Universidad Rey Juan Carlos

Escuela Técnica Superior de Ingeniería Informática

Toma inteligente de decisiones en la recogida de residuos urbanos. Un enfoque basado en datos.

Trabajo Fin de Grado

Autor:
Marcos Ferrer Zalve

Mayo 2025

A mis abuelos, por acompañarme en este camino,
a mis padres, por su apoyo incondicional siempre.

Índice

1. Introducción **5**

2. Problema propuesto **5**
2.1. Limitaciones 6

3. Qué es un grafo **7**

4. División de un grafo **7**
4.1. Creando la primera división 7
4.2. Postproceso de las zonas 9

5. La función de evaluación **12**

6. Resolviendo el TSP **14**
6.1. Cómputo del límite inferior 14
6.1.1. Algoritmo 1-tree 14
6.1.2. Algoritmo Held-Karp 15
6.2. Creación de rutas iniciales 16
6.2.1. Algoritmo Nearest Neighbor 17
6.3. Optimización de recorridos 19
6.3.1. Algoritmo 2-opt 19
6.3.2. Algoritmo de Simulated Annealing 21
6.3.3. Algoritmo de Búsqueda Tabú 23

7. Elección de hiperparámetros **26**
7.1. Elección de los hiperparámetros de Simulated Annealing 26
7.2. Elección de los hiperparámetros de Búsqueda Tabú 28

8. Análisis de rendimiento **29**
8.1. Grafos pequeños 29
8.2. Grafos grandes 29
8.3. Grafos importados de *TSPLIB* 30
8.4. Conclusiones tras el análisis 32

9. Resolución del problema planteado **33**

10. Conclusiones finales y pasos futuros **35**

Referencias **36**

Anexos **37**

Resumen

Este trabajo aborda el problema de optimización de rutas de recogida de residuos para una flota de camiones, garantizando que cada vehículo no exceda su capacidad y complete un recorrido eficiente. Para modelar el problema, se utiliza un grafo completo, no dirigido y ponderado, donde los vértices representan contenedores (incluido un depósito común) y las aristas indican el coste de desplazamiento entre ellos. El objetivo es dividir este grafo en subgrafos (zonas) de forma que cada uno cumpla con las restricciones de capacidad y, sobre cada subgrafo, resolver una instancia del Problema del Viajante (TSP).

La división en zonas se realiza a través de un algoritmo basado en el ángulo respecto al depósito central, seguido de un postprocesado para equilibrar pesos y eliminar zonas pequeñas. A cada zona se le aplica un algoritmo de evaluación que estima el coste total del recorrido en función de una versión modificada de la distancia Manhattan sobre coordenadas geográficas, considerando además tiempos de recogida por contenedor.

Posteriormente, se comparan distintos enfoques de resolución del TSP, métodos de estimación de cota inferior como 1-tree y Held-Karp, algoritmos constructivos como Nearest Neighbor, y metaheurísticas de optimización como 2-opt, Simulated Annealing y Búsqueda Tabú. Se realiza un análisis exhaustivo del rendimiento y calidad de las soluciones generadas, tanto en grafos sintéticos como en instancias reales de la biblioteca TSPLIB.

Los resultados permiten definir un flujo óptimo de resolución basado en la combinación de algoritmos, con especial énfasis en la inicialización con Nearest Neighbor y la posterior mejora mediante Búsqueda Tabú o Simulated Annealing. También se derivan fórmulas empíricas para ajustar hiperparámetros de forma adaptativa al tamaño del grafo. Este estudio pretende ofrecer un análisis de calidad en el ámbito de la optimiazción de la logística urbana.

Conceptos clave

- **Grafo**: Estructura matemática compuesta por nodos (contenedores) y aristas, utilizada para modelar relaciones entre ubicaciones geográficas.
- **TSP (Travelling Salesman Problem)**: Problema clásico de optimización que consiste en encontrar el recorrido más corto que visita todos los nodos exactamente una vez y regresa al punto de partida.
- **1-tree**: Algoritmo que proporciona una cota inferior para el TSP mediante la construcción de un árbol mínimo con una conexión adicional.
- **Held-Karp**: Algoritmo de relajación lagrangiana que mejora la estimación de la cota inferior del TSP usando penalizaciones iterativas.
- **Nearest Neighbor**: Algoritmo codicioso que construye una solución inicial eligiendo siempre el nodo más cercano no visitado.
- **2-opt**: Heurística de mejora local que optimiza un recorrido intercambiando pares de aristas para reducir la distancia total.
- **Simulated Annealing**: Metaheurística inspirada en el recocido físico que permite aceptar soluciones peores para escapar de óptimos locales.
- **Búsqueda Tabú**: Algoritmo de búsqueda local mejorada que evita ciclos y estancamientos mediante una memoria de soluciones recientes prohibidas.

- **Función de evaluación**: Fórmula utilizada para calcular el coste temporal de un recorrido, basada en distancia y tiempo de servicio.
- **Zonificación radial**: Método de división del grafo en subgrafos basado en el ángulo respecto al depósito central para distribuir carga entre camiones.

1. Introducción

Las secciones siguientes exploran cuál es el problema propuesto y la solución aportada. Más allá de ello, este trabajo pretende ser un análisis de distintos algoritmos que son comúnmente usados para resolver este tipo de problemas, tratando de justificar la elección final de cuál es el mejor para nuestro propósito, así como compararlos, hablando de las ventajas y desventajas que presenta usar uno frente al otro.

Al final del documento se encuentran varios anexos, detallando cómo se ha modelado el problema, las pruebas realizadas para validar su correcto funcionamiento y las dependencias de este proyecto, entre otros. Estos anexos tienen como objeto facilitar el uso del código implementado por terceros, así como mostrar la base sobre la que se cimentan los algoritmos evaluados.

2. Problema propuesto

El problema presentado es el siguiente: **¿Cómo se pueden organizar las rutas de recogida de residuos de una serie de camiones de modo que cada camión pueda completar su recorrido sin superar su capacidad y este camino sea lo más rápido posible?**

Lo primero, definiremos el espacio del problema como un *grafo* G compuesto de una serie de *vértices* V (que en nuestro xaso serán los **contenedores**) y aristas E tal que

$$G = (V, E) \tag{1}$$

Se presentan dos objetivos. Primero, debemos encontrar una serie de *subgrafos* $G_s = (V_s, E_s)$ tal que $G_i \subset G$ y $G_1 \cup G_2 \cup G_3 \cup \cdots \cup G_n = G \ \forall \ i \in \{1, n\}$

Además, cada subgrafo G_s debe cumplir que la suma del peso de todos sus contenedores sea menor que la capacidad de un camión. Expresado formalmente, dado $G_s = (V_s, E_s)$, un subgrafo G_s de G debe satisfacer

$$\sum_{i=0}^{n} peso(N_i) \leq T \tag{2}$$

Donde i es el índice de un nodo, $peso()$ es una función que devuelve el peso de este y T es la capacidad de un camión.

Adicionalmente, para cada subgrafo G_s, debemos encontrar un camino óptimo (o cercano al óptimo) que incluya todos los contenedores de G_s y sea lo más corto posible. Es decir, para cualquier ruta R_s de G_s, se deben cumplir las siguientes restricciones:

$$\begin{aligned} &\forall \ V_s \in R_s \rightarrow V_s \in G_s \\ &\forall \ V_s \in G_s \rightarrow V_s \in R_s \end{aligned} \tag{3}$$

Todos estos caminos R_s cumpliran ser un ciclo hamiltoniano de G_s.

Una vez dividido el grafo y obtenidos los subgrafos, podemos evaluar el problema de encontrar R_s para cada G_s como una instancia del *Travelling Salesman Problem*, sobre el que podremos aplicar uno o varios algoritmos, evaluando sus soluciones usando una **función de evaluación** $f(R)$ que nos dé el valor (o puntuación) dicha solución. El valor de $f(R)$ será pues usado para encontrar un camino que satisfaga nuestras restricciones.

Uno de los objetivos de este documento es explorar varios algoritmos capaces de dar una solución aproximada al *TSP* de modo que encontremos la que nos aporte un recorrido mejor, teniendo en cuenta tanto el valor de la funcion de evaluación $f(R)$ como el tiempo que se ha tardado en ser obtenida.

2.1. Limitaciones

Antes de hablar de los algoritmos desarrollados, se deben tener en cuenta una serie de limitaciones, seleccionadas para simplificar el problema. Estas limitaciones son:

1. La capacidad t_c de cada camión es la misma para todos los camiones, es decir

$$t_c = T;\ \forall\ c \in C \tag{4}$$

   donde C es el conjunto de camiones y t_c es la capacidad del camión $c \in C$

2. Se considera que en cada grafo G hay solo un vértice $V_0 \in G$, (denominado **depósito**) del que salen todos los camiones.

3. Todo grafo G y, por consiguiente; todo subgrafo G_s tiene una densidad de 1 provocando que todos los grafos y subgrafos sean completos. Esto significa que desde cualquier vértice de un grafo, ya sea este un depósito o un contenedor, se puede llegar a cualquier otro. Es decir,

$$\forall\ (V_i, V_j) \in G \rightarrow \exists\ E \in G \tag{5}$$

   debido a esto, la fórmula de la densidad $d(G)$ es siempre 1. Esto significa que

$$d(G) = \frac{e}{v \cdot (v-1)} = 1 \tag{6}$$

   siendo e el número de aristas de G y v el número de vértices de G.

4. Todos los camiones tienen una autonomía suficiente para recorrer un subgrafo. A efectos prácticos, podemos decir que

$$a_c = A = \infty;\ \forall\ c \in C \tag{7}$$

   siendo a_c la autonomía del camión c.

Habiendo expuesto el problema presente y las limitaciones establecidas, las siguientes secciones explican los algoritmos utilizados para la división del grafo G y la resolución del TSP para cada G_s.

3. Qué es un grafo

Como se indíca en la sección anterior, el espacio del problema se define como un *grafo*, el cual se denominará G. Un grafo es una **estructura de datos abstracta** utilizada para modelar relaciones entre entidades. Está compuesto por un conjunto de elementos denominados *nodos* (que en el ámbito de este documento son los contenedores y el depósito), y un conjunto de conexiones entre ellos llamadas *aristas* o *enlaces*. Cada nodo representa una entidad individual que en este estudio es un punto geográfico a visitar, y cada arista indica una posible conexión o camino entre dos contenedores.

En este caso, el grafo es **no dirigido**, **ponderado** y **completo**.

Se trata de un grafo no dirigido porque el coste de ir del contenedor A al contenedor B es el mismo en ambos sentidos. Además, es ponderado debido a que cada arista tiene asociado un valor que representa su puntuación. Finalmente, se considera completo para garantizar que cualquier contenedor puede ser alcanzado desde cualquier otro.

4. División de un grafo

A continuación, se expone cómo se puede dividir un grafo G en n subgrafos G_s en los que se cumpla la ecuación del *peso total de una zona* (ecuación 2). De ahora en adelante, nos referiremos a un subgrafo G_s como una *zona*

Para la solución desarrollada, se ha optado por una división en zonas radial. Esto quiere decir que la selección se ha hecho en función del ángulo α que cada nodo V de una zona forma con el depósito V_0.

El algoritmo que divide el grafo en estas zonas realiza dos iteraciones. En la primera, se realizan una serie de divisiones, creando n zonas G_s. Al finalizar esta primera iteración, se vuelven a procesar las n zonas, intentando que todas tengan un peso similar, y tratando de eliminar cualquier zona G_s en la que haya menos de 3 contenedores. Formalmente, tratando de eliminar cualquier subgrafo G_s que cumpla

$$v_s \leq 2 \tag{8}$$

donde v es el número de contenedores de la zona s

4.1. Creando la primera división

Esta sección corresponde a la primera parte del algoritmo de división en zonas. Se pretende explicar este haciendo uso del pseudocódigo mostrado a continuación.

Algorithm 1 División en zonas

function DIVIDEGRAPH(T)
  $C \leftarrow V_0$
  for $V \in G$ **do**
    if $V \neq V_0$ **then**
      $A \leftarrow V$
      $\Delta x = A_x - C_x$
      $\Delta y = A_y - C_y$
      $A_{angle} = atan2(y, x)$
    end if
  end for
  $orderedNodes = sortOnAngle(V_i \ \forall \ V_i \in G)$
  $zones = createZones(orderedNodes, T)$
  $zones = postprocess(zones, T)$
  return *zones*
end function

Este código se encarga de comparar el ángulo que forma el depósito V_0 con cada $V_s \in G$. Para ello, se obtiene la diferencia entre las coordenadas del contenedor V_s y las del depósito V_0 y se calcula el arcotangente de Δx y Δy (la diferencia de ambas coordenadas entre V_0 y V_s) usando la función $atan2$, perteneciente al paquete `math` de python, cuto resultado depende de los valores de x e y, respondiendo a:

$$atan2(\Delta y, \Delta x) = \begin{cases} \arctan \frac{\Delta y}{\Delta x} & \text{if} \quad \Delta x > 0, \\ \arctan \frac{\Delta y}{\Delta x} + \pi & \text{if} \quad \Delta x > 0 \ and \ \Delta y \leq 0, \\ \arctan \frac{\Delta y}{\Delta x} - \pi & \text{if} \quad \Delta x < 0 \ and \ \Delta y < 0, \\ +\frac{\pi}{2} & \text{if} \quad \Delta x = 0 \ and \ \Delta y > 0, \\ -\frac{\pi}{2} & \text{if} \quad \Delta x = 0 \ and \ \Delta y < 0, \\ undefined & \text{if} \quad \Delta x = 0 \ and \ \Delta y = 0. \end{cases} \tag{9}$$

Una vez determinado $V_{angle} \forall \ V \in G_s$, se crea una lista de todos los contenedores, ordenados según este, haciendo uso de la función `sortOnAngle`, que ordena todos los nodos de un grafo G en función del ángulo establecido para todos sus contenedores, haciendo que se cumpla:

$$V_i \in orderedNodes \ \forall \ V_i \in G \tag{10}$$

Este resultado lo guardaremos en la lista *orderedNodes*. Cuando se ha creado y poblado, se llama a la función `createZones`, usándo dicha lista como argumento junto a T, la capacidad con la que cuenta cada camión.

En el siguiente pseudocódigo se expone dicha función:

Algorithm 2 Asignación a una zona

```
function createZones(orderedNodes, T)
    zones ← [ ]
    currentZone ← [ ]
    currentWeight ← 0
    for V ∈ orderedNodes do
        if currentWeight + V_weight > T then
            zones.append([V_0] + currentZone)
            currentZone = V
            currentWeight = V_weight
        else
            currentZone.append(V)
            currentWeight = currentWeight + V_weight
        end if
    end for
    if currentZone ≠ [ ] then
        zones.append([V_0] + currentZone)
    end if
    return zones
end function
```

Siguiéndolo, se ve que crea una zona vacía y añade los contenedores V_i que se obtienen de la lista $orderedNodes$ hasta que incluir un contenedor más hace que esa zona supere la capacidad de un camión. En ese momento, se añade V_0 a la zona actual y se actualiza la lista de zonas definitivas $zones$.

Este algoritmo sigue el paradigma de la programación voraz, que garantiza que una zona nunca superará la capacidad T de un camión.

Si se observa el pseudocódigo, se ve que una vez obtenida esta primera división se llama a la función `postprocess`, que corresponde a la segunda iteración del algoritmo mencionada anteriormente y se explora en profundidad en la sección siguiente.

4.2. Postproceso de las zonas

Como se ha mencionado anteriormente, las zonas sufren un postproceso. Su objetivo es suplir las carencias que puede tener el algoritmo que nos aporta la división inicial ya que este solo itera una vez sobre los nodos, provocando que zonas se creen con pocos contenedores, especialmente en el último subgrafo.

La siguiente explicación se apoya en el pseudocódigo correspondiente a la función `postprocess`. Se deestaca que se ha simplificado el este respecto a la implementación original para reducir su complejidad y hacerla más fácil de seguir.

Algorithm 3 Postproceso de las zonas

```
function POSTPROCESS(zones, T)
    for Z_i ∈ zones do
        A ← Z_{i−1}
        B ← Z_i
        C ← Z_{i+1}
        if B_size > 3  &  (A_size > 3 | 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 ≤ 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, el significado de los distintos símbolos usados es:

- La función `weight(x)` nos devuelve el peso del nodo x.
- B es la *i-ésima* zona contenida en *zones*.
- A y C son la zona anterior y posterior a B, respectivamente.
- $A_x \mid B_x \mid C_x$ es el contenedor x de la zona A, B o C
- T es, como siempre, la capacidad de los camiones.

Como se mencionaba anteriormente, el objetivo de esta función es equilibrar el peso de las zonas y eliminar las que contengan solo dos contenedores. 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(X, y)` se encarga de esto mismo, mover el contenedor y a la zona X. 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 demasiado pequeñas. 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:

- Mover B_0 a la anterior zona A
- Mover B_{-2} a la siguiente zona C
- Mover B_{-2} a la anterior zona A
- 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 V_0), y si es el caso, se elimina del listado de zonas.

Ahora que ya tenemos un listado de n zonas, se crea un subgrafo G_s para cada una de ellas. Esto se realiza mediante una función que se ha omitido de los pseudocódigos, pero que obtiene todas las aristas conectadas a los contenedores de cada zona y crea un nuevo subgrafo $G_s \subset G$, añadiéndolo a una lista que es devuelta al finalizar con todas las zonas del grafo.

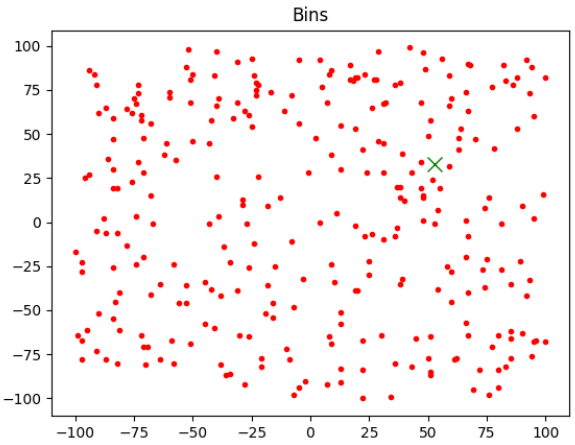

Figura 1: Grafo con 294 contenedores antes de dividir

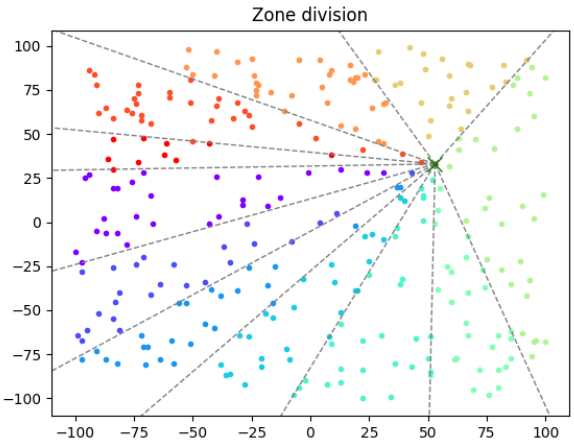

Figura 2: El mismo grafo dividido en 11 zonas de $T = 1{,}500$

Obtenidos todos los subgrafos, podemos empezar a buscar la solución de los n *TSPs* que se nos presentan.

5. La función de evaluación

Para poder decidir el valor de cada camino propuesto como solución del TSP, se debe definir una función de evaluación $f(R)$ lo calcule dicho valor. Antes de explorar cómo se encuentra la ruta final de cada subgrafo, definiremos esta función.

Volviendo a la sección 2, podemos ver que el objetivo de los caminos resultantes para cada zona es que sean lo más rápidos posible. Para ello, se debe determinar el valor de una arista individual, pues todo camino R de un grafo G esta formado por n aristas, siendo n el número de contenedores presentes en el camino.

Puesto que nos interesa saber el tiempo que se tarda en recorrer una arista, sabemos que el valor de una arista E vendrá dado por una fórmula que, a partir de su distancia y la velocidad media para recorrerlo, calcule el tiempo que se tarda en atravesar. Llamaremos de ahora en adelante $h(E)$ a la fórmula que nos da el valor de una arista E. Una primera aproximación de dicha fórmula sería:

$$h(E) = \overline{v} \cdot s + c \tag{11}$$

donde $\overline{v}$ es la velocidad media de la arista E, l es la longitud de E y c es una constante que corresponderá al tiempo que se tarda en recoger un contenedor, ya que por cada arista recorrida se visita un nuevo contenedor, el cual hay que recoger.

La siguiente pregunta a responder es, **¿cómo se obtiene la longitud de una arista?** Para resolver esto, se ha optado por una variación de la **distancia Manhattan**, ya que esta aproxima más la distancia real que podemos esperar en una ciudad que otros métodos. Antes de hablar de la modificación realizada a la distancia Manhattan, se debe especificar qué es esta distancia.

La distancia Manhattan se define como la distancia desde un punto a hasta otro punto b realizando movimientos únicamente en una dirección. Es decir, si ambos puntos se encuentran en un plano cartesiano de coordenadas, para ir de a a b solo podríamos realizar movimientos que cambien el valor de una de nuestras coordenadas. Dichos movimientos son paralelos a los ejes de coordenadas X e Y.

El cálculo formal de esta distancia es el resultado de sumar la diferencia absoluta en las coordenadas de ambos puntos a, b y su resultado viene dado por la fórmula:

$$d(a, b) = |a_x - b_x| + |a_y - b_y| \tag{12}$$

El problema con este cálculo viene cuando se considera que el grafo no se encuentra sobre un plano bidimensional, sino que sus nodos se encuentran situados encima de una esfera, por lo que este cálculo de la distancia no es plenamente fiel a la realidad. Para solucionarlo, usamos la **fórmula Haversine** en combinación con la distancia Manhattan.

Para realizar este cálculo, debemos primero encontrar diferencia en radianes entre las coordenadas x e y de los puntos a y b. A partir de este momento, al estar hablando de puntos en una esfera, se llamará a x **latitud** y a y **altitud**. Esta diferencia viene dada por aplicar las siguientes fórmulas:

$$\begin{aligned} \Delta latitud &= |a_{latitud} - b_{latitud}| \\ \Delta altitud &= |a_{altitud} - b_{altitud}| \end{aligned} \tag{13}$$

Una vez obtenidas $\Delta latitud$ y $\Delta altitud$, se usan para calcular la distancia recorrida en la altitud

y latitud, usando las fórmulas:

$$\begin{aligned} d_{latitud} &= 2r \cdot atan2(\frac{\sin(\Delta latitud)}{2}, \sqrt{1 - (\frac{\sin(\Delta latitud)}{2})^2}) \\ d_{altitud} &= 2r \cdot atan2(\frac{\sin(\Delta altitud)}{2}, \sqrt{1 - (\frac{\sin(\Delta altitud)}{2})^2}) \end{aligned} \tag{14}$$

donde r es el radio de la esfera en la que se encuentran los puntos a y b, y $atan2()$ viene dada por la fórmula 9. Finalmente, se obtiene la distancia entre ambos puntos con:

$$d(a, b) = d_{latitud} + d_{altitud} \tag{15}$$

6. Resolviendo el TSP

Antes de explicarse la resolución *TSP* para cada $G_s \in G$, se expone el proceso realizado hasta dar con el procedimiento final.

Para determinar qué algoritmo (o combinación de ellos) se va a usar en la solución definitiva, se han implementado varios métodos para aproximar la solución y se han comparado entre sí, tanto en el tiempo que han tardado en encontrar una solución cómo en el valor de nuestra función de evaluación para cada ruta R encontrado. Así mismo, se ha implementado un algoritmo que computa un **límite inferior** para una instancia del TSP, con el que podemos aproximar cómo de cerca se encuentra cada solución a la óptima.

A continuación se explican todos los algoritmos usados en este trabajo, divididos en tres grandes categorías, *límite inferior* [6.1], *creación de recorridos* [6.2] y *optimización de recorridos* [6.3].

Cabe destacar que todos los verifican antes de ejecutarse que el grafo no es un caso simple, donde el número de contenedores es menor o igual a 2, ya que estos casos siempre se manejan individualmente al tener soluciones triviales.

6.1. Cómputo del límite inferior

Dentro de esta categoría encontramos dos algoritmos; **1-tree** y **Held-Karp**. Aunque si que es cierto que la estimación de un límite inferior que produce Held-Karp es mejor que la producida por 1-tree, hay que mencionar que este es usado cómo base de la solución proporcionada por Held-Karp.

6.1.1. Algoritmo 1-tree

Este algoritmo busca encontrar un límite inferior para cualquier camino R_s de un grafo G. Funciona creando un árbol mínimo de G y añadiendo las dos aristas más cortas desde el contenedor inicial *start* del árbol mínimo.

1-tree sigue los siguientes pasos para encontrar un límite inferior:

1. Se elige un contenedor inicial s
2. Se computa el **Árbol de Mínimo Recubrimiento** del grafo G excluyendo s usando el algoritmo de **Prim**.
3. Se obtienen todos los contenedores que se unen a s por una sola arista y se ordenan ascendentemente por la longitud de esta arista.
4. Se añaden los dos contenedores con las aristas mas cortas al MST y se añade el contenedor s.

Algorithm 4 1-Tree Construction

```
function ONE_TREE(G, start)
    (cost, T) ← PRIM(G, start)
    aux ← distances from start to all nodes
    sort aux in ascending order by weight
    added ← 0
    for each (v, w) in C[1 :] do
        if (start, v) ∉ T and (v, start) ∉ T then
            cost ← cost + w
            append (start, v) to T
            added ← added + 1
            if added = 2 then
                break
            end if
        end if
    end for
    return (T, cost)
end function
```

El motivo por el que este algoritmo proporciona un límite inferior para cualquier camino R_s de G es que crea un camino R' que contiene un ciclo. Este ciclo cumple menos restricciones que un ciclo hamiltoniano, por lo que obtenemos un valor inferior al óptimo y, por lo tanto, un límite inferior.

6.1.2. Algoritmo Held-Karp

Como se ha expresado al inicio de esta sección, el algoritmo **Held-Karp** proporciona un límite inferior más cercano al óptimo que **1-tree**, ya que utiliza el resultado de este repetidamente.

El **limite inferior Held-Karp** da una solución a la relajación lineal del TSP. Esta relajación permite usar variables no enteras, convirtiendo el problema en uno de programación lineal. Es decir, mientras que en el TSP estándar se cumple que una arista $x_{ij} = 0 \ \vee \ x_{ij} = 1$ para indicar si se ha visitado o no, en su versión lineal la misma arista cumple $x_{ij} \in [0, 1]$, pudiendo tomar cualquier valor entre 0 y 1.[10]

El algoritmo funciona siguiendo el siguiente proceso:

1. Se inicializan las penalizaciones o multiplicadores de Lagrange π_i de cada nodo y se guarda el valor de la matriz de distancias D. Se establece el número inicial de iteraciones it a 0.
2. Se crea una matriz de distancias D' análoga a la matriz de distancias original D. La matriz D' se forma sumando a cada arista G_{ij} contenida en D las penalizaciones π_i y π_j de los contenedores i y j respectivamente.
3. Se genera el **1-tree** usando esta nueva matriz de distancias.
4. Se encuentra el *grado* de cada contenedor del 1-tree. Con este grado, calculamos el subgradiente $subGrad$ de cada contenedor $a \in G$ usando la formula.

$$subGrad_a = grado_a - 2 \tag{16}$$

5. Se calcula la cota inferior usando el valor del **1-tree**. Podemos obtener este valor gracias a la fórmula

$$lb_{HeldKarp} = lb_{1-tree} - \sum_{i=0}^{n} 2\pi_i \tag{17}$$

en la que n es el total de contenedores y π_i es la penalización del contenedor i. Esta fórmula corresponde al valor de la función dual de la relajación lagrangiana. [9]

6. Si el valor obtenido en el paso anterior es el mejor límite inferior lo guardamos. Volvemos a poner la matriz de distancias D a su valor original.

7. Actualizamos los valores de $\pi_i \ \forall \ i \in G$ usando la fórmula

$$\pi_i = (\frac{1}{it+1}) \cdot subGrad_i \tag{18}$$

   Donde it es la iteración del algoritmo actual.

8. En caso de que $grado_i = 2 \ \forall \ i \in G$, se termina de iterar y se devuelve el mejor límite inferior encontrado. De no cumplirse esta condición y no haber llegado al número máximo de iteraciones, se suma 1 a it y se vuelve al paso 2.

Para ilustrar mejor el proceso, el pseudocódigo del algoritmo se encuentra a continuación.

Algorithm 5 Held-Karp Lower Bound

```
function HELD_KARP_LB(G, start, miter)
    π ← [0,0,...,0,0]
    best_lb ← −∞
    best_edges ← None
    original ← G.distances
    for it = 0 to miter − 1 do
        adjusted ← G.distances_ij + π_i + π_j for all (i, j)
        G.distances ← adjusted
        (T, cost) ← ONE_TREE(G, start)
        degree ← node degrees in T
        subgrad ← [degree_i − 2] for all i
        lb ← cost − 2 · Σπ
        if lb > best_lb then
            best_lb ← lb
            best_edges ← T
        end if
        if all degree_i = 2 then
            break
        end if
        t ← 1/(it + 1)
        for i = 0 to n − 1 do
            π_i ← π_i + t · subgrad_i
        end for
        G.distances ← original
    end for
    return (best_edges, best_lb)
end function
```

Una vez se ha obtenido el límite inferior, se pasa a usar los algoritmos para crear rutas iniciales y finalmente, resolver el TSP.

6.2. Creación de rutas iniciales

Esta categoría contiene un único algoritmo, encargado de crear un recorrido desde el cual iniciar el proceso de optimización y resolución del TSP. El algoritmo es **Nearest Neighbor**, el cual se explica

a continuación. Cabe señalar que también se han probado los algoritmos de optimización partiendo de un camino aleatorio para verificar si el resultado obtenido justifica el coste que supone ejecutar Nearest Neighbor, ya que cuenta con una complejidad temporal de $O(n^2)$, siendo n el número de contenedores del grafo G donde se ejecuta. Más adelante se mostrarán ambos resultados con cada algoritmo desarrollado.

A partir de esta sección, se mostrará el resultado de ejecutar los algoritmos sobre el siguiente grafo de 50 contenedores.

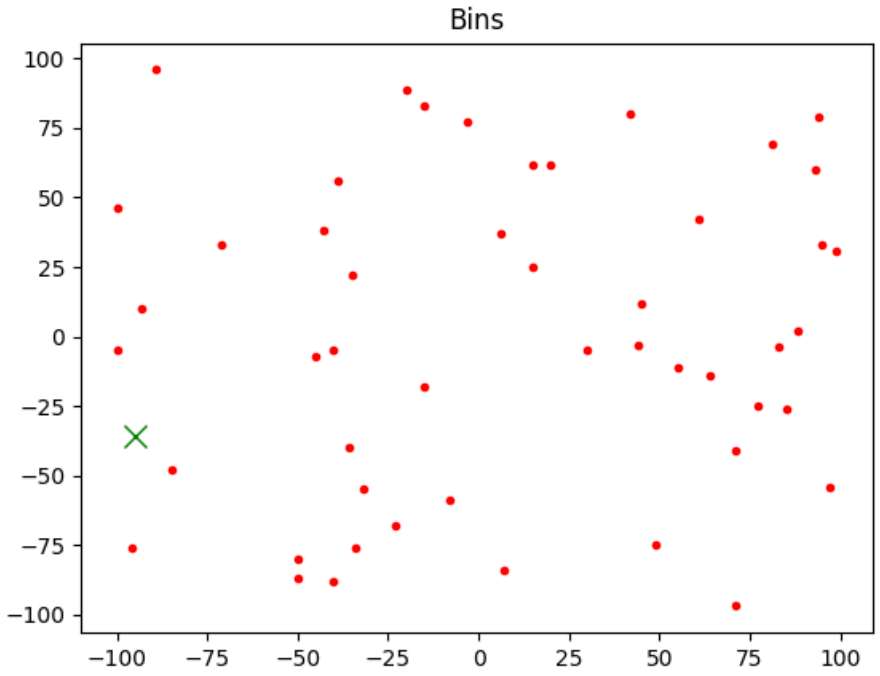

Figura 3: Grafo original con 50 contenedores

6.2.1. Algoritmo Nearest Neighbor

El algoritmo de Nearest Neighbor es bastante simple. Éste se encarga de crear un camino R inicial a partir de un nodo s. A continuación, visita su *vecino* más cercano (siendo un vecino un nodo que solo está conectado a s por una arista) y se repite el proceso desde este vecino.

Podemos ver su funcionamiento paso a paso para encontrar una ruta R para un grafo G siguiendo los siguientes pasos:

1. Se inicializa una lista vacía de los contenedores que forman el camino R y otra de los que se han visitado. Se elige un contenedor inicial s. Se crea una cola de prioridad para almacenar los nodos descubiertos que quedan por visitar y se añade a esta s.
2. Se obtiene el primer elemento de la cola de prioridad u y se marca como visitado.
3. A partir del contenedor u, se obtienen sus vecinos en orden de distancia.
4. Se añade el primer vecino de u que aún no se haya visitado.
5. Si la cola de prioridad está vacía se finaliza el algoritmo. En caso de no ser así, se vuelve al paso 2.

Algorithm 6 Nearest Neighbor Tour

```
function NEAREST_NEIGHBOR(G, start)
    path ← [ ]
    visited ← [False] × G.nodes
    pq ← empty priority queue
    PUSH(pq, start)
    while pq not empty do
        u ← POP(pq)
        visited[u] ← True
        append u to path
        neighbors ← all (n, distance_{u,n}) in G.distances[u]
        sort neighbors by increasing distance_{u,n}
        for each n in neighbors do
            if not visited[n] then
                PUSH(pq, n)
                break
            end if
        end for
    end while
    append path[0] to path                                  ▷ Close the tour
    return (path, EVALUATE(path))
end function
```

Cabe destacar que se ha utilizado una implementación haciendo uso de una cola de prioridad para mejorar la eficiencia del algoritmo en grafos de gran tamaño. [15]

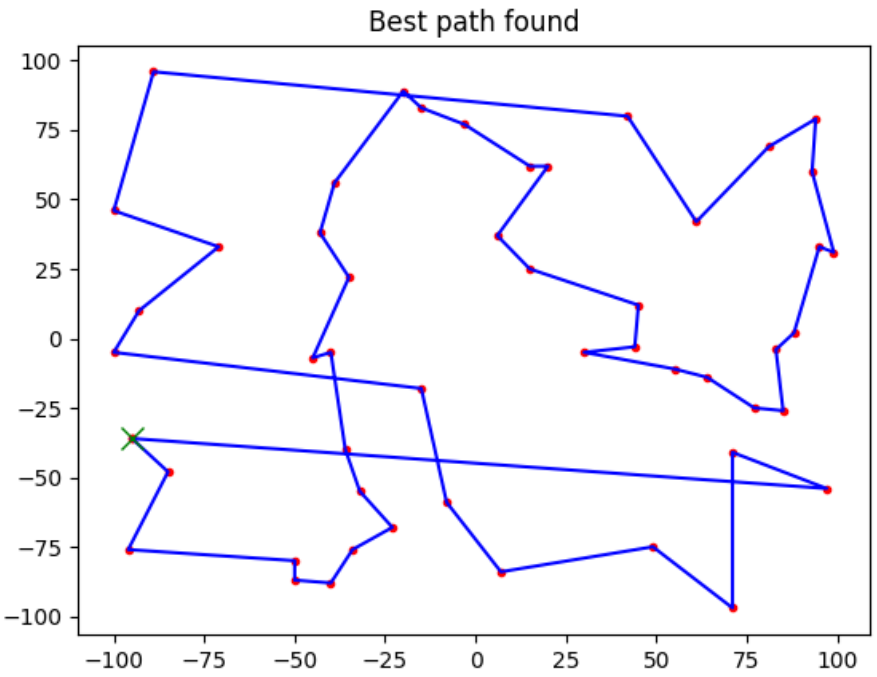

Figura 4: El grafo de la figura 3 tras ejecutar Nearest Neighbor

6.3. Optimización de recorridos

Aquí encontramos los diferentes algoritmos usados para acercar un recorrido inicial al óptimo. Todos ellos son algoritmos basados en metaheurísticas que intentan mejorar el resultado de nuestra función de evaluación. Las siguientes secciones están dedicadas cada una a un algoritmo distinto.

6.3.1. Algoritmo 2-opt

El algoritmo **2-opt** es un algoritmo de mejora local basado en la modificación reiterada de una ruta usando movimiento simple [10]. 2-opt realiza un movimiento que consiste en eliminar dos aristas formando dos nuevas rutas, y conectándolas de modo que la ruta resultante R' sea distinta a la ruta original R.

Siguiendo el estudio de *Englert, Röglin y Vöcking* [5], 2-opt puede requerir un número exponencial de pasos para encontrar un óptimo local, aunque se encontró que en muestras aleatorias, el peor caso suele tener eficiencia polinomial ($O(n^2)$).

Antes de explicar el algoritmo, es importante ver qué movimiento se realiza sobre el grafo. El movimiento empleado trata de invertir el orden en el que se recorren los nodos de una ruta comprendidos entre dos contenedores i y j. Es decir, dada una ruta $R = [0, 1, 4, 3, 2, 5, 0]$, realizar el movimiento sobre esta ruta con $i = 4$ y $j = 2$ nos devuelve como resultado $R' = [0, 1, 2, 3, 4, 5, 0]$. Este movimiento se ha llamado **flip**.

A nivel técnico, el movimiento *flip* hace que dadas dos posiciones x e y de la lista l que representa R, el elemento l_x pase a ser el l_y, el l_{x+1} sea el l_{y-1} y así sucesivamente.

Explicado el movimiento fundamental que realiza 2-opt, los pasos que se siguen son los siguientes:

1. Se obtiene la ruta inicial R, su coste ($f(R)$) y se inicializa el flag de mejora a *Verdadero*.
2. Se generan todos los pares de indices (i, j) de contenedores en R.
3. Para cada par (i, j) con $i \neq j$ y $j > i + 1$, se realiza la operación *flip*. En caso de que el R' resultante tenga un coste menor que el mejor actual, se actualiza la mejor ruta encontrada.
4. En caso de que no se haya encontrado ningún R' mejor a la solución mejor actual, se establece el flag de mejora a *Falso* y se finaliza el algoritmo. En caso contrario, se vuelve al paso 2, partiendo del coste del mejor R' encontrado como nueva ruta inicial.

Algorithm 7 2-Opt

```
function TWO_OPT(path)
    best ← path
    best_value ← EVALUATE(best)
    improved ← true
    while improved do
        improved ← false
        for i ← 0 to n − 2 do
            for j ← i + 2 to n − 1 do
                new_path ← FLIP(best, i, j)
                new_value ← EVALUATE(new_path)
                if new_value < best_value then
                    best ← new_path
                    best_value ← new_value
                    improved ← true
                end if
            end for
        end for
    end while
    return (best, best_value)
end function
```

A pesar de ser un algoritmo simple, 2-opt es muy usado para encontrar soluciones rápidas al TSP ya que tiene un tiempo de ejecución bajo en comparación a la calidad de las soluciones encontradas, aunque cuenta con la desventaja de quedar encajado en mínimos locales de los que es difícil salir.

Al ver los resultados de ejecutarse sobre un camino inicial aleatorio frente al resultado de Nearest Neighbor (NN), se puede observar que hay una clara mejora usando este último, siendo el resultado del camino aleatorio *204.007* y el de NN *186.641*, que supone una mejora del 4,3 %.

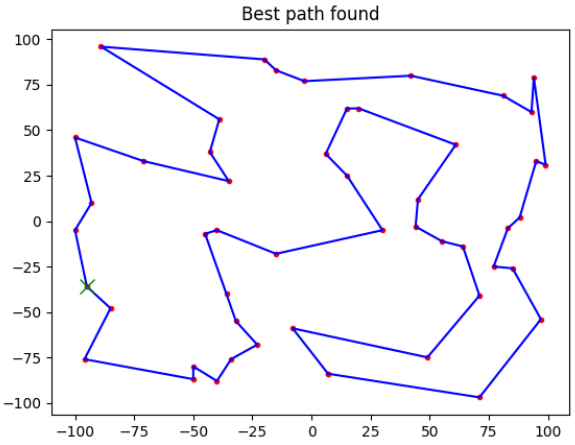

Figura 5: 2-opt con ruta inicial aleatoria.

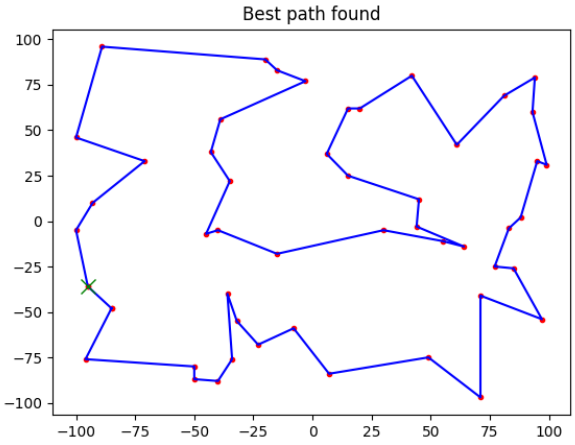

Figura 6: 2-opt partiendo de la solución de NN.

Gracias a esto, se puede empezar a observar la clara ventaja que ofrece el uso de Nearest Neighbor antes de ejecutar un algoritmo más avanzado. Esto nos demuestra que 2-opt **no es un algoritmo constructivo**. Es decir, el resultado final depende ampliamente de la ruta R elegida como punto de partida.

6.3.2. Algoritmo de Simulated Annealing

Simulated Annealing (abreviado como SA) es un algoritmo probabilístico utilizado para encontrar el óptimo global de una función objetivo a través de aproximaciones iterativas. Su nombre y fundamento provienen del proceso físico del **recocido en metalurgia**, en el cual un material se calienta a una temperatura elevada y posteriormente se enfría lentamente, permitiendo que los átomos se reorganicen hacia una estructura de menor energía, es decir, más estable. Este proceso físico fue adaptado por Kirkpatrick et al. (1983) al ámbito de la optimización combinatoria, donde el estado del sistema representa una solución posible y la energía del sistema corresponde al valor de la función objetivo [12].

A diferencia de algoritmos deterministas que pueden quedar atrapados en mínimos locales, SA permite aceptar con cierta probabilidad soluciones peores que la actual, especialmente en etapas tempranas del proceso, gracias al parámetro de temperatura que va disminuyendo gradualmente en función de un factor de enfriamiento. Gracias a este mecanismo, SA evita quedar atrapado permanentemente en un óptimo local, permitiendo avanzar en la exploración del espacio de soluciones [2]. La probabilidad de aceptación se modela habitualmente mediante la función de Boltzmann:

$$P(\Delta E, T) = \exp\left(-\frac{\Delta E}{T}\right) \tag{19}$$

donde ΔE es el incremento en la función objetivo y T es la temperatura en ese paso.

SA ha sido ampliamente aplicado en problemas como el TSP, con buenos resultados en la práctica cuando se implementa con un esquema de enfriamiento adecuado [14]. A pesar de no garantizar una convergencia al óptimo global en tiempo polinomial, sí puede demostrar convergencia bajo ciertas condiciones teóricas con enfriamiento logarítmico [7].

El algoritmo implementado parte de una ruta R inicial y va evaluando cambios. En la codificación realizada, se aprovecha el movimiento `flip` usado en **2-opt** ya que tiene mejor potencial que realizar movimientos aleatorios. Una vez generada una ruta R' resultado de realizar `flip` con dos índices aleatorios, se comprueba si su valor es mejor que el de R. Es decir, si $f(R') < f(R)$. De ser así, se acepta la nueva ruta y se continúa iterando. En caso contrario, se genera un número aleatorio λ y se compara con un valor P, siendo este el resultado de aplicar la fórmula 19. En caso de que $\lambda \leq P$, la solución se aceptará aun siendo peor. Cabe destacar que, cuantas más iteraciones se hayan realizado, menor será P, por lo que será más difícil aceptar rutas peores que la actual.

Los pasos a seguir de la ejecución son:

1. Se inicializa la ruta R inicial y el valor de la función de evaluación $f(R)$. Se inicializa la temperatura y el valor de enfriamiento α.
2. Se selecciona el índice en R de dos contenedores aleatorios para realizar el movimiento `flip`. Llamaremos a estos índices i y j.
3. Se realiza `flip` en la ruta actual R usando i y j como índices sobre los que realizar el movimiento para obtener R' y $f(R')$.
4. Se compara $f(R')$ con $f(R)$ y se calculan los valores de λ y P. Si se cumple $f(R') < f(R)$ o $\lambda \leq P$, se acepta la nueva ruta como R.
5. Se aplica el enfriamiento con la fórmula:

$$T' = T \cdot \alpha \tag{20}$$

En caso de haber mejorado la solución, se restablece el contador de iteraciones estancado y se añade 1 al total de iteraciones. En caso contrario, se actualizan ambos contadores *stagnated* y *it* sumándoles 1.

6. Si la temperatura no ha alcanzado su valor mínimo permitido o el número de iteraciones con solución estancada, se vuelve al paso 2.

Algorithm 8 Simulated Annealing for TSP

```
function SimulatedAnnealing(path, niter, mstag)
    curr ← path
    curr_val ← evaluate(curr)
    best, best_val ← curr, curr_val
    T ← 1000, α ← 0,999, stagnated ← 0
    while T > 10^−255 do
        if stagnated ≥ mstag then
            break
        end if
        (i, j) ← random indices of R with i < j
        next ← flip(curr, i, j)
        val ← evaluate(next)
        if val < curr_val or rand([0, 1]) < exp (curr_val − val)/T then
            curr, curr_val ← next, val
            if val < best_val then
                best, best_val ← next, val
                stagnated ← 0
            end if
        else
            stagnated ← stagnated + 1
        end if
        T ← T · α
    end while
    return (best, best_val)
end function
```

Como se puede ver, en este algoritmo se aplica una comprobación para evitar ejecuciones cuando no se encuentra una solución mejor. Esto permite al algoritmo parar antes de terminar si no hay mejoría en un tiempo razonable [13, 18].

Sobre este grafo (figura 3), los resultados son mejores en el caso de partir de una ruta R aleatoria, siendo este *193.291*, que partiendo de la ruta aportada como resultado por Nearest Neighbor, arrojando esta ejecución un valor de *200.770*. Esto se debe a que Nearest Neighbor tiende a quedar atrapado en óptimos locales al construir soluciones de forma codiciosa, lo cual es ampliamente reconocido en la literatura [16]. En cambio, un punto de partida aleatorio puede facilitar una mejor exploración del espacio de soluciones mediante el enfriamiento simulado, permitiendo escapar de óptimos locales de forma más efectiva [1].

Cabe destacar que en caso de partir del resultado de Nearest Neighbor, el resultado es peor que con el algoritmo **2-opt**, ya que este último aplica mejoras locales que optimizan sistemáticamente los intercambios de aristas, mejorando consistentemente la calidad de la solución inicial [4].

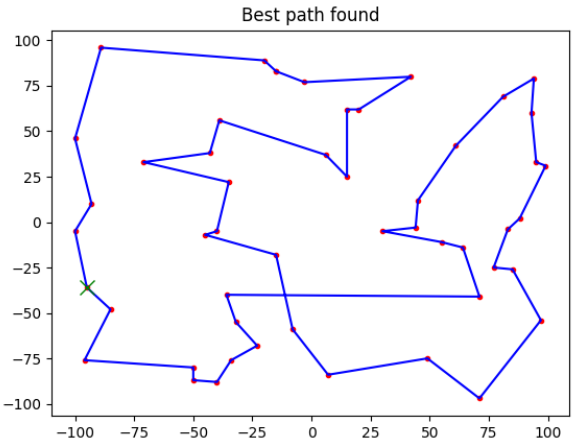

Figura 7: SA con ruta inicial aleatoria.

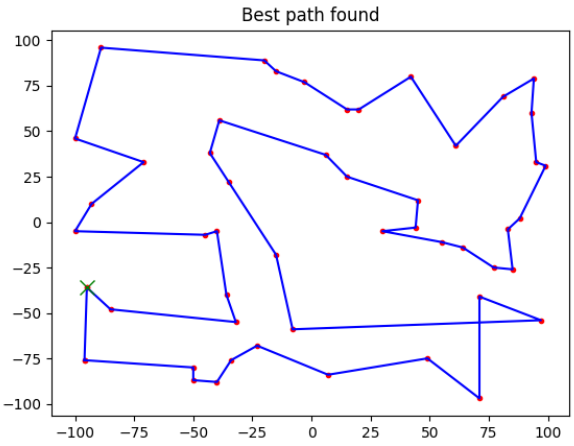

Figura 8: SA partiendo de la solución de NN.

6.3.3. Algoritmo de Búsqueda Tabú

La Búsqueda Tabú (TS por sus siglas en inglés) es un método de búsqueda metaheurístico utilizado ampliamente en la optimización matemática para superar mínimos locales y explorar eficientemente el espacio de soluciones. Fue propuesto originalmente por Fred Glover como una extensión de técnicas de búsqueda local, incorporando una memoria adaptativa para evitar volver a visitar soluciones previamente exploradas [8].

TS emplea una lista para restringir temporalmente ciertos movimientos. Esta estrategia le permite explorar de forma más efectiva el espacio de soluciones, manteniendo un equilibrio entre *intensificación* (explotación de zonas prometedoras) y *diversificación* (exploración de nuevas regiones del espacio de búsqueda).

La implementación usada parte de una ruta R y explora todos los caminos vecinos, buscando una solución mejor que la actual. Los vecinos de R se obtienen usando una función auxiliar `get_neighbors()` que llama al movimiento `flip` con todos los pares de índices (i, j) de R. Una vez obtenidos todos los R' vecinos de R, se comprueba el valor de $f(R')$ para cada uno de ellos y se añade el mejor de todos a la lista tabú. Si un vecino se vuelve a evaluar pero está en esta lista, se ignorará.

La ejecución paso a paso del algoritmo es la siguiente:

1. Se inicia una lista tabú l vacía y se establece la mejor solución como R, con valor $f(R)$. Se establece el tamaño máximo de la lista tabú σ.
2. Se obtienen todos los vecinos de R usando `get_neighbors()` y se evalúa cada uno de ellos en caso de no estar en l. Se consigue así el mejor R' vecino de R.
3. Se añade a l el R' seleccionado en el paso anterior. Si al añadirlo se supera el tamaño máximo σ, se elimina el elemento más antiguo de l.
4. En caso de que $f(R') < f(R)$, se actualiza la mejor ruta.
5. Si no se ha actualizado la mejor ruta, se incrementa en 1 el contador de generaciones sin mejora. En caso contrario, se restablece a 0.
6. Se aumenta el número de iteraciones realizadas. Si se supera el máximo de iteraciones o el máximo de iteraciones sin mejora, se finaliza la ejecución. En caso contrario, se vuelve al paso 2.

Algorithm 9 Tabu Search for TSP

```
function TabuSearch(path, niter, mstag, tsize)
    best ← path
    curr ← best
    tabu ← [ ]
    i, stagnated ← 0, 0
    for it = 1 to niter do
        if stagnated ≥ mstag then
            break
        end if
        neighbors ← GetNeighbors(curr)
        best_n, best_val ← None, ∞
        for all n ∈ neighbors do
            if n ∉ tabu then
                val ← evaluate(n)
                if val < best_val then
                    best_n, best_val ← n, val
                end if
            end if
        end for
        curr ← best_n
        append(tabu, best_n)
        if length(tabu) > tsize then
            pop_front(tabu)
        end if
        if evaluate(curr) < evaluate(best) then
            best ← curr
            stagnated ← 0
        else
            stagnated ← stagnated + 1
        end if
    end for
    return (best, evaluate(best))
end function
```

Al ejecutar este algoritmo sobre el grafo usado de referencia (figura 3), vemos que el resultado es ligeramente mejor al de ambos, Simulated Annealing,y 2-opt, obteniendo un valor de *192.295* cuando la ruta inicial es aleatoria y de *185.360* cuando la ruta inicial es el resultado de **NN**.

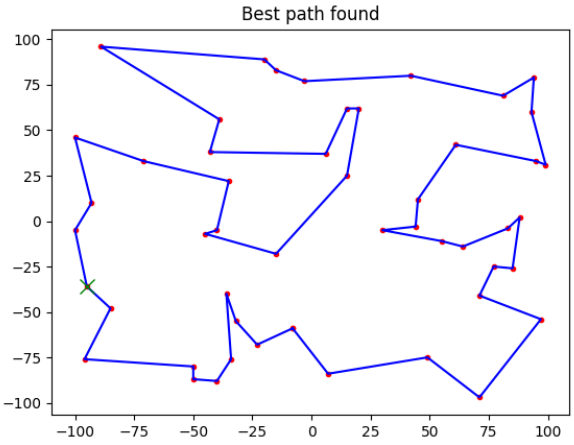

Figura 9: TS con ruta inicial aleatoria.

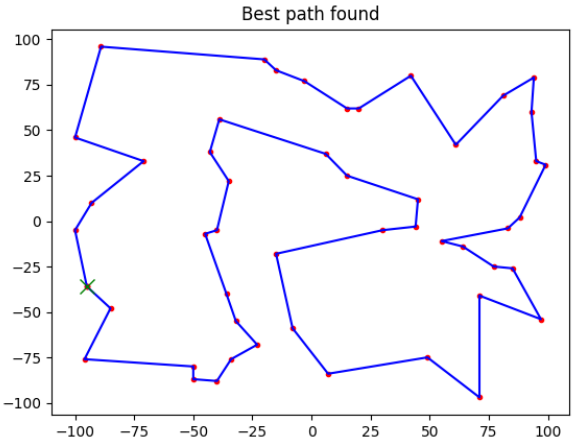

Figura 10: TS partiendo de la solución de NN.

Antes de finalizar esta sección, se destaca que a la vista de los resultados se volvieron a ejecutar tanto **Simulated Annealing** como **Búsqueda Tabú**, pero partiendo del camino inicial que se genera tras ejecutar *2-opt* sobre el resultado de *Nearest Neighbor*. Es decir, el flujo de esta solución fue el siguiente:

$$nearestNeightbor(G) = R_1 \rightarrow 2opt(G, R_1) = R_2 \rightarrow \begin{cases} simulatedAnnealing(G, R_2) \\ tabuSearch(G, R_2) \end{cases} \Rightarrow R_3$$

Aplicando esto, la solución obtenida fue la misma; *184.824* en ambos casos. A continuación, se muestran dos imágenes con las dos rutas generadas, donde se puede observar que efectivamente no solo tienen el mismo valor de $f(R_3)$, sino que son las mismas rutas finales.

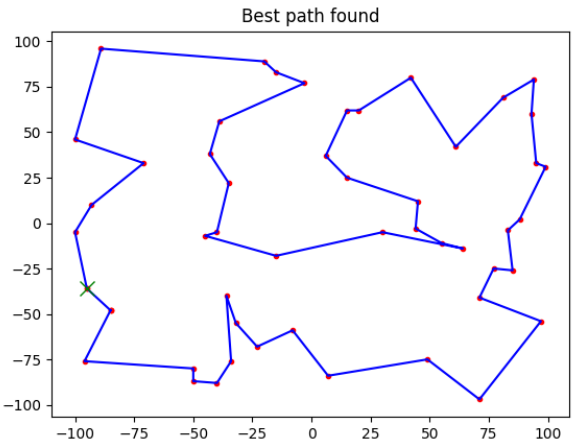

Figura 11: SA partiendo de la ruta de 2opt.

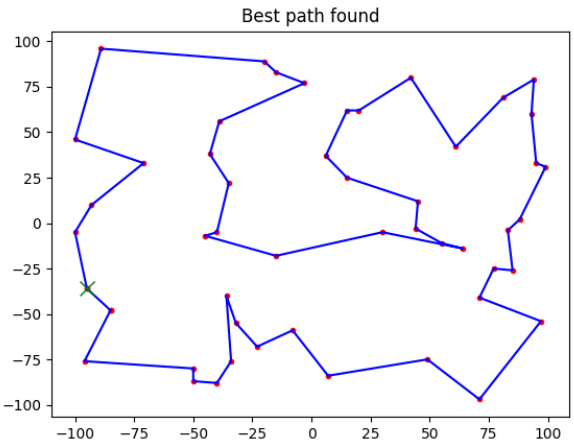

Figura 12: TS partiendo de ruta de 2opt.

7. Elección de hiperparámetros

Algunos de los algoritmos utilizados en este trabajo incorporan los denominados **hiperparámetros**, un concepto ampliamente utilizado en el campo del *machine learning* y en la optimización heurística en general [6, 3]. Los hiperparámetros son parámetros cuyo valor no se aprende automáticamente durante la ejecución del algoritmo, sino que deben ser establecidos previamente y pueden tener un impacto significativo en la calidad de la solución y en la velocidad de convergencia.

Para determinar los valores más adecuados de los hiperparámetros en este contexto, se han llevado a cabo dos procesos de *benchmarking*, uno para el algoritmo de **Simulated Annealing** y otro para **Búsqueda Tabú**. En ambos casos, se evaluaron múltiples combinaciones de hiperparámetros mediante experimentos sistemáticos, con el objetivo de identificar aquellas configuraciones que ofrecen un mejor rendimiento en función del número de contenedores del grafo sobre el cual se calcula la solución.

El análisis experimental permitió seleccionar valores cuasi-óptimos para los distintos tamaños de instancia, contribuyendo así a una mejora sustancial en el rendimiento global de los algoritmos. Esta estrategia de ajuste de hiperparámetros basada en *benchmarks* es fundamental en la práctica para adaptar métodos heurísticos a problemas específicos, tal y como se ha evidenciado en estudios recientes sobre metaheurísticas aplicadas al TSP y variantes [11, 17].

En las siguientes secciones, se explica el proceso realizado para encontrar los hiperparámetros finalmente usados.

7.1. Elección de los hiperparámetros de Simulated Annealing

Para el algoritmo de Simulated Annealing (sección 6.3.2), se han buscado los valores de la **temperatura**, α (valor de enfriamiento) y del **máximo de iteraciones sin mejora** ($mstag$) que aportan unos resultados más cercanos al óptimo.

El *benchmark* usado realiza una muestra aleatoria de *4.000* grupos de hiperparámetros, escogiendo entre un rango permitido de valores para cada uno de ellos. Dicho rango está comprendido en los siguientes límites:

- $T \in \{1000, 2000, 3000, \ldots, 10000\}$
- $\alpha \in \{0{,}9, 0{,}901, 0{,}902, \ldots, 0{,}999\}$
- $mstag \in \{1000, 2000, 3000, \ldots, 10000\}$

Un ejemplo de grupo sería $\{T = 2000 \mid \alpha = 0{,}957 \mid mstag = 500\}$.

Tras haberse seleccionado los grupos a ejecutar, se realiza una ejecución de SA por grupo en cada uno de los grafos seleccionados. Dichos grafos tienen de 25 a 200 contenedores, lo que nos permite derivar una fórmula para los hiperparámetros en función del número de contenedores.

Después de la ejecución del *benchmark*, se obtuvo la siguiente gráfica, la cual se usó para derivar fórmulas para el cálculo del valor de cada uno de los hiperparámetros en función del número de contenedores del grafo.

Podemos ver la evolución de los mejores valores en función del número de contenedores n_k del grafo.

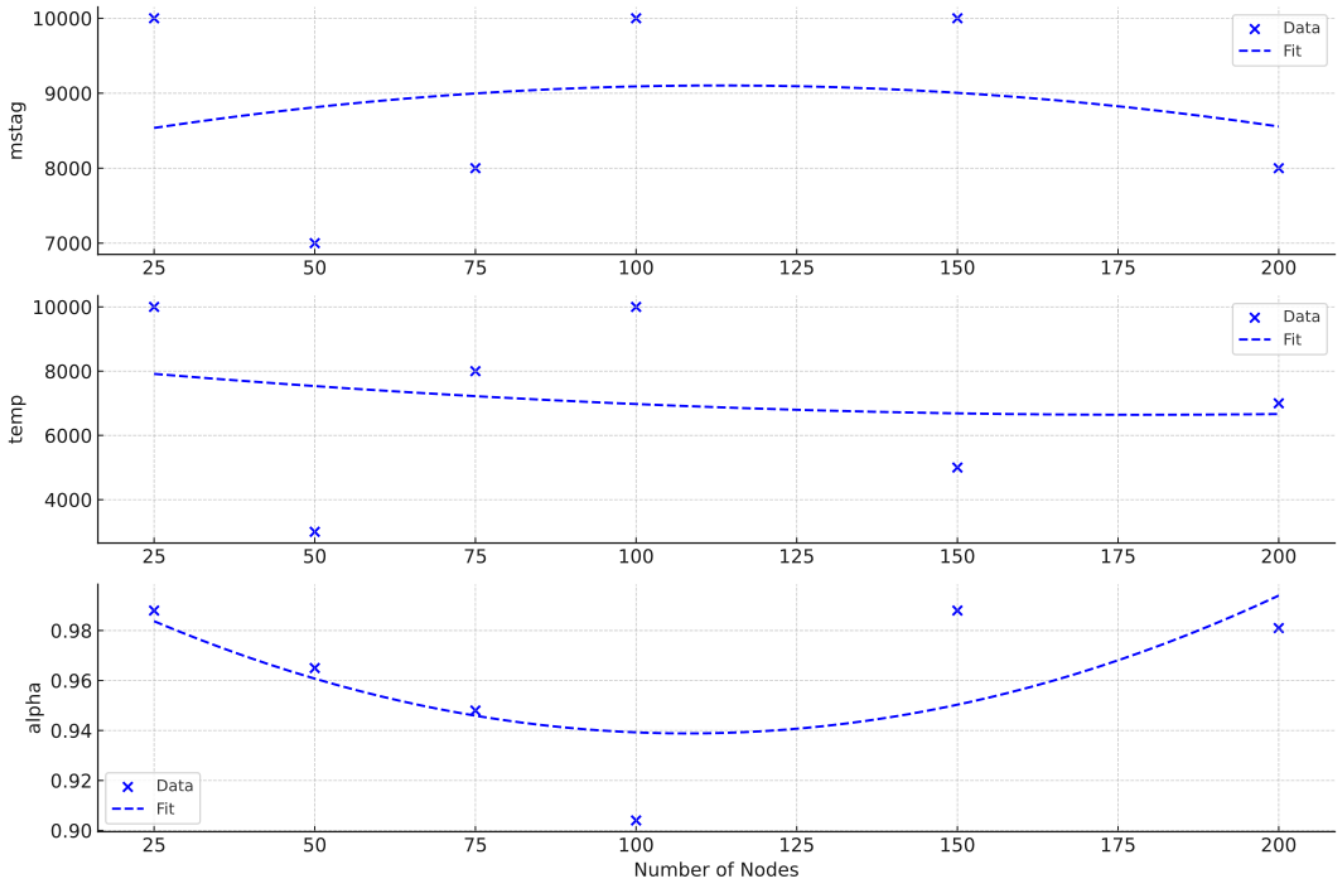

Figura 13: Evolución de los hiperparámetros en función de n_k

De donde derivamos las siguientes fórmulas del cálculo de los hiperparámetros:

$$T = -2{,}3542 \cdot 10^{-6}\, n^5 + 0{,}00106083\, n^4 - 0{,}17325\, n^3 + 13{,}22917\, n^2 - 441{,}0\, n + 11000 \tag{21}$$

$$\alpha = 1{,}71959 \cdot 10^{-10}\, n^5 - 8{,}57735 \cdot 10^{-8}\, n^4 + 1{,}63778 \cdot 10^{-5}\, n^3 - 0{,}0015453\, n^2 + 0{,}070927\, n + 0{,}8398 \tag{22}$$

$$mstag = -6{,}0937 \cdot 10^{-6}\, n^5 + 0{,}00277188\, n^4 - 0{,}4728125\, n^3 + 39{,}5625\, n^2 - 1531{,}25\, n + 21000 \tag{23}$$

Las cuales se aplicarán de forma general. En caso de que los grafos sean pequeños ($n_k < 25$) o grandes ($n_k > 200$), se aplicarán los siguientes valores, decididos de antemano:

$$T = \begin{cases} 10{,}000 & \text{if} \quad 3 \leq n < 20, \\ 7{,}000 & \text{if} \quad 200 < n \leq \infty. \end{cases} \tag{24}$$

$$\alpha = \begin{cases} 0{,}98 & \text{if} \quad 3 \leq n < 20, \\ 0{,}98 & \text{if} \quad 200 < n \leq \infty. \end{cases} \tag{25}$$

$$mstag = \begin{cases} 8{,}000 & \text{if} \quad 3 \leq n < 20, \\ 8{,}000 & \text{if} \quad 200 < n \leq \infty. \end{cases} \tag{26}$$

Las fórmulas 21, 22 y 23, aún no siendo totalmente exactas debido al tamaño de la muestra de datos, dan los resultados que se han visto mejores en grafos de *25, 50, 75, 100, 150* y *200* contenedores. No se ha podido realizar un muestreo más exhaustivo debido a una limitación de potencia en los equipos usados, pues realizar un gran número de iteraciones de cualquiera de estos algoritmos requiere de bastante potencia y tiempo.

7.2. Elección de los hiperparámetros de Búsqueda Tabú

El algoritmo de Búsqueda Tabú (sección 6.3.3) cuenta con tres hiperparámetros. Estos son el **máximo de iteraciones sin mejora** (*mstag*), el **número máximo de iteraciones** (*niter*) y el tamaño máximo de la lista tabú σ. Como en la sección anterior, también se selecciona una muestra aleatoria de varios grupos de hiperparámetros. En este caso, la muestra es de *50*, pues a diferencia del benchmark de *Simulated Annealing*, el número total de posibles grupos es de *150* debido a restricciones de potencia, ya que la Búsqueda Tabú necesita una mayor cantidad de recursos que el anterior. Los límites de los rangos de valores elegidos son los siguientes:

- $mstag \in \{100, 200, 300, \ldots, 1000\}$
- $niter \in \{100{,}000, 200{,}000, 300{,}000, \ldots, 500{,}000\}$
- $\sigma \in \{100{,}000, 150{,}000, 200{,}000, \ldots, 300{,}000\}$

Al igual que con SA, se seleccionan los grupos y se ejecutan en los grafos seleccionados. Tras obtener los resultados, se puede observar que la influencia de los hiperparámetros en la solución final es mínimo, ya que los diez mejores resultados para todos los archivos de prueba llegan a la misma solución, aún teniendo valores distintos. Por lo tanto, se ha concluido que los valores finales son:

- $mstag = 250$
- $niter = 100{,}000$
- $\sigma = 100{,}000$

Número de contenedores	**Rango de** *mstag*	**Rango de** *niter*	**Rango de** σ	**Valor de** $f(G)$
25	$800 - 1{,}000$	$100{,}000 - 500{,}000$	$100{,}000 - 300{,}000$	122,294
50	$800 - 1{,}000$	$100{,}000 - 500{,}000$	$100{,}000 - 300{,}000$	184,297
75	$800 - 1{,}000$	$100{,}000 - 500{,}000$	$100{,}000 - 300{,}000$	204,431
100	$800 - 1{,}000$	$100{,}000 - 500{,}000$	$100{,}000 - 300{,}000$	222,935
150	$800 - 1{,}000$	$100{,}000 - 500{,}000$	$100{,}000 - 300{,}000$	300,418

Cuadro 1: Comparativa de los 10 mejores resultados para cada grafo.

Como se puede observar, se han seleccionado valores bajos dentro de los intervalos ya que al demostrarse el bajo efecto que tienen, se puede optar por ellos para minimizar el impacto en el tiempo de ejecución. Para validar esto mismo, se han realizado ejecuciones sobre los mismos grafos con los valores de los hiperparámetros mostrados anteriormente y se han obtenido las mismas soluciones que se observan en la tabla 1 menos en el caso del grafo con 50 contenedores, en cuyo caso el resultado es ligeramente peor.

Se destaca que para este algoritmo se ha excluido el grafo de *200* contenedores debido a limitaciones de potencia y tiempo.

8. Análisis de rendimiento

Para comprobar el rendimiento y la calidad de las soluciones aportadas por los distintos algoritmos, se han realizado tres *benchmarks* completos. Uno sobre grafos **pequeños** (de 20 a 70 contenedores), otro en grafos **grandes** (de 71 a 120 contenedores) y un último en instancias seleccionadas de **TSPLIB**, una librería que cuenta con instancias de grafos en los que se aporta su solución óptima.

Los algoritmos probados son los expuestos anteriormente (secciiones 6.2 y 6.3), así como combinaciones de estos, en los que la ruta R resultante de uno se establece como la ruta inicial del siguiente algoritmo.

En las siguientes secciones se muestran todos los datos obtenidos con el fin de determinar un flujo final de resolución, en el que la calidad de la solución sea siempre lo más alta posible. Este flujo se concluirá tras presentar todos los datos obtenidos.

8.1. Grafos pequeños

Para las pruebas sobre grafos pequeños (entre 20 y 70 contenedores), se han obtenido los siguientes resultados:

Algoritmo(s)	$f(R)$	**Tiempo de ejecución** (s)	**Porcentaje al límite inferior**
Nearest Neighbor (NN)	191,836	0,1s	43,96 %
2-opt	180,317	0,23s	35,26 %
Simulated Annealing (SA)	171,901	0,37s	28,86 %
Búsqueda Tabú (TS)	166,576	8,15s	25,01 %
NN + 2-opt	169,891	0,19s	27,65 %
NN + SA	167,590	0,35s	25,9 %
NN + TS	162,444	6,74s	22,04 %
NN + 2-opt + SA	165,712	0,32s	24,42 %
NN + 2-opt + TS	163,089	7,38s	22,46 %
NN + 2-opt + TS + SA	162,988	0,27s	22,37 %
NN + 2-opt + SA + TS	162,793	7,01s	22,2 %

Cuadro 2: Resultados promedio del benchmark en grafos de 20 a 70 contenedores

Observando estos valores, se puede ver claramente que en el caso de grafos pequeños, el flujo que mejor solución aporta es el de **Nearest Neighbor + Búsqueda Tabú**, seguido de cerca por el último, que combina todos los flujos acabando con la búsqueda tabú.

También cabe destacar la gran mejoría que hay al usar *Nearest Neighbor* antes de ejecutar otro algoritmo más complejo, mejorando sustancialmente la calidad de las soluciones aportadas. Este primer paso mejora el resultado notablemente más que las siguientes adiciones de algoritmos más complejos. Esto se debe a que cada vez es más computacionalmente complejo y costoso mejorar una solución, más aún si se parte de una ruta que puede estar ya encerrada en un mínimo local, lo que prácticamente imposibilita la mejora pasado determinado punto. Este efecto está presente principalmente en grafos más grandes, como veremos a continuación.

8.2. Grafos grandes

Sobre grafos más grandes (de 71 a 120 contenedores) podemos derivar más conclusiones, debido al aumento exponencial del entorno de soluciones.

Algoritmo(s)	$f(R)$	**Tiempo de ejecución (s)**	**Porcentaje al límite inferior**
Nearest Neighbor (NN)	*282.681*	0,13s	43,6 %
2-opt	*263.295*	1,58s	33,8 %
Simulated Annealing (SA)	*317.998*	0,43s	60,1 %
Búsqueda Tabú (TS)	*244.750*	94,38s	24,29 %
NN + 2-opt	*247.873*	1,23s	25,94 %
NN + SA	*255.828*	0,44s	29,72 %
NN + TS	*237.829*	62,29s	20,82 %
NN + 2-opt + SA	*246.164*	0,42s	25,04 %
NN + 2-opt + TS	*240.379*	73,7s	22,09 %
NN + 2-opt + TS + SA	*240.274*	0,45s	22,04 %
NN + 2-opt + SA + TS	*239.578*	72,85s	21,66 %

Cuadro 3: Resultados promedio del benchmark en grafos de *71* a *120* contenedores

Con esta tabla y teniendo en cuenta los resultados obtenidos anteriormente (tabla 2) podemos llegar a numerosas conclusiones. Primero, salta a la vista la importante bajada en la calidad de la solución de Simulated Annealing. Esto indíca que hay una relación proporcional fuerte entre el número de contenedores y el valor de $simulatedAnnealing(G, R) = R_1 \Rightarrow f(R_1)$.

El resto de algoritmos tienden a ofrecer mejores resultados, aunque con diferencias mínimas, que en grafos más pequeños. Esto es debido al aumento del *espacio de soluciones* mencionado anteriormente, lo que permite mejorar la **exploración** que pueden realizar los distintos algoritmos. Es importante destacar que la diferencia entre las dos mejores soluciones (*NN + TS* contra el flujo completo) ha aumentado, pasando de un 0,16 % a un 0,84 %, lo que muestra que el último método funciona mejor en grafos más pequeños.

Por último, con estos resultados se ve claramente el aumento del tiempo de ejecución en grafos grandes. Mientras que antes el algoritmo más lento tardaba 7,38s en ejecutarse, ahora el tiempo medio más alto ha subido a 94,38s, un incremento que implica un uso de más de 12 veces el tiempo requerido. Este hecho se debe, entre otros factores, al incremento factorial del tamaño del espacio de soluciones ya que este viene dado por la fórmula

$$\frac{(n-1)!}{2} \tag{27}$$

Aplicándola, vemos que en un grafo de 20 contenedores hay un total de $6{,}08 \times 10^{16}$ soluciones, mientras que en uno de 200 contenedores hay $3{,}16 \times 10^{374}$, un aumento muy grande que provoca tener que explorar un espacio sustancialmente más amplio.

8.3. Grafos importados de *TSPLIB*

La librería **TSPLIB** contiene numerosos grafos con distinto número de nodos y características. Además de estos recursos, cada uno tiene asociada su solución óptima, por lo que es muy útil para obtener la calidad real de los algoritmos. Se han seleccionado 7 grafos de entre 76 y 176 contenedores.

La siguiente gráfica muestra los valores obtenidos por los mejores algoritmos en este *benchmark*.

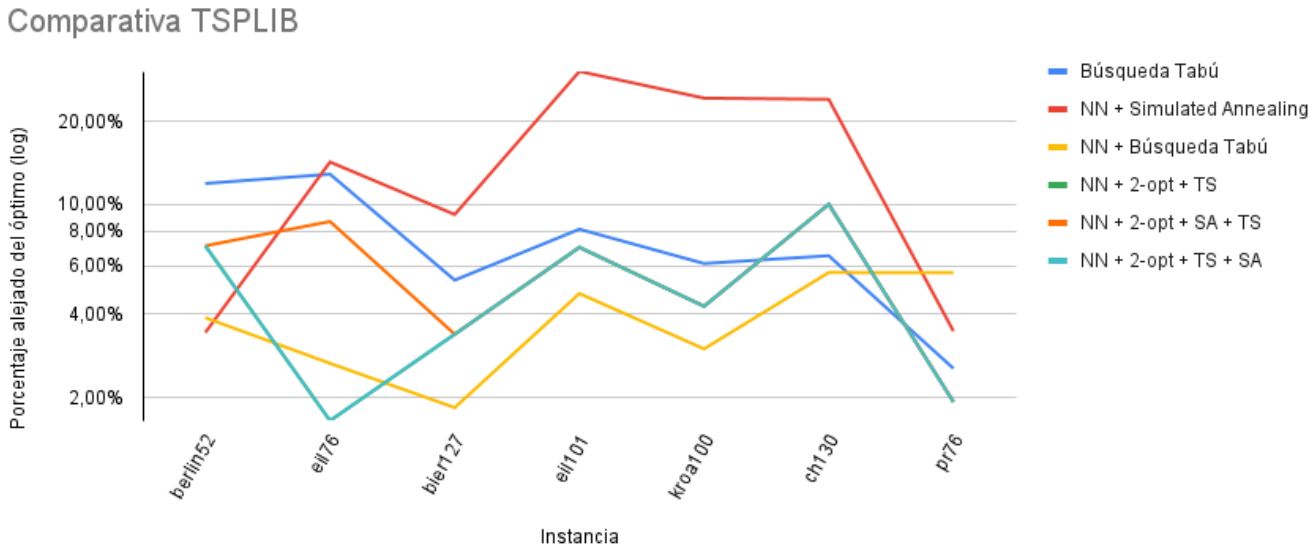

Figura 14: Porcentaje alejado del óptimo de distintos algoritmos

Gracias a este gráfico, se puede ver muy claramente la estrecha relación entre número de contenedores y calidad de la solución de **Simulated Annealing**, aunque cabe destacar que en la instancia más pequeña de todas,*Berlin_52*, obtuvo la mejor solución, seguido de cerca por **Neareest Neighbor + Búsqueda Tabú**. Como norma general, este algoritmo aporta la mejor solución, aunque se puede ver que en grafos más pequeños, los flujos más complejos obtienen una mejor solución.

Destaca también la paridad entre las soluciones aportadas por los tres flujos más complejos que, menos en la instancia *Eil_76*, obtienen los mismos resultados entre sí.

En cuanto al tiempo empleado por cada algoritmo, observando la siguiente gráfica se obtienen las conclusiones descritas más adelante.

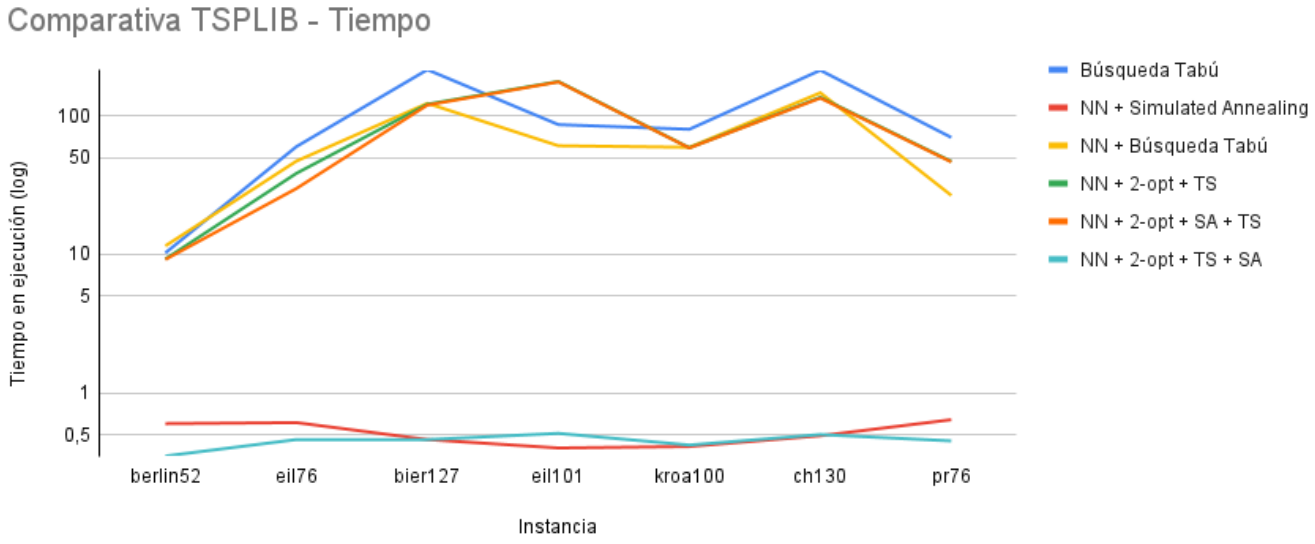

Figura 15: Tiempo de ejecución de distintos algoritmos

En el caso de los flujos que incluyen la *Búsqueda Tabú* como su último algoritmo, a más complejo es, más rápido se llega a una solución. Esto es debido a que se ha reducido el espacio a explorar

drásticamente, y la Búsqueda Tabú se beneficia ampliamente de ello.

En los flujos cuyo algoritmo principal es *Simulated Annealing* pasa lo contrario debido a que este no se beneficia tanto de la ventaja que aporta partir de una solución mejor. De hecho, no solo a nivel temporal, sino en el valor de $f(R)$ se ve que puede ser perjudicial. A pesar de lo descrito, el tiempo de ejecución de estos flujos es **muy inferior** al de los anteriores, del orden de unas 100 veces más rápido.

8.4. Conclusiones tras el análisis

Tras el análisis realizado, sacamos las siguientes conclusiones.

- Para grafos pequeños, presumiblemente con menos de 30 nodos, el mejor flujo a ejecutar es **NN + 2-opt + SA + TS**. Al ser ejecutado en grafos pequeños, la diferencia temporal no es lo suficientemente grande para justificar el uso de un algoritmo más rápido(tabla 2).
- En grafos más grandes, el flujo que suele aportar una mejor solución es **NN + TS**, siendo algo mejor que los flujos más complejos. A pesar de esto, el tiempo de ejecución aumenta mucho en comparación con grafos más reducidos.
- En caso de grafos muy grande, con más de 200 contenedores, el tiempo empleado en encontrar una solución usando *NN + TS* no justifica la mejora encontrada frente al flujo **NN + 2-opt + TS + SA**, el cual completa su ejecución aproximadamente 100 veces más rápido.
- De no ser acompañado por la *Búsqueda Tabú*, **Simulated Annealing** tiene una calidad de solución muy baja en grafos grandes. Esto indica que se beneficia ampliamente de partir de una buena (o muy buena) ruta previa.

Con estas conclusiones, se ha codificado el flujo definitivo de ejecución, el cual se expone en la siguiente sección.

9. Resolución del problema planteado

Gracias a los resultados obtenidos en los *benchmarks* (sección 8), podemos plantear los pasos a seguir para definir un flujo de solución. Dicho flujo es el siguiente:

1. Se evalúa el número de contenedores n_k del subgrafo G_k y se decide el algoritmo (o grupo de algorírmos) a ejecutar.
2. Se ejecuta el flujo seleccionado y se muestran los resultados mediante un gráfico. Se devuelven también la ruta R_k encontrada y el valor $f(R_k)$ de esta.

Algorithm 10 Run: Execute Best Search Strategy

```
function Run(dir, name)
    n ← GetNumberOfNodes(graph)
    (pnn, vnn) ← NearestNeighbor(dir, name)
    if n < 30 then
        (p2opt, v2opt) ← TwoOpt(pnn, dir, name)
        (psa, vsa) ← SimulatedAnnealing(p2opt, dir, name)
        (p1, v1) ← TabuSearch(psa, dir, name)
        return (p1, v1)
    else if n > 200 then
        (p2opt, v2opt) ← TwoOpt(pnn, dir, name)
        (pts, vts) ← TabuSearch(p2opt, dir, name)
        (p1, v1) ← SimulatedAnnealing(pts, dir, name)
        return (p1, v1)
    else
        (p1, v1) ← TabuSearch(pnn, dir, name)
        return (p1, v1)
    end if
end function
```

Como se observa en el pseudocódigo, se aprovecha el número de contenedores n_k del grafo a resolver, aplicando las conclusiones de sección anterior para utilizar una serie de algoritmos u otra en función de este valor.

Para mostrar el funcionamiento final del proceso, se ha partido del grafo de la figura 1, siendo dividido como se expone en la sección 4 y obteniendo la distribución que se mostró en la figura 2. Aplicando el flujo de resolución final sobre todos los G_k generados, se obtienen los siguientes resultados:

Subgrafo	**Valor de la función de evaluación**	**Número de contenedores**	**Tiempo transcurrido**
G_1	59,736	29	2,85s
G_2	64,829	26	1,61s
G_3	65,937	27	1,75s
G_4	53,341	27	1,73s
G_5	57,160	30	2,27s
G_6	48,621	29	2,44s
G_7	61,701	31	1,79s
G_8	37,308	25	1,58s
G_9	48,049	35	2,37s
G_{10}	59,262	34	2,2s
G_{11}	38,495	11	0,64s
Total	594,443	294	21,23s

Cuadro 4: Resultados sobre el grafo de la figura 1

A continuación se muestra un ejemplo de dos rutas generadas.

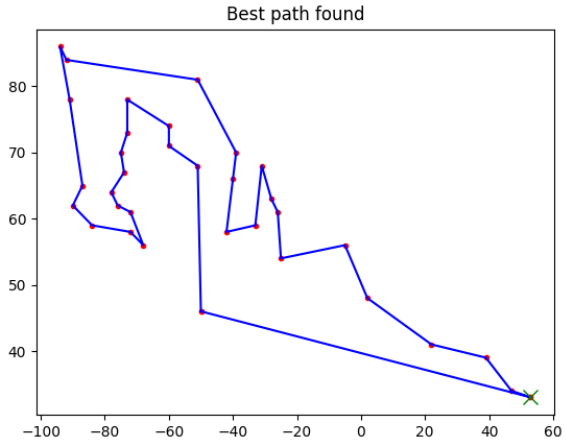

Figura 16: Recorrido final del subrafo G_{10}

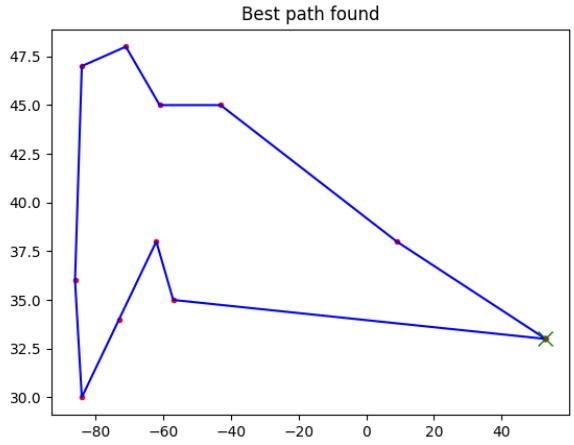

Figura 17: Recorrido final del subrafo G_{11}

Finalmente se ha generado una gráfica mostrando todos los caminos sobre la distribución original:

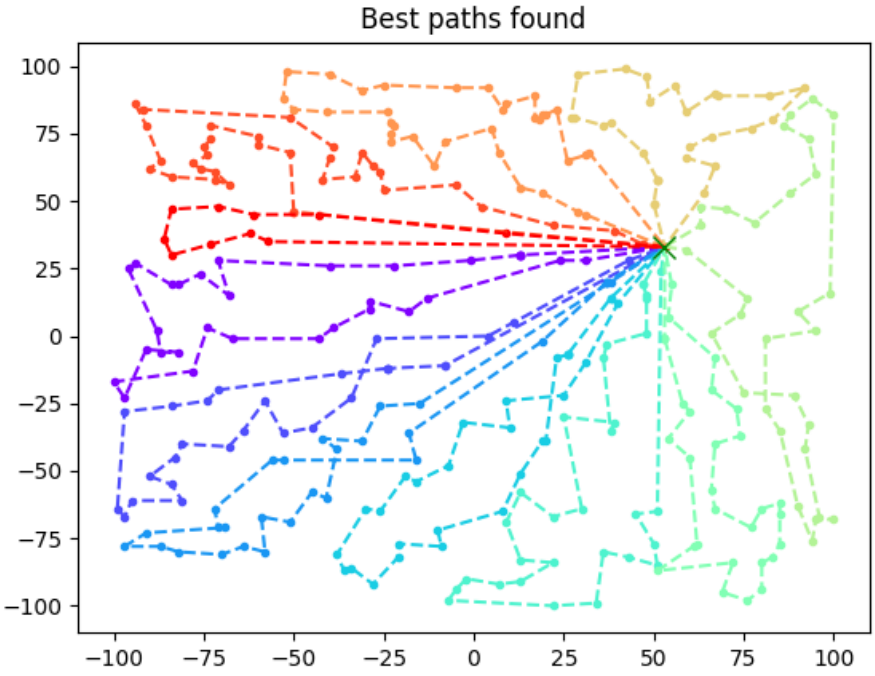

Figura 18: Grafo final de 294 contenedores, dividido y con sus rutas calculadas.

10. Conclusiones finales y pasos futuros

Para finalizar, se ha podido llegar a un flujo de trabajo, demostrado mediante el uso de *benchmarks* y pruebas de esfuerzo, en el que se aprovechan los datos conocidos sobre el grafo G en el que se calculan las rutas.

El objetivo del documento se ha podido cumplir, ya que se ha realizado un análisis de distintos métodos de resolución, cimentado en un análisis basado en datos que permite mejorar sustancialmente las soluciones actuales en materia de gestión de residuos urbanos.

Cabe destacar que a pesar de haber encontrado una solución final que es mayoritariamente satisfactoria, los resultados finales siempre deberían ser evaluados visualmente por una persona, pues hay veces que se pueden aplicar correcciones simples finales que incrementan la calidad de la solución final.

Como cierre, indicar los pasos que se podrían realizar en un futuro. Debido a las limitaciones de potencia indicadas a lo largo del documento, ha quedado pendiente realizar más *benchmarks* para pulir aún más el flujo final. Queda también como posibles mejoras añadir más algoritmos que comparar para hacer que el análisis realizado sea más exhaustivo.

Referencias

[1] E. H. L. Aarts y J. H. M. Korst. "Simulated annealing and boltzmann machines". En: *Wiley-Interscience* (1988).

[2] E.H.L. Aarts y J.H.M. Korst. *Simulated annealing: theory and applications*. Springer, 1988.

[3] James Bergstra et al. "Algorithms for hyper-parameter optimization". En: *Advances in neural information processing systems* 24 (2011).

[4] G. A. Croes. "A method for solving traveling-salesman problems". En: *Operations Research* 6.6 (1958), págs. 791-812. DOI: 10.1287/opre.6.6.791.

[5] Matthias Englert, Heiko Röglin y Berthold Vöcking. "Worst-case and probabilistic analysis of the 2-opt algorithm for the tsp". En: *Algorithmica* 68.1 (2014), págs. 190-264. DOI: 10.1007/s00453-013-9801-4.

[6] Matthias Feurer y Frank Hutter. "Hyperparameter optimization". En: *Automated Machine Learning* (2019), págs. 3-33.

[7] S. Geman y D. Geman. "Stochastic relaxation, gibbs distributions, and the bayesian restoration of images". En: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-6.6 (1984), págs. 721-741.

[8] Fred Glover. "Future paths for integer programming and links to artificial intelligence". En: *Computers & Operations Research* 13.5 (1986), págs. 533-549.

[9] Michael Held y Richard M. Karp. "The traveling-salesman problem and minimum spanning trees". En: *Operations Research* 18.6 (1971), págs. 1138-1162. DOI: 10.1287/opre.18.6.1138.

[10] 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 Emile H. L. Aarts y Jan Karel Lenstra. Wiley, 1997, págs. 215-310.

[11] Kaan Karabulut. "Performance analysis of simulated annealing, tabu search and genetic algorithms for the traveling salesman problem". En: *International Journal of Computer Applications* 57.29 (2012), págs. 1-5.

[12] S. Kirkpatrick, C. D. Gelatt y M. P. Vecchi. "Optimization by simulated annealing". En: *Science* 220.4598 (1983), págs. 671-680.

[13] 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.

[14] P.J.M. van Laarhoven y E.H.L. Aarts. *Simulated annealing: theory and applications*. Springer, 1987.

[15] Md. Ziaur Rahman et al. "Improvement of the nearest neighbor heuristic search algorithm for traveling salesman problem". En: *Journal of Engineering Advancements* 5.1 (2024), págs. 25-34. DOI: 10.38032/jea.2024.01.004.

[16] D. J. Rosenkrantz, R. E. Stearns y P. M. Lewis. "An analysis of several heuristics for the traveling salesman problem". En: *SIAM Journal on Computing* 6.3 (1977), págs. 563-581. DOI: 10.1137/0206041.

[17] Kenneth Sørensen y Fred Glover. *Metaheuristics—theory and applications*. Springer, 2013.

[18] P. J. M. Van Laarhoven y E. H. L. Aarts. *Simulated annealing: theory and applications*. Springer, 1987.

ANEXO I: Libreria

Para poder ejecutar y probar los distintos algoritmos tratados en este trabajo, se ha codificado una librería completa, llamada `model` que incluye definiciones de grafos, métodos y funciones clásicas de estos, y métodos auxiliares que facilitan la implementación de nuevo código usando esta librería. En paralelo a esta, se ha desarrollado la librería `algorithms`, que contiene el código fuente de los algoritmos tratados aquí como pseudocódigo, entre otros.

A lo largo de este anexo, se profundiza en los detalles técnicos de estas librerías, con hincapié especial en `model`.

Las clases `Node` y `Edge`

Como se explica en la sección 3, un grafo está compuesto de dos partes, los **nodos** y las **aristas**. Para modelar ambos, se han usado dos clases.

Empezando por la que modela los nodos de un grafo, la clase `Node`. Esta cuenta con los datos que nos dan información referente a los nodos de un grafo, que son los siguientes:

- El **índice** del nodo, que nos permite encontrarlos al ser un dato único. Es decir, cada nodo tiene un solo índice, y este es único entre todos los nodos de un grafo.
- El **peso** o valor de un nodo. Es el valor asociado a cada nodo. En nuestro problema, si tenemos en cuenta que nuestros nodos son contenedores, el peso será la masa de dicho contenedor.
- El atributo **centro**, que identifica el nodo como punto central del grafo. Este nodo cuenta con características únicas, como ser el orígen y destino de los recorridos encontrados por nuestros algoritmos.
- El atributo **visitado**, que puede actualizarse para indicar si un nodo ha sido visitado y así, poder excluirlo de una búsqueda.
- Las **coordenadas** del nodo, que indícan su posición en el espacio.

La clase `Node`, además de aportar información acerca de un nodo, nos permite comparar dos nodos entre sí para obtener la distancia que hay entre ambos, usando las fórmulas indicadas en la sección 5, o usando las fórmulas de la distancia Euclídea

$$|\sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}| \tag{28}$$

para saber la distancia de ambos nodos en un plano bidimensional.

Por otro lado, la clase `Edge` modela las aristas del grafo y tiene los siguientes atributos:

- El **plano** de la arista, ya que puede unir nodos en un espacio bidimensional o tridimensional.
- La **longitud** de la arista, calculada en función del plano en el que se encuentran los nodos que une.
- La **velocidad** promedio a la que se puede recorrer la arista.

- Sus nodos **orígen** y **destino**.
- La **distancia** de la arista y el **tiempo** que se tarda en recorrerla.
- El **valor** que tiene asociado, que va en función de su longitud y el tiempo que se tarda en recorrerla (ver sección 5)

Esta clase no contiene ninguna función auxiliar, pero utiliza las de la clase `Node` para obtener el valor de sus atributos.

La clase `Graph`

Gracias a las clases `Node` y `Edge`, podemos modelar un grafo. Esta estructura guarda el número de nodos y aristas que tiene, así como una representación de ella tanto a modo de **matriz de adyacencia** y **lista de adyacencia**. También se almacena un diccionario que permite enlazar cada objeto `Node` con su índice, para acelerar el acceso a un nodo específico sabiendo solo su índice.

Además de los métodos esperados usados para construir un grafo, se cuenta con una serie de funciones extra que permiten incrementar la funcionalidad de esta clase. Estas funciones nos permiten poblar un grafo mediante un archivo descriptivo, mediante un archivo `.tsp` propio de la librería *TSPLIB*, guardar y cargar un grafo ya especificado anteriormente y crear las divisiones y subgrafos necesarios, de los que se habla más a fondo en la sección 4. La clase cuenta también con métodos de recorridos clásicos en grafos como los algoritmos de **Dijkstra** y **Prim**.

Para más información acerca de estas tres clases, se puede leer la documentación ubicada dentro del código, que sigue los estándares para los `docstrings` en python. También se puede leer esta documentación al usar alguna de las funciones propias de la librería.

Excepciones

Con el objetivo de dotar al código de un sistema de control de errores más sólido y comprensible, se ha implementado un conjunto de excepciones personalizadas que encapsulan de forma clara las distintas situaciones anómalas que pueden surgir durante la ejecución del programa. Estas excepciones no solo permiten detectar y gestionar errores de forma más específica que con las clases genéricas del lenguaje, sino que también proporcionan mensajes detallados que facilitan significativamente el diagnóstico de problemas, tanto en etapas de desarrollo como durante el uso del sistema por parte de terceros.

La definición de estas excepciones responde a la necesidad de capturar situaciones concretas relacionadas con la manipulación de estructuras de grafos, como la ausencia de nodos o aristas, duplicación de elementos, grafos vacíos o configuraciones inválidas. Cada clase de excepción incluye un mensaje de error predeterminado que describe de forma clara la causa del problema, pudiendo ser personalizado si se desea proporcionar mayor contexto. Además, en algunos casos se admite información adicional, como identificadores de nodos o trayectorias implicadas, lo que mejora aún más la trazabilidad del error.

El uso de excepciones propias permite, por tanto, separar las condiciones de error según su naturaleza específica, facilitando un tratamiento más ordenado y predecible dentro del flujo del programa. Desde una perspectiva de diseño, esto promueve una programación defensiva y mejora la **mantenibilidad** del código, ya que cada excepción representa un contrato explícito sobre cómo deben manejarse ciertos fallos estructurales o lógicos en la interacción con el grafo.

Para una descripción más clara de las excepciones, el usuario puede referirse a la documentación de la librería.

Antes de concluir, cabe destacar que se ha desarrollado un *script auxiliar* que permite generar archivos descriptivos de un grafo fácilmente, indicando los parámetros deseados. Este script se encuentra en la ruta de la librería: `/utils/create_models.py`. Los argumentos que se le pueden indicar son: la cantidad de archivos de grafos a generar, el número de nodos máximo y mínimo, las velocidades y pesos máximos y mínimos de las aristas, y el rango de coordenadas en el que deben encontrarse los nodos. Para más información, se puede ejecutar el *script* con el argumento `-h` para mostrar la ayuda.

ANEXO II: Pruebas

A modo de comprobación del correcto funcionamiento de todo el código descrito en el Anexo I (10), se ha desarrollado también una librería de pruebas en la que se validan todas las funciones utilizadas en el código de las tres clases principales.

Estas pruebas verifican que todas las funciones devuelven el valor esperado y pueden ser ejecutadas tanto individualmente como en conjunto, para comprobar que todo el sistema funciona correctamente. Es recomendable ejecutar el paquete de pruebas antes de utilizar la librería por primera vez, ya que puede haber errores al importarla. Para ello, se debe ejecutar el siguiente comando:

```
python3 -m unittest tests/tests.py
```

Al ejecutarlo, también se comprobará que todos los algoritmos expuestos en este documento funcionan correctamente usando una serie de grafos incluidos con la librería.

ANEXO III: Ejecución de un *benchmark*

La librería cuenta con el código necesario para realizar benchmarks que prueben el desempeño de los distintos algoritmos y combinaciones de estos. Para poder ejecutarlo, se debe ejecutar el comando

```
python3 benchmark/benchmark.py
```

en el directorio raíz de la librería. Se le pueden indicar una serie de *flags* para cambiar el número de archivos a probar, sus dimensiones y otros parámetros. Para más información acerca de los posibles argumentos, se puede llamar al script con el *flag* `-h`, el cual mostrará la ayuda.

Adicionalmente, se ha dejado el script de `bash run_benchmark.sh`, utilizado en el proyecto para desarrollar la sección 8.

ANEXO IV: Librerias necesarias

Para poder ejecutar correctamente cualquier archivo de la librería, se deben tener instaladas las siguientes dependencias:

- **Python** → *versión 3.10 o más alta* debido al uso de sentencias y funciones que se incluyeron en esta versión.
- **DEAP** → *versión 1.4.2 o más alta*. Un *framework de computación evolutiva* usado en algunas funciones experimentales que no se han llegado a incluir en la versión final de este documento.
- **matplotlib** → *versión 3.7.1 o más alta*. Librería usada para producir los diferentes gráficos usados en el proyecto.
- **seaborn** → *versión 0.13.2 o más alta*. Librería de visualización de datos basada en *matplotlib*.
- **tabulate** → *versión 0.9.0 o más alta*. Necesaria para ejecutar los *benchmarks*. Usada para dar un formato más legible a las tablas de resultados.
- **tqdm** → *versión 4.57.0 o más alta*. Used to display some benchmarking loops.

Todas estas librerías se pueden instalar automáticamente ejecutando el comando

```
pip install -r requirements.txt
```

en la carpeta raíz del proyecto. Para más información sobre las dependencias, se puede consultar el documento `requirements.txt`, ubicado en la raíz del proyecto.

Se recomienda ejecutar la librería en alguna distribución de **Linux**, ya que su desarrollo se ha realizado en un Sistema Operativo de esa familia y no se puede asegurar su correcto funcionamiento en otras plataformas.