Trabajo Fin de Máster - Junio/2023

Estudio de metaheurísticos GRASP y Tabu Search aplicados al problema de máxima dispersión

Autor: **Ignacio Arnau Martínez**

Tutora: ANNA MARTÍNEZ GAVARA

Índice general

Capítulo 1

Resumen

El problema de la dispersión es una cuestión crítica y bien reconocida en el campo de la optimización metaheurística, ya que su relevancia se extiende a una amplia variedad de aplicaciones en diferentes áreas. Por ejemplo, dentor del área de optimización, la dispersión se refiere a la separación espacial entre las soluciones en la población, lo cual es fundamental para garantizar una búsqueda global efectiva en el espacio de soluciones.

En este estudio, se llevará a cabo una revisión de la literatura existente sobre el problema de la dispersión, con el objetivo de contextualizar su evolución histórica y destacar los principales avances en este campo. Posteriormente, se abordará el problema de la dispersión con restricciones de capacidad y coste, una variante que se acerca más a los problemas reales que se presentan en diversas áreas de aplicación.

Para resolver esta variante del problema, se propondrán tres algoritmos metaheurísticos eficientes, los cuales son un algoritmo GRASP, un algoritmo Tabu Search y un algoritmo SBTS. Dichos métodos se caracterizan por su capacidad para enfrentar situaciones en las que las soluciones deben satisfacer múltiples restricciones prácticas. Para evaluar la eficacia de los algoritmos propuestos, se llevarán a cabo diversos experimentos sobre distintas instancias numéricas, permitiendo así una comparación con el estado del arte en la resolución de este problema.

En resumen, este estudio aborda de manera exhaustiva el problema de la dis-

persión con restricciones de capacidad y coste, presentando una revisión de su evolución histórica y proponiendo una solución eficiente y efectiva a través de la implementación de algoritmos metaheurísticos y un algoritmo lineal exacto.

Capítulo 2

Introducción

La optimización es un campo fundamental en matemáticas y ciencias de la computación, que se centra en encontrar la mejor solución posible entre un conjunto de posibles alternativas, generalmente en un contexto donde los recursos son limitados. A lo largo de la historia, la optimización ha desempeñado un papel crucial en una amplia gama de aplicaciones, desde la planificación logística y la gestión de recursos hasta la ingeniería y la toma de decisiones en negocios y gobierno. Sin embargo, uno de los desafíos más notables en la optimización es el problema de la dispersión, que se refiere a la dificultad de encontrar soluciones óptimas en conjuntos de soluciones que se extienden a través de un espacio de búsqueda vasto y complejo.

La historia del área de optimización se remonta a los siglos XVIII y XIX con el desarrollo de métodos como el cálculo de variaciones y la teoría de juegos. Sin embargo, fue en el siglo XX cuando se establecieron las bases modernas de la optimización, con el surgimiento de la programación lineal, un método para resolver problemas de optimización linealmente restringidos desarrollado por George Dantzig en la década de 1940, cuando publicó el método Simplex. Esto marcó el inicio de una revolución en la resolución de problemas de optimización y condujo al desarrollo de algoritmos exactos y heurísticos más sofisticados.

En el campo de la optimización, nos podemos encontrar con diferentes tipos de problema los cuales se pueden clasificar en las siguientes categorías:

1. **Optimización Lineal:** Se aplica en problemas donde la función objetivo y las restricciones son lineales. Alguna aplicación de este tipo de optimiza-

ción podría ser en industria, para determinar la cantidad óptima de productos de manera que se maximice el beneficio

2. **Optimización No Lineal:** Se aplica en problemas donde la función objetivo y las restricciones son no lineales. Dicha optimización también se puede aplicar en la industria, cuando por ejemplo la fórmula para obtener el beneficio implica elevar al cuadrado una de las variables.

3. **Optimización Entera:** En problemas de optimización entera, algunas o todas las variables de decisión deben tomar valores enteros. Esto agrega una capa adicional de complejidad. Algún ejemplo de este tipo de problema sería optimizar la cantidad de electrodomésticos a fabricar de manera que se maximice el beneficio. A diferencia del primer caso, en este problema no podríamos obtener una solución donde se fabrique 12,5 lavadoras.

4. **Optimización Combinatoria:** Estos problemas implican encontrar la mejor combinación de elementos de un conjunto finito. Ejemplos incluyen el problema del viajante (TSP) y el problema de la mochila.

Al igual que hay distintos tipos de problemas según sus cualidades, hay diferentes formas de resolver cada uno de estos problemas. Aunque existen muchos algoritmos de resolución y estos se pueden clasificar de distintas formas, la clasificación que he utilizado es la siguiente.

1. **Algoritmos de Resolución Exacta:** Estos tipos de algoritmos obtienen la solución exacta del problema. Se suelen usar habitualmente para problemas de programación lineal y programación lineal entera. Puede ser muy costoso obtener la solución óptima con estos algoritmos según el tamaño del problema.

   Dentro de esta categoría, los métodos más importantes son:

   a) **Método Simplex:** Fue publicado en 1947 por George Dantzig para resolver problemas de optimización lineal. Como hemos mencionado antes, este método fue importante puesto impulso al desarrollo de nuevos métodos en todo el área de la optimización. Aún en la actualidad, este método se continua utilizando para resolver este tipo de problemas.

De manera resumida, este algoritmo consiste en explorar los vértices del poliedro generado por las soluciones, de manera que en cada iteración se va mejorando la solución obtenida en la solución anterior. Como los vértices del poliedro son finitos, tras un número finito de iteraciones se obtiene la solución óptima, aunque cuanto mayor sea el tamaño del problema, más iteraciones habrá que realizar para llegar a la solución óptima.

b) **Branch & Bound:** El método Branch & Bound (B&B) (o Ramificación y Poda) fue propuesto por A. H. Land y A. G. Doig en 1960 para resolver problemas de optimización lineal entera. Al igual que el método anterior, este algoritmo también ha resultado muy importante puesto que incluso a día de hoy, sigue siendo de gran utilidad en la práctica.

Este método interpreta a las soluciones posibles como un árbol de soluciones, donde cada rama conduce a una posible solución posterior a la actual. La característica de esta es que el algoritmo se encarga de detectar en qué ramificación las soluciones dadas ya no están siendo óptimas, para «podar» esa rama del árbol y no continuar malgastando recursos en dichas ramas, puesto que ya tenemos una solución (solución incumbente) mejor que la que obtendríamos de esas ramas. Por eso, este método recibe dicho nombre, puesto que vamos creando ramas y podando las que no son útiles.

2. **Algoritmos Metaheurísticos:** Aunque más adelante daremos una explicación más extensas de que son los algoritmos metaheurísticos, podríamos definir este tipo de algoritmos como unas técnicas de optimización que tratan de buscar soluciones de calidad en un tiempo razonable. Es decir, estos métodos no garantizan hallar la solución óptima del problema, como si lo hace los algoritmos anteriores. Esto no siempre es un defecto, puesto que podríamos tener en casos en los que obtener la solución óptima es computacionalmente muy costoso y obtener, mediante algoritmos metaheurísticos, una solución muy parecida en un tiempo significativamente menor.

Es precisamente en este tipo de problemas donde se utilizan estos algorit-

mos, que suelen ser problemas de optimización no lineal y optimización combinatoria. Uno de los muchos problemas que usan algoritmos metaheurísticos para su resolución es, entre otros, el problema de la dispersión, el cual estudiaremos en profundidad y resolveremos utilizando 3 algoritmos metaheurísticos diferentes.

Capítulo 3

Problema

3.1. Historia y literatura del problema de la dispersión

El problema de la dispersión ha sido objeto de estudio en la comunidad científica durante varias décadas. Inicialmente, el problema de la dispersión se planteó inicialmente como una forma de optimizar el transporte de cargas desde un conjunto de puntos de origen a un conjunto de puntos de destino. En los años siguientes, el problema de la dispersión comenzó a ganar importancia en diversas áreas de aplicación, como el diseño de redes de transporte, la planificación de rutas de distribución, la gestión de inventarios, entre otras. Debido a este crecimiento, paulatinamente se fueron identificando nuevas aplicaciones del problema, y por ello surgieron nuevas variantes y restricciones que complicaron su resolución.

Es por ello que el problema de la dispersión se convirtió en un tema de investigación muy popular en la comunidad científica, especialmente en el campo de la optimización metaheurística, puesto que los enfoques que aportaban esta tipo de optimización permitía abordar con mayor eficacia las variantes más complejas del problema de la dispersión, como por ejemplo aquellos problemas que involucraban varias restricciones.

Aunque se puede rastrear su origen hasta la década de 1960 (cuando se comenzó a utilizar la programación matemática para resolver problemas de optimización

en la industria) los autores del artículo [17] proponen 3 periodos en el desarrollo de los problemas de la dispersión y la diversidad.

1. **Periodo prematuro (1980 - 2000)**:

   Los primeros artículos sobre el problema de la dispersión y la diversidad se remontan a finales de la década de 1970. Shier (1977) [14] fue el primero en reconocer el problema de la dispersión como un problema de optimización continuo, mientras que Chandrasekaran & Daughety (1981) [15] estudiaron los problemas discretos de la dispersión y la diversidad.

   Hasta entonces, el problema más estudiado fue el de la diversidad, aunque el de la dispersión tenía una gran importancia práctica en la modelación de la ubicación de instalaciones. En 1988, Kuby [10] propuso la primera versión discreta del problema de la dispersión, que consistía en situar p instalaciones en un conjunto de localizaciones posibles para maximizar la mínima distancia entre ellas. Más tarde, se extendió este problema para maximizar la suma de las distancias, y ambos problemas recibieron el nombre de Problema de la Dispersión MaxMin (MMDP) y Problema de la Dispersión MaxSum. (MDP)

   Sabemos que el MDP se puede formular de manera trivial de la siguiente manera

$$\begin{aligned} \text{máx} \quad & \sum_{i<j} d_{ij} x_i x_j \\ \text{s.t.} \quad & \sum_{i=1}^{n} x_i = m \\ & x_i \in \{0,1\} \qquad \forall i = 1,\ldots,n. \end{aligned} \tag{3.1.1}$$

   donde la variable x_i toma el valor 1 si el elemento i es seleccionado y toma el valor 0 en caso contrario.

   Pero dicha formulación es no linear (debido a la multiplicación de variables), por lo que Kuby lo reformuló de la siguiente manera:

$$\begin{aligned} \text{máx} \quad & \sum_{i<j} z_{ij} d_{ij} \\ \text{s.t.} \quad & \sum_{i=1}^{n} x_i = m \\ & z_{ij} \leq x_i \quad \forall i, j = 1, ..., n : j > i \\ & z_{ij} \leq x_i \quad \forall i, j = 1, ..., n : j > i \\ & z_{ij}, x_i \in \{0, 1\} \quad i, j = 1, ..., n \end{aligned} \tag{3.1.2}$$

Además, también formuló el problema MMDP de la siguiente manera:

$$\begin{aligned} \text{máx} \quad & D \\ \text{s.t.} \quad & \sum_{i=1}^{n} x_i = m \\ & D \leq d_{ij}(1 + C(1 - x_i) + C(1 - x_j)) i, j = 1, ..., n : j > i \\ & x_i \in \{0, 1\} \quad i = 1, ..., n \end{aligned} \tag{3.1.3}$$

donde C es una constante con un valor muy elevado que hace activa la segunda restricción cuando las localizaciones i y j son seleccionadas. Como se puede observar, dicha formulación también es linear.

En 1990, Erkut [3] introdujo una heurística simple y un método exacto de Branch & Bound, mientras que Kincaid (1992) [9] propuso dos heurísticas basadas en intercambios para el problema de dispersión discreta p-MaxMin.

En esta línea, Ghosh (1996) [4] demostró que el Problema MinMax es NP-duro y propuso una heurística voraz aleatorizada, lo que supuso un primer paso para extender las heurísticas simples a las más complejas. Unos años más tarde, Glover et al. (1998) [7] propusieron cuatro heurísticas diferentes considerando restricciones adicionales para facilitar la transición entre soluciones y adaptarse a diferentes configuraciones del problema.

Al final de este periodo, Agca, Eksioglu & Ghosh (2000) [1] propusieron una aproximación Lagrangiana del problema, que mostró buenos resultados en instancias de 100 nodos, aunque el tiempo computacional era elevado.

2. **Periodo de expansión (2000 - 2010)**:

   Este período se llama "período de expansión"porque los límites que definen el área se ampliaron significativamente. Desde el punto de vista teórico, se clasificaron muchos problemas de dispersión en términos de sus nodos y bordes, mientras que desde el punto de vista práctico, la mayoría de los trabajos publicados consideraron el subgrafo completo inducido por los puntos seleccionados.

   Durante el segundo período de investigación en el área del problema de la dispersión, hubo un gran crecimiento, enfocándose principalmente en el modelo MaxSum, llegándose a desarrollar hasta 30 métodos para resolver este modelo. También se estudió el modelo MaxMin, aunque en menor medida debido a los desafíos que planteaba (y sigue planteando) para los métodos heurísticos.

   Centrándonos en el modelo MaxMin, que es el que estudiaremos en el artículo, en el apartado anterior habíamos mencionado varios autores que propusieron heurísticas para resolver el Problema de Dispersión, entre ellos Erkut, Kincaid y Ghosh. Mientras que las heurísticas de Kincaid, basadas en metodologías complejas, exploraban el vecindario mediante muestreo aleatorio, la heurística de Ghosh examina todo el vecindario en la búsqueda local, implementando la llamada mejor estrategia.

   En contraste, en este periodo, Resende, Martí, Gallego y Duarte (2010) [18] aplicaron la metodología GRASP al problema MaxMin, pero con una implementación eficiente capaz de obtener soluciones de alta calidad en tiempos de ejecución cortos, superando todos los desarrollos anteriores. Aunque sin lugar a dudas este fue el mejor artículo sobre el modelo MaxMin de este periodo, no entraremos en detalle de su planteamiento debido a su extensión, el cual está explicado en el artículo.

3. **Periodo de desarrollo (2010 - Actualidad)**:

   En la última década se han propuesto muchos métodos para resolver problemas de optimización como MaxSum y MaxMin, lo que ha llevado a una producción científica moderada pero con métodos cada vez más complejos para poder competir con la literatura ya existente. Sin embargo, otros modelos de diversidad han recibido poca atención y actualmente se están

desarrollando métodos más eficientes para ellos. En particular, los modelos restrictivos han atraído el interés de los investigadores en esta última década.

En lo que se refiere al problema de la dispersión con restricciones de capacidad y coste, durante esta periodo no existe mucha literatura puesto que solo encontramos 2 artículos que proponen nuevos métodos para resolver este problema. Sin embargo, destacamos también la aportación de Parreño et al (2021) [13], ya que en este artículo los autores realizan un estudio sobre la dispersión que se obtiene utilizando las funciones objetivos MaxMin, MaxSum, MaxMinSum y MinDiff. Tras realizar el análisis de estas funciones, obtienen que el modelo MaxMin evita seleccionar elementos cercanos aunque puede no seleccionar puntos que están a media o larga distancia mientras que el modelo MaxSum favorece la selección de estos puntos que están a media y larga distancia aunque no evita que estos puntos puedan estar cercanos entre sí. Además, afirman que una combinación de ambos modelos daría lugar a un modelo más robusto.

Los otros dos artículos mencionados anteriormente sí que proponen nuevos métodos para la resolución del problema de la dispersión con restricciones. De manera resumida, aunque los explicaremos de manera detallada más adelante, el primer artículo de Martínez-Gavara et al (2021) [11] proponen un método GRASP y un Tabu Search mientras que el artículo de Lozano-Osorio et al (2022) [8] proponen un método VNS, donde constantemente se varía las vecindades que se exploran

A continuación, debido a que hemos mencionado brevemente los diferentes modelos que surgieron durante estas etapas, incluimos una tablas resumen que explica dichos modelos.

Problema		Función Objetivo	Restricciones	Cardinalidad	Contexto
MaxSum	Kuby (1988)	$\sum_{i<j,i,j\in M} d_{ij}$	$\|M\| = m$	Fija	Diversidad
MaxMin	Kuby (1988)	$\min_{i<j,i,j\in M} d_{ij}$	$\|M\| = m$	Fija	Dispersión e Igualdad
MaxMin/Cap/Cost	Rosenkrantz et al. (2000)	$\min_{i<j,i,j\in M} d_{ij}$	$CAP(M) \geq B, COST(M) \leq K$	Variable	Dispersión e Igualdad
MaxMean	Prokopyev et al. (2009)	$\frac{\sum_{i<j,i,j\in M} d_{ij}}{\|M\|}$	$\|M\| \geq 2$	Variable	Diversidad
MaxMinSum	Prokopyev et al. (2009)	$\min_{i\in M} \sum_{j\in M, j\neq i} d_{ij}$	$\|M\| = m$	Fijada	Diversidad
MinDiff	Prokopyev et al. (2009)	$\max_{i\in M} \sum_{j\in M, j\neq i} d_{ij} - \min_{i\in M} \sum_{j\in M, j\neq i} d_{ij}$	$\|M\| = m$	Fijada	Igualdad

3.2. Problemas de dispersión con restricciones de capacidad y coste

Habitualmente, todos las aplicaciones reales en los que se usa el problema de la dispersión con restricciones de capacidad y coste comparten la misma idea sobre la definición del problema:

Dado un conjunto de instalaciones de las cuales sabemos las distancias entre ellas y la capacidad y coste asignado para cada una de ellas, el objetivo es elegir un subconjunto de éstas de manera que la dispersión entre los elementos de dicho subconjunto sea la mayor posible. Sin embargo, la elección de este subconjunto de estas instalaciones está condicionado a las capacidades y coste que tiene asignado cada instalación, de manera que se cumpla que el coste sea menor que un coste límite fijado y que la capacidad sea mayor que un capacidad mínima fijada.

Esta definición del problema es la más común para los problemas de la dispersión con restricciones de capacidad y coste. Sin embargo, sería un error pensar que la dispersión, la capacidad y el coste son iguales para estos problemas. Veamos esto con detalle:

1. **Dispersión:** Según el tipo de problema a resolver, la dispersión que se busca optimizar puede ser distinta. Como hemos mencionado en el apartado anterior, hay varios modelos para optimizar esta dispersión, como por ejemplo casos en los que buscamos los elementos del subconjunto escogido estén muy separados entre sí aunque la suma de las distancias entre estos elementos no sea la máxima posible; u otros casos en las que la suma de la distancias entre los elementos sea la máxima posible aunque haya elementos cercanos entre sí.

2. **Capacidad y coste:** Algo parecido ocurre con estos términos. Cuando nos referimos a la capacidad de las distintas instalaciones, no solo se refiere al concepto de capacidad como medidas de almacenamiento, también puede referirse a otros tipos de medidas. Un ejemplo real podría ser el siguiente: se necesitan instalar antenas telefónicas en una zona de manera que toda este área disponga de señal y el coste total de las antenas no superé

el límite de 1M €. En este ejemplo sencillo, la capacidad del problema se refiere a la área con señal que hay en la zona.

Aunque es menos común, también puede darse problemas en los que el coste no represente el coste económico del problema. Un ejemplo podría darse en un problema de diseño de suministros de energía, en el que no se quiera sobrepasar un límite emisiones de gases contaminantes, de manera que el sistema de suministros sea lo menos contaminante posible. En este ejemplo ficticio, el coste representaría la cantidad de gases contaminantes que emitiría cada central.

3.3. Aplicaciones

Continuando con el anterior apartado, el problema de la dispersión es relevante en muchas áreas de aplicación donde se necesitan soluciones óptimas a problemas complejos. Algunas de estas áreas de aplicación donde es necesario resolver el problema de la dispersión con restricciones de capacidad y coste son las siguientes:

1. **Diseño de redes de telecomunicaciones:** En la planificación y diseño de redes de telecomunicaciones, es necesario resolver este problema para determinar la ubicación y la asignación de recursos de red (como torres de transmisión, antenas, etc) de manera que se maximice la cobertura de la red y se minimice el costo total de construcción y operación.

2. **Planificación de rutas y logística:** En la planificación de rutas y la logística, es importante maximizar la eficiencia y minimizar los costos. El problema de la dispersión se puede utilizar para optimizar la planificación de rutas, a la vez que se maximiza la cobertura de la red de transporte y se minimiza la redundancia en los movimientos de los vehículos y el coste total del transporte.

3. **Conservación de la diversidad biológica**: Aunque es menos probable, también se podría utilizar el problema de la dispersión en el campo de la biología como es la conservación de la diversidad biológica y la gestión de recursos naturales. Por ejemplo, si se desea preservar la biodiversidad en un área determinada, se podría utilizar este problema para planificar la ubicación

y la distribución de diferentes especies de plantas y animales de manera que se maximice la diversidad

En general, el problema de la dispersión es relevante en cualquier aplicación que requiera la optimización de soluciones complejas y en las que sea importante cubrir todo el espacio de búsqueda para encontrar soluciones óptimas.

Capítulo 4

Resolución

4.1. Algoritmos Metaheurísticos

Una de las varias definiciones que tiene la metaheurística, es la que aportan Sörensen y Glover (2013) en [20] es la siguiente: “Una metaheurística es un marco algorítmico de alto nivel e independiente del problema que proporciona un conjunto de directrices o estrategias para desarrollar algoritmos de optimización heurísticos”

En este marco algorítmico como definen los autores, obtenemos los algoritmos metaheurísticos, que son técnicas de optimización que se utilizan para resolver problemas de optimización combinatoria y numérica, en los que se busca encontrar una solución óptima o cerca de la óptima. En contrates con los métodos de optimización exactos, que garantizan la solución óptima (aunque pueden ser computacionalmente costosos), los algoritmos metaheurísticos son heurísticas que utilizan estrategias de búsqueda inteligentes para encontrar soluciones de calidad en un tiempo razonable.

Una de las muchas virtudes que tienen estos algoritmos es que son capaces de explorar el espacio de búsqueda de soluciones de manera más eficiente mediante la generación de soluciones aleatorias y la evaluación de su calidad a través de una función objetivo. A partir de ahí, los algoritmos metaheurísticos utilizan diferentes operaciones como añadir y eliminar elementos de la solución o intercambiar un elemento de la solución por otro fuera de ella, para así generar nuevas soluciones y mejorar gradualmente la calidad de las soluciones genera-

das.

Dos de los algoritmos metaheurísticos más conocidos son el algoritmo GRASP y el Tabu Search, los cuales emplearemos más adelante para obtener nuestras soluciones

Sobre las aplicaciones que tienen estos algoritmos, encontramos aplicaciones en varios campos, como en la gestión de la producción, la planificación de la logística o la economía entre otras.

En este contexto y debido a que el problema de la dispersión con restricciones es NP-duro, demostrado por Ghosh (1996) [4], no es de extrañar el uso de estos algoritmos metaheurísticos para resolver este problema, hasta el punto de que el problema de la dispersión es un desafío importante en la optimización metaheurística puesto que se requiere una atención especial para garantizar que las metaheurísticas puedan encontrar soluciones óptimas para este problema.

4.2. Literatura del problema de la dispersión con restricciones de capacidad y coste

4.2.1. Árticulo de Rosenkrantz et al. (2000)

Como hemos mencionado en el capítulo anterior, la literatura existente sobre el problema de la dispersión es muy extensa. Sin embargo, en lo que se refiere al problema de la dispersión con restricciones de capacidad y coste no hay apenas literatura.

El primer artículo en el que se propone una solución para este problema es aquel escrito por Rosenkrantz et al. (2000) [19]. De forma resumida, en este artículo se proponen diferentes modelos según la dispersión, la capacidad y el coste sean función objetivo o restricciones. De estas tres posibles opciones, nos centraremos en aquella en la que la función objetivo es la dispersión (o en este caso, la Max-Min Distancia) y la capacidad y el coste actúan como restricciones.

Para resolver este modelo, se propone una algoritmo voraz en la que en cada iteración se elige la localización que más mejora a la solución, satisfaciendo una distancia mínima y la restricción de capacidad. Para escoger este sitio se crean dos listas, una con los sitios ordenados por capacidad y otra con las distintas distancias del problema y en cada iteración se añade el sitio con mayor capacidad cuya distancia es el máximo de las distancia. Sin embargo, esto algoritmo no garantiza que se cumpla la restricción de la capacidad . Por tanto, al final del algoritmo, comprueba si se cumple la restricción de capacidad.

Este modelo de solución tiene varios inconvenientes: el primero de ellos (como hemos comentado antes) es que no se garantiza que la restricción de capacidad se cumple. El siguiente inconveniente, es que en en ningún momento se tiene en cuenta la restricción de coste.

4.2.2. Árticulo de Martínez-Gavara et al.

Años más tarde, y tomando la línea de este artículo, Martínez-Gavara et. al (2021) [11] propusieron otros modelos de solución para resolver este problema. En este artículos, los autores propones dos metodologías distintas, un GRASP y un Tabu Search, para resolver el problema de la dispersión con restricciones de capacidad y coste.

Al igual que en este documento, la función objetivo que utilizan los autores para plantear el modelo matemático es la función objetivo Min-Max Distancia, y por tanto, consideran la capacidad y el coste como restricciones. Lo primero que realizan los autores en el artículo, a modo de comparación respecto a la literatura anterior de Rosenkrantz et al. (2000) [19], es adaptar el algoritmo voraz que proponían los autores en este artículo de manera que se tenga en cuenta el coste. Para ello, la única modificación que realizan es ordenar los nodos por el ratio entre la capacidad y el coste.

Una vez realizada esta modificación, lo cual servirá para poder comparar resultados, los autores proponen la primera de sus metodologías, el método GRASP. La función voraz que utiliza dicho método es la suma de las distancias, capacidades y costes normalizados, donde cada sumando está multiplicado por un parámetro distinto, los cuales varían entre 0 y 1. La función voraz es la siguiente:

$$g(i) = \beta_d \cdot \frac{d_i}{\max\limits_{j \in CL} d_j} + \beta_c \cdot \frac{c_i}{\max\limits_{j \in CL} c_j} + \beta_k \cdot \frac{(\max\limits_{j \in CL} k_j - k_i)}{\max\limits_{j \in CL} k_j}$$

donde $d_i = \sum_{j \in M} d_{ij}$ siendo M el conjunto de sitios seleccionados, c_i y k_i la capacidad y el coste del nodo i respectivamente y CL la lista de candidatos posibles

Para terminar con este método, señalar también que los autores proponen el mismo método GRASP anterior, pero con una serie de filtros para acelerar la fase de mejora del algoritmo. No nos centraremos en explicar este método (debido a que es prácticamente igual que el anterior) aunque en el artículo está explicado detalladamente.

El siguiente método que proponen los autores es un Tabu Search. De manera resumida (puesto que más adelante explicaremos detalladamente en que consiste dicho algoritmo), el método consta de tres fases: fase de construcción, fase a corto plazo (short term phase) y fase a largo plazo (long term phase). Para la fase de construcción, la función voraz que se utiliza es la que misma que se utiliza en el método GRASP aunque la forma de elegir el mejor sitio difiere un poco, puesto que en este caso siempre se elige aquel sitio cuyo valor de la función voraz es máximo.

En cuanto a las dos siguientes fases, explicaremos brevemente en que consisten, ya que en el artículo se encuentra una explicación detalla de estas: en la fase a corto plazo se realizan intercambios de manera que se obtenga un óptimo local a la vez que se van explorando diferentes opciones y al cabo de un número de iteraciones, comienza la fase a largo plazo, la cual examina en profundidad las regiones del espacio de solución menos visitadas, para construir soluciones que pueden ser mejores que las encontradas en la anterior fase.

Una vez que los autores han realizado los diferentes experimentos tanto para ajustar los parámetros (fine tunning) como de comprobación de los métodos, los autores sacan las siguientes conclusiones: para instancias de tamaño pequeño y medio ($n = 50, 150$) el método GRASP obtiene mucho mejores resultados que el artículo compuesto Rosenkrantz et al. y mejores resultados que el método Tabu Search. Sin embargo, para instancias de gran tamaño ($n = 500$) es el método

Tabu Search el que mejores resultado obtiene, obteniendo muchísimos mejores resultados que el método GRASP y el algoritmo voraz del artículo de partida.

4.2.3. Árticulo de Lozano-Osorio et al.

Un año más tarde, Lozano-Osorio et al. (2022) [8], continuando el estudio explicado anteriormente, proponen una nueva solución para resolver el problema de la dispersión con restricciones de capacidad y coste con la función objetivo Min-Max con el objetivo de mejorar los resultados de los estudios anteriores para instancias de tamaño medio y grande.

El método que se propone como solución para este problema es Multi Start Basic VNS. La principal diferencia que presenta frente a los métodos del artículo de Corberán et al, son las vecindades que se exploran durante la fase de mejora

El método que proponen los autores está basado en los algoritmos metaheurísticos VNS (Variable Neighborhood Search). Este tipo de algoritmo fue introducido por Mladenovic and Hansen (1997) [12] el cual se basa en, en la fase de mejora de la solución, obtener óptimos locales y además escapar de lo "valles"que lo contienen. Es decir, que al encontrar un óptimo local, en vez de seguir explorando la vecindad de esa solución, busca otra vecindad completamente diferente si una mejora se produce, para así poder obtener otro óptimo local y repetir el proceso.

Este cambio de vecindades no ocurre en los algoritmos de GRASP, puesto que en este método se construye una solución y al obtener el óptimo local de esta solución el método se vuelve a repetir, ni en el Tabú Search, que aunque explora más vecindades que el GRASP, si no ajustamos un tamaño adecuado de la Tabu List para la instancia del problema puede ocurrir que el algoritmo no salga del "valle"que contiene ese óptimo local.

Basándose en este algoritmo, los autores proponen el método MS-BNVS (Multi Start - Basic VNS), que están divido en dos fases, la fase de construcción (se correspondería al MS) y la fase de mejora (se correspondería con el BVNS). La fase de construcción la denominan Multi Start (Múltiples comienzos) puesto que desarrollan dos métodos diferentes para construir la primera solución. Ambos métodos son métodos voraces, pero se diferencian en que uno usa como función

voraz el ratio entre la capacidad y coste (denominada C1 en el artículo) y otra función voraz solo basada en las distancias (denominada C2). En adición, los autores desarrollan otro método (denominado Caux) en caso de que los primeros métodos no obtengan soluciones factibles.

Una vez completada la fase de construcción, el método comienza con la fase de mejora. Para ello, los autores desarrollan hasta tres distintos métodos de búsqueda local: el primero basado en intercambios (denominado LS1); el segundo basado en movimientos de añadir y quitar nodos según la solución cumpla o no las restricciones de capacidad (denominado LS2); y el último, similar a LS2 pero en caso de que una nueva posible solución iguale a la mejor solución durante el proceso de búsqueda local, esta se desempata teniendo en cuanta el valor de una función basada en las distancias de cada solución. Esta explicación es a modo de resumen, y en el artículo citado anteriormente se encuentre mucho más detallado con los correspondientes pseudo-códigos

En conclusión, el objetivo que se proponía obtener al desarrollar el método se cumple, puesto que éste es capaz de encontrar soluciones de gran calidad para instancias de gran tamaño ($n \geq 500$) sin perder la eficacia para instancias de menor tamaño.

Capítulo 5

Propuesta metodológica

5.1. Definición del Problema

Como hemos ido mencionando en los puntos anteriores, en este trabajo vamos resolver el problema de la dispersión con restricciones de capacidad y coste. Para ello, como también se ha comentado, emplearemos la función objetivo MaxMin ya que nos interesa que todos los nodos de la solución estén lo más alejados posible entre ellos. Por tanto, el problema a resolver se define de la siguiente manera:

Dado un conjunto de n nodos posibles, X, conectadas por aristas en E, el problema consiste en encontrar un subconjunto S de X que satisfaga restricciones de capacidad y coste, de modo que la distancia mínima entre los sitios en S se maximice. Sea B la capacidad mínima requerida y sea K el presupuesto máximo permitido. Entonces, para cada sitio $i \in X$, definimos $b_i \geq 0$ y $k_i \geq 0$ como su capacidad y coste, respectivamente. Sea $d_{ij} \geq 0$ la distancia entre los sitios i y j. El modelo de programación matemática para el problema de la dispersión generalizado se basa en las variables binarias estándar z_i que toman el valor 1 si se selecciona el sitio i y 0 en caso contrario. Por tanto, podemos formular el modelo de la siguiente manera:

$$\begin{aligned} \text{máx} \min_{i,j \in X, i \neq j} \quad & d_{ij} \cdot z_i \cdot z_j \\ \text{s.t.} \quad & \sum_{i=1}^{n} b_i \cdot z_i \geq B \\ & \sum_{i=1}^{n} k_i \cdot z_i \leq K \\ & z_i \in \{0,1\} \quad \forall i \in S \end{aligned} \tag{5.1.1}$$

Destacar que la cardinalidad del subconjunto S no está fijada, si no que depende de la capacidad y el coste de los nodos ya seleccionados.

En cuanto a la función objetivo, supongamos que $f(S)$ es la función objetivo, la cual mide la mínima distancia entre los pares de nodos incluidos en la solución. Matemáticamente, podemos expresar la función objetivo como:

$$f(S) = \min_{i,j \in X, i \neq j} d_{ij}$$

Por tanto, el objetivo del problema es encontrar el subconjunto S^* el cual maximiza $f(S)$ para todo $S \subseteq X$ y satisface las restricciones de capacidad y coste.

Como hemos apuntado anteriormente, el principal inconveniente de este problema es que la función objetivo es no lineal, lo que provoca que no se puedan aplicar algoritmos exactos para su resolución, como Gurobi o el método Complex. Sin embargo, podemos reformular este problema cuadrático dando lugar al siguiente problema de programación lineal entera. Esta reformulación, fue propuesta en [10] para el problema de p-dispersión, y adaptado por Anna Martínez Gavara en [16] para el problema generalizado de la dispersión.

$$\begin{aligned} \text{máx} \quad & m \\ \text{s.t.} \quad & \sum_{i=1}^{n} b_i \cdot z_i \geq B \\ & \sum_{i=1}^{n} k_i \cdot z_i \leq K \\ & m \leq d_{ij} + M(2 - z_i - z_j) \quad \forall 1 \leq i \leq j \leq n \\ & z_i \in \{0,1\} \quad \forall i \in S \end{aligned} \tag{5.1.2}$$

La cota superior M sobre las distancias, garantizan que m es la mínima dis-

tancia entre los sitios seleccionados. Este modelo, que es equivalente al que hemos escrito anteriormente, tiene la ventaja de que se puede resolver usando programación lineal entera, aunque debido a la cantidad de restricciones (para un problema de 50 nodos tendríamos unas 1375 restricciones) solo se podría plantear resolverlo para problemas con instancias de pequeñas o medio tamaño, pues es muy costoso computacionalmente.

Otras reformulaciones del modelo (5.1.1) son posibles, como la que se propone en [8], añadiendo las restricciones de capacidad y coste a la reformulación que propuso Sayah and Irnich en [2] del problema de la p-dispersión. Dicha reformulación de Sayah e Irnich, se basa en que la distancia mínima es una de las distancias que se tiene como entrada. No incluiremos dicho modelo en este estudio (se puede encontrar en el articulo [8]), pero destacamos que con dicha reformulación, seguimos teniendo un número elevado de restricciones, tal que para n variables tendríamos $n(n-1)/2+3$ restricciones.

5.2. Programación

Como hemos comentado anteriormente, los distintos métodos que vamos a comentar los hemos desarrollado en lenguaje **Python**.

Todos los métodos que mencionáremos, se aplican sobre una instancia. Esta instancia es un archivo de texto el cuál contiene la siguiente información:

1. **Nodos:** Representa el número de nodos que componen en el problema. A partir de ahora la denominaremos como n.

2. **Matriz de Adyacencia:** Matriz con las distancias entre cada nodo. Se trata de una matriz simétrica (puesto que la distancia del nodo i al nodo j es la misma que la del nodo j al nodo i) cuya diagonal es 0, ya que la distancia de un nodo consigo mismo es nula. A la esta distancia la denominamos d_{ij}.

3. **Costes:** Coste de cada nodo. Al coste del nodo i lo denominaremos a_i.

4. **Capacidades:** Capacidad de cada nodo. A la capacidad del nodo i lo denominaremos c_i.

5. **Coste Máximo:** Coste máximo que puede tener la solución. A partir de ahora la denominaremos K

6. **Capacidad mínima:** Capacidad mínima que debe tener la solución. A partir de ahora la denominaremos B.

Indicar que las instancias también incluían el coste de unidad para cada nodo, pero no lo tendremos en cuenta para nuestro problema.

Por tanto, el primer paso antes de aplicar el método es leer correctamente la instancias o instancias que queremos simular. Una vez leída la instancia, el paso siguiente es crear la solución (de momento vacía) que nos devolverá nuestro método. De mismo modo que las instancias, cada solución contiene los siguientes elementos:

1. **Instancia:** Instancia sobre la que calculamos la solución.

2. **Nodos seleccionados:** Lista con los nodos que componen la solución. Esta lista la denominaremos como S.

3. **Función objetivo:** Función objetivo de la solución. En esta caso, se trata de la mínima distancia que hay entre los nodos de la solución, es decir:

$$d^* = \min d_{i,j} \quad \forall i, j \in S$$

4. **Nodos Críticos:** Para cada nodo de la solución, nodo o nodos de la solución con los que tiene la mínima distancia, por tanto:

$$cr_i = \{j : j \in S \wedge d^i_{min}\} \quad \text{donde } d^i_{min} = \min d_{ij} \quad \forall j \in S$$

5. **Coste:** Coste total de los nodos incluidos en la solución.

6. **Capacidad:** Capacidad total de los nodos incluidos en la solución.

7. **Tiempo para la mejor solución:** Tiempo que tarda el método en encontrar la mejor solución. No hay que confundirse con el tiempo que tarda el método en ejecutarse.

Una vez que hemos creado nuestra solución, ya podemos ejecutar los métodos que hemos desarrollado.

5.3. Algoritmo GRASP

5.3.1. Definición e historia

El primer método que proponemos en este trabajo es un algoritmo GRASP. El algoritmo GRASP (Greedy Randomized Adaptive Search Procedure) fue propuesto por primera vez en 1995 por Thomas A. Feo y Mauricio G. C. Resende. En este artículo, [21], se puede ver de manera detallada como funciona dicho método. Además, se definen los diversos componentes que componen un GRASP y cómo desarrollar tales heurísticas para problemas de optimización combinatoria. Por último, el artículo concluye con una breve revisión de la literatura de implementaciones de GRASP y menciona dos aplicaciones industriales.

El método GRASP, como se menciona en el artículo, se basa en los algoritmos voraces, que de manera resumida, se tratan de algoritmos metaheurísticos que crean una solución eligiendo la opción que más mejora a la función objetivo en cada paso local. Sin embargo, estos algoritmos tienen el inconveniente de que al construir varias soluciones, las soluciones halladas son muy parecidas. Esto se debe a que el único factor aleatorio que hay es el nodo inicial de la solución. Por esta razón, surgen los algoritmos GRASP. La idea es muy parecida a los algoritmos voraces, pero hay más variedad entre las soluciones que se obtienen, afirmación que explicaremos más adelante.

Explicando brevemente, dentro de cada iteración de GRASP hay dos fases: la primera construye de manera inteligente una solución inicial mediante una función voraz aleatoria adaptativa; la segunda aplica un procedimiento de búsqueda local a la solución construida con la esperanza de encontrar una mejora. Con esta explicación ya podemos observar la principal ventaja con los algoritmos voraces: se trata del proceso de búsqueda local que aplicamos a la solución obtenida del algoritmo voraz.

En esta fase de búsqueda local, aparecen dos conceptos importantes que serán claves para que dicho proceso no devuelva en todas las iteraciones la misma solución: la lista restringida de candidatos (RCL) y el parámetro umbral, α (threeshold), los cuales explicaremos en los siguientes apartados.

Podemos entender como funciona el método GRASP con el siguiente ejemplo: Si imaginamos que el espacio del problema es una cordillera con varias montañas, en cada iteración del algoritmo GRASP nos situamos en un punto de una montaña (fase de construcción) y posteriormente procedemos a escalar dicha montaña (fase de mejora). Cuando alcanzamos la cima, comienza la siguiente iteración. De esta manera, a lo largo del método habremos escalado diferentes montañas, aunque no podemos asegurar que alguna de esas montañas sea las más alta de la cordillera (que sería la solución óptima de nuestro problema).

5.3.2. Fase de Construcción

Como hemos mencionado anteriormente, el método GRASP está compuesto de dos fases, una fase de construcción y otra fase de mejora. En este apartado, nos centraremos en la fase de construcción.

Como su nombre indica, en esta fase el algoritmo construye una posible solución del problema. Para ello, el algoritmo en cada iteración evalúa los nodos fuera de la solución mediante la función voraz y entre aquellos que cumple un mínimo valor de esta función, se elige uno al azar. Por tanto, una vez que ya se ha construido la solución, el proceso termina y daría paso a la fase de mejora.

Con un ejemplo, siguiendo con el ejemplo de las cordilleras del apartado anterior, la fase de construcción consiste en situarnos en un punto de alguna montaña, para así poder escalarla. Además, este punto se intenta encontrar lo más arriba de la montaña posible.

Destacar que, según la solución voraz empleada, la solución que obtengamos puede ser no factible debido a que no cumple las restricciones del problema. Para estos casos, lo que se hará será aplicar una función voraz que solo tenga en cuenta las capacidades y coste, como se explicará más adelante.

El pseudo-código del algoritmo que hemos considerado para la fase de construcción es el siguiente:

Algorithm 1: Fase de Construcción

```
1  S ← ∅;
2  CL ← X;
3  i ← Random(CL);
4  S ← {i};
5  Evaluate(S);
6  CL ← Update(CL);
7  while CL ≠ ∅ do
8      g_min = mín_{x∈CL} g(x);
9      g_max = máx_{x∈CL} g(x);
10     μ ← g_min + α · (g_max − g_min);
11     RCL ← {i ∈ CL / g(i) ≥ μ};
12     S ← Random(RCL);
13     Evaluate(S);
14     CL ← Update(CL);
15 end
16 return S
```

El algoritmo comienza añadiendo a la solución un nodo al azar y se crea la CL (lista de candidatos), la cual está compuesta por aquellos nodos que se puedan añadir a la solución sin violar las restricciones, que en este caso solo se trata de la restricción de coste. Una vez creada la CL, el método comienza a iterar.

En cada iteración del método, evaluamos la función voraz para cada nodo de la CL y según el umbral escogido creamos la RCL. La condición que hemos escogido para que un nodo se añada a la RCL es la siguiente:

$$RCL = \{i \ / \ g(i) \geq \min_{j \in CL} g(j) + \alpha \cdot (\max_{j \in CL} g(j) - \min_{j \in CL} g(j)\}$$ donde $g(i)$ es la evaluación de la función voraz para el nodo i y α es el threeshold escogido.

Cuando ya hemos creado la RCL, añadimos a la solución un nodo al azar de la RCL y volvemos a crear la CL para la solución actual. El algoritmo acaba cuando no se pueden añadir nodos a la solución, i.e, cuando la CL está vacía.

A simple vista se puede deducir que la elección de la función voraz g es muy importante y que es la principal responsable de la calidad del método. Por tanto, en

este trabajo hemos desarrollado varias funciones voraces, que son las siguientes:

1. **BestDistGreedyFunction:** Esta función se define como:

$$g_1(i) = \min_{j \in S} d_{ij}$$

donde S representa la solución y d_{ij} representa la distancia los nodos i y j

Esta segunda función voraz, a diferencia de la anterior, no evalúa los nodos por su capacidad y coste sino que solo tiene en cuenta las distancias de estos.

La ventaja de este función es que las soluciones serán de mayor calidad, es decir, los nodos estarán más dispersos entre sí. Por otro lado, el inconveniente que representa es que las soluciones obtenidas pueden ser no factibles ya que en ningún momento estamos teniendo en cuenta las restricciones del problema.

2. **DistCapCostGreedyFunction:** La ultima función voraz que hemos desarrollado, se define de la siguiente manera:

$$g_2(i) = \beta_d \cdot \frac{\bar{d}_i}{\max_{j \in CL} d_{ij}} + \beta_b \cdot \frac{b_i}{\max_{j \in CL} b_j} + \beta_k \cdot \frac{\max_{j \in CL} k_j - k_i}{\max_{j \in CL} k_j}$$

donde $\bar{d}_i = \sum_{j \in S} d_{ij}$; β_d, β_b y $\beta_k \in [0,1]$; y d_{ij}, b_i y k_i es lo mismo que en las funciones anteriores.

A diferencia de las funciones anteriores, esta función voraz sí que tiene en cuenta tanto las distancias, como las capacidades y coste de los nodos. Además, cada término tiene asociado un parámetro.

En resumen, lo que hace esta función es, para un nodo, calcular un valor que represente la distancia, otro valor que represente la capacidad y otro

valor que represente el coste. Cada uno de estos valores está multiplicado por un parámetro entre 0 y 1 y por último, se suman estos valores parametrizados.

En esta función, los parámetros juegan un papel importante. El valor de cada parámetro representa que aspecto tiene más importancia a la hora de escoger los nodos. Por ejemplo, si el valor de $\beta_d = 0$ entonces estaríamos en un caso parecido a la primera función ya que no estaríamos teniendo en cuenta las distancias. Si por otro lado, tuviéramos que $\beta_d = 0.5$ mientras que $\beta_c = \beta_k = 0.25$, la función da el doble de importancia a las distancias que a las capacidades y coste, sin dejar de tenerlas en cuenta.

Por tanto, esta función tiene varias ventajas, ya que las soluciones pueden ser de buena calidad y además factibles. Sin embargo, el inconveniente que encontramos es que tenemos que realizar varias experimentos para encontrar el valor de los parámetros adecuados.

3. **RetroactiveGreedyFunction:** Esta función voraz es una propuesta novedosa que se propone en este estudio. Se define esta función de la siguiente manera:

$$g_3(i) = \beta_{dist} \cdot \frac{\hat{d}_i - \min\limits_{j \in CL} d_{ij}}{\max\limits_{j \in CL} d_{ij} - \min\limits_{j \in CL} d_{ij}} + \beta_{ratio} \cdot \frac{r_i - \min\limits_{j \in CL} r_j}{\max\limits_{j \in CL} r_j - \min\limits_{j \in CL} r_j}$$

donde S representa la solución; $\hat{d}_i = \min\limits_{j \in S} d_{ij}$ representa la distancia entre el nodo i y la solución S; y donde $r_j = \frac{b_j}{k_j}$ donde b_j y k_j son la capacidad y el coste respectivamente del nodo j.

La idea de esta función es tener en cuenta tanto las distancias como el ratio entre la capacidad y el coste. Para ello, lo que hacemos es normalizar la distancia del nodo candidato a la solución por la distancia mínima y máxima que tiene la solución con todos los candidatos posibles; y de igual manera para el ratio, normalizando el ratio del nodo candidato por el ratio máximo y mínimo de todos los candidatos posibles.

Puede suceder que la lista de candidatos a entrar en la solución solo tenga un elemento, es decir, $|CL| = 1$. En ese caso, consideramos que el denominador del primer término es $\max_{j \in CL} d_{ij}$ y el denominador del segundo término es $\max_{j \in CL} r_j$

Con esta normalización de las distancias y el ratio obtenemos que una función que tiene en cuenta tanto las distancias como el ratio, y la importancia dada a cada una es regulada por los parámetros β_{dist} y β_{ratio}. Sin embargo, a diferencia de la función voraz definida en el apartado anterior, $DistCapCostGreedyFunction$, estos parámetros no son fijos, si no que estos parámetros irán variando según el método construya soluciones.

Comenzamos dando toda la importancia al término de la distancia, buscando obtener la mejor solución posibles, y si las soluciones construidas no son factibles, progresivamente se restará importancia al término de la distancia a favor del término del ratio y terminará dando toda la importancia al término del ratio, para así facilitar la construcción de una solución factible. Así, los valores que van tomando estos parámetros son:

a) $\beta_{dist} = 1$ y $\beta_{ratio} = 0$

b) $\beta_{dist} = 0.8$ y $\beta_{ratio} = 0.2$

c) $\beta_{dist} = 0.6$ y $\beta_{ratio} = 0.4$

d) $\beta_{dist} = 0.4$ y $\beta_{ratio} = 0.6$

e) $\beta_{dist} = 0.2$ y $\beta_{ratio} = 0.8$

f) $\beta_{dist} = 0$ y $\beta_{ratio} = 1$

Con esta metodología, conseguimos que siempre se obtenga una solución factible

4. **ProfitableGreedyFuncion:** Esta función se define como:

$$g_4(i) = \frac{b(i)}{k(i)}$$

donde $b(i)$ y $k(i)$ son la capacidad y el coste respectivamente del nodo i

Como se puede deducir, esta función evalúa como de rentable es el nodo según el coste y la capacidad de éste, es decir, que nodo tiene más capacidad a menor coste. Por tanto, cuanto mayor sea el valor de $g_2(i)$ mayor será la rentabilidad del nodo.

La ventaja de esta función es que las soluciones son muy comúnmente factibles. Los inconvenientes sin embargo, uno es que no tiene en cuenta las distancias con los nodos de la solución por lo que las soluciones que se hallan no sean de calidad; el segundo es que los nodos que aparecen en las soluciones suelen ser los mismos, ya que los nodos más rentables siempre son los mismos en cada iteración.

Esta función no la usaremos como primera opción para el GRASP, si no que la usaremos para obtener soluciones cuando las funciones definidas anteriormente no son capaces de construir una solución factible tras las iteraciones máximas establecidas.

El último parámetro que podemos variar es el umbral, α. Conforme hemos definido la RCL, tenemos que un α cercano a 1 implica que la condición para entrar a la RCL es muy estricta y por tanto las soluciones halladas pueden ser muy parecidas (y por tanto, el algoritmo sería más parecido a un algortimo voraz); mientras que un α cercano a 0 implicaría que la condición para entrar a la RCL es muy flexible y por tanto, prácticamente estaríamos añadiendo al azar un nodo de la CL, lo cual empeora la calidad de las soluciones creadas.

No se puede saber de manera exacta cuál es el mejor α, ya que este puede depender según la instancia que estemos simulando. Por ello, en este trabajo hemos realzado varias simulaciones con distintos valores de α para estudiar el comportamiento de las soluciones.

5.3.3. Fase de Mejora

Una vez que se ha construido una solución en la fase de construcción, comienza la fase de mejora del algoritmo GRASP. Esta fase consiste en mejorar la solución construida en la fase anterior.

Para mejorar la solución construida empleamos un algoritmo de búsqueda local, el cual irá analizando diferentes posibles soluciones que mejoren a la solución actual (y que están muy próximas en el espacio del problema a la solución actual). Para ello, el algoritmo en cada iteración, toma un nodo de la solución y explora entre los nodos que están fuera de la solución posibles intercambios que puedan mejorar la solución.

Si regresamos al ejemplo de las cordilleras, esta fase de mejora comienza cuando ya estamos situados en algún lugar de alguna montaña y ahora procedemos escalar dicha montaña hasta la cima, y para escalar la cima, buscamos entre los sitios que tenemos al lado si hay alguno más alto al que podemos ir. Una vez que llegamos a la cima, la fase de mejora termina.

El pseudo-código del algoritmo de búsqueda local que hemos utilizado para esta fase, es el siguiente:

Algorithm 2: Fase de Mejora

```
1  SolutionShift ← True;
2  while SolutionShift = True do
3      d*_min ← 0;
4      for i ∈ S do
5          NodeOut ← i;
6          d_out ← MinDist(NodeOut, S) ;        /* Calcula la mínima
           distancia del nodo i con la solución */
7          CL ← {CL : S = S\{NodeOut}};
8          if CL ≠ ∅ then
9              for candidate ∈ CL do
10                 d_cand ← MinDist(candidate, S);
11                 if d_cand > d_out & d_cand > d*_min then
12                     NodeIn ← candidate;
13                     d*_min ← d_cand;
14                     SolutionShift ← True;
15                 end
16             end
17         end
18         S ← Shift(S, NodeOut, NodeIn);
19     end
20 end
21 return S
```

La idea principal de este algoritmo es comprobar si para cada nodo de la solución, existe uno fuera de ella que mejora la función objetivo del problema, que en este caso es la mínima distancia. Por tanto, cuando no exista ningún nodo fue-

ra de la solución que mejore a algún nodo de la solución, el algoritmo terminará.

De manera detallada, el algoritmo comienza probando el primer nodo de la solución, i. De este, toma la mínima distancia con la solución (que es la misma que con las de sus nodos críticos) y posteriormente calcula la CL si el nodo i no estuviese en la solución. Si hay candidatos para añadir en la solución ($CL \neq \emptyset$), para cada candidato, calcula cual sería la mínima distancia con la solución sin el nodo i. Si hay alguna más grande (y mayor la mínima distancia de la solución), se intercambia este nodo por el nodo i. Finalmente, el algoritmo toma el siguiente nodo de la solución y vuelve a comenzar las comprobaciones.

5.4. Algoritmo Tabu Search

5.4.1. Definición e historia

El siguiente método que proponemos en este trabajo es un Tabu Search (Búsqueda Tabú). El algoritmo Tabu Search, al igual que el algoritmo GRASP, es un algoritmo metaheurístico utilizada para resolver problemas de optimización combinatoria. Sin embargo, en este algoritmo incluimos el concepto de memoria mediante un concepto clave: la lista tabú.

Antes de seguir con la explicación, repasamos el origen de este método. Los orígenes de este algoritmo datan de 1970, aunque no fue hasta 1989 cuando F. Glover oficializó tanto el nombre como la metodología. En [5], Glover presenta La Parte I de este estudio, en el cual se indica los principios básicos del Tabu Search, que van desde el proceso de memoria a corto plazo en el proceso de búsqueda, hasta los procesos de memoria a medio y largo plazo para intensificar y diversificar la búsqueda. Además, se incluyen estructuras de datos ilustrativas para implementar las condiciones de tabú (y los criterios de aspiración asociados) que sustentan estos procesos.

Posterior a este artículo, F. Glover presentó 'Tabu Search - Parte II', [6], la cual examina consideraciones más avanzadas, aplicando las ideas básicas a entornos especiales y delineando una estructura de movimiento dinámica para garantizar la finitud. También describe métodos de Tabu Search para resolver problemas de

programación entera mixta y ofrece un breve resumen de la experiencia práctica adicional, incluido el uso del Tabu Search para guiar otros tipos de procesos, como los de las redes neuronales.

Una vez repasado la historia de este método, prosigamos con la explicación. La lista tabú es la clave principal de este método y es lo que lo diferencia del método GRASP que hemos visto anteriormente. Durante el proceso de búsqueda local para optimizar las soluciones encontradas, la función de la lista tabú es almacenar soluciones que han sido obtenidas en iteraciones anteriores y obliga a algoritmo a no volver a ninguna de estas soluciones. Aunque los explicaremos con más detalle a continuación, hacer uso de la lista tabú presenta varias ventajas: una de estas ventajas es que se evitan ciclos indeseados; otra ventaja, y probablemente la más importante, es que escapa de los óptimos locales lo cual produce que el algoritmo explore diferentes zonas del espacio del problema.

Este algoritmo también se puede diferenciar en una fase de construcción, para partir de una solución inicial, y otra fase de búsqueda local, con en el objetivo de mejorar la solución. A continuación, explicaremos en profundidad como se define el algoritmo en cada fase.

5.4.2. Fase de Construcción

El objetivo de esta fase es construir una solución, con la mayor calidad posible, sobre la cual se realizará la búsqueda local para ir mejorando esta solución en cada iteración.

Para construir la solución, hemos propuesto 3 diferentes de formas:

1. **Voraz:** Se construye la solución aplicando un algoritmo voraz donde, la función voraz es:

$$g(i) = \min_{j \in S} d_{ij}$$

donde S representa la solución y d_{ij} representa la distancia los nodos i y j

Los nodos que se añaden a la solución son aquellos con mayor valor de esta función.

2. **Aleatoria:** Se construye la solución añadiendo los nodos de manera aleatoria hasta que no sea posible añadir otro más debido a la restricción del coste. Si la solución obtenida no es factible (ya que no se cumple la capacidad mínima), se vuelve a repetir el proceso.

3. **GRASP:** La solución inicial es construida por un algoritmo GRASP. La función voraz que emplearemos será la que mejor rendimiento tenga, lo cual se estudiará en el apartado de experimentos.

El pseudo-código que hemos desarrollado para esta fase de construcción es el siguiente:

Algorithm 3: Fase de Construcción Tabu Search

```
1  S ← ∅;
2  CL ← X;
3  i ← Random(CL);
4  S ← {i};
5  Evaluate(S);
6  CL ← Update(CL);
7  S ← SolutionConstruction(CL, n_c, method);
8  Evaluate(S);
9  CL ← Update(CL);
10 return S
```

Como en la fase de construcción del método GRASP, el algoritmo comienza añadiendo a la solución un nodo aleatorio. Una vez añadido, se actualiza la lista de candidatos, CL.

La línea siguiente del código es la más importante, puesto que es en la que crearemos la solución. Como se puede observar, la función que crea las soluciones, $SolutionConstruction$, disponemos de 3 parámetros: la lista de candidatos CL; el número de construcciones, n_c; y por último, el método con el cual crearemos la solución.

En cuanto a la lista de candidatos CL no es necesario detallar nada, puesto que es necesaria para saber que nodos pueden ser añadidos a la solución actual.

Respecto al número de construcciones, este parámetro indica cuantas veces se repetirá el método que crea las soluciones, y por tanto, cuantas soluciones se construirán. Al finalizar el número indicado de construcciones, tendremos en cuenta la que mayor calidad tenga.

El último parámetro que tenemos es el método que utilizaremos para construir la solución. Como explicamos previo al pseudo-código, tenemos 3 tipos distintos de métodos: si indicamos $method = Greedy$, la solución se construirá aplicando un algoritmo totalmente voraz; si indicamos $method = GRASP$, la solución se construirá de la misma manera que en el método GRASP; y si indicamos que $method = Random$, entonces la solución se creará añadiendo los nodos de forma aleatoria.

No podemos saber a ciencia cierta cual es la mejor combinación de parámetros, puesto que puede variar en función de la instancia que estemos considerando. Por esta razón, hemos realizado diferentes experimentos con las distintas combinaciones para concluir cual de ellas presenta mejores resultados.

5.4.3. Fase de Mejora

El objetivo de esta fase es mejorar la solución que se ha creado en la fase de construcción explicada anteriormente.

Para mejorar está solución, lo que haremos será aplicar un algoritmo de Búsqueda Local, de manera que intercambie nodos de la solución por nodos fuera de esta que mejoren la calidad de la solución. Sin embargo, no es exactamente igual que el algoritmo de Búsqueda local que hemos aplicado en el método GRASP.

En este método de Búsqueda Tabú, queremos que el algoritmo explore varias zonas del espacio del problema, cosa que no ocurriría si aplicamos la misma Búsqueda Local usada en el método GRASP, puesto que durante este búsqueda del óptimo local solo exploraríamos parte de la zona donde se encuentra este óptimo. Por tanto, para conseguir nuestro objetivo de explorar varias zonas del espacio del problema añadimos un concepto clave, introducido anteriormente: la lista Tabú.

Como hemos mencionado anteriormente, la lista Tabú recuerda las soluciones que se han ido obteniendo durante el proceso de mejora para no volver a ninguna de ellas durante unas iteraciones y poder así analizar diferentes soluciones, y por tanto explorar más espacio del problema. Para recordar estas soluciones, la lista Tabú almacena en cada iteración los nodos que han salido de la solución (en nuestro caso, solo será uno) y prohíbe a ese nodo volver a ser incluido en la solución durante unas iteraciones.

El número de iteraciones que el nodo está bloqueado, y por tanto no disponible para ser considerado para ser añadido a la solución, lo determina la longitud de la lista Tabú. La longitud de la lista Tabú, l, es un parámetro que se tiene que ajustar ya que a priori no podemos conocer cual es el valor que mejor se ajusta. Además, es importante realizar un buen ajuste debido a que la longitud de la lista Tabú es muy importante, ya que condicionará de gran manera el método: por un lado, una lista Tabú muy pequeña puede resultar en que el algoritmo repita las soluciones en ciclos y por tanto, no salir de esa zona de exploración, aunque tiene más facilidad para alcanzar el óptimo local de esa zona; por otro lado, una lista Tabú muy grande explorará varias zonas del espacio del problema pero probablemente la calidad de las soluciones no representen una gran mejora puesto que es difícil que se alcancen los óptimos locales de las zona que explora.

En cuanto al algoritmo de Búsqueda Local que hemos desarrollado, la idea es la misma que en el método anterior. Para cada nodo de la solución, se busca uno fuera de la solución y que no se encuentre en la lista Tabú y si existe un nodo que mejora a la solución, se intercambian dichos nodos, entrando el nodo que estaba en la solución en la lista Tabú. Sin embargo, si no hay ninguna solución mejor que en la que nos encontramos (y por tanto para ningún nodo de la solución existe otro fuera de ella y de la lista Tabú cuyo intercambio mejore la solución), el algoritmo no termina, si no que realiza el intercambio que menos empeore la solución. De esta manera, el algoritmo siempre está analizando varias soluciones y por tanto explorando el espacio del problema.

Es precisamente en este punto donde la Lista Tabú muestra su importancia, ya que evita que se repitan soluciones anteriores. Si se pudieran repetir soluciones anteriores, entraríamos en bucle en la que solo se repetirían dos soluciones, que serian el óptimo local y la solución previa a este, lo cual no nos interesa.

Finalmente, el algoritmo termina cuando se cumplen un cierto número iteraciones, el cual hay que fijar antes de que el algoritmo comienza. Este número de iteraciones máximo también es un parámetro que hay que ajustar y que también es importante: un valor pequeño de iteraciones puede provocar que la Búsqueda Tabú no explore el suficiente espacio del problema para obtener una solución de calidad; un valor elevado sin embargo, puede provocar que el método obtenga siempre las mismas soluciones, ya que es posible que explore las mismas zonas, además que el tiempo de ejecución incrementaría.

Continuando con el ejemplo de la cordillera, mientras que el método GRASP se situaba en un punto de una montaña, la escalaba hasta la cima y finalizaba, el método de Búsqueda Tabú se sitúa en un punto de la montaña y empieza a escalarla. Sin embargo, cuando no puede llegar más alto, baja por otro camino y comienza a escalarla por otra parte. En este proceso de bajar, el método puede cambiar de montaña y comenzar a escalar otra montaña.

El pseudo-código que hemos desarrollado para esta fase es la siguiente:

Algorithm 4: Fase de Mejora Tabú

```
1  TL ← ∅;
2  Shift ← True;
3  Move ← 0;
4  S_best ← S;
5  while Shift = True & Move ≤ Moves do
6      nodes_out ← NodeMinDist(S) ;        /* Calcula los nodos que
        tienen la menor distancia respecto de la solución */
7      S ← removeNode(node_out, S);
8      CL ← updateCL(S);
9      TL ← TL ∪ {node_out};
10     node_in, d*_min ← 0;
11     Candidates ← {i ∈ CL : i ∉ TL};
12     if Candidates ≠ ∅ then
13         for i ∈ Candidates do
14             d_cand ← MinDist(i, S);
15             if d_cand > d*_min then
16                 d*_min ← d_cand;
17                 node_in ← i;
18             end
19         end
20     end
21     else
22         Shift ← False;
23     end
24     if Shift then
25         S ← addNode(node_in, S);
26     end
27     if f(S) > f(S*) then
28         S* ← S;
29     end
30 end
31 return S_best
```

El código comienza creando la lista tabú sin ningún elemento. Posteriormente, inicializamos las variables $Shift$ y $Move$, las cuales son responsables de que

el bucle continúe iterando o finalice. Brevemente, la variable $Shift$ indica si hay algún cambio posible a realizar, ya que puede ocurrir el caso de que no haya nodos candidatos a entrar en la solución debido a la lista Tabú. Por tanto, al comenzar la fase de mejora esté variable toma el valor $True$. Por su parte la variable $Move$, indica número de soluciones que se han obtenido.

Ahora, nos disponemos a entrar en el bucle que irá cambiando la solución. Este bucle seguirá iterando siempre y cuando haya un cambio disponible y siempre que no se sobrepase el número de iteraciones máximas fijadas, indicado en el parámetro $Moves$.

Ya dentro del bucle, el algoritmo comienza buscando el nodo que será eliminado de la solución, aplicando para ello la función $NodeMinDist$ y el cual se asignará a la variable $node_{out}$. Para calcular dicho nodo, se calculan el nodo que tiene menor distancia con respecto a los nodos de la solución (si hubiera más de uno elegimos uno al azar) y se comprueba si para este nodo existe otro fuera de la solución por el cual se puede intercambiar de manera que la solución siga siendo compatible.

Por tanto, se elimina $node_{out}$ de la solución, se actualiza la lista de candidatos CL y se añade este nodo a la lista Tabú, TL. El siguiente paso es analizar los nodos fuera de la solución que puedan ser candidatos a entrar en la solución. Para ello, inicializamos los parámetros $node_{in}$ y $d*_{min}$ (que representa la distancia con la solución) a 0. Además, creamos una lista, $Candidates$, con los nodos que están en la lista de candidatos CL y no se encuentran en la lista Tabú TL.

Una vez que hemos definido los elementos necesarios, comenzamos con la Búsqueda Local: Primero, comprobamos que la lista $Candidates$ contiene algún nodo, puesto que si está vacía no se puede realizar ningún cambio y por tanto el algoritmo finaliza; en caso de que esta lista contenga al menos un nodo, para cada nodo hallamos la mínima distancia respecto a los nodos de la solución (en la cual hemos eliminado $node_{out}$) y aquel nodo cuya distancia sea mayor será el que entre en la solución, almacenándolo en la variable $node_{in}$. Si finalmente se ha realizado un cambio, y por tanto $Shift = True$, entonces se añade a la solución el nodo $node_{in}$.

Obtenida la nueva solución, el algoritmo comprueba si la solución S es mejor que la mejor solución hasta el momento, S_{best} y en caso afirmativo, se actualiza la mejor solución. Para finalizar, cuando el número de iteraciones sobre pasan al número de iteraciones fijadas, el algoritmo devuelve la mejor solución, S_{best}

5.5. Algoritmo SBTS

5.5.1. Definición e historia

El último método que vamos a desarrollar en este trabajo es un algoritmo SBTS (Solution Based Tabu Search). Al igual que el algoritmo de Tabu Search, se trata de un algoritmo metaheurístico para resolver problemas de optimización combinatoria. Aunque en este algoritmo también se incluye el concepto de memoria, la diferencian con el método Tabu Search anterior (y con los algoritmos Tabú Search en general) es la manera en la que utilizamos este concepto

El algoritmo SBTS es una de los algoritmos metaheurísticos más recientes, puesto que el primer algoritmo SBTS para resolver el problema de la dispersión data de 2017, presentado en [22] para resolver el problema de la dispersión con una función objetivo MinDiff. Si nos centramos en nuestro problema es incluso aún más reciente, puesto que el primer algoritmo SBTS adaptado al problema de la dispersión con restricciones de capacidad data de este año, 2023. El artículo del que hablamos [23], se publicó de manera online en marzo de 2023 y día de hoy, todavía no se ha publicado físicamente.

Dicho artículo comienza haciendo una repaso de la literatura de los algoritmos SBTS en cuanto al problema de la dispersión para posteriormente definir su método SBTS. Una vez explicado todas las componentes e ideas claves del método, lo compara con el estado del arte de los algoritmos empleados para resolver este problema, concluyendo en que dicho método mejora a dichos algoritmos

Una vez que construimos una solución y empleamos un algoritmo de Búsqueda Local para mejorar la solución, idealmente nos gustaría guardarnos las soluciones que hemos visitado. Sin embargo, guardarnos todas estas soluciones y comprobar si cada solución obtenida ha sido visitada es muy ineficiente desde

el punto de vista computacional, lo que provocaría un gran aumento del tiempo de ejecución del método. Para resolver este problema, una primera opción es guardarnos los últimos l nodos que han sido eliminados de la solución, almacenándolos en una lista, dando lugar al algoritmo Tabu Search. Aunque de esta manera evitamos analizar soluciones ya exploradas, también se reduce el número de soluciones a explorar. Veamos un ejemplo:

Supongamos que en una iteración pasamos de la solución $S_1 = [1,2,3,4]$ a la solución $S_2 = [1,2,3,5]$, y entonces añadimos el nodo 4 lista tabú. Supongamos que ahora pasamos de la solución $S_3 = [1,2,3,6]$ y por tanto añadimos el nodo 5 lista tabú. Como el nodo 4 sigue en la solución, no podremos volver a la solución S_1, pero tampoco podremos observar las posibles soluciones $[4,2,3,6]$, $[1,4,3,6]$ y $[1,2,4,6]$. Notar que no valdría que la lista tabú tuviera longitud 1 (y entonces solo almacenar el último nodo eliminado) ya que en ese caso sí que podría darse el caso de volver a la solución S_1

Sin embargo, como se explica en [23], el algoritmo SBTS sí que es capaz de recordar las soluciones que ha visitado sin tener que guardar todos los nodos que componen la solución en la memoria. Para ello, se emplean las denominadas funciones Hash. De manera resumida, ya que lo explicaremos más adelante, dada una tabla Hash y una solución, lo que consigue cada función Hash es mapear la solución a un índice de la tabla Hash, la cual solo toma valores 0 y 1. Por tanto, para comprobar si una solución ya ha sido visitada es aplicar las funciones Hash y comprobar el valor que obtenemos en la tabla Hash.

Para acabar con la definición del algoritmo, este también se caracteriza por la manera de explorar las vecindades de la solución, puesto que las vecindades son más flexibles, como por ejemplo a la hora de variar el tamaño en cada iteración. En contraste con el Tabu Search, en cada iteración las vecindades que exploramos solo son aquellas soluciones en las que cambiamos un nodo por otro, por lo que el tamaño (sin contar los nodos que son tabú) siempre es el mismo. En cambio, como proponen en [23], para cada iteración se proponen explorar las vecindades resultantes de añadir un nodo a la solución; de eliminar un nodo de la solución; y por último de intercambiar un nodo de la solución por otro fuera de esta.

En este trabajo solo se considera las vecindades resultantes de intercambiar nodos, pero en el artículo citado anteriormente se explica de manera detallada como actuaría el algoritmo en este caso.

Como en los métodos anteriores, también se pueden distinguir dos fases en este método: la fase de construcción y la fase de mejora.

5.5.2. Fase de Construcción

La fase de construcción para el método de SBTS es exactamente igual que la utilizado para el método Tabu Search: se emplea una de las tres formas explicadas en el método anterior (voraz, aleatoria y GRASP) para construir una solución a la cual aplicarle una búsqueda local en la fase de mejora.

Como tanto el pseudo-código, los parámetros y la explicación es igual, no incluiremos nada en este apartado.

5.5.3. Fase de Mejora

Al igual que los métodos anteriores, el objetivo de esta fase es realizar un búsqueda local por el espacio del problema partiendo de la solución obtenida en la fase de construcción par intentar mejorarla lo máximo posible.

Como en el método Tabu Search, queremos que el algoritmo no se quede estancado en el óptimo local más próximo a la solución de partida (como ocurre en el método GRASP), si no que explore diferentes zonas del espacio del problema. Para ello, en cada iteración se realiza un cambio en la solución, aunque la nueva solución sea peor que la anterior y además, la solución que ha sido cambiada se marca como Tabú para no volver visitarla. Para ello, no necesitamos guardar la solución completa, si no que lo haremos a partir de las funciones hash. Así, antes de cambiar la solución debemos comprobar que dicha solución no es tabú.

Expliquemos de manera detallada en que consisten estas funciones y tabla hash y como se utilizan para comprobar si una solución es tabú: Comenzamos definiendo la tabla hash, H como una matriz nula de 3 filas y c columnas. Este nú-

mero de columnas se podría considerar como un parámetro para el cual estimar su mejor valor, pero en este trabajo lo hemos fijado a 10^8. La tabla H es una tabla binaria (solo toma los valores 0 y 1), y en esta tabla será donde nos guardaremos las soluciones visitadas. Una vez inicializada la tabla H, consideramos las siguientes funciones hash, h_k con $k = 1,2,3$. Como hemos mencionado anteriormente, estas funciones sirven como un mapeo donde a una solución se le asigna una columna de H. Por tanto, si queremos marcar una solución, S, como tabú haríamos lo siguiente:

$$H[k, h_k(S)] \leftarrow 1 \quad k \in \{1,2,3\}$$

Y si queremos comprobar si una solución es tabú, tendriamos que verificar si se cumple la siguiente igualdad:

$$H[k, h_k(S)] = 1 \quad k \in \{1,2,3\}$$

Por tanto solo nos queda definir las funciones h_k. Estas se definen de la siguiente manera:

$$h_k(S) = (\sum_{i=1}^{n} w_k(i) \cdot x_i) \ \ mod \ \ c$$

donde $w_k(i) = w_k(i-1) + \beta_k + rand(\beta_k/2)$, $w_k(0) = \beta_k$, β_k $(k = 1,2,3)$ es un parámetro que toma los valores $(300,400,500)$ respectivamente, $rand(\beta_k/2)$ toma un número entero aleatorio entre 0 y $\beta_k/2$, $x_i = 1$ si $i \in S$ o $x_i = 0$ en caso contrario y c es el número de columnas de la tabla H.

Usando cada unos de estos elementos, conseguimos el objetivo de memorizar las soluciones visitadas sin tener que guardar las soluciones enteras, y entonces, memorizamos las soluciones con un coste computacional mucho más bajo.

El psuedo-código del algoritmo que hemos desarrollado es el siguiente:

Algorithm 5: Fase de Mejora SBTS

```
1  shift ← True;
2  move ← 0;
3  S_best ← S;
4  Hash_M ← 0_{3×10^8} ;                                  /* Matriz Hash */
5  Hash_f ← 0_{3×1} ;                                     /* Funciones Hash */
6  (Hash_M, Hash_f) ← HashFunctions(S, Hahs_M) ; /* Actualizamos la
     Matriz Hash */
7  isHashTabu ← True;
8  while move ≤ moves do
9      while isHashTabu == True & shift == True do
10         node_out ← CalculateNodeOut(S);
11         S ← removeNode(node_out, S);
12         CL ← update(CL);
13         if CL ≠ ∅ then
14             node_in ← CalculateNodeIn(CL, S);
15         end
16         else
17             shift ← False;
18         end
19         if shift == True then
20             S ← addNode(node_in, S);
21             (HashTabu, Hash_Maux) ← HashTabu(S, Hash_M);
22         end
23         if HashTabuSolution == True then
24             S ← S\{node_in} ∪ {node_out};
25             CL ← CL\{node_in};
26         end
27         else
28             Hash_M ← Hash_Maux;
29             isHashTabu ← False;
30         end
31     end
32     if f(S) > f(S_best then
33         S_best ← S;
34     end
35 end
36 return S_best
```

El algoritmo comienza inicializando las variables auxiliares que necesitamos para su correcto funcionamiento, como $shift$ que indica si hay algún cambio posible para el nodo escogido para salir de la solución; $move$ nos indica las veces que se ha cambiado de solución; $isHashTabu$ que nos indica si la solución obtenida al producir el intercambio es tabú; y por último las funciones hash y la matriz hash, $Hash_f$ y $Hash_M$ que se inicializan a 0.

Una vez inicializado estas variables, el algoritmo empieza a realizar movimientos y se detendrá cuando el número de movimientos realizados, $move$ sea mayor que un número prefijado de movimientos, $moves$. Este parámetro $moves$ se escogerá según el tiempo de ejecución del algoritmo. Ya dentro del bucle, lo primero que nos encontramos es otro bucle, el cual se ejecutará siempre que la solución obtenida más adelante sea Tabú y haya algún cambio disponible. En caso de que no sea Tabú, esta solución pasará a ser la nueva solución o sin embargo, si no hubiera ningún cambio disponible, el algoritmo finalizaría devolviendo dicha solución.

Para determinar que nodo será eliminado de la solución, el algoritmo hace uso de la función $CalculateNodeOut$, que tiene como parámetro de entrada la solución en esa iteración, S. Esta función, halla los nodos que tienen la menor distancia con la solución. En caso de que hubiera más de un nodo, se elige uno al azar. Para este nodo, se comprueba si hay un nodo fuera de la solución tal que al hacer el intercambio, la solución siga siendo compatible. En caso de que no existiera dicho cambio, se obtiene el siguiente nodo de la solución con la menor distancia y se repite el proceso hasta obtener un nodo de la solución que pueda ser intercambiado por otro fuera de ella.

Cuando el algoritmo ya tiene el nodo que va a ser eliminado de la solución, este es eliminado y se actualiza la lista de candidatos, CL, para dicha solución. El siguiente paso es encontrar que nodo será añadido a la solución. Para ello, aplicando la misma idea que en el método anterior, se obtiene el nodo con mayor distancia a la solución y que al ser añadido, la nueva solución sea factible. Posteriormente, se añade el nodo obtenido a la solución y se comprueba la si la nueva solución es Tabu. Para ello, se aplica la función $HashTabu$ la cual calcula las funciones hash de la nueva solución y devuelve las variables $HashTabuSolution$ que indica si la solución es tabú, y la nueva matriz Hash en caso de que no lo

fuera. Finalmente, comprobando el valor de $HashTabuSolution$ se determina si la solución es Tabú (y por tanto hay que hallar una nueva) o por el contrario si no lo es, y por lo tanto la solución obtenida es valida y se actualiza la matriz Hash.

Para finalizar, el algoritmo compara la solución obtenido con la mejor solución hasta el momento, S_{best}, y en caso de que la solución obtenida fuera de mejor calidad (i.e, la mínima distancia es mayor) se actualiza la mejor solución.

Cuando el número de movimientos ha excedido al número fijado, el algoritmo devuelve S_{best}

Capítulo 6

Experimentos

En este apartado, incluiremos todos los experimentos que hemos realizado en este estudio, tanto los experimentos preliminares (para optimizar los distintos parámetros de los métodos o hacer un primer análisis de que método es mejor) como el experimento final, en el cual comparamos los tres algoritmos propuestos para todas las instancias.

Las instancias que hemos utilizado para este estudio, las hemos obtenido del artículo [8]. Como se explica en el artículo anterior, Martínez Gavara et al. [11] propusieron un conjunto de referencia de 200 instancias para el Problema de la Dispersión Generalizada, denominado GDP. Los autores las generaron a partir de un subconjunto de 50 instancias Max-Min, con diferentes características de tamaño y tipo, seleccionadas de la librería MDPLIB.

Por tanto, se obtiene una instancia de GDP a partir de cada una de estas instancias al agregar una capacidad b_i y un costo k_i para cada elemento i. En particular, estos valores se generan de manera aleatoria con una distribución uniforme entre 1 y 1000, y $\frac{b_i}{2}$ y $2 \cdot a_i$ respectivamente.

Por último, se calcula la capacidad mínima requerida B como la capacidad total multiplicada por un factor φ_b de 0.2 o 0.3, así como el presupuesto máximo K que se calcula como la suma de todos los valores de costos multiplicados por un factor φ_k de 0.3 y 0.2.

El resultado final es un total de 120 instancias, de las cuales 40 tienen un tamaño

de 50 nodos, 40 tiene un tamaño de 150 nodos y 40 tienen un tamaño de 500 nodos. Además, dentro cada categoría de tamaño, las categorías se clasifican según los factores φ_b y φ_k, desglosándose estas 40 instancias en grupos de 10 instancias.

Es importante destacar la importancia de estos factores, ya que influyen directamente en la dificultad para resolver este problema. Por ejemplo, una instancia cuyo factor $\varphi_b = 0.3$ y $\varphi_k = 0.2$ está proponiendo un problema donde se tiene más capacidad mínima y un presupuesto más bajo, lo cual restringe mucho el problema. Sucede lo contrario para las instancias con $\varphi_b = 0.2$ y $\varphi_k = 0.3$, las cuales proponen problemas con menor capacidad mínima y mayor coste máximo.

Por tanto, tener instancias con mayor o menor dificultad para su resolución es muy útil, ya que nos permite ver como se comportan nuestros algoritmos para distintas situaciones.

Antes de continuar con los experimentos, es importante explicar cómo analizaremos los resultados que obtengamos de cada método a comparar. Para ello, recopilaremos los datos en una tabla agrupada por la dificultad de las instancias y donde para cada método a comparar y para cada dificultad calcularemos las siguientes medidas:

- $\#Mejor$**:** Número de veces que el método genera la mejor solución respecto de los otros métodos.
- $\%dev$**:** Medida que calcula como de alejado está la mejor solución obtenida por el método respecto a la mejor solución obtenida entre los 3 métodos. Esta medida se calcula de la siguiente manera:

  $$\%dev = \frac{s_{best} - s_a}{s_{best}}$$

  donde s_{best} es la mejor solución obtenida entre los 3 métodos y s_a la mejor solución del método.

  Puesto que en el conjunto de entrenamiento tenemos instancias de diferentes tamaños, primero debemos calcular esta medida para cada tamaño y posteriormente se hace la media.

Por ejemplo, si comparamos el método GRASP con el método Tabu Search, y disponemos de 12 instancias, una de cada tamaño y dificultad, para cada tamaño tendremos un valor en la celda GRASP-$\varphi_b = 0.2$, $\varphi_k = 0.2$, es decir, tendremos 3 valores de $\%dev$ para esa celda. Por tanto, el resultado final de esa celda será la media de esos 3 valores

- $Tiempo(s)$**:** Medida que calcula los segundos en segundos cada método en ejecutarse. Al igual que antes, para calcular esta medida realizamos la media por tamaño

Por último, destacar que todas las instancias, simulaciones y resultados están anexados al final del documento y son totalmente reproducibles con un entorno de Python.

6.1. Experimentos preliminares

Una vez explicadas las instancias, vamos a comentar los distintos experimentos que hemos realizado.

6.1.1. Experimentos sobre el método GRASP

Para estos experimentos previos, usaremos un conjunto de entrenamiento compuesto por el 10% de las instancias, es decir, un total de 12 instancias. Estas instancias están repartidas equitativamente por el tamaño del problema y por la combinación de los factores φ_b y φ_k, i.e, hay 4 instancias de cada tamaño donde cada una de ellas tiene un dificultad diferente.

Los experimentos que vamos a realizar en este apartado son los siguientes:

1. **Ajuste de los parámetros β_d, β_c y β_k:** Una de las posibles funciones voraces que habíamos considerado para construir la solución inicial era la función $DistCapCostGreedyFunction$.

   Dicha función utiliza estos 3 parámetros, los cuales regulaban la importancia que dicha función voraz le dedicaba a buscar soluciones con mejor distancia, con mayor capacidad o con menor coste. Por tanto, el experimento

consistirá en decidir cuales de la siguiente combinación de parámetros es mejor:

$$[\beta_d, \beta_c y \beta_k] = [1/3, 1/3, 1/3],\ [1/2, 1/4, 1/4],\ [3/4, 1/8, 1/8]$$

2. **Función voraz a emplear:** Como su nombre indica, este experimento consiste decidir cual de las 3 posibles funciones voraces será la que utilizaremos para construir la solución inicial. Las tres posibilidades son:
   - **BestDistGreedyFunction**
   - **DistCapCostGreedyFunction,** con los parámetros ajustados.
   - **RetroactiveGreedyFunction**
3. **Ajuste del parámetro del umbral, α:** Este experimento consiste en decidir cual de los siguientes valores de α obtiene mejores resultados:

$$\alpha = 0.4, 0.5, 0.6, 0.7$$

Ajuste de los parámetros β_d, β_c y β_k

Los resultados que hemos obtenido han sido los siguientes

φ_b	φ_k	[1/2, 1/4, 1/4]			[1/3, 1/3, 1/3]			[3/4, 1/8, 1/8]		
		#Mejor	% dev	Tiempo (s)	#Mejor	% dev	Tiempo (s)	#Mejor	% dev	Tiempo (s)
0.2	0.2	1	0,49%	163,36	2	0,56%	144,97	0	1,82%	234,31
0.2	0.3	1	0,40%	238,96	0	1,14%	234,33	2	0,85%	242,58
0.3	0.2	1	1,14%	1010,95	2	0,53%	993,14	0	1,86%	1004,04
0.3	0.3	1	2,49%	240,73	1	0,79%	239,04	1	1,17%	282,84
Resumen		4	1,13%	413,50	5	0,76%	402,87	3	1,43%	440,94

Tabla 6.1.1: Experimento para ajustar los parámetros β_d, β_c, β_k

A primera vista, podemos observar que la combinación de $\beta's$ que más veces produce la mejor solución es $[1/3, 1/3, 1/3]$, con un total de 5 (sobre 12). Además, es la combinación con menor tiempo y $\%dev$, lo cual indica que las soluciones que obtiene están muy a cercanas a la mejor solución de cada momento y que genera estas soluciones de manera más rápida.

Analizando más en profundidad, esta combinación de parámetros obtiene sus mejores resultados para las instancias con mayor dificultad, lo cual tiene sentido porque dicha combinación tiende a buscar soluciones donde la distancia, la capacidad y el coste esté equilibrado, haciendo que las soluciones generadas sean factibles, a diferencia de las otras combinaciones.

Aunque tienen rendimientos parecidos, podríamos decir que la siguiente mejor combinación es $[1/2, 1/4, 1/4]$, puesto que siempre consigue obtener una mejor solución para cada dificultad. Además, obtiene el $\%dev$ más bajo para la dificultad más fácil, lo cual era de esperar puesta que esta combinación tiende a producir soluciones iniciales con mejores distancias.

Por último, la peor combinación de parámetros es $[3/4, 1/8, 1/8]$. Vemos que es la combinación con mayor tiempo de media y mayor $\%dev$, lo cual se debe a que esta combinación está muy cerca de ser una construcción voraz. Esto se puede ver en los resultados que obtiene en las dificultad más fácil, dónde es el mejor método, mientras que para la más difícil es el peor método ya que no consigue generar ninguna mejor solución.

Por ello, tras analizar los resultados del experimento, la mejor combinación de parámetros para la función voraz $DistCapCostGreedyFunction$ es $\beta_d = 1/3$, $\beta_b = 1/3$ y $\beta_k = 1/3$

Experimento previo sobre la función voraz:

Los resultados que hemos obtenido han sido los siguientes:

φ_b	φ_k	Dist			Mix			Retroactive		
		#Mejor	% dev	Tiempo (s)	#Mejor	% dev	Tiempo (s)	#Mejor	% dev	Tiempo (s)
0.2	0.2	2	0,39%	44,06	0	2,88%	56,44	1	0,02%	61,79
0.2	0.3	1	0,55%	80,42	0	1,04%	103,73	2	0,21%	78,88
0.3	0.2	0	6,40%	89,02	2	1,08%	55,24	1	3,87%	89,65
0.3	0.3	1	0,56%	75,51	1	0,60%	100,87	1	0,71%	166,45
Resumen		4	1,98%	72,25	3	1,40%	79,07	5	1,20%	99,19

Tabla 6.1.2: Rendimiento de las distintas funciones voraces

donde $Dist$ representa a la función $BestDistGreedyFunction$, Mix representa la función $DistCapCostGreedyFunction$ optimizada, y donde $Retroactive$

representa la función $RetroactiveGreedyFunction$.

En primera instancia, observamos que la función voraz que más veces obtiene la mejor solución es la función $Retroactive$, con 5 de 12. Se puede observar como esta función siempre obtiene al menos una mejor solución para cada una de las dificultades, a diferencias de las otras funciones voraces, por lo que este método GRASP usando esta función voraz es muy constante.

Analizando en profundidad, esta función obtiene mejores resultados cuando la dificultad del problema es más fácil, lo cual tiene sentido puesto que dicha función comienza dando total importancia a buscar la mejor distancia posible y a menudo que falla le va restando importancia. Por último, destacar que también es la función voraz con menos $\%dev$, destacando en instancias con $\varphi_b = 0.2$ y $\varphi_k = 0.2$, donde obtiene un valor de 0.02 %. Sin embargo, también se aprecia que este método es el que más tiempo consume, especialmente en la última dificultad.

La siguiente función con mejores resultados es la función $Dist$, que es la que más se centra en buscar siempre la mejor distancia comparado con las otras dos funciones voraces. Esto se ve reflejado en sus resultados, ya que en los problemas de mayor dificultad no consigue producir soluciones de calidad (para la mayor dificultad nunca obtiene la mejor solución) y en cambio para problemas con menor dificultad sí que consigue producir alguna mejor solución.

Por ultimo, en cuanto a la función Mix, con los parámetros $[1/3, 1/3, 1/3]$ que hemos elegido anteriormente, obtenemos que es la función que peores resultados obtiene, puesto que para las dificultades más fáciles no consigue producir ninguna mejor solución y además tiene unos valores de $\%dev$ elevados. En cambio, cuanto mayor es la dificultad mejores datos obtiene, obteniendo su mejor resultado en la instancia con mayor dificultad. Esto se debe a que este método es conservador, y da la misma importancia a obtener la mejor distancia, la mayor capacidad y el menor coste. Por ello, en problemas con menor dificultad las soluciones que producen no son de la suficiente calidad.

Por tanto, después de haber estudiado los resultados, la función voraz con la que construiremos el método GRASP será la función $RetroactiveGreedyFunction$.

Experimento previo para optimizar α:

Los resultados obtenidos han sido los siguientes:

φ_b	φ_k	0.4			0.5			0.6			0.7		
		#Mejor	% dev	Tiempo (s)	#Mejor	% dev	Tiempo (s)	#Mejor	% dev	Tiempo (s)	#Mejor	% dev	Tiempo (s)
0.2	0.2	1	1,15%	142,88	0	1,38%	134,48	1	0,45%	136,22	1	1,17%	130,38
0.2	0.3	1	1,47%	215,91	0	1,46%	211,67	1	0,72%	210,53	1	1,27%	211,36
0.3	0.2	0	4,97%	985,30	0	2,96%	950,49	1	1,98%	846,63	2	1,08%	770,13
0.3	0.3	1	0,79%	217,19	1	0,63%	214,62	0	1,77%	217,30	1	1,03%	212,54
Resumen		3	2,10%	390,32	1	1,61%	377,81	3	1,23%	352,67	5	1,13%	331,10

Tabla 6.1.3: Ajuste del parámetro α

Observamos que el valor de α que produce más veces la mejor solución es $\alpha = 0.7$, con un total de 5 veces sobre 12. Esta opción es la más ambiciosa para obtener soluciones con la mejor distancia posible, ya que recordemos que cuanto más alto era el valor de α, más difícil es para los candidatos entrar en la lista de candidatos restringida, i.e, está lista de candidatos restringida estará formada por los nodos que más mejoran a la solución.

Para este valor de α, siempre se obtiene al menos una mejor solución para todo las dificultades, obteniendo el mejor resultado para las instancias con menor dificultad, ya que como comentábamos antes, al estar más restringida la RCL, la construcción es cada vez más parecido a un algoritmo voraz. Además, este método es el que menor $\%dev$ y menor tiempo presenta, por lo que es sin duda la mejor opción.

Los dos siguientes mejores valores para α son 0.4 y 0.6, donde ambos producen la mejor solución 3 veces. Observamos que ambos valores tienen rendimientos parecidos, donde para una de las dificultades no producen ninguna mejor solución. Sin embargo, podríamos decir que el valor $\alpha = 0.6$ es mejor opción debido a que tarda menos tiempo y a que el $\%dev$ del valor $\alpha = 0.4$ es muy elevado

Por último, el valor que peores resultados obtiene es $\alpha = 0.5$, donde solo obtiene alguna mejor solución para una dificultad.

Tras el análisis de este experimento, se puede concluir sin ninguna duda que $\alpha = 0.7$ es el valor que mejor rendimiento presenta y que por tanto, emplearemos en

el método GRASP.

Conclusiones método GRASP

Después de haber estudiado todos estos experimentos previos realizados para optimizar el método GRASP, concluimos que el método GRASP más optimizado empleará la función voraz $RetroactiveGreedyFunction$ para construir las soluciones y usará un valor de $\alpha = 0.7$ para regular la lista restringida de candidatos.

6.1.2. Experimentos sobre el método Tabu Search

El único experimento que realizaremos para optimizar nuestro método Tabu Search será para decidir que construcción usaremos para obtener la solución inicial sobre la que partirá el método Tabu. Aunque no se ha realizado en este trabajo, también se podría realizar un experimento para optimizar la longitud de la lista tabú. Sin embargo, en este trabajo hemos considerado que la longitud de la lista tabú es el 10 % del tamaño de la instancia.

Volviendo al experimento sobre las construcciones, las posibilidades son:

1. Construcción voraz, utilizando la función $RetroactiveGreedyFunction$
2. Construcción aleatoria
3. Construcción GRASP optimizada, 50 repeticiones para las instancias de tamaño 50 y 150; y 5 repeticiones para las instancias de tamaño 500.

Así, los resultados que obtenemos de este experimento han sido los siguientes:

φ_b	φ_k	GRASP			Voraz			Aleatorio		
		#Mejor	% dev	Tiempo (s)	#Mejor	% dev	Tiempo (s)	#Mejor	% dev	Tiempo (s)
0.2	0.2	2	0,71 %	38,18	1	3,44 %	18,42	0	4,35 %	17,64
0.2	0.3	2	0,80 %	50,78	1	1,98 %	17,78	0	4,15 %	17,31
0.3	0.2	1	0,92 %	44,78	1	3,29 %	24,46	1	11,56 %	22,57
0.3	0.3	2	0,43 %	54,93	1	1,26 %	17,57	0	3,80 %	17,60
Resumen		7	0,71 %	47,17	4	2,49 %	19,56	1	5,96 %	18,78

Tabla 6.1.4: Rendimiento de las construcciones aplicadas al Tabu

Claramente, observamos que los mejores resultados son obtenidos cuando el método Tabu Search parte de una solución construida por el método GRASP optimizado. Los resultados hablan por sí solos: la construcción GRASP es mejor que las otras construcciones en 3 de las 4 dificultades posibles, obteniendo al final que usando este método, se construye la mejor solución 7 veces de 12. Además esta construcción también tiene el $\%dev$ más bajo, lo cual indica que las soluciones que genera son de buena calidad. Sin embargo, este método es mucho más costoso computacionalmente, doblando en tiempo a los demás algoritmos.

La segunda mejor construcción, aunque alejada de la primera, se trata de la construcción voraz que, salvo en una dificultad, produce al menos una mejor solución. Precisamente, en la dificultad en la que no logra producir ninguna mejor solución es en la más difícil ya que, como hemos comentado antes, las soluciones que construye no suelen ser factibles debido a que dicha construcción solo se centra en buscar la mejor distancia.

Como era de esperar, la construcción aleatoria es la que peor rendimiento ofrece de las 3. Esto se debe a que dicha construcción no genera soluciones de suficiente calidad y el Tabu Search precisamente no se centra en buscar la mejor solución local a la solución inicial, si no que trata de explorar todo el espacio. Además, se puede apreciar que su $\%dev$ es muy elevado, lo que nos indica que las soluciones que se generan están muy alejadas de las mejores soluciones obtenidas.

Por tanto, a modo de conclusión, la construcción que emplearemos para generar la solución sobre la que partirá el método Tabú será la construcción GRASP.

6.2. Experimentos finales

Experimento final para decidir el mejor método

Una vez que tenemos todos los métodos optimizados, estamos en disposición de comparar los tres métodos que se han propuesto en este trabajo. Para decidir que método es mejor, aplicaremos los tres métodos a todas las instancias que tenemos disponibles. Para que sea una comparación justa, hemos adaptado las repeticiones de cada método según las instancias para que todos computen

un tiempo parecido. Por tanto, tenemos lo siguiente:

- **Instancias de tamaño** 50

  1. GRASP optimizado con 150 repeticiones
  2. Tabu Search con 200 movimientos, partiendo de una solución GRASP optimizado construida 100 veces
  3. SBTS con 400 movimientos, partiendo de una solución GRASP optimizado construida 100 veces

- **Instancias de tamaño** 150

  1. GRASP optimizado con 100 repeticiones
  2. Tabu Search con 200 movimientos, partiendo de una solución GRASP optimizado construida 50 veces
  3. SBTS con 400 movimientos, partiendo de una solución GRASP optimizado construida 50 veces

- **Instancias de tamaño** 500

  1. GRASP optimizado con 6 repeticiones
  2. Tabu Search con 100 movimientos, partiendo de una solución GRASP optimizado construida 3 veces
  3. SBTS con 400 movimientos, partiendo de una solución GRASP optimizado construida 3 veces

Además, puesto que se trata del último experimento, hemos sustituido la medida $\%dev$ por la medida #Mejor SoA, la cual representa el número de veces que la solución construida supera a la solución del estado del arte actual. Sin embargo, solo disponemos de los resultados del estado del arte para las instancias de tamaño $n = 500$, por lo que solo podemos comparar nuestros resultados para un total de 40 instancias.

Los resultados que obtenemos son los siguientes:

φ_b	φ_k	GRASP			TABU			SBTS		
		#Mejor	#Mejor SoA	Tiempo (s)	#Mejor	#Mejor SoA	Tiempo (s)	#Mejor	#Mejor SoA	Tiempo (s)
0.2	0.2	5	1	35,23	13	4	25,41	12	5	43,78
0.2	0.3	11	4	55,00	9	1	34,73	10	1	54,30
0.3	0.2	0	0	45,70	10	0	32,91	20	1	50,02
0.3	0.3	6	3	58,98	10	3	37,93	14	4	58,01
Resumen		22	8	48,73	42	8	32,74	56	11	51,53

Tabla 6.2.1: Rendimiento de los métodos GRASP, TS y SBTS

Observamos que el método que más veces construye la mejor solución es el método SBTS, con un total de 56 veces sobre 120. Además, también es el método que más veces construye una mejor solución que la del estado del arte, con un total de 11 veces sobre 40. Más en profundidad, se observa que tiene un rendimiento regular siendo siempre el mejor o el segundo método para todas las dificultades, obteniendo el mejor rendimiento para la mayor dificultad. Esto se debe a que dicho método también es capaz de explorar el espacio de soluciones y buscar el máximo local de la zona, de manera que al encontrar el máximo local, el método sigue explorando otra soluciones.

El siguiente método con mejor rendimiento, es el método Tabu Search. Aunque obtiene peores resultados que el método SBTS, también es un método muy regular, logrando al menos la mejor solución para alguna dificultad. Este método consigue mejorar a los resultados del estado del arte en un total de 8 instancias, lo cual es un gran resultado. Además, este método es el que menor tiempo tarda en ejecutarse, casi 20 segundos menos que el método $SBTS$

Por último, el método que peores resultado obtiene es el método GRASP. Observamos que para la dificultad más difícil no logra producir ninguna mejor solución. Sin embargo, para la dificultad más fácil sí que consigue ser el método que mejor rendimiento obtiene. Por como hemos desarrollado el método GRASP, esto es normal, puesto que para las instancias más difíciles no consigue obtener soluciones factibles en las primeras instancias, causando por tanto que la función voraz se centre más en obtener una solución factible que en una solución de calidad. Por este mismo razonamiento, para las instancias de menor dificultad, el método sí que logra obtener soluciones factibles en las primeras instancias y por tanto, las soluciones son de alta calidad.

Dado los resultados obtenidos y su correspondiente explicación, llegamos a la conclusión que el mejor método de los que hemos desarrollado para resolver el

problema de la dispersión y el coste, es el método SBTS, seguido por el método Tabu Search y por último, el método GRASP. Sin embargo, esto no quiere decir que el método GRASP sea un mal método, ya que los métodos SBTS y Tabu Search parten de una solución creada por dicho GRASP.

Esto hace que ambos métodos puedan explorar el espacio del máximo local obtenido por el GRASP, de manera que si la zona tiene cerca otras zonas con mejores máximos locales, estos dos métodos explorarán dicha zona y, en la mayoría de los casos, mejoraría la solución de partida. Además, como se puede observar en la tabla, el método GRASP consigue mejorar a los resultados del estado del arte las mismas veces que el método Tabu Search, lo cual nos indicia que el método tiene un buen rendimiento

En resumen, la comparación resultante entre los métodos y los resultados del estado del arte es la siguiente:

Método	**#Mejor SoA**
GRASP	**8**
Tabu Search	**8**
SBTS	**11**
Estado del arte	**13**

Tabla 6.2.2: Comparación métodos vs estado del arte

De las 40 instancias de las que disponemos datos, en más de la mitad nuestros métodos mejoran a los resultados del estado del arte, exactamente en 27 instancias. Analizando por métodos, destacamos que ningún método consigue tener más mejores soluciones que las soluciones del estado del arte, aunque el método SBTS está cerca de ello, con 11 mejores soluciones frente a las 13 del estado del arte. Esto, además, tiene sentido con los resultados obtenidos, puesto que el método SBTS tenía mejor rendimiento que los demás.

Puesto que el método SBTS es el que mejor rendimiento tiene, hemos realizado una comparación con los resultados de dicho método frente a los resultados del estado del arte, obteniendo lo siguiente:

Método	**#Mejor SoA**
SBTS	**22**
Estado del arte	**18**

Tabla 6.2.3: Método SBTS vs Estado del Arte

Observamos que el método mejora a los resultados del estado del arte en más de la mitad de las instancias, lo que significa que el método que hemos desarrollado es completamente funcional e incluso, llega a mejorar a algunos de los métodos actuales.

Por ello, podemos concluir que los métodos desarrollados tiene un gran rendimiento, siendo el método SBTS el referente principal, y posiblemente optimizando tanto el algoritmo como algún parámetro, se podrían obtener unos resultados incluso mejores, los cuales serían capaces de superar a los que actualmente tiene el estado del arte.

Capítulo 7

Conclusiones

En este trabajo, hemos estudiado el campo de la optimización a través de la aplicación de algoritmos metaheurísticos, específicamente GRASP y Tabu, para abordar el desafiante e importante problema de la máxima dispersión con restricciones de capacidad y coste. Para entender el porqué de su importancia, hemos realizado un recorrido histórico del problema de la dispersión, destacando su relevancia a lo largo del tiempo y las diferentes perspectivas que se han aplicado en su resolución.

Además, basándonos en algunos artículos sobre este problema, hemos diseñado y evaluado tres métodos metaheurísticos diferentes, que son un algoritmo GRASP, un algoritmo Tabu Search y un algoritmo SBTS.

Con los experimentos realizados, hemos comprobado que el método SBTS supera a los otros dos y además, en varias ocasiones mejora a los resultados existentes en relación con este problema. Gracias a estos logros, en un futuro se podría plantear el uso de estos métodos para resolver problemas complejos en diversas áreas.

Bibliografía

[1] Ağca, S., Eksioglu, B., Ghosh, Jay B. Lagrangian solution of maximum dispersion problems. *Naval Research Logistic*, 47(2):97–114, 2000. URL: http://refhub.elsevier.com/S0377-2217(21)00654-8/sbref0037.

[2] David Sayah, Stefan Irnich. A new compact formulation for the discrete p-dispersion problem. *European Journal of Operational Research*, 256(1):62–67, 2017. URL: https://doi.org/10.1016/j.ejor.2016.06.036.

[3] Erhan Erkut. The discrete p-dispersion problem. *European Journal of Operational Research*, 46(1):48–60, 1990. URL: http://refhub.elsevier.com/S0377-2217(21)00654-8/sbref0020.

[4] Jay B. Ghosh. Computational aspects of the maximum diversity problem. *Operations Research Letters*, 19(4):175–181, 1996. URL: http://refhub.elsevier.com/S0377-2217(21)00654-8/sbref0027.

[5] F. Glover. Tabu search - part i. *ORSA Journal On Computing*, 1(3):190–206, 1989. URL: https://doi.org/10.1287/ijoc.1.3.190.

[6] F. Glover. Tabu search - part ii. *ORSA Journal On Computing*, 2(1):4–32, 1990. URL: https://doi.org/10.1287/ijoc.2.1.4.

[7] Glover, F., Kuo, Ching-Chung, Dhir, Krishna S. Good solutions to discrete noxious location problems via metaheuristics. *Annals of Operations Research*, 19(11):696–701, 1998. URL: http://refhub.elsevier.com/S0377-2217(21)00654-8/sbref0030.

[8] Isaac Lozano-Osorio, Anna Martínez-Gavara, Rafael Martí, Abraham Duarte. Max–min dispersion with capacity and cost for a practical location problem. *Expert Systems with Applications*, 200, 2022. URL: https://www.sciencedirect.com/science/article/pii/S0957417422003384.

[9] Rex K. Kincaid. Good solutions to discrete noxious location problems via metaheuristics. *Annals of Operations Research*, 40(1-4):265–281, 1992. URL: http://refhub.elsevier.com/S0377-2217(21)00654-8/sbref0037.

[10] Michael J. Kuby. Programming models for facility dispersion: the p-dispersion and maxisum dispersion problems. *Mathematical and Computer Modelling*, 10(10):792, 1988. URL: http://refhub.elsevier.com/S0377-2217(21)00654-8/sbref0040.

[11] Martínez-Gavara, A., Corberán, T. & Martí, R. Grasp and tabu search for the generalized dispersion problem. *Expert Systems with Applications*, 173, 2020. URL: https://www.sciencedirect.com/science/article/pii/S0957417421001445.

[12] N. Mladenović & P. Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100, 1997. URL: http://refhub.elsevier.com/S0957-4174(22)00338-4/sb26.

[13] Parreño, F., Álvarez-Valdés,. R & Martí, R. Measuring diversity. a review and an empirical analysis. *European Journal of Operational Research*, 289(2):515–532, 2021. URL: http://refhub.elsevier.com/S0377-2217(21)00654-8/sbref0058.

[14] Shier. D. R. A min-max theorem for p-center problems on a tree. *Transportation Science*, 11(3):243–252, 1977. URL: http://refhub.elsevier.com/S0377-2217(21)00654-8/sbref0070.

[15] R. Chandrasekaran, A. Daughety. Location on tree networks: P-centre and n-dispersion problems. *Mathematics of Operation Research*, 6(1):50–57, 1981. URL: http://refhub.elsevier.com/S0377-2217(21)00654-8/sbref0012.

[16] Rafael Martí, Anna Martínez-Gavara, Jesús Sánchez-Oro. The capacitated dispersion problem: an optimization model and a memetic algorithm. *Memetic Computing*, 13:131–146, 2021. URL: https://doi.org/10.1007/s12293-020-00318-1.

[17] Rafael Martí, Anna Martínez-Gavara, Sergio Pérez-Peló, Jesús Sánchez-Oro. A review on discrete diversity and dispersion maximization from an or perspective. *European Journal of Operational Research*, 299:795–813, 2021. URL: http://refhub.elsevier.com/S0957-4174(22)00338-4/sb23.

[18] Resende, M. G., Martí, R., Gallego, M., Duarte, A. Grasp and path relinking for the max–min diversity problem. *Computers & Operations Research*, 37(3):498–508, 2010. URL: http://refhub.elsevier.com/S0377-2217(21)00654-8/sbref0065.

[19] Rosenkrantz, D. J., Tayi,. G. K. & Ravi, S. S. Facility dispersion problems under capacity and cost constraints. *Journal of Combinatorial Optimization*, 4(1):7–33, 2000. URL: http://refhub.elsevier.com/S0377-2217(21)00654-8/sbref0066.

[20] Kenneth Sörensen and Fred W. Glover. *Metaheuristics*, pages 960–970. Springer US, Boston, MA, 2013. doi:10.1007/978-1-4419-1153-7_1167.

[21] Thomas A. Feo, Mauricio G. C. Resende . Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995. URL: https://doi.org/10.1007/BF01096763.

[22] Yang Wang, Qinghua Wu, Fred Glover. Effective metaheuristic algorithms for the minimum differential dispersion problem. *European Journal of Operational Research*, 258(3):829–843, 2017. URL: https://www.sciencedirect.com/science/article/pii/S0377221716308694.

[23] Zhi Lu, Anna Martínez-Gavara, Jin-Kao Hao, Xiangjing Lai. Solution-based tabu search for the capacitated dispersion problem. *Expert Systems with Applications*, 223, 2023. URL: https://www.sciencedirect.com/science/article/pii/S0957417423003573.

Capítulo 8

Anexo

En este capítulo, adjuntamos todos los códigos de los métodos desarrollados, todas las simulaciones realizadas para los distintos experimentos y todos los resultados sobre los que están basadas las tablas incluidas en el documento.

Recordar que todos las simulaciones pueden ser totalmente reproducibles en un entorno de Python. Para acceder a toda la información anterior esta disponible en el siguiente GitHub, al cual se puede acceder clickando en el siguiente enlace:

https://github.com/Nachetebm10/TFM-Min-Max