Universidad de Castilla-La Mancha

Escuela Superior de Ingeniería Informática

Trabajo Fin de Grado

Grado en Ingeniería Informática

Tecnología Específica de

Computación

Estudio de heurísticos sencillos para problemas de diversidad

Mª Mercedes Guijarro Gil

Junio, 2024

Trabajo Fin de Grado

Grado en Ingeniería Informática

Tecnología Específica de

Computación

Estudio de heurísticos sencillos para problemas de diversidad

Autor: Mª Mercedes Guijarro Gil

Tutor: Francisco Parreño Torres

Tutor: Juan Ángel Aledo Sánchez

Junio, 2024

“Viaje antes que destino,
Fuerza antes que debilidad,
Vida antes que muerte.”

Declaración de Autoría

Yo, Mª Mercedes Guijarro Gil con DNI 04626665P, declaro que soy el único autor del trabajo fin de grado titulado “Estudio de heurísticos sencillos para problemas de diversidad” y que el citado trabajo no infringe las leyes en vigor sobre propiedad intelectual y que todo el material no original contenido en dicho trabajo está apropiadamente atribuido a sus legítimos autores.

Albacete, a 13 de junio de 2024

Fdo: Mª Mercedes Guijarro Gil

Resumen

Este trabajo se enfoca en el desarrollo y evaluación de heurísticas simples para resolver problemas de diversidad, entendida como la representación equitativa de diferentes elementos o características dentro de un conjunto.

Se proporciona una introducción exhaustiva a los conceptos clave relacionados con la optimización, la complejidad y las metodologías de resolución de problemas utilizadas. Además, se argumenta la necesidad de adoptar enfoques heurísticos para abordar la diversidad de manera eficiente.

Se describe en detalle el diseño experimental utilizado, incluyendo los algoritmos desarrollados y los conjuntos de datos empleados. Se presentan los resultados de una serie de experimentos, donde se comparan los diferentes heurísticos propuestos utilizando técnicas estadísticas adecuadas.

Las conclusiones del estudio resumen los principales hallazgos y discuten sus implicaciones en el contexto de la optimización y la resolución de problemas de diversidad. También se plantean posibles vías de investigación futuras para ampliar el conocimiento sobre este tema y mejorar las técnicas heurísticas desarrolladas.

Agradecimientos

En el transcurso de este camino hacia la culminación de mi Trabajo de Fin de Grado, me encuentro profundamente agradecida por el apoyo y la ayuda de quienes han estado a mi lado en cada paso de este viaje.

En primer lugar, deseo expresar mi más sincero agradecimiento a mis tutores Juan Ángel Aledo y Francisco Parreño. Gracias por la ayuda y la paciencia para llevar a cabo el desarrollo de este proyecto. Agradezco especialmente vuestro tiempo y dedicación, siempre disponibles para resolver dudas y proporcionar la orientación necesaria. Sin vuestra guía, este trabajo no habría sido posible.

Agradezco a mi familia por el apoyo constante que me han brindado. A mis padres, quienes han sido mi ejemplo de sacrificio y dedicación, les debo un agradecimiento especial por creer en mí. Su confianza en mis capacidades me ha dado la fuerza necesaria para seguir adelante, incluso en los momentos más difíciles. A mi hermana, por ser mi confidente y mi apoyo incondicional. Gracias por estar ahí, con palabras de ánimo que han sido fundamentales para mantenerme motivada.

A mis amigos, en especial a Sandra y Miguel, quienes han compartido risas, consejos y momentos de desahogo, les expreso mi más profundo agradecimiento. Vuestra amistad ha sido una fuente inagotable de fortaleza y apoyo. Gracias por los momentos de distracción que aliviaron el estrés y por las palabras de ánimo que me ayudaron a seguir adelante cuando las dificultades parecían insuperables.

Y por último, y más importante, quiero agradecer a Pedro, sin el cual no habría llegado hasta aquí. Gracias por tu ayuda, comprensión y por haber estado a mi lado a lo largo de este camino. Tu afecto y apoyo incondicional han sido mi refugio en los momentos de agobio y mi inspiración para seguir adelante. Gracias por creer en mí y por tu paciencia. Tu presencia ha sido esencial para alcanzar esta meta, y no tengo palabras suficientes para expresar mi gratitud.

A todos, mi más sincero agradecimiento. Este logro es tanto mío como vuestro.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1 Introducción

El problema de la diversidad consiste en la selección de un conjunto de elementos que maximice ciertas características o propiedades entre ellos, como la diferencia, la distancia o la variedad. Este desafío tiene una relevancia significativa en la sociedad actual, ya que tiene diversas aplicaciones en áreas como la logística, planificación urbana y gestión de recursos, donde la optimización de la selección de elementos es fundamental.

Dentro de este problema se encuentra el problema de la máxima diversidad, un problema con numerosas investigaciones y una complejidad computacional muy alta [1][2][11]. El trabajo se centra en el estudio y desarrollo de algoritmos para una variante de este problema conocida como *Max-Min*, cuyo objetivo es encontrar buenas soluciones lo más cerca posible a la óptima en un tiempo aceptable.

1.2 Objetivos

El presente trabajo tiene como objetivo abordar el problema de diversidad, un desafío complejo en el ámbito de la optimización combinatoria. Dado que este problema está clasificado como *NP-Hard*, se necesitan soluciones eficientes y prácticas. En este contexto, se propone explorar y evaluar la efectividad de diversas técnicas heurísticas para optimizar la diversidad en conjuntos de elementos.

Para lograrlo, nos enfocaremos en los siguientes puntos:

1. Presentar el problema de diversidad como un desafío de optimización combinatoria y su clasificación como *NP-Hard*.
2. Describir los diferentes modelos de diversidad que existen, entrando más en detalle en el modelo elegido para este problema.
3. Comprender las técnicas heurísticas para abordar el problema, reconociendo sus ventajas y limitaciones.
4. Describir los conjuntos de datos utilizados y la implementación eficiente de los algoritmos seleccionados.
5. Aplicar técnicas estadísticas adecuadas para analizar y medir el funcionamiento de los algoritmos heurísticos.
6. Evaluar la eficiencia y eficacia de los algoritmos heurísticos mediante experimentos y casos de prueba usando varios conjuntos de datos de tamaño variable y configuraciones de parámetros.
7. Por último, proporcionar conclusiones basadas en los resultados obtenidos en el estudio, así como sugerencias para futuras investigaciones en este campo.

Las competencias de la rama de Computación que serán aplicadas durante el trabajo son las siguientes:

- **[CM3]** Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar, desarrollar e implementar aquella que garantice el mejor rendimiento de acuerdo con los requisitos establecidos.

- **[CM4]** Capacidad para conocer los fundamentos, paradigmas y técnicas propias de los sistemas inteligentes y analizar, diseñar y construir sistemas, servicios y aplicaciones informáticas que utilicen dichas técnicas en cualquier ámbito de aplicación.
- **[CM5]** Capacidad para adquirir, obtener, formalizar y representar el conocimiento humano en una forma computable para la resolución de problemas mediante un sistema informático en cualquier ámbito de aplicación, particularmente los relacionados con aspectos de computación, percepción y actuación en ambientes o entornos inteligentes.

1.3 Estructura del proyecto

El trabajo se estructura en 5 capítulos de la siguiente manera:

1. Capítulo 1: Introducción. Se presenta una introducción al problema de la diversidad, un desafío en el ámbito de la optimización combinatoria. Además, se detallará la estructura del trabajo, incluyendo una descripción de los capítulos que lo componen y su contenido principal.
2. Capítulo 2: Estado del Arte. Se profundiza en el problema de la diversidad, desde sus fundamentos en optimización combinatoria hasta los principales modelos y técnicas heurísticas existentes para abordarlo. Se presta especial atención al modelo seleccionado y a las técnicas empleadas en este estudio
3. Capítulo 3: Metodología y desarrollo. Se detalla la metodología empleada, incluyendo la descripción de los conjuntos de datos, los algoritmos y su implementación eficiente.
4. Capítulo 4: Experimentos y Resultados. Se presenta los resultados de los algoritmos, detallando la configuración experimental, mostrando resultados con gráficos y tablas, y realizando un análisis estadístico.
5. Capítulo 5: Conclusiones y Trabajo a futuro. Por último, se recogen todas las conclusiones obtenidas del trabajo. Se discuten las implicaciones y se proponen líneas futuras de investigación.

Capítulo 2

Estado del Arte

2.1 Introducción

El problema de la diversidad se sitúa dentro del ámbito de la optimización combinatoria, con aplicaciones en diversos campos como las redes sociales, el análisis de datos, la optimización de la producción, la logística y la distribución, entre otros.

Este capítulo tiene como objetivo presentar el problema de la diversidad. Para ello, se exponen los fundamentos teóricos del problema, que incluyen el concepto de optimización y la complejidad computacional. Estos conceptos son esenciales para entender la naturaleza del problema y las estrategias que se pueden emplear para resolverlo de manera efectiva.

A continuación, se profundiza en las diferentes metodologías que se pueden utilizar para abordar el problema de la diversidad. Estas metodologías van desde métodos exactos, que garantizan encontrar la solución óptima, hasta heurísticas y metaheurísticas, que buscan soluciones de buena calidad en un tiempo razonable.

Finalmente, se enfoca en el propio problema de diversidad, un área ampliamente estudiada que cuenta con diversos modelos matemáticos para su estudio. Se describe la variante elegida para este trabajo, justificando su elección en base a las características del problema y los objetivos que persigue.

2.2 Introducción a la optimización

La optimización persigue encontrar la mejor solución posible (en adelante, solución óptima) a un problema dado dentro de un conjunto de soluciones factibles utilizando diferentes técnicas matemáticas.

En esencia, la resolución de este tipo de problemas consiste en maximizar o minimizar una función objetivo. Por lo tanto, el valor máximo o mínimo del conjunto de soluciones factibles será el óptimo buscado.

La optimización combinatoria busca un objeto óptimo en una colección finita de objetos. Típicamente, la colección tiene una representación concisa (como un grafo), mientras que el número de objetos es enorme (más precisamente, crece exponencialmente con el tamaño de la representación). Por lo tanto, no es una opción testear todos los objetos uno por uno y seleccionar el mejor. Se deben encontrar métodos más eficientes [9].

Además, el concepto de optimización combinatoria puede ser ambiguo, ya que algunos problemas buscan dispersión entre los puntos seleccionados, mientras que otros priorizan la representatividad, en la que los puntos seleccionados representan una clase del conjunto.

Dentro del contexto de la optimización combinatoria, algunos problemas tienen una complejidad computacional tan alta que obtener un resultado óptimo es impracticable en tiempos razonables. Algunos de los ejemplos más conocidos de la optimización combinatoria son: el problema del viajante, el problema de la mochila, el problema de coloración de grafos y el problema de la asignación de tareas.

Para resolver estos problemas se pueden usar diversas técnicas, como algoritmos de búsqueda exhaustiva (por ejemplo, la fuerza bruta), algoritmos heurísticos (por ejemplo, la búsqueda local) y algoritmos metaheurísticos (por ejemplo, los algoritmos genéticos).

2.3 Complejidad computacional

Antes de adentrarnos en el problema, es fundamental abordar el concepto de complejidad computacional: qué implica y qué factores determinan que algunos problemas sean más difíciles que otros en el ámbito computacional.

“La Teoría de la Complejidad Computacional es la parte de la teoría de la computación que estudia los recursos requeridos durante el cálculo para resolver un problema. Un

cálculo resulta complejo si es difícil de realizar. En este contexto podemos definir la complejidad de cálculo como la cantidad de recursos necesarios para efectuar un cálculo. Así, un cálculo difícil requerirá más recursos que uno de menor dificultad. Los recursos comúnmente estudiados son el tiempo (número de pasos de ejecución de un algoritmo para resolver un problema) y el espacio (cantidad de memoria utilizada para resolver un problema)" [25].

Puesto que la computación está en continuo cambio debido a las innovaciones tecnológicas, el tiempo y el espacio se ven reducidos y aumentados respectivamente, disponiendo así de más recursos. Por ende, es necesario representar la complejidad con otro tipo de notación basada en el aumento de operaciones requeridas en función de la entrada. Esto facilita la comparación de algoritmos de manera más sencilla puesto que el tiempo de ejecución varía de unas máquinas a otras siendo dependiente de la arquitectura de estas.

"Los tiempos que se consideran, "razonables" son los que están limitados por un polinomio (p), utilizándose en la mayoría de los casos la notación de E. Landau, "*O*", para medirlos, su forma es $O(p)$.

Si un algoritmo tiene complejidad $O(p(n))$ significa que, al ejecutarlo en una computadora con los mismos datos, pero con valores incrementales de n, los tiempos resultantes de ejecución serán siempre menores que $|p(n)|$" [28].

Lo ideal es que el tiempo de resolución del problema esté delimitado por un polinomio, ya que pueden resolverse en tiempos razonables.

Los principales órdenes de complejidad computacional representados con la anotación asintótica aparecen en la Tabla 1 [28].

Orden de complejidad	Descripción
Constante	Operación simple como sería asignar un valor a una variable. Se representa como $O(1)$.
Logarítmica	Se representa como $O(Log\ n)$. Aparece en operaciones donde la complejidad no crece al mismo nivel que n como puede ser una búsqueda binaria.
Lineal	Representado como $O(n)$. Se da principalmente en bucles que contienen operaciones simples del tipo constante $(O(1))$. Teniendo en cuenta esto la complejidad sería a $n * O(1)$ dando como resultado $O(n)$.
Casi lineal	Combina un bucle externo y otro interno con una complejidad $O(Log\ n)$. Dentro de esta complejidad se encuentra el algoritmo de ordenación *Quick Sort* y su complejidad se puede representar como $O(n\ *\ Log\ n)$.
Cuadrática	Esta complejidad se representa como $O(n^2)$ y se encuentra en bucles anidados donde se itera n veces en el bucle externo y n veces en el bucle interno.
Exponencial	Está representada como $O(2^n)$. Conforme aumenta el valor de n el tiempo de ejecución del algoritmo se duplica. Suelen aparecer en problemas relacionados con la búsqueda exhaustiva como la fuerza bruta para descifrar contraseñas.
Factorial	Se representa como $O(n!)$ e itera sobre todas las permutaciones de elementos de entrada.

Tabla 1. Órdenes de complejidad

Además, en función de su complejidad los problemas pueden clasificarse dentro de clases [29]:

- Clase P: Clase que suele estar asociada a los problemas de decisión cuyos tiempos de resolución son polinomiales por una máquina de Turing determinista, es decir, pueden resolverse en un tiempo n^k, siendo k una constante y n el

tamaño de la entrada. Como ejemplos de este tipo de problemas encontramos el ordenamiento de elementos de una lista o búsqueda de un elemento en una lista ordenada.

- Clase NP: Al igual que la clase P, se suelen asociar a problemas de decisión, incluyendo también aquellos resolubles por una máquina de *Turing* en tiempo polinomial no determinista, pudiendo verificar desde problemas solubles en tiempo polinomial a exponencial. Ejemplos de esta clase pueden ser el problema del viajante (*Traveling Salesman Problem, TSP*) o el problema de la mochila *(Knapsack Problem*)
- Clase *NP-Hard*, problemas que son al menos tan difíciles como los más complejos en la clase NP, pero sin estar necesariamente contenidos en ella. Estos problemas no siempre tienen soluciones que puedan ser verificadas en tiempo polinomial. Ejemplos de esta clase incluyen los problemas de satisfacción booleana *(Boolean Satisfiability Problem, BSP*) y el problema del viajante (*Travelling Salesman Problem, TSP*). El problema abordado en este trabajo, el problema de la máxima diversidad pertenece a esta última clase de complejidad.

2.4 Metodologías

Para resolver problemas de optimización se pueden utilizar dos metodologías: los métodos exactos y los métodos de resolución aproximada.

En este punto exploraremos estas metodologías, sus características y algunos de los algoritmos más conocidos.

2.4.1 Métodos exactos

En la categoría de métodos exactos se incluyen algoritmos que emplean técnicas analíticas o matemáticas para garantizar la convergencia hacia una solución óptima, en el caso de que exista. Estos métodos están diseñados teniendo en cuenta supuestos y características particulares, como continuidad, el tamaño limitado del espacio de búsqueda o la linealidad, entre otros. Se basan en teoremas matemáticos para desarrollar procedimientos que aseguran una solución óptima [27].

Uno de los principales problemas es que cuando tenemos un gran número de soluciones que explorar, se necesita un tiempo de computación muy elevado.

Algunos de los algoritmos más estudiados son:

- *Simplex*: Se utiliza para resolver problemas de programación lineal, donde se busca maximizar o minimizar una función lineal sujeta a un conjunto de restricciones lineales. A través de iteraciones, el algoritmo se mueve de una solución a otra adyacente que mejora el valor objetivo hasta que se alcanza la solución óptima.
- *Branch & Bound* (ramificación y acotación) [3]: Este algoritmo divide iterativamente el conjunto de soluciones. Explora en el subconjunto si hay posibilidades de encontrar una mejor solución que la obtenida hasta el momento. Como variaciones a este algoritmo tenemos los planos de corte y el *Branch & Cut*.
- Métodos de programación entera: Se utilizan para resolver problemas de optimización donde las variables de decisión deben tomar valores enteros. Son especialmente útiles cuando el problema puede ser formulado como un modelo matemático con variables enteras y restricciones lineales o no lineales. Existen dos tipos principales de programación entera: la programación entera binaria y la programación entera mixta. La diferencia fundamental entre ambas radica en el tipo de valores que pueden tomar cada variable. En la programación entera binaria, las variables solo pueden tomar valores binarios (0 o 1) mientras que, en la programación entera mixta, las variables pueden ser enteras y/o continuas.

2.4.2 Métodos de resolución aproximada

Estos métodos permiten obtener soluciones de alta calidad en un tiempo menor que el empleado en los métodos de resolución exacta. Este trabajo estará enfocado en la resolución del problema mediante estos métodos.

2.4.2.1 Métodos heurísticos

Podemos definir lo que es un método heurístico como [12]:

"Un método heurístico es un procedimiento para resolver un problema de optimización bien definido mediante una aproximación intuitiva, en la que la estructura del problema se utiliza de forma inteligente para obtener una buena solución."

En comparación con los métodos exactos, los métodos de resolución aproximada no buscan necesariamente una solución óptima, sino que se centran en encontrar una buena solución una buena solución en un tiempo aceptable.

Entre ellos, podemos destacar:

- Algoritmo *Greedy*: Es un método heurístico que toma decisiones locales óptimas en cada paso con la esperanza de encontrar una solución global óptima. En cada paso, selecciona la opción más prometedora sin considerar el impacto a largo plazo.
- Búsqueda aleatoria: Método heurístico simple que genera soluciones aleatorias y evalúa su calidad. Puede ser útil como punto de partida para algoritmos más sofisticados.
- Métodos constructivos: Construyen paso a paso una solución del problema mediante iteraciones. En cada paso se elige la opción que parece más prometedora según unos criterios establecidos.
- Métodos de búsqueda local: Parten de una solución inicial y la mejorar progresivamente.

2.4.2.2 Métodos metaheurísticos

Podemos definir los métodos metaheurísticos como [12]:

"Los procedimientos Metaheurísticos son una clase de métodos aproximados que están diseñados para resolver problemas difíciles de optimización combinatoria, en los que los heurísticos clásicos no son efectivos. Los Metaheurísticos proporcionan un marco general para crear nuevos algoritmos híbridos combinando diferentes conceptos derivados de la inteligencia artificial, la evolución biológica y los mecanismos estadísticos."

Estos métodos surgieron con el objetivo de mejorar los resultados obtenidos por los heurísticos tradicionales. El término fue introducido por Fred Glover en 1986.

Algunos ejemplos de este tipo de algoritmos son:

- *Tabu Search*: es una técnica de optimización que se basa principalmente en utilizar memoria a corto plazo para guiar la exploración del espacio de búsqueda, evitando caer en óptimos locales. En la memoria se almacena una lista de tabú es que contiene las soluciones visitadas para evitar explorar de nuevo esas áreas. Además, *Tabu Search* tiene dos componentes principales: la intensificación y la diversificación. La intensificación consiste en la modificación de las reglas de selección para dar lugar a combinaciones de movimientos y características de solución que históricamente han dado buenos resultados. Por otra parte, la diversificación fomenta que el proceso de búsqueda explore regiones no visitadas previamente, generando soluciones completamente distintas [17].
- GRASP (*Greedy Randomized Adaptive Search Procedure*): es una metahurística utilizada en problemas cuyas iteraciones consisten en 2 fases principales: construcción y búsqueda local. La fase de construcción crea una solución factible a partir de los elementos encontrados en un vecindario seleccionando aquellos elementos que mejoren la solución. Esto se hace a través de una función *greedy* o voraz que recorre la lista de elementos eligiendo aquellos que vayan mejorando la solución. Por otro lado, la fase de búsqueda local se encarga de mejorar la solución obtenida en la fase de construcción reemplazando de manera iterativa la solución actual por una solución mejor dentro del vecindario de la solución actual. Esta fase termina cuando no se encuentra una solución mejor en dicho vecindario [30].

2.5 El problema de la diversidad

2.5.1 Introducción

Podemos definir la diversidad como la presencia de diferencias o variedades entre elementos, por lo que el problema reside en identificar elementos que sean lo más diferentes entre sí. En términos prácticos, se emplea una distancia como medida de la diversidad entre los distintos elementos.

Maximizar la diversidad consiste en seleccionar un subconjunto de elementos de entre todo el conjunto con el objetivo de alcanzar el máximo grado de variabilidad.

Existen numerosas variantes propuestas para abordar esta problemática dentro del ámbito de la optimización combinatoria. Por ello, es importante tener en cuenta que términos como diversidad, dispersión y equidad son utilizados de manera intercambiable en muchos contextos de optimización, teniendo significados similares.

El problema de Máxima Diversidad (*Maximum Diversity Problem* o MDP), también conocido como *Max-Sum Diversity Problem*, consiste en seleccionar un subconjunto de elementos de máxima diversidad de un conjunto dado que maximice la diversidad entre ellos. Otra variante ampliamente estudiada es el *Max-Min Diversity Problem* (*MMDP*), en el cual se busca maximizar la mínima distancia entre los elementos seleccionados del subconjunto.

2.5.2 Variantes de diversidad

El problema de maximizar la diversidad implica la selección de un subconjunto de elementos *M* de un conjunto *N*, con el objetivo de maximizar la diversidad entre los elementos del subconjunto [10].

Las dos variantes más conocidas de diversidad consisten en maximizar la suma de la diversidad (*MaxSum*) y la maximización de la diversidad mínima (*MaxMin*). La investigación sobre estas dos variantes incluye numerosos estudios [18] [19] [21], métodos exactos [18] [20] [22] [23] y heurísticas [20] [23].

La variante *MaxSum* busca maximizar la suma total de una medida de diversidad entre los elementos seleccionados de un conjunto. Una interesante adaptación de esta variante es la *MaxMin*. Esta variante consiste en la maximización de la dispersión media [11].

Posteriormente, se introdujeron dos variantes en el ámbito de la diversidad llamados modelos de equidad (*MaxMinSum* y el *MinDiff*), ya que incorporan el concepto de equidad entre los candidatos [1].

2.5.2.1 Variante *MaxSum*

Como se ha mencionado anteriormente, esta variante tiene como objetivo maximizar la suma total de una medida de diversidad entre los elementos seleccionados de un conjunto. En otras palabras, el objetivo es seleccionar un subconjunto de elementos de manera que la suma de las diversidades entre ellos sea máxima.

Este problema puede formularse de la siguiente forma:

$$Maximizar \sum_{i<j} d_{ij} x_i x_j$$

$$\sum_{i=1}^{n} x_i = m$$

$$x_i \in \{0,1\}, i = 1, \dots, n$$

Donde se emplea d_{ij} para representar la distancia entre los puntos i y j, n es el número total de elementos del problema y m el número de elementos seleccionados. Además, la variable x_i puede tomar los valores 0 y 1, indicando si ese punto i es elegido o no [11]. Este trabajo se centra en la variante *MaxMin*, que será visto en detalle a continuación.

2.5.2.2 Variante *MaxMin*

La variante *MaxMin* busca maximizar la distancia mínima de los elementos seleccionados de un conjunto.

Podemos formular este problema de la siguiente forma:

$$Maximizar\ Min_{i<j}\ d_{ij} x_i x_j$$

$$\sum_{i=1}^{n} x_i = m$$

$$x_i \in \{0,1\}, i = 1, \dots, n$$

Aquí d_{ij} es la distancia entre los puntos i y j, n es el número total de elementos del problema y m el número de elementos seleccionados. La principal diferencia con

respecto al *MDP* (*Maximun Diversity Problem*), radica en que mide la diversidad como la mínima de todas las distancias entre los nodos seleccionados.

Este problema también se conoce como problema de dispersión [11].

Capítulo 3

Metodología y Desarrollo

3.1 Introducción

En este capítulo se expone la metodología seguida en este trabajo, que incluye desde la adquisición y preparación de los datos hasta la implementación y evaluación de las soluciones. Se detallarán los elementos clave y las herramientas empleadas en el proceso, junto con el contexto teórico necesario para comprender las técnicas y algoritmos utilizados.

El desarrollo de este trabajo conlleva la aplicación de diversos algoritmos heurísticos, los cuales serán descritos en detalle.

Este capítulo también proporciona una visión global del entorno y las herramientas utilizadas, además de los conjuntos de datos que se han utilizado para la experimentación.

3.2 Metodología

Para el desarrollo de los algoritmos, se ha utilizado la metodología Scrum, un marco de trabajo ágil que se basa en un sistema de mejora continua y de manera iterativa permitiendo tratar con problemas complejos de manera eficiente a través unos periodos de tiempo (periodos de una semana en el caso de este desarrollo) denominados *sprints*.

El primer sprint conllevó el planteamiento de los algoritmos, junto con las funciones de lectura de ficheros. El segundo se centró en la generación del gráfico junto con el sistema de almacenamiento de los resultados en una hoja de cálculo, que permitía analizar los resultados de los algoritmos.

Cada *sprint* finalizaba con una serie de pruebas para validar que los algoritmos funcionaran correctamente, revisando y analizando los resultados obtenidos tras cada una de las pruebas. Tras analizar los resultados, se buscaban posibles incongruencias, fallos o *bugs* en los algoritmos. Entonces se procedía a comenzar el siguiente sprint, con el que se busca realizar mejoras sobre lo ya desarrollado y/o solucionar errores de programación.

3.3 Selección y preparación del Conjunto de datos

Para llevar a cabo este trabajo, es fundamental seleccionar un conjunto de datos que se ajuste a la representación del problema de máxima diversidad y que esté disponible para realizar pruebas y evaluar las soluciones propuestas.

Se ha optado por utilizar el conjunto de datos de la librería *MDPLIB* [13] que contiene 770 instancias clasificadas en diferentes subconjuntos según su origen. En particular, existen tres conjuntos de instancias según el tipo de valores en sus matrices de distancia: euclidiana, real y entera.

De entre todos los conjuntos disponibles, hemos seleccionado el de datos *GKD-d*, ya que tiene un mayor número de instancias. Dicho conjunto, consta de 70 matrices que representan las distancias euclidianas entre puntos generados aleatoriamente. Cada punto está representado por dos coordenadas, generadas aleatoriamente dentro de un rango de 0 a 100.

En la Tabla 2 se muestra un resumen de las instancias del conjunto de datos, donde n es el tamaño del problema y m el número de elementos seleccionados.

Instancia	Tipo	*n*	*m*
GKD_d_[1-10]_n25_coor.txt	GKD-d	25	3
GKD_d_[1-10]_n50_coor.txt	GKD-d	50	5
GKD_d_[1-10]_n100_coor.txt	GKD-d	100	10
GKD_d_[1-10]_n200_coor.txt	GKD-d	200	20
GKD_d_[1-10]_n500_coor.txt	GKD-d	500	50
GKD_d_[1-10]_n1000_coor.txt	GKD-d	1000	100

Tabla 2. Resumen instancias del conjunto de datos.

El formato para cada uno de los ficheros es el siguiente:

- En la primera línea del fichero se encuentra un entero n que representa el número total de elementos.
- Las siguientes $n * (n-1)/2$ líneas contienen la distancia entre los elementos en el formato: elemento_A elemento_B distancia.
- El resto de líneas del fichero contienen las coordenadas *(x,y)* de cada elemento por orden ascendente.

Es importante destacar que el conjunto de datos utilizado no necesitó de una preparación adicional, es decir, no hubo que hacer limpieza de datos, normalización o una transformación de los datos puesto que la información ya estaba disponible para su uso, simplificando así el desarrollo de los algoritmos.

3.4 Tecnologías del desarrollo

Para el desarrollo e implementación de los algoritmos se ha hecho uso de las siguientes tecnologías:

- *Visual Studio Code*: entorno de desarrollo integrado (*IDE*) utilizado para codificar los algoritmos en el lenguaje de programación Python. Este entorno proporciona las herramientas necesarias para la edición, depuración y gestión del proyecto, lo que facilita el desarrollo y mantenimiento de código.

- *Python*: lenguaje de programación interpretado utilizado para el desarrollo de los algoritmos para el problema de máxima diversidad. Fue elegido debido a su facilidad de uso, versatilidad y por el número de librerías disponibles para la optimización y ciencia de datos.

La combinación de ambos ha facilitado el proceso de diseño, implementación y desarrollo, obteniendo una puesta en marcha eficiente y mantenible.

3.5 Implementación de Algoritmos

En este apartado, se detalla la implementación de los algoritmos seleccionados, un aspecto crucial para abordar de manera efectiva el problema de máxima diversidad.
Cada uno de estos algoritmos ofrece un enfoque distinto para obtener conjuntos de elementos que maximicen la diversidad.
El objetivo es proporcionar una exposición detallada de la implementación, diseño y funcionamiento de cada uno, brindando de esta manera una visión clara de su aplicación en el problema a abordar. En todos ellos se busca maximizar la distancia mínima desde diferentes enfoques aplicando distintas técnicas de optimización para obtener en cada caso buenas soluciones que se acerquen lo máximo posible a la solución óptima en un tiempo limitado.

3.5.1 Algoritmo *Greedy*

En general, un algoritmo voraz se describe como aquel que toma una secuencia de decisiones, eligiendo en cada paso la mejor opción disponible según una función objetivo o algún criterio definido. Este tipo de algoritmo avanza siempre tomando la decisión óptima en cada instante, sin retroceder a elecciones previas [14].
Es un algoritmo bastante simple, directo y eficiente, resultando útil para abordar diversos problemas de distintas áreas de la combinatoria

A continuación, en la Figura 1 se muestra el pseudocódigo del algoritmo *Greedy*.

```
Algoritmo Greedy:
      n = tamaño de la solución
      i=interacción actual
      Añadir nodo aleatorio a la solución
      mientras i < n
            Recorrer los nodos candidatos
                  (*) Añadir a solución el nodo que maximice la distancia mínima
                  i = i+1
                  (**) Quitar nodo seleccionado de los candidatos
      Devolver solución
```

Figura 1. Pseudocódigo del algoritmo *Greedy*.

(*) Se recorren todos los nodos candidatos y escogemos el que maximice la distancia mínima a los nodos solución.

(**) El nodo elegido que se ha añadido a la solución es eliminado de la lista de candidatos.

3.5.2 Algoritmo *Tabu Search*

Como se mencionó en el Capítulo 2, el algoritmo *Tabu Search* es una metaheurística que guía un procedimiento de búsqueda local para explorar el espacio de soluciones más allá de la optimalidad local.

En la Figura 2 se muestra el pseudocódigo para este algoritmo.

```
Algoritmo Tabu Search:
      m=tiempo máximo
      t = tiempo actual
      lista tabu = vacía
      i = iteraciones sin mejora
      max_i= máximo de iteraciones sin mejora
      (*) Solución actual = construir solución inicial
      Mientras t < m
            Si i < max_i entonces:
                  (**) Búsqueda local (eliminar/añadir)
                  Si nueva solución es mejor que solución actual:
                        Solución actual = nueva solución
                        i = 0
                  Si no:
                        i = i+1
                  (***) Actualizar lista tabu
            Si no:
                  (****) Diversificar
                  i = 0

      Devolver solución actual
```

Figura 2. Pseudocódigo del algoritmo *Tabu Search*.

El enfoque implementado se caracteriza por su eficiencia y capacidad para explorar el espacio de soluciones de manera efectiva [34].

El componente principal es un algoritmo de búsqueda local simple y altamente eficiente, diseñado para minimizar la complejidad de las iteraciones mientras maximiza la diversidad de las soluciones. A diferencia de otros métodos de búsqueda que utilizan una vecindad de coste cuadrático basada en movimientos de intercambio, este enfoque innova al realizar estos movimientos en dos pasos separados: eliminar y añadir. Esta estrategia tiene una complejidad lineal, lo que resulta en un coste computacional considerablemente menor en comparación con otros enfoques.

En la fase inicial del algoritmo, se establece un tiempo de ejecución y se inicializa la lista tabú, que se utiliza para evitar movimientos repetidos que podrían llevar a repetir soluciones.

(*) El proceso de construcción de la solución inicial implica la selección aleatoria de un punto inicial y la adición progresiva de puntos al conjunto de la solución. Durante este proceso, se busca maximizar la distancia mínima entre los puntos para garantizar una solución inicial de calidad.

(**) Después de construir la solución inicial, pasamos a la fase de búsqueda local. Esta fase consta de dos pasos principales: eliminar y añadir. En el primer paso, se elimina un nodo aleatorio de la solución actual mientras que, en el siguiente paso, se agrega un nuevo nodo que maximice la distancia mínima con respecto al resto de puntos de la solución. Para ambas acciones, se verifica que el nodo a eliminar o a añadir no figure en la lista tabú, evitando así la repetición de soluciones. Si se detecta un nodo en la lista, se selecciona otro que no esté presente en la misma.

Después de cada movimiento se actualiza la lista tabú (***), que contiene los *k* últimos nodos que han sido eliminados y los últimos *k* nodos añadidos a la solución. A continuación, se actualizan las distancias entre los puntos para reflejar los cambios en la solución y comprobar si mejora la solución actual. Este proceso se realiza siempre que no se haya superado el máximo número de iteraciones sin mejora.

Si no se producen mejoras en un número determinado de iteraciones, se introduce un mecanismo de diversificación (****) para explorar nuevas áreas del conjunto. Esta diversificación consiste en reiniciar la solución a través de la construcción de una nueva

solución partiendo de un punto inicial aleatorio, repitiendo la fase de búsqueda local. Finalmente, el algoritmo se detiene cuando alcanza el tiempo máximo especificado.

3.5.3 Algoritmo GRASP

Como fue mencionado en el Capítulo 2, este algoritmo consta de dos fases principales: construcción y búsqueda local. Cada iteración por tanto comienza con la construcción de la solución factible (*). Tras la fase de construcción, la solución generada se convierte en la solución actual. Comenzando de este modo la segunda fase: búsqueda local. En dicha fase se buscan mejores soluciones en el vecindario de la solución actual, comprobando en cada vecino si este mejora nuestra actual solución (**), en cuyo caso pasaría a ser la nueva solución (***).

En la Figura 3 encontramos el pseudocódigo correspondiente al algoritmo GRASP.

```
Algoritmo GRASP:
      m=tiempo máximo
      i=tiempo actual
      Mientras i < m
            (*) Construcción solución
            Mejoramos la solución
            (**) Si mejora la solución actual:
                  (***) Solución actual = nueva solución

      Devolver solución
```

Figura 3. Pseudocódigo del algoritmo GRASP.

3.5.4 Algoritmo *Hill Climbing*

El algoritmo *Hill Climbing* es una técnica heurística de búsqueda local utilizada en problemas de optimización. Su objetivo es mejorar de manera iterativa una solución explorando soluciones cercanas en el espacio de búsqueda comenzando por una solución aleatoria o arbitraria. Este proceso se repite hasta encontrar un óptimo local, una solución mejor que cualquiera de las vecinas. Este óptimo local puede o no ser la solución óptima global, es decir, la solución óptima al problema. El principal problema de este algoritmo es que puede quedar atrapado en estos óptimos locales siendo incapaz de encontrar mejores soluciones. Pese a ello, este algoritmo permite encontrar

buenas soluciones en un intervalo de tiempo corto y, con ciertas variaciones del algoritmo (como *Simulated Annealing*), es posible solventar este problema [16] [31].

En la Figura 4 aparece el pseudocódigo de un algoritmo *Hill Climbing*.

```
Algoritmo Hill_Climbing:
       m=máximo de iteraciones
       c=número de iteraciones sin mejorar solución
       i=interacción actual
       Parte de una solución inicial
       Mientras i < m
              (*) Buscar peor nodo solución
              (**) Buscar entre los k vecinos del peor nodo
              Si la nueva solución mejora la anterior entonces
                     solución = nueva solución
                     c=número de iteraciones sin mejorar solución
              Si no
                     Buscar siguiente peor nodo de la solución
                     Restar 1 a c
              Si c=0
                     (***) Diversificar solución

       Devolver solución
```

Figura 4. Pseudocódigo algoritmo *Hill Climbing*.

En este algoritmo partimos de una solución inicial generada por el algoritmo *Greedy* (descrito anteriormente).

Establecemos un criterio de parada, como por ejemplo un número máximo de iteraciones o que el algoritmo termine si la solución no mejora en *x* iteraciones.

En cada iteración, buscamos el peor nodo de la solución actual, exploramos los *k* vecinos del peor nodo y escogemos la mejor posible solución. Luego, mediante la función objetivo comprobamos si la posible solución obtenida mejora, y en caso de que sí, actualizamos la solución actual.

Si la solución no mejora al reemplazar el peor nodo, pasamos a comprobar el siguiente peor nodo de la solución.

(*) Se itera sobre todos los nodos de la solución y calculamos la distancia mínima entre cada nodo y los demás. Luego, devolvemos como peor nodo el que tiene la menor distancia mínima.

(**) Se itera entre todos los *k* nodos más cercanos al peor nodo que no se encuentran en la solución actual, y calculamos, mediante la función objetivo, cuál es el mejor nodo. En nuestro caso, será aquel que maximice la distancia mínima.

(***) Si tras n iteraciones la solución no mejora, implementamos una estrategia de escape para evitar quedar atrapados en óptimos locales. En este caso, intercambiamos el nodo peor nodo por un nodo aleatorio que no forme parte de la solución actual.

Capítulo 4

Experimentos y Resultados

4.1 Introducción

En este capítulo, se presentan los experimentos realizados para evaluar el rendimiento de los algoritmos diseñados en el capítulo anterior, enfocados en la variante *MaxMin* del problema de diversidad. El objetivo de estos experimentos es analizar la efectividad de los algoritmos propuestos en términos de calidad de las soluciones encontradas y tiempo de ejecución.

En primer lugar, se presenta la metodología experimental utilizada, que incluye la descripción del entorno de pruebas, la configuración de los parámetros de los algoritmos, los métodos de evaluación aplicados y los escenarios de prueba seleccionados. Posteriormente, se detallan los resultados obtenidos en cada experimento, acompañados de un análisis exhaustivo.

4.2 Metodología experimental

La metodología experimental se centra en asegurar la validez y replicabilidad de los resultados obtenidos, proporcionando una descripción exhaustiva del entorno de pruebas, la configuración de los parámetros de los algoritmos propuestos, los métodos de evaluación utilizados y los escenarios de prueba considerados.

La metodología establecida en este apartado es fundamental para garantizar que los resultados presentados en los apartados siguientes sean robustos y fiables, permitiendo una interpretación precisa y una comparación objetiva de las heurísticas evaluados.

4.2.1 Descripción del Entorno de Pruebas

Debido a que el tiempo de ejecución de los algoritmos depende en gran parte de las características técnicas del equipo en el que se ejecutan, conviene tener en cuenta las especificaciones del equipo utilizado para las pruebas. Dichas especificaciones aparecen detalladas en la Tabla 3.

Característica	Detalles
Procesador AMD Ryzen 7 5800X	Procesador con 16 núcleos a 3.8Ghz
Memoria RAM	Memoria principal de 32GB
Sistema operativo	Windows 11 Home 64 bits
Placa base	B550 AORUS ELITE V2
Almacenamiento	Samsung SSD 980 1TB

Tabla 3 . Especificaciones técnicas del equipo de pruebas.

4.2.2 Elementos de desarrollo

Como se especificó en el Capítulo 3, los algoritmos han sido desarrollados haciendo uso del lenguaje de programación Python. Se trata de un lenguaje de programación interpretado, es decir, no necesita ser compilado puesto que una pieza de software, el intérprete, se encarga de traducir las instrucciones en lenguaje Python para que las entienda la máquina donde se ejecuta dicho código.

La desventaja de hacer uso de un lenguaje interpretado es la perdida de rendimiento en comparación a otros lenguajes de programación compilados como puede ser C o C++, sin embargo, esto se ve fuertemente compensado por la gran cantidad de librerías, la facilidad, versatilidad y sencilla sintaxis que Python ofrece.

En la Tabla 4 se especifican las librerías utilizadas para el desarrollo de los algoritmos.

Librería	Descripción	Uso
NumPy	Librería que permite cálculos científicos en Python. Proporciona soporte para matrices y diferentes funciones matemáticas.	Manipulación de matrices de distancia.
matplotlib	Biblioteca de visualización de datos. Permite dibujar gráficos estáticos, animados e interactivos.	Visualización de datos a través de gráficos
math	Permite realizar operaciones matemáticas desde las más básicas hasta más complejas.	Redondeo de cálculos.
random	Librería que permite la generación de números aleatorios.	Selección de elementos de manera aleatoria y generación de semillas para reproducir mismos resultados en las ejecuciones de algoritmos.
pandas	Librería para manipulación y análisis de datos.	Generación de un fichero Excel con los datos obtenidos de las ejecuciones de los algoritmos.

Tabla 4. Librerías utilizadas para el desarrollo de los algoritmos.

4.2.3 Configuración de parámetros

Con el fin de tener un mayor control sobre la ejecución de los algoritmos, estos pueden ser configurados mediante los parámetros especificados en la Tabla 5.

Parámetro	**Descripción**	**Algoritmos**
m	Tamaño de la solución, establecido al 10% del tamaño del problema.	- *Greedy* - *Tabu Search* - *GRASP*
max_time	Máximo tiempo de ejecución del algoritmo. Este tiempo de ejecución coincide con el valor del tamaño del problema (n) cuando $n < 1000$. Para el caso de n=1000 este valor será de 500 segundos.	- *Tabu Search* - *GRASP*
tabu_size	Tamaño de la lista tabú, coincide con el valor de m.	- *Tabu Search*
num_neighbors	Número de vecinos a explorar, coincide con el valor de m.	- *GRASP* - *Hill Climbing*
initial_solution	Solución inicial.	- *Hill Climbing*

Tabla 5. Descripción de parámetros.

4.3 Resultados y análisis

4.3.1 Comparación de Resultados

Las Tablas 6, 7, 8, 9, 10 y 11 presentan los errores relativos para cada uno de los algoritmos evaluados. Estos errores se calcularon comparando los resultados obtenidos para cada algoritmo con la solución exacta. Cada tabla corresponde a diferentes tamaños de conjuntos de datos y escenarios específicos, proporcionando una visión más detallada que incluye la instancia, el valor exacto y el error relativo de cada algoritmo.

En la Tabla 6 se muestra la comparativa para *max_time*=n=25 y m=3.

Instancia	Exacto	*Greedy*	*Hill Climbing*	*Tabu Search*	*GRASP*
GKD_d_1_n25_coor.txt	80,2850	32,45%	4,45%	0,00%	0,00%
GKD_d_2_n25_coor.txt	78,4206	7,11%	26,04%	0,00%	0,00%
GKD_d_3_n25_coor.txt	81,2088	17,94%	2,22%	0,00%	0,00%
GKD_d_4_n25_coor.txt	81,5193	6,34%	0,81%	0,00%	0,00%
GKD_d_5_n25_coor.txt	77,9585	27,83%	0,00%	0,00%	0,00%
GKD_d_6_n25_coor.txt	74,0686	6,60%	4,19%	0,00%	0,00%
GKD_d_7_n25_coor.txt	77,0082	17,10%	0,00%	0,00%	0,00%
GKD_d_8_n25_coor.txt	91,6072	9,71%	0,00%	0,00%	0,00%
GKD_d_9_n25_coor.txt	85,5529	0,82%	3,83%	0,00%	0,00%
GKD_d_10_n25_coor.txt	88,9370	37,91%	0,00%	0,00%	0,00%

Tabla 6. Comparación con resultados exactos para n=25.

En la Tabla 7 se muestra la comparativa para *max_time*=n=50 y m=5.

Instancia	Exacto	*Greedy*	*Hill Climbing*	*Tabu Search*	*GRASP*
GKD_d_1_n50_coor.txt	54,2142	11,05%	4,43%	0,00%	0,00%
GKD_d_2_n50_coor.txt	54,2074	15,74%	2,49%	0,00%	0,00%
GKD_d_3_n50_coor.txt	55,7473	10,99%	18,54%	0,00%	0,00%
GKD_d_4_n50_coor.txt	53,8571	6,25%	0,00%	0,00%	0,00%
GKD_d_5_n50_coor.txt	56,5081	12,06%	0,00%	0,00%	0,00%
GKD_d_6_n50_coor.txt	50,8343	12,04%	7,53%	0,00%	3,69%
GKD_d_7_n50_coor.txt	57,8489	6,15%	17,61%	0,00%	0,00%
GKD_d_8_n50_coor.txt	50,6257	6,11%	6,11%	0,00%	0,00%
GKD_d_9_n50_coor.txt	52,2195	15,46%	0,00%	0,00%	0,00%
GKD_d_10_n50_coor.txt	57,4800	19,66%	0,00%	0,00%	0,00%

Tabla 7. Comparación con resultados exactos para n=50.

En la Tabla 8 se muestra la comparativa para *max_time*=n=100 y m=10.

Instancia	Exacto	*Greedy*	*Hill Climbing*	*Tabu Search*	*GRASP*
GKD_d_1_n100_coor.txt	34,1105	3,59%	3,61%	0,00%	0,00%
GKD_d_2_n100_coor.txt	34,8199	16,49%	15,51%	0,00%	0,00%
GKD_d_3_n100_coor.txt	36,0607	15,76%	4,69%	0,00%	0,00%
GKD_d_4_n100_coor.txt	34,5924	7,92%	4,84%	0,51%	0,51%
GKD_d_5_n100_coor.txt	35,6098	17,48%	6,09%	0,00%	0,00%
GKD_d_6_n100_coor.txt	33,6376	16,39%	7,74%	0,00%	0,00%
GKD_d_7_n100_coor.txt	35,0592	13,26%	14,01%	0,00%	0,00%
GKD_d_8_n100_coor.txt	33,7839	11,15%	11,85%	0,00%	0,00%
GKD_d_9_n100_coor.txt	35,3872	15,64%	18,40%	0,00%	2,11%
GKD_d_10_n100_coor.txt	34,9274	21,38%	5,51%	0,00%	3,51%

Tabla 8. Comparación con resultados exactos para n=100.

En la Tabla 9 se muestra la comparativa para *max_time*=n=250 y m=25.

Instancia	Exacto	*Greedy*	*Hill Climbing*	*Tabu Search*	*GRASP*
GKD_d_1_n250_coor.txt	20,3458	18,32%	6,55%	4,16%	3,96%
GKD_d_2_n250_coor.txt	20,5475	18,54%	11,76%	6,23%	5,49%
GKD_d_3_n250_coor.txt	20,3847	22,65%	19,69%	6,28%	5,56%
GKD_d_4_n250_coor.txt	19,8068	17,59%	14,19%	7,01%	6,75%
GKD_d_5_n250_coor.txt	20,9566	20,94%	12,31%	5,51%	4,96%
GKD_d_6_n250_coor.txt	20,0879	13,28%	16,36%	5,14%	4,67%
GKD_d_7_n250_coor.txt	20,5443	13,71%	14,32%	6,33%	4,31%
GKD_d_8_n250_coor.txt	20,3227	17,76%	14,65%	5,78%	3,54%
GKD_d_9_n250_coor.txt	20,1763	14,32%	15,45%	7,22%	6,49%
GKD_d_10_n250_coor.txt	19,8228	19,33%	10,20%	3,94%	4,91%

Tabla 9. Comparación con resultados exactos para n=250.

En la Tabla 10 se muestra la comparativa para *max_time*=*n*=500 y *m*=50.

Instancia	Exacto	*Greedy*	*Hill Climbing*	*Tabu Search*	*GRASP*
GKD_d_1_n500_coor.txt	13,7191	17,68%	12,31%	12,31%	6,75%
GKD_d_2_n500_coor.txt	13,9359	18,71%	14,62%	12,24%	9,16%
GKD_d_3_n500_coor.txt	13,7710	17,95%	15,23%	9,18%	7,40%
GKD_d_4_n500_coor.txt	13,9535	19,13%	12,60%	12,31%	8,53%
GKD_d_5_n500_coor.txt	13,7691	17,93%	17,66%	12,14%	9,45%
GKD_d_6_n500_coor.txt	13,8903	15,37%	13,87%	11,29%	9,30%
GKD_d_7_n500_coor.txt	13,7898	16,29%	17,60%	11,51%	8,83%
GKD_d_8_n500_coor.txt	14,1009	16,48%	13,93%	9,83%	8,11%
GKD_d_9_n500_coor.txt	13,7167	19,61%	11,70%	10,78%	6,03%
GKD_d_10_n500_coor.txt	13,9660	20,86%	18,29%	11,43%	7,49%

Tabla 10. Comparación con resultados exactos para n=500.

En la Tabla 11 se muestra la comparativa para *max_time*=n=1000 y m=100.

Instancia	Exacto	*Greedy*	*Hill Climbing*	*Tabu Search*	*GRASP*
GKD_d_1_n1000_coor.txt	9,58187	18,67%	15,56%	13,37%	10,81%
GKD_d_2_n1000_coor.txt	9,50384	20,74%	18,68%	14,53%	12,98%
GKD_d_3_n1000_coor.txt	9,41115	16,33%	15,49%	13,40%	10,83%
GKD_d_4_n1000_coor.txt	9,5069	19,43%	14,79%	13,51%	11,46%
GKD_d_5_n1000_coor.txt	9,60207	17,64%	17,11%	14,78%	11,81%
GKD_d_6_n1000_coor.txt	9,52157	15,51%	14,06%	12,99%	11,56%
GKD_d_7_n1000_coor.txt	9,55244	19,07%	14,35%	12,25%	11,66%
GKD_d_8_n1000_coor.txt	9,52252	17,45%	17,12%	14,47%	11,40%
GKD_d_9_n1000_coor.txt	9,53432	19,04%	15,49%	14,09%	11,80%
GKD_d_10_n1000_coor.txt	9,49714	23,42%	14,30%	14,49%	11,58%

Tabla 11. Comparación con resultados exactos para n=1000.

La siguiente gráfica (Figura 5) presenta la media de los errores relativos para cada conjunto de datos y cada algoritmo. Esta visualización proporciona una perspectiva general del desempeño de cada algoritmo en diferentes conjuntos de datos, destacando las tendencias en la precisión relativa de las soluciones obtenidas.

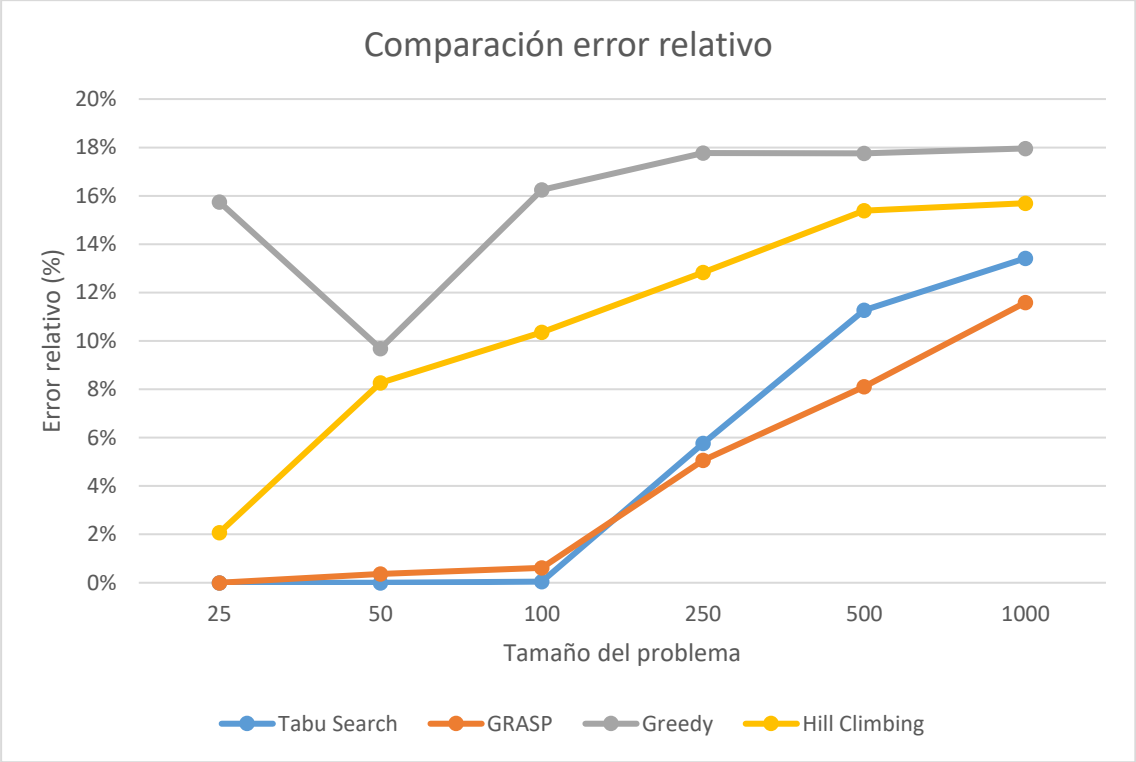

Figura 5. Gráfica comparativa de los errores relativos.

El análisis de los datos proporcionados revela una tendencia similar en cuanto al error relativo en relación con el tamaño del conjunto de datos. En conjunto de datos más pequeños, los algoritmos *Tabu Search* y *GRASP* muestran errores relativos casi despreciables, siendo 0 o aproximándose. En contraste, el algoritmo *Greedy* y *Hill Climbing* muestran errores relativamente más altos desde el inicio.

A medida que el tamaño del conjunto de datos aumenta, se observa una tendencia creciente en el error relativo para todos los algoritmos evaluados, aunque en distintos niveles. Por ejemplo, en problemas de tamaño 250, el algoritmo *Tabu Search* y el *GRASP* mantiene un error relativo bajo, alrededor del 5%, mientras que el algoritmo *Greedy* presenta un error significativamente mayor, alrededor del 18%.

Al considerar un conjunto de datos de tamaño 1000, se evidencia una diferencia significativa en los errores relativos entre algoritmos. Los algoritmos *Greedy* y *Hill Climbing* presentan errores relativos más altos, aproximadamente del 18% y 15% respectivamente. Por otro lado, aunque *Tabu Search* y *GRASP* comienzan con un error mínimo, ambos alcanzan un error relativo considerable del 13% aproximadamente.

Este análisis subraya la importancia de considerar el tamaño del conjunto de datos al evaluar el rendimiento de los algoritmos. Aunque todos los algoritmos experimentan un aumento en el error relativo a medida que los conjuntos de datos se vuelven más grandes, el algoritmo *GRASP* demuestra una capacidad superior para mantener el error relativo en niveles más bajos, lo que sugiere una mayor eficacia en la resolución de problemas de diversidad en conjuntos de datos más grandes.

4.3.2 Ejemplos gráficos de los resultados obtenidos

A continuación, se presentan ejemplos que ilustran los resultados obtenidos por los algoritmos *Tabu Search* y GRASP para conjuntos de datos de tamaño 50,100 y 250. Los gráficos muestran la disposición de los nodos en el espacio, donde los nodos están representados en color azul y los nodos seleccionados como solución están resaltados en rojo.

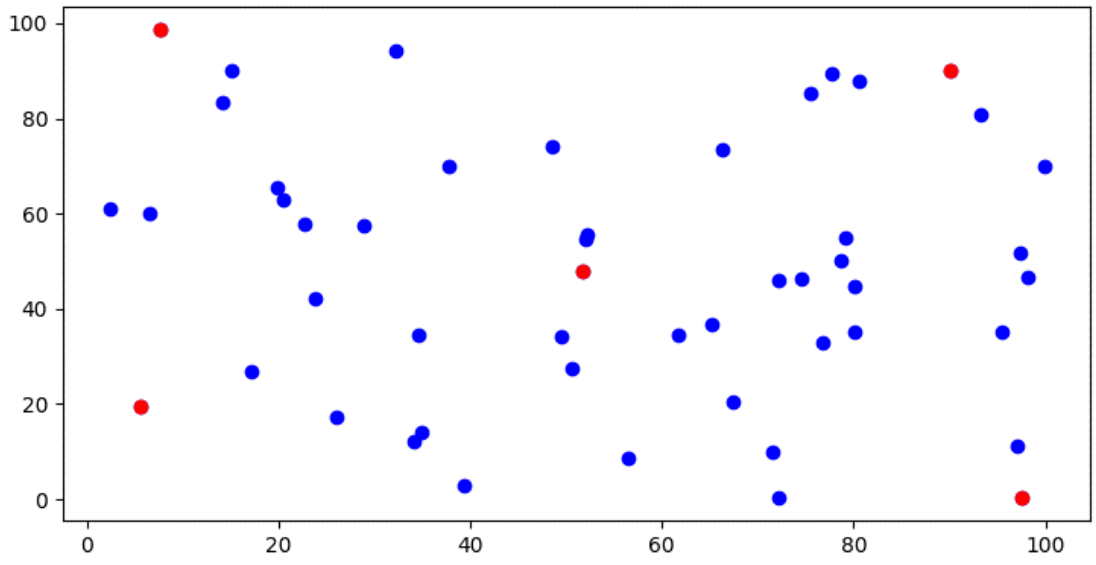

Figura 6. Resultados *Tabu Search* y *GRASP* para la instancia GKD_d_2_n50_coord.txt.

Como se muestra en la Figura 6 y podemos ver en la Tabla 19, tanto el algoritmo *Tabu Search* como el *GRASP* lograron obtener la solución exacta para la instancia GKD_d_2_50_coord.txt con una distancia mínima de 54,2074.

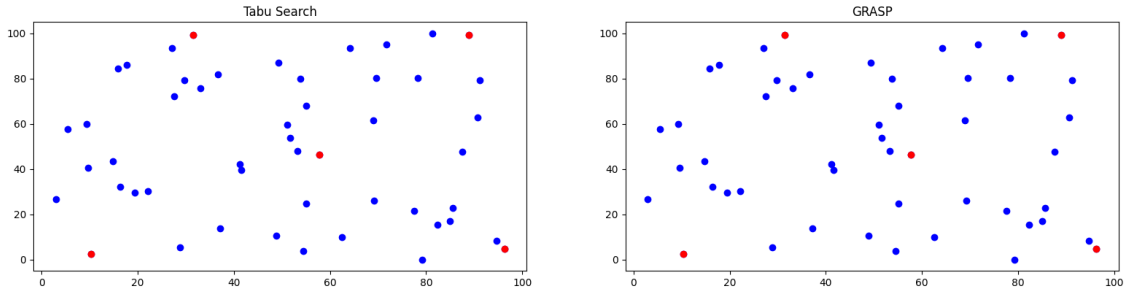

Figura 7. Resultados *Tabu Search* y *GRASP* para la instancia GKD_d_6_n50_coord.txt.

En este segundo ejemplo, ilustrado en la Figura 7, se observan diferencias en las soluciones de los dos algoritmos. El algoritmo *Tabu Search* obtiene la solución exacta con una distancia mínima de 50,8343, mientras que el algoritmo *GRASP* obtiene una distancia mínima de 48,96 con un error relativo del 3,69%.

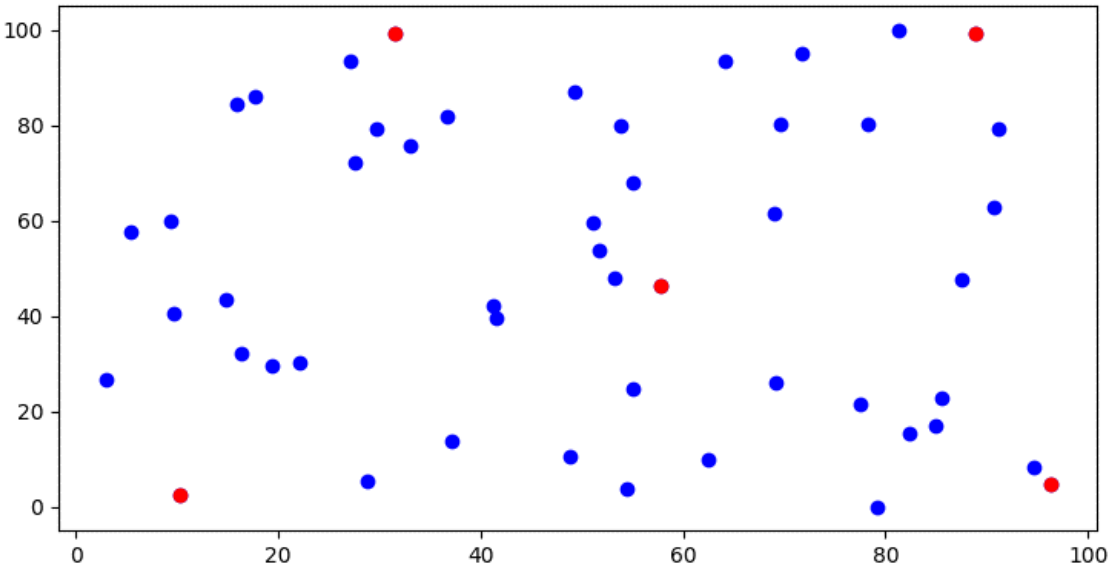

Figura 8. Resultados *Tabu Search* y *GRASP* para la instancia GKD_d_4_n100_coord.txt.

Como se muestra en la Figura 8, los resultados obtenidos para la instancia GKD_d_4_n100_coord.txt son idénticos para ambos algoritmos, con una distancia mínima de 34,4143 y un error relativo de 0,51%.

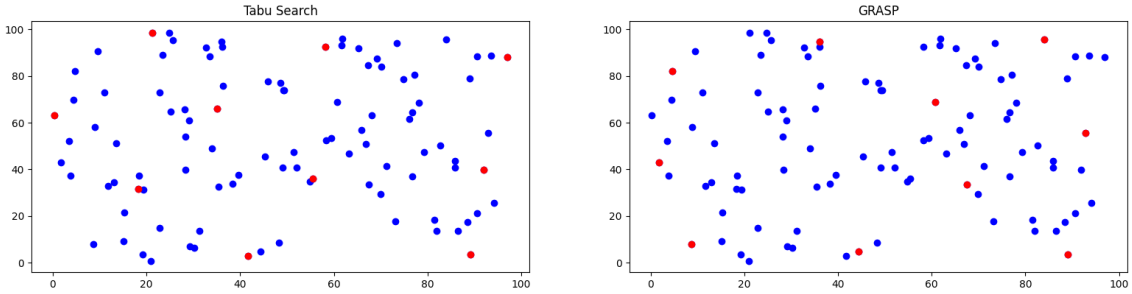

Figura 9. Resultados *Tabu Search* y *GRASP* para la instancia GKD_d_10_n100_coord.txt.

En la Figura 9, se presenta otro ejemplo para el tamaño 100. En este caso el algoritmo *Tabu Search* obtiene el resultado exacto, mientras que el algoritmo *GRASP* obtiene una distancia mínima de 33,7002 con un error relativo del 3,51%.

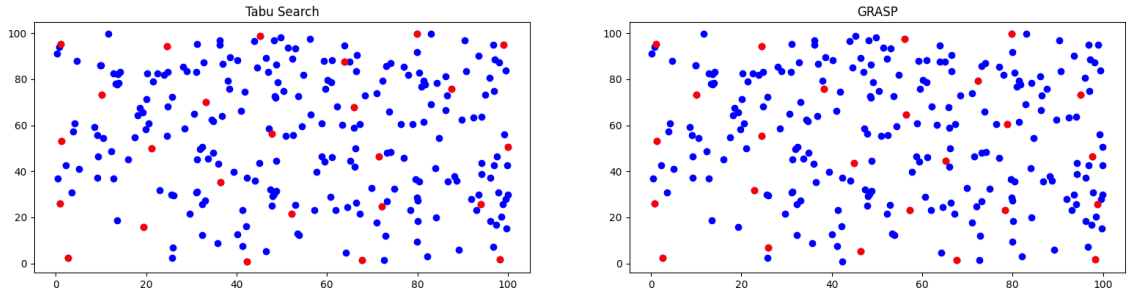

Figura 10. Resultados *Tabu Search* y *GRASP* para la instancia GKD_d_5_n250_coord.txt

Por último, en la Figura 10, podemos ver un ejemplo para un tamaño de 250, donde se destacan mejor las diferencias entre los algoritmos. En este caso, el algoritmo *Tabu Search* obtiene una distancia mínima de 19,8026 con un error relativo del 5,51%, mientras que el algoritmo GRASP logra una distancia mínima de 19,9164 con un error relativo del 4,96%.

Estos resultados indican que, si bien ambos algoritmos pueden ser efectivos en la resolución del problema de diversidad, pueden existir variaciones en su rendimiento dependiendo de las características específicas del conjunto de datos.

Capítulo 5

Conclusiones y Trabajo Futuro

5.1 Conclusiones

A lo largo de este trabajo, se ha abordado el tema del problema de la máxima diversidad, centrándose en la implementación de varios algoritmos para la búsqueda de soluciones en la variante *MaxMin* del problema. Desde el inicio del proyecto, se presentó el problema y se establecieron los objetivos que guiaron nuestra investigación.

El desarrollo del trabajo incluyó una revisión exhaustiva de los conceptos básicos sobre la complejidad computacional, heurísticas y metaheurísticas, así como las metodologías necesarias para comprender el problema que estamos analizando. Esta contextualización fue fundamental para comprender las dificultades actuales en el campo de la computación al enfrentar problemas de diversidad.

Posteriormente, se introdujeron los algoritmos y los conjuntos de datos utilizados, junto con las tecnologías y herramientas empleadas a lo largo del desarrollo. Se detalló el funcionamiento de cada uno de los algoritmos implementados en el trabajo, proporcionando una base sólida para la experimentación y evaluación posterior.

Finalmente, se ejecutaron una serie de casos de prueba para los algoritmos implementados, cuyos resultados fueron analizados para realizar comparaciones entre ellos y con los resultados obtenidos a partir de métodos exactos. Este análisis nos

permitió evaluar la eficacia de los algoritmos en diferentes escenarios y conjuntos de datos, proporcionando una visión completa de su rendimiento.

En última instancia, este estudio no solo ha proporcionado una visión detallada del problema de máxima diversidad y la eficacia de los algoritmos, sino que también ha avanzado en nuestro entendimiento de las complejidades de este problema y ha identificado áreas potenciales para mejoras y refinamientos en los algoritmos existentes.

En un mundo en constante evolución donde la optimización computacional desempeña un papel crucial en numerosos ámbitos, esperamos que este trabajo sirva como referencia útil para futuras exploraciones y contribuya al desarrollo de soluciones más efectivas y eficientes para problemas de diversidad.

5.2 Trabajo futuro

La realización de este trabajo me ha proporcionado una base sólida de conocimientos sobre optimización combinatoria y la optimización de la diversidad. Pese a que los algoritmos y metaheurísticas utilizados en el trabajo dan buenos resultados, es posible avanzar en ciertas direcciones que permitan ampliar y mejorar el presente trabajo:

- Considerar otras heurísticas como pueden ser los algoritmos genéticos, optimización por enjambre de partículas (*PSO*) o los algoritmos evolutivos multiobjetivo (*MOEA*) para tratar con la función objetivo *MaxMin* y *MaxSum* al mismo tiempo.
- Mejorar las Heurísticas existentes:
    - Algoritmo *Tabu Search*: puede mejorar su eficiencia mediante la implementación de estrategias avanzadas de memoria, como la memoria a largo plazo, que almacena buenas soluciones, que habían sido descartadas en un principio para no volver a visitarlas.
    - *GRASP*: puede integrar técnicas de búsqueda local más sofisticadas como puede ser la búsqueda en entornos variables (*VNS*), que consiste en cambiar de manera sistemática el entorno de búsqueda evitando quedar atrapado en óptimos locales.

- *Hill Climbing*: adición de técnicas como búsqueda con reinicios aleatorios o búsqueda de entornos múltiples permitiría al algoritmo salir de óptimos locales.

Finalmente, se podría hacer uso de la paralelización, donde el espacio de búsqueda se reparta entre diferentes hilos y se mejore considerablemente el tiempo de ejecución de los algoritmos.

Bibliografía

[1] Prokopyev, O.A., Kong, N. and Martinez-Torres, D.L. (2009) *'The equitable dispersion problem', European Journal of Operational Research*, 197(1), pp. 59–67. doi:10.1016/j.ejor.2008.06.005.

[2] Martí, R. et al. (2011) '*Heuristics and metaheuristics for the maximum diversity problem', Journal of Heuristic*s, 19(4), pp. 591–615. doi:10.1007/s10732-011-9172-4.

[3] Martí, R., Gallego, M. and Duarte, A. (2010) '*A branch and bound algorithm for the maximum diversity problem', European Journal of Operational Research*, 200(1), pp. 36–44. doi:10.1016/j.ejor.2008.12.023.

[4] Cutillas-Lozano, J.-M., Franco, M.-Á. and Giménez, D. (2015) *'Comparing variable width backtracking and metaheuristics, experiments with the maximum diversity problem*', *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*. doi:10.1145/2739482.2764883.

[5] Puchinger, J. and Raidl, G.R. (2005) '*Combining metaheuristics and exact algorithms in combinatorial optimization: A survey and classification*', *Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach*, pp. 41–53. doi:10.1007/11499305_5.

[6] Duarte, A. and Martí, R. (2007) '*Tabu search and GRASP for the maximum diversity problem', European Journal of Operational Research*, 178(1), pp. 71–84. doi:10.1016/j.ejor.2006.01.021.

[7] Martí, R. (2003) 'Multi-start methods', International Series in Operations Research & Management Science, pp. 355–368. doi:10.1007/0-306-48056-5_12.

[8] Aringhieri, R., Cordone, R. and Melzani, Y. (2007) '*Tabu search versus GRASP for the maximum diversity problem*', 4OR, 6(1), pp. 45–60. doi:10.1007/s10288-007-0033-9.

[9] Schrijver, A. (2003) Combinatorial optimization. Berlin: Springer.

[10] Glover, F., Ching-Chung, K. and Dhir, K.S. (1995) 'A discrete optimization model for preserving biological diversity', Applied Mathematical Modelling, 19(11), pp. 696–701. doi:10.1016/0307-904x(95)00083-v.

[11] Parreño, F., Álvarez-Valdés, R. and Martí, R. (2021) 'Measuring diversity. A review and an empirical analysis', European Journal of Operational Research, 289(2), pp. 515–532. doi:10.1016/j.ejor.2020.07.053.

[12] Martí, R. (2003). Procedimientos metaheurísticos en optimización combinatoria. Matemátiques, Universidad de Valencia, 1(1), 3-62.

[13] Martí, R., A. Duarte, A. Martínez-Gavara, and J. Sánchez-Oro (2021) The MDPLIB 2.0 Library of Benchmark Instances for Diversity Problems. https://www.uv.es/rmarti/paper/mdp.html

[14] Curtis, S.A. (2003) 'The classification of Greedy Algorithms', Science of Computer Programming, 49(1–3), pp. 125–157. doi:10.1016/j.scico.2003.09.001.

[15] Feo, T.A. and Resende, M.G. (1995) 'Greedy randomized adaptive search procedures', Journal of Global Optimization, 6(2), pp. 109–133. doi:10.1007/bf01096763.

[16] AI: Search algorithms: Hill Climbing (2024) Codecademy. Consultado el 20 de abril, en: https://www.codecademy.com/resources/docs/ai/search-algorithms/hill-climbing.

[17] Glover, F. and Laguna, M. (1997) 'Tabu search principles', Tabu Search, pp. 125–151. doi:10.1007/978-1-4615-6089-0_5.

[18] Ağca, S., B. Eksioglu, J. B. Ghosh; "Lagrangian solution of maximum dispersion problems". Naval Research Logistics, 47: 97–114, 2000.

[19] Erkut, E., S. Neuman; "Analytical models for locating undesirable facilities". European Journal of Operational Research, 40: 275–291, 1989.

[20] Ghosh, J. B; "Computational aspects of the maximum diversity problem". Operations Research Letters, 19: 175–181, 1996.

[21] Kuo, C. C., F. Glover, K. S. Dhir; "Analyzing and modeling the maximum diversity problem by zero-one programming". Decision Sciences, 24:1171–1185, 1993.

[22] Pisinger, D; "Upper bounds and exact algorithms for p-dispersion problems". Computers and Operations Research, 33: 1380–1398, 2006.

[23] Resende, M. G. C., Martí, M. Gallego, A. Duarte: "GRASP and path relinking for the max–min diversity problem". Computers and Operations Research, 37(3): 498-508, 2010.

[24] Ravi, S. S., D. J. Rosenkrantz, G.K. Tayi; "Heuristic and special case algorithms for dispersion problems". Operations Research, 42: 299-310, 1994.

[25] Cortéz, A. (2004). Teoría de la complejidad computacional y teoría de la computabilidad. RISI, 1(1), 102-105.

[27] Morillo, D., Moreno, L., & Díaz, J. (2014). Metodologías analíticas y heurísticas para la Solución del Problema de Programación de Tareas con Recursos Restringidos (RCPSP): una revisión. Parte 1. Ingeniería y Ciencia, 10(19), 247-271.

[28] Complejidad computacional (2024) Numerenturorg. Consultado el 12 de mayo, en: https://numerentur.org/complejidad-computacional/

[29] Arora, S. and Barak, B. (2016) Computational complexity a modern approach. New York: Cambridge University Press.

[30] Resende, M.G. and Ribeiro, C.C. (2003) 'Greedy randomized adaptive search procedures', International Series in Operations Research & Management Science, pp. 219–249. doi:10.1007/0-306-48056-5_8.

[31] Russell, S.J. et al. (2011) Inteligencia artificial: Un Enfoque Moderno. Madrid: Pearson Educación.

[32] Mercedes Guijarro Gil, "mdp", Github, https://github.com/merguijarrog/mdp

[33] Homer, S., Selman, A. L., & Homer, S. (2011). Computability and complexity theory (Vol. 194). New York: Springer.

[34] Porumbel, D. C., Hao, J.-K., & Glover, F. (2011). A simple and effective algorithm for the Maxmin Diversity Problem. Annals of Operations Research, 186(1), 275–293. https://doi.org/10.1007/s10479-011-0898-z

Anexo I. Estructura del proyecto

En este anexo se recoge la estructura del proyecto y cómo ejecutarlo. Todo el contenido se puede encontrar en el repositorio de *Github* [32].

I.1 Contenido

El proyecto consta de los siguientes directorios y archivos principales:

- *instances*/: directorio que contiene los conjuntos de datos utilizados para evaluar los algoritmos. Están en formato txt.
- *src*/: directorio que contiene todos los archivos fuente de los diferentes algoritmos, los archivos para procesar los ficheros txt y un fichero para poder mostrar gráficas para representar el conjunto de datos.
    - main.py: fichero principal que procesa cada uno de los ficheros.
    - save_info.py: fichero que guarda la solución que devuelve cada uno de los algoritmos.
    - greedy.py: implementación del algoritmo *Greedy*.
    - tabu_algorithm.py: implementación del algoritmo *Tabu Search*.
    - hill_climbing.py: implementación de la búsqueda local.
    - grasp_algorithm.py: implementación del algoritmo GRASP.
    - print_graphic.py: permite representar el conjunto de datos y la solución obtenida para un algoritmo.

  - resultados.xlsx: *excel* que contiene todos los resultados obtenidos para cada uno de los algoritmos.
  - maxmin_v2.xlsx: Excel que contiene los resultados obtenidos con métodos exactos.
  - Comparación_resultados.xlsx: comparación de resultados entre los algoritmos *Tabu* y *GRASP* con los métodos exactos junto con el error relativo.
  - TFGMercedesGuijarro.pdf: contiene la memoria del trabajo.
- README.MD: contiene la información necesaria para entender la estructura del proyecto, la instalación de las dependencias necesarias y la ejecución de los algoritmos.
- requirements.txt: contiene las librerías de Python necesarias.

En la Figura 10 podemos ver la estructura del proyecto.

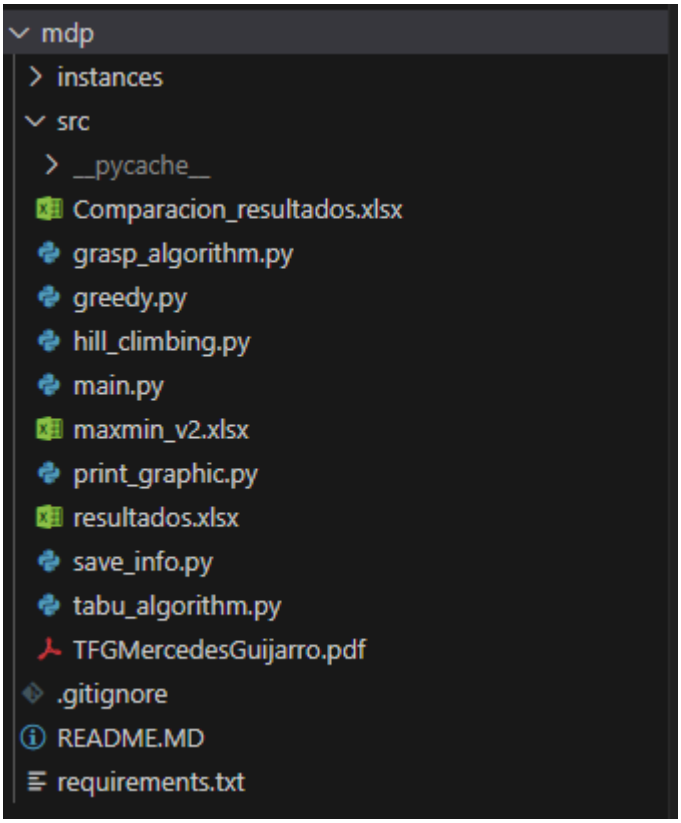

Figura 11. Estructura del proyecto.

Para ejecutar el proyecto basta con situarnos en el directorio *src* y utilizar el comando: "*python .\main.py*", con eso se ejecutarán cada uno de los algoritmos para las instancias

que tenemos en la carpeta *instances* y se almacenará toda la información en el fichero resultados.