

**Universidad  
Rey Juan Carlos**

Escuela Técnica Superior  
de Ingeniería Informática

**Grado en Ingeniería Informática**

**Curso 2025-2026**

**Trabajo Fin de Grado**

**TOMA DE DECISIONES INTELIGENTES EN LA  
RECOGIDA DE RESIDUOS URBANOS**

**Autor: Marcos Ferrer Zalve  
Tutor: Marin Lujak**



# Agradecimientos

A mi abuela, por entenderme y ponerme los pies en la tierra como sólo ella sabe. Gracias por estar siempre ahí, incondicionalmente. Gracias por enseñarme desde pequeño a ser mejor persona.

A mi abuelo, mi "socio". Gracias por preocuparte siempre por mí, por tu sabiduría, tus palabras, tus enseñanzas y sobre todo, por ser no sólo mi abuelo si no también mi compañero y mi apoyo.

A mi padre, por todos los días de camino a Alcázar, todo lo que he aprendido cuando le acompañaba día a día, por enseñarme lo que significa hacer las cosas bien y demostrarme que siempre puedo seguir.

A mi madre, por todas las charlas esas tardes muertas, por los cafés que nos servían para descansar del día a día, por aguantarme aún cuando era insoportable. En definitiva, por ser la mejor madre que nadie podría pedir.

Gracias a vosotros soy quien soy y no puedo más que dedicaros este proyecto a vosotros. Gracias por todo.



# Resumen

La recolección urbana de residuos representa un reto logístico de gran relevancia debido a su impacto económico, ambiental y operativo en las ciudades modernas. La eficiencia en la planificación de rutas de recogida es un factor clave para reducir costes, optimizar recursos y minimizar emisiones contaminantes asociadas al transporte de residuos urbanos. Este trabajo propone un sistema inteligente y escalable de planificación de rutas basado en técnicas algorítmicas aplicadas al problema de optimización de rutas, aproximando el problema como un Problema del Viajante Múltiple (mTSP por sus siglas en inglés).

En el enfoque seleccionado, se realizan dos grandes operaciones o etapas. En una primera etapa, se divide la red de contenedores en zonas independientes que respetan el límite de capacidad de los camiones usados mediante un algoritmo de barrido polar (*Sweep*), generando  $m$  subproblemas individuales tipo *Traveling Salesman Problem*, los cuales se resuelven en una segunda etapa considerando una autonomía infinita para los camiones que resuelven cada uno de ellos. En esta segunda operación, cada subproblema se resuelve mediante el uso del algoritmo de Recocido Simulado.

Los resultados experimentales, obtenidos sobre grafos sintéticos con distintas distribuciones espaciales, evidencian que el sistema propuesto permite reducir los tiempos de cómputo y generar rutas de alta calidad, manteniendo la viabilidad operativa incluso en instancias con centenares de nodos. Esto lo convierte en una enfoque prometedor para la resolución de este tipo de retos logísticos.

## Palabras clave:

- Optimización de rutas
- mTSP
- Recogida de residuos



# Índice de contenidos

<b>Índice de tablas</b>	<b>IX</b>
<b>Índice de figuras</b>	<b>XI</b>
<b>Índice de Algoritmos</b>	<b>XIII</b>
<b>1. Introducción</b>	<b>1</b>
<b>2. Estado del arte</b>	<b>5</b>
2.1. Trabajos similares . . . . .	8
<b>3. Metodología</b>	<b>9</b>
3.1. Definición . . . . .	9
3.2. Resolución . . . . .	11
3.3. Método de resolución del TSP . . . . .	12
3.4. Resolviendo el TSP . . . . .	13
<b>4. Zonificación</b>	<b>19</b>
<b>5. Experimentos</b>	<b>27</b>
5.1. Descripción del entorno experimental . . . . .	27
5.2. Comparación de métodos de zonificación . . . . .	31
5.2.1. Prueba 1: Escalado por tamaño del grafo . . . . .	31
5.2.2. Prueba 2: Escalado por capacidad de camión . . . . .	35
5.3. Experimentación con la librería CVRPLib . . . . .	39
<b>6. Conclusiones y trabajo futuro</b>	<b>45</b>
<b>Bibliografía</b>	<b>47</b>
<b>Apéndices</b>	<b>51</b>
<b>A. Librería: Ejecución de tests y benchmarks</b>	<b>53</b>
<b>B. Librería: Guía de uso</b>	<b>55</b>

B.1.	Librería	55
B.1.1.	Excepciones Personalizadas	55
B.1.2.	Estructuras de Datos	57
B.1.3.	Algoritmos de Optimización	61
B.2.	Utilidades	64
B.2.1.	Generación de Datasets Sintéticos	64
B.2.2.	Funciones Auxiliares	65
B.2.3.	Notas Importantes	65

# Índice de tablas

- |    |  |    |
|----|--|----|
| 1. | Resultados obtenidos sobre instancias de CVRPLIB . . . . . | 39 |
|----|--|----|



# Índice de figuras

1.	Ejemplo de zonas sobre un grafo de 294 y zonas de hasta 1500kg . . . . .	22
2.	Grafo distribuido en clústers . . . . .	28
3.	Grafo con distribución normal . . . . .	29
4.	Grafo con distribución uniforme . . . . .	30
5.	Comparativa del número medio de nodos encontrados en cada zona para distintas distribuciones de nodos. . . . .	32
6.	Comparativa del número medio de nodos encontrados en cada zona para distintas distribuciones de nodos. . . . .	32
7.	Número de zonas generadas en distintas distribuciones de nodos, . . . . .	33
8.	Comparativa del tiempo medio de ejecución del algoritmo <i>sweep</i> para distintas distribuciones de nodos. . . . .	33
9.	Comparativa del tiempo total de ejecución para distintas distribuciones de nodos. . . . .	34
10.	Comparativa del valor total obtenido por el algoritmo para distintas distribuciones de nodos. . . . .	34
11.	Evolución del número de zonas creadas con el cambio en la capacidad de los camiones para distintas distribuciones de nodos. . . . .	36
12.	Comparativa del tiempo de división de zonas del algoritmo <i>sweep</i> para distintas distribuciones de nodos. . . . .	36
13.	Comparativa del tiempo total de ejecución para distintas distribuciones de nodos. . . . .	37
14.	Comparativa del valor total obtenido por el algoritmo para distintas distribuciones de nodos. . . . .	38
15.	Rutas generadas por el sistema para las distintas instancias de la librería CVRPLIB. . . . .	41
16.	Rutas óptimas proporcionadas por la librería CVRPLIB. . . . .	42



# Índice de Algoritmos

1.	Estimación de la temperatura inicial en recocido simulado . . . . .	14
2.	Recocido Simulado para el Problema del Viajante . . . . .	16
3.	Zonificación por barrido angular . . . . .	20
4.	Postproceso de las zonas . . . . .	24



# 1

## Introducción

La recolección de residuos urbanos es un servicio esencial a nivel global que depende en gran medida de la planificación de las rutas de los vehículos usados. Optimizar la logística para minimizar costes y mejorar la calidad del servicio es crucial debido a la creciente urbanización a nivel global, lo cual hace que la gestión de residuos urbanos se vuelva un reto logístico de gran importancia en nuestra sociedad. Una recolección eficiente es esencial para la mejora de la calidad de vida de una ciudad debido a su influencia en la salubridad de esta y a las emisiones contaminantes asociadas a los vertederos y centros de procesamiento de residuos. Este trabajo propone un enfoque de toma inteligente de decisiones basado en datos con los que mejorar este servicio esencial.

Según un estudio del *World Bank*, la recogida de residuos urbanos representa entre el 20 % y el 50 % del presupuesto de gestión municipal [16]. Además, cada kilómetro adicional recorrido por los camiones de recogida supone no solo un incremento de costes operativos, sino también una fuente de emisión de gases de efecto invernadero, bien directamente en el uso de camiones con motores de combustión tradicional, o indirectamente en casos en los que los camiones cuentan con propulsión eléctrica. Estudios recientes estiman que el transporte de residuos urbanos genera entre **7 y 50 kg de CO<sub>2</sub> equivalentes por tonelada recogida**, con valores típicos alrededor de **25 kg/t** en distintas ciudades de Europa y Oriente Medio [12, 17]. Además, un estudio realizado en Madrid sugiere que el transporte de residuos supone aproximadamente un **10 %** del impacto climático total del ciclo de gestión, una cifra comparable a la fase de tratamiento [10]. Estos datos subrayan la importancia de mejorar la logística en la recogida de los residuos, aplicando soluciones algorítmicas que puedan contribuir significativamente a la reducción de emisiones, principalmente reduciendo la longitud de las

---

rutas utilizadas y mejorando su eficiencia, lo que reduciría también el impacto económico que supone este servicio para las ciudades modernas.

En muchas ciudades, la planificación de rutas se hace de forma manual o usando herramientas básicas que no siempre se adaptan correctamente a las necesidades presentadas por este reto. Esto provoca una gran falta de adaptabilidad al entorno complejo y creciente de una ciudad. A pesar de que se pueden paliar algunos de los defectos del planteamiento tradicional con medidas simples como la expedición de las rutas de recogida en horarios nocturnos, reduciendo el impacto del estado del tráfico, esto no es más que un simple parche que evita atacar al problema inicial de falta de información que se da en estas operativas en las ciudades.

Este desafío puede modelarse matemáticamente como un *Vehicle Routing Problem* (VRP), un problema clásico de la optimización combinatoria. El VRP busca determinar un conjunto de rutas óptimas o casi óptimas para una flota de vehículos que deben atender a un conjunto de clientes, partiendo y regresando a uno o más depósitos. Es una generalización del *Travelling Salesman Problem* (TSP), el cual consiste en encontrar el recorrido más corto que visita cada nodo exactamente una vez y regresa al punto de partida, y está clasificado como un problema *NP-hard*. Desde su formulación original por Dantzig y Ramser en 1959 [1], el VRP ha evolucionado hacia múltiples variantes para adaptarse a restricciones reales como capacidades, ventanas temporales, múltiples depósitos, entre otras [6, 14].

Una de las más relevantes en el contexto de la recogida de residuos es el *Capacitated Vehicle Routing Problem* (CVRP por sus siglas en inglés), que incorpora una limitación de capacidad en los vehículos. Esto significa que cada ruta generada debe respetar el límite máximo de carga que el camión puede transportar, lo que en este caso se traduce en la suma de los pesos de los contenedores asignados a esa ruta. Esta restricción es fundamental, y muy común; en problemas de recogida de residuos industriales, urbanos o de gran volumen, donde cada camión puede transportar un número limitado de contenedores o carga total definida.

Resolver el CVRP es extremadamente costoso cuando aumenta el número de nodos del problema debido a su naturaleza NP-Hard [8]. Como alternativa, en este trabajo se adopta un enfoque modular que separa el problema en dos fases: la división del grafo completo en zonas independientes que respetan la capacidad de los camiones, y la posterior resolución de un *Travelling Salesman Problem* (TSP) en cada zona. Esta simplificación del problema original pretende reducir significativamente la carga computacional sin renunciar a soluciones de alta calidad, a pesar de sacrificar la optimalidad que se puede obtener con un CVRP.

Este enfoque de descomposición se relaciona con el *Multiple Travelling Salesman Problem* (MTSP), un problema de optimización combinatoria donde  $m$

vendedores deben visitar colectivamente un conjunto de  $n$  ciudades, minimizando el coste total de recorrido, con la restricción de que cada ciudad sea visitada exactamente una vez por un único vendedor. El CVRP es una extensión del MTSP que incorpora restricciones de capacidad. Mientras que en el MTSP los agentes pueden visitar cualquier número de nodos sin límite, en el CVRP cada vehículo tiene una capacidad máxima  $Q$  que restringe la suma de demandas  $q_i$  de los nodos asignados a su ruta.

Este trabajo adopta, por tanto, una metodología que modela el problema como un grafo ponderado completo, cuyas zonas se dividen en función del peso total de los contenedores, respetando la capacidad del vehículo asignado. Una vez particionado el grafo, cada zona se resuelve individualmente mediante técnicas propias del TSP. Esta división no solo reduce el coste computacional, sino que permite una mayor escalabilidad y paralelización del proceso de optimización, elementos clave en contextos urbanos de alta densidad y variabilidad operativa.

No obstante, la modularidad del enfoque presenta una contrapartida: la posible pérdida de optimalidad global. Al dividir el grafo completo en zonas aisladas, se renuncia a explorar rutas potencialmente mejores que cruzan los límites entre zonas. Por ejemplo, dos contenedores cercanos entre sí pueden ser asignados a rutas distintas simplemente por pertenecer a zonas diferentes, generando recorridos redundantes.

El *Vehicle Routing Problem* se ha consolidado como uno de los problemas fundamentales en el ámbito de la optimización combinatoria y la logística operativa, surgiendo como una evolución del TSP introducida formalmente por Dantzig y Ramser en 1959 orientada a optimizar la planificación de rutas de una flota de camiones cisterna de gasolina [1]. A partir de esta formulación inicial, se han propuesto múltiples variantes del VRP para adaptarse a restricciones del mundo real, como la capacidad de los vehículos (CVRP), ventanas temporales (VRPTW), el uso de múltiples depósitos (MDVRP), restricciones de acceso, ambientales, entre muchas otras. [6, 14]

En lugar de basarse en reglas fijas que, como ha sido comentado anteriormente, no toman en cuenta el estado cambiante de las ciudades, la propuesta de este trabajo es un sistema de planificación basado en datos actuales sobre la distribución geográfica de los contenedores, su peso y la capacidad operativa de los camiones, permitiendo obtener una solución ajustada a la realidad, permitiendo cambios en función del nivel de llenado de los contenedores. Modelar una ciudad como un grafo permite aplicar no solo el enfoque propuesto en este trabajo, sino también otros similares que tienen la capacidad de mejorar ampliamente la calidad de vida de los ciudadanos.

El modelo propuesto en este trabajo puede evolucionar hacia un sistema completamente dinámico si se integra con tecnologías emergentes como el IoT. Este modelo es similar al propuesto en otros trabajos como [11], en el que se usan

---

distintos métodos de división de mapas para aplicar posteriormente algoritmos de mejora de rutas. También es similar al estudiado en [13], donde se propone una mejora a un método de división en zonas para lograr una mayor eficiencia en la asignación de distintas zonas. Las diferencias de este trabajo y los dos citados se expone en la sección 2.1.

Más allá del interés computacional, la división de una ciudad en zonas de recogida es una evolución natural de la tradicional división en distritos y barrios que ya se sigue hoy en día, pues deja de tomar estas divisiones como zonas inalterables y fijas que pueden suponer una limitación en las propuestas actuales de logística urbana.

El objetivo de este trabajo es el estudio de un método de resolución que permita establecer rutas para la recogida de los residuos urbanos teniendo en cuenta la capacidad de los camiones que realizarán estas rutas y el peso de los contenedores encontrados en el recorrido de estas.

El siguiente trabajo se estructura de la siguiente manera. En la Sección 2 se revisa el estado del arte en torno al problema del ruteo de vehículos y sus principales variantes, así como algunos de los algoritmos heurísticos comúnmente usados para este tipo de problemas. La Sección 3 describe el modelado final del problema y la metodología general. En la Sección 4 se presentan los algoritmos de zonificación implementados, incluyendo sus principios, funcionamiento y justificación. La Sección 5 detalla el enfoque de resolución adoptado para generar las rutas dentro de cada zona. En la Sección 6 se analizan los experimentos realizados y se comparan los resultados obtenidos entre los distintos métodos propuestos. Finalmente, en las Secciones 7 y 8 se recogen las conclusiones del trabajo y las posibles líneas de desarrollo futuro.

# 2

## Estado del arte

La recolección de residuos urbanos plantea un desafío logístico de naturaleza combinatoria. Modelar este problema adecuadamente permite aplicar herramientas matemáticas y computacionales para obtener rutas eficientes que minimicen tanto costes operativos como impacto ambiental. El marco más extendido para abordar este tipo de problemas es el *Vehicle Routing Problem (VRP)*.

El *Vehicle Routing Problem* (VRP) es un problema clásico de la teoría de grafos y la optimización combinatoria, formalizado por Dantzig y Ramser en 1959 [1]. En su forma básica, el VRP consiste en determinar un conjunto óptimo de rutas para una flota de vehículos que parte de un depósito central y debe visitar un conjunto de clientes o tareas (nodos), regresando nuevamente al depósito. El objetivo suele ser minimizar la distancia total recorrida, el coste o el tiempo de operación.

Matemáticamente, el problema se define sobre un grafo completo y no solo conectado debido a que existe una arista  $E$  entre cada par de nodos  $V$ ; y ponderado  $G = (V, E)$ , donde  $V = \{v_0, v_1, \dots, v_n\}$  representa el conjunto de nodos (con  $v_0$  siendo el depósito), y  $E$  el conjunto de aristas con pesos asociados que representan las distancias o costes entre nodos. La flota consta de  $k$  vehículos idénticos, y cada uno debe seguir una ruta que comience y termine en el depósito, atendiendo a un subconjunto de los clientes.

Este problema es una generalización del *Travelling Salesman Problem (TSP)*, donde un único viajante debe visitar todos los nodos una sola vez y volver al punto de partida, minimizando el recorrido total. El TSP es un problema *NP-hard*, y su dificultad aumenta factorialmente con el número de nodos, lo que implica que

---

el VRP, al ser aún más general, presenta una complejidad incluso mayor.

El *Capacitated Vehicle Routing Problem (CVRP)* extiende el VRP incluyendo una restricción adicional: cada vehículo tiene una capacidad máxima  $Q$ , y cada cliente un requerimiento de carga  $q_i$ , de modo que la suma de demandas en una ruta no puede superar dicha capacidad. Esta variante es especialmente relevante para la recogida de residuos, donde los camiones tienen un volumen o peso límite que no puede excederse. En el caso del CVRP, todos los vehículos parten de un mismo depósito y deben regresar a él tras completar su ruta.

Por su parte, el *Capacitated Multiple Travelling Salesman Problem (CMTSP)* es una extensión del TSP en la que varios vehículos deben recorrer distintos subconjuntos de clientes, partiendo de un punto y regresando al mismo. A pesar de que el CVRP y el CMTSP son problemas relacionados, no son equivalentes ya que el CVRP incorpora explícitamente las restricciones de capacidad y el CMTSP trata de asignar y optimizar una serie de rutas cerradas. En la práctica, algunos enfoques para el CVRP dividen el problema en subgrafos, asignando cada uno a un vehículo, y posteriormente resuelven un TSP dentro de cada subgrafo. Esta estrategia modular reduce la complejidad computacional, aunque puede sacrificar parte de la optimalidad global.

Resolver de forma exacta un VRP con restricciones, como el CVRP, constituye un problema NP-hard, lo que significa que no existe un algoritmo conocido que resuelva todas sus instancias en tiempo polinómico. Esto ha sido demostrado desde su formulación y se mantiene vigente incluso en variantes simplificadas del problema, como el TSP [7, 14].

En cuanto a complejidad computacional asintótica, el VRP en su forma general presenta:

$$O(k^n \cdot n!) \text{ (exploración completa con } k \text{ rutas y } n \text{ clientes)} \quad (1)$$

En contraste, el uso de enfoques modulares como el **CMTSP**, en los que se divide el grafo original en varias zonas más pequeñas (una por vehículo), permite resolver cada subproblema de forma independiente usando algoritmos más eficientes. Si cada zona contiene  $n_i$  nodos y se aplica, por ejemplo, una heurística tipo Nearest Neighbor ( $O(n_i^2)$ ) o un algoritmo de mejora como 2-opt, la complejidad total del proceso sería la suma de la complejidad de aplicar el algoritmo heurístico elegido sobre cada una de las subzonas.

Esta mejora es una de las principales motivaciones para adoptar estrategias tipo CMTSP: **se reduce considerablemente la complejidad computacional**, a costa de no garantizar la optimalidad global.

Resolver este tipo de problemas sugiere pues el uso de estrategias de resolución escalables y eficientes. Algunas técnicas efectivas se clasifican en función de su

objetivo; la zonificación del grafo inicial, la construcción inicial de rutas, la mejora local de estas rutas y una optimización global a través del uso de metaheurísticas, que son estrategias de resolución usadas para encontrar soluciones a problemas de optimización complejos en los que encontrar la solución óptima no siempre es posible. En el contexto del enfoque modular elegido en este trabajo, se puede seleccionar entre diversos algoritmos para cada uno de estos objetivos.

La primera etapa del proceso consiste en dividir el grafo en subconjuntos de nodos que serán atendidos por camiones independientes, dando lugar a subproblemas del tipo TSP o CTSP. Esta división puede producirse mediante el uso de varios algoritmos, entre los que se encuentra la técnica de barrido o *sweep*, que agrupa los nodos en función de su ángulo respecto al depósito, ordenándose en sentido horario tras transformar las coordenadas de cada nodo a coordenadas polares y agrupando nodos secuencialmente mientras se respetan las restricciones de capacidad de los vehículos [2]. Otro método es el de **barrido clásico**, presentado por Gillet y Miller [3] en el que se ordenan los nodos por ángulo creciente.

Una vez divididos los nodos en zonas, se procede a generar una solución inicial para cada una de ellas. En este proceso no se busca obtener la ruta óptima, sino una base razonable para posteriormente ser refinada mediante el uso de algoritmos más complejos. Una de las heurísticas más naturales que se usan para esta fase es *Nearest Neighbor*, algoritmo que simula visitar siempre el nodo más cercano al actual que aún no ha sido visitado. Este algoritmo es bastante rápido, con una complejidad de  $O(n^2)$ , pero puede generar rutas de baja calidad en casos en los que las zonas cuenten con nodos muy alejados del resto. Otra de las heurísticas más usadas es *Clarke-Wright*, en el que se parte de una ruta cíclica entre el depósito y cada nodo, calculando los ahorros de unir rutas entre sí. A pesar de contar con una mayor complejidad que Nearest Neighbor, ( $O(n^2 \log n)$ ), tiende a dar soluciones iniciales más robustas. [4]

A partir de estas rutas iniciales, se suelen aplicar algoritmos heurísticos de mejora local en los que se explora el vecindario de las soluciones anteriores en busca de una mejor. El más conocido es *2-opt*, que elimina pares de aristas y reconecta los caminos cortados en orden inverso para comprobar si esto reduce la longitud total. Cuenta con variantes como *3-opt* o *n-opt*, funcionalmente iguales, en los que se aumenta el número de aristas a reconnectar, aumentando la flexibilidad a cambio de una mayor complejidad. [4]

Cuando las heurísticas de mejora local ya no encuentran una solución mejor, se recurre a estrategias globales de optimización que permiten escapar de los mínimos locales. Estas estrategias son las conocidas como metaheurísticas. En el ámbito de este trabajo, se han usado dos de las más populares; *Simulated Annealing*, que es capaz de aceptar soluciones peores para escapar de estos mínimos usando una variable decreciente llamada temperatura y ha demostrado ser eficaz en zonas complejas con muchos mínimos locales, y *Tabu Search*, que explora posibles soluciones y las añade a una lista que evita que sean analizadas de nuevo

si ya se ha descartado. [4]

## 2.1. Trabajos similares

Como se mencionaba anteriormente, se han encontrado dos trabajos similares al presente. Estos son [11] y [13].

En [11], se proponen varios algoritmos alternativos, todos derivados del *sweep*, para dividir el grafo inicial, aplicando luego varias metaheurísticas para resolver cada subproblema. Este trabajo se diferencia del aquí expuesto ya que su principal objetivo es encontrar un algoritmo *sweep* que dé una mejor solución a los actuales.

Por su parte, [13] también se centra en proponer una alternativa para el algoritmo *sweep*, desarrollando una implementación modificada que considera puntos lejanos y grafos con nodos muy agrupados para generar divisiones mejores.

La diferencia principal entre estos trabajos y el desarrollado aquí es que ellos buscan mejorar el algoritmo *sweep*, mientras que aquí se busca aplicar *sweep* para solucionar instancias de grafos, buscando una solución más rápida que las encontradas con los algoritmos de resolución del CVRP.

# 3

## Metodología

### 3.1. Definición

Los problemas de planificación de rutas de recogida se tienden a modelar formalmente mediante un grafo ponderado, no dirigido y completo. Este grafo, denotado como  $G = (V, E)$ , representa el entorno operativo, donde:

- $V$  es el conjunto de vértices, cada uno correspondiente a un contenedor de residuos o cliente.
- $E$  es el conjunto de aristas, cada una correspondiente a una conexión directa entre dos vértices y ponderada según el coste (distancia de recorrido) entre ellos.

En este modelo, se asume la existencia de un único vértice especial  $V_0 \in V$ , que representa el **depósito** o punto central desde el que parte y al que vuelve cada camión.

Dado el conjunto de todos los nodos  $V$ , el objetivo inicial consiste en dividir el grafo  $G$  en un conjunto de subgrafos disjuntos  $\{G_1, G_2, \dots, G_M\}$  donde cada subgrafo será recorrido por un único vehículo. Se debe garantizar pues, que todos los vértices del grafo original estén contenidos en exactamente uno de los subgrafos generados:

$$G_i \subset G \quad \text{y} \quad \bigcup_{i=1}^M G_i = G \quad (2)$$

Asimismo, cada subgrafo  $G_i = (V_i, E_i)$  debe satisfacer una restricción de capacidad. Es decir, la suma del peso de los vértices que lo componen debe ser menor que la capacidad máxima del camión asignado. Formalmente:

$$\sum_{v_j \in V_i} p(v_j) \leq T \quad (3)$$

donde  $T$  representa la capacidad de un camión y  $p(v_j)$  el peso asociado a cada nodo  $v_j \in V_i$ .

Posteriormente, se debe encontrar una ruta para cada subgrafo  $G_i$  que recorra todos sus vértices exactamente una vez, minimizando el coste total. Esta ruta, denotada como  $R_i$ , debe cumplir las siguientes condiciones:

$$\begin{aligned} \forall v_j \in V_i &\Rightarrow v_j \in R_i \\ \forall v_j \in R_i &\Rightarrow v_j \in V_i \end{aligned} \quad (4)$$

La solución completa al problema implica entonces dos fases: una partición eficiente del grafo original en subgrafos factibles, y la optimización de la ruta en cada uno de ellos, de modo que se minimice el coste total del sistema en cuanto a la distancia total recorrida.

Para simplificar el problema y hacerlo computacionalmente abordable, se adoptan una serie de restricciones razonables dentro del dominio de la recolección urbana de residuos:

1. **Capacidad uniforme.** Se asume que todos los camiones tienen la misma capacidad  $T$ .
2. **Depósito único.** Todo vehículo parte y regresa a un único punto central, el depósito  $V_0 \in V$ . No se considera en este trabajo el uso de múltiples depósitos.
3. **Grafo completo.** Se considera que el grafo  $G$  es *completo*, es decir, existe una arista directa entre cualquier par de nodos. Esto habilita el uso directo de heurísticas del TSP:

$$\forall (v_i, v_j) \in V \Rightarrow \exists \text{ camino entre } v_i \text{ y } v_j \quad (5)$$

4. **Autonomía ilimitada.** Se considera que todos los vehículos disponen de la misma autonomía, suficiente para recorrer cualquier subgrafo asignado.

Esto equivale a afirmar:

$$A = \infty; \quad \forall C \quad (6)$$

donde  $A$  representa la autonomía de los camiones  $C$

Estas simplificaciones del problema permiten centrar el enfoque en las decisiones algorítmicas de partición del grafo y optimización de rutas, dejando para futuros trabajos aspectos como restricciones de autonomía, acceso vial o ventanas temporales.

## 3.2. Resolución

Aunque el problema general es un CVRP, este trabajo no lo aborda como una única instancia global, sino mediante una **reformulación práctica** basada en la estrategia de dividir para conquistar. Específicamente, se sigue una estrategia del tipo *cluster-first, route-second* inspirada en el CMTSP. En esta estrategia primero se divide el grafo original en  $k$  subzonas y se asigna cada una a un vehículo. Después, de forma independiente se busca una ruta para cada subzona mediante el uso de metaheurísticas. Esto es una descomposición del problema basada en el CMTSP, pero no una formulación de este.

Esta decisión se justifica tanto por razones teóricas como prácticas. Para empezar, la complejidad del CVRP es exponencial, volviendo su resolución prohibitiva en cuanto a costes computacionales. Además, dividir el grafo en zonas factibles permite aplicar heurísticas eficientes del TSP, con una complejidad significativamente inferior, facilitando la escalabilidad u paralelización del cálculo de rutas. Como se indicó en la Sección 2.1, se han encontrado trabajos que siguen este enfoque, y, aunque como se expone en la sección referenciada no buscan lo mismo que este trabajo, si que muestran que un enfoque así proporciona resultados prometedores.

Por tanto, este enfoque permite resolver subproblemas más manejables sin sacrificar significativamente la calidad global de la solución, aunque se aleje de la optimalidad que se podría obtener al usar un método de resolución del CVRP; lo que lo convierte en una estrategia factible.

Una vez generada una ruta  $R = [v_0, v_1, \dots, v_n]$  sobre un subgrafo  $G_s$ , la cual debe ser un **ciclo Hamiltoniano**, pues debe recorrer todos los nodos  $v_n$  del subgrafo  $G_s$  y regresar al nodo inicial, es necesario evaluar su calidad para guiar el proceso de optimización. La función de evaluación implementada en este trabajo calcula el **coste total de la ruta**, definido en este caso como la **suma de las distancias euclidianas bidimensionales** entre los vértices consecutivos del recorrido, incluyendo el retorno al vértice inicial  $v_0$ .

Formalmente, dado un recorrido  $R = [v_0, v_1, \dots, v_n]$ , el coste total de la ruta se define como:

$$f(R) = d(v_n, v_0) + \sum_{i=0}^{n-1} d(v_i, v_{i+1}) \quad (7)$$

donde la función  $d(v_i, v_j)$  representa la distancia entre dos nodos del grafo. Esta distancia entre dos nodos  $v_i$  y  $v_j$ , cuyas coordenadas cartesianas son  $(x_i, y_i)$  y  $(x_j, y_j)$ , se calcula mediante la fórmula estándar de la distancia euclíadiana en dos dimensiones:

$$d(v_i, v_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \quad (8)$$

Este valor representa el coste de desplazamiento entre dichos vértices. En este trabajo se asume un entorno plano y a velocidad constante, por lo que el coste se interpreta directamente como distancia de viaje. La función  $f(R)$  define por tanto la **aptitud** de una ruta y constituye la métrica principal a minimizar en los algoritmos de mejora local y en las metaheurísticas aplicadas en los grafos. Esta elección es coherente con los objetivos del sistema, que prioriza rutas más cortas y eficientes en términos de distancia recorrida.

Una vez particionado el grafo en zonas factibles y definida la función de evaluación, cada subgrafo  $G_i$  es resuelto individualmente mediante un algoritmo seleccionado por su eficacia en la resolución de los TSP.

### 3.3. Método de resolución del TSP

Para resolver los subproblemas del tipo TSP que se generan se pueden aplicar muchos tipos de algoritmos. Los tres más comunes son las heurísticas constructivas, las metaheurísticas y los métodos exactos. Cada uno de estos grupos tiene sus propias ventajas y desventajas.

Las heurísticas constructivas incluyen algoritmos como *Nearest Neighbor* y se caracterizan por su rapidez. Sin embargo, este enfoque de resolución tiende a provocar que las soluciones acaben atrapadas en óptimos locales, produciendo soluciones alejadas del óptimo local.

Por otra parte, los métodos exactos garantizan la obtención de una solución óptima, pero cuentan con una complejidad mucho más elevada, lo que causa que estos métodos necesiten grandes tiempos de ejecución para dar con un resultado.

Las metaheurísticas presentan pues un término medio entre los métodos ex-

puestos anteriormente, proporcionando mecanismos de búsqueda sofisticados que permiten explorar el espacio de soluciones y evitar caer en óptimos locales. Esto tiene como consecuencia tiempos de ejecución mayores que el de las heurísticas constructivas, pero estos tiempos se mantienen menores que en el caso de los métodos exactos. Es por este término medio que en este trabajo se ha decidido usar una metaheurística para encontrar las rutas sobre los subgrafos producidos tras la división del grafo inicial.

Entre las numerosas metaheurísticas existentes, se ha decidido usar *Simulated Annealing*. Esta decisión se debe a la relativa simplicidad de este algoritmo, tanto conceptualmente como en implementación. Además, este algoritmo cuenta con mecánicas que le permiten escapar de óptimos locales, pues es capaz de valorar soluciones que son peores a priori pero pueden derivar en mejores rutas finales. Además, en comparación con otros algoritmos metaheurísticos como la Búsqueda Tabú, que requieren el uso de una extensa lista de soluciones ya probadas, *Simulated Annealing* requiere de un menor uso de memoria.

En la Sección 3.4 se explica en qué consiste este algoritmo y se explica la implementación específica realizada para este proyecto.

### 3.4. Resolviendo el TSP

Como se ha indicado en la sección anterior, el enfoque de solución para los problemas TSP generados aplica un algoritmo metaheurístico con el objetivo de obtener la mejor solución posible para cada zona. El algoritmo utilizado es *Simulated Annealing*, partiendo de una ruta inicial aleatoria para cada uno de los subgrafos a resolver.

La implementación de Simulated Annealing desarrollada para este propósito presenta varias modificaciones respecto al esquema clásico propuesto por Kirkpatrick et al. [5]. La primera diferencia es el cálculo de la temperatura inicial, obtenido a partir de un número de movimientos posibles, y su probabilidad de aceptación deseada. La fórmula seguida es:

$$T_0 = -\frac{\sum_{q=1}^Q (C' - C)_q}{Q \ln(P_0)} \quad (9)$$

donde  $Q$  es el número de movimientos muestreados,  $P_0$  es la probabilidad de aceptación y  $(C' - C)_q$  es la diferencia del coste asociado al  $q$ -ésimo movimiento evaluado. muestra. [15].

Esta fórmula da lugar al siguiente pseudocódigo:

---

**Algoritmo 1** Estimación de la temperatura inicial en recocido simulado

---

```

1: function ESTIMATE_INITIAL_TEMPERATURE( $P_0$ ,  $camino_{inicial}$ )
2:    $Q \leftarrow \max(\lfloor \frac{\lceil \text{len}(camino_{inicial}) \rceil}{10} \rfloor, 1)$                                  $\triangleright$  Número de muestras
3:    $\Delta \leftarrow$  lista vacía
4:   for  $q \leftarrow 1$  hasta  $Q$  do
5:     seleccionar aleatoriamente dos índices  $i, j$  tal que  $i < j$ 
6:      $camino' \leftarrow \text{FLIP}(camino, i, j)$ 
7:      $C \leftarrow \text{EVALUATE}(camino)$ 
8:      $C' \leftarrow \text{EVALUATE}(camino')$ 
9:      $\delta_q \leftarrow |C' - C|$ 
10:    añadir  $\delta_q$  a  $\Delta$ 
11:   end for
12:    $\bar{\Delta} \leftarrow \text{MEDIA}(\Delta)$ 
13:    $T_0 \leftarrow -\frac{\bar{\Delta}}{\ln(P_0)}$ 
14:   return  $T_0$ 
15: end function

```

---

En el cual se escoge un número de movimientos a probar a partir de la longitud del camino inicial aleatorio ( $camino_{inicial}$ ) como se ve en la línea 2. Posteriormente se inicializa una lista  $\Delta$  que contendrá todas las diferencias en valor entre el camino inicial y el camino resultante de aplicar un movimiento. Después, se ejecuta un bucle  $Q$  veces en el que se seleccionan dos índices  $i, j$  para realizar un movimiento sobre  $camino_{inicial}$ , obteniendo  $camino'$  y calculando su diferencia absoluta en valor  $\delta_q$ , el cual se añadirá a la lista  $\Delta$  (líneas 4-10).

Tras esto, se calcula el valor medio ( $\bar{\Delta}$  de todos los valores de  $\Delta$  y se resuelve la fórmula inicial para obtener la temperatura inicial  $T_0$  (líneas 12-14).

Además de este cambio en el algoritmo respecto a la versión original, se establece un criterio de parada del algoritmo alternativo. Este es un número de iteraciones sin encontrar una mejora en el valor de la ruta calculada respecto a la mejor, como se hace en [18].

Por último, se establece el valor del factor de enfriamiento  $\alpha$  en 0,99, que corresponde a valores dentro del intervalo planteado en [18].

Por último, se han usado dos estrategias distintas para la búsqueda de soluciones candidatas, ya que se ha demostrado experimentalmente que aplicar esto en diferentes fases del algoritmo produce mejores resultados [9]. En la implementación usada se ha usado el operador *flip*, el cual se intercambia el orden en el que se recorren los nodos comprendidos entre dos índices  $i, j$  elegidos aleatoriamente. Esta función se aplica hasta que se alcanza una temperatura  $T < 10^{-8}$ , momento en el cual se empieza a usar el operador *swap*, que genera rutas alternativas intercambiando la posición de dos nodos  $i, j$ , elegidos aleatoriamente.

### Capítulo 3. Metodología

---

Con estos cambios y valores expuestos, se explica el siguiente pseudocódigo, correspondiente a la implementación de *Simulated Annealing* utilizada en el proyecto:

**Algoritmo 2** Recocido Simulado para el Problema del Viajante

---

```

1: function SIMULATED_ANNEALING(path, maxstagnated,  $\alpha$ , maxiterations)
2:   current_path  $\leftarrow$  path
3:   current_value  $\leftarrow$  EVALUATE(current_path)
4:   best_path  $\leftarrow$  current_path
5:   best_value  $\leftarrow$  current_value
6:   if temperature = None then
7:     temperature  $\leftarrow$  ESTIMATE_INITIAL_TEMPERATURE(0.9, path)  $\triangleright$  Ver
    Pseudocódigo 1
8:   end if
9:   stagnated  $\leftarrow$  0
10:  it  $\leftarrow$  0
11:  while stagnated  $<$  maxstagnated & it  $<$  maxiterations do
12:    Seleccionar i, j aleatoriamente con i  $<$  j
13:    if temperature  $<$   $10^{-8}$  then
14:      next_path  $\leftarrow$  SWAP(current_path, i, j)
15:    else
16:      next_path  $\leftarrow$  FLIP(current_path, i, j)
17:    end if
18:    next_value  $\leftarrow$  EVALUATE(next_path)
19:     $\delta \leftarrow$  next_value  $-$  current_value
20:    accept  $\leftarrow$  False
21:    if  $\delta < 0$  then
22:      accept  $\leftarrow$  True
23:    else
24:       $x \leftarrow -\delta/\text{temperature}$ 
25:      prob  $\leftarrow$  exp(x)
26:      if random[0, 1]  $<$  prob then
27:        accept  $\leftarrow$  True
28:      end if
29:    end if
30:    if accept then
31:      current_path  $\leftarrow$  next_path
32:      current_value  $\leftarrow$  next_value
33:      if current_value  $<$  best_value then
34:        best_path  $\leftarrow$  current_path
35:        best_value  $\leftarrow$  current_value
36:        stagnated  $\leftarrow$  0
37:      else
38:        stagnated  $\leftarrow$  stagnated + 1
39:      end if
40:    else
41:      stagnated  $\leftarrow$  stagnated + 1
42:    end if
43:    temperature  $\leftarrow$  temperature  $\times$   $\alpha$ 
44:    it  $\leftarrow$  it + 1
45:  end while return best_path, best_value
46: end function

```

---

Siguiendo el pseudocódigo, vemos que se inicializan las variables correspondientes a la ruta inicial y a su valor, y se establecen estos como la mejor ruta y valor encontrados respectivamente (líneas 2-5). A continuación, se escoge un valor para la temperatura inicial (ver 1) y se establece el número de iteraciones e iteraciones sin mejora a 0.

Una vez establecido el valor de las variables al inicio, se procede a ejecutar el bucle principal, en el que se buscará una ruta mejor a la inicial iterativamente. El bucle comienza generando una nueva ruta a partir de la actual. Esta es elegida usando una función, la cual depende de la temperatura actual, como ya se expuso anteriormente (líneas 12-17). Después, se calcula el valor de la nueva ruta y la diferencia entre este valor y el valor de la mejor ruta encontrada (líneas 18-19). Si la diferencia  $\delta$  es negativa o si se cumple:

$$\text{random}[0, 1) < p = e^{\frac{-\delta}{t}} \quad (10)$$

donde  $t$  es la temperatura actual, se establecerá la ruta evaluada como la actual, y si es la mejor se pondrá el contador de iteraciones estancadas a cero (líneas 20-36). Si por el contrario no se cumple  $\delta < 0$  ni 10 o la ruta aceptada no es mejor que la mejor encontrada, se aumentará en uno las iteraciones estancadas (líneas 37-41). Por último, se recalcula la temperatura aplicando el factor de enfriamiento  $\alpha$  y se aumenta en uno el número de iteraciones totales ejecutadas.

Al acabar la ejecución, se devolverá el valor de la mejor ruta encontrada, así como el orden de visita de los nodos del subgrafo de dicha ruta.



# 4

## Zonificación

La primera fase de resolución del problema consiste en dividir el grafo completo  $G = (V, E)$ , que representa la red de contenedores de residuos, en un conjunto de zonas  $\{Z_1, Z_2, \dots, Z_M\}$ , de modo que cada una pueda ser asignada a un único camión. Esta partición tiene como objetivo principal garantizar que la carga total de cada zona no supere la capacidad  $T$  de los vehículos disponibles, así como facilitar la coherencia espacial de las zonas, facilitando así su posterior optimización.

La estrategia de zonificación adoptada en este trabajo se basa en un enfoque geométrico de barrido polar o *sweep* [3], donde los nodos se ordenan por su ángulo polar respecto a un punto de referencia (en este caso, el depósito  $V_0$ ) y se agrupan secuencialmente en zonas hasta agotar la capacidad de los camiones  $T$ . Esta técnica demuestra ser efectiva para generar zonas contiguas, compactas, estructuradas espacialmente y con pocos solapamientos en problemas de logística urbana.

En este trabajo, el ángulo polar  $\theta$  de cada nodo  $v_i$  con respecto al depósito  $V_0$  se define como:

$$\theta(v_i) = \text{atan2}(y_i - y_0, x_i - x_0) \quad (11)$$

donde  $(x_i, y_i)$  son las coordenadas del nodo  $v_i$  y  $x_0, y_0$  las del depósito  $V_0$ . Este ángulo se utiliza como criterio de ordenación para construir las zonas.

Este enfoque presenta varias ventajas prácticas. Su simplicidad computacional la hace eficiente para grafos de tamaño medio-grande gracias a su complejidad

---

de  $O(n \log n)$ . Además, permite la generación de zonas como un proceso independiente a la optimización de rutas. Sin embargo, su desempeño depende fuertemente de la distribución espacial de los nodos. En mapas urbanos no simétricos, la dirección de barido puede producir zonas muy distintas, lo cual justifica la comparación de ambos enfoques implementados en este trabajo.

La implementación técnica se compone de varias fases sucesivas. El procedimiento se ejecuta sobre el conjunto de nodos del grafo excluyendo al depósito, que será incluido posteriormente en cada una de las zonas generadas. El proceso completo se resume en el siguiente pseudocódigo, adaptado del flujo propuesto por Gillet & Miller en su trabajo de 1974 [3]:

---

### Algoritmo 3 Zonificación por barrido angular

---

```

1: function DIVIDE_GRAPH(nodos, centro, T, sentido)
2:   for cada  $v$  en nodos do
3:     calcular ángulo  $\theta_v \leftarrow \text{atan2}(y_v - y_0, x_v - x_0)$ 
4:   end for
5:   if sentido es ascendente then
6:     ordenar nodos por  $\theta$  creciente
7:   else
8:     ordenar nodos por  $\theta$  decreciente
9:   end if
10:  zonas  $\leftarrow$  lista vacía
11:  zona_actual  $\leftarrow$  lista vacía
12:  carga  $\leftarrow 0$ 
13:  for nodo en nodos ordenados do
14:    if carga + peso(nodo)  $> T$  then
15:      añadir [centro] + zona_actual a zonas
16:      zona_actual  $\leftarrow$  [nodo]
17:      carga  $\leftarrow$  peso(nodo)
18:    else
19:      añadir nodo a zona_actual
20:      actualizar carga
21:    end if
22:  end for
23:  if zona_actual no está vacía then
24:    añadir [centro] + zona_actual a zonas
25:  end if
26:  zonas  $\leftarrow$  MERGE_SMALL_ZONES(zonas, T)
27:  zonas  $\leftarrow$  POSTPROCESS_ZONES(zonas, T)
28:  return zonas
29: end function

```

---

La primera fase corresponde al cálculo del ángulo polar de cada nodo en rela-

ción con  $V_0$  utilizando la función `atan2` de python, lo que garantiza que el ángulo obtenido se encuentra correctamente definido en el rango  $[-\pi, \pi]$ , abarcando así todas las posibles direcciones en el plano. Este proceso corresponde a las líneas 2 a 4. La función `atan2` es un procedimiento propio del lenguaje python que corresponde a la siguiente definición matemática:

$$atan2(\Delta y, \Delta x) = \begin{cases} \arctan \frac{\Delta y}{\Delta x} & \text{if } \Delta x > 0, \\ \arctan \frac{\Delta y}{\Delta x} + \pi & \text{if } \Delta x > 0 \text{ and } \Delta y \leq 0, \\ \arctan \frac{\Delta y}{\Delta x} - \pi & \text{if } \Delta x < 0 \text{ and } \Delta y < 0, \\ +\frac{\pi}{2} & \text{if } \Delta x = 0 \text{ and } \Delta y > 0, \\ -\frac{\pi}{2} & \text{if } \Delta x = 0 \text{ and } \Delta y < 0, \\ \text{undefined} & \text{if } \Delta x = 0 \text{ and } \Delta y = 0. \end{cases} \quad (12)$$

Una vez obtenidos los ángulos, los nodos se ordenan de forma ascendente o descendente, según la variante del algoritmo que se desee aplicar (ver líneas 5-9 del pseudocódigo). Esta ordenación determina el recorrido del barrido sobre el plano. Formalmente:

- En el caso **ascendente**:

$$\theta(v_1) < \theta(v_2) < \dots < \theta(v_n), \quad \forall v_i \in V \quad (13)$$

- En el caso **descendente**:

$$\theta(v_1) > \theta(v_2) > \dots > \theta(v_n), \quad \forall v_i \in V \quad (14)$$

Esta ordenación angular crea una secuencia determinista de nodos que se agruparán progresivamente en función de la capacidad de los camiones  $T$ , formando las zonas definitivas.

La estrategia ascendente realiza un barrido en sentido horario, comenzando por el nodo que forma el ángulo menor. En términos computacionales, se realiza una ordenación ascendente de los ángulos de los nodos. Este método es particularmente eficaz en grafos con distribución uniforme o simétrica alrededor del depósito, donde se espera que los nodos más cercanos angularmente también estén próximos en el espacio euclíadiano.

Por otra parte, el enfoque descendente invierte el sentido del barrido, comenzando desde el ángulo máximo y ordenando los nodos de forma decreciente, produciendo una segmentación del grafo distinta aunque el conjunto de entrada sea el mismo a la división ascendente. Ambos métodos se comparan en la Sección 5, valorando tanto el coste de sus soluciones, como el tiempo de ejecución.

Las siguientes figuras muestran un ejemplo gráfico del resultado de ambos métodos sobre un grafo de 294 nodos con una distribución uniforme y una capa-

---

ciudad de los camiones  $T = 1,500\text{kg}$ , elegida arbitrariamente en esta sección para mostrar las diferencias entre las dos implementaciones, ascendente y descendente. El depósito  $V_0$  se marca con una cruz verde y cada zona esta dividida por una línea discontinua y se representa con un color distinto.

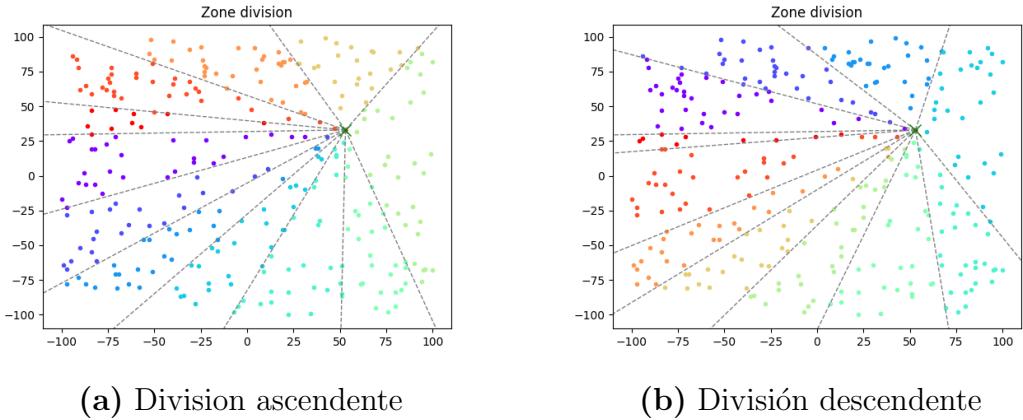


Figura 1: Ejemplo de zonas sobre un grafo de 294 y zonas de hasta 1500kg

Una vez ordenados, los nodos son procesados secuencialmente (líneas 10 a 22). Se crea una nueva zona mientras la suma acumulada del peso de los contenedores no supere la capacidad de los camiones. Cuando se excede este límite, se finaliza la zona actual y se inicia una nueva. Esta operación garantiza que cada subgrafo generado cumple con la restricción de capacidad impuesta en el problema. Este paso genera un conjunto inicial de zonas:

$$Z = \{Z_1, Z_2, \dots, Z_m\} \quad \text{t.q.} \quad \sum_{v \in Z_i} \text{peso}(v) \leq T, \quad \forall i \in \{1, \dots, m\} \quad (15)$$

donde  $T$  es la capacidad máxima de carga de los camiones.

Tras el agrupamiento, el nodo correspondiente al depósito  $V_0$  se inserta en cada zona como primer nodo (líneas 23-25 del pseudocódigo). Esto permite que la resolución posterior del TSP en cada subgrafo se realice partiendo y terminando en el depósito, como se exige por el modelo.

Cabe destacar que las zonas generadas inicialmente pueden presentar casos subóptimos, como zonas con pocos contenedores o con cargas desequilibradas. Para mejorar esta estructura, se aplica un **refinamiento posterior** compuesto por dos funciones.

La primera función, *merge\_small\_zones* (línea 26), identifica las zonas pequeñas, considerado como zonas pequeñas aquellas con dos o menos nodos (excluyendo el depósito), y las intenta fusionar con otras zonas. Formalmente, una

zona pequeña es aquella en la que se cumple:

$$|Z_i| - 1 \leq 2 \iff |Z_i| \leq 3 \quad (16)$$

Tras ejecutar esta función, se ejecuta otro procedimiento, *postprocess\_zones* (línea 27), que analiza los primeros y últimos nodos de cada zona y los intenta reubicar en las zonas vecinas si hacerlo mejora el equilibrio de carga y no supera el límite de capacidad. Para clarificar este concepto, se explica su pseudocódigo a continuación:

---

**Algoritmo 4** Postproceso de las zonas

---

```
function POSTPROCESS(zones, T)
    for  $Z_i \in zones$  do
         $A \leftarrow Z_{i-1}$ 
         $B \leftarrow Z_i$ 
         $C \leftarrow Z_{i+1}$ 
        if  $B_{size} > 3 \ \& \ (A_{size} > 3 \mid C_{size} > 3)$  then
            if  $(A_{weight} + weight(B_0)) < T$  then
                move( $A, B_0$ )
            end if
            if  $(C_{weight} + weight(B_{-2})) < T$  then
                move( $C, B_{-2}$ )
            end if
        end if
        if  $B_{size} \leq 3$  then
            if  $(A_{weight} + weight(B_0)) < T$  then
                move( $A, B_0$ )
            end if
            if  $(C_{weight} + weight(B_{-2})) < T$  then
                move( $C, A_{-2}$ )
            end if
            if  $(A_{weight} + weight(B_{-2})) < T$  then
                move( $A, B_{-2}$ )
            end if
            if  $(C_{weight} + weight(B_0)) < T$  then
                move( $C, B_0$ )
            end if
        end if
        if isEmpty( $A$ ) then
            zones.remove(zone)
        end if
    end for
    return zones
end function
```

---

Antes de continuar, se especifica el significado de los distintos símbolos usados. La función  $weight$  devuelve el peso del nodo  $x$ .  $B$  hace referencia a la  $i$ -ésima zona contenida en  $zones$ .  $A$  y  $C$  son la zona anterior y posterior a  $B$ , respectivamente.  $A_x \mid B_x \mid C_x$  corresponden al contenedor  $x$  de las zonas  $A$ ,  $B$  o  $C$ , y por último,  $T$  es la capacidad de los camiones.

Como se mencionaba anteriormente, el objetivo de esta función es equilibrar el peso de las zonas. Para ello, se analizan dos casos principales.

En el primer caso, la zona actual  $B$  tiene más de dos contenedores, y sus zonas adyacentes  $A$  y  $C$  también. En esta situación, se intentan pasar los contenedores en los límites de las zonas (a partir de ahora, **contenedores frontera**)  $B_0$  y  $B_{-2}$  a la zona anterior y siguiente respectivamente. La función *move* se encarga de esto mismo, mover un contenedor a otra zona. Cabe destacar que el motivo de que el contenedor frontera sea  $B_{-2}$  y no  $B_{-1}$  es porque este último es el depósito, el cual pertenece a todas las zonas.

El segundo caso involucra cuatro comprobaciones y trata de eliminar las zonas con menos de dos nodos. Estas cuatro comprobaciones corresponden a los posibles movimientos a realizar, ya que en este caso ambos contenedores son contenedores frontera. Estos movimientos se realizarán **si y solo si** no hacen que la zona a la que se mueven los contenedores supere la capacidad de nuestros camiones  $T$ . Los movimientos que se intentan realizar son cuatro: mover  $B_0$  a la anterior zona  $A$ , mover  $B_{-2}$  a la siguiente zona  $C$ , mover  $B_{-2}$  a la anterior zona  $A$  y mover  $B_0$  a la siguiente zona  $C$ .

Por último, al acabar de procesar todas las zonas se comprueba si alguna está vacía (destacar que en este caso, una zona vacía es una zona que solo contiene el depósito), y si es el caso, se elimina del listado de zonas.

Este refinamiento tiene como finalidad evitar divisiones que puedan resultar ineficientes, además de lograr una estructura de zonas más homogénea que favorezca la posterior fase de optimización de rutas.

El enfoque de zonificación adaptado es una estrategia eficiente y escalable, aplicable a entornos urbanos. Su simplicidad de implementación y su baja complejidad computacional lo convierten en un método robusto para preprocesar el grafo y dividirlo antes de aplicar heurísticas más costosas. Ambos enfoques producen soluciones validas y compatibles con el modelo planteado en el proyecto, dividiendo el problema inicial en varios subproblemas. En la Sección 5 se presentan los resultados comparativos entre ambas variantes, incluyendo diversas métricas usadas en la comparación de los métodos.



# 5

## Experimentos

El objetivo principal de esta sección es evaluar el rendimiento del enfoque modular propuesto en los apartados anteriores. Para ello, se han llevado a cabo dos experimentos principales que analizan el comportamiento del sistema bajo distintas configuraciones de entrada, midiendo su escalabilidad, eficiencia computacional y calidad de las soluciones obtenidas.

Ambos experimentos se han ejecutado empleando la misma arquitectura algorítmica: un proceso de zonificación del grafo (*sweep* ascendente o descendente), seguido de la resolución de rutas sobre cada zona utilizando el enfoque expuesto en la Sección 3.2, ejecutando el algoritmo metaheurístico *Simulated Annealing*. Las métricas seleccionadas permiten medir no solo la calidad de las rutas generadas (valor de la solución), sino también el comportamiento estructural de la partición (tamaño de zonas) y la eficiencia temporal del sistema (tiempos de zonificación y total).

### 5.1. Descripción del entorno experimental

Las pruebas se han realizado en un equipo con las siguientes características técnicas:

- **Procesador:** Intel Core i7-12700H — 12 núcleos @2.3-4.5GHz
- **Memoria RAM:** 32GB DDR4

- **Sistema operativo:** Ubuntu 22.04.4 LTS
- Python 3.11, con una implementación propia de los algoritmos

Cada prueba se ha ejecutado sobre instancias de grafos cuyos nodos y depósito han sido ubicados en coordenadas aleatorias dentro de las restricciones de cada tipo de grafo, las cuales se expondrán a continuación. Estos grafos se encuentran en un plano bidimensional con una dimensión de  $200 \times 200$ , creando un total de 40,000 posibles ubicaciones para los nodos de cada grafo. Se han utilizado tres tipos de distribuciones espaciales para los grafos. Estas distribuciones son en **clústers**, siguiendo una distribución **normal** y siguiendo una distribución **uniforme**.

En los dos experimentos que se expondrán a continuación se han realizado un total de 10 ejecuciones por cada instancia de grafo con el fin de eliminar posibles variaciones en los resultados debidas a demoras en la ejecución ajenas al propio programa.

A continuación, se explican las diferencias entre los tres tipos de grafos utilizados.

En el caso de los grafos cuyos nodos están distribuidos en clústers, estos se encuentran agrupados en varias regiones densas (agrupaciones) con un centro aleatorio para cada una de ellas. El número de clústers generados depende del número de nodos del grafo, siendo este una décima parte del total de nodos, eligiendo los centros de las agrupaciones de forma aleatoria y distribuyendo nodos en una región alrededor del punto central de esta. En la imagen inferior, se puede observar un grafo de 300 nodos, divididos en tres clústers. El depósito ( $V_0$ ) está marcado con una cruz verde y se ubica en las coordenadas  $(0, -50)$ .

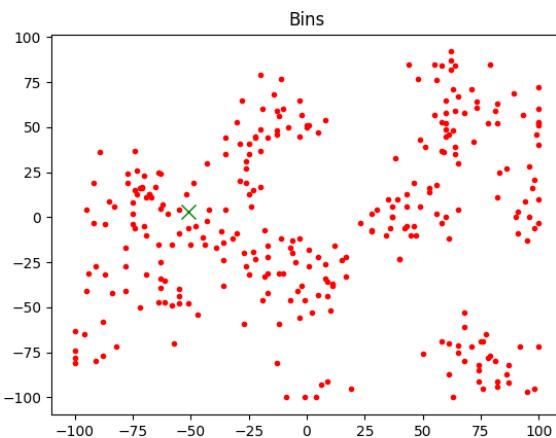


Figura 2: Grafo distribuido en clústers

En el caso de los grafos que siguen una distribución normal, los nodos se generan siguiendo una curva gaussiana. Esta curva está creada con los valores siguientes:

$$\mu = \frac{x + y}{2} \quad (17)$$

$$\sigma = \frac{x - y}{6} \quad (18)$$

donde  $x$  e  $y$  son dos valores definidos en la generación que permiten alterar la apariencia de los grafos generados con esta distribución. De este modo, y gracias a la desviación estándar elegida, se garantiza que alrededor del 99,7 % de los nodos generados se a  $\pm 3$  desviaciones estándar del punto central del plano  $(0, 0)$ . La siguiente imagen muestra un ejemplo de un grafo que sigue esta distribución, con el centro marcado con una cruz verde en las coordenadas  $(27, 36)$ .

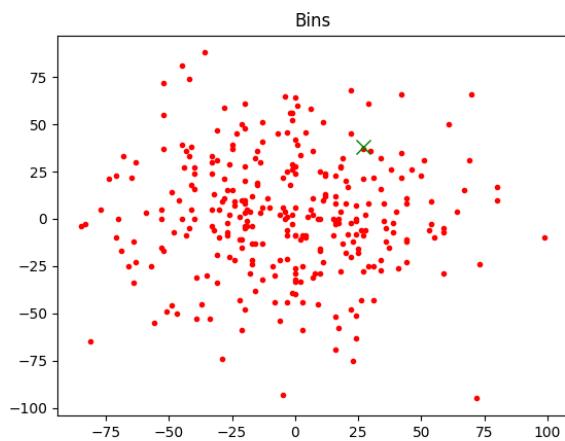


Figura 3: Grafo con distribución normal

Finalmente, los grafos generados con una distribución uniforme tienen sus nodos generados de forma completamente aleatoria, donde cualquier punto del plano tiene las mismas probabilidades de contener un nodo. Se muestra una imagen de esta distribución con el centro marcado mediante una cruz verde en las coordenadas  $(-30, -45)$ .

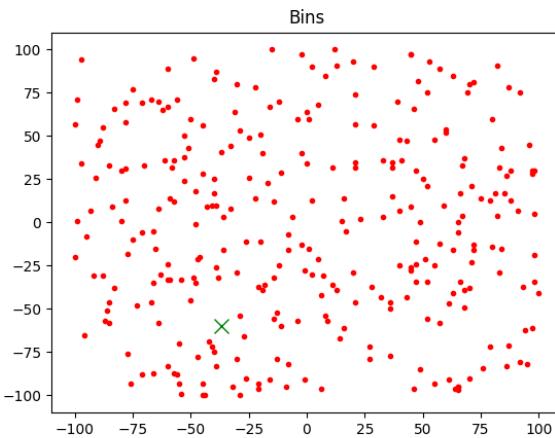


Figura 4: Grafo con distribución uniforme

Se han realizado dos pruebas para comparar los métodos de *sweep* ascendente y *sweep* descendente. En la primera de ellas se estudia el rendimiento del sistema ante grafos de diferentes tamaños, desde 25 hasta 275 nodos y de los tres tipos que se han expuesto anteriormente. El peso de los nodos de los grafos de esta prueba es un valor aleatorio contenido en el rango [100, 250], y los camiones utilizados cuentan siempre con la misma capacidad  $T = 1,500$ .

En la segunda prueba se han fijado tres grafos, todos ellos con 350 nodos, y cada uno siguiendo un tipo de distribución entre las expuestas. En esta prueba se ha variado la capacidad máxima de los camiones, desde 1,000kg hasta 15,000kg, con una granularidad de 100kg. Los nodos de los grafos tendrán un peso aleatorio dentro del rango [100, 250], al igual que en la primera prueba. Este test permite analizar la relación entre la capacidad de carga y el número y estructura de las zonas resultantes, así como su impacto en el coste y los tiempos de cómputo.

Además, se ha realizado un último experimento en el que se ejecuta el enfoque desarrollado en este proyecto en diferentes instancias de la librería *CVRPLIB*<sup>1</sup>, la cual proporciona distintos grafos junto con la capacidad de los agentes que se deben usar, así como la solución óptima a dichas instancias. Debido a que los grafos de esta librería cuentan con una capacidad y número de nodos de los grafos preestablecidas, no se han incluido estos grafos en las dos pruebas explicadas anteriormente.

---

<sup>1</sup><https://vrp.galgos.inf.puc-rio.br/index.php/en/>

## 5.2. Comparación de métodos de zonificación

En esta sección se presentan las dos pruebas en relación con el escalado por tamaño del grafo y el escalado por la capacidad de camiones que se han explicado en la sección anterior.

### 5.2.1. Prueba 1: Escalado por tamaño del grafo

Esta prueba permite evaluar el escalado temporal del sistema (exponencial o lineal), el tipo de distribución que genera zonas más equilibradas y la evolución del coste agregado de las rutas conforme crece el tamaño de los grafos, así como el tiempo de ejecución total, el cual es especialmente interesante para las instancias más grandes. A continuación, se muestran las gráficas con los resultados de los experimentos.

Comenzando con los resultados acerca de la distribución de las zonas, se analizan dos métricas: el número mínimo y medio de nodos por zona. No se ha considerado el número máximo de nodos por zona ya que al mantener la capacidad de los camiones en 1,500 es esperable que el máximo de nodos por zona sea siempre igual o muy similar.

Analizando el número mínimo de nodos por zona a lo largo de las diferentes configuraciones del experimento, se observa que este valor oscila entre 2 y 9 nodos, con una mayor concentración en torno a los 7 y 8 nodos. Esta distribución está fuertemente condicionada por la capacidad de los camiones ( $T$ ) y por la distribución de pesos asignados a los nodos. Dado que los contenedores tienen pesos variables pero moderadamente acotados, la mayoría de las zonas tienden a llenarse al incluir entre 7 y 8 nodos antes de alcanzar el umbral de capacidad cuando el peso de estos se encuentra en el rango de [100, 250] y la capacidad de los camiones se mantiene en 1,500. Esto implica que, en la práctica, el proceso de zonificación produce subconjuntos de tamaño reducido, lo que favorece la eficiencia del enfoque de este proyecto al generar instancias pequeñas de TSP, más fáciles de resolver. La existencia de zonas con solo 2 o 3 nodos suele deberse a la presencia de contenedores particularmente pesados que limitan la capacidad restante de la zona, o bien a situaciones limítrofes de agrupamiento al final del barrido. Esta tendencia valida que el algoritmo *sweep* tiende a generar zonas homogéneas en tamaño, lo cual es beneficioso para el balance de carga entre rutas.

## 5.2. Comparación de métodos de zonificación

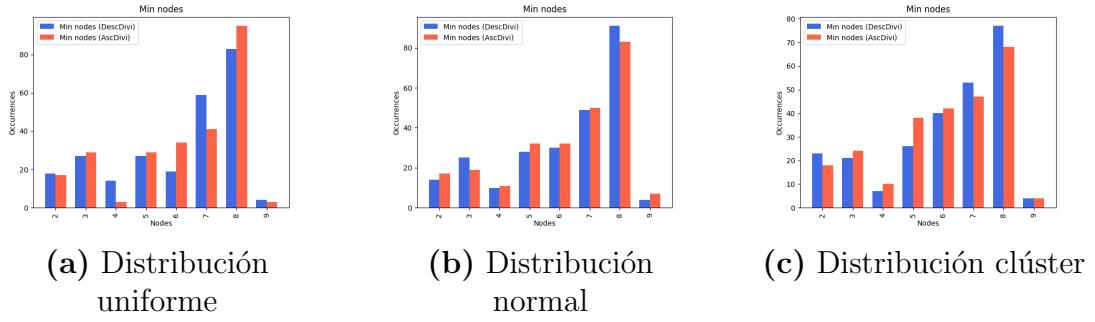


Figura 5: Comparativa del número medio de nodos encontrados en cada zona para distintas distribuciones de nodos.

En cuanto al número medio de nodos por zona, este oscila de forma más estable entre 7 y 9, con una clara concentración entre los valores de 8 y 9 nodos por zona. Este comportamiento, en combinación con la concentración del mínimo entre 6 y 8 nodos, refuerza la hipótesis de que la zonificación por barrido angular tiende a generar zonas de tamaño uniforme en términos de número de vértices, independientemente de la distribución espacial del grafo o de la capacidad exacta usada en cada prueba. Esta regularidad estructural es una propiedad muy deseable dentro del enfoque modular tipo CMTSP, ya que permite aplicar algoritmos de resolución homogéneos, mejorar el balance de carga entre vehículos y facilita tanto la planificación paralela como la escalabilidad del sistema. Además, zonas con un número moderado de vértices (en este caso, entre 7 y 9) permiten aplicar el algoritmo de *simulated annealing* seleccionado en este proyecto sin producir tiempos de ejecución muy altos, ya que el comportamiento de este hace que el tiempo en encontrar una solución aumente mucho con el tamaño del grafo sobre el que se busca una ruta. Esto permite no comprometer significativamente la calidad de la solución.

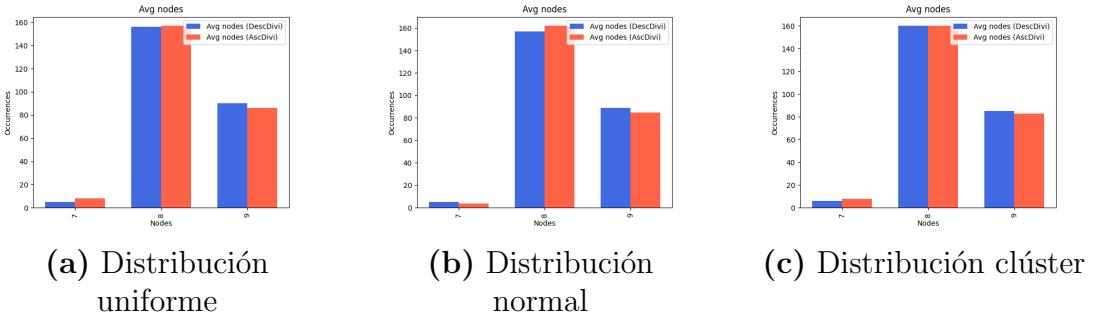


Figura 6: Comparativa del número medio de nodos encontrados en cada zona para distintas distribuciones de nodos.

Viendo los resultados para el número de zonas, se ve una tendencia lineal a aumentar el número de zonas creadas cuando se mantiene la misma capacidad

para los camiones, sin mucha diferencia entre grafos con distribuciones distintas. Este hecho corrobora la tendencia lineal que esta métrica presenta. Estos resultados son esperables y sirven a modo de confirmación de que la tendencia observada es esperada cuando se mantiene una demanda y capacidad igual.

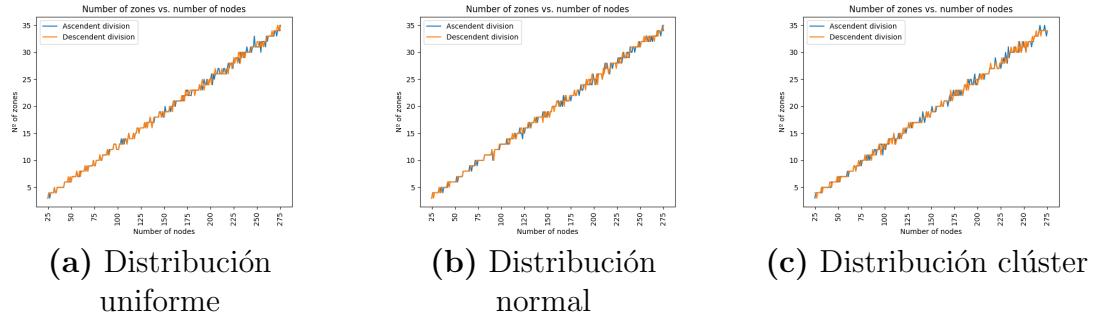


Figura 7: Número de zonas generadas en distintas distribuciones de nodos,

A continuación, se presentan los resultados correspondientes a las métricas del tiempo. Cabe destacar que todos los tiempos están expresados en milisegundos.

En el tiempo de división se puede observar una relación exponencial entre el tamaño del grafo y el tiempo de división. Esto tiene sentido pues la función *sweep* tiene una complejidad de  $O(n \log n)$ , que puede verse encarecida por el postproceso aplicado. Cabe destacar que las diferencias entre los tiempos de instancias con número de nodos similar se debe a la propia naturaleza aleatoria de los grafos empleados.

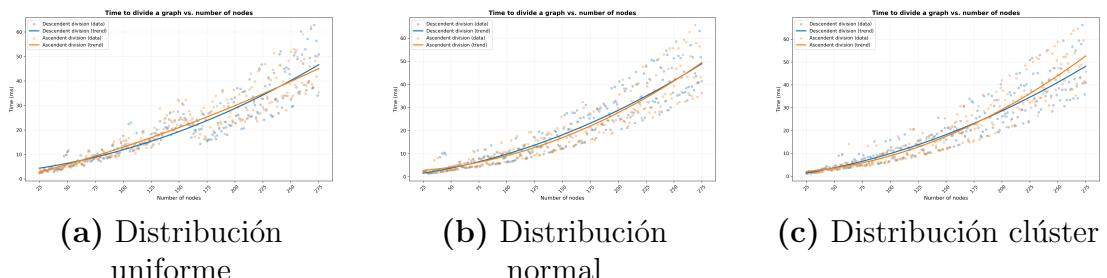


Figura 8: Comparativa del tiempo medio de ejecución del algoritmo *sweep* para distintas distribuciones de nodos.

*Nota:* El eje *y* de las gráficas expresa el tiempo transcurrido en *milisegundos*

A pesar del crecimiento del número de zonas y del incremento del tamaño del grafo, el tiempo total de resolución muestra una tendencia lineal al alza. Esto se debe a que el enfoque utilizado en este trabajo divide el problema original en subproblemas más pequeños (zonas), lo que limita el tamaño de cada TSP a resolver. Dado que los algoritmos aplicados en cada subgrafo tienen complejidades

como  $O(n^2)$ , pero operan sobre conjuntos acotados de nodos, el crecimiento global del tiempo tiende a ser lineal con respecto al número total de contenedores. Esta propiedad confirma la escalabilidad del enfoque modular adoptado, especialmente frente a formulaciones globales del CVRP. También cabe destacar el pequeño impacto que supone la operación *sweep* frente al tiempo total de resolución.

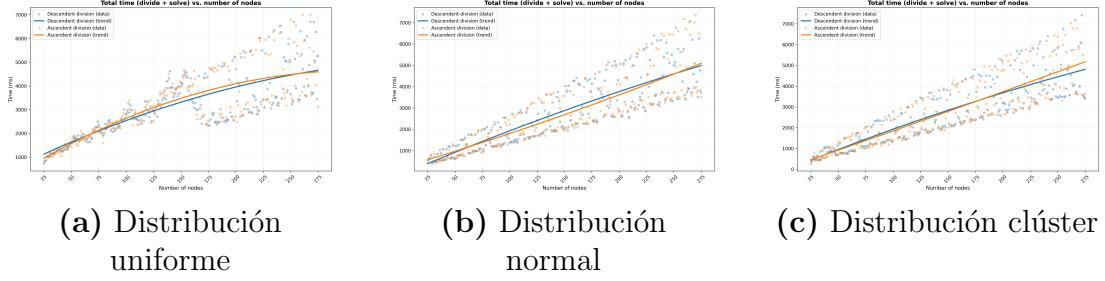


Figura 9: Comparativa del tiempo total de ejecución para distintas distribuciones de nodos.

*Nota:* El eje *y* de las gráficas expresa el tiempo transcurrido en *milisegundos*

Además del tiempo total, también se observa una tendencia lineal en el **valor total de las soluciones obtenidas**, es decir, en la suma de las distancias recorridas por todos los camiones. Este crecimiento lineal es consistente con la expansión progresiva del grafo, ya que a medida que se añaden más contenedores, es necesario recorrer distancias adicionales para cubrirlos. Dado que los nuevos nodos se distribuyen en el espacio de forma razonablemente uniforme (o agrupada, dependiendo del experimento), y que cada uno requiere ser visitado una única vez, el aumento del coste total tiende a ser proporcional al número de nodos añadidos. En otras palabras, si se duplica el número de contenedores, se requiere aproximadamente el doble de recorrido para cubrirlos, siempre que la eficiencia de las rutas se mantenga. Esta proporcionalidad también refleja la robustez del enfoque modular empleado: al mantener la calidad relativa de las rutas en cada zona, el sistema escala de forma predecible en términos de distancia total recorrida.

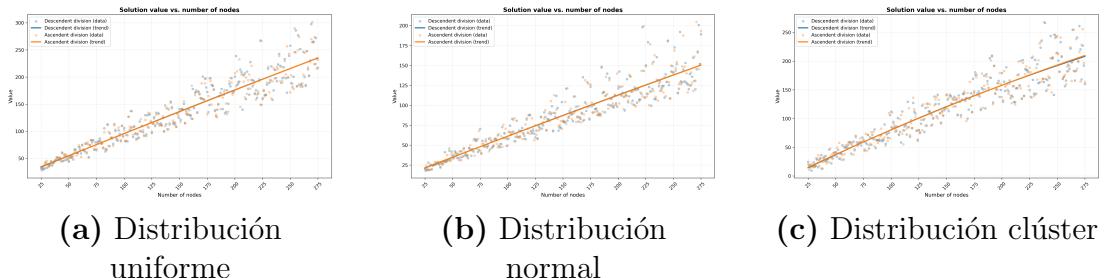


Figura 10: Comparativa del valor total obtenido por el algoritmo para distintas distribuciones de nodos.

*Nota:* El eje *y* de las gráficas expresa el valor de la solución encontrada.

A partir de los resultados obtenidos en este experimento, se pueden extraer varias conclusiones relevantes sobre el comportamiento de la zonificación por bárrido angular ante grafos de diferente tamaño. En primer lugar, se observa que el **tiempo de ejecución de la etapa de zonificación tiende a crecer de forma casi exponencial**, en línea con la complejidad teórica del algoritmo Sweep, cuya ordenación inicial presenta una complejidad de  $O(n \log n)$  y cuyas etapas de fusión y postprocesado pueden escalar desfavorablemente en función del número de zonas. No obstante, el **tiempo total de ejecución del sistema** (zonificación y resolución de rutas) muestra un crecimiento más **proporcional y tendencialmente lineal**, lo que sugiere una buena escalabilidad del enfoque modular CMTSP en el que se basa esta propuesta. En cuanto a la estructura de las zonas generadas, tanto el número **mínimo** como el **promedio de nodos por zona** se mantienen de forma relativamente estable, concentrándose principalmente entre 7 y 9 nodos, lo cual indica que el método produce divisiones uniformes y bien balanceadas en número de vértices. Finalmente, cabe destacar que no se observan diferencias significativas entre las tres distribuciones espaciales analizadas (uniforme, normal y en clústers), lo que refuerza la **robustez y generalidad del enfoque de zonificación** propuesto, capaz de adaptarse a distintos patrones urbanos sin necesidad de ajustes específicos.

### 5.2.2. Prueba 2: Escalado por capacidad de camión

El análisis realizado en esta prueba resulta crucial para observar como la zonificación responde ante distintos límites de carga, identificando si la eficiencia de la solución mejora con camiones de mayor capacidad y detectando si aparecen cuellos de botella computacionales al reducir el número de zonas y, por ende, aumentar su tamaño. A continuación, se muestran las gráficas con los resultados de los experimentos.

Comenzando con los resultados sobre el número de zonas creadas, los resultados muestran una **tendencia descendente no lineal** en la cantidad de zonas a medida que aumenta dicha capacidad. Concretamente, la curva resultante presenta una forma de **campana invertida suavizada**, donde la reducción en el número de zonas es rápida en los primeros incrementos de capacidad y se vuelve progresivamente más estable conforme se alcanzan valores altos. Este comportamiento es coherente con la lógica del problema: al aumentar la capacidad de carga de los vehículos, **cada zona puede incluir un mayor número de nodos**, lo que reduce la necesidad de crear múltiples zonas para cubrir todos los vértices del grafo. Sin embargo, el descenso no es constante ni lineal, ya que factores como la distribución de pesos de los nodos y la lógica secuencial del algoritmo de zonificación pueden introducir puntos de inflexión. En conjunto, estos resultados validan la adaptabilidad del algoritmo de zonificación frente a diferentes capacidades operativas, mostrando una reducción efectiva en el número de subproblemas

generados sin comprometer la validez de las zonas.

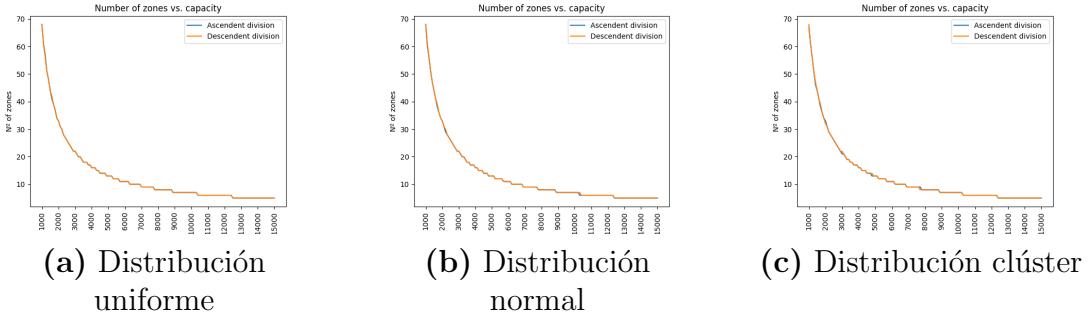


Figura 11: Evolución del número de zonas creadas con el cambio en la capacidad de los camiones para distintas distribuciones de nodos.

El tiempo de división en esta prueba muestra una evolución muy similar a la de la prueba anterior, manteniéndose dentro de un rango y con una tendencia casi constante, con algunas oscilaciones. Esta estabilidad se explica por el hecho de que la complejidad computacional del algoritmo de barrido angular *sweep* depende principalmente del número de nodos del grafo, y no tanto de la capacidad de los camiones. Dado que en este experimento el número de vértices del grafo permanece fijo, el proceso de cálculo de ángulos, ordenación y asignación secuencial de nodos a zonas, que domina la complejidad del algoritmo *sweep* también se mantiene constante. Aunque el número final de zonas sí varía con la capacidad, los pasos computacionales necesarios para recorrer y agrupar los nodos son prácticamente los mismos. Por tanto, es lógico que los tiempos de división no presenten un descenso significativo incluso cuando el número de zonas se reduce. Esto demuestra además la robustez y escalabilidad del algoritmo seleccionado en contextos con diferentes capacidades operativas, permitiendo obtener resultados estables en tiempos controlados.

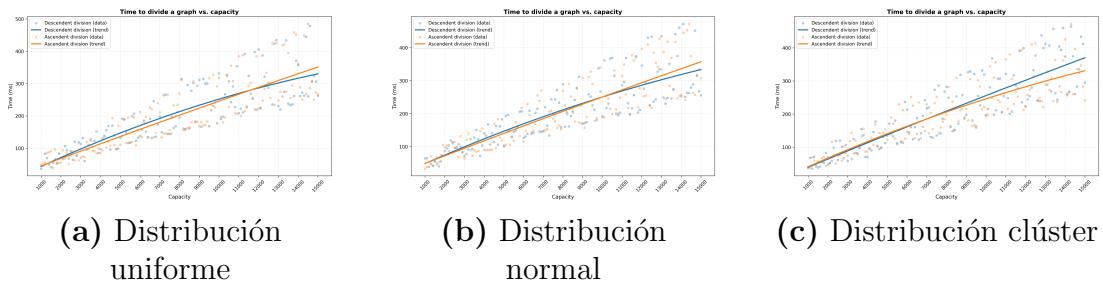


Figura 12: Comparativa del tiempo de división de zonas del algoritmo *sweep* para distintas distribuciones de nodos.

*Nota:* El eje *y* de las gráficas expresa el tiempo transcurrido en *milisegundos*

El tiempo total de ejecución presenta una curva con forma característica:

**desciende rápidamente** en las primeras capacidades, se estabiliza a partir de un cierto punto medio y luego muestra una **ligera tendencia ascendente** hacia los valores más altos de capacidad. Este comportamiento puede ser el resultado de dos fuerzas opuestas. Por un lado, al aumentar la capacidad de los camiones, se reduce el número total de zonas, lo que implica menos rutas a optimizar y, por tanto, menores tiempos de cómputo en las fases posteriores al barrido. Esta es la causa principal del descenso inicial. Sin embargo, a medida que las zonas se vuelven más grandes y densas (al incluir más nodos), el tiempo requerido para resolver cada subgrafo también aumenta, ya que el algoritmo aplicado tiene complejidad creciente con el tamaño del subgrafo. Esto explica la estabilización y el ligero incremento posterior del tiempo total: aunque hay menos zonas, las rutas son internamente más complejas. El resultado es una curva con mínimo local, que sugiere que existe una capacidad óptima en la que se equilibra el número de zonas con la complejidad de cada una.

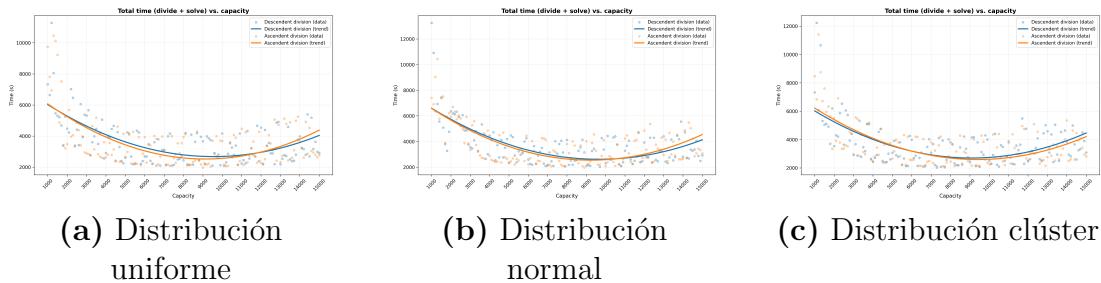


Figura 13: Comparativa del tiempo total de ejecución para distintas distribuciones de nodos.

*Nota:* El eje *y* de las gráficas expresa el tiempo transcurrido en *milisegundos*

Aunque las tendencias generales en los gráficos de tiempo de división y tiempo total son claras, se observa una variabilidad considerable entre puntos consecutivos a lo largo de todo el experimento. Esta dispersión no es puntual, sino que está **presente de forma continua**, generando oscilaciones alrededor de la tendencia principal. Estas variaciones pueden explicarse por el comportamiento **sensible del algoritmo de zonificación** frente a pequeños cambios en la capacidad. Dado que el proceso de barrido angular depende de la acumulación secuencial del peso de los nodos, **incluso una diferencia mínima en la capacidad** puede alterar la forma en la que se agrupan los nodos, afectando tanto al número de zonas generadas como a su estructura. Esto, a su vez, modifica el número y tipo de rutas que deben resolverse, impactando directamente en el tiempo total de ejecución. A pesar de estas oscilaciones, la **tendencia global** es claramente reconocible, lo que indica que el sistema mantiene un comportamiento consistente en su escalabilidad.

Finalmente, el valor total de la solución (es decir, la suma de las longitudes de todas las rutas generadas) sigue una **curva decreciente con forma de campana**.

**na invertida** a medida que aumenta la capacidad de los camiones. Este comportamiento es coherente con lo esperado: al tener vehículos con mayor capacidad, se pueden generar **menos zonas** y, por tanto, **menos rutas independientes**, reduciendo el número de trayectos de ida y vuelta al depósito y mejorando la eficiencia global del sistema. A partir de cierta capacidad, esta mejora se **estabiliza**, ya que las zonas no pueden crecer indefinidamente sin comprometer otros factores logísticos como la distancia. Por ello, a partir de una capacidad intermedia, el valor de la solución tiende a estabilizarse sin mostrar una tendencia claramente creciente ni decreciente. Esta estabilización sugiere la existencia de un punto de retorno decreciente, más allá del cual seguir aumentando la capacidad no supone una reducción significativa en la distancia total recorrida.

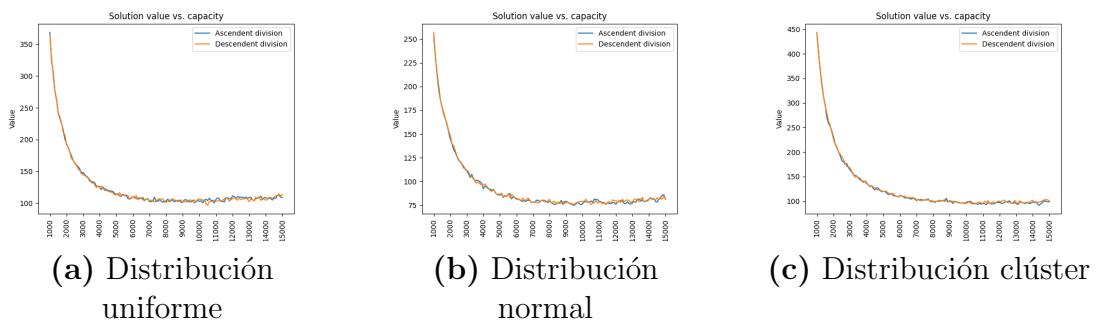


Figura 14: Comparativa del valor total obtenido por el algoritmo para distintas distribuciones de nodos.

*Nota:* El eje *y* de las gráficas expresa el valor de la solución encontrada.

Esta segunda prueba ha permitido analizar cómo se comporta el sistema ante distintas capacidades máximas de los camiones, manteniendo constante el grafo. Los resultados muestran que el número de zonas decrece siguiendo una curva con forma de campana descendente, ya que un mayor volumen de carga permite agrupar más nodos por camión, reduciendo la fragmentación del grafo. Esta reducción en el número de zonas impacta también en el tiempo total de ejecución, que disminuye rápidamente al principio del experimento y se estabiliza a partir de cierto umbral de capacidad. Sin embargo, se detecta una leve tendencia creciente en la parte final, posiblemente debido al aumento del tamaño de las zonas, lo que conlleva rutas más complejas de resolver. En cuanto al valor de la solución, se observa una evolución similar, con una caída progresiva que se estabiliza sin mostrar una tendencia claramente ascendente en su tramo final. Estas evidencias confirman que aumentar la capacidad mejora la eficiencia del sistema hasta cierto punto, a partir del cual los beneficios marginales se reducen.

Cabe destacar que no se han encontrado **diferencias significativas** entre las divisiones ascendentes y descendentes ni en la calidad de las soluciones, ni en el número de zonas generadas, ni en los tiempos de ejecución, incluso en el tiempo de división. Dado que ambos enfoques generan soluciones correctas y eficientes, se

**puede optar indistintamente por uno u otro.** No obstante, la simplicidad y reproducibilidad del enfoque ascendente lo convierten en una buena elección por defecto, al no observarse ventajas objetivas de rendimiento en la alternativa descendente.

### 5.3. Experimentación con la librería CVRPLib

La simplificación del enfoque CVRP empleada en este trabajo se ha evaluado sobre seis instancias de la biblioteca pública CVRPLIB<sup>2</sup>, ampliamente utilizada como referencia estándar en la validación de algoritmos para problemas de ruteo de vehículos. Cada instancia presenta una combinación única de número de nodos, capacidad máxima de vehículos y configuración espacial, lo que permite poner a prueba la robustez del sistema propuesto.

El enfoque presentado en este trabajo divide el grafo en zonas factibles y aplica posteriormente una resolución local a cada subproblema. La siguiente tabla resume los resultados obtenidos, comparando el número de zonas generadas automáticamente con el número mínimo de rutas requeridas ( $k$ ) para que la capacidad total cubra la demanda total del grafo:

Instancia	Número de nodos (n)	Mínimo zonas (k)	Zonas obtenidas	Tiempo resolución (seg)
X-n115-k10	114	10	11	1.74
X-n157-k13	156	13	13	2.17
X-n209-k16	208	16	16	2.86
X-n256-k16	255	16	17	3.28
X-n393-k38	392	38	39	5.84
X-n856-k95	856	95	95	15.81

Tabla 1: Resultados obtenidos sobre instancias de CVRPLIB

Los resultados presentados en la Tabla 1 muestran que el método propuesto es capaz de generar una división de zonas eficiente y consistente con las capacidades de los vehículos para instancias estándar de la biblioteca CVRPLIB. En la mayoría de los casos, el número de zonas generadas coincide exactamente con el número mínimo requerido (según el parámetro  $k$  de cada instancia), y en los casos en que se supera ligeramente dicho valor, el exceso es mínimo (una única zona adicional), lo cual puede justificarse por ligeras variaciones en la distribución espacial de los nodos o por restricciones de balanceo de carga dentro del algoritmo de particionado. Estos resultados sugieren que el enfoque de zonificación aplicado es razonablemente robusto y adecuado para su propósito, ofreciendo soluciones que respetan las restricciones de capacidad sin sacrificar la coherencia espacial.

En cuanto al rendimiento temporal, cabe destacar que los tiempos de ejecución se han mantenido en rangos muy bajos, incluso en las instancias de mayor

---

<sup>2</sup><https://vrp.galgos.inf.puc-rio.br/index.php/en/>

tamaño. Por ejemplo, la instancia  $X\text{-}n856\text{-}k95$ , con casi 900 nodos y 95 vehículos con rutas a calcular, ha sido resuelta en menos de 16 segundos. Estos resultados son especialmente relevantes si se tiene en cuenta que el sistema no ha sido paralelizado y ha ejecutado los procesos de forma completamente secuencial. Por tanto, existe un margen claro de mejora en cuanto a rendimiento computacional mediante la aplicación de técnicas de paralelización, por ejemplo, ejecutando en paralelo los procesos de resolución de zonas, lo cual permitiría escalar el sistema a entornos de producción o escenarios urbanos aún más complejos sin comprometer la eficiencia temporal.

Para visualizar el comportamiento del sistema en instancias estándar, la Figura 15 muestra las rutas generadas para cuatro de los seis casos extraídos (a excepción de las instancias  $X\text{-}n393\text{-}k38$  y  $X\text{-}n856\text{-}k95$ , pues se generan tantas rutas que el resultado no resulta legible) de la librería CVRPLIB. Se observa cómo el sistema divide el grafo inicial en zonas coherentes, espacialmente agrupadas y asignadas a vehículos independientes. Cada color representa una ruta individual que parte y regresa al depósito, cumpliendo con la estructura del problema. Las visualizaciones refuerzan la validez de la zonificación aplicada y permiten verificar visualmente la correcta cobertura del conjunto de nodos, así como la ausencia de solapamientos entre zonas. Este análisis cualitativo complementa los resultados numéricos presentados anteriormente, reforzando la efectividad y escalabilidad del enfoque.

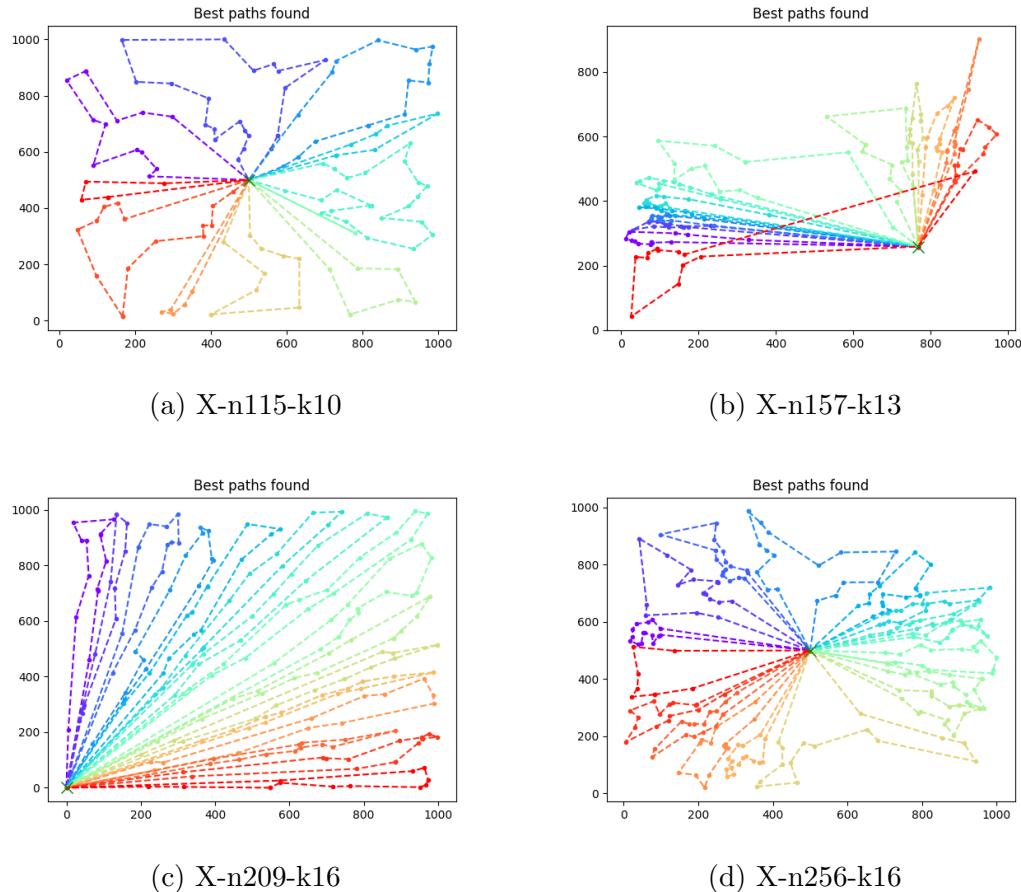


Figura 15: Rutas generadas por el sistema para las distintas instancias de la librería CVRPLIB.

*Nota:* Cada color indica una ruta independiente.

A continuación, se presentan las representaciones gráficas oficiales de las rutas óptimas proporcionadas por CVRPLIB para las mismas instancias utilizadas en este experimento. Estas figuras permiten una comparación visual directa entre las rutas generadas por el sistema propuesto y las soluciones óptimas de referencia, facilitando la evaluación cualitativa del comportamiento espacial de las rutas obtenidas.

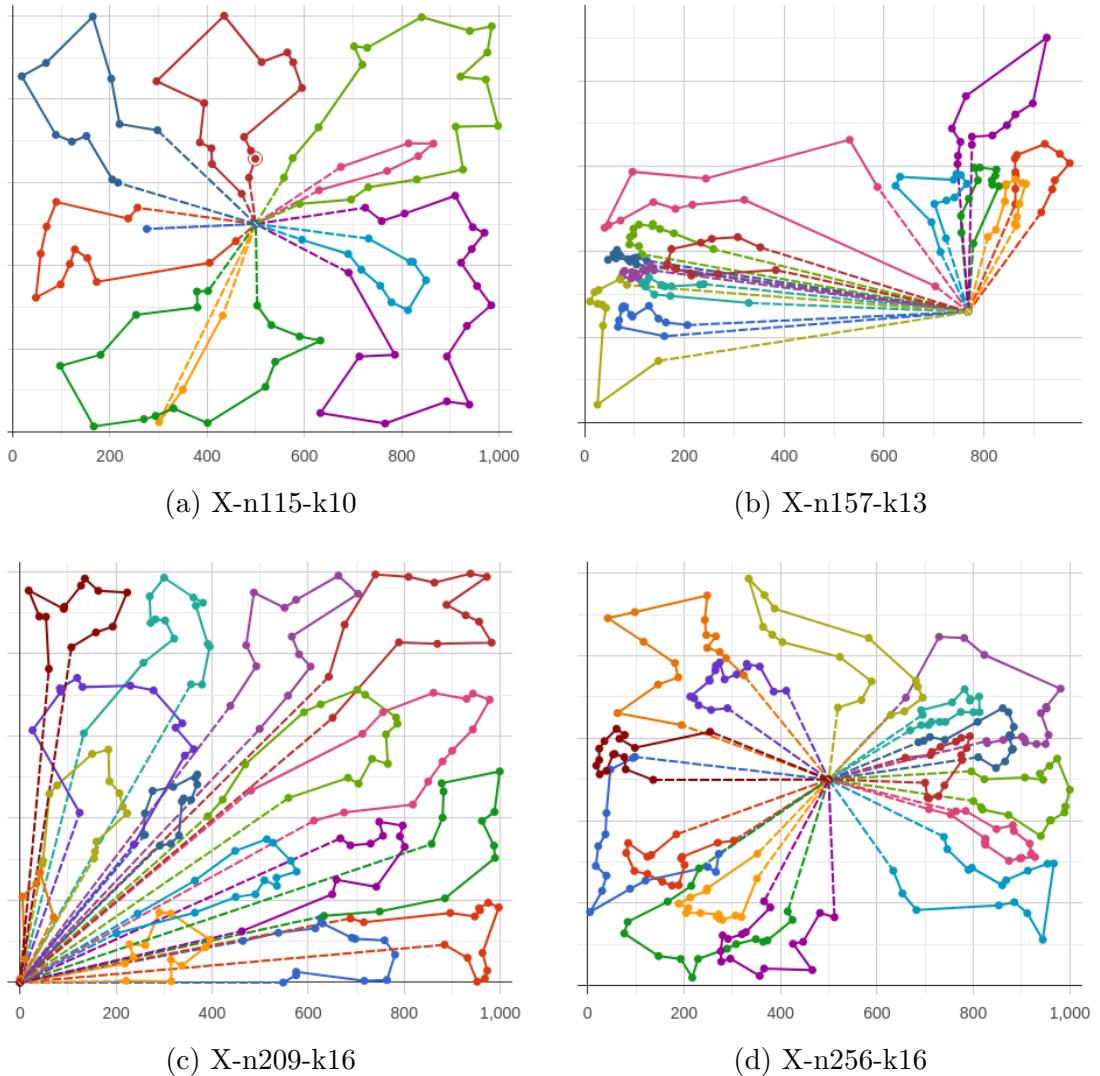


Figura 16: Rutas óptimas proporcionadas por la librería CVRPLIB.

*Nota:* Cada color indica una ruta independiente.

Como se puede observar comparando con estos resultados, el enfoque usado en este trabajo no dista mucho del óptimo proporcionado, teniendo en cuenta que la resolución de estos problemas haciendo uso de algoritmos para el CVRP permite zonas más flexibles, explicando así la mayoría de las diferencias encontradas. Se puede ver que los resultados son especialmente similares cuando el depósito se encuentra en el centro del grafo (caso de  $X\text{-}n115\text{-}k10$  y  $X\text{-}n256\text{-}k16$ ), ya que la división en zonas óptima se asemeja mucho más a la proporcionada por el enfoque usado.

Una observación interesante es que, incluso sin forzar el valor óptimo de rutas, el sistema logra ajustarse a la configuración mínima para el número de zonas en la mayoría de los casos, lo cual sugiere una buena integración entre la distribución

espacial de los nodos y la lógica de zonificación empleada. Además, el hecho de no observar grandes desviaciones entre zonas generadas por el enfoque planteado y zonas mínimas refuerza la idea de que este enfoque modular, basado en el modelo CMTSP, puede servir como alternativa eficaz al CVRP clásico sin incurrir en una pérdida imposible de asumir de eficiencia o factibilidad operativa.



# 6

## Conclusiones y trabajo futuro

Este trabajo ha desarrollado un sistema modular para la resolución del problema de rutas de recogida de residuos urbanos, estructurado en dos fases diferenciadas: una etapa inicial de zonificación del grafo en subproblemas compatibles con la capacidad de los camiones, y una posterior fase de resolución de rutas mediante enfoques tipo TSP para cada zona.

Los resultados obtenidos en el primer conjunto de experimentos han permitido evaluar la escalabilidad y estabilidad del proceso de zonificación ante grafos sintéticos de distinto tamaño y distribución espacial. Se ha observado que, aunque el tiempo de ejecución de la etapa de zonificación crece de manera casi exponencial, en línea con la complejidad teórica del algoritmo de barrido angular, el tiempo total del sistema (zonificación más resolución de rutas) mantiene un crecimiento tendencialmente lineal, lo que confirma la buena escalabilidad del enfoque modular empleado. Además, el número mínimo y promedio de nodos por zona permanece relativamente estable (entre 7 y 9 vértices en el caso específico de las pruebas realizadas), lo que indica una partición balanceada. También se ha constatado que el rendimiento del sistema es robusto ante distintas distribuciones de nodos (uniforme, normal y en clústeres), sin requerir ajustes específicos, lo cual refuerza la generalidad del método de zonificación.

El segundo experimento, centrado en analizar el impacto de la capacidad máxima de los vehículos sobre un mismo grafo, ha puesto de manifiesto que el número de zonas decrece siguiendo una curva con forma de campana descendente: a mayor capacidad, mayor número de nodos agrupados por zona, y por tanto menor fragmentación. Esto se traduce en una reducción inicial significativa del

---

tiempo de ejecución total, que luego se estabiliza e incluso muestra una leve tendencia creciente debido al aumento de complejidad en la resolución de rutas más extensas. En paralelo, el valor de la solución (distancia total recorrida) también desciende inicialmente y luego se estabiliza, confirmando que el aumento de capacidad aporta beneficios hasta cierto umbral, más allá del cual los rendimientos marginales disminuyen. Además, se ha evaluado la influencia de dos estrategias de partición (ascendente y descendente), y se ha comprobado que no existen diferencias significativas entre ellas, ni en términos de calidad de solución, ni en tiempos de ejecución ni en el número de zonas generadas. Por tanto, se concluye que ambos enfoques son válidos y que puede optarse por el método ascendente por su mayor simplicidad y reproducibilidad.

Complementando estos estudios, se ha validado la propuesta sobre instancias reales de la biblioteca CVRPLIB. El sistema ha mostrado una alta eficiencia computacional, resolviendo incluso las instancias más grandes (como X-n856-k95) en menos de 16 segundos, y generando un número de zonas muy cercano al mínimo teórico requerido por la capacidad de los vehículos. Este comportamiento confirma la validez del enfoque como solución viable para problemas reales, aun considerando que el sistema opera actualmente en modo secuencial, sin aprovechar el paralelismo. Esta circunstancia abre una clara línea de mejora que podría reducir aún más los tiempos de cómputo y permitir el uso del sistema en entornos urbanos de gran escala.

Sin embargo, es importante señalar que el modelo actual ha sido desarrollado bajo un conjunto de hipótesis simplificadoras que han permitido centrar el análisis en la eficiencia del enfoque propuesto. Las principales limitaciones establecidas son la capacidad homogénea de los vehículos, el uso de un único depósito, la consideración de los grafos utilizados en las pruebas como grafos conexos y la autonomía ilimitada de los vehículos utilizados.

Además de superar estas limitaciones, una mejora clave reside en perfeccionar el proceso de zonificación. Actualmente, las zonas generadas son disjuntas, lo cual simplifica el procesamiento, pero no siempre refleja adecuadamente la dinámica urbana real. En futuras líneas de trabajo se propone el desarrollo de métodos de zonificación más avanzados, capaces de generar zonas superpuestas, flexibles o adaptativas, utilizando datos reales como densidad de residuos, características geográficas o hábitos de generación por fracción horaria. Técnicas de clustering basadas en aprendizaje automático o modelos difusos permitirían una representación más precisa del espacio urbano y una mayor eficiencia global del sistema.

Por tanto se puede concluir que el enfoque modular que se ha implementado en este trabajo a partir de algoritmos ya existentes ha mostrado resultados robustos, tiempos de ejecución que resultan aceptables y una buena capacidad para su aplicación en grafos con un gran número de nodos.

En este trabajo se han establecido restricciones cuya eliminación marca un cla-

ro camino para futuras mejoras del proyecto. En primer lugar, el planteamiento actual considera ciertas simplificaciones necesarias para garantizar la viabilidad computacional y la claridad metodológica, como la asunción de zonas rígidas y no superpuestas, así como la utilización de datos estáticos para la generación y demanda de residuos. Estas simplificaciones, si bien permiten una buena aproximación inicial, pueden alejarse de la realidad dinámica del entorno en el que se podrían aplicar algoritmos como los propuestos en este trabajo, donde la generación de residuos varía temporalmente y las áreas de influencia pueden ser difusas o solaparse. Esto provoca que esta solución no se podría aplicar al mundo real, pero debido a la complejidad del problema, todas estas limitaciones se han aplicado con el fin de simplificar el trabajo a realizar.

Otra limitación del trabajo actual radica en la consideración estática de las capacidades y características de los vehículos, sin contemplar cambios o incidencias durante la operación, como averías, retrasos o variaciones en la carga, o otros factores como la autonomía con la que cuentan, ya que en este trabajo se establece que dicha autonomía es ilimitada.

Desde el punto de vista algorítmico, se podrían mejorar los resultados obtenidos en los experimentos con el uso de algoritmos más sofisticados, como pueden ser los algoritmos evolutivos, las técnicas de optimización multiobjetivo, el aprendizaje automático o métodos exactos.

Por lo que partiendo del enfoque planteado en este trabajo se podrían aplicar mejores algoritmos y levantar las restricciones planteadas para poder estudiar la viabilidad de implementar una propuesta de este tipo en un proceso real que cumpla con las condiciones necesarias del problema propuesto aquí.

Toda la implementación realizada se puede encontrar públicamente en el repositorio de *Github*<sup>1</sup> indicado en el pie de página.

---

<sup>1</sup><https://github.com/LovetheFrogs/TFG-Gestion-de-residuos-urbanos>

## Bibliografía

- [1] George B Dantzig y John H Ramser. “The truck dispatching problem”. En: *Management Science* 6.1 (1959), págs. 80-91. DOI: 10.1287/mnsc.6.1.80.
- [2] Gianpaolo Ghiani, Gilbert Laporte y Raffaele Musmanno. *Introduction to Logistics Systems Management*. John Wiley & Sons, 2013. DOI: 10.1002/9781118492185.
- [3] B.E. Gillett y L.R. Miller. “A Heuristic Algorithm for the Vehicle-Dispatch Problem”. En: *Operations Research* 22.2 (1974), págs. 340-349. DOI: 10.1287/opre.22.2.340.
- [4] David S. Johnson y Lyle A. McGeoch. “The Traveling Salesman Problem: A Case Study in Local Optimization”. En: *Local Search in Combinatorial Optimization*. Ed. por E. H. L. Aarts y J. K. Lenstra. Preliminary version dated November 20, 1995. London: John Wiley y Sons, 1997, págs. 215-310. DOI: 10.1515/9780691187563-011.
- [5] S. Kirkpatrick, C. D. Gelatt y M. P. Vecchi. “Optimization by simulated annealing”. En: *Science* 220.4598 (1983), págs. 671-680. DOI: 10.1126/science.220.4598.671.
- [6] Gilbert Laporte. “Fifty years of vehicle routing”. En: *Transportation Science* 43.4 (2009), págs. 408-416. URL: <https://www.jstor.org/stable/25769465>.
- [7] Gilbert Laporte. “The vehicle routing problem: An overview of exact and approximate algorithms”. En: *European Journal of Operational Research* 59.3 (1992), págs. 345-358. DOI: 10.1016/0377-2217(92)90192-C.
- [8] Jan Karel Lenstra y Alexander H. G. Rinnooy Kan. “Complexity of vehicle routing and scheduling problems”. En: *Networks* 11 (1981), págs. 221-227. URL: <https://api.semanticscholar.org/CorpusID:206312678>.
- [9] T. C. Martins, A. K. Sato y M. S. G. Tsuzuki. “Adaptive Neighborhood Heuristics for Simulated Annealing over Continuous Variables”. En: *Simulated Annealing – Advances, Applications and Hybridizations*. Open Access chapter. InTech, 2012, págs. 1-18. DOI: 10.5772/50302.

## BIBLIOGRAFÍA

---

- [10] Javier Pérez et al. “A methodology for estimating the carbon footprint of waste collection vehicles under different scenarios: Application to Madrid”. En: *Transportation Research Part D: Transport and Environment* 57 (2017), págs. 234-245. DOI: 10.1016/j.trd.2017.03.007.
- [11] Z. J. Peña et al. “Distance based sweep nearest algorithm to solve capacitated vehicle routing problem”. En: *International Journal of Advanced Computer Science and Applications* 10.10 (2019), págs. 259-264. DOI: 10.14569/IJACSA.2019.0101036.
- [12] Kiana Rouhi, Majid Shafiepour Motlagh y Fatemeh Dalir. “Developing a carbon footprint model and environmental impact analysis of municipal solid waste transportation: A case study of Tehran, Iran”. En: *Journal of the Air & Waste Management Association* 73.3 (2023), e1-e15. DOI: 10.1080/10962247.2023.2271424.
- [13] L. Z. Tarawneh, M. D. Serhan y S. W. Yoon. “Solving Capacitated Vehicle Routing Problem Using Two Phase Heuristic Method”. En: *Proceedings of the 8th Annual World Conference of the Society for Industrial and Systems Engineering (SISE 2019)*. Baltimore, MD, USA, 2019, págs. 171-177. ISBN: 978-1-938496-17-2.
- [14] Paolo Toth y Daniele Vigo. *Vehicle Routing: Problems, Methods, and Applications*. Society for Industrial y Applied Mathematics (SIAM), 2014. ISBN: 978-1611973594. DOI: 10.1137/1.9781611973594.
- [15] Nghia T. Vo, Mark B. H. Breese y Herbert O. Moser. “Feasibility of Simulated Annealing Tomography”. En: *arXiv preprint arXiv:1411.4622* (2014). arXiv: 1411.4622 [physics.med-ph].
- [16] World Bank. *Solid Waste Management*. <https://www.worldbank.org/en/topic/urbandevelopment/brief/solid-waste-management>. Accessed: November 12, 2023.
- [17] Cevat Yaman et al. “Investigation and modelling of greenhouse gas emissions resulting from waste collection and transport activities”. En: *Waste Management & Research* 37.12 (2019), págs. 1282-1290. DOI: 10.1177/0734242X19882482.
- [18] Jiapu Zhang. “Simulated annealing: in mathematical global optimization computation, hybrid with local or global search, and practical applications in crystallography and molecular modelling”. En: *arXiv preprint arXiv:1308.6220* (2013). Chapter 1 in “Simulated Annealing: Strategies, Potential Uses and Advantages”, NOVA Science Publishers, 2014, pp. 1–34. URL: <https://arxiv.org/abs/1308.6220>.



# Apéndices



# A

## Librería: Ejecución de tests y benchmarks

Para la ejecución de todos los procesos de este trabajo, se ha codificado una librería de python completa en la que se encuentran métodos para generar, dividir y resolver cualquier tipo de grafos, así como herramientas para generar grafos, librerías de testing y benchmarks varios usados para crear los experimentos. A continuación, se explica como ejecutar estos benchmarks, como usar la librería y los requerimientos necesarios.

La instalación de la librería puede hacerse en cualquier equipo. Tras su descarga, se pueden ejecutar todos los archivos de python **como módulos**. La librería permite la creación de objetos tipo Grafo, así como de sus componentes (Nodos y Aristas). Además, esta implementación está pensada para ser utilizada en conjunto a los algoritmos desarrollados. Entre estos se encuentra el algoritmo **Sweep** utilizado para dividir los grafos en zonas, la implementación de **Simulated Annealing** usada para resolver los problemas tipo TSP y otras metaheurísticas que no han sido utilizadas en este trabajo. También se incluyen distintos archivos de grafos de distintos tamaños, grafos de la librería TSPLIB y de CVRPLIB.

Se pueden ejecutar una serie de pruebas unitarias que comprueban el correcto funcionamiento de todo el código de los algoritmos y las estructuras de datos. Estos tests se pueden ejecutar localmente escribiendo el comando siguiente desde la raíz del proyecto:

```
python3 -m unittest Algorithm.Library.tests.tests
```

Además, se ha incluido el código necesario para ejecutar los experimentos. Para ejecutar los experimentos 1 y 2, se puede usar el comando desde la raíz del proyecto en un Sistema Operativo base Linux:

```
./Algorithm/Library/benchmark/run_benchmark.sh
```

---

O en otros sistemas operativos ejecutar los dos experimentos independientemente con

```
python3 -m Algorithm.Library.benchmark.benchmark_zones_tcap  
python3 -m Algorithm.Library.benchmark.benchmark_zones_gsize
```

Siendo el primero el correspondiente al Experimento 2 y el segundo el que corresponde al Experimento 1. Para ejecutar sobre los mapas de CVRPLIB, se puede ejecutar el comando

```
python3 -m unittest Algorithm.Library.cvrplib_instances
```

en la raíz del proyecto.

Por último, se ha incluido un código de muestra que crea un grafo, lo divide y resuelve cada una de las divisiones. Este puede encontrarse en `/Algorithm/Library/main.py` y sirve de muestra del funcionamiento del proceso planteado.

Las librerías necesarias se pueden encontrar en el archivo `/Algorithm/Library/requirements.txt`. Se recomienda usar la versión de python 3.10 o superior, ya que el código no ha sido probado en versiones inferiores.

El código de los tests y el benchmark se puede encontrar públicamente en el repositorio de *Github*<sup>1</sup> indicado en el pie de página.

---

<sup>1</sup><https://github.com/LovetheFrogs/TFG-Gestion-de-residuos-urbanos/tree/main/Algorithm/Library>

# B

## Librería: Guía de uso

A continuación se incluye una guía de uso de la librería desarrollada para este proyecto. Esta guía se divide en dos secciones, **Librería** (B.1), que es una guía de uso del código con los algoritmos y las estructuras de datos usadas en este proyecto y la sección **Utilidades** (B.2), con la guía de uso del *script* utilizado para generar los grafos sintéticos.

Cabe destacar que en esta guía se encuentran funciones que no han sido utilizadas en la versión final del proyecto, pero se han incluido y mantenido en el código para servir en caso de realizar algún desarrollo a partir de esta librería.

### B.1. Librería

La librería principal contiene las definiciones de las estructuras de datos fundamentales (`Node`, `Edge`, `Graph`) y los algoritmos de optimización.

#### B.1.1. Excepciones Personalizadas

El módulo `exceptions.py` define excepciones personalizadas utilizadas en toda la biblioteca:

##### **NodeNotFound**

**Descripción:** Se lanza cuando se busca un nodo que no existe en la estructura.

**Parámetros:**

- **id** (int): Identificador del nodo no encontrado.
- **message** (str, opcional): Descripción del error. Por defecto: “The node searched for was not found in the structure. Index searched.”.

## DuplicateNode

**Descripción:** Se lanza cuando se intenta añadir un nodo que ya existe en el grafo.

**Parámetros:**

- **message** (str, opcional): Descripción del error. Por defecto: “The node is already in the Graph”.

## DuplicateEdge

**Descripción:** Se lanza cuando se intenta añadir una arista que ya existe en el grafo.

**Parámetros:**

- **message** (str, opcional): Descripción del error. Por defecto: “The edge is already in the Graph”.

## EdgeNotFound

**Descripción:** Se lanza cuando una arista no se encuentra en el grafo.

**Parámetros:**

- **path** (str): Nodos origen y destino de la arista no encontrada.
- **message** (str, opcional): Descripción del error. Por defecto: “The edge was not found in the structure.”

## NoCenterDefined

**Descripción:** Se lanza cuando el grafo no tiene definido un nodo central.

**Parámetros:**

- **message** (str, opcional): Descripción del error. Por defecto: “A node has not been set to be the center of the graph.”

## EmptyGraph

**Descripción:** Se lanza cuando el grafo no contiene nodos ni aristas.

**Parámetros:**

- **message** (str, opcional): Descripción del error. Por defecto: “The graph does not have any edges or nodes in it.”

## WrongEdgeType

**Descripción:** Se lanza cuando una instancia TSPLib no es de tipo EUC\_2D.

**Parámetros:**

- **message** (str, opcional): Descripción del error. Por defecto: “The edge type is not EUC\_2D.”

### B.1.2. Estructuras de Datos

#### Clase Node

**Descripción:** Implementa un objeto Nodo personalizado para facilitar el acceso a los datos almacenados, como su peso, coordenadas e índice.

**Parámetros del constructor:**

- **index** (int): Índice del nodo. El usuario debe asegurar que sea único.
- **weight** (float): Peso del nodo.
- **x** (float): Coordenada x del nodo.
- **y** (float): Coordenada y del nodo.
- **center** (bool, opcional): Indica si el nodo es el nodo central. Por defecto: False.

**Atributos:**

- **index** (int): Índice del nodo.
- **weight** (float): Peso del nodo. 0 si es el nodo central.
- **center** (bool): Indica si es el nodo central.
- **visited** (bool): Estado de visita del nodo.
- **coordinates** (tuple): Tupla con las coordenadas (x, y) del nodo.
- **angle** (float): Ángulo formado con el nodo central.

**Métodos principales:**

- **get\_distance(b: Node) ->float**: Calcula la distancia Manhattan entre dos nodos en una esfera 3D utilizando la fórmula de Haversine. Este método proporciona una mejor estimación de la distancia real que el cálculo en un plano 2D.
- **get\_distance\_twod(b: Node) ->float**: Calcula la distancia euclíadiana 2D entre dos nodos.
- **change\_status()**: Cambia el estado de visita del nodo.

- `set_weight(weight: float)`: Establece el peso del nodo a un nuevo valor.

#### Ejemplo de uso:

```
# Crear un nodo normal
nodo1 = Node(index=1, weight=150.0, x=40.4168, y=-3.7038)

# Crear un nodo central
centro = Node(index=0, weight=0, x=40.4165, y=-3.7026, center=True)

# Calcular distancia en un plano bidimensional
distancia = nodo1.get_distance_twod(centro)
```

## Clase Edge

**Descripción:** Implementa un objeto Arista personalizado que facilita el acceso a datos como la longitud, velocidad, y nodos origen y destino.

#### Parámetros del constructor:

- `speed (float)`: Velocidad promedio para atravesar la arista.
- `origin (Node)`: Nodo de origen de la arista.
- `dest (Node)`: Nodo de destino de la arista.
- `bidimensional (bool, opcional)`: Indica si usar cálculo 2D. Por defecto: False.

#### Atributos:

- `length (float)`: Longitud de la arista desde origen a destino.
- `speed (float)`: Velocidad promedio para atravesar la arista.
- `origin (Node)`: Nodo de origen.
- `dest (Node)`: Nodo de destino.
- `time (float)`: Tiempo que toma atravesar la arista.
- `value (float)`: Costo de la arista más 0.033 para recogida del nodo (2 minutos).

#### Ejemplo de uso:

```
origen = Node(1, 100, 40.4168, -3.7038)
destino = Node(2, 120, 40.4200, -3.7050)
arista = Edge(speed=50.0, origen, destino)
```

## Clase Graph

**Descripción:** Contiene la definición de la estructura de grafo y las funciones utilizadas para su construcción, actualización y procesamiento.

#### Atributos principales:

- **graph** (dict): Representación interna del grafo.
- **node\_list** (list[Node]): Lista de todos los nodos del grafo.
- **edge\_list** (list[Edge]): Lista de todas las aristas del grafo.
- **nodes** (int): Número de nodos en el grafo.
- **edges** (int): Número de aristas en el grafo.
- **center** (Node): Nodo central del grafo.
- **distancess** (list[list[float]]): Matriz de valores del grafo.

**Métodos de construcción:**

- **add\_node(node: Node)**: Añade un nodo al grafo.
- **add\_edge(edge: Edge)**: Añade una arista al grafo.
- **set\_center(node: Node)**: Establece el nodo central del grafo.
- **get\_node(idx: int) ->Node**: Obtiene un nodo del grafo por su índice.
- **get\_edge(origin: Node, dest: Node) ->Edge**: Obtiene una arista del grafo.
- **set\_distance\_matrix()**: Crea una matriz de distancias a partir del grafo.

**Métodos de carga de datos:**

- **populate\_from\_file(file: str, verbose: bool = False)**: Popula el grafo desde un archivo de texto con formato específico. El nodo con índice 0 se establece como centro.

**Formato del archivo:**

```
n
idx_1 weight_1 x_1 y_1
idx_2 weight_2 x_2 y_2
...
idx_n weight_n x_n y_n
m
speed_1 origin_1 dest_1
speed_2 origin_2 dest_2
...
speed_m origin_m dest_m
```

- **populate\_from\_tsplib(file: str)**: Popula el grafo desde una instancia TSPLib. El tipo de arista debe ser EUC\_2D.
- **populate\_from\_cvrplib(file: str)**: Popula el grafo desde una instancia CVRPLib. El tipo de arista debe ser EUC\_2D.
- **populate\_from\_database(cur, verbose: bool = False)**: Popula el grafo desde una base de datos utilizando un cursor psycopg2.

**Métodos de información:**

- **total\_weight() ->float**: Calcula el peso total de todos los nodos del grafo.

- `can_pickup_all(truck_capacity: float, truck_count: int) ->bool`: Determina si todos los camiones pueden recoger los contenedores en una ronda.
- `get_min_num_zones(truck_capacity: float) ->int`: Calcula el número mínimo de zonas necesarias maximizando la capacidad de cada camión.

#### **Algoritmos clásicos de grafos:**

- `bfs(source: Node) ->list[int]`: Realiza una búsqueda en anchura (BFS) desde el nodo fuente.
- `dijkstra(start: int) ->list[float]`: Implementa el algoritmo de Dijkstra para encontrar el camino más corto desde un nodo inicial a todos los demás nodos.
- `prim(start: int | Node) ->tuple[float, list[tuple[int, int]]]`: Calcula el árbol de expansión mínima (MST) usando el algoritmo de Prim. Devuelve el valor del MST y una lista de las aristas.
- `precompute_shortest_paths()`: Precalcula el camino más corto entre todos los pares de nodos usando Dijkstra repetidamente.

#### **División en zonas:**

- `divide_graph_ascendent(truck_capacity: float) ->list[list[Node]]`: Divide el grafo en zonas de forma ascendente según los ángulos de los nodos. Minimiza el número de zonas asegurando que cada zona pueda ser recogida por un camión.
- `divide_graph_decreasing(truck_capacity: float) ->list[list[Node]]`: Divide el grafo en zonas de forma descendente según los ángulos de los nodos.
- `create_subgraph(nodes: list[Node]) ->Graph`: Crea un subgrafo a partir de una lista de nodos del grafo actual.

#### **Métodos auxiliares:**

- `create_points(path: list[int]) ->list[tuple[float, float]]`: Genera una lista de coordenadas a partir de una lista de índices de nodos.
- `save(path: str)`: Guarda el grafo en un archivo usando pickle.
- `load(path: str) ->Graph | None`: Función estática que carga un grafo desde un archivo.
- `wipe()`: Elimina todos los datos del objeto para reutilizarlo.

#### **Ejemplo de uso completo:**

```
from Algorithm.Library.problem.model import Graph, Node, Edge

# Crear un grafo vacío
grafo = Graph()

# Añadir nodos
centro = Node(0, 0, 40.4165, -3.7026, center=True)
```

```
nodo1 = Node(1, 150.0, 40.4168, -3.7038)
nodo2 = Node(2, 200.0, 40.4170, -3.7040)

grafo.add_node(centro)
grafo.add_node(nodo1)
grafo.add_node(nodo2)

# Añadir aristas
arista1 = Edge(50.0, centro, nodo1)
arista2 = Edge(45.0, centro, nodo2)
arista3 = Edge(55.0, nodo1, nodo2)

grafo.add_edge(arista1)
grafo.add_edge(arista2)
grafo.add_edge(arista3)

# Establecer centro y matriz de distancias
grafo.set_center(centro)
grafo.set_distance_matrix()

# O cargar desde archivo
grafo2 = Graph()
grafo2.populate_from_file("data/dataset1.txt", verbose=True)

# Dividir en zonas
zonas = grafo2.divide_graph_ascending(truck_capacity=500.0)
```

### B.1.3. Algoritmos de Optimización

#### Clase Algorithms

**Descripción:** Contiene los diferentes algoritmos de búsqueda de caminos para resolver el Problema del Viajante (TSP).

##### Parámetros del constructor:

- graph (Graph): El objeto grafo sobre el cual se calcularán los caminos.

##### Métodos de división:

- divide(zone\_weight: float, dir: str | None = None, name: str = , asc: bool = False) ->tuple[list[Graph], list[list[int]]]: Divide el grafo en zonas de un peso dado y crea los subgrafos correspondientes.

##### Parámetros:

- zone\_weight: Peso máximo de las zonas.
- dir: Directorio donde guardar las gráficas. Si es None, se muestran en pantalla.
- name: Nombre para añadir a las gráficas.
- asc: Usar división ascendente si es True, descendente si es False.

**Devuelve:** Tupla con lista de subgrafos y lista de zonas (listas de índices de nodos).

#### Evaluación:

- `evaluate(individual: list[int]) ->float`: Calcula el valor de un camino sumando los valores de las aristas que lo componen.

#### Cotas inferiores:

- `one_tree(start: int | Node = 0) ->tuple[list[tuple[int, int]], float]`: Calcula el 1-árbol del grafo, que puede usarse como cota inferior para el valor de un tour TSP.

#### Algoritmos constructivos:

- `nearest_neighbor(start: int | Node = 0, dir: str | None = None, name: str = ) ->tuple[list[int], float]`: Ejecuta el algoritmo de *Nearest Neighbor*. Construye un tour visitando siempre el nodo no visitado más cercano.

#### Metaheurísticas:

- `run_ga_tsp(path: list[int] | None = None, ngen: int = 3000, cxpb: float = 0.7, mutpb: float = 0.2, pop_size: int = 1000, dir: str | None = None, name: str = , vrb: bool = False) ->tuple[list[int], float]`: Ejecuta un Algoritmo Genético para resolver el TSP.

#### Parámetros:

- `path`: Camino inicial. Si es `None`, comienza con uno aleatorio.
  - `ngen`: Número de generaciones.
  - `cxbp`: Probabilidad de cruce.
  - `mutpb`: Probabilidad de mutación.
  - `pop_size`: Tamaño de la población.
  - `dir`: Directorio para guardar gráficas.
  - `name`: Nombre para las gráficas.
  - `vrb`: Modo verbose.
- `run_two_opt(path: list[int] | None = None, dir: str | None = None, name: str = , vrb: bool = False) ->tuple[list[int], float]`: Ejecuta optimización *2-opt*. Toma dos aristas de un camino, las elimina y añade dos nuevas verificando si mejora el valor.

#### Parámetros:

- `path`: Camino inicial. Si es `None`, comienza con uno aleatorio.
- `dir`: Directorio para guardar gráficas.
- `name`: Nombre para las gráficas.
- `vrb`: Modo verbose.

- `run_sa(path: list[int] | None = None, dir: str | None = None, name: str = , vrb: bool = False) ->tuple[list[int], float]`: Ejecuta *Simulated Annealing*. Acepta soluciones peores con cierta probabilidad para escapar de óptimos locales.

**Parámetros:**

- `path`: Camino inicial. Si es `None`, comienza con uno aleatorio.
  - `dir`: Directorio para guardar gráficas.
  - `name`: Nombre para las gráficas.
  - `vrb`: Modo verbose.
- `run_tabu_search(path: list[int] = None, niter: int = 10000, mstag: int = 250, dir: str | None = None, name: str = , vrb: bool = False) ->tuple[list[int], float]`: Ejecuta Búsqueda Tabú. Explora vecindarios usando una lista tabú que almacena vecinos ya explorados.

**Parámetros:**

- `path`: Camino inicial. Si es `None`, comienza con uno aleatorio.
  - `niter`: Número de iteraciones a completar.
  - `mstag`: Máximo de iteraciones sin encontrar una ruta mejor.
  - `dir`: Directorio para guardar gráficas.
  - `name`: Nombre para las gráficas.
  - `vrb`: Modo verbose.
- `run(dir: str | None = None, name: str = , vrb: bool = False) ->tuple[list[int], float]`: Ejecuta el mejor algoritmo encontrado según los benchmarks (actualmente *Simulated Annealing*) desde una ruta inicial aleatoria.

**Parámetros:**

- `dir`: Directorio para guardar gráficas.
- `name`: Nombre para las gráficas.
- `vrb`: Modo verbose.

**Visualización:**

- `plot_multiple_paths(paths: list[list[tuple[float, float]]], dir: str | None = None, name: str | None = None) ->plt`: Prepara y grafica múltiples caminos en un mismo plot.

**Ejemplo de uso:**

```
from Algorithm.Library.problem.model import Graph
from Algorithm.Library.problem.algorithms import Algorithms

# Cargar grafo
grafo = Graph()
grafo.populate_from_file("data/dataset1.txt")

# Crear objeto de algoritmos
alg = Algorithms(grafo)
```

```
# Dividir en zonas
subgrafos, zonas = alg.divide(zone_weight=500.0,
                               dir="resultados",
                               name="division1")

# Ejecutar algoritmo en cada zona
for i, subgrafo in enumerate(subgrafos):
    alg_sub = Algorithms(subgrafo)

    # Vecino más cercano
    camino_nn, valor_nn = alg_sub.nearest_neighbor()

    # Simulated Annealing (mejor algoritmo)
    camino_sa, valor_sa = alg_sub.run_sa(path=camino_nn[:-1],
                                           vrb=True,
                                           dir="resultados",
                                           name=f"zona_{i}")

    print(f"Zona {i}: Valor = {valor_sa}")

    # O usar el método run() que ejecuta el mejor algoritmo
    camino_final, valor_final = alg_sub.run(dir="resultados",
                                              name="final",
                                              vrb=True)
```

## B.2. Utilidades

### B.2.1. Generación de Datasets Sintéticos

El módulo `create_models.py` permite generar datasets sintéticos para pruebas y benchmarking.

**Uso desde línea de comandos:**

```
python create_models.py [opciones]
```

**Opciones disponibles:**

- `-f, --files`: Número de archivos a generar (default: 1)
- `-n, --min_nodes`: Número mínimo de nodos (default: 200)
- `-N, --max_nodes`: Número máximo de nodos (default: 200)
- `-x, --min_x`: Coordenada x mínima (default: -100)
- `-X, --max_x`: Coordenada x máxima (default: 100)
- `-y, --min_y`: Coordenada y mínima (default: -100)
- `-Y, --max_y`: Coordenada y máxima (default: 100)

- **-w, --min\_weight:** Peso mínimo de nodos (default: 100.0)
- **-W, --max\_weight:** Peso máximo de nodos (default: 250.0)
- **-s, --min\_speed:** Velocidad mínima en aristas (default: 20.0)
- **-S, --max\_speed:** Velocidad máxima en aristas (default: 60.0)
- **-v, --verbose:** Modo verbose
- **-d, --distribution:** Distribución de coordenadas:
  - u: Uniforme
  - n: Normal
  - k: Por clusters

**Ejemplos:**

```
# Generar 5 datasets con distribución uniforme
python create_models.py -f 5 -n 100 -N 200 -d u -v

# Generar dataset con distribución normal
python create_models.py -f 1 -n 150 -N 200 -d n -v

# Generar dataset con clusters
python create_models.py -f 3 -n 200 -N 300 -d k -v
```

**Salida:** Los datasets se generan en `Algorithm/Library/utils/datasets/` con el formato `.txt`. Se crea también un archivo `log.txt` con estadísticas de los datasets generados.

## B.2.2. Funciones Auxiliares

La biblioteca incluye funciones auxiliares en el módulo `utils` para visualización y logging:

- `printProgressBar()`: Muestra una barra de progreso en la consola durante operaciones largas.

## B.2.3. Notas Importantes

1. **Densidad del grafo:** La biblioteca está diseñada para trabajar con grafos completos (`densidad = 1`), donde cada nodo está conectado con todos los demás.
2. **Nodo central:** El nodo con índice 0 se establece automáticamente como centro al cargar desde archivo. En otros métodos, debe establecerse explícitamente.
3. **Distancias:** Por defecto se usa la distancia Manhattan en esfera 3D (Haversine) en el cálculo de la distancia entre dos nodos. Para problemas 2D estándar, usar `bidimensional=True` al crear aristas.
4. **Capacidad de camiones:** Los algoritmos asumen que todos los camiones tienen la misma capacidad.

5. **Optimización:** El método `run()` ejecuta automáticamente el mejor algoritmo encontrado en los benchmarks (*Simulated Annealing*). Para control fino, usar los métodos individuales.

Para más información sobre cualquier función, todos los archivos de código explicados aquí se encuentran comentados para su mayor entendimiento.

El código de toda la librería se puede encontrar públicamente en el repositorio de *Github*<sup>1</sup> indicado en el pie de página.

---

<sup>1</sup><https://github.com/LovetheFrogs/TFG-Gestion-de-residuos-urbanos/tree/main/Algorithm/Library>