la segunda define aquellos derivados del desarrollo del mismo.

2.1. Objetivo principal

El objetivo principal de este trabajo consiste en el diseño y desarrollo de un algoritmo que hace uso de metaheurísticas para la obtención de soluciones de alta calidad en un tiempo de ejecución reducido para el problema MWDSP.

2.2. Objetivos secundarios

Se han considerado los siguientes objetivos secundarios, derivados del objetivo principal:

- Expandir el conocimiento en el área de diseño y desarrollo de metaheurísticas y su aplicación en diferentes problemas que no pueden ser resueltos con algoritmos convencionales.
- Profundizar en el diseño de algoritmos de manera modular, de modo que

puedan ser fácilmente configurados y ajustados para resolver diferentes instancias de problemas o variaciones del mismo.

- Fortalecer y mejorar en el lenguaje de programación Java estudiado a lo largo del grado, orientándolo al uso de estructuras de datos eficientes para la reducción de la complejidad del algoritmo.
- Estudiar e integrar herramientas de paralelización en Java que obtienen soluciones precisas en un menor tiempo de cómputo que aquellas ejecutadas de forma secuencial.
- Promover el desarrollo de código limpio, legible y documentado, que pueda ser utilizado de forma sencilla con instrucciones de uso definidas.
- Integrar una metodología de trabajo ágil, desarrollando un proyecto de investigación que sigue el método científico.
- Explorar diferentes técnicas de optimización y ajustes de parámetros para lograr un rendimiento óptimo del algoritmo propuesto.
- Garantizar la calidad del algoritmo propuesto mediante la medición constante de los tiempos de ejecución, además de su almacenamiento y formato en hojas de cálculo, para su posterior comparación con el estado del arte.

3

Descripción algorítmica

En este capítulo se describen las metaheurísticas implementadas que constituyen el algoritmo final. Se introduce la metaheurística GRASP, con dos fases para la construcción inicial de la solución y la búsqueda de óptimos locales, intercaladas con una fase de purga adicional, así como la metaheurística *Iterated greedy* como proceso para la búsqueda de mejores soluciones mediante fases de destrucción y reconstrucción iterativas, intercalando intensificación y diversificación. Finalmente, se presenta un diagrama de flujo con la estructura final del algoritmo.

3.1. Greedy Randomized Adaptive Search Procedure

Para la resolución de instancias de MWDSP, este trabajo propone un algoritmo basado en el proceso *Greedy Randomized Adaptive Search Procedure* (GRASP) para la generación de soluciones de alta calidad y diversidad. Se trata de una metaheurística trayectorial multiarranque ampliamente utilizada en problemas de optimización, que fue presentada inicialmente en [24], pero definida formalmente en [25]. GRASP se compone de dos fases fundamentales: la construcción de la solución y la búsqueda local. Esta primera procura generar una solución fiable y de alta calidad desde cero [26], mientras que la segunda se enfoca en encontrar un óptimo local dentro de una vecindad basada en la solución generada en la fase inicial. Asimismo, se ha implementado un paso de purga intermedio que eli-

minará aquellos nodos redundantes de la solución, evitando añadir complejidad innecesaria a la búsqueda local. La principal ventaja que distingue a GRASP de un algoritmo voraz convencional radica en la incorporación del concepto de aleatoriedad durante la generación de la solución inicial. En contraste con las soluciones obtenidas mediante un enfoque voraz estándar, las soluciones generadas por GRASP difieren entre sí, lo que amplía la región explorada inicialmente en el espacio de soluciones. Si se desea ahondar en las aplicaciones de la metaheurística *GRASP* en otros problemas de optimización, pueden ser consultadas en [27, 28, 29].

3.1.1. Constructivo

El enfoque metodológico para la construcción de la solución utilizado en el algoritmo propuesto se denomina *Constructive Greedy-Random* (CGR), y sigue el esquema constructivo tradicional de GRASP. El fragmento 1 a continuación presenta el pseudocódigo de la estrategia constructiva propuesta:

Algoritmo 1 $\texttt{CGR}(I = G(V, A), \alpha_{GR})$

1: $v \leftarrow Random(V)$
2: $S \leftarrow \{v\}$
3: $CV \leftarrow \{v\} \cup N(v)$
4: $CL \leftarrow V \setminus \{v\}$
5: **while** $|CV| < n$ **do**
6: $\quad g_{min} \leftarrow min_{c \in CL}\ g(c)$
7: $\quad g_{max} \leftarrow max_{c \in CL}\ g(c)$
8: $\quad \mu \leftarrow g_{max} - \alpha_{GR} \cdot (g_{max} - g_{min})$
9: $\quad RCL \leftarrow \{c \in CL : g(c) \geq \mu\}$
10: $\quad v \leftarrow Random(RCL)$
11: $\quad S \leftarrow S \cup \{v\}$
12: $\quad CV \leftarrow CV \cup \{v\} \cup N(v)$
13: $\quad CL \leftarrow CL \backslash \{v\}$
14: **end while**
15: **return** S

En este constructivo, y como es típico en GRASP, el primer elemento se escoge al azar, favoreciendo la exploración del espacio de soluciones (paso 1). Este elemento se añade a la solución (paso 2), se inicializan los nodos dominados CV con el elemento y sus nodos adyacentes (paso 3) y se construye la lista de candidatos (CL) con todos los elementos menos el previamente seleccionado (paso 4). Mientras que la solución no sea factible, es decir, que exista algún nodo que no se encuentre dominado, se añaden elementos a la solución de forma iterativa utilizando un criterio voraz que será explicado a continuación (pasos 5 - 14).

En cada una de las iteraciones del bucle, la lista de candidatos no escogidos es evaluada con una función voraz $g(x)$. Para guiar la posterior construcción, se obtiene el mínimo (g_{min}) y el máximo valor (g_{max}) de la función voraz (pasos 6 y 7). Tomando en consideración el contexto del problema MWDSP y la necesidad de que la función voraz se ejecute rápidamente, ya que se calculará para cada uno de los nodos no seleccionados en cada iteración, se emplea la siguiente función:

$$g(i) = \frac{|N(i) \setminus CV|}{w_i} \tag{3.1}$$

Es decir, el valor de la función voraz se calcula como el número de conexiones del candidato a nodos no dominados entre el peso del nodo. De esta forma, un nodo con pocas conexiones y gran peso tendrá menos posibilidades de que sea escogido frente a uno con muchas conexiones y poco peso. Aunque el cálculo de las conexiones a nodos no dominados pudiera ser complejo ($O(n)$ en ciertos casos), se ha implementado una representación de la solución (explicada en la sección 4.2.2) que consigue que este cálculo sea considerablemente más sencillo ($O(1)$).

Posteriormente, el umbral μ es calculado con la función de establecer qué candidatos pertenecerán a la lista de candidatos restringida RCL (pasos 8 y 9). Este cálculo es dependiente del parámetro de entrada α_{GR}, cuyos valores pueden oscilar entre 0 y 1. Es importante destacar dos de los valores que puede tomar este parámetro:

1. Si el parámetro $\alpha_{GR} = 0$, el umbral $\mu = g_{max}$. Por lo tanto, la RCL únicamente estará compuesta del 'mejor' nodo (que cumple que $g(c) \geq g_{max}$). En caso de que existieran varios nodos con dicho valor de la función voraz, se escogería aleatoriamente uno de ellos. Esto resultará en un algoritmo constructivo completamente voraz.

2. Si el parámetro $\alpha_{GR} = 1$, el umbral $\mu = g_{min}$. Por lo tanto, la RCL se compondrá de todos los posibles nodos de la CL ($\forall c \in CL : g(c) \geq g_{min}$). Esto resultará en un algoritmo constructivo completamente aleatorio.

Considerando estos dos casos, se puede concluir que a menor valor de α_{GR} mayor componente voraz tendrá el constructivo. En la sección 5.1 se analizan varios valores de este parámetro y su impacto en la construcción de una solución óptima, encontrando un balance entre intensificación y diversificación.

Una vez se define la lista de candidatos restringida con el valor del umbral, se escoge aleatoriamente uno de ellos (paso 10) y se añade a la solución (paso 11). Tras ello, se actualizan tanto los nodos actualmente dominados, añadiendo el nodo seleccionado y sus adyacentes (paso 12), como la lista de candidatos,

retirando el nodo seleccionado de la misma (paso 13). El algoritmo constructivo termina cuando se encuentra una solución factible, retornándola (paso 15).

3.1.2. Purga

La metaheurística GRASP tradicional no incluye un paso intermedio de purga de nodos; sin embargo, ha sido incorporada con el fin de reducir complejidad innecesaria en la segunda etapa de GRASP, la búsqueda local. De este modo, se eliminan aquellos nodos redundantes cuya exclusión no afecta la solución, dado que todos sus nodos adyacentes seguirían estando dominados por otros nodos que ya pertenecen a la solución construida en la etapa previa.

El algoritmo 2 muestra el pseudocódigo del procedimiento de purga:

Algoritmo 2 `Purge`$(S = [v_a, v_b \dots v_n])$

1: **for** candidate $c \in S$ **do**
2: $\quad ANC = CV \cap (\{c\} \cup N(c))$
3: $\quad$ **if** $|ANC| = n$ **then**
4: $\quad\quad S \leftarrow S \setminus \{c\}$
5: $\quad\quad CV \leftarrow ANC$
6: $\quad$ **end if**
7: **end for**
8: **return** S

Durante la purga, se iterarán por todos los nodos que construyen la solución (paso 1). En cada iteración, se inicializará una variable ANC, que almacena los nodos dominados menos el nodo candidato y sus adyacentes (paso 2). En caso de que todos los nodos se encuentren dominados tras eliminar al candidato y sus adyacentes (paso 3), se eliminará el nodo finalmente de la solución (paso 4) y se actualizará el conjunto de nodos dominados (paso 5). Tras eliminar los nodos redundantes, se devolverá la solución ya purgada (paso 8).

3.1.3. Búsqueda local

La segunda fase de GRASP consiste en encontrar, dentro de la vecindad de la solución generada con el constructivo, y purgada con el paso intermedio anterior, un óptimo local. En GRASP se han abordado búsquedas locales tanto simples como muy complejas, utilizando hasta metaheurísticas completas en múltiples trabajos de investigación, como *Variable Neighborhood Search* en [30] o algoritmos genéticos en [31], para una exploración más exhausta del espacio de soluciones.

Se propone en este trabajo una búsqueda local (LS, del inglés *Local Search*)

simple, con el fin de encontrar el óptimo local en un tiempo computacional bajo. Para definir una búsqueda local, primero se debe de concretar cuál será el movimiento realizado. Para mantener la factibilidad de las soluciones generadas con esta búsqueda local, se plantea un movimiento de intercambio. Es decir, dado un elemento $v_i \in S$, y un conjunto $J = \{ v_1, v_2, ..., v_n \mid v_j \in (V \setminus S) \}$, se eliminará el elemento v_i de la solución, sustituyéndolo por el conjunto J. Adicionalmente, en el contexto de MWDSP, una solución se considera superior a otra si la suma del peso de sus nodos es menor. Por consiguiente, únicamente se deberá realizar el intercambio si peso del nodo v_i es mayor al peso del conjunto J, resultando así en una solución mejorada. Más formalmente:

$$Intercambio(S, v_i, J) = (S \setminus v_i) \cup J \Leftrightarrow w_i > w_J \tag{3.2}$$

De esta forma, se puede definir la vecindad $N(S)$ que será explorada durante la búsqueda local, siendo esta el conjunto de soluciones que pueden ser construidas realizando un único movimiento de intercambio. Matemáticamente:

$$N(S) = \{S' \leftarrow Intercambio(S, v_i, J),\ \forall v_i \in S \wedge \forall J \in (V \setminus S)\} \tag{3.3}$$

Para recorrer esta vecindad, existen dos principales enfoques tradicionales: *best* y *first improvement*. Mientras que el *best improvement* (BI) explora toda la vecindad al completo, almacenando y ejecutando el mejor intercambio en cada iteración, *first improvement* (FI) realiza el primer movimiento posible encontrado durante la exploración que mejore la solución desde la que se parte.

First improvement, habitualmente, es una solución computacionalmente más rápida que *best improvement*, aunque no asegura los mejores resultados en todas las ocasiones. Para MWDSP, las ejecuciones realizadas con BI en la sección 5.1 han asegurado unas soluciones superiores frente a aquellas realizadas con FI. Ya que *best improvement* realiza siempre el mejor movimiento en cada iteración, no es necesario ejecutar ningún código que altere inicialmente la solución, como sí ocurre en *first improvement*. Tomando en cuenta que se utiliza *best improvement* como estrategia de búsqueda local, se realizará aquel intercambio con mayor diferencia de peso entre el nodo eliminado y aquel o la suma de aquellos que lo sustituyen.

3.2. Iterated greedy

La metaheurística *Iterated Greedy* (IG) fue propuesta por R. Ruiz y T.Stüzle, por primera vez en [32], como método que combinaba la intensificación y diversificación de la solución mediante un proceso de construcción y destrucción de la solución. De esta forma, se puede extender aún más el espacio de búsqueda, explorando un mayor número de vecindades, con una alta probabilidad de encontrar otros óptimos locales.

Existen varias propuestas a la hora de aplicar IG dentro del algoritmo: o bien se aplica sobre la mejor solución encontrada durante el proceso de construcción con GRASP, o bien sobre cada una de las generadas con GRASP. Debido al alto coste computacional de esta última opción, se ha decidido optar por la primera, utilizando así la mejor solución generada durante la construcción GRASP como entrada para el *iterated greedy*.

El algoritmo se compone de una primera fase de destrucción, en la que se eliminan un número de elementos de la solución, determinado por un parámetro porcentual de entrada β, cuyo valor ha sido determinado experimentalmente en la sección 5.1. De esta forma, se llega a una solución parcial no factible, que posteriormente es reconstruida para recuperar esa factibilidad perdida. Además, se puede incluir una fase extra de optimización local de la solución reconstruida. En este trabajo, ya que la búsqueda local propuesta anteriormente es poco costosa computacionalmente, se ha decidido incluir tanto la purga como la búsqueda local sobre dicha solución. Estos cuatro pasos se ejecutarán de forma iterativa mientras se mejore la solución o hasta que se superen un número máximo de iteraciones sin mejora, determinado por el parámetro θ, también establecido experimentalmente en la sección 5.1.

El algoritmo 3 muestra el pseudocódigo del algoritmo de IG propuesto. Los parámetros de entrada del algoritmo son θ y β, especificados anteriormente, y la solución S, siendo la solución de la que parte IG, que a su vez es la mejor solución construida utilizando la metaheurística GRASP explicada en la sección 3.1. Inicialmente, se establece la solución S como la mejor solución encontrada hasta el momento, representada en el pseudocódigo como S_{best} (paso 1). Posteriormente, se inicializa la variable i a 0, que reflejará el número de iteraciones sin mejora realizadas (paso 2). El algoritmo iterará siempre y cuando las iteraciones realizadas sin mejora sean menores al parámetro θ (pasos 3 - 14). En cada iteración, se realizará, sobre la mejor solución, un proceso de destrucción (paso 4), dando lugar a la solución S'. Sobre ella, se realizará la fase de reconstrucción (paso 5), generando así una solución factible S''. Para asegurar que se trata de un óptimo local, se realizarán la purga presentada en la sección 3.1.2 (paso 6) y la búsqueda local presentada en la 3.1.3 (paso 7) sobre S'', dando lugar a la solución S''''. Si se mejora la solución S_{best} encontrada previamente (paso 8), se actualizará la mejor solución (paso 9) y se reiniciará el contador de iteraciones sin mejora (paso 10).

En caso contrario, se incrementará el contador de iteraciones sin mejora (paso 12) y se reiniciará el bucle. Finalemente, se devolverá la mejor solución encontrada con el algoritmo *iterative greedy* (paso 15).

Algoritmo 3 `IteratedGreedy`(θ, β, S)

1: $S_{best} \leftarrow S$
2: $i = 0$
3: **while** $i < \theta$ **do**
4: $S' \leftarrow Destruccion(S_{best}, \beta)$
5: $S'' \leftarrow Reconstruccion(S')$
6: $S''' \leftarrow Purga(S'')$
7: $S'''' \leftarrow BusquedaLocal(S''')$
8: **if** $MWDSP(S'''') > MWDSP(S_{best})$ **then**
9: $S_{best} \leftarrow S''''$
10: $i = 0$
11: **else**
12: $i = i + 1$
13: **end if**
14: **end while**
15: **return** S_{best}

Habiendo presentado el esquema algorítmico de IG, se debe de explicar en profundidad las fases de destrucción y reconstrucción de la solución utilizadas. En el esquema clásico de *iterative greedy*, la destrucción se realiza escogiendo nodos de forma aleatoria, mientras que la construcción se realiza siguiendo un esquema voraz para recuperar la factibilidad de la solución.

Sin embargo, en este trabajo se propone tanto este planteamiento clásico como otro que se aleja de él. Este segundo se compone de una destrucción siguiendo un criterio voraz y una reconstrucción utilizando la metaheurística GRASP (sección 3.1.1). De esta forma, es posible analizar el impacto que tiene la intensificación y diversificación en MWDSP.

El proceso de destrucción aleatorio se realiza seleccionando, de forma iterativa, elementos al azar de la solución, eliminando un total de $\beta \cdot k$ elementos. Para la reconstrucción, se utiliza el criterio voraz presentado en la ecuación 3.1.

En el caso de la destrucción voraz, se eliminará iterativamente el nodo v_r que menos contribuye a la función objetivo, es decir, que tenga un mayor peso y cuyo número de conexiones a otros nodos sea menor, hasta haber eliminado $\beta \cdot k$ elementos. Matemáticamente:

$$v_r = \arg\min_{v' \in S} \frac{|N(v')|}{w_{v'}} \tag{3.4}$$

3.3. Estructura del algoritmo final

A continuación, se refleja, con el objetivo de definir el algoritmo final en su totalidad, un diagrama que muestra la construcción total de la mejor solución encontrada.

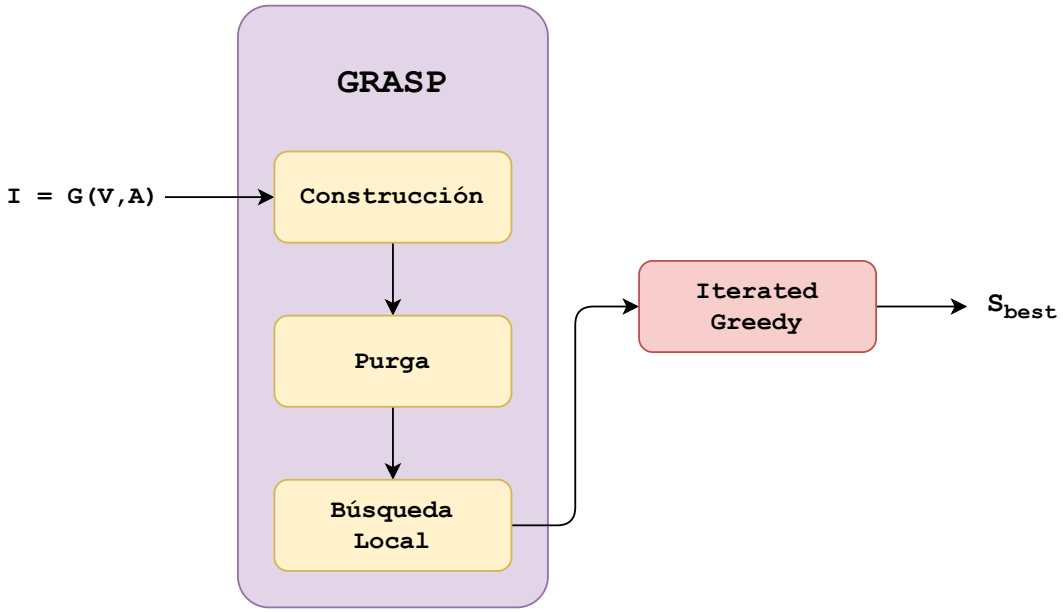


Figura 3.1: Diagrama de flujo del algoritmo propuesto

El algoritmo, representado en la figura 3.1, comienza con la instancia, compuesta por el grafo $G(V, A)$, sobre la que se realiza la fase de la construcción de la solución. Con una solución factible ya creada, se puede decidir si realizar la fase de la purga descrita en 3.1.2 y/o la fase de búsqueda local de la sección 3.1.3. Tras estas fases, finaliza la metaheurística GRASP, y se encuentra la última decisión, ejecutando o no *iterated greedy* como post-proceso de la solución formada en GRASP, generando, en ambos casos, la solución de salida S.

4

Descripción informática

En este capítulo, se explican las diferentes metodologías utilizadas, las representación de las estructuras principales de la instancia y de la solución y la organización, diseño e implementación a nivel de programación de los algoritmos descritos en el capítulo 3. El código ha sido desarrollada en Java 19, siguiendo el paradigma de programación orientada objetos. Éste es muy adecuado para la representación de problemas de optimización, permitiendo modelar clases diferentes para cada elemento del algoritmo, haciendo uso de la herencia y polimorfismo para modificar la configuración del mismo sin realizar cambios profundos sobre el código desarrollado.

4.1. Metodología y Herramientas

Para el desarrollo de este Trabajo de Fin de Grado, se ha hecho uno de metodologías ágiles (SCRUM) con el fin de organizar el trabajo y desarrollar de forma iterativa el proyecto. Se realizaron un total de 7 sprints de dos semanas cada uno, añadiendo valor al software implementado. Se organizaron reuniones al final de cada uno de los sprints con los tutores, con la finalidad de establecer cuáles eran las metas del sprint siguiente, similares a la ceremonia ágil *Sprint planning*.

Para almacenar las historias de usuario (US, del inglés *User Stories*) creadas en cada una de las reuniones de planificación, se hizo uso de un tablero de Trello con las columnas *To Do*, *Doing* y *Done* clásicas de los procesos ágiles. En la figura 4.1, se muestra un ejemplo del estado del tablero en uno de los sprints

intermedios del proyecto.

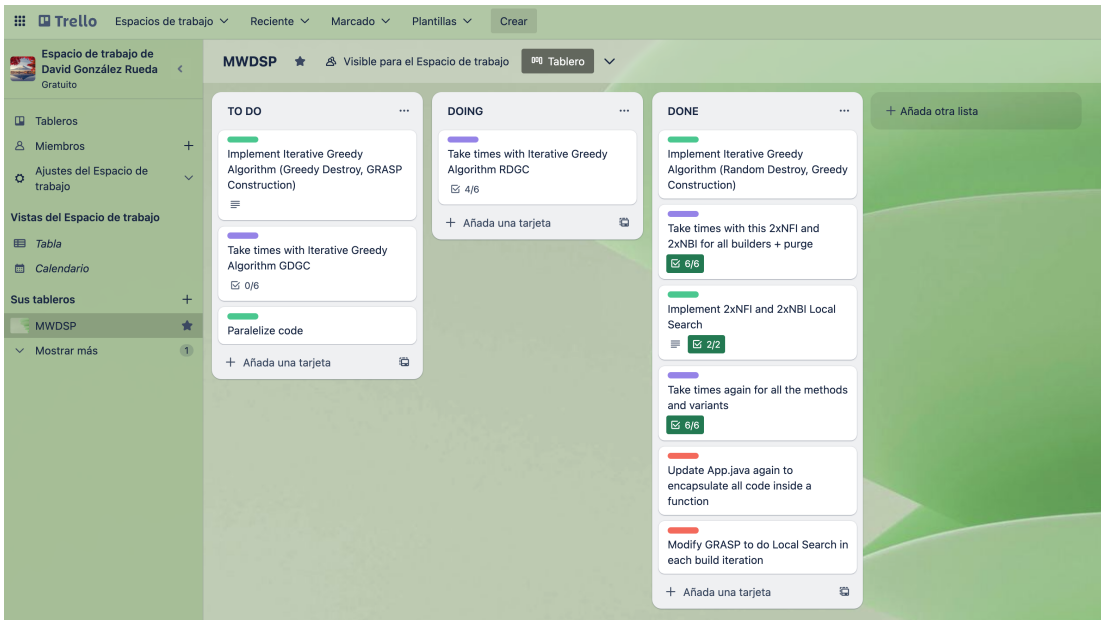


Figura 4.1: Estado del tablero ágil utilizado en uno de los sprints intermedios, con las columnas *To Do*, *Doing* y *Done*

El proceso de desarrollo del Trabajo de Fin de Grado se compuso de los siguientes pasos:

- Se generó el proyecto base y el repositorio de GitHub donde sería almacenado todo el código. Adicionalmente, se generó la clase `Instance`, junto con un método para parsear el subconjunto de instancias seleccionado que formaría el conjunto preliminar utilizado para realizar los experimentos.
- Se desarrolló la clase `Solution` y la creación de un constructivo de soluciones completamente aleatorio como punto de partida. Además, se crean las tablas de tiempos que almacenarían los resultados intermedios obtenidos durante el desarrollo.
- Se implementaron tanto el constructivo voraz como el constructivo GRASP, así como la fase de purga intermedia explicada en el capítulo anterior. Inmediatamente después, se creó la primera aproximación de búsqueda local `1xN`. Los conocimientos necesarios para la implementación de GRASP y la búsqueda local fueron adoptados mediante diversas reuniones teóricas más extensas con los tutores.
- Se realizaron las búsquedas locales `2xN` (tanto FI como BI), de tal forma que, debido a la implementación en el código, se posibilitó la creación de búsquedas locales `NxN`, explicado en la sección 4.3.3.

- Se creó una clase `Algorithm` como interfaz que contendría todos los métodos pertinentes para la ejecución del algoritmo de forma sencilla. Junto a ello, se crearon ambos planteamientos de *Iterative Greedy* mencionados en el capítulo anterior.
- Finalmente, se refactorizó y comentó el código para facilitar su legibilidad, comprensibilidad y uso, además de desarrollar otras herramientas (explicadas en el anexo A) como:
  - Creaciones de logs en los constructivos para conocer exactamente los pasos ejecutados y el valor de la solución en cada uno de ellos.
  - Creación de una clase pseudorandomizada para evitar diferentes resultados en diferentes ejecuciones debido a la semilla. (`CustomRandom`).
  - Creación de ficheros con los resultados obtenidos para facilitar la toma de tiempos.
  - Creación de ficheros de constantes (`Constants`) para almacenar datos fijos del código.

Una vez implementadas toda la funcionalidad planteada junto a los tutores, se comenzaron los experimentos necesarios con el fin de establecer los mejores parámetros de entrada, así como las mejores implementaciones de los constructivos, búsquedas locales e *iterative greedy*. Estos experimentos y sus resultados podrán ser comprobados en la sección 5.1. Por último, se ha finalizado el Trabajo de Fin de Grado con la creación de este documento, con un primer borrador, un proceso de varias correcciones por parte de los tutores, su correspondiente arreglo y una última revisión.

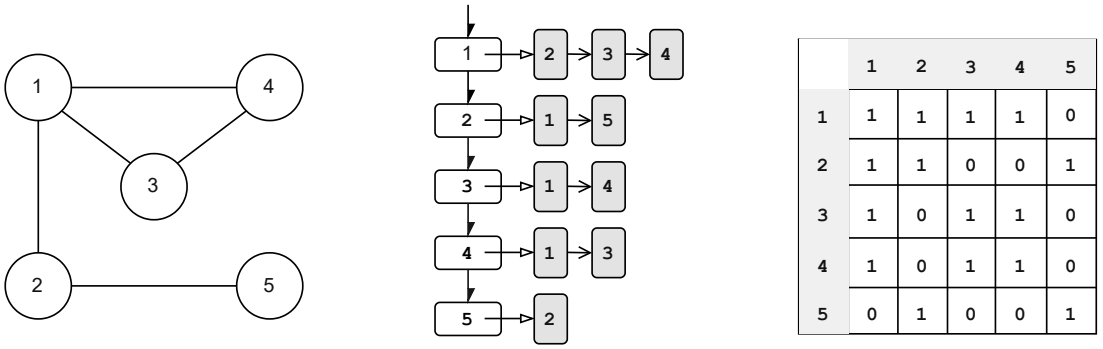
Se ha de mencionar que el uso de GitHub como plataforma para el control de versiones con la tecnología Git, junto a Visual Studio Code como principal herramienta de desarrollo, ha facilitado el control sobre el código. De esta forma, se podría recuperar el código generado en una versión anterior con el fin de realizar modificaciones, además de asegurar el almacenamiento de este en una plataforma segura, independiente del dispositivo en el que se realizara el desarrollo. El acceso al código realizado es público, con el fin de agilizar nuevos planteamientos sobre el problema MWDSP en el ámbito científico.

4.2. Representación de la instancia y solución

4.2.1. Representación de la instancia

Las instancias del problema MWDSP pueden ser representadas de múltiples formas diferentes, ya que están basadas directamente en la decisión de la representación del grafo que la compone. Si bien es cierto que existen dos formas muy

extendidas de realizar dicha representación, siendo estas la matriz y la lista de adyacencia, se han de tener en cuenta las diferentes operaciones e instrucciones que se ejecutarán posteriormente en el algoritmo sobre estas estructuras, ya que pueden tener un importante impacto en el tiempo de ejecución final.



	1	2	3	4	5
1	1	1	1	1	0
2	1	1	0	0	1
3	1	0	1	1	0
4	1	0	1	1	0
5	0	1	0	0	1

Figura 4.2: Instancia del problema MWDSP y su representación como lista de adyacencia y como matriz de adyacencia, de izquierda a derecha, respectivamente

Para representar las conexiones de cada uno de los nodos del grafo, se propone el uso de una lista de adyacencia. Las principales ventajas de escoger, en el contexto del problema MWDSP, la lista de adyacencia sobre la matriz de adyacencia son las siguientes:

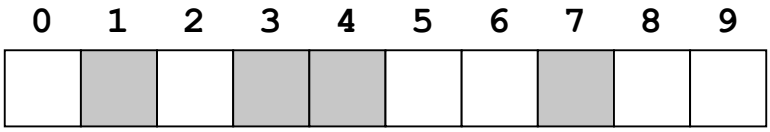
1. Para recuperar las conexiones de cada uno de los nodos, la lista de adyacencia tiene una complejidad $O(1)$, ya que únicamente se debe de acceder a la posición del nodo deseado, que tendrá un puntero a la lista de conexiones. Por otra parte, en la matriz de adyacencia, se debería de recorrer la fila o columna del nodo, comprobando si tiene una conexión (es decir, que tiene valor 1) o no la tiene (0) con el resto de los nodos (complejidad $O(n)$). Es importante mencionar que nunca se accede a una conexión individual del nodo, sino a la totalidad de ellas.

2. La creación y modificación de las listas de adyacencia son más costosas en tiempo que en la matriz de adyacencia. Sin embargo, estas estructuras son únicamente utilizadas para lectura a lo largo de la ejecución del algoritmo, donde las listas de adyacencia son superiores. Además, cabe mencionar que la creación de la instancia no computa en el tiempo final de ejecución del algoritmo.

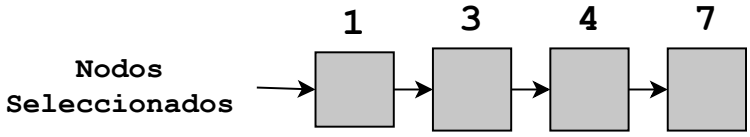
3. Se debe prestar especial atención a las aplicaciones del problema estudiado, que tienden a ser de gran complejidad y tamaño. Las matrices de adyacencia pueden ser computacionalmente costosas en estos casos, ocupando un gran espacio en memoria y repercutiendo en la ejecución del algoritmo.

4.2.2. Representación de la solución

La pieza de información más relevante de una solución de una instancia MWDSP es qué nodos han sido escogidos. Inicialmente, teniendo presente que la instancia se compone de k nodos, se puede considerar que la forma más sencilla de almacenar los nodos pertenecientes a la solución es un `Array` de k posiciones que almacena un 1 si el nodo se ha escogido y un 0 en caso contrario (figura 4.3(a)). Otra alternativa sería una lista que almacenara únicamente aquellos nodos seleccionados (figura 4.3(b)). Ambas fueron implementadas al comienzo del desarrollo del proyecto.



(a) Representación de los nodos escogidos, en color gris, en una instancia de 10 nodos como un `Array`.



(b) Representación de los nodos escogidos en una instancia de 10 nodos como una lista.

Figura 4.3: Representación de la solución como Array (4.3(a)) o como lista (4.3(b))

Sin embargo, al igual que las matrices de adyacencia como representación del grafo en la instancia, los `Array` contienen mucha información redundante (se conoce que un nodo no está escogido si no pertenece al conjunto de aquellos sí escogidos) y pueden ser computacionalmente costosas en la memoria en instancias de gran complejidad y tamaño. En el caso de las listas de adyacencia, añadir nuevos nodos puede tener complejidad $O(n)$ en caso de tener que ampliar el `Array` interno que la compone o de comprobar si un nodo se encuentra o no en la solución.

Debido a ello, se propone la estructura `HashSet`[1] de Java, descartando las otras dos. Se trata de una implementación de un `Set`, que almacena elementos no repetidos en cualquier orden. La principal ventaja de utilizarla es que las operaciones básicas necesarias para la ejecución del algoritmo (añadir nodos, quitarlos y saber si un nodo pertenece a la solución) tienen complejidad constante $O(1)$ [33].

[1] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/HashSet.html

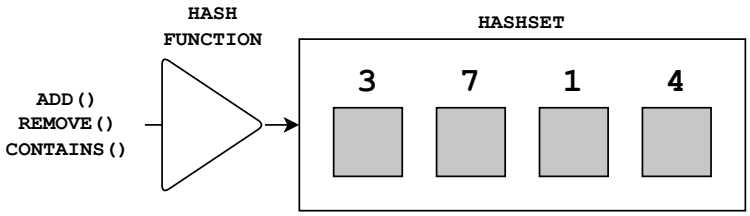


Figura 4.4: Representación de los nodos escogidos en una instancia de 10 nodos como un `HashSet`.

4.3. Diseño e implementación

En esta sección se describen todas las decisiones de diseño tomadas durante el desarrollo del proyecto, y cómo ha sido estructurado el código en las distintas clases, su función y la relación que existen entre ellas. Para facilitar esta descripción se harán uso de diagramas UML que permitirán sintetizar esta información gráficamente. Se fragmentará el diagrama general con el fin de realizar el análisis detallado de cada una de las secciones.

4.3.1. Clases principales

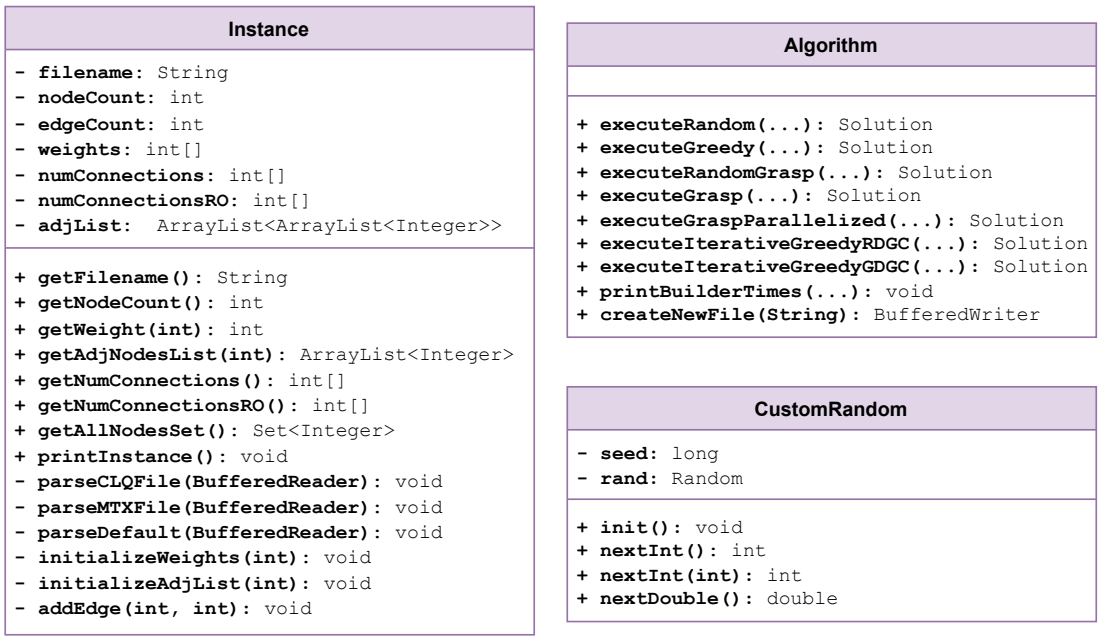


Figura 4.5: Diagrama UML de las clases principales no relacionadas (I). Clase `Instance`, `Algorithm` y `CustomRandom`

Inicialmente, se han de explicar aquellas clases fundamentales para la estructura del problema. Aquí se incluyen las encargadas de representar una instancia, una solución, recoger la funcionalidad principal y gestionar la aleatoriedad.

Para modelar la instancia del problema MWDSP, se ha diseñado la clase `Instance`, mostrada en la figura 4.5. Esta se encarga de almacenar la información relativa a la instancia, como el número total de nodos y aristas, los pesos de cada uno de los nodos (`weights`), el número de conexiones de cada nodo a otros y la representación del grafo como lista de adyacencia, `adjList`, presentada anteriormente en la sección 4.2.1.

Seguidamente, la clase que encapsula la lógica principal de la aplicación es la clase `Algorithm`. Esta clase no dispone de variables internas, sino que contiene las ejecuciones de los diferentes métodos constructivos (aleatorio, voraz, GRASP con α aleatorio y GRASP con α definido). Además, se implementó una paralelización del constructivo GRASP, explicada en detalle en la sección 4.3.2, cuyos resultados en tiempo fueron superiores al GRASP serializado (pueden ser comprobados en la sección 5.1). Adicionalmente, capturan la lógica de las metaheurísticas *Iterated Greedy* presentadas en la sección 3.2, tanto de destrucción aleatoria y reconstrucción voraz (`RDGC`), como de destrucción voraz y reconstrucción GRASP (`GDGC`).

Por otra parte, la gestión de la aleatoriedad es imprescindible para el diseño de algoritmos de optimización, ya que se requiere que, para la comprobación de los resultados y la toma de tiempos, se pueda asegurar la reproducibilidad de los resultados. De esta forma, se implementa una clase `CustomRandom`, que contiene un objeto `Random` de Java, y una semilla fija que asegura que, aun ejecutando el código múltiples veces, se mantengan los mismos resultados. Por defecto, Java utiliza semillas diferentes en la inicialización de los objetos `Random` de Java, dificultado la experimentación al no poder reproducir los resultados. Esta clase se ha definido como una clase abstracta y estática, evitando la posibilidad de instanciar un objeto de la misma y obligando al usuario a utilizar el mismo generador de aleatorios, siguiendo un patrón *Singleton*, garantizando que se utiliza el mismo durante la ejecución.

Finalmente, la clase `Solution`, cuyo `UML` puede encontrarse en la figura 4.6, es utilizada para almacenar la información necesaria de la solución. A continuación se resume la funcionalidad de las variables más importantes:

- `totalWeight`: suma del peso total de los nodos seleccionados. Facilita la comprobación de una solución respecto a otra en diversos puntos del algoritmo, como la búsqueda local o *iterated greedy*.
- `numDomNodes`: número de nodos dominados de la instancia. Permite comprobar la factibilidad de una solución de forma sencilla, ya que lo será siempre que sea igual al número total de nodos de la instancia (`totalNodeCount`).
- `domNodes`: número de nodos que dominan a cada uno de los nodos. O lo que

es lo mismo, para cada uno de los nodos, número de sus nodos adyacentes que han sido seleccionados.

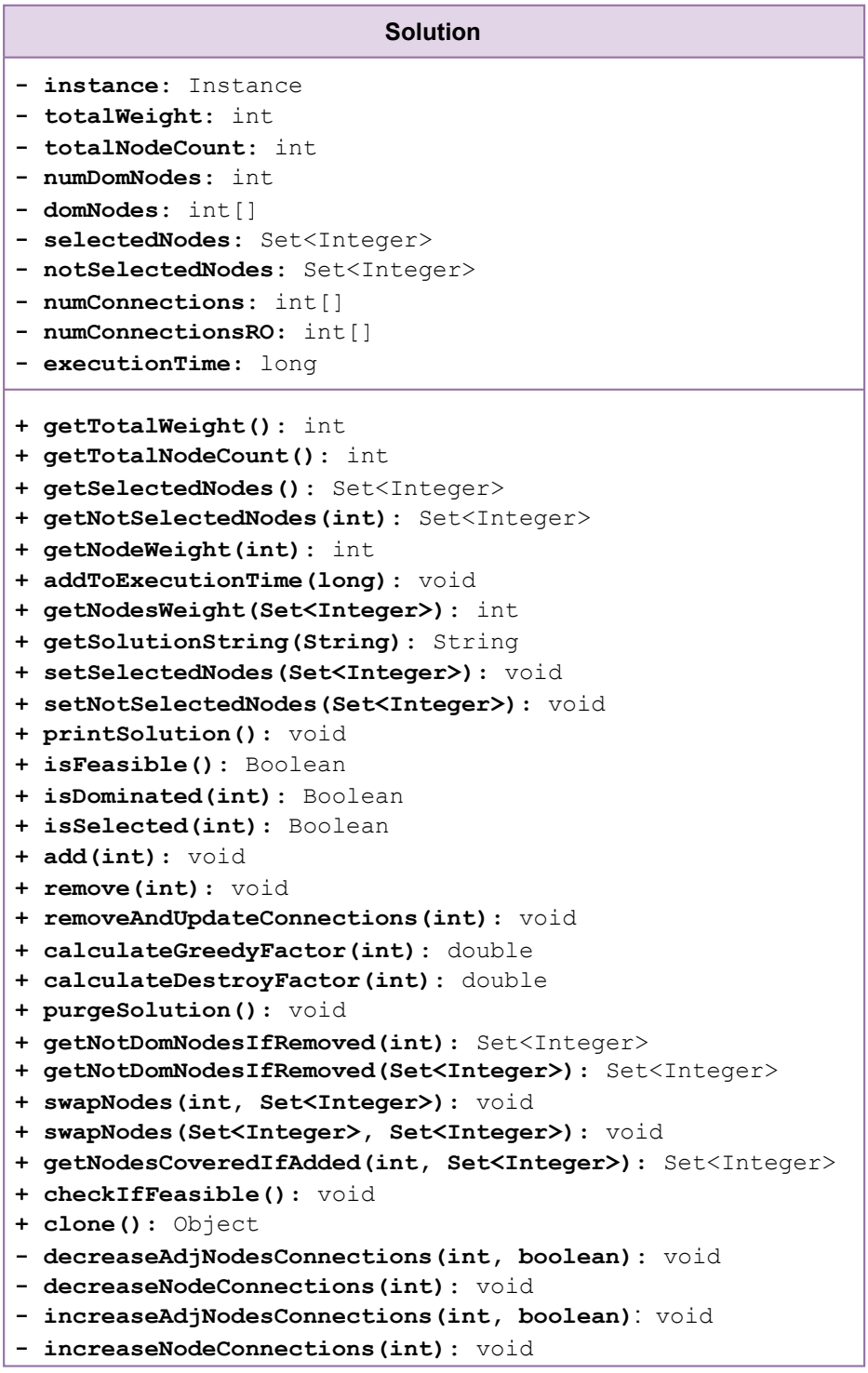


Figura 4.6: Diagrama UML de las clases principales no relacionadas (II). Clase `Solution`

- `selectedNodes, notSelectedNodes`: como se ha explicado anteriormente en la sección 4.2.2, `HashSet` de nodos seleccionados y no seleccionados, respectivamente.

- `numConnections`: para cada uno de los nodos, número de conexiones a nodos no dominados. Se actualizará cada vez que se añada o se retire un nodo de la solución.

4.3.2. Constructivos

Una vez explicadas las clases principales sobre la que se apoya la mayoría de la lógica y estructura del algoritmo, se muestran las clases utilizadas para implementar la construcción de las soluciones. La figura 4.7 muestra la interfaz y clases utilizadas para implementar el procedimiento constructivo descrito en la sección 3.1.1, así como un constructivo aleatorio y voraz.

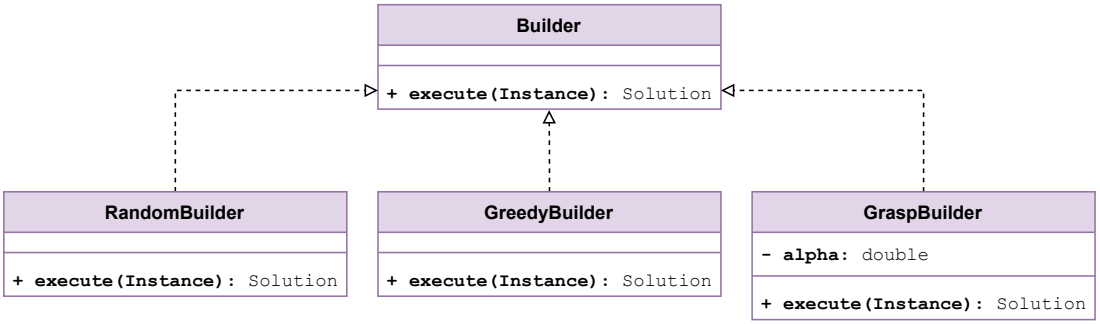


Figura 4.7: Diagrama UML de la interfaz `Builder` y las clases que la implementan

La interfaz `Builder` precisa el modo en el que se deberán de construir las soluciones. Toda clase constructiva deberá de implementar esta interfaz, que únicamente tiene un método `execute` encargado de recibir una instancia y devolver una solución. Inicialmente, durante el proceso de desarrollo, se implementaron un constructivo aleatorio `RandomBuilder`, que iterativamente añade elementos a la solución hasta que esta es factible, y un constructivo voraz `GreedyBuilder`, que añade el mejor nodo en cada iteración siguiendo la fórmula presentada en la ecuación 3.1. Sin embargo, se propuso finalmente el constructivo `GraspBuilder`, clase que representa el algoritmo presentado en la sección 3.1.1, con mejores resultados que los dos anteriores (mostrados en la sección 5.1).

Para incrementar el rendimiento del constructivo GRASP, una vez demostrados los mejores resultados frente a los otros dos constructivos, se implementó una paralelización con las herramientas proporcionadas por Java. En la figura 4.8 se muestra un diagrama UML con la implementación de dicha paralelización.

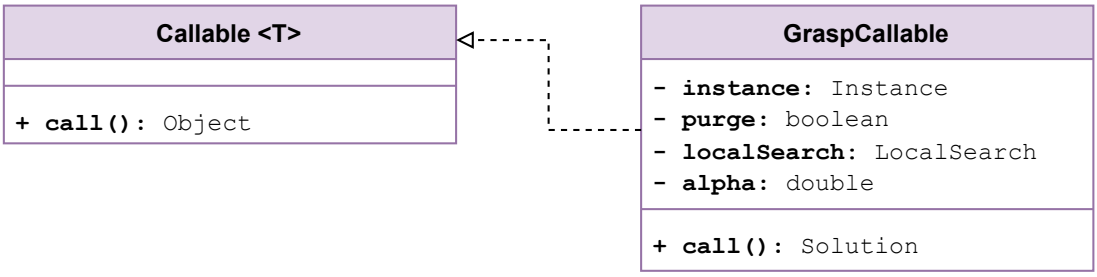


Figura 4.8: Diagrama UML de la implementación de la paralelización del constructivo GRASP.

La interfaz `Callable`[2] es una interfaz de Java que permite que un hilo, a diferencia de la interfaz `Runnable`, ejecute una lógica y devuelva un resultado. Ya que el constructivo GRASP es ejecutado 100 veces, se crea un *pool* de 100 hilos. Un *pool* es un conjunto de hilos que se crean inicialmente y a los cuales se les pueden asignar tareas. Cuando una nueva tarea debe ejecutarse en paralelo, en lugar de crear un nuevo hilo cada vez, se toma un hilo del *pool* existente y se le asigna la tarea. Una vez que el hilo ha completado su tarea, puede regresar al *pool* y estar disponible para ejecutar otra. Esto reduce el coste de crear y destruir hilos continuamente. La creación del *pool* se realiza mediante el método estático `Executors.newFixedThreadPool` (siendo `Executors` una clase utilitaria de Java que permite administrar la creación y ejecución de tareas en hilos de manera más controlada), y a cada uno de ellos se le asigna el `GraspCallable` (que ejecuta la construcción, purga y búsqueda local propias de GRASP). Finalmente, se recogen dichos resultados y se devuelve únicamente el mejor de todos ellos.

4.3.3. Búsqueda local

Finalizada la explicación de los constructivos, la figura 4.9 muestra la estructuración de las clases que representan las búsquedas locales detalladas en la sección 3.1.3, y dos búsquedas locales adicionales explicadas a continuación.

La interfaz `LocalSearch` define el modelo para construir cualquier búsqueda local. Al igual que la interfaz de los constructivos, únicamente se compone de un método `execute`, donde recibe la solución construida y opcionalmente purgada y encuentra un óptimo local. Las clases `LocalSearch1xNFI` y `LocalSearch1xNBI` implementan el algoritmo mostrado en la sección 3.1.3.

[2]https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Callable.html

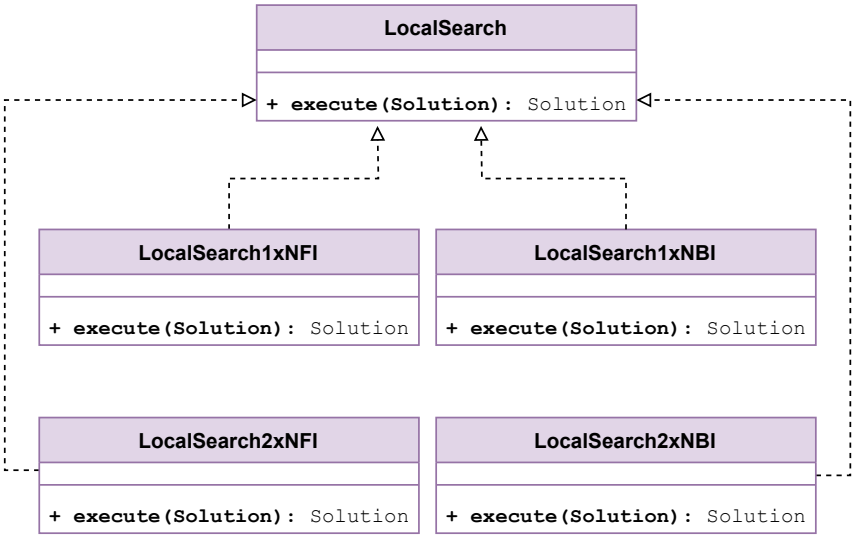


Figura 4.9: Diagrama UML de la interfaz `Local Search` y las clases que la implementan.

Las clases `LocalSearch2xNFI` y `LocalSearch2xNBI` implementan otras búsquedas locales desarrolladas en el proyecto con el objetivo de encontrar mejores soluciones que aquellas dadas por las búsqueda local `1xN`. Los intercambios en las búsquedas locales de MWDSP podrán darse únicamente bajo dos condiciones: los nodos descubiertos (no dominados) por la eliminación del nodo o nodos candidatos están cubiertos por los nodos por los que será intercambiado, y los peso del nodo o nodos que serán eliminados no puede sobrepasar el peso de los nodos por los que serán intercambiados (si no, sería una peor solución).

En las búsquedas locales `1xN`, los nodos no dominados son fáciles de identificar, ya que sólo se ha de comprobar si alguno de sus nodos adyacentes quedará descubierto si se retira. Sin embargo, en las búsquedas donde más de un nodo es eliminado, el cálculo de los nodos descubiertos puede ser más complejo. Dado el siguiente ejemplo:

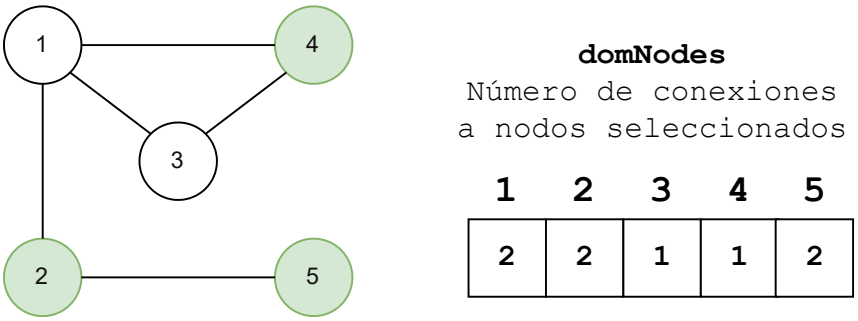


Figura 4.10: Posible solución de un sistema sobre el que se aplicará una búsqueda local `2xN`. Los nodos seleccionados se encuentran sombreados de verde.

Si se retirasen los nodos 2 y 4 de la solución, los nodos 1, 3 y 4 estarían descubiertos. Sin embargo, si se comprueba cada uno de los nodos de forma individual, y sin tener en cuenta los demás, ocurriría lo siguiente:

- **Nodo 2**: podría ser eliminado sin descubrir ningún nodo, ya que el nodo 4 y el nodo 5 dominarían el resto del grafo.
- **Nodo 4**: podría ser eliminado y sólo quedarían sin dominar el nodo 3 y el 4 (ya que no se tiene en cuenta que el 2 también está siendo eliminado, dejando descubierto el nodo 1).

Para evitarlo, se ha implementado el método `getNotDomNodesIfRemoved` en la clase `Solution` que, dado un `Set` de n nodos y mediante el uso de un `HashMap` de Java, detecta el número de conexiones a nodos seleccionados que serán eliminadas de cada uno de los nodos, para así obtener los nodos descubiertos de una forma global. Debido a ello, las búsquedas locales `2xN` pueden ser escalables a búsquedas locales `NxN`, modificando únicamente el número de nodos que quieran retirarse de la solución.

4.3.4. Iterated Greedy

Finalmente, las siguientes clases fueron utilizadas para la implementación de los dos planteamientos de la metaheurística *Iterated greedy*, presentados anteriormente en la sección 3.2.

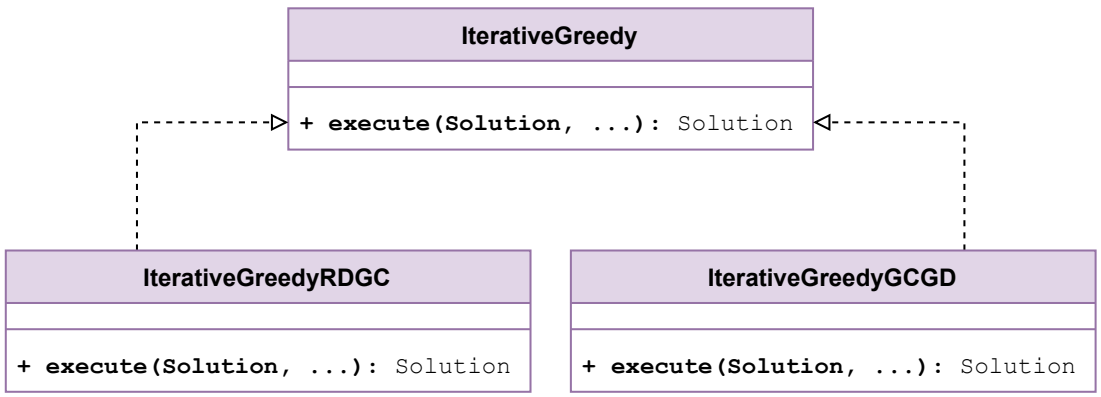


Figura 4.11: Diagrama UML de la interfaz `Iterated Greedy` y las clases que la implementan.

Al igual que las interfaces `Builder` y `LocalSearch`, la interfaz `IterativeGreedy` únicamente se compone de un método `execute`, que recibirá la solución construida con GRASP de las etapas anteriores y devolverá la solución obtenida

de la destrucción y reconstrucción iterativas propias de la metaheurística. La clase `IterativeGreedyRDGC` ejecuta una destrucción aleatoria con una reconstrucción voraz (*Random destruction, Greedy Construction*), mientras que la clase `IterativeGreedyGDGC` ejecuta una destrucción voraz con una reconstrucción GRASP (*Greedy Destruction, Grasp Construction*).

5

Experimentación

Este capítulo tiene dos objetivos: establecer cuales son los mejores parámetros de entrada del algoritmo y comparar el rendimiento del algoritmo propuesto frente al estado del arte actual. Con el fin de conseguir ambos, se divide esta sección en dos fases: los experimentos preliminares y los finales.

El algoritmo ha sido desarrollado con la tecnología Java 19, y los experimentos han sido ejecutados en un Apple MacBook Pro, de procesador Quad-Core Intel Core i5 (2GHz), con 16 GB de RAM (3733 MHz LPDDR).

El conjunto de instancias utilizado para realizar las comparaciones es aquel presentado en [23], siendo este el que presenta los mejores resultados encontrados en la literatura, pudiendo realizar así una comparativa justa. Se constituye de 540 instancias, agrupadas a su vez en instancias SMPI, que van de 50 a 250 vértices y 50 a 5000 aristas, y 210 instancias LPI más complejas, con de 300 a 1000 vértices, y hasta 20000 aristas. Los pesos de estas instancias han sido distribuidos de forma uniforme en el intervalo [20, 70]. Estas instancias se encuentran bajo una carpeta denominada `T1`, y disponibles públicamente en el siguiente enlace.

Para evaluar los experimentos, se utilizan las siguientes estadísticas: Promedio, media de la función objetivo; T(s), tiempo de ejecución en segundos; Desv.(%), desviación porcentual media respecto al mejor resultado del experimento encontrado; y Best(#), número de las mejores soluciones encontradas. Los mejores valores encontrados de cada columna serán marcados en negrita en cada uno de los experimentos.

5.1. Experimentos preliminares

La finalidad de los experimentos preliminares es establecer el mejor valor de los parámetros de entrada y variaciones de constructivos, búsquedas locales, *iterative greedy*, etc. Se utilizarán un subconjunto representativo de 24 de las 540 instancias, con el objetivo de no sobreajustar los parámetros.

Inicialmente, se debe de establecer el mejor valor de α_{GR} del constructivo GRASP. Para ello, se han evaluado los valores $\alpha_{GR} = \{0{,}25, 0{,}5, 0{,}75, RND\}$, siendo este último un valor aleatorio en cada una de las 100 construcciones iterativas que se han decidido realizar. La tabla 5.1 muestra los resultados obtenidos.

α_{GR}	Promedio	T(s)	Desv. (%)	Best (#)
0.25	1777.00	**0.05**	1.25	11
0.5	1900.33	0.07	9.71	1
0.75	2267.71	0.08	34.86	0
RND	**1764.50**	0.07	**1.02**	**15**

Tabla 5.1: Comparativa de los valores asociados a α_{GR} durante el constructivo GRASP.

Analizando los resultados, se puede considerar que la mejor opción en cuanto al promedio es aquella en la que $\alpha_{GR} = RND$, con una desviación promedio del 1.02 %. Si se toman en cuenta el resto de resultados, a mayor componente aleatoria del constructivo (incrementando el valor de α_{GR}), se obtienen peores valores del promedio. Si bien se podría establecer el valor de este parámetro aleatoriamente para el constructivo (y utilizarlo durante la totalidad del algoritmo), se estudiarán más adelante el efecto de las purgas y búsquedas locales sobre la función objetivo con los distintos valores de α_{GR} estudiados en este experimento.

El siguiente experimento tiene como finalidad comparar los resultados obtenidos del constructivo GRASP con $\alpha_{GR} = RND$, el aleatorio y el voraz. La tabla 5.2 refleja la superioridad del primero con respecto a los otros dos en términos de calidad de solución. Obtiene el mejor valor en todas las instancias del conjunto preliminar, con una diferencia de velocidad inapreciable entre los diferentes constructivos.

Constructivo	Promedio	T(s)	Desv. (%)	Best (#)
Aleatorio (100 it.)	4237.13	0.02	116.31	0
Voraz	1791.71	**0.00**	5.91	1
GRASP	**1764.50**	0.07	**0.00**	**24**

Tabla 5.2: Comparativa de los constructivos aleatorio, voraz y GRASP.

Como se ha mencionado anteriormente, el mejor valor de α_{GR} puede variar dependiendo de la búsqueda local utilizada y su impacto en el algoritmo. A continuación se muestran dos mapas de calor que muestran los resultados promedios de la función objetivo obtenidos (tabla 5.3) y el tiempo en segundos (tabla 5.4) de la combinación de los diferentes valores de α_{GR} con las búsquedas locales `1xNFI`, `1xNBI`, `2xNFI` y `2xNBI`.

α_{GR} \ LS	1xNFI	1xNBI	2xNFI	2xNBI
0.25	1699.79	1699.13	1720.33	1720.79
0.5	1730.67	1726.75	1790.67	1790.08
0.75	1782.92	1781.54	1862.71	1875.79
RND	1703.00	1702.46	1712.00	1712.00

Tabla 5.3: Mapa de calor de los promedios de la función objetivo obtenidos considerando diferentes búsquedas locales (LS) y valores de α_{GR}.

α_{GR} \ LS	1xNFI	1xNBI	2xNFI	2xNBI
0.25	1.34	2.33	0.38	0.49
0.5	2.97	5.67	0.78	1.47
0.75	5.72	11.38	3.22	6.19
RND	3.52	7.31	1.79	3.59

Tabla 5.4: Mapa de calor de los tiempos de ejecución obtenidos considerando diferentes búsquedas locales (LS) y valores de α_{GR}.

Debido al gran volumen de datos mostrados, y para facilitar la lectura, se ha seguido un esquema de mapa de calor, en el que los colores verdes son asignados a los mejores resultados, los rojos a los peores, y los intermedios con gradientes a tonalidades amarillas. Analizando únicamente los promedios, se puede observar que el mejor resultado obtenido ha sido la combinación de $\alpha_{GR} = 0{,}25$ con la búsqueda local `1xNBI`. Si bien los resultados obtenidos de las búsquedas locales con $\alpha_{GR} = RND$ son cercanos al mejor mencionado anteriormente, se puede observar un empeoramiento de los resultados conforme se aumenta el valor de α_{GR}. Además, de media, las LS `2xN` son peores que las `1xN`. Por otra parte, los tiempos de ejecución son mejores en las `2xN`, obteniendo el mejor tiempo con la combinación de $\alpha_{GR} = 0{,}25$ con la búsqueda local `2xNFI`. Ya que se priorizan los resultados antes que los tiempos de ejecución, se considera $\alpha_{GR} = 0{,}25$ con la búsqueda local `1xNBI` como la mejor combinación, y ésta será utilizada en el algoritmo final.

Se comparan de nuevo todos los constructivos, ahora añadiendo las búsqueda local `1xNBI` al constructivo inicial, para comprobar si el mejor encontrado en el

experimento representado en la Tabla 5.2 se mantiene al añadir la búsqueda local. La tabla 5.5 muestra los resultados obtenidos.

Constructivo + `1xNBI`	Promedio	T(s)	Desv. (%)	Best (#)
Aleatorio (100 it.)	1821.58	6.78	4.21	7
Voraz	1739.83	**0.01**	4.22	3
GRASP	**1699.13**	2.33	**0.05**	**23**

Tabla 5.5: Comparativa de los constructivos con la búsqueda local `1xNBI`.

Como se muestra en la tabla anterior, el constructivo GRASP con $\alpha_{GR} = 0{,}25$ y LS `1xNBI` obtiene prácticamente la totalidad de mejores resultados frente a los otros constructivos con esta misma búsqueda local, reafirmando así que este constructivo es muy superior al aleatorio y voraz.

Por otra parte, se debe de comprobar si la purga tiene un efecto negativo o positivo sobre la combinación constructivo con búsqueda local. Para ello, se ha excluido la purga en la ejecución del algoritmo con el constructivo GRASP y la búsqueda local `1xNBI`. Los datos conseguidos se presentan en la tabla 5.6 a continuación.

GRASP + 1xNBI	Promedio	T(s)	Desv. (%)	Best (#)
Con purga	**1699.13**	**2.33**	**0.00**	**24**
Sin purga	1699.42	4.28	0.06	22

Tabla 5.6: Comparativa de GRASP + `1xNFI` con y sin purga.

Se puede observar que la utilización de la purga es de gran utilidad, tanto en promedio como en tiempo (ya que la búsqueda local no tiene que lidiar con extracciones de nodos redundantes en la solución, decreciendo así el número de iteraciones ejecutadas). Se decide, por tanto, que el constructivo GRASP utilizado de forma final en el algoritmo sea la combinación del constructivo con $\alpha_{GR} = 0{,}25$, el proceso de purga y la búsqueda local `1xNBI`.

Finalmente, se deben de configurar los parámetros para la metaheurística IG. Como se ha detallado en la sección 3.2, este procedimiento se aplica tras el constructivo, por lo que es necesario estudiar los mejores valores de β, θ y la mejor destrucción y reconstrucción de la solución. Inicialmente, se estudia el parámetro β, encargado de establecer el porcentaje de destrucción de la solución, con los valores $\beta = \{0{,}1, 0{,}2, 0{,}3, 0{,}4, 0{,}5\}$, utilizando tanto la destrucción aleatoria con reconstrucción voraz (`RDGC`, tabla 5.7) como la destrucción voraz con reconstrucción GRASP (`GDGC`, tabla 5.8). No se estudian valores superiores a 0.5, ya que se estaría destruyendo más de la mitad de la solución, lo que equivaldría a construir

una de cero en el contexto de algoritmos de optimización. Además, se fija un valor estático de $\theta = 5$, con el objetivo de que éste no influya en los resultados obtenidos.

β (RDGC)	Promedio	T(s)	Desv. (%)	Best (#)
0.1	1699.13	2.47	0.97	11
0.2	1699.13	2.44	0.97	11
0.3	1698.92	2.43	0.96	11
0.4	1698.21	2.41	0.88	12
0.5	**1696.42**	**2.39**	**0.75**	**15**

Tabla 5.7: Análisis del impacto sobre el algoritmo IG de diferentes valores de β utilizando `RDGC`.

β (GDGC)	Promedio	T(s)	Desv. (%)	Best (#)
0.1	1698.71	2.43	0.94	13
0.2	1698.83	2.48	0.95	11
0.3	1698.29	2.51	0.86	12
0.4	1695.67	2.42	**0.50**	13
0.5	**1695.21**	**2.41**	0.52	**15**

Tabla 5.8: Análisis del impacto sobre el algoritmo IG de diferentes valores de β utilizando `GDGC`.

En ambos casos, es evidente que a medida que el valor de β aumenta, los resultados mejoran, siendo el óptimo alcanzado con $\beta = 0{,}5$. El tiempo de ejecución se mantiene constante, siendo únicamente varias décimas más rápido ejecutar ambas parejas de destrucción-reconstrucción con $\beta = 0{,}5$. Con estos mismos datos, se estudia cuál es la mejor alternativa a la hora de la destrucción y reconstrucción, realizando un promedio de aquellos valores obtenidos de la tabla 5.7 (`RDGC`) y 5.8 (`GDGC`). Estos resultados se muestran a continuación en la tabla 5.9.

Método IG	Promedio	T(s)	Desv. (%)	Best (#)
RDGC	1698.36	2.43	0.91	12
GDGC	**1697.34**	**2.45**	**0.75**	**13**

Tabla 5.9: Comparación de los métodos de destrucción-reconstrucción del algoritmo IG.

Una vez determinado que el mejor método de destrucción-reconstrucción es `GDGC`, con una destrucción inicial de la solución del 50 % ($\beta = 0{,}5$), se han de

fijar el mejor número de iteraciones sin mejora ejecutadas, representado por el parámetro θ. A continuación, en la tabla 5.10, se reflejan los datos obtenidos con los valores de $\theta = \{5, 8, 12, 20, 40\}$.

Theta	Promedio	T(s)	Desv. (%)	Best (#)
5	1695.21	2.41	1.66	10
8	1693.63	**2.37**	1.31	11
12	1692.42	2.53	1.06	13
20	1692.17	2.58	1.04	14
40	**1691.50**	2.73	**0.83**	**16**

Tabla 5.10: Estudio de los diferentes valores de θ y su impacto sobre el algoritmo IG `GDGC`.

Como se puede observar, a mayor número de iteraciones sin mejora, mejor es la solución obtenida. Sin embargo, a partir de 40 iteraciones, la calidad de la solución es similar, y el tiempo de ejecución sufre un alto impacto, por lo que se decide tomar $\theta = 40$ como el mejor valor de este parámetro.

Habiendo finalizado el estudio de todos los parámetros y variantes posibles del algoritmo, se decide analizar si la paralelización del mismo proporcionaría una calidad de soluciones similar a la obtenida con la variante más prometedora de forma secuencial, pero en un tiempo de ejecución menor. Los resultados obtenidos han sido reflejados en tabla 5.11.

Variante	Promedio	T(s)	Desv. (%)	Best (#)
Secuencial	**1691.42**	2.75	**0.31**	**18**
Paralelizada	1697.21	**1.22**	0.70	14

Tabla 5.11: Comparativa del algoritmo ejecutado secuencialmente y de forma paralelizada.

La paralelización del código resulta en una ejecución hasta 2.25 veces más rápida del algoritmo completo. Sin embargo, debido a la aleatoriedad propia de los *threads*, no se pueden conseguir resultados consistentes, ya que varían según el intercalado de las instrucciones de cada uno de los 100 constructivos GRASP en el procesador.

Resumiendo todos los resultados obtenidos durante los experimentos preliminares, se puede afirmar que la mejor variante obtenida es un constructivo GRASP con $\alpha_{GR} = 0{,}25$, con purga y búsqueda local `1xNBI`, al que se le aplica la metaheurística IG con destrucción voraz del 50 % de la solución obtenida anteriormente ($\beta = 0{,}5$) y reconstrucción GRASP, que ejecutará un total de 40 iteraciones

sin mejora ($\theta = 40$), tras las cuales devolverá la mejor solución encontrada. La tabla 5.12 muestra la contribución de los diferentes métodos aplicados sobre el constructivo inicial GRASP.

Algoritmo	Promedio	T(s)	Desv. (%)	Best (#)
GRASP	1777.00	**0.05**	5.07	3
GRASP + LS	1699.13	2.33	1.24	13
GRASP + LS + IG	**1691.42**	2.75	**0.00**	**24**

Tabla 5.12: Contribución de los diferentes componentes sobre el algoritmo final.

5.2. Experimento final

Como se indica en el resumen de este capítulo, este experimento final tiene como finalidad comparar el rendimiento del algoritmo desarrollado con los mejores resultados obtenidos en la literatura, siendo estos obtenidos con el algoritmo de búsqueda tabú híbrida con un *resolver* IP (`HTS-DS`), propuesto en [23] y descrito en la sección 1.3. Se realizará este experimento sobre todas las instancias T1, a diferencia de los experimentos preliminares que únicamente ejecutaban sobre un subconjunto descriptivo de T1. La tabla 5.13 muestra el resultado obtenido al comparar el algoritmo propuesto en este TFG frente a `HTS-DS`, donde el mejor resultado de cada métrica es resaltado en negrita.

Algoritmo	Promedio	T(s)	Desv. (%)	Best (#)
$\text{GRASP}_{\text{1xNBI + IG. GDGC}}$	1619.09	**4.06**	3.98	5
HTS-DS	**1540.23**	6.18	**0.11**	**48**

Tabla 5.13: Comparativa del algoritmo propuesto en este TFG y el estado del arte.

En este experimento final, se puede observar que la calidad general de las soluciones obtenidas del estado del arte es superior a la propuesta. Sin embargo, únicamente dista un 4 % de esta en promedio, y es 1.52 veces más rápida que aquellas obtenidas mediante `HTS-DS`. Esta diferencia en tiempo podría ser utilizada para la implementación y ejecución de nuevas técnicas y metaheurísticas sobre el algoritmo descrito en este trabajo, pudiendo conseguir mejores resultados que los actuales.

6

Conclusiones y trabajos futuros

La realización de este TFG ha permitido descubrir, afianzar y experimentar nuevos conocimientos del sector de investigación de los problemas algorítmicos que se presentan, de forma abstracta, en el mundo que nos rodea. Se han aprendido metaheurísticas como *Greedy Randomized Adaptive Search Procedure* e *Iterated Greedy*, no impartidas en el grado de Diseño de Desarrollo de Videojuegos o Ingeniería de Computadores, además de consolidar otros esquemas algorítmicos y estructuras de datos sí conocidas, como los algoritmos voraces, los *sets*, *colas* o *pilas*. De esta forma, todos los objetivos presentados en el capítulo 2 han sido cumplidos de forma satisfactoria, proporcionando un extenso catálogo de herramientas que podrán ser utilizadas tanto de forma personal como profesional. La creación de clases de manera modular, el uso correcto de las estructuras de datos y el lenguaje Java para la optimización de la complejidad del código, la documentación del código y la adopción de metodologías ágiles; todo ello ha contribuido de forma positiva a la mejora en las habilidades de programación y trabajo, fomentando aquellas cualidades requeridas en el grado.

Por otra parte, se puede aseverar que se ha cumplido el objetivo principal, que consistía en proponer, diseñar y desarrollar un algoritmo que, haciendo uso de metaheurísticas, pudiera ofrecer unas soluciones de alta calidad en un tiempo de ejecución reducido, que fueran competentes con el estado del arte. Si bien los promedios no han superado a aquellos propuestos anteriormente, el tiempo de ejecución del algoritmo es más reducido, dando lugar a unas soluciones de buena calidad (con poca desviación).

Respecto a las futuras líneas de experimentación del problema MWDSP, se

podría implementar una búsqueda tabú (*Tabu Search, TS*, en inglés), aumentando el espacio de búsqueda de soluciones. Otra posible propuesta sería tratar de estabilizar los resultados obtenidos a través de la paralelización del código, permitiendo la integración de otras metaheurísticas o procedimientos al ahorrar aún más tiempo de ejecución. La modificación del parámetro β, responsable de la destrucción en IG, dependiendo de la iteración sin mejora en la que se encuentre el algoritmo, o del parámetro α_{GR} durante el constructivo GRASP, también podría ayudar a la diversificación de las soluciones.

Se espera que la investigación realizada en este TFG sobre el problema MWDSP pueda ser utilizada en el sector del videojuego, con el objetivo de proporcionar una automatización desde en diseño de niveles, como en arquitectura de redes en sistemas distribuidos.

Bibliografía

[1] M. R. Garey and D. S. Johnson, *Computers and intractability*. freeman San Francisco, 1979, vol. 174.

[2] F. Wang, H. Du, E. Camacho, K. Xu, W. Lee, Y. Shi, and S. Shan, "On positive influence dominating sets in social networks," *Theoretical Computer Science*, vol. 412, no. 3, pp. 265–269, 2011, combinatorial Optimization and Applications. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397509007221

[3] G. Wang, H. Wang, X. Tao, and J. Zhang, *Finding Weighted Positive Influence Dominating Set to Make Impact to Negatives: A Study on Online Social Networks in the New Millennium*. Boston, MA: Springer US, 2014, pp. 67–80. [Online]. Available: https://doi.org/10.1007/978-1-4899-7439-6_5

[4] M. M. Daliri Khomami, A. Rezvanian, N. Bagherpour, and M. R. Meybodi, "Minimum positive influence dominating set and its application in influence maximization: a learning automata approach," *Applied Intelligence*, vol. 48, no. 3, pp. 570–593, Mar 2018. [Online]. Available: https://doi.org/10.1007/s10489-017-0987-z

[5] J. Yu, N. Wang, G. Wang, and D. Yu, "Connected dominating sets in wireless ad hoc and sensor networks – a comprehensive survey," *Computer Communications*, vol. 36, no. 2, pp. 121–134, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S014036641200374X

[6] S. Wuchty, "Controllability in protein interaction networks," *Proceedings of the National Academy of Sciences*, vol. 111, no. 19, pp. 7156–7160, 2014. [Online]. Available: https://www.pnas.org/doi/abs/10.1073/pnas.1311231111

[7] J. C. Nacher and T. Akutsu, "Minimum dominating set-based methods for analyzing biological networks," *Methods*, vol. 102, pp. 57–63, 2016, pan-omics analysis of biological data. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1046202315300967

[8] V. V. Vazirani, *Approximation Algorithms*, 1st ed. Berlin, Germany: Springer, Jul. 2001.

[9] A. Caprara, P. Toth, and M. Fischetti, "Algorithms for the set covering problem," *Annals of Operations Research*, 2000. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-0034558102&doi=10.1023%2fa%3a1019225027893&partnerID=40&md5=6aba92b05cfe5485ca3996f532cb5694

[10] B. N. Clark, C. J. Colbourn, and D. S. Johnson, "Unit disk graphs," *Discrete Mathematics*, vol. 86, no. 1, pp. 165–177, 1990. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0012365X9090358O

[11] C. Ambühl, T. Erlebach, M. Mihalák, and M. Nunkesser, "Constant-factor approximation for minimum-weight (connected) dominating sets in unit disk graphs," in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, J. Díaz, K. Jansen, J. D. P. Rolim, and U. Zwick, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 3–14.

[12] Y. Huang, X. Gao, Z. Zhang, and W. Wu, "A better constant-factor approximation for weighted dominating set in unit disk graph," *Journal of Combinatorial Optimization*, vol. 18, no. 2, pp. 179–194, Aug 2009. [Online]. Available: https://doi.org/10.1007/s10878-008-9146-0

[13] D. Dai and C. Yu, "A 5+ϵ-approximation algorithm for minimum weighted dominating set in unit disk graph," *Theoretical Computer Science*, vol. 410, no. 8, pp. 756–765, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397508008499

[14] T. Erlebach and M. Mihalák, "A $(4+\epsilon)$-approximation for the minimum-weight dominating set problem in unit disk graphs," in *Approximation and Online Algorithms*, E. Bampis and K. Jansen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 135–146.

[15] X. Zhu, W. Wang, S. Shan, Z. Wang, and W. Wu, "A ptas for the minimum weighted dominating set problem with smooth weights on unit disk graphs," *Journal of Combinatorial Optimization*, vol. 23, no. 4, pp. 443–450, May 2012. [Online]. Available: https://doi.org/10.1007/s10878-010-9357-z

[16] J. Li and Y. Jin, "A ptas for theweighted unit disk cover problem," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9134, p. 898 – 909, 2015. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84950157760&doi=10.1007%2f978-3-662-47672-7_73&partnerID=40&md5=881d7e138053394cb12129e9d12ef39f

[17] R. Jovanovic, M. Tuba, and D. Simian, "Ant colony optimization applied to minimum weight dominating set problem," 2010, p. 322 – 326. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-79952633691&partnerID=40&md5=5521f09ecde19430d797eead35774ff3

[18] A. Potluri and A. Singh, "Hybrid metaheuristic algorithms for minimum weight dominating set," *Applied Soft Computing*, vol. 13, no. 1, pp. 76–88, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1568494612003092

[19] G. Lin, W. Zhu, and M. M. Ali, "An effective hybrid memetic algorithm for the minimum weight dominating set problem," vol. 20, no. 6, p. 892 – 907, 2016. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85002628777&doi=10.1109%2fTEVC.2016.2538819&partnerID=40&md5=13080befcc91919e175b182a9eb5d077

[20] S. Bouamama and C. Blum, "A hybrid algorithmic model for the minimum weight dominating set problem," *Simulation Modelling Practice and Theory*, vol. 64, pp. 57–68, 2016, advances on Information and Communication Systems. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1569190X15001574

[21] Y. Wang, S. Cai, and M. Yin, "Local search for minimum weight dominating set with two-level configuration checking and frequency based scoring function," *Journal of Artificial Intelligence Research*, vol. 58, pp. 267–295, 2017.

[22] D. Chalupa, "An order-based algorithm for minimum dominating set with application in graph mining," *Information Sciences*, vol. 426, pp. 101–116, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025517310277

[23] M. Albuquerque and T. Vidal, "An efficient matheuristic for the minimum-weight dominating set problem," *Applied Soft Computing*, vol. 72, pp. 527–538, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1568494618303922

[24] T. A. Feo and M. G. Resende, "A probabilistic heuristic for a computationally difficult set covering problem," *Operations Research Letters*, vol. 8, no. 2, pp. 67–71, 1989. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167637789900023

[25] T. A. Feo, M. G. C. Resende, and S. H. Smith, "A greedy randomized adaptive search procedure for maximum independent set," *Operations Research*, vol. 42, no. 5, pp. 860–878, 1994. [Online]. Available: https://EconPapers.repec.org/RePEc:inm:oropre:v:42:y:1994:i:5:p:860-878

[26] F. W. Glover and G. A. Kochenberger, *Handbook of metaheuristics*. Springer Science & Business Media, 2006, vol. 57.

[27] Y. Li, P. M. Pardalos, and M. G. Resende, "A greedy randomized adaptive search procedure for the quadratic assignment problem." *Quadratic assignment and related problems*, vol. 16, pp. 237–261, 1993.

[28] S. Binato, G. C. De Oliveira, and J. L. De Araújo, "A greedy randomized adaptive search procedure for transmission expansion planning," *IEEE Transactions on Power Systems*, vol. 16, no. 2, pp. 247–253, 2001.

[29] D.-H. Lee, J. H. Chen, and J. X. Cao, "The continuous berth allocation problem: A greedy randomized adaptive search solution," *Transportation Research Part E: Logistics and Transportation Review*, vol. 46, no. 6, pp. 1017–1029, 2010.

[30] Y. Gao, X. Gao, X. Li, B. Yao, and G. Chen, "An embedded grasp-vns based two-layer framework for tour recommendation," *IEEE Transactions on Services Computing*, vol. 15, no. 02, pp. 847–859, mar 2022.

[31] A. Saad, A. Kafafy, O. Abd-El-Raof, and N. El-Hefnawy, "A grasp-genetic metaheuristic applied on multi-processor task scheduling systems," in *2018 13th International Conference on Computer Engineering and Systems (ICCES)*, 2018, pp. 109–115.

[32] R. Ruiz and T. Stützle, "A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem," *European Journal of Operational Research*, vol. 44, pp. 2033–2049, 03 2007.

[33] W. Guo, "Flexible selection of output format for sets in java collections: Algorithms and their complexity and reusability," *WSEAS TRANSACTIONS ON MATHEMATICS*, vol. 6, no. 2, p. 309, 2007.

Apéndices

A

Otras herramientas desarrolladas

A.1. Logs de pasos ejecutados

La comprobación de la exactitud del algoritmo es esencial durante el proceso de desarrollo. Herramientas como el depurador de código integrado en el IDE (*Integrated Development Environment*, en inglés) pueden ser muy útiles para asegurar la lógica ejecutada en cada uno de los pasos realizados. Sin embargo, puede ser tedioso y confuso en procedimientos largos o incluso en la revisión de los resultados obtenidos en cada uno de los pasos del algoritmo descritos en el capítulo 3.

Con la finalidad de facilitar dicho proceso, se ha implementado un sistema de *logging* (o registro), que imprime el peso de la solución en cada paso ejecutado por los constructivos para cada instancia perteneciente al subconjunto con el que se esté experimentando. Todos estos ficheros serán nombrados concatenando el nombre de la instancia y la terminación `.txt`, asegurando así que puedan ser leidos por cualquier editor de texto fácilmente. Todos serán almacenados en la carpeta `logs` del proyecto.

El siguiente fragmento muestra un ejemplo de los resultados obtenidos por el constructivo GRASP con $\alpha_{GR} = 0{,}25$. Cada una de las líneas representa la solución obtenida de una de las iteraciones de GRASP, mientras que cada uno de los valores mostrados representan el valor de la solución tras la construcción inicial, el proceso de purga y la búsqueda local, respectivamente. Además, si el algoritmo ha establecido la solución construida como la mejor hasta el momento, aparecerá la palabra `UPDATED`.

```
1 GRASP with alpha 0.25
2
3 676 - 656 - 640 - UPDATED
4 619 - 599 - 599 - UPDATED
5 726 - 627 - 619 -
6 735 - 680 - 662 -
```

```
718 - 648 - 639 -
737 - 659 - 625 -
643 - 623 - 615 -
610 - 590 - 583 - UPDATED
653 - 653 - 632 -
659 - 637 - 628 -
705 - 676 - 594 -
722 - 657 - 623 -
[...]
```

Código A.1: `Problem.dat_50_50_0.txt`

A.2. Creación de CSV para la toma de tiempos

La toma de tiempos es otro de los procesos esenciales para la experimentación necesaria, descrita en el capítulo 5. Tomarlos de forma manual, a través de los resultados impresos por la consola, lleva mucho tiempo y esfuerzo.

Para simplificar esta tarea, al ejecutar el algoritmo completo sobre el conjunto de instancias seleccionado, se genera un fichero `ALL_RESULTS.txt`, que almacena un fichero CSV que contiene el nombre de la instancia, el peso final de la solución obtenida por el algoritmo y el tiempo total de ejecución. Para dar soporte a esta funcionalidad, la clase `Solution` dispone de un campo `executionTime` que almacenará el tiempo transcurrido desde el inicio del constructivo hasta la devolución de la mejor solución encontrada. Con este formato, se pueden importar los resultados directamente a Google Sheets o Excel sin dificultad.

A continuación, se muestra un pequeño fragmento de este fichero:

```
RESULTS
Problem.dat_50_50_0,579,0.105
Problem.dat_50_50_1,552,0.066
Problem.dat_50_50_2,508,0.09
Problem.dat_50_50_3,537,0.053
Problem.dat_50_50_4,494,0.059
Problem.dat_50_50_5,534,0.05
Problem.dat_50_50_6,529,0.097
Problem.dat_50_50_7,503,0.054
Problem.dat_50_50_8,591,0.054
Problem.dat_50_50_9,531,0.054
Problem.dat_50_100_0,385,0.093
[...]
```

Código A.2: `ALL_RESULTS.txt`

B

Resultados por cada instancia de T1

Instancia	$GRASP_{1xNBI + IG. GDGC}$		HTS-DS	
	Promedio	T (s)	Promedio	T(s)
Problem.dat_50_50	535.83	0.07	531.30	0.10
Problem.dat_50_100	369.61	0.05	370.90	0.10
Problem.dat_50_250	174.31	0.05	175.70	0.10
Problem.dat_50_500	95.33	0.04	94.90	0.10
Problem.dat_50_750	60.84	0.05	63.10	0.10
Problem.dat_50_1000	41.47	0.03	41.50	0.00
Problem.dat_100_100	1076.90	0.13	1061.00	0.20
Problem.dat_100_250	616.22	0.09	618.90	0.50
Problem.dat_100_500	356.51	0.10	355.60	0.60
Problem.dat_100_750	260.63	0.09	255.80	0.70
Problem.dat_100_1000	204.25	0.09	203.60	0.90
Problem.dat_100_2000	108.02	0.11	107.40	0.60
Problem.dat_150_150	1624.06	0.27	1580.50	0.50
Problem.dat_150_250	1247.67	0.25	1218.20	0.60
Problem.dat_150_500	755.99	0.27	744.60	3.00
Problem.dat_150_750	562.92	0.22	546.10	7.00
Problem.dat_150_1000	436.31	0.21	432.80	8.10
Problem.dat_150_2000	246.80	0.22	240.80	7.80
Problem.dat_150_3000	169.78	0.23	166.90	7.50
Problem.dat_200_250	1960.63	0.47	1909.70	0.60

Tabla B.1: Comparativa del algoritmo propuesto ($GRASP_{1xNBI + IG. GDGC}$) y el estado del arte (`HTS-DS`) respecto a cada instancia (I).

Instancias	$GRASP_{1xNBI + IG. GDGC}$		HTS-DS	
	Promedio	T (s)	Promedio	T(s)
Problem.dat_250_500	1875.72	1.02	1805.90	2.40
Problem.dat_250_750	1415.81	0.91	1361.90	8.20
Problem.dat_250_1000	1142.87	0.83	1089.90	7.90
Problem.dat_250_2000	650.43	0.70	621.60	8.40
Problem.dat_250_3000	467.15	0.70	448.00	7.60
Problem.dat_250_5000	305.55	0.71	289.50	7.80
Problem.dat_300_300	3289.50	1.37	3175.40	1.30
Problem.dat_300_500	2531.48	1.37	2435.60	1.60
Problem.dat_300_750	1929.01	1.48	1853.80	7.50
Problem.dat_300_1000	1566.74	1.41	1494.00	8.70
Problem.dat_300_2000	906.22	1.11	862.40	8.90
Problem.dat_300_3000	653.27	1.05	624.10	9.50
Problem.dat_300_5000	431.61	1.05	406.10	8.10
Problem.dat_500_500	5546.82	4.81	5304.70	2.70
Problem.dat_500_1000	3796.68	5.74	3607.60	10.40
Problem.dat_500_2000	2337.50	5.11	2176.80	10.80
Problem.dat_500_5000	1123.35	3.57	1042.30	10.90
Problem.dat_500_10000	635.18	3.21	587.20	9.90
Problem.dat_800_1000	8060.57	19.58	7655.00	5.20
Problem.dat_800_2000	5261.78	20.37	4987.30	14.30
Problem.dat_800_5000	2617.66	13.79	2432.60	14.20
Problem.dat_800_10000	1530.92	9.98	1393.70	13.90
Problem.dat_1000_1000	11155.21	29.41	10574.40	8.70
Problem.dat_1000_5000	3947.79	28.63	3656.60	15.70
Problem.dat_1000_10000	2306.02	20.51	2099.80	15.90
Problem.dat_1000_15000	1674.36	16.41	1519.70	16.20
Problem.dat_1000_20000	1322.74	14.33	1200.90	17.90

Tabla B.2: Comparativa del algoritmo propuesto ($GRASP_{1xNBI + IG. GDGC}$) y el estado del arte (HTS-DS) respecto a cada instancia (II).