Universidad Nacional Autónoma de México
Escuela Nacional de Estudios Superiores
Unidad Morelia

Aplicación de metaheurísticas en el problema de "clustering"

Investigadores participantes:

Adriana Menchaca Méndez [1]
Elizabeth Montero Ureta [2]
Luis Miguel García Velázquez [1]
Marisol Flores Garrido [1]
Rolando Menchaca Méndez [3]
Saúl Zapotecas Martínez [4]

Alumnos participantes:

Eduardo Manuel Ceja Cruz[1]
Jorge Oscar Vázquez Fuerte[1]
Juan Pablo Maldonado Castro[1]
Saúl Armas Gamiño[1]

Morelia, Michoacán, México. Marzo, 2024

[1]Escuela Nacional de Estudios Superiores, Unidad Morelia, UNAM, México
[2]Departamento de Informática, Universidad Técnica Federico Santa María, Valparaíso, Chile
[3]Centro de Investigación en Computación, IPN, México
[4]Instituto Nacional de Astrofísica, Óptica y Electrónica, Tonantzintla, Puebla, México

II

Trabajo realizado con el apoyo del Programa UNAM-DGAPA-PAPIME PE106923.

Agradecimientos

Queremos expresar nuestro agradecimiento al estudiantado de la Licenciatura en Tecnologías para la Información en Ciencias de la ENES Unidad Morelia por sus valiosos comentarios y colaboración en la mejora de este material didáctico.

Resumen

El desafío del "clustering", también conocido como segmentación de datos, es de vital importancia en el campo de la ciencia de la computación debido a su aplicación en diversas situaciones del mundo real. Estas incluyen el análisis de secuencias genómicas, la detección de fraudes, la identificación de grupos poblacionales para la formulación de políticas públicas y la planificación y logística. Aunque existen algoritmos tradicionales para abordar esta problemática, sus limitaciones han generado un aumento en el uso de metaheurísticas.

En muchas instituciones de educación superior, los estudiantes se matriculan en cursos que tienen como objetivo enseñar técnicas metaheurísticas para abordar problemas de optimización desafiantes. El problema del "clustering" se puede considerar como una tarea de optimización, y a menudo las instancias que se intentan resolver generan definiciones de problemas complejos. Por lo tanto, el estudio y la comprensión de las metaheurísticas aplicadas a la segmentación de datos pueden enriquecer la formación de profesionales en el campo de la computación.

En este documento, se exploran dos problemas relacionados con la segmentación de datos: 1) la definición de zonas geográficas para abordar el problema logístico de las entregas y 2) la definición de grupos en el contexto del problema de vehículo compartido. Para resolver estas problemáticas, se presentan enfoques de optimización tanto mono-objetivo como multi-objetivo, utilizando metaheurísticas como el recocido simulado y los algoritmos evolutivos.

Además, con el fin de facilitar la comprensión y la aplicación práctica de las propuestas presentadas, se incluyen implementaciones en el lenguaje de programación Python. Estas implementaciones se proporcionan como complemento al documento, permitiendo a los lectores explorar y experimentar

con los enfoques descritos de manera más concreta. El uso de Python como lenguaje de programación para estas implementaciones ofrece una ventaja adicional, ya que es ampliamente utilizado en el campo de la computación y cuenta con una gran cantidad de recursos y bibliotecas disponibles para el análisis de datos y la implementación de algoritmos de optimización.

Índice general

Capítulo 1

Introducción

En el mundo de la ingeniería, la ciencia y la industria, es común encontrarse con desafiantes problemas de optimización que requieren soluciones eficientes. Aunque existen numerosos métodos que buscan encontrar la mejor solución posible, es importante tener en cuenta que no hay un método universal que pueda resolver todos los problemas de optimización tal como lo estipula el teorema del "no free lunch".

Una de las técnicas utilizadas para abordar problemas de optimización de alta complejidad son las heurísticas. Estas técnicas buscan encontrar soluciones de calidad en un tiempo de cómputo razonable, aunque no garantizan encontrar la solución óptima. En este documento, nos centraremos en la aplicación de dos metaheurísticas particulares en el contexto del agrupamiento: el recocido simulado y los algoritmos evolutivos.

El recocido simulado es una metaheurística que se inspira en el proceso de enfriamiento y recalentamiento de los metales para encontrar soluciones cercanas a la óptima. Por otro lado, los algoritmos evolutivos se basan en los principios de la selección natural y la evolución biológica para generar soluciones prometedoras.

Estas metaheurísticas ofrecen una forma más general y flexible de abordar diversos tipos de problemas de agrupamiento. Aunque no garantizan encontrar la solución óptima en todos los casos, su capacidad para encontrar soluciones de calidad en un tiempo razonable las convierte en herramientas valiosas en el campo de la optimización.

1.1. Optimización mono-objetivo

Un problema de optimización mono-objetivo se plantea al buscar los valores óptimos de un conjunto de n variables de decisión, representadas por el vector $\vec{x} = [x_1, x_2, \ldots, x_n]^T$. Estas variables pueden ser continuas o discretas y se buscan de manera que satisfagan m condiciones de desigualdad y p condiciones de igualdad. El objetivo es minimizar o maximizar una función objetivo $f(\vec{x})$.

Es importante destacar que minimizar la función objetivo $f(\vec{x})$ es equivalente a maximizar $-f(\vec{x})$. De esta manera, el problema de optimización general se puede expresar de la siguiente manera:

$$\text{Minimizar:} f(\vec{x}) \tag{1.1}$$

tal que:

$$\begin{aligned} g_i(\vec{x}) &\leq 0, \quad i = 1, 2, \ldots, m \\ h_j(\vec{x}) &= 0, \quad j = 1, 2, \ldots, p \end{aligned}$$

donde $g_i(\vec{x})$ representa las condiciones de desigualdad y $h_j(\vec{x})$ representa las condiciones de igualdad. El objetivo es encontrar los valores de $\vec{x}$ que cumplen con todas las restricciones y minimizan (o maximizan) la función objetivo $f(\vec{x})$.

A continuación, se presentan algunas definiciones clave que resultan fundamentales para comprender el problema de optimización. Para una explicación más detallada, se puede consultar el trabajo de referencia [3], donde se aborda con mayor profundidad el tema en discusión. Es relevante tener en cuenta estas definiciones, ya que sentarán las bases para el desarrollo y comprensión de los conceptos que se explorarán a lo largo de este documento.

Definición 1.1.1 (Región factible) *La región factible se denota como* Ω*:* $\Omega = \{\vec{x} \mid \vec{x}$ *cumple todas las restricciones del problema*$\}$*.*

Definición 1.1.2 (Óptimo global) *Un punto* $\vec{x} \in \Omega$ *es un* **mínimo global***, si y solo si* $f(\vec{x}) \leq f(\vec{y})$ *para todo* $\vec{y} \in \Omega$*.*

Definición 1.1.3 (Óptimo local) *Un punto* $\vec{x} \in \Omega$ *es un* **mínimo local***, si y solo si* $f(\vec{x}) \leq f(\vec{y})$ *para todo* $\vec{y} \in N(\vec{x}) \cap \Omega$*. Donde* $N(\vec{x})$ *es un vecindario de* $\vec{x}$*.*

1.2. Optimización multi-objetivo

Un problema de optimización multi-objetivo es aquel en el que se busca encontrar la mejor solución posible considerando múltiples objetivos que pueden entrar en conflicto entre sí. En lugar de buscar una única solución óptima, se busca un conjunto de soluciones que sean las mejores en términos de los diferentes objetivos establecidos. Este tipo de problemas establecen una dificultad más compleja, ya que implica encontrar un equilibrio entre los diferentes objetivos y tomar decisiones que sean lo más beneficiosas posible en términos de todos los objetivos considerados.

El problema de optimización multi-objetivo general puede ser escrito como sigue:

$$\text{mín } \vec{f}(\vec{x}) = [f_1(\vec{x}), f_2(\vec{x}), \ldots, f_k(\vec{x})]^T \tag{1.2}$$

tal que:

$$g_i(\vec{x}) \leq 0, i = 1, 2, \ldots, m$$
$$h_j(\vec{x}) = 0, j = 1, 2, \ldots, p$$

donde $\vec{x} = [x_1, x_2, \ldots, x_n]^T$ es el vector de variables de decisión, las cuales pueden ser continuas o discretas; g_i y h_j son las restricciones de desigualdad e igualdad, respectivamente; $\vec{f}$ es el vector de k funciones objetivo que se desean minimizar y que están en conflicto entre sí. Recordemos que minimizar $f_i(\vec{x})$ es equivalente a maximizar $-f_i(\vec{x})$.

En un problema de optimización multiobjetivo, las funciones objetivo están en conflicto entre sí, lo que significa que optimizar una de ellas puede resultar en un deterioro de las demás. Por lo tanto, cuando abordamos este tipo de problemas, nuestro objetivo es encontrar un conjunto de soluciones conocidas como **óptimos de Pareto**, en lugar de buscar una única solución como en la optimización mono-objetivo.

Los óptimos de Pareto representan un conjunto de soluciones que no pueden ser mejoradas en términos de todos los objetivos simultáneamente. Cada solución en este conjunto ofrece un equilibrio óptimo entre los diferentes objetivos y no se puede mejorar en uno sin empeorar en los demás.

Es importante destacar que en la optimización multiobjetivo se busca encontrar una gama de soluciones que satisfagan diferentes combinaciones de objetivos, en lugar de buscar una solución única que maximice o minimice un solo objetivo. Este enfoque más completo y holístico nos permite explorar y comprender las compensaciones y trade-offs entre los diferentes objetivos.

A continuación se enuncian algunas definiciones importantes en optimización multi-objetivo, para mas información consultar [1].

Definición 1.2.1 (Dominancia) *Un punto $\vec{x}$ domina a un punto $\vec{y}$, denotado por $\vec{x} \preceq \vec{y}$, si y sólo si $f_i(\vec{x}) \leq f_i(\vec{y})$ para toda $i = 1, \cdots, k$ y $f_j(\vec{x}) < f_j(\vec{y})$ para al menos una j.*

Definición 1.2.2 (Óptimo de Pareto) *Un punto $\vec{x}^* \in \Omega$, donde Ω es la región factible, es un óptimo de Pareto si no existe otro punto $\vec{x} \in \Omega$ tal que $\vec{x} \preceq \vec{x}^*$.*

Definición 1.2.3 (Conjunto de óptimos de Pareto) *El conjunto de óptimos de Pareto se denota como $\mathcal{PS}$: $\mathcal{PS} = \{\vec{x} \mid \vec{x}$ es un óptimo de Pareto$\}$.*

Definición 1.2.4 (Frente de Pareto) *La imagen del conjunto de óptimos de Pareto se conoce como frente de Pareto $\mathcal{F}$: $\mathcal{F} = \{F(\vec{x}) \mid \vec{x} \in \mathcal{PS}\}$.*

Para comprender el concepto de dominancia, consideremos el siguiente ejemplo. Sea $\vec{w} = [1, 1]$, $\vec{x} = [1, 2]$, $\vec{y} = [2, 1]$ y $\vec{z} = [1, 3]$, decimos que:

- $\vec{x}$ y $\vec{y}$ son no dominadas entre sí porque $\vec{x}$ es mejor que $\vec{y}$ en la primera componente pero es peor en la segunda componente.
- $\vec{x}$ domina a $\vec{z}$ porque $\vec{x}$ es igual a $\vec{z}$ en la primera componente pero es mejor en la segunda componente.
- $\vec{x}$, $\vec{y}$ y $\vec{z}$ son dominadas por $\vec{w}$ porque son iguales en una de las componentes pero peor en la restante.

1.3. Recocido Simulado

El algoritmo de Recocido Simulado (RS) es un proceso estocástico discreto que se basa en la probabilidad de pasar de una solución a otra. Esta probabilidad depende únicamente de la solución inmediata anterior y del tiempo. Supongamos que la solución actual es x. RS define un vecindario para x, representado por $N(x)$, y elige una solución y del vecindario. La probabilidad de pasar de la solución x a la solución y es igual a 1 si $f(y) \leq f(x)$. En caso contrario, la probabilidad se calcula utilizando la fórmula $e^{\frac{-(f(y)-f(x))}{T(t)}}$.

Aquí, f es la función de costo que se busca minimizar, y $T(t)$ representa la temperatura en el tiempo t.

Al comienzo de la búsqueda, la temperatura es alta y, por lo tanto, la probabilidad de aceptar soluciones peores es alta. Sin embargo, a medida que pasa el tiempo, la temperatura disminuye, lo que reduce la probabilidad de aceptar soluciones peores. Ver Algoritmo 1.

Es importante destacar que el algoritmo de metrópolis [2] es considerado el precursor de los métodos de Recocido Simulado.

Algoritmo 1: Recocido Simulado

Entrada: Temperatura inicial T_0, temperatura final T_f, función de cambio de temperatura T, función a minimizar f, y parámetros de entrada para el problema que se quiere resolver.

Salida : Mejor solución encontrada x_{best}.

Generar solución inicial x;
Establecer la temperatura actual: $t \leftarrow T_0$;
Actualizar la información de la mejor solución encontrada: $x_{best} \leftarrow x$;
while $t \geq T_f$ **do**
    Elegir aleatoriamente una solución y del vecindario, $y \in N(x)$;
    if $f(y) \leq f(x_{best})$ **then**
        $x_{best} = y$;
    end
    if $f(y) \leq f(x)$ *or* $Random(0,1) < e^{\frac{-(f(y)-f(x))}{t}}$ **then**
        $x \leftarrow y$;
    end
    Actualizar temperatura: $t \leftarrow T(t)$);
end
return x_{best}, $f(x_{best})$;

1.4. Algoritmos Evolutivos

Los Algoritmos Evolutivos (AEs) son métodos inspirados en el proceso de evolución de los seres vivos en la naturaleza. En dicho proceso, los individuos compiten por los recursos limitados, y aquellos que son más fuertes tienen mayores probabilidades de sobrevivir y reproducirse, transmitiendo así su información genética. Del mismo modo, los AEs trabajan con poblaciones de soluciones potenciales a un problema de optimización.

Estas soluciones son manipuladas utilizando operadores genéticos, como la cruza y la mutación, que simulan los procesos de recombinación y mutación genética en organismos biológicos. Cada solución se considera un individuo y se le asigna un valor de aptitud, que refleja qué tan buena es en la resolución del problema de optimización planteado. Los individuos con un valor de aptitud más alto tienen mayores posibilidades de reproducirse y sobrevivir en la población, mientras que aquellos con valores bajos tienen menos probabilidades.

El Algoritmo 2 muestra el procedimiento general de un algoritmo evolutivo, el cual consiste en la repetición de ciclos de selección, reproducción y reemplazo en la población de soluciones. De esta manera, los AEs buscan encontrar la mejor solución posible al problema de optimización, a medida que la población evoluciona y se adapta a lo largo de su proceso evolutivo.

Algoritmo 2: Algoritmo evolutivo

Entrada: Número máximo de generaciones G, tamaño de la población μ, probabilidad de cruza p_c, probabilidad de mutación p_m, función a minimizar f, y parámetros de entrada para el problema que se quiere resolver.

Salida : Mejor solución encontrada.

```
1  pob_actual ← Población inicial de tamaño μ;
2  Asignar aptitud a cada individuo de pob_actual considerando f;
3  t ← 1;
4  while t <= G do
5  |  padres ← Utilizar un método de selección para determinar los μ
   |  individuos, de pob_actual, que actuarán como padres;
6  |  hijos ← ∅;
7  |  foreach i, j ∈ padres do
8  |  |  if random(0,1) < p_c then
9  |  |  |  Aplicar operador de cruza a i y j y generar dos hijos h_1 y h_2;
10 |  |  else
11 |  |  |  Generar dos hijos h_1 y h_2 de manera asexual: h_1 ← i, h_2 ← j;
12 |  |  end
13 |  |  Aplicar operador de mutación a h_1 y h_2, usando p_m;
14 |  |  hijos ← hijos ∪{h_1, h_2};
15 |  end
16 |  pob_actual ← Seleccionar los μ individuos sobrevivientes;
17 |  t ← t + 1;
18 end
19 x_best ← Mejor individuo de la población actual;
20 Regresar x_best, f_best;
```

Capítulo 2

Logística de entregas

2.1. Descripción del problema

Una empresa mexicana requiere una aplicación que le permita diseñar la logística de entrega de sus productos en todo el país. Cada estado mexicano cuenta con miles de puntos de venta por lo que es necesario dividir cada estado en K zonas geográficas. Cada zona tiene asignada una persona conductora, un camión de carga y una bodega. Las y los conductores deben visitar todos los puntos de venta de su zona, entregar los productos y regresar a la bodega. Cada zona es cubierta en una jornada laboral por lo que el número de puntos de venta que tiene asociados no puede ser muy grande. La ruta que siguen las y los conductores se debe diseñar de tal forma que minimice el tiempo total de traslado. Los objetivos de la aplicación son los siguientes:

1. **Diseñar zonas que no se traslapen geográficamente.** Cada zona es asociada a una área geográfica y no puede haber áreas traslapadas.

2. **Minimizar el tiempo de traslado requerido para cubrir cada zona.** Es decir, que el tiempo requerido para visitar todos los puntos de venta de una zona sea mínimo.

3. **Diseñar zonas que tengan asociadas jornadas de trabajo similares.** Es decir, que el tiempo requerido por las y los conductores para entregar todos los productos de su zona sea similar. Dicho tiempo contempla tanto el tiempo de traslado como el tiempo de descarga de los productos.

2.1.1. Parámetros de entrada

$Q = \{q_i\}$ Conjunto de pares ordenados con la ubicación geográfica de cada punto de venta q_i, donde $q_i = (longitud_i, latitud_i)$

K Número de conductores/camiones disponibles

$t_{i,j}$ Tiempo de traslado del punto de venta q_i al punto de venta q_j

a_i Tiempo de atención en el punto de venta q_i

2.1.2. Variables

$C_k = \{q_i\}$ Conjunto de puntos de venta, q_i, en la zona k

$C = \{C_k\}$ Conjunto de zonas geográficas

$$z_{i,j,k} = \begin{cases} 1, & \text{si se visita el punto } q_i \text{ y enseguida el punto } q_j \text{ en la zona } k \\ 0, & \text{en otro caso} \end{cases}$$

$$y_{i,k} = \begin{cases} 1, & \text{si se visita el punto } q_i \text{ en la zona } k \\ 0, & \text{en otro caso} \end{cases}$$

2.1.3. Ejercicios

1. Generar una instancia, **LE-instancia-1** del problema:

   a) Generar un conjunto con al menos 500 puntos de venta. Para ello se debe obtener el conjunto de datos correspondiente a las actividades económicas del estado de Michoacán reportadas en el Instituto Nacional de Estadística y Geografía. Dicho conjunto está disponible en: `https://www.inegi.org.mx/app/descarga/`. Los atributos con los que se va a trabajar son: id, latitud y longitud.

   b) Graficar el conjunto de puntos de venta generados, utilizando la latitud y longitud de cada punto.

   c) Agregar un atributo llamado **tiempo_servicio**. Este atributo indicará el tiempo estimado en segundos, que está parado el vehículo para realizar la descarga de productos en dicho punto de venta. Para esta instancia se van a generar números enteros aleatorios entre 600 y 1800.

   d) Construir una matriz que aproxime el tiempo de traslado (segundos) entre puntos de venta. Para ello utilice la distancia **haversine** que considera superficies esféricas y asuma que los vehículos van a una velocidad de 50 km/h.

2. Generar una instancia, **LE-instancia-2** del problema:

   a) Generar un conjunto con al menos 2000 puntos de venta. Para ello se debe obtener el conjunto de datos correspondiente a las actividades económicas del estado de Michoacán reportadas en el Instituto Nacional de Estadística y Geografía. Dicho conjunto está disponible en: `https://www.inegi.org.mx/app/descarga/`. Los
   atributos con los que se va a trabajar son: id, latitud y longitud.

   b) Graficar el conjunto de puntos de venta generados, utilizando la latitud y longitud de cada punto.

   c) Agregar un atributo llamado **tiempo_servicio**. Este atributo indicará el tiempo estimado, en segundos, que está parado el vehículo para realizar la descarga de productos en dicho punto de venta. Para esta instancia se van a generar números enteros aleatorios entre 1800 y 3600.

   d) Construir una matriz que aproxime el tiempo de traslado (segundos) entre puntos de venta. Para ello utilice la distancia **haversine** que considera superficies esféricas y asuma que los vehículos van a una velocidad de 50 km/h.

2.2. Definición del problema de optimización

Para abordar este problema, vamos a dividirlo en dos partes: (i) definir zonas que no se traslapen geográficamente y (ii) diseñar zonas con jornadas de trabajo similares.

Parte I

En la primera parte, vamos a adoptar la idea utilizada en una técnica clásica de agrupamiento conocida como *K-means*: Sean $\mu_1, \cdots, \mu_K$ K centroides, definimos los grupos de la siguiente forma:

$$min f_1(Q, K) = \sum_{k=1}^{K} \sum_{q_i \in Q} y_{i,k} \cdot ||q_i - \mu_k|| \tag{2.1}$$

Por lo tanto, $y_{i,k} = 1$ si el punto q_i tiene como centroide más cercano a μ_k; y $y_{i,k} = 0$, en otro caso. Finalmente, definimos cada zona geográfica C_k como sigue:

$$C_k = \{q_i\} \mid y_{i,k} = 1 \tag{2.2}$$

Parte II

Sea T la matriz que almacena los tiempos de traslado entre puntos de venta, definimos la función que calcula el tiempo total de traslado en una zona C_k como sigue:

$$f_2(C_k, z_{i,j,k}) = \sum_{\forall q_i, q_j \in C_k} t_{i,j} \cdot z_{i,j,k} \tag{2.3}$$

donde $t_{i,j} = T[i,j]$ y $z_{i,j,k} = 1$ si cuando recorremos la zona k pasamos primero al punto q_i e inmediatamente al punto q_j. Por otro lado, definimos la función que calcula el tiempo total de servicio en una zona C_k de la siguiente forma:

$$f_3(C_k) = \sum_{\forall q_i \in C_k} a_i \tag{2.4}$$

Una vez definidas f_1 y f_2, podemos calcular el costo (tiempo de entrega) de una zona C_k como sigue:

$$Cost(C_k, z_{i,j,k}) = f_3(C_k) + \min_{z_{i,j,k}} f_2(C_k, z_{i,j,k}) \tag{2.5}$$

Función principal

Finalmente, definimos la función que evalúa qué tan diferentes son los costos de las diferentes zonas geográficas de la siguiente forma:

$$f(C, K) = \sqrt{\frac{1}{N} \sum_{k=1}^{K} (Cost(C_k, z_{i,j,k}) - \mu)^2} \tag{2.6}$$

donde $\mu = \frac{1}{N} \sum_{k=1}^{K} Cost(C_k)$.

2.2.1. Optimización mono-objetivo

Como podemos observar en la sección anterior, el proceso de optimización para generar las zonas geográficas (problema de agrupamiento) se debe realizar antes del proceso de optimización para diseñar las rutas dentro de las mismas (problema del agente viajero). Por otro lado, la función f se puede evaluar hasta que se tengan definidas las zonas geográficas y sus rutas. Por lo anterior, el problema no se puede ver como un problema de optimización multi-objetivo pues se realizan procesos de optimización independientes. A continuación se define el problema de optimización principal:

$$\text{mín } f(C, K) \tag{2.7}$$

2.3. Recocido Simulado

Representación de una solución

Como podemos observar en la definición del problema, nuestra tarea es definir grupos (zonas geográficas). Dado que lo vamos a hacer a partir de centroides, nuestra solución será un arreglo de K centroides. Cada uno de ellos tendrá dos componentes: latitud y longitud. Una vez definidos los centroides se podrán definir las zonas geográficas, colocando una etiqueta a cada punto de venta. Si $K = 3$ y los puntos están en el intervalo $[0, 1]$, un ejemplo de solución es:

$$x = [(0.23, 0.35), (0.5, 0.67), (0.98, 0.2)]$$

Solución inicial

La solución inicial serán K posiciones aleatorias (latitud, longitud), las cuales se generarán a partir de las latitudes y longitudes de los puntos de venta. Si $K = 10$ y usamos el conjunto de datos **LE-instancia-1**, una posible solución inicial es:

$$x = \begin{bmatrix} (19.99123175, -102.26229657), \\ (19.15614173, -102.53844539), \\ (20.05527029, -102.72145232), \\ (19.42908538, -102.03153402), \\ (19.77210272, -100.22789034), \\ (19.84984083, -102.06591731), \\ (19.69645406, -100.56077976), \\ (19.64840776, -101.24015918), \\ (19.05894359, -102.3305985), \\ (19.02041432, -102.10128631) \end{bmatrix}$$

Una vez que se tienen definidos los centroides se puede realizar el agrupamiento. A cada punto de venta se le pone la etiqueta del centroide que está más cercano a él. A continuación se muestra la etiqueta de los primeros cinco puntos de venta:

id	**latitud**	**longitud**	**tiempo_servicio**	**etiqueta**
0	19.452952	-101.732303	1788	3
1	20.272433	-102.561430	1723	2
2	19.983756	-101.761406	1161	5
3	20.049552	-102.729363	762	2
4	19.855134	-102.054702	1309	5

Función objetivo

Para hacer el cálculo de la función objetivo, se realizará lo siguiente en cada zona geográfica:

1. Definir el orden en el que se van a visitar los puntos de venta, de tal forma que el costo de traslado se minimice. Para ello nos vamos a auxiliar del módulo **ortools** utilizando una búsqueda local guiada.

2. Dado el orden de visita calcular el tiempo de traslado total para visitar todos los puntos de venta.

3. Calcular el tiempo de servicio total, considerando todos los puntos de venta de la zona.

4. Calcular el costo total de la zona como la suma del tiempo de traslado total y el tiempo de servicio total.

Finalmente, se va a calcular la desviación estándar del costo total de las zonas geográficas. Considerando la solución inicial creada en la sección anterior, tenemos los siguientes costos totales por zona:

zona	**tiempo_total_traslado**	**tiempo_total_servicio**	**tiempo_total**
0	15123	74735	89858
1	17823	35884	53707
2	10279	31793	42072
3	16525	88110	104635
4	6359	15984	22343
5	13051	56606	69657
6	21980	71785	93765
7	33554	171131	204685
8	1005	21614	22619
9	18846	33916	52762

Por lo tanto, el valor de la función objetivo es:

$$f(x) = 53585.69804434761$$

Vecindario

El vecindario de una solución consiste en un conjunto de soluciones que se pueden generar a partir de pequeños cambios a la solución actual. Para este ejemplo vamos a construir una solución vecina de dos formas posibles.

1. Si la relación entre la zona con menor costo y la de mayor es mayor a 1.5, entonces se elimina el centroide de la zona menos costosa y se genera un centroide aleatorio en la región de la zona más costosa. Siguiendo el ejemplo de la solución inicial, vemos que la zona 4 es la que tiene el menor costo y la zona 7 tiene el mayor costo. Dado que $1.5(22343) < 203644$, el centroide de la zona 4 será sustituido por la ubicación de un

punto de venta aleatorio en la zona 7.

$$y = \begin{bmatrix} (19.99123175, -102.26229657), \\ (19.15614173, -102.53844539), \\ (20.05527029, -102.72145232), \\ (19.42908538, -102.03153402), \\ (19.70538598, -101.19809768), \\ (19.84984083, -102.06591731), \\ (19.69645406, -100.56077976), \\ (19.64840776, -101.24015918), \\ (19.05894359, -102.3305985), \\ (19.02041432, -102.10128631) \end{bmatrix}$$

2. De lo contrario, se hace un pequeño cambio en un centroide aleatorio. Para esto vamos a generar un vector de tamaño dos con valores aleatorios utilizando una distribución normal con media 0 y desviación estándar igual a 0.005. Sea $pos = 8$ la posición del centroide aleatorio y $[-0.00024049, -0.00789772]$, tenemos que:

$$y = \begin{bmatrix} (19.99123175, -102.26229657), \\ (19.15614173, -102.53844539), \\ (20.05527029, -102.72145232), \\ (19.42908538, -102.03153402), \\ (19.77210272, -100.22789034), \\ (19.84984083, -102.06591731), \\ (19.69645406, -100.56077976), \\ (19.64840776, -101.24015918), \\ (19.05894359, -102.3305985), \\ (19.02041432, -102.10128631) \end{bmatrix} + \begin{bmatrix} (0,0), \\ (0,0), \\ (0,0), \\ (0,0), \\ (0,0), \\ (0,0), \\ (0,0), \\ (0,0), \\ (-0.00024049, -0.00789772), \\ (0,0) \end{bmatrix}$$

$$y = \begin{bmatrix} (19.99123175, -102.26229657), \\ (19.15614173, -102.53844539), \\ (20.05527029, -102.72145232), \\ (19.42908538, -102.03153402), \\ (19.77210272, -100.22789034), \\ (19.84984083, -102.06591731), \\ (19.69645406, -100.56077976), \\ (19.64840776, -101.24015918), \\ (19.0587031, -102.33849622), \\ (19.02041432, -102.10128631) \end{bmatrix}$$

Función de cambio en la temperatura

En este caso vamos a disminuir la temperatura usando la siguiente función lineal:

$$T(t) = 0.9 \cdot t \tag{2.8}$$

2.3.1. Práctica I

1. Implementar la propuesta de RS descrita en la Sección 2.3 y utilizarla para resolver **LE-instancia-1** (ver Sección 2.1).

2. Dado que RS es un algoritmo estocástico, ejecutarlo M veces utilizando los mismos parámetros. Reportar la mejor solución, la peor solución y la solución en la mediana de acuerdo al valor de la función objetivo. Así como el valor de la función objetivo promedio y su desviación estándar.

3. Ejecutar RS $M = 21$ veces utilizando una temperatura inicial igual a 100 y una temperatura final igual a 0.01. Se deben reportar las estadísticas solicitadas en el punto anterior.

4. Graficar las zonas geográficas creadas a partir de la mejor solución encontrada en el punto anterior.

5. Utilizar el diseño de RS propuesto en la Sección 2.3 para resolver la instancia **LE-instancia-2** (ver Sección 2.1). Se debe hacer un estudio estadístico y graficar las zonas geográficas creadas a partir de la mejor solución encontrada.

6. Proponer dos funciones para el cambio de temperatura y utilizarlas en el algoritmo de RS, propuesto en la Sección 2.3, para resolver la instancia **LE-instancia-2**. Hacer un estudio estadístico de cada versión y presentar una tabla comparativa considerando las tres versiones (versión de la Sección 2.3 y las dos versiones nuevas). Finalmente graficar las zonas geográficas creadas a partir de la mejor solución encontrada.

7. Proponer un algoritmo distinto para la generación de una solución vecina y utilizarlo en el algoritmo de RS, propuesto en la

Sección 2.3, para resolver la instancia **LE-instancia-2**. Hacer un estudio estadístico y presentar una tabla comparativa considerando las dos versiones (versión de la Sección 2.3 y versión nueva). Finalmente graficar las zonas geográficas creadas a partir de la mejor solución encontrada.

Capítulo 3

Vehículo compartido

3.1. Descripción del problema

El gobierno de una ciudad quiere implementar un sistema para promover el "**car pooling**" con la finalidad de disminuir los gastos de transporte de la ciudadanía y disminuir la huella de carbono. La aplicación utilizará un grafo dirigido que representa la infraestructura vial de la ciudad. Cada nodo de este grafo representa el punto de intersección de dos o más calles, una arista dirigida (a, b) significa que hay una calle con dirección de a a b. El sistema contempla dos tipos de personas: A y B. Las personas tipo A son aquellas que requieren un transporte y deben indicar su nodo origen y su nodo destino. Mientras que las personas tipo B son aquellas que desean compartir su vehículo y deben indicar su nodo origen, su nodo destino y el número de lugares disponibles en su vehículo. La aplicación deberá determinar los grupos de personas que utilizarán un mismo vehículo, la ruta que seguirá dicho vehículo, así como los puntos de subida y bajada de las personas. Para simplificar el problema, la ruta que seguirá el vehículo será la ruta más corta desde el punto de origen de la persona B, dueña del vehículo, hasta el punto final. Los objetivos de la aplicación son:

1. **Minimizar la suma de las distancias que tienen que recorrer las personas tipo** A.

    - La distancia recorrida por una *persona tipo A asignada a un vehículo* contempla la distancia recorrida para abordar el vehículo

en algún nodo de su ruta, la distancia desde el nodo de bajada hasta el destino final y la distancia recorrida en el vehículo.

- La distancia que recorre una *persona tipo A que no tiene asignado un vehículo* es igual a la distancia de la ruta más corta entre su punto de origen y su punto destino.

2. **Minimizar el número de personas que no tienen asignado un grupo**.

Dado que la velocidad promedio de una persona caminando es de aproximadamente 4 km/h y la velocidad promedio de un automóvil en una ciudad como la Ciudad de México es de 14 km/h[1], las distancias que recorran las personas sin el uso de un automóvil serán ponderadas por un factor igual a 3.5.

3.1.1. Parámetros de entrada

$G = (V, E)$	Grafo vial
K_A	Número de personas tipo A
K_B	Número de personas tipo B
$A = \{a_i\}$	Conjunto de pares ordenados $a_i = (o_{a_i}, d_{a_i})$ que representan el nodo origen y nodo destino de las personas tipo A respectivamente, donde $0 \leq i < K_A$
$B = \{b_i\}$	Conjunto de 3-tuplas $b_i = (o_{b_i}, d_{b_i}, c_{b_i})$ que representan el nodo origen, el nodo destino y la capacidad de las personas tipo B respectivamente, donde $0 \leq i < K_B$

[1]CDMX es la cuarta ciudad más lenta del mundo y la primera a nivel nacional, El Universal, Septiembre 27, 2019, URL: https://www.eluniversal.com.mx/autopistas/cdmx-es-la-cuarta-ciudad-mas-lenta-del-mundo-y-la-primera-nivel-nacional/

3.1.2. Variables

$G_A = \{g_{a_i}\}$ Etiquetas de los grupos asignados a las personas tipo A, g_{a_i} es el grupo asignado a la persona i de tipo A. $g_{a_i} = -1$ significa que la persona a_i no tiene asignado un grupo

$G_B = \{g_{b_i}\}$ Etiquetas de los grupos asignados a las personas tipo B, g_{b_i} es el grupo asignado a la persona i de tipo B

C_k Conjunto de personas en el grupo k, con $-1 \leq k < K_B$. $C_k = \{a_i \mid g_{a_i} = k\} \cup \{b_i \mid g_{b_i} = k\}$

$C = \{C_k\}$ Conjunto de grupos, con $-1 \leq k < K_B$

3.1.3. Ejercicios

1. Generar una instancia, **VC-instancia-1**, del problema:

   a) Generar un grafo $G = (V, E)$ con al menos 100 vértices.

   b) Generar $K_A = 10$ personas tipo A.

   c) Generar $K_B = 3$ personas tipo B, las capacidades de los vehículos estarán entre 1 y 4.

   d) Graficar G junto con las rutas que seguirán los usuarios tipo B.

2. Generar una instancia, **VC-instancia-2**, del problema:

   a) Generar un grafo $G = (V, E)$ con al menos 200 vértices.

   b) Generar $K_A = 100$ personas tipo A.

   c) Generar $K_B = 20$ personas tipo B, las capacidades de los vehículos estarán entre 1 y 4.

   d) Graficar G junto con las rutas que seguirán los usuarios tipo B.

3.2. Definición del problema de optimización

Sea $route(v_0, v_f) = [v_0, v_1, \cdots, v_f] \mid v_i \in V$ la ruta más corta para ir de un nodo origen v_0 a un nodo destino v_f, $dist(v_0, v_f)$ la distancia de la ruta más corta entre el nodo v_0 y el nodo v_f, $nn_1(v, r)$ el nodo más cercano de la ruta r al nodo origen v, $nn_2(r, v)$ el nodo más cercano de la ruta r al nodo destino

v; definimos la función para calcular la suma de las distancias recorridas por las personas tipo A de la siguiente forma:

$$f_1(G_A, G_B) = \sum_{g_{a_i} \in G_A, g_{a_i} \neq -1} (3.5 \cdot (dist(o_{a_i}, u_1) + dist(u_2, d_{a_i})) + dist(u_1, u_2)) + \sum_{g_{a_i} \in G_A, g_{a_i} = -1} 3.5 \cdot dist(o_{a_i}, d_{a_i}) \tag{3.1}$$

donde $u_1 = nn_1(o_{a_i}, r_{b_j})$, $u_2 = nn_2(r_{b_j}, d_{a_i})$, $r_{b_j} = route(o_{b_j}, d_{b_j})$ y $g_{b_j} = g_{a_i}$. Por otro lado, definimos la cantidad de personas tipo A que no tienen asignado un vehículo como sigue:

$$f_2(G_A) = \sum_{g_{a_i} \in G_A, g_{a_i} = -1} 1 \tag{3.2}$$

3.2.1. Optimización mono-objetivo

Nuestro objetivo es encontrar una manera eficiente para que las personas tipo A se transporten, sin importar si están asignadas a un vehículo o no. Por lo anterior, definimos el problema de optimización como sigue:

$$\text{mín } f_1(G_A, G_B) \tag{3.3}$$

tal que para cada $b_j \in B$ se cumpla:

$$g(G_A, G_B) = \sum_{g_{a_i} \in G_A, g_{a_i} = g_{b_j}} 1 \quad \leq c_{b_j} \tag{3.4}$$

3.2.2. Optimización multi-objetivo

A diferencia del caso mono-objetivo, en este caso vamos a considerar que las personas tipo A pueden preferir transportarse en vehículo aunque ello implique que su camino no sea el más eficiente. Por lo anterior, definimos el problema de optimización como sigue:

$$\text{mín } F(G_A, G_B) = [f_1(G_A, G_B), f_2(G_A)]^T \tag{3.5}$$

tal que para cada $b_j \in B$ se cumpla:

$$g(G_A, G_B) = \sum_{g_{a_i} \in G_A, g_{a_i} = g_{b_j}} 1 \quad \leq c_{b_j} \tag{3.6}$$

3.3. Recocido Simulado: Caso mono-objetivo

Representación de una solución

Como podemos observar en la definición del problema, nuestra tarea es asignar una etiqueta de grupo a cada persona (A ó B). Dado que las personas tipo B son aquellas que pueden prestar un vehículo para formar un grupo, tomaremos su índice como la etiqueta de su grupo y dicha etiqueta podrá ser asignada a las personas tipo A. Cada índice i tendrá c_{b_i} copias, es decir, la capacidad del vehículo de la persona b_i. En caso de que haya más personas A que etiquetas disponibles, rellenaremos el resto con etiquetas -1. Por lo anterior, veremos una solución como un arreglo con las etiquetas de las personas tipo A. Si tenemos diez personas tipo A y tres personas tipo B con capacidades de 1, 2 y 1 respectivamente, un ejemplo de solución es:

$$G_A = [0, 1, 1, 2, -1, -1, -1, -1, -1, -1]$$

La solución anterior indica que la persona a_0 está asignada al vehículo de la persona b_0, mientras que las personas a_1 y a_2 están asignadas al vehículo de la persona b_1 y, la persona a_3 está asignada al vehículo del usuario b_2. Finalmente, las personas a_4, a_5, a_6, a_7, a_8 y a_9 no tienen asignado un vehículo. Las soluciones restantes al problema las podemos generar haciendo permutaciones de las posiciones de G_A, por ejemplo:

$$G_{A_{new}} = [-1, 1, -1, -1, 1, -1, -1, -1, 0, 2]$$

Dado que el problema indica que no es necesario llenar los vehículos cuando es menos costoso para las personas A trasportarse de su origen a su destino que transportarse a los nodos de las rutas que siguen las personas tipo B, agregaremos valores con -1 a nuestra solución. En este caso agregaremos el mismo número de -1 que personas de tipo A:

$$G_{A_{ext}} = [0, 1, 1, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]$$

Cuando se evalúa una solución en la función objetivo solo se van a considerar las primeras K_A posiciones que son las que indican las etiquetas de las personas tipo A. Por lo tanto, la solución $G_{A_{ext}}$ es equivalente a la solución G_A.

Solución inicial

La solución inicial será una permutación aleatoria del arreglo $G_{A_{ext}}$, por ejemplo:

$$G_{A_0} = [-1, -1, -1, -1, 1, -1, -1, -1, 1, 0, -1, -1, -1, 2]$$

La solución G_{A_0} indica que las personas a_0, a_1, a_2, a_3, a_5, a_6, a_7 no tienen asignado un vehículo, mientras que las persona a_4 y a_8 están asignadas al vehículo de la persona b_1 y, la persona a_9 está asignada el vehículo de la persona b_0.

Vecindario

El vecindario de una solución es un conjunto de soluciones que se pueden generar realizando pequeños cambios a la solución actual. Para este ejemplo, vamos a construir una solución vecina realizando n intercambios entre las posiciones del arreglo. Donde n es un parámetro que indica el usuario. Si definimos $n = 1$, una solución vecina de $G_{A_{ext}}$ puede ser:

$$G_{A_{new}} = [0, 1, 1, -1, -1, -1, -1, 2, -1, -1, -1, -1, -1, -1]$$

Función de cambio en la temperatura

En este caso vamos a disminuir la temperatura usando la siguiente función lineal:

$$T(t) = 0.9 \cdot t \tag{3.7}$$

3.3.1. Práctica I

1. Implementar la propuesta de RS descrita en la Sección 3.3 y utilizarla para resolver **VC-instancia-1** (ver Sección 3.1).

2. Dado que RS es un algoritmo estocástico, ejecutarlo M veces utilizando los mismos parámetros. Reportar la mejor solución, la peor solución y la solución en la mediana de acuerdo al valor de la función objetivo. Así como el valor de la función objetivo promedio y su desviación estándar.

3. Ejecutar RS utilizando uno, dos y tres intercambios para generar al vecino, una temperatura inicial igual a 1000 y una temperatura final igual a 0.001. Cada versión se debe ejecutar $M = 100$ veces y se deben reportar las estadísticas solicitadas en el punto anterior.

4. Graficar los grupos creados a partir de la mejor solución encontrada en el punto anterior. Para ello se deben graficar las rutas de cada persona tipo A y cada persona tipo B que pertenezcan a un mismo grupo con un mismo color. La ruta que se grafica es la ruta más corta entre los nodos iniciales y finales de cada persona.

5. Graficar por separado las rutas pertenecientes a las personas de cada grupo.

3.4. Algoritmo Evolutivo: Caso mono-objetivo

Representación de las soluciones

Debido a cómo trabajan los operadores de cruza y mutación para permutaciones en los AEs, veremos una solución como una permutación de los índices de una solución base. Sea $G_{A_{base}}$ nuestra solución base y x_{base} sus índices:

$$\begin{aligned} G_{A_{base}} &= [0, 1, 1, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1] \\ x_{base} &= [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]. \end{aligned}$$

Creamos una solución x generando una permutación de x_{base}, por ejemplo:

$$x = [8, 6, 4, 11, 2, 13, 9, 1, 7, 10, 3, 0, 5, 12]$$

Las etiquetas correspondientes a x quedan de la siguiente forma:

$$G_{A_x} = [-1, -1, -1, -1, 1, -1, -1, 1, -1, -1]$$

Población inicial

La población inicial se creará generando n permutaciones de la solución base x_{base}. Sea $n = 3$, una posible población de soluciones es:

$$x_1 = [11, 1, 7, 13, 2, 12, 6, 5, 10, 0, 3, 4, 9, 8]$$
$$x_2 = [4, 6, 2, 12, 7, 0, 3, 10, 1, 11, 5, 9, 8, 13]$$
$$x_3 = [9, 12, 3, 6, 1, 11, 8, 0, 7, 5, 4, 13, 10, 2]$$

y sus etiquetas correspondientes quedan como sigue:

$$G_{A_{x_1}} = [-1, 1, -1, -1, 1, -1, -1, -1, -1, 0]$$
$$G_{A_{x_2}} = [-1, -1, 1, -1, -1, 0, 2, -1, 1, -1]$$
$$G_{A_{x_3}} = [-1, -1, 2, -1, 1, -1, -1, 0, -1, -1]$$

Operador de cruza

Para este ejemplo, vamos a implementar el operador de cruza llamado **Position-based Crossover**. Este método recibe como parámetros dos permutaciones padre, p_1 y p_2, y crea una permutación hija, h_1. Para esto, se eligen n posiciones aleatorias del padre p_1 y se copian al hijo h_1, el resto de las posiciones se rellenan con los elementos del padre p_2 de izquierda a derecha, sin considerar los que se copiaron del padre p_1. Si consideramos las siguientes soluciones padre:

$$p_1 = [8, 6, 4, 11, 2, 13, 9, 1, 7, 10, 3, 0, 5, 12]$$
$$p_2 = [13, 4, 11, 2, 0, 8, 10, 12, 1, 9, 6, 3, 5, 7]$$

sea $n = 8$ y $random\_pos = [0, 6, 4, 2, 5, 3, 11, 9]$ las posiciones aleatorias, obtenemos la siguiente solución hija:

$$h_1 = [8, 12, 4, 11, 2, 13, 9, 1, 6, 10, 3, 0, 5, 7]$$

Position-based Crossover puede generar un segundo hijo, h_2, utilizando los mismos padres, p_1 y p_2. Para ello, se repite el proceso pero invirtiendo los padres. Es decir, se copian las posiciones aleatorias desde el padre p_2 y el resto se rellenan con los elementos del padre p_1. Siguiendo el ejemplo anterior, el segundo hjo queda como sigue:

$$h_2 = [13, 6, 11, 2, 0, 8, 10, 4, 1, 9, 7, 3, 5, 12]$$

Operador de mutación

La mutación de una permutación la haremos de manera similar a la generación de un vecino en RS. Es decir, vamos a realizar n intercambios entre las posiciones del arreglo. Donde n es un parámetro que indica el usuario.

Selección de padres y sobrevivientes

Para este problema vamos a elegir las soluciones padres de manera aleatoria y los sobrevivientes se elegirán haciendo una selección +. Es decir, se unirán las poblaciones de padres e hijos y se seleccionarán a las mejores soluciones para pasar a la siguiente iteración.

3.4.1. Práctica II

1. Implementar la propuesta de AE descrita en la Sección 3.4 y utilizarla para resolver la instancia **VC-instancia-1** (ver Sección 3.1).

2. Dado que los AEs son algoritmos estocásticos, ejecutar M veces la propuesta utilizando los mismos parámetros. Reportar la mejor solución, la peor solución y la solución en la mediana de acuerdo al valor de la función objetivo. Así como el valor de la función objetivo promedio y su desviación estándar.

3. Ejecutar el AE utilizando $G = 100$, $\mu = 100$, $p_c = 0.9$, $p_m = 0.1$, $\frac{K_A}{2}$ posiciones fijas para el operador de cruza y un intercambio en el operador de mutación. El AE se debe ejecutar $M = 100$ veces y se deben reportar las estadísticas solicitadas en el punto anterior.

4. Graficar los grupos creados a partir de la mejor solución encontrada en el punto anterior. Para ello se deben graficar las rutas de cada persona tipo A y cada persona tipo B que pertenezcan a un mismo grupo con un mismo color. La ruta que se grafica es la ruta más corta entre los nodos iniciales y finales de cada persona.

5. Graficar por separado las rutas pertenecientes a las personas de cada grupo.

6. Utilizar los diseños de RS y AE propuestos en las Secciones 3.3 y 3.4 para resolver **VC-instancia-2** (ver Sección 3.1). Deberá presentar una tabla comparativa de ambas metaheurísticas considerando al menos tres conjuntos de parámetros diferentes.

7. Graficar los grupos creados a partir de la mejor solución encontrada por RS.

8. Graficar los grupos creados a partir de la mejor solución encontrada por el AE.

3.5. Algoritmo Evolutivo: Caso multi-objetivo

En la actualidad, existen varios métodos que buscan el conjunto de óptimos de Pareto de un problema de optimización multi-objetivo. Dentro de ellos, los métodos basados en AEs son ampliamente utilizados por dos razones principales:

1. Su objetivo es que cada solución de la población converja a una solución óptima de Pareto. Por lo anterior, obtienen una aproximación del frente de Pareto en una sola ejecución. Esto no sucede cuando se transforma el problema multi-objetivo en problemas mono-objetivo, resolver cada problema mono-objetivo genera una única solución del conjunto de óptimos de Pareto.

2. Son capaces de lidiar con cualquier tipo de funciones objetivo, por ejemplo, funciones no continuas, no diferenciables e incluso funciones que no pueden ser expresadas de manera algebraica (simuladores, redes neuronales, etc.)

Por lo anterior, utilizaremos un AE para resolver el problema de optimización multi-objetivo. Nuestro algoritmo estará basado en el AE diseñado en la Sección 3.4. El único cambio que se implementará será en la selección de las soluciones sobrevivientes. En el caso multi-objetivo aplicaremos un método conocido como **jerarquización de Pareto**: Dado un conjunto de soluciones P se obtienen las soluciones no dominadas y se almacenan en un conjunto P_1. Se vuelve a aplicar el mismo procedimiento con las soluciones restantes de P y se almacenan en P_2. Se repite este proceso hasta que P se

queda vacío. Posteriormente, se van seleccionando las soluciones de los diferentes frentes: $P_1, P_2, \cdots, P_k$, empezando por P_1, hasta tener el número de soluciones requeridas. Si en algún momento i, se tienen más soluciones en P_i que las que deseamos seleccionar, elegiremos de manera aleatoria la cantidad deseada.

Supongamos que tenemos los siguientes vectores correspondientes a los valores objetivo de una población de cinco soluciones:

$$P = \{[249.5, 8], [279.5, 7], [284.5, 9], [233, 7], [264.5, 7]\}$$

y que queremos seleccionar dos soluciones. Aplicamos jerarquización de Pareto de la siguiente forma. Las soluciones no dominadas se almacenan en P_1 y en P se quedan las soluciones dominadas:

$$\begin{aligned} P_1 &= \{[233, 7]\} \\ P &= \{[249.5, 8], [279.5, 7], [284.5, 9], [264.5, 7]\} \end{aligned}$$

Repetimos el proceso y obtenemos P_2 con las siguientes soluciones no dominadas y P con las soluciones dominadas:

$$\begin{aligned} P_2 &= \{[249.5, 8], [264.5, 7]\} \\ P &= \{[279.5, 7], [284.5, 9]\} \end{aligned}$$

De igual forma calculamos P_3 y actualizamos P:

$$\begin{aligned} P_3 &= \{[279.5, 7]\} \\ P &= \{[284.5, 9]\} \end{aligned}$$

Finalmente, tenemos que:

$$\begin{aligned} P_4 &= \{[284.5, 9]\} \\ P &= \emptyset \end{aligned}$$

Una vez que tenemos los frentes P_i vamos seleccionando soluciones, empezando con $i = 1$, hasta obtener las dos soluciones requeridas. En este caso seleccionamos las soluciones de P_1 y elegimos una solución aleatoria de P_2.

Es importante mencionar que los algoritmos evolutivos multi-objetivo tienen dos objetivos principales:

1. **Convergencia**: Encontrar soluciones dentro del conjunto de óptimos de Pareto.

2. **Distribución**: Encontrar soluciones que estén bien distribuidas a lo largo del frente de Pareto.

En problemas con pocos objetivos, 2 ó 3, se utiliza el concepto de dominancia de Pareto para alcanzar convergencia (por ejemplo, usando jerarquización de Pareto) y algún método que garantice diversidad en las soluciones para lograr una buena distribución. Este último se aplica regularmente al seleccionar soluciones dentro de un conjunto de soluciones no dominadas. Algunos ejemplos de métodos que han sido utilizados para generar una buena distribución son: *clustering*, distancia de *crowding*, contribución a algún indicador, etc.

3.5.1. Indicadores

Existen varios indicadores que miden la calidad de una aproximación del frente de Pareto. Un ejemplo es el indicador de **hipervolumen** [4] denotado por Hv, el cual mide el espacio cubierto por las soluciones que se encuentran dentro de la aproximación del frente de Pareto. Este indicador es de los más utilizados por dos razones principales: (i) es compatible con las relaciones de dominancia de Pareto y (ii) solo requiere de la definición de un punto de referencia para su cálculo. Si $\mathcal{L}$ denota la medida de *Lebesgue*, Hv se define como:

$$Hv(A, \vec{y}_{ref}) = \mathcal{L}\left(\bigcup_{\vec{z}\in A}\{\vec{y} \mid \vec{z} < \vec{y} < \vec{y}_{ref}\}\right) \tag{3.8}$$

donde $\vec{y}_{ref}$ denota el punto de referencia, el cual debe ser dominado por todos los puntos en A. Valores más grandes en Hv significan que es una mejor aproximación del frente de Pareto.

3.5.2. Práctica III

1. Implemente una función que reciba como parámetros dos vectores con los valores objetivo correspondientes a dos soluciones, $\vec{x}$ y $\vec{y}$, y regrese -1 si son no dominadas entre sí, 1 si $\vec{x} \preceq \vec{y}$, y 0 si $\vec{y} \preceq \vec{x}$.

2. Implementar una función que reciba cómo parámetro un conjunto de soluciones y regrese el conjunto dividido en dos subconjuntos. El primer subconjunto debe contener las soluciones dominadas y el segundo las soluciones no dominadas.

3. Implementar la propuesta de AE descrita en la Sección 3.4 y utilizarla para resolver la instancia **VC-instancia-1** (ver Sección 3.1). Ejecutar el AE utilizando $G = 100$, $\mu = 100$, $p_c = 0.9$, $p_m = 0.1$, $\frac{K_A}{2}$ posiciones fijas para el operador de cruza y un intercambio en el operador de mutación.

4. Graficar la aproximación del frente de Pareto encontrado por el algoritmo implementado en el punto anterior.

5. Evaluar el frente de Pareto encontrado utilizando el indicador de hipervolumen.

6. Dado que los AEs son algoritmos estocásticos, ejecutar $M = 100$ veces la propuesta utilizando los mismos parámetros. Reportar la mejor solución, la peor solución y la solución en la mediana de acuerdo al valor del hipervolumen. Así como el valor del hipervolumen promedio y su desviación estándar.

7. Graficar los frentes de Pareto correspondientes al mejor, peor y valor en la mediana del hipervolumen.

8. Graficar los grupos creados a partir de la solución, de la aproximación del frente de Pareto, que deja la menor cantidad de usuarios tipo A sin vehículo. Para ello se deben graficar las rutas de cada persona tipo A y cada persona tipo B que pertenezcan a un mismo grupo con un mismo color. La ruta que se grafica es la ruta más corta entre los nodos iniciales y finales de cada persona.

9. Graficar por separado las rutas pertenecientes a las personas de cada grupo.

10. Utilizar el AE propuesto para resolver la instancia **VC-instancia-2** (ver Sección 3.1). Deberá presentar una tabla comparativa, utilizando el indicador de hipervolumen para evaluar las aproximaciones del frente de Pareto encontradas, y reportar al menos tres conjuntos de parámetros diferentes.

11. Graficar las agrupaciones generadas por las soluciones que puedan ser interesantes para el tomador de decisiones.

12. Combinar la técnica de jerarquización de Pareto con una técnica que mantenga diversidad en las soluciones a lo largo del frente de Pareto. Utilizar dicha combinación en el algoritmo evolutivo diseñado y usarlo para resolver la instancia **VC-instancia-2** (ver Sección 3.1). Se debe graficar el frente de Pareto obtenido y las agrupaciones generadas por las soluciones que crean pueden ser interesantes para el tomador de decisiones.

13. Utilizar una técnica de cruza diferente y resolver la instancia **VC-instancia-2** (ver Sección 3.1). Se debe graficar el frente de Pareto obtenido y las agrupaciones generadas por las soluciones que puedan ser interesantes para el tomador de decisiones.

Bibliografía

[1] Miettinen Kaisa. *Nonlinear Multiobjective Optimization*. International Series in Operations Research & Management Science. Springer New York, NY, first edition, 1998.

[2] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, and Augusta H. Teller. Equation of state calculations by fast computing machines. *Chemical Physics*, 21(6):1087–1092, 1953.

[3] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, NY, USA, second edition, 2006.

[4] Eckart Zitzler and Lothar Thiele. Multiobjective optimization using evolutionary algorithms—a comparative study. In A. E. Eiben, editor, *Parallel Problem Solving from Nature V*, pages 292–301, Amsterdam, September 1998. Springer-Verlag.

Material complementario

Índice general

Capítulo 1

Logística de entregas

1.1. Parámetros de entrada

	Descripción
$Q = \{q_i\}$	Conjunto de pares ordenados con la ubicación geográfica de cada punto de venta
K	Número de conductores/camiones disponibles
$t_{i,j}$	Tiempo de traslado del punto de venta q_i al punto de venta q_j
a_i	Tiempo de atención en el punto de venta q_i

1.2. Variables

	Descripción
$C_k = \{q_i\}$	Conjunto de puntos de venta en la zona k
$C = \{C_k\}$	Conjunto de zonas geográficas
$z_{i,j,k}$	1, si se visita el punto q_i y enseguida el punto q_j en la zona k, 0, en otro caso
$y_{i,k}$	1, si se visita el punto q_i en la zona k, 0, en otro caso

1.3. Preparación del ambiente

Las bibliotecas/módulos que se van a utilizar son: 1. numpy 2. pandas 3. matplotlib 4. scipy 4. haversine 5. ortools

Para instalarlas usamos:

```
!pip install numpy
!pip install pandas
!pip install matplotlib
!pip install scipy
!pip install haversine
!pip install ortools
```

Además utilizaremos el archivo **tsp.py** el cual utiliza el módulo **ortools** para resolver el problema del agente viajero.

[1]:
```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from scipy.spatial import ConvexHull
from haversine import haversine, Unit
from tsp import routing_google
```

Asignamos una semilla de números aleatorios para poder generar siempre la misma secuencia de números aleatorios.

[2]:
```
np.random.seed(0)
```

1.4. Obtención de datos

1.4.1. Instancia de prueba 1: LE-instancia-1

Para generar instancias del problema, vamos a utilizar el conjunto de datos 'denue_inegi_16_.csv', correspondiente a las actividades económicas del estado de Michoacán reportadas en el Instituto Nacional de Estadística y Geografía. Los atributos con los que vamos a trabajar son: 'id', 'latitud' y 'longitud' y vamos a generar una muestra aleatoria de 500 puntos de venta.

[3]:
```
n_pdvs = 500
df = pd.read_csv("denue_inegi_16_.csv", encoding='latin-1')
df_pdvs = df[["latitud", "longitud"]].sample(n_pdvs)
df_pdvs
```

```
/var/folders/g9/vn2vfpcs56x_6scmywd7vn2c0000gn/T/ipykernel_14525/11310035.
 ↪py:2:
DtypeWarning: Columns (35) have mixed types. Specify dtype option on import␣
 ↪or
set low_memory=False.
  df = pd.read_csv("denue_inegi_16_.csv", encoding='latin-1')
```

[3]:
```
          latitud    longitud
66905   19.452952 -101.732303
181612  20.272433 -102.561430
99919   19.983756 -101.761406
190241  20.049552 -102.729363
24964   19.855134 -102.054702
...           ...         ...
222508  20.055931 -102.721927
170164  19.096221 -101.522277
1702    19.689010 -100.536500
163661  19.676502 -101.226584
58235   19.706095 -101.176884

[500 rows x 2 columns]
```

Agregamos índices que correspondan con los índices en memoria.

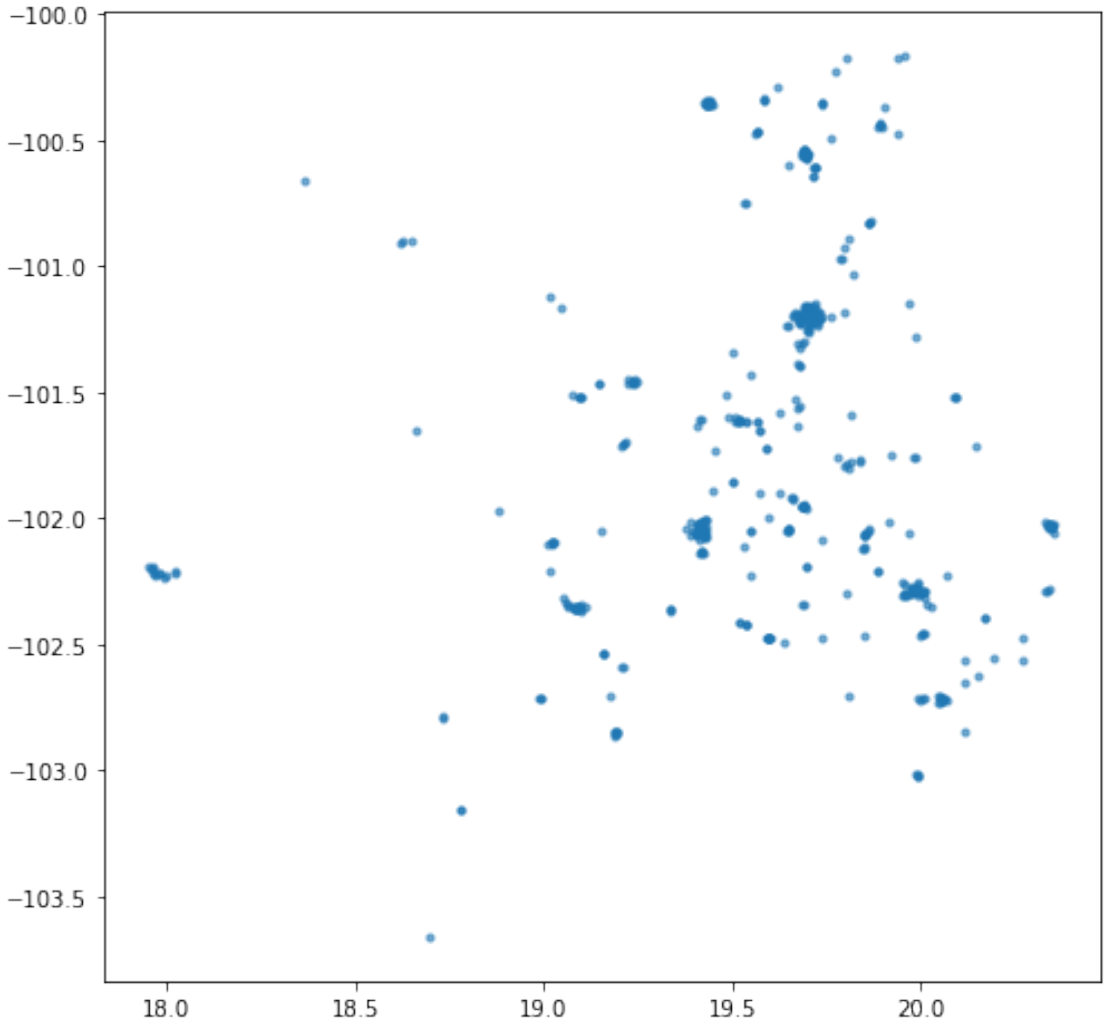
[4]:
```
indexes = [i for i in range(n_pdvs)]
df_pdvs["id"] = indexes
df_pdvs.set_index("id", inplace=True)
df_pdvs
```

[4]:
```
        latitud    longitud
id
0     19.452952 -101.732303
1     20.272433 -102.561430
2     19.983756 -101.761406
3     20.049552 -102.729363
4     19.855134 -102.054702
..         ...         ...
495   20.055931 -102.721927
496   19.096221 -101.522277
497   19.689010 -100.536500
498   19.676502 -101.226584
499   19.706095 -101.176884

[500 rows x 2 columns]
```

Creamos una función que nos permita graficar los puntos de venta de acuerdo a su latitud y longitud.

[5]:
```
def plot_pdvs(df_pdvs):
    fig, ax = plt.subplots(1, figsize=(8, 8))
    plt.scatter(df_pdvs.latitud, df_pdvs.longitud, alpha=0.6, s=10)
```

[6]:
```
plot_pdvs(df_pdvs)
```



Agregamos un atributo llamado **tiempo_servicio**, el cual indicará el tiempo estimado (segundos) que se requiere para descargar los productos en el punto de venta. Dicho atributo tendrá valores enteros aleatorios entre 600 y 1800.

[7]:
```
service_time = np.random.randint(600, 1800, size=500)
df_pdvs["tiempo_servicio"] = service_time
df_pdvs
```

[7]:

id	latitud	longitud	tiempo_servicio
0	19.452952	-101.732303	1788
1	20.272433	-102.561430	1723

```
2    19.983756 -101.761406            1161
3    20.049552 -102.729363             762
4    19.855134 -102.054702            1309
..         ...         ...             ...
495  20.055931 -102.721927             639
496  19.096221 -101.522277            1643
497  19.689010 -100.536500             748
498  19.676502 -101.226584             656
499  19.706095 -101.176884            1543

[500 rows x 3 columns]
```

Finalmente, vamos a construir la matriz que almacenará una aproximación del tiempo requerido para ir de un punto de venta a otro en segundos. Para ello vamos a utilizar la distancia **haversine** y asumiremos que los vehículos van a una velocidad de 50 km por hora.

[8]:
```
def get_time_matrix(df_pdvs, speed):
    N = df_pdvs.shape[0]
    cost_matrix = np.zeros((N,N))

    for i in range(N):
        for j in range(i+1,N):
            distance = haversine((df_pdvs.iloc[i].latitud, df_pdvs.
↪iloc[i].longitud), (df_pdvs.iloc[j].latitud, df_pdvs.iloc[j].longitud),␣
↪unit=Unit.KILOMETERS)
            cost_matrix[i,j] = cost_matrix[j,i] = 3600*distance/speed

    return pd.DataFrame(cost_matrix)
```

[9]:
```
df_T = get_time_matrix(df_pdvs, 50)
```

A continuación se muestra el costo en segundos de los primeros cuatro puntos de venta a los primeros cuatro puntos de venta.

[10]:
```
df_T.iloc[:4,:4]
```

[10]:

	0	1	2	3
0	0.000000	9056.476458	4255.299307	8902.617446
1	9056.476458	0.000000	6442.637055	2185.625356
2	4255.299307	6442.637055	0.000000	7300.397779
3	8902.617446	2185.625356	7300.397779	0.000000

Ejercicio 1: Generar un conjunto con al menos 2000 puntos de venta, utilizando del conjunto de datos correspondiente a las actividades económicas del estado de Michoacan reportadas en el Instituto Nacional de Estadística y Geografía.

Ejercicio 2: Graficar el conjunto de puntos de venta generados, utilizando la latitud y longitud de cada punto.

Ejercicio 3: Agregar el atributo llamado {**tiempo_servicio**}. **Para esta instancia se van a generar números enteros aleatorios entre** 1800 **y** 3600**.**

Ejercicio 4: Construir una matriz que aproxime el tiempo de traslado (segundos) entre puntos de venta.

1.5. Definición del problema de optimización

Para abordar este problema, vamos a dividirlo en dos partes: (i) definir zonas que no se traslapen geográficamente y (ii) diseñar zonas con jornadas de trabajo similares.

1.5.1. Parte I

En la primera parte, vamos a adoptar la idea utilizada en *K-means*: Sean $\mu_1, \cdots, \mu_K$ K **centroides, definimos los grupos de la siguiente forma:**

$$min f_1(Q, K) = \sum_{k=1}^{K} \sum_{q_i \in Q} y_{i,k} \cdot ||q_i - \mu_k|| \tag{1.1}$$

$y_{i,k} = 1$ **si el punto** q_i **tiene como centroide más cercano a** μ_k**; y** $y_{i,k} = 0$**, en otro caso. Finalmente, definimos cada zona geográfica** C_k **como sigue:**

$$C_k = \{q_i\} \mid y_{i,k} = 1 \tag{1.2}$$

1.5.2. Parte II

Sea T **la matriz que almacena los tiempos de traslado entre puntos de venta, definimos la función que calcula el tiempo total de traslado en una zona** C_k **como sigue:**

$$f_2(C_k, z_{i,j,k}) = \sum_{\forall q_i, q_j \in C_k} t_{i,j} \cdot z_{i,j,k} \tag{1.3}$$

donde $t_{i,j} = T[i, j]$ **y** $z_{i,j,k} = 1$ **si cuando recorremos la zona** k **pasamos primero al punto** q_i **e inmediatamente al punto** q_j**. Por otro lado, definimos la función que calcula el tiempo total de servicio en una zona** C_k **de la siguiente forma:**

$$f_3(C_k) = \sum_{\forall q_i \in C_k} a_i \tag{1.4}$$

Una vez definidas f_2 **y** f_3**, podemos calcular el costo (tiempo de entrega) de una zona** C_k **como sigue:**

$$Cost(C_k, z_{i,j,k}) = f_3(C_k) + \min_{z_{i,j,k}} f_2(C_k, z_{i,j,k}) \tag{1.5}$$

1.5.3. Optimización mono-objetivo

Finalmente, definimos la función que evalua qué tan diferentes son los costos de las diferentes zonas geográficas de la siguiente forma:

$$f(C,K) = \sqrt{\frac{1}{N}\sum_{k=1}^{K}(Cost(C_k, z_{i,j,k}) - \mu)^2} \tag{1.6}$$

donde $\mu = \frac{1}{N}\sum_{k=1}^{K} Cost(C_k)$. A continuación se define el problema de optimización principal:

$$\text{mín } f(C,K) \tag{1.7}$$

1.6. Recocido Simulado

1.6.1. Representación de una solución

Como podemos observar en la definición del problema, nuestra tarea es definir grupos (zonas geográficas). Dado que lo vamos a hacer a partir de centroides, nuestra solución será un vector de K centroides. Una vez definidos los centroides se podrán definir las zonas geográficas, colocando una etiqueta a cada punto de venta. La forma en la que organizaremos la información será la siguiente:

1. **Centroides: Arreglo de numpy**
2. **Etiquetas: Arreglo de numpy**

Debido a que vamos a estar calculando distancias entre puntos de venta y centroides, los puntos de venta se van a almacenar en un arreglo de numpy.

[11]:
```
np_pdvs = df_pdvs[["latitud", "longitud"]].to_numpy()
```

1.6.2. Generación de la solución inicial

Para la instancia LE-instancia-1 se van a considerar 10 zonas geográficas.

[12]:
```
K = 10
```

La solución inicial serán K posiciones aleatorias (latitud, longitud) las cuales se generarán a partir de las latitudes y longitudes de los puntos de venta.

[13]:
```
def get_random_centroids(df_pdvs, k):
    centroids = df_pdvs.sample(k)
    return centroids[['latitud', 'longitud']].to_numpy()
```

[14]:
```
np_centroids = get_random_centroids(df_pdvs, K)
np_centroids
```

[14]:
```
array([[  19.99123175, -102.26229657],
       [  19.15614173, -102.53844539],
       [  20.05527029, -102.72145232],
       [  19.42908538, -102.03153402],
       [  19.77210272, -100.22789034],
       [  19.84984083, -102.06591731],
       [  19.69645406, -100.56077976],
       [  19.64840776, -101.24015918],
       [  19.05894359, -102.3305985 ],
       [  19.02041432, -102.10128631]])
```

Una vez que se tienen definidos los centroides se puede realizar el agrupamiento. Primero vamos a definir las funciones que nos permiten calcular la distancia entre los puntos de venta y los centroides. La siguiente función calcula la distancia de cada punto de venta a un centroide en particular.

[15]:
```
def distance_to_centroid(np_pdvs, np_centroid):
    d = [ haversine(np_pdvs[i], np_centroid, unit=Unit.KILOMETERS) for i␣
 ↪in range(len(np_pdvs))]
    return np.array(d)
```

A continuación construimos una función que nos regresa una matriz donde cada fila corresponde a un punto de venta y cada columna a un centroide. Es decir que la fila i contiene las distancias del punto de venta q_i a cada uno de los centroides.

[16]:
```
def get_distance_to_centroids(np_pdvs, np_centroids, k):
    m = [ distance_to_centroid(np_pdvs, np_centroids[i]) for i in␣
 ↪range(k)]
    return np.array(m).T
```

A partir de las distancias podemos definir las etiquetas que indican la zona geográfica a la que pertenece cada punto con las siguientes funciones.

[17]:
```
def get_label_of_nearest_centroid(distances_matrix):
    return np.argmin(distances_matrix, axis=1)
```

Finalmente, construimos una función que realiza el agrupamiento. La función regresa un DataFrame con los puntos de venta etiquetados.

[18]:
```
def clustering(np_pdvs, np_centroids, df_pdvs):
    np_distances_to_centroids = get_distance_to_centroids(np_pdvs,␣
 ↪np_centroids, K)
    np_labels = get_label_of_nearest_centroid(np_distances_to_centroids)
    df_pdvs_labeled = df_pdvs.copy()
    df_pdvs_labeled["etiqueta"] = np_labels
```

```
    return df_pdvs_labeled
```

[19]:
```
df_pdvs_labeled = clustering(np_pdvs, np_centroids, df_pdvs)
```

Imprimimos los primeros y últimos cinco puntos de venta del DataFrame.

[20]:
```
df_pdvs_labeled.head()
```

[20]:

id	latitud	longitud	tiempo_servicio	etiqueta
0	19.452952	-101.732303	1788	3
1	20.272433	-102.561430	1723	2
2	19.983756	-101.761406	1161	5
3	20.049552	-102.729363	762	2
4	19.855134	-102.054702	1309	5

[21]:
```
df_pdvs_labeled.tail()
```

[21]:

id	latitud	longitud	tiempo_servicio	etiqueta
495	20.055931	-102.721927	639	2
496	19.096221	-101.522277	1643	9
497	19.689010	-100.536500	748	6
498	19.676502	-101.226584	656	7
499	19.706095	-101.176884	1543	7

A continuación definimos una función que nos regresa los puntos de venta de una zona geográfica.

[22]:
```
def get_cluster(df_pdvs_labeled, geo_zone):
    zone = df_pdvs_labeled["etiqueta"] == geo_zone
    return df_pdvs_labeled[zone]
```

Imprimimos los puntos de venta de la zona geográfica "0".

[23]:
```
get_cluster(df_pdvs_labeled, 0)
```

[23]:

id	latitud	longitud	tiempo_servicio	etiqueta
7	20.010046	-102.315376	693	0
8	19.976702	-102.296178	739	0
15	20.343817	-102.032844	1325	0
16	19.988598	-102.274546	1743	0
22	19.976128	-102.274052	866	0
..	...	...	...	...

```
461  19.952764 -102.303968            1309         0
467  20.337809 -102.284829            1132         0
476  20.346380 -102.040748             937         0
484  20.005688 -102.454586            1071         0
485  20.008065 -102.288415            1265         0

[64 rows x 4 columns]
```

1.6.3. Función objetivo

Para generar las rutas de cada zona geográfica vamos a utilizar la función routing_google. Esta función busca una solución al problema del agente viajero (TSP por sus siglas en inglés). Recordemos que este problema consiste en visitar N ciudades, sin pasar dos veces por una misma ciudad, y regresar a la ciudad de origen. En nuestro caso, queremos visitar N puntos de venta partiendo de una bodega y regresando a la misma. La función routing_google recibe como argumento la matriz de distancias entre los puntos geográficos que se quieren visitar y regresa tanto el orden en que deben ser visitados como el tiempo requerido de traslado.

A continuación obtenemos el orden en que debemos visitar los puntos de venta, si quisieramos visitar los primeros diez.

[24]:
```
route, route_time = routing_google(df_T.iloc[:10,:10].values.tolist())
print (route, route_time)
```

```
[6, 10, 1, 7, 3, 5, 9, 8, 2, 4] 21464
```

Creamos una función que obtenga la ruta que se debe seguir en una zona geográfica.

[25]:
```
def get_route_zone(indexes, df_T):
    if not indexes:
        return [], 0

    weight_matrix = df_T.loc[indexes,indexes].values.tolist()
    route, route_time = routing_google(weight_matrix)

    tour = []
    for e in route:
        tour.append(indexes[e-1])

    return tour, route_time
```

Definimos la ruta para la zona geográfica 0:

[26]:
```
df_zone = get_cluster(df_pdvs_labeled, 0)
indexes = df_zone.index.tolist()
tour, cost = get_route_zone(indexes, df_T)
print(tour, cost)
```

```
[431, 457, 388, 47, 437, 287, 253, 484, 400, 53, 7, 141, 416, 8, 406, 48,␣
 ↪173,
461, 271, 79, 128, 216, 89, 218, 55, 71, 292, 16, 22, 367, 37, 427, 113,␣
 ↪117,
332, 136, 208, 64, 436, 280, 325, 485, 201, 90, 442, 58, 467, 348, 99, 476,␣
 ↪178,
359, 300, 441, 214, 93, 212, 229, 15, 353, 193, 245, 137, 446] 15123
```

Ahora definimos una función que diseñe todos los grupos/zonas: obtiene los puntos de venta en cada zona, el orden en que se visitan y los costos de traslado y tiempo de servicio. La información la regresa en un DataFrame.

[27]:
```
def get_zones(df_pdvs_labeled, df_T, K):
    labels = [k for k in range(K)]
    tours = []
    transfer_costs = []
    service_costs = []
    for k in labels:
        df_zone = get_cluster(df_pdvs_labeled, k)
        service_cost = df_zone.tiempo_servicio.sum()
        indexes_zone = df_zone.index.tolist()
        tour, transfer_time = get_route_zone(indexes_zone, df_T)

        service_costs.append(service_cost)
        transfer_costs.append(transfer_time)
        tours.append(tour)

    dict_zones = {"Ruta": tours, "tiempo_total_traslado": transfer_costs,␣
 ↪"tiempo_total_servicio":service_costs}
    return pd.DataFrame(dict_zones)
```

[28]:
```
df_zones = get_zones(df_pdvs_labeled, df_T, K)
```

[29]:
```
df_zones
```

[29]:
```
                                                Ruta ␣
 ↪tiempo_total_traslado  \
0  [431, 457, 388, 47, 437, 287, 253, 484, 400, 5...                  ␣
 ↪15123
```

```
1  [87, 297, 63, 238, 282, 288, 279, 154, 72, 490...                ␣
↪17823
2  [422, 1, 104, 241, 478, 373, 384, 217, 38, 132...                ␣
↪10279
3  [112, 310, 315, 378, 470, 267, 24, 223, 76, 21...                ␣
↪16525
4  [284, 473, 396, 324, 114, 340, 420, 451, 207, ...                 ␣
↪6359
5  [225, 2, 70, 335, 42, 394, 171, 430, 440, 150,...                ␣
↪13051
6  [174, 127, 423, 321, 455, 32, 107, 10, 161, 45...                ␣
↪21980
7  [333, 5, 306, 397, 301, 408, 151, 213, 97, 285...                ␣
↪32515
8  [45, 80, 304, 129, 330, 337, 262, 172, 65, 415...                 ␣
↪1005
9  [135, 363, 496, 143, 344, 30, 258, 162, 222, 4...                ␣
↪18846
```

	tiempo_total_servicio
0	74735
1	35884
2	31793
3	88110
4	15984
5	56606
6	71785
7	171131
8	21614
9	33916

Finalmente, definimos la función objetivo como sigue:

[30]:
```
def f(df_zones):
    df_zones["tiempo_total"] = df_zones.tiempo_total_traslado + df_zones.
↪tiempo_total_servicio
    return df_zones.tiempo_total.std()
```

[31]:
```
f(df_zones)
```

[31]: 53307.90816504933

[32]:
```
df_zones
```

[32]:
```
                                                  Ruta ␣
 ↪tiempo_total_traslado  \
0  [431, 457, 388, 47, 437, 287, 253, 484, 400, 5...                ␣
 ↪15123
1  [87, 297, 63, 238, 282, 288, 279, 154, 72, 490...                ␣
 ↪17823
2  [422, 1, 104, 241, 478, 373, 384, 217, 38, 132...                ␣
 ↪10279
3  [112, 310, 315, 378, 470, 267, 24, 223, 76, 21...                ␣
 ↪16525
4  [284, 473, 396, 324, 114, 340, 420, 451, 207, ...                 ␣
 ↪6359
5  [225, 2, 70, 335, 42, 394, 171, 430, 440, 150,...                ␣
 ↪13051
6  [174, 127, 423, 321, 455, 32, 107, 10, 161, 45...                ␣
 ↪21980
7  [333, 5, 306, 397, 301, 408, 151, 213, 97, 285...                ␣
 ↪32515
8  [45, 80, 304, 129, 330, 337, 262, 172, 65, 415...                 ␣
 ↪1005
9  [135, 363, 496, 143, 344, 30, 258, 162, 222, 4...                ␣
 ↪18846

   tiempo_total_servicio  tiempo_total
0                  74735         89858
1                  35884         53707
2                  31793         42072
3                  88110        104635
4                  15984         22343
5                  56606         69657
6                  71785         93765
7                 171131        203646
8                  21614         22619
9                  33916         52762
```

1.6.4. Vecindario

El vecindario de una solución consiste en un conjunto de soluciones que se pueden generar a partir de pequeños cambios a la solución actual. Para este ejemplo vamos a construir una solución vecina de dos formas posibles. Si la relación entre la zona con menor costo y la de mayor es mayor a 1.5, entonces se elimina el centroide de la zona menos costosa y se genera un centroide aleatorio en la región de la zona más costosa. De lo contrario, se hace un pequeño cambio en la

posición de un centroide aleatorio.

[33]:
```
def get_neighbor(np_centroids, df_pdvs_labeled, df_zones, K):
    new_centroids = np_centroids.copy()
    if 1.5*df_zones.tiempo_total.min() < df_zones.tiempo_total.max():
        min_idx = df_zones.tiempo_total.idxmin()
        max_idx = df_zones.tiempo_total.idxmax()
        centroid = get_cluster(df_pdvs_labeled, max_idx).
↪sample(1)[['latitud', 'longitud']].to_numpy()
        new_centroids[min_idx] = centroid
    else:
        idx = np.random.randint(K)
        new_centroids[idx] += np.random.normal(loc=0.0, scale=0.005,␣
↪size=2)

    return new_centroids
```

A continuación se genera una solución vecina.

[34]:
```
new_np_centroids = get_neighbor(np_centroids, df_pdvs_labeled, df_zones,␣
↪K)
new_df_pdvs_labeled = clustering(np_pdvs, new_np_centroids, df_pdvs)
new_df_zones = get_zones(new_df_pdvs_labeled, df_T, K)
```

Los nuevos centroides quedan de la siguiente forma:

[35]:
```
new_np_centroids
```

[35]:
```
array([[  19.99123175, -102.26229657],
       [  19.15614173, -102.53844539],
       [  20.05527029, -102.72145232],
       [  19.42908538, -102.03153402],
       [  19.70538598, -101.19809768],
       [  19.84984083, -102.06591731],
       [  19.69645406, -100.56077976],
       [  19.64840776, -101.24015918],
       [  19.05894359, -102.3305985 ],
       [  19.02041432, -102.10128631]])
```

Las zonas geográficas generadas a partir de los nuevos centroides quedan como sigue:

[36]:
```
new_df_zones
```

[36]:
```
                                                  Ruta ␣
↪tiempo_total_traslado  \
0  [431, 457, 388, 47, 437, 287, 253, 484, 400, 5...                ␣
↪15123
1  [87, 297, 63, 238, 282, 288, 279, 154, 72, 490...                ␣
↪17823
2  [422, 1, 104, 241, 478, 373, 384, 217, 38, 132...                ␣
↪10279
3  [112, 310, 315, 378, 470, 267, 24, 223, 76, 21...                ␣
↪16525
4  [423, 333, 5, 306, 397, 301, 408, 151, 213, 97...                ␣
↪11381
5  [225, 2, 70, 335, 42, 394, 171, 430, 440, 150,...                ␣
↪13051
6  [321, 455, 32, 107, 10, 161, 29, 115, 456, 358...                ␣
↪27949
7  [277, 169, 220, 61, 462, 140, 370, 270, 479, 1...                ␣
↪19638
8  [45, 80, 304, 129, 330, 337, 262, 172, 65, 415...                  ␣
↪1005
9  [135, 363, 496, 143, 344, 30, 258, 162, 222, 4...                ␣
↪18846

   tiempo_total_servicio
0                  74735
1                  35884
2                  31793
3                  88110
4                 113367
5                  56606
6                  86925
7                  58608
8                  21614
9                  33916
```

Calculamos su valor en la función objetivo:

[37]:
```
f(new_df_zones)
```

[37]:
```
33296.32913300403
```

1.6.5. Función de cambio de temperatura

En este caso vamos a disminuir la temperatura usando la siguiente función lineal:

[38]:
```
def T(t):
    return 0.9*t
```

1.6.6. Algoritmo completo

[39]:
```
def SimulatedAnnealing(t_0, t_f, T, f, df_pdvs, np_pdvs, df_T, k):
    t = t_0
    x_np_centroids = get_random_centroids(df_pdvs, k)
    x_df_pdvs_labeled = clustering(np_pdvs, x_np_centroids, df_pdvs)
    x_df_zones = get_zones(x_df_pdvs_labeled, df_T, k)
    fx = f(x_df_zones)

    print("Solución inicial: ", fx)
    #print(x_df_zones)

    xbest_np_centroids, xbest_df_pdvs_labeled, xbest_df_zones, fbest =␣
↪x_np_centroids, x_df_pdvs_labeled, x_df_zones, fx

    while t >= t_f:
        y_np_centroids = get_neighbor(x_np_centroids, x_df_pdvs_labeled,␣
↪x_df_zones, k)
        y_df_pdvs_labeled = clustering(np_pdvs, y_np_centroids, df_pdvs)
        y_df_zones = get_zones(y_df_pdvs_labeled, df_T, k)
        fy = f(y_df_zones)

        if fy <= fbest:
            xbest_np_centroids, xbest_df_pdvs_labeled, xbest_df_zones,␣
↪fbest = y_np_centroids, y_df_pdvs_labeled, y_df_zones, fy

        if fy <= fx or np.random.uniform(0,1) < np.exp( -(fy-fx)/t ):
            x_np_centroids, x_df_pdvs_labeled, x_df_zones, fy =␣
↪y_np_centroids, y_df_pdvs_labeled, y_df_zones, fy

        t = T(t)

    return xbest_np_centroids, xbest_df_pdvs_labeled, xbest_df_zones,␣
↪fbest
```

[40]:
```
np_centroids, df_pdvs_labeled, df_zones, fbest =␣
↪SimulatedAnnealing(t_0=100, t_f=0.01, T=T, f=f, df_pdvs=df_pdvs,␣
↪np_pdvs=np_pdvs, df_T=df_T, k=K)
```

```
Solución inicial:  92116.01596199822
```

Solución encontrada:

[41]:
```
df_zones
```

[41]:
```
                                                Ruta ␣
↪tiempo_total_traslado  \
0  [320, 286, 298, 434, 52, 62, 283, 88, 399, 410...                ␣
↪8189
1  [31, 290, 168, 216, 89, 218, 55, 367, 37, 8, 4...                ␣
↪16238
2  [107, 455, 321, 32, 147, 264, 17, 127, 174, 27...                ␣
↪18539
3  [110, 481, 119, 4, 428, 249, 50, 494, 339, 226...                ␣
↪9893
4  [370, 140, 270, 479, 105, 69, 138, 164, 393, 2...                ␣
↪19439
5  [10, 161, 29, 115, 456, 358, 204, 424, 19, 207...                ␣
↪10601
6  [135, 363, 344, 143, 496, 378, 315, 310, 112, ...                ␣
↪14322
7  [423, 333, 5, 306, 397, 301, 408, 151, 285, 97...                ␣
↪9639
8  [192, 373, 478, 241, 104, 1, 422, 442, 90, 400...                ␣
↪15239
9  [471, 108, 345, 374, 398, 411, 59, 75, 242, 24...                ␣
↪32300

   tiempo_total_servicio  tiempo_total
0                  65669         73858
1                  62098         78336
2                  37823         56362
3                  45834         55727
4                  65710         85149
5                  52264         62865
6                  73645         87967
7                  64085         73724
8                  58549         73788
9                  75881        108181
```

[42]:
```
fbest
```

[42]:
```
15812.046898558776
```

Dado que Recocido Simulado es un algoritmo estocástico es necesario hacer un

estudio estadístico de su comportamiento. Para ello vamos a ejecutar el algoritmo M veces, utilizando los mismos parámetros, y reportaremos la mejor solución, la peor solución y la solución en la mediana de acuerdo al valor de la función objetivo. Así como el valor de la función objetivo promedio y su desviación estándar.

[43]:
```
def get_statistics(M, t_0, t_f, T, f, df_pdvs, np_pdvs, df_T, k):
    sols = []
    for _ in range(M):
        sol = SimulatedAnnealing(t_0, t_f, T, f, df_pdvs, np_pdvs, df_T,␣
 ↪k)
        sols.append(sol)

    sols.sort(key = lambda x: x[-1])
    best = sols[0]
    worst = sols[-1]
    median = sols[M//2]
    fmean = np.mean([x[-1] for x in sols])
    fstd = np.std([x[-1] for x in sols])

    return best, worst, median, fmean, fstd
```

A continuación ejecutamos la metaheurística 21 veces, usando una temperatura inicial igual a 100 y una temperatura final de 0.01.

[44]:
```
M = 21
t_0 = 100
t_f = 0.01
SA_results = get_statistics(M, t_0, t_f, T, f, df_pdvs=df_pdvs,␣
 ↪np_pdvs=np_pdvs, df_T=df_T, k=K)
```

```
Solución inicial:  52231.09736599027
Solución inicial:  55911.66032591055
Solución inicial:  48450.644100293786
Solución inicial:  59353.70426154265
Solución inicial:  37178.121168982645
Solución inicial:  49758.20869810774
Solución inicial:  49979.133992319454
Solución inicial:  47214.790522203475
Solución inicial:  36683.540144956925
Solución inicial:  61216.81247654837
Solución inicial:  79488.68135079918
Solución inicial:  66832.68136041096
Solución inicial:  45038.567978641004
Solución inicial:  52436.40518920673
```

```
Solución inicial:  37909.807007334304
Solución inicial:  36812.729707112034
Solución inicial:  60557.4584786126
Solución inicial:  55414.955476838564
Solución inicial:  63781.98252240902
Solución inicial:  63179.43132943752
Solución inicial:  55300.27294447409
```

La mejor solución encontrada es:

[45]:
```
SA_results[0][-2]
```

[45]:
```
                                                  Ruta ␣
↪tiempo_total_traslado  \
0  [87, 297, 63, 490, 72, 369, 125, 206, 493, 146...                ␣
↪22768
1  [0, 445, 278, 438, 121, 469, 432, 275, 381, 12...                 ␣
↪8864
2  [431, 240, 167, 454, 199, 211, 54, 248, 177, 3...                ␣
↪10840
3  [423, 333, 5, 306, 397, 301, 347, 331, 230, 41...                 ␣
↪9702
4  [110, 31, 290, 168, 216, 89, 128, 218, 55, 22,...                ␣
↪16011
5  [192, 373, 478, 241, 104, 1, 422, 442, 90, 400...                ␣
↪17759
6  [321, 455, 32, 107, 10, 161, 29, 115, 456, 358...                ␣
↪19149
7  [215, 413, 380, 111, 394, 335, 70, 2, 42, 171,...                ␣
↪15513
8  [95, 265, 239, 480, 498, 357, 228, 303, 139, 2...                 ␣
↪8540
9  [444, 277, 169, 220, 61, 462, 140, 370, 270, 4...                ␣
↪22575

   tiempo_total_servicio  tiempo_total
0                  41030         63798
1                  54965         63829
2                  56316         67156
3                  69927         79629
4                  57998         74009
5                  54110         71869
6                  86240        105389
7                  58451         73964
```

```
8                  66936          75476
9                  55585          78160
```

[46]:
```
SA_results[0][-1]
```

[46]:
```
11915.064567829902
```

La peor solución encontrada es:

[47]:
```
SA_results[1][-2]
```

[47]:
```
                                                  Ruta ␣
↪tiempo_total_traslado  \
0  [270, 393, 219, 164, 138, 69, 105, 479, 140, 3...                ␣
↪14938
1  [85, 240, 167, 454, 54, 211, 199, 431, 457, 38...                ␣
↪15816
2  [19, 424, 207, 451, 420, 340, 114, 324, 396, 4...                ␣
↪17587
3  [248, 177, 371, 383, 389, 392, 390, 460, 477, ...                 ␣
↪9090
4  [204, 358, 456, 115, 29, 161, 10, 148, 463, 31...                 ␣
↪9751
5  [471, 108, 345, 374, 398, 411, 59, 75, 242, 24...                ␣
↪32300
6  [333, 5, 306, 397, 286, 298, 52, 88, 399, 410,...                ␣
↪12448
7  [418, 318, 128, 216, 89, 218, 55, 22, 367, 37,...                ␣
↪10365
8  [225, 2, 70, 335, 42, 394, 171, 430, 440, 150,...                ␣
↪10560
9  [320, 257, 18, 44, 78, 317, 247, 103, 250, 425...                ␣
↪12999

   tiempo_total_servicio  tiempo_total
0                  62535         77473
1                  49526         65342
2                  42148         59735
3                  65849         74939
4                  48783         58534
5                  75881        108181
6                  85601         98049
7                  67008         77373
8                  51356         61916
```

```
9                  52871          65870
```

[48]:
```
SA_results[1][-1]
```

[48]:
```
16650.960157834073
```

La solución en la mediana es:

[49]:
```
SA_results[2][-2]
```

[49]:
```
                                                  Ruta ␣
↪tiempo_total_traslado  \
0  [225, 2, 70, 335, 110, 481, 119, 4, 428, 249, ...                ␣
↪16941
1  [19, 424, 207, 451, 420, 340, 114, 324, 396, 4...                ␣
↪15948
2  [204, 358, 456, 115, 29, 161, 10, 148, 463, 31...                 ␣
↪9394
3  [457, 388, 47, 85, 254, 261, 294, 312, 192, 16...                ␣
↪18656
4  [54, 211, 199, 454, 167, 240, 431, 418, 318, 1...                ␣
↪13142
5  [78, 317, 247, 103, 155, 95, 265, 250, 425, 36...                ␣
↪13665
6  [319, 112, 310, 315, 378, 267, 24, 470, 0, 181...                ␣
↪20072
7  [30, 233, 123, 91, 40, 468, 133, 341, 170, 450...                ␣
↪10886
8  [239, 303, 139, 236, 118, 387, 260, 109, 179, ...                 ␣
↪5265
9  [87, 297, 63, 238, 282, 288, 279, 154, 72, 490...                ␣
↪29731

   tiempo_total_servicio  tiempo_total
0                  61566         78507
1                  38986         54934
2                  47939         57333
3                  43732         62388
4                  76915         90057
5                  57326         70991
6                  52111         72183
7                  68626         79512
8                  79537         84802
9                  74820        104551
```

[50]:
```
SA_results[2][-1]
```

[50]:
```
15361.510160137252
```

El valor promedio en la función objetivo es:

[51]:
```
SA_results[3]
```

[51]:
```
14919.604538677218
```

Finalmente, la desviación estándar del valor en la función objetivo es:

[52]:
```
SA_results[4]
```

[52]:
```
1320.7035172140677
```

Creamos una función que nos permite graficar las zonas creadas. Para ello definimos primero una función que genera los colores que se van a utilizar en cada zona.

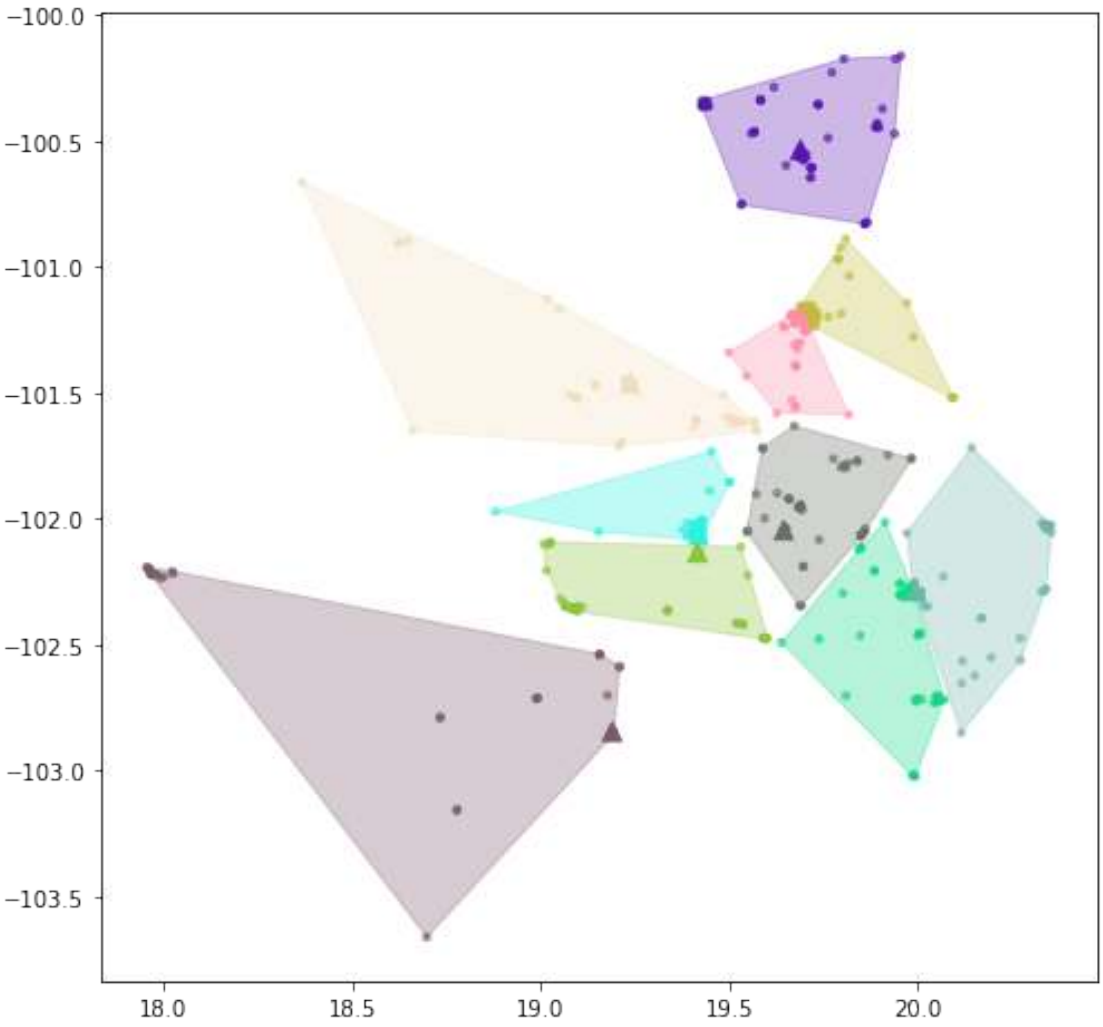
[53]:
```
def get_colors(k):
    color_dict = {}

    for i in range(k):
        color = '#%06X' % np.random.randint(0, 0xFFFFFF)
        color_dict.update({i: color})

    return color_dict
```

[54]:
```
def plot_groups(df_pdvs_labeled, np_centroids, k):
    df = df_pdvs_labeled[["latitud", "longitud"]]
    fig, ax = plt.subplots(1, figsize=(8, 8))
    colors_dict = get_colors(k)

    df['color'] = df_pdvs_labeled.etiqueta.map(colors_dict)
    plt.scatter(df.latitud, df.longitud, c=df.color, alpha=0.6, s=10)
    plt.scatter(np_centroids[:, 0], np_centroids[:, 1], marker='^',␣
↪c=list(colors_dict.values()), s=70)

    for i in df_pdvs_labeled.etiqueta.unique():
        points = df[df_pdvs_labeled.etiqueta == i][['latitud',␣
↪'longitud']].values
        hull = ConvexHull(points)
        x_hull = np.append(points[hull.vertices, 0], points[hull.
↪vertices, 0][0])
```

```
        y_hull = np.append(points[hull.vertices, 1], points[hull.
↪vertices, 1][0])
        plt.fill(x_hull, y_hull, alpha=0.3, c=colors_dict[i])
    plt.show()
```

[55]:
```
best_np_centroids = SA_results[0][0]
best_df_pdvs_labeled = SA_results[0][1]
plot_groups(best_df_pdvs_labeled, best_np_centroids, K)
```

```
/var/folders/g9/vn2vfpcs56x_6scmywd7vn2c0000gn/T/ipykernel_14525/2753626627.
↪py:6
: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df['color'] = df_pdvs_labeled.etiqueta.map(colors_dict)
```



Ejercicio 5: Resolver la instancia generada en el *ejercicio 1* con el algoritmo de RS propuesto. Se debe hacer un estudio estadístico y graficar las zonas geográficas creadas a partir de la mejor solución encontrada.

Ejercicio 6: Proponer dos funciones para el cambio de temperatura y utilizarlas en el algoritmo de RS diseñado para resolver la instancia generada en el *ejercicio 1*. Hacer un estudio estadístico de cada versión y presentar una tabla comparativa considerando las tres versiones (versión original y las dos versiones nuevas). Finalmente graficar las zonas geográficas creadas a partir de la mejor solución encontrada.

Ejercicio 7: Proponer un algoritmo distinto para la generación de una solucicón vecina y utilizarlo en el algoritmo de RS diseñado para resolver la instancia generada en el *ejercicio 1*. Hacer un estudio estadístico y presentar una tabla comparativa considerando las dos versiones (versión original y versión nueva).

Finalmente graficar las zonas geográficas creadas a partir de la mejor solución encontrada.

Capítulo 2

Vehículo compartido

2.1. Parámetros de entrada

	Descripción
$G = (V, E)$	**Grafo vial**
K_A	**Número de personas tipo** A
K_B	**Número de personas tipo** B
$ A = { a_i }$	**Conjunto de pares ordenados** $a_i = (o_{a_i}, d_{a_i})$ **que representan el nodo origen y nodo destino de las personas tipo** A**, donde** $0 \leq i < K_A$
$B = \{b_i\}$	**Conjunto de 3-tuplas** $b_i = (o_{b_i}, d_{b_i}, c_{b_i})$ **que representan el nodo origen, el nodo destino y la capacidad de las personas tipo** B**, donde** $0 \leq i < K_B$

2.2. Variables

	Descripción
$G_A = [g_{a_0}, \cdots, g_{a_{K_A-1}}]$	**Etiquetas de los grupos asignados a las personas tipo** A**,** g_{a_i} **es el grupo asignado a la persona** i **de tipo** A**.** $g_{a_i} = -1$ **significa que la persona** a_i **no tiene asignado un grupo**
$G_B = [g_{b_0}, \cdots, g_{b_{K_B-1}}]$	**Etiquetas de los grupos asignados a las personas tipo** B**,** g_{b_i} **es el grupo asignado a la persona** i **de tipo** B

	Descripción
$C_k = \{a_i \parallel g_{a_i} = k\} \cup \{b_i \parallel g_{b_i} = k\}$	**Conjunto de personas en el grupo k, con $-1 \leq k < K\_B$**
$C = \{C_{-1}, C_0, \cdots, C_{K_B-1}\}$	**Conjunto de grupos**

2.3. Preparación del ambiente

Las bibliotecas/módulos que se van a utilizar son: 1. osmnx 2. networkx 3. numpy 4. pandas 5. matplotlib 6. pymoo

Para instalarlas usamos:

```
!pip install osmnx
!pip install networkx
!pip install numpy
!pip install pandas
!pip install matplotlib
!pip install -U pymoo
```

[2]:
```
import networkx as nx
import numpy as np
import osmnx as ox
import pandas as pd
import matplotlib.colors as mcolors
import matplotlib.pyplot as plt
```

Asignamos una semilla de números aleatorios para poder generar siempre la misma secuencia de números aleatorios.

[3]:
```
np.random.seed(0)
```

2.4. Obtención de datos

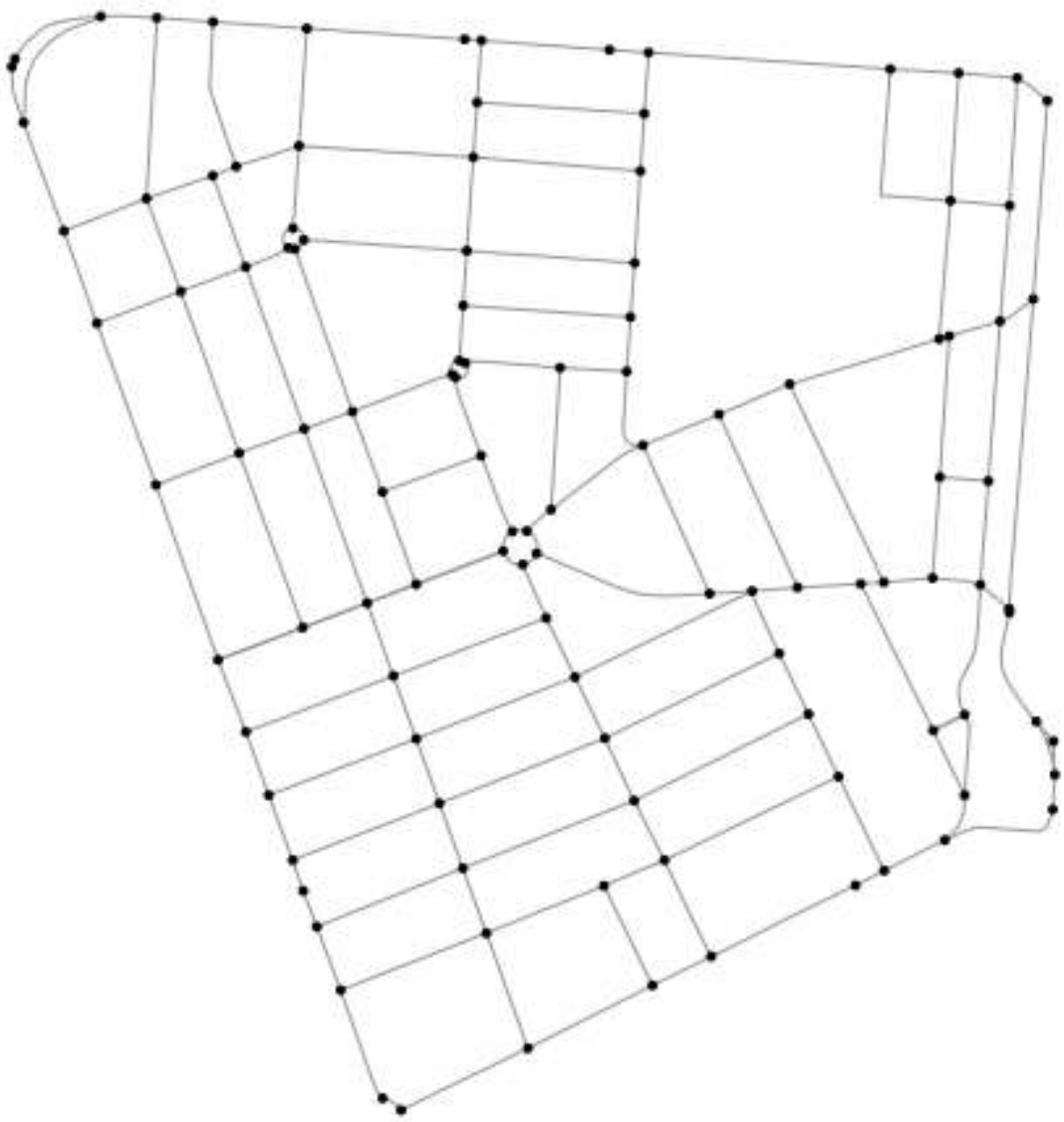
2.4.1. Instancia de prueba 1: VC-instancia-1

Para generar instancias del problema, vamos a utilizar el módulo osmnx para obtener el grafo de las calles de Ciudad Jardín, Córdoba, España considerando que la movilidad será a través de un automóvil. Dicho grafo cuenta con 112 intersecciones entre calles.

[4]:
```
Graph = ox.graph_from_place('Ciudad Jardín, Córdoba, Spain',␣
 ↪network_type="drive")
print("Número de nodos:", len(Graph.nodes))
```

```
Número de nodos: 112
```

[5]:
```
ox.plot_graph(Graph, bgcolor="w", node_color="black")
```



[5]:
```
(<Figure size 576x576 with 1 Axes>, <AxesSubplot:>)
```

Para esta instancia vamos a generar 10 **personas tipo** A **y** 3 **personas tipo** B **seleccionando aleatoriamente el nodo origen y destino de cada una.**

[6]:
```
K_A = 10
A = np.random.choice(Graph.nodes,(10,2))
```

```
print("Identificadores de los nodos:\n", A)
```

```
Identificadores de los nodos:
 [[ 330001516  330001519]
 [ 330004224  330004259]
 [ 330004259 4119801564]
 [ 329999809  408608730]
 [ 330000034  330001346]
 [ 409738622  330004374]
 [ 409738623  409738623]
 [ 329999856  330003830]
 [ 330004226 4119801563]
 [ 330001350  409738622]]
```

[7]:
```
K_B = 3
B = np.random.choice(Graph.nodes,(K_B,2))
print("Identificadores de los nodos:\n", B)
```

```
Identificadores de los nodos:
 [[330001518 409738623]
 [408608727 330001348]
 [330000061 404247466]]
```

Para visualizar las coordenadas geográficas de los nodos definimos la siguiente función:

[8]:
```
def print_coordinates(n, Graph):
    print("( longitud: ", Graph.nodes[n]['x'], ", " + "latitud: ", Graph.
 ↪nodes[n]['y'], ")")
```

Nodo origen de la persona a_0:

[9]:
```
print_coordinates(A[0][0], Graph)
```

```
( longitud:  -4.7895508 , latitud:  37.8851172 )
```

Nodo destino de la persona a_1:

[10]:
```
print_coordinates(A[0][1], Graph)
```

```
( longitud:  -4.7863348 , latitud:  37.8849424 )
```

Las capacidades de los vehículos de las personas tipo B se eligen de forma aleatoria seleccionando un número entre 1 y 4.

[11]:
```
B = [list(i)+[np.random.randint(1,4)] for i in B]
B
```

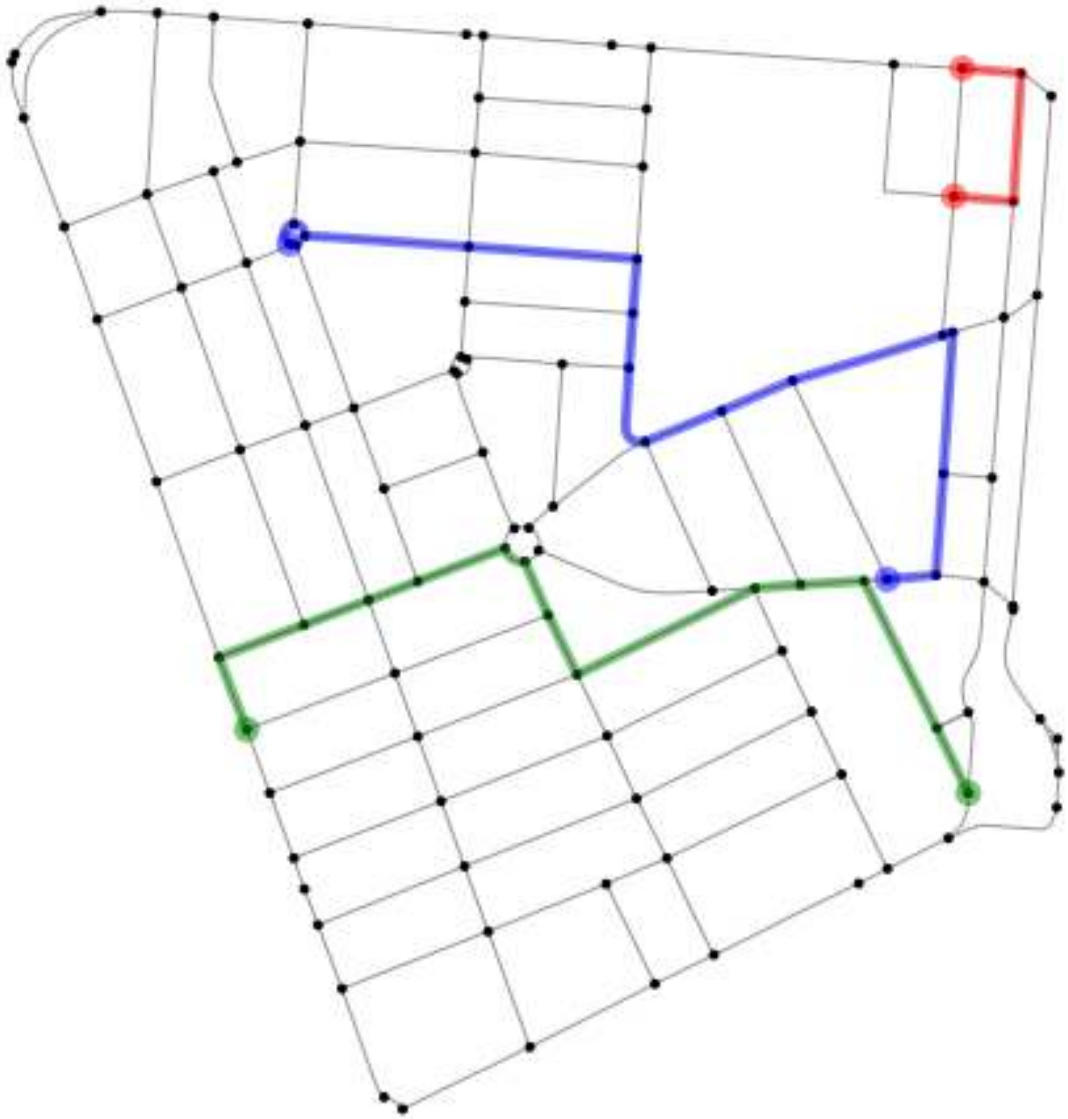
[11]:
```
[[330001518, 409738623, 1],
 [408608727, 330001348, 2],
 [330000061, 404247466, 1]]
```

Para generar los caminos que recorrerán las personas tipo B vamos a utilizar el módulo networkx y la función shortest_path, la cual cálcula el camino más corto utilizando el algoritmo de dijkstra, indicando que el peso de las aristas será la distancia entre un nodo y otro.

[12]:
```
B_routes = [nx.shortest_path(Graph, b[0], b[1], weight="distance") for b␣
↪in B]
```

A continuación se muestran los caminos dentro del grafo.

[13]:
```
ox.plot.plot_graph_routes(Graph, B_routes, ['r', 'b', 'g'], bgcolor="w",␣
↪node_color="black")
```



[13]: (<Figure size 576x576 with 1 Axes>, <AxesSubplot:>)

Para organizar los parámetros de entrada creamos dos clases UserA y UserB. En el caso de una persona tipo A almacenamos su nodo de origen, su nodo destino y la distancia que recorrería si no fuese asignado a un vehículo. Para esto último utilizamos la función shortest_path_length del módulo networkx. Además creamos tres diccionarios, el primero nos permite almacenar tanto el nodo más cercano de la ruta de una persona B al nodo origen de la persona A como la distancia entre ambos nodos. El segundo nos permite almacenar los mismos datos pero considerando la distancia desde los nodos de la ruta de la persona B hacia el nodo destino de la persona A. Finalmente, el tercer diccionario

nos permite almacenar la distancia recorrida por la persona A mientras está abordo del vehículo asignado. Estos diccionarios se irán llenando conforme la información sea requerida por los algoritmos de búsqueda.

[14]:
```
class UserA:
    def __init__(self, a, Graph):
        self.source = a[0]
        self.target = a[1]
        self.independent_distance = nx.shortest_path_length(Graph, self.
↪source, self.target, weight="distance")
        self.distanceToB = {}
        self.distanceFromB = {}
        self.sharedDistanceWithB = {}

    def __repr__(self):
        mystr = "\nOrigen: " + str(self.source) + "\nDestino: " +␣
↪str(self.target)
        mystr += "\nDistancia del camino más corto: " + str(self.
↪independent_distance)
        mystr += "\nDistancias hacia los usuarios B: "
        for k, v in self.distanceToB.items():
            mystr += "\n\tb_" + str(k) + ": " + str(v[0]) + ", " +␣
↪str(v[1])
        mystr += "\nDistancias desde los usuarios B: "
        for k, v in self.distanceFromB.items():
            mystr += "\n\tb_" + str(k) + ": " + str(v[0]) + ", " +␣
↪str(v[1])
        mystr += "\nDistancia compartida con los usuarios B: "
        for k, v in self.sharedDistanceWithB.items():
            mystr += "\n\tb_" + str(k) + ": " + str(v[0]) + ", " +␣
↪str(v[1])

        return mystr
```

En el caso de una persona tipo B almacenamos su etiqueta de grupo, su nodo de origen, su nodo destino, su capacidad y la ruta que va a seguir el vehículo.

[15]:
```
class UserB:
    def __init__(self, i, b, Graph):
        self.label = i
        self.source = b[0]
        self.target = b[1]
        self.capacity = b[2]
```

```
        self.path = nx.shortest_path(Graph, self.source, self.target,␣
↪weight="distance")

    def __repr__(self):
        mystr = "\nGrupo: " + str(self.label) + "\nOrigen: " + str(self.
↪source) + "\nDestino: " + str(self.target)
        mystr += "\nCapacidad: " + str(self.capacity)
        mystr += "\nRuta: "
        for node in self.path:
            mystr += " " + str(node)

        return mystr
```

A continuación creamos los objetos correspondientes a las personas tipo A e imprimimos los primeros dos: a_0 y a_1.

[16]:
```
def createUsersA(A, Graph):
    return [UserA(a,Graph) for a in A]
```

[17]:
```
myUsersA = createUsersA(A, Graph)
for user in myUsersA[:2]:
    print(user)
```

```
Origen: 330001516
Destino: 330001519
Distancia del camino más corto: 3
Distancias hacia los usuarios B:
Distancias desde los usuarios B:
Distancia compartida con los usuarios B:

Origen: 330004224
Destino: 330004259
Distancia del camino más corto: 3
Distancias hacia los usuarios B:
Distancias desde los usuarios B:
Distancia compartida con los usuarios B:
```

A continuación creamos los objetos correspondientes a las personas tipo B y los imprimimos.

[18]:
```
def createUsersB(B, Graph):
    return [UserB(index, b, Graph) for index, b in enumerate(B)]
```

[19]:
```
myUsersB = createUsersB(B, Graph)
for user in myUsersB:
    print(user)
```

```
Grupo: 0
Origen: 330001518
Destino: 409738623
Capacidad: 1
Ruta:  330001518 330001519 409738624 409738623

Grupo: 1
Origen: 408608727
Destino: 330001348
Capacidad: 2
Ruta:  408608727 408608730 5388004543 408608729 409738622 408608726␣
 ↪408608723
5401477694 330004115 330004223 330004224 329999804 330001350 330001346␣
 ↪330001348

Grupo: 2
Origen: 330000061
Destino: 404247466
Capacidad: 1
Ruta:  330000061 329999371 330003738 330000010 330003465 329999409 329999855
329999856 329999857 330000237 408608725 329999725 5387996982 404247466
```

2.5. Definición del problema de optimización

Sea $route(v\_0, v\_f) = [v\_0, v\_1, \cdots, v\_f]; |; v\_i \in V$ **la ruta más corta para ir de un nodo origen $ v_0 $ a un nodo destino** v_f**,** $dist(v_0, v_f)$ **la distancia de la ruta más corta entre el nodo** v_0 **y el nodo** v_f**,** $nn_1(v, r)$ **el nodo más cercano de la ruta** r **al nodo origen** v**,** $nn_2(r, v)$ **el nodo más cercano de la ruta** r **al nodo destino** v**, definimos la función para calcular la suma de las distancias recorridas por los usuarios tipo** A **de la siguiente forma:**

$$f_1(G_A, G_B) = \sum_{g_{a_i} \in G_A, g_{a_i} \neq -1} (3.5 \cdot (dist(o_{a_i}, u_1) + dist(u_2, d_{a_i})) + dist(u_1, u_2)) + \sum_{g_{a_i} \in G_A, g_{a_i} = -1} 3.5 \cdot dist(o_{a_i}, d_{a_i}) \tag{2.1}$$

donde $u_1 = nn_1(o_{a_i}, r_{b_j})$**,** $u_2 = nn_2(r_{b_j}, d_{a_i})$**,** $r_{b_j} = route(o_{b_j}, d_{b_j})$ **y** $g_{b_j} = g_{a_i}$**. Por otro**

lado, definimos la cantidad de personas tipo A que no tienen asignado un vehículo como sigue:

$$f_2(G_A) = \sum_{g_{a_i} \in G_A, g_{a_i} = -1} 1 \tag{2.2}$$

2.5.1. Optimización mono-objetivo

$$\text{mín } f_1(G_A, G_B) \tag{2.3}$$

tal que para cada $b_j \in B$ se cumpla:

$$g(G_A, G_B) = \sum_{g_{a_i} \in G_A, g_{a_i} = g_{b_j}} 1 \quad \leq c_{b_j} \tag{2.4}$$

2.5.2. Optimización multi-objetivo

$$\text{mín } F(G_A, G_B) = [f_1(G_A, G_B), f_2(G_A)]^T \tag{2.5}$$

tal que para cada $b_j \in B$ se cumpla:

$$\sum_{g_{a_i} \in G_A, g_{a_i} = g_{b_j}} 1 \quad \leq c_{b_j} \tag{2.6}$$

2.6. Problema 1 - Recocido Simulado: Caso mono-objetivo

2.6.1. Representación de una solución

Como podemos observar en la definición del problema, nuestra tarea es asignar una etiqueta de grupo a cada persona (A ó B). Dado que las personas tipo B son aquellas que pueden prestar un vehículo para formar un grupo, tomaremos su índice como la etiqueta de su grupo y dicha etiqueta podrá ser asignada a las personas tipo A. Cada índice i tendrá c_{b_i} copias, es decir, la capacidad del vehículo de la persona b_i. En caso de que haya más personas A que etiquetas disponibles, rellenaremos el resto con etiquetas -1. Por lo anterior, veremos una solución como un arreglo con las etiquetas de las personas tipo A, por ejemplo:

[20]:
```
G_A = []
for i in range(K_B):
    G_A += myUsersB[i].capacity*[myUsersB[i].label]
remaining = K_A - len(G_A)
if remaining > 0:
```

```
    G_A += [-1]*remaining
print("G_A = ", G_A)
```

```
G_A =  [0, 1, 1, 2, -1, -1, -1, -1, -1, -1]
```

La solución anterior indica que la persona a_0 está asignada al vehículo de la persona b_0, mientras que las personas a_1 y a_2 están asignadas al vehículo de la persona b_1 y, la persona a_3 está asignada al vehículo del usuario b_2. Finalmente, las personas a_4, a_5, a_6, a_7, a_8 y a_9 no tienen asignado un vehículo. Las soluciones restantes al problema las podemos generar haciendo permutaciones de las posiciones de G_A, por ejemplo:

[21]:
```
G_A_new = G_A.copy()
np.random.shuffle(G_A_new)
print("G_A_new = ", G_A_new)
```

```
G_A_new =  [-1, 1, -1, -1, 1, -1, -1, -1, 0, 2]
```

Dado que el problema indica que no es necesario llenar los vehículos cuando es menos costoso para las personas A trasportarse de su origen a su destino que transportarse a los nodos de las rutas que siguen las personas tipo B, agregaremos valores con -1 a nuestra solución. En este caso agregaremos el mismo número de -1 que personas de tipo A:

[22]:
```
G_A_ext = G_A + [-1]*(len(G_A)-remaining)
print("G_A_ext = ", G_A_ext)
```

```
G_A_ext =  [0, 1, 1, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
```

Cuando se evalúa una solución en la función objetivo solo se van a considerar las primeras K_A posiciones que son las que indican las etiquetas de las personas tipo A. Por lo tanto, la solución G_A_Ext es equivalente a la solución G_A. A continuación definimos una función que nos permite obtener la permutación inicial a partir de los usuarios A y B.

[23]:
```
def initial_permutation(myUsersA:list, K_A:int, myUsersB:list, K_B:int):
    permutation = []
    for i in range(K_B):
        permutation += [myUsersB[i].label]*myUsersB[i].capacity
    permutation += [-1]*K_A
    return np.array(permutation)
```

[24]:
```
print(initial_permutation(myUsersA, K_A, myUsersB, K_B))
```

```
[ 0  1  1  2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
```

2.6.2. Generación de la solución inicial

La solución inicial será una permutación aleatoria de las etiquetas creadas en la sección anterior.

[25]:
```
def get_initial_solution(myUsersA, K_A, myUsersB, K_B):
    x = initial_permutation(myUsersA, K_A, myUsersB, K_B)
    np.random.shuffle(x)
    return x
```

[26]:
```
print(get_initial_solution(myUsersA, K_A, myUsersB, K_B))
```

```
[-1 -1 -1 -1  1 -1 -1 -1  1  0 -1 -1 -1  2]
```

La solución anterior indica que las personas a_0, a_1, a_2, a_3, a_5, a_6, a_7 no tienen asignado un vehículo, mientras que las persona a_4 y a_8 están asignadas al vehículo de la persona b_1 y, la persona a_9 está asignada el vehículo de la persona b_0.

2.6.3. Función objetivo

A continuación definimos la función $dist$ que nos permite calcular la distancia mínima entre un nodo de origen y nodo destino.

[27]:
```
def dist(source, target, Graph):
    return nx.shortest_path_length(Graph, source, target,␣
 ↪weight="distance")
```

Ahora definimos la función nn_1 que nos permite encontrar el nodo más cercano de una ruta a un nodo de origen. Dicha función regresa tanto la distancia entre los nodos como el nodo encontrado.

[28]:
```
def nn_1(source, path, Graph):
    return min([[dist(source, node, Graph), node] for node in path])
```

De igual forma definimos la función nn_2 que nos permite encontrar el nodo más cercano de una ruta a un nodo destino.

[29]:
```
def nn_2(path, target, Graph):
    return min([[dist(node, target, Graph), node] for node in path])
```

Nodo más cercano de la ruta de la persona b_0 al nodo origen de la persona a_0:

[30]:
```
nn_1(myUsersA[0].source, myUsersB[0].path, Graph)
```

[30]:
```
[2, 330001518]
```

Nodo más cercano de la ruta de la persona b_0 al nodo destino de la persona a_0:

[31]:
```
nn_2(myUsersB[0].path, myUsersA[0].target, Graph)
```

[31]:
```
[0, 330001519]
```

Una vez implementadas las funciones $dist$, nn_1 y nn_2 definimos la función objetivo como sigue:

[32]:
```
def f1(G_A, myUsersA, K_A, myUsersB, Graph):
    distance = 0
    for index, group in enumerate(G_A[:K_A]):
        if group == -1:
            distance += 3.5*myUsersA[index].independent_distance
        else:
            if group not in myUsersA[index].distanceToB:
                myUsersA[index].distanceToB[group] = nn_1(myUsersA[index].
 ↪source, myUsersB[group].path, Graph)

            if group not in myUsersA[index].distanceFromB:
                myUsersA[index].distanceFromB[group] =␣
 ↪nn_2(myUsersB[group].path, myUsersA[index].target, Graph)

            if group not in myUsersA[index].sharedDistanceWithB:
                myUsersA[index].sharedDistanceWithB[group] =␣
 ↪dist(myUsersA[index].distanceToB[group][1], myUsersA[index].
 ↪distanceFromB[group][1], Graph)

            distance += 3.5*(myUsersA[index].distanceToB[group][0] +␣
 ↪myUsersA[index].distanceFromB[group][0]) + myUsersA[index].
 ↪sharedDistanceWithB[group]
    return distance
```

Calculamos el valor de la función objetivo para la solución G_A_ext.

[33]:
```
print("f1(", G_A_ext, ") = ", f1(G_A_ext, myUsersA, K_A, myUsersB, Graph))
```

```
f1( [0, 1, 1, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1] ) =  210.0
```

2.6.4. Vecindario

El vecindario de una solución es un conjunto de soluciones que se pueden generar realizando pequeños cambios a la solución actual. Para este ejemplo, vamos a construir una solución vecina realizando n intercambios entre las posiciones del arreglo. Donde n es un parámetro que indica el usuario.

[34]:
```
def get_neighbor(G_A, n):
    new_G_A = G_A.copy()
    for _ in range(n):
        indexes = np.random.choice(len(G_A), 2, replace=False)
        new_G_A[indexes[0]], new_G_A[indexes[1]] =new_G_A[indexes[1]],␣
↪new_G_A[indexes[0]]
    return new_G_A
```

A continuación se genera una solución vecina para G_A_ext realizando un único intercambio.

[35]:
```
np.random.seed(1)
print("G_A_ext:", G_A_ext)
print("Vecino:", get_neighbor(G_A_ext, 1))
```

```
G_A_ext: [0, 1, 1, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
Vecino: [0, 1, 1, -1, -1, -1, -1, 2, -1, -1, -1, -1, -1, -1]
```

2.6.5. Función de cambio en la temperatura

En este caso vamos a disminuir la temperatura usando la siguiente función lineal:

[36]:
```
def T(t):
    return 0.9*t
```

2.6.6. Algoritmo completo

[37]:
```
def SimulatedAnnealing(t_0, t_f, T, f, myUsersA, K_A, myUsersB, K_B,␣
↪Graph, n_swaps):
    t = t_0
    x = get_initial_solution(myUsersA, K_A, myUsersB, K_B)
    fx = f(x, myUsersA, K_A, myUsersB, Graph)
    xbest, fbest = x, fx

    while t >= t_f:
        y = get_neighbor(x, n_swaps)
        fy = f(y, myUsersA, K_A, myUsersB, Graph)

        if fy <= fbest:
            xbest, fbest = y, fy

        if fy <= fx or np.random.uniform(0,1) < np.exp( -(fy-fx)/t ):
            x, fx = y, fy
```

```
        t = T(t)

    return xbest, fbest
```

[38]:
```
SimulatedAnnealing(t_0=100, t_f=0.01, T=T, f=f1, myUsersA=myUsersA,␣
↪K_A=K_A, myUsersB=myUsersB, K_B=K_B, Graph=Graph, n_swaps=1)
```

[38]:
```
(array([-1,  1, -1,  2, -1,  1, -1, -1, -1,  0, -1, -1, -1, -1]), 200.0)
```

2.6.7. Estadísticas

Dado que RS es un algoritmo estocástico es necesario hacer un estudio estadístico de su comportamiento. Para ello vamos a ejecutar el algoritmo M veces, utilizando los mismos parámetros, y reportaremos la mejor solución, la peor solución y la solución en la mediana de acuerdo al valor de la función objetivo. Así como el valor de la función objetivo promedio y su desviación estándar.

[39]:
```
def get_statistics(M, t_0, t_f, T, f, myUsersA, K_A, myUsersB, K_B,␣
↪Graph, n_swaps):
    sols = []
    for _ in range(M):
        sol = SimulatedAnnealing(t_0, t_f, T, f, myUsersA, K_A, myUsersB,␣
↪K_B, Graph, n_swaps)
        sols.append(sol)

    sols.sort(key = lambda x: x[1])
    best = sols[0]
    worst = sols[-1]
    median = sols[M//2]
    fmean = np.mean([x[1] for x in sols])
    fstd = np.std([x[1] for x in sols])

    return best, worst, median, fmean, fstd
```

A continuación ejecutamos la metaheurística utilizando uno, dos y tres intercambios para generar al vecino, una temperatura inicial igual a 1000 y una temperatura final igual a 0.001. Cada versión se ejecuta 100 veces.

[40]:
```
M = 100
SA_results_1 =get_statistics(M, 1000, 0.001, T, f1, myUsersA, K_A,␣
↪myUsersB, K_B, Graph, 1)
SA_results_2 =get_statistics(M, 1000, 0.001, T, f1, myUsersA, K_A,␣
↪myUsersB, K_B, Graph, 2)
```

```
SA_results_3 =get_statistics(M, 1000, 0.001, T, f1, myUsersA, K_A,␣
 ↪myUsersB, K_B, Graph, 3)
```

Utilizamos el módulo pandas para presentar los datos.

[41]:
```
results = pd.DataFrame()
results.index = ["SA: 1 swaps", "SA: 2 swaps", "SA: 3 swaps"]
results["x_best"] = [SA_results_1[0][0][:K_A], SA_results_2[0][0][:K_A],␣
 ↪SA_results_3[0][0][:K_A]]
results["f_best"] = [SA_results_1[0][1], SA_results_2[0][1],␣
 ↪SA_results_3[0][1]]

results["x_worst"] = [SA_results_1[1][0][:K_A], SA_results_2[1][0][:K_A],␣
 ↪SA_results_3[1][0][:K_A]]
results["f_worst"] = [SA_results_1[1][1], SA_results_2[1][1],␣
 ↪SA_results_3[1][1]]

results["x_median"] = [SA_results_1[2][0][:K_A], SA_results_2[2][0][:
 ↪K_A], SA_results_3[2][0][:K_A]]
results["f_median"] = [SA_results_1[2][1], SA_results_2[2][1],␣
 ↪SA_results_3[2][1]]

results["f_mean"] = [SA_results_1[3], SA_results_2[3], SA_results_3[3]]
results["f_std"] = [SA_results_1[4], SA_results_2[4], SA_results_3[4]]
```

[42]:
```
results
```

[42]:

```
                                          x_best  f_best  \
SA: 1 swaps  [0, -1, -1, 2, -1, 1, -1, -1, -1, 1]   160.0
SA: 2 swaps  [0, -1, -1, 2, -1, 1, -1, -1, -1, 1]   160.0
SA: 3 swaps  [0, -1, -1, 2, -1, 1, -1, -1, -1, 1]   160.0

                                           x_worst  f_worst  \
SA: 1 swaps  [-1, -1, -1, 2, -1, 1, -1, -1, -1, -1]    205.0
SA: 2 swaps    [-1, -1, -1, 2, 1, 1, -1, -1, -1, 0]    193.0
SA: 3 swaps  [-1, -1, 1, -1, -1, -1, -1, -1, -1, 1]    185.0

                                         x_median  f_median   f_mean  \
SA: 1 swaps   [-1, -1, -1, 2, 1, -1, 0, -1, -1, 1]     163.0  168.645
SA: 2 swaps  [-1, -1, -1, 2, 1, -1, -1, -1, -1, 1]     163.0  166.170
SA: 3 swaps  [-1, -1, -1, 2, -1, 1, -1, -1, -1, 1]     162.5  164.480

                 f_std
SA: 1 swaps  11.688947
```

```
SA: 2 swaps   7.794941
SA: 3 swaps   4.944148
```

2.6.8. Grupos creados

Implementamos una función que nos permita agrupar las rutas de acuerdo a las etiquetas asignadas.

[43]:
```
def create_groups(G_A, myUsersA, K_A, myUsersB, K_B):
    clusters = {}
    clusters[-1] = []
    for i in range(K_B):
        clusters[i] = [myUsersB[i].path]

    for i, group in enumerate(G_A):
        if group == -1:
            clusters[-1].append(nx.shortest_path(Graph, myUsersA[i].
 ↪source, myUsersA[i].target, weight="distance"))
        else:
            clusters[group].append(nx.shortest_path(Graph, myUsersA[i].
 ↪source, myUsersA[i].target, weight="distance"))
    return clusters
```

Utilizamos la mejor solución para agrupar las rutas según las etiquetas asignadas.

[44]:
```
best = SA_results_3[0][0][:K_A]
clusters = create_groups(best, myUsersA, K_A, myUsersB, K_B)
```

Creamos una función que nos permita crear n colores diferentes:

[45]:
```
def get_colors(n):
    keys = list(mcolors.CSS4_COLORS.keys())
    return list(np.random.choice(keys,n))
```

[46]:
```
get_colors(3)
```

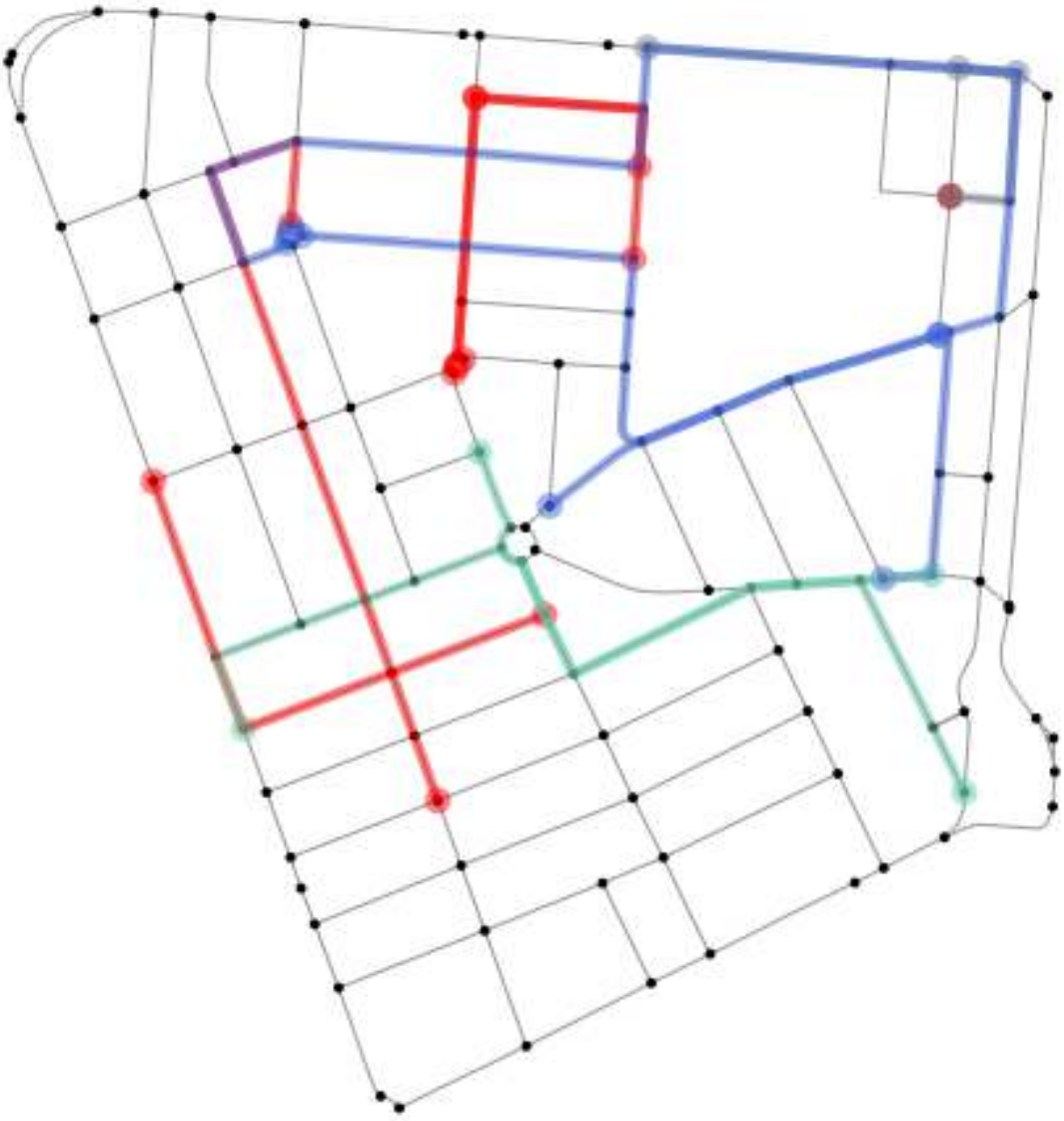
[46]:
```
['lightblue', 'red', 'lightgray']
```

Creamos una función que grafique las rutas de todos los usuarios de tal forma que los usuarios asignados a un mismo grupo tengan un mismo color. Las rutas coloreadas en rojo siempre pertenecen a los usuarios que no tienen asignado un vehículo.

[47]:
```
def plot_clustered_routes(clusters, colors, Graph):
    aux_routes = []
```

```
    aux_colors = []
    colors.append("red")
    for k, v in clusters.items():
        for path in v:
            aux_routes.append(path)
            aux_colors.append(colors[k])

    ox.plot.plot_graph_routes(Graph, aux_routes, aux_colors, bgcolor="w",␣
↪node_color="black")
```

[48]:
```
plot_clustered_routes(clusters, get_colors(K_B), Graph)
```



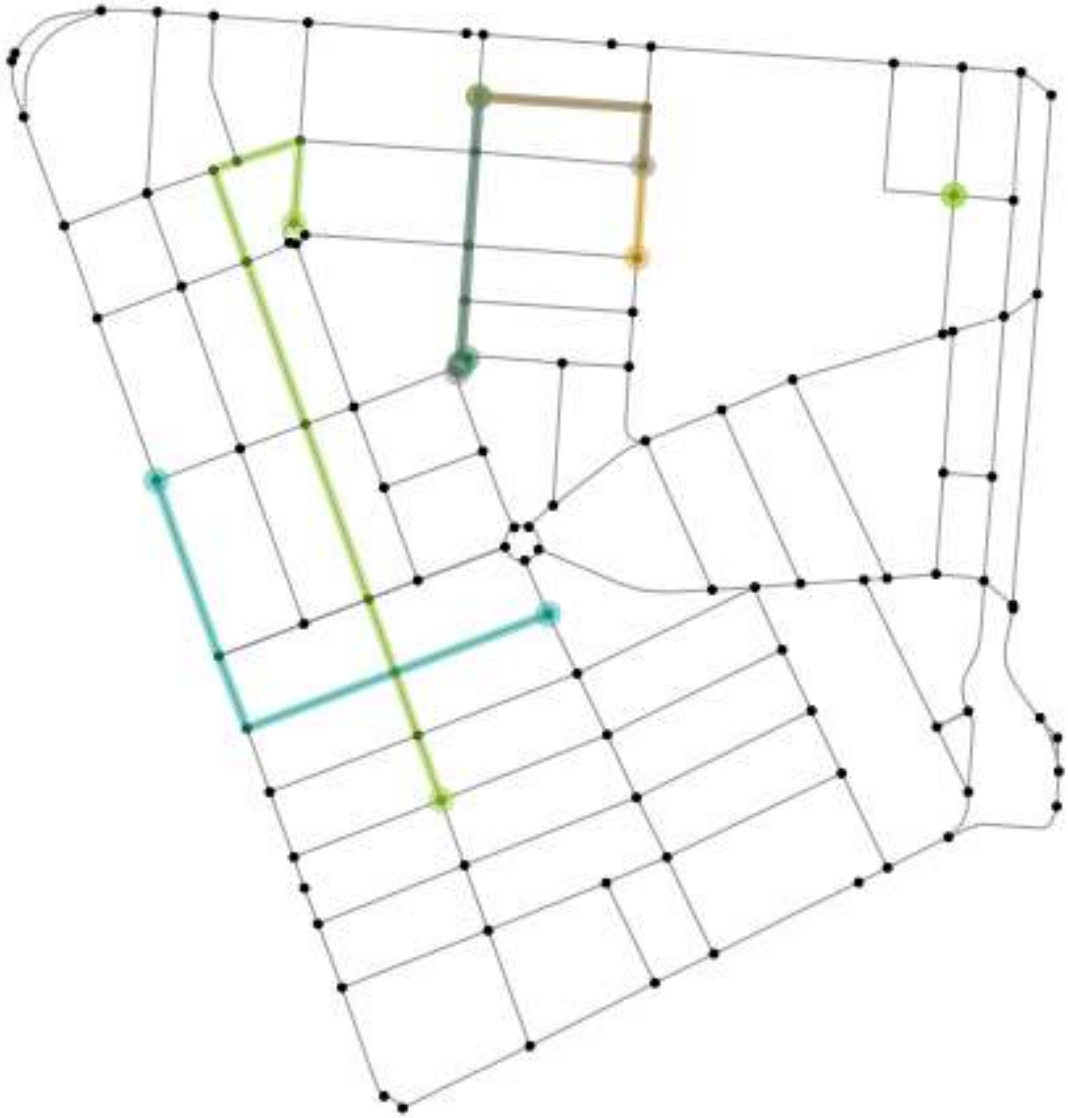
Creamos una función que nos permita graficar las rutas de un grupo.

[49]:
```
def plot_cluster(group, clusters, Graph):
    n_routes = len(clusters[group])
    print("Número de usuarios en el grupo", group, ": ", n_routes)
    if n_routes > 1:
        ox.plot.plot_graph_routes(Graph, clusters[group],␣
↪get_colors(n_routes), bgcolor="w", node_color="black")
    else:
        ox.plot.plot_graph_route(Graph, clusters[group][0],␣
↪route_color="b", bgcolor="w", node_color="black")
```

Usuarios A que no tienen asignado un vehículo.

[50]:
```
plot_cluster(-1, clusters, Graph)
```

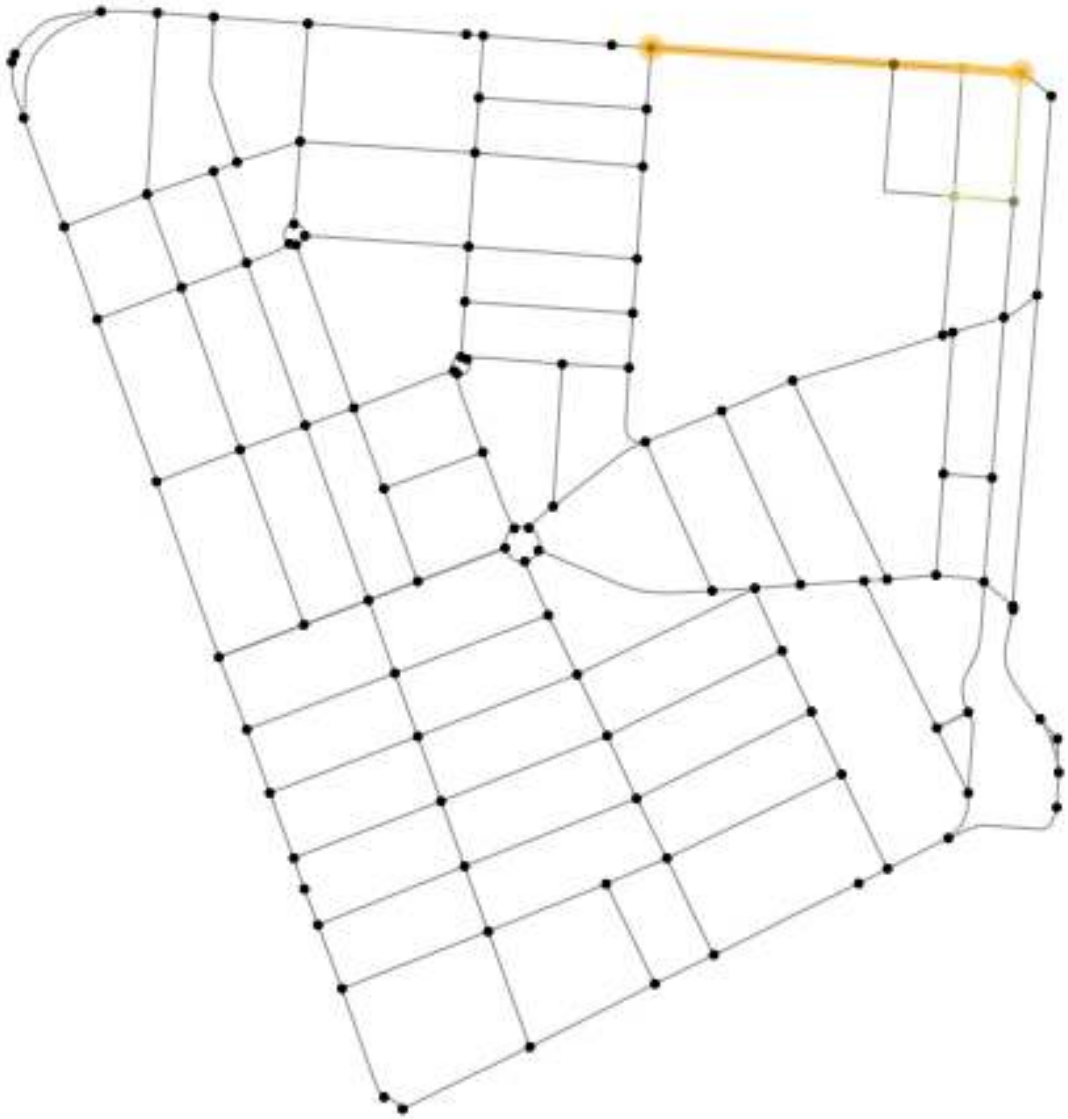
```
Número de usuarios en el grupo -1 :  6
```



Usuarios asignados al grupo 0.

[51]:
```
plot_cluster(0, clusters, Graph)
```

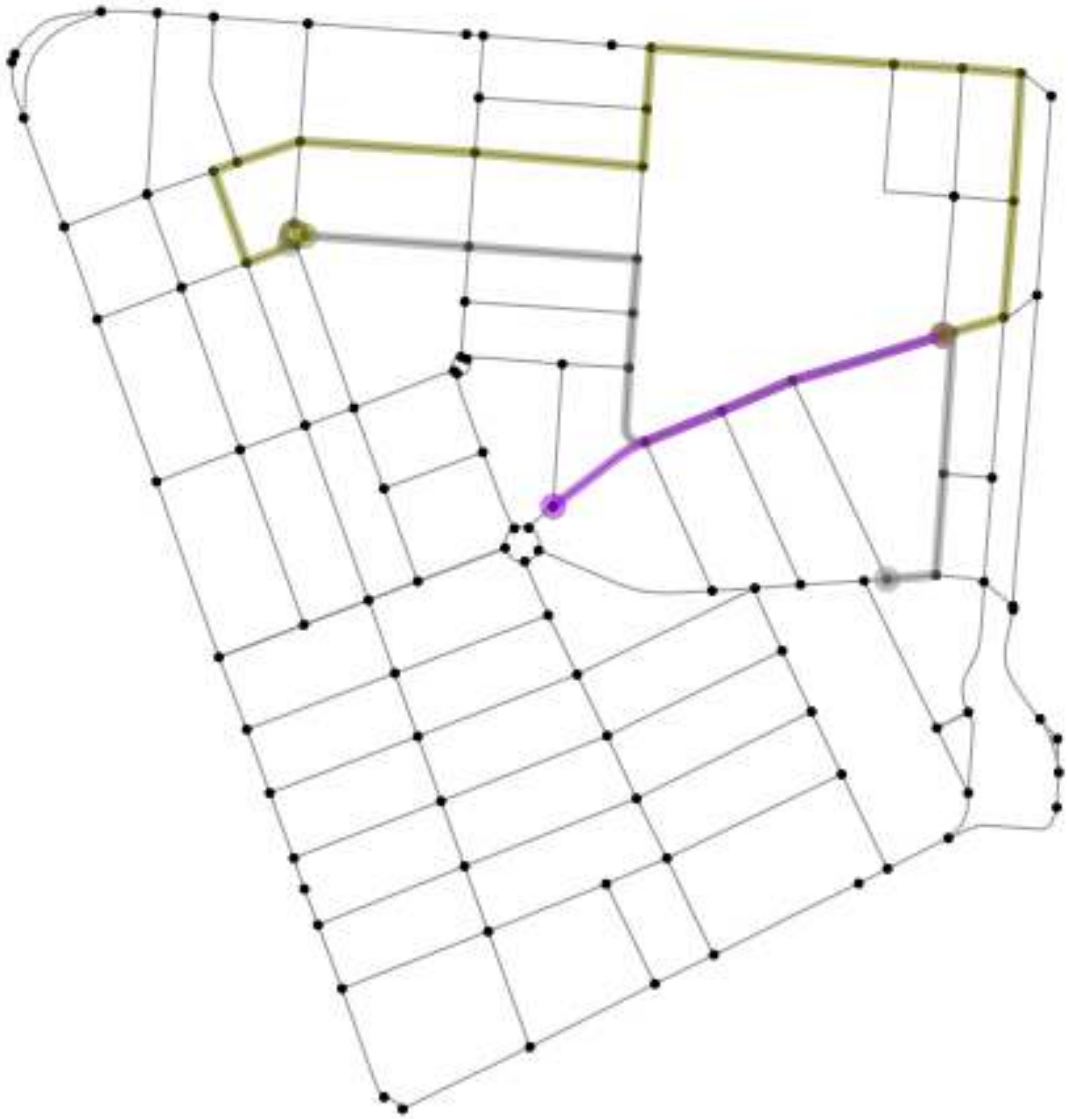
```
Número de usuarios en el grupo 0 :  2
```



Usuarios asignados al grupo 1**.**

[52]:
```
plot_cluster(1, clusters, Graph)
```

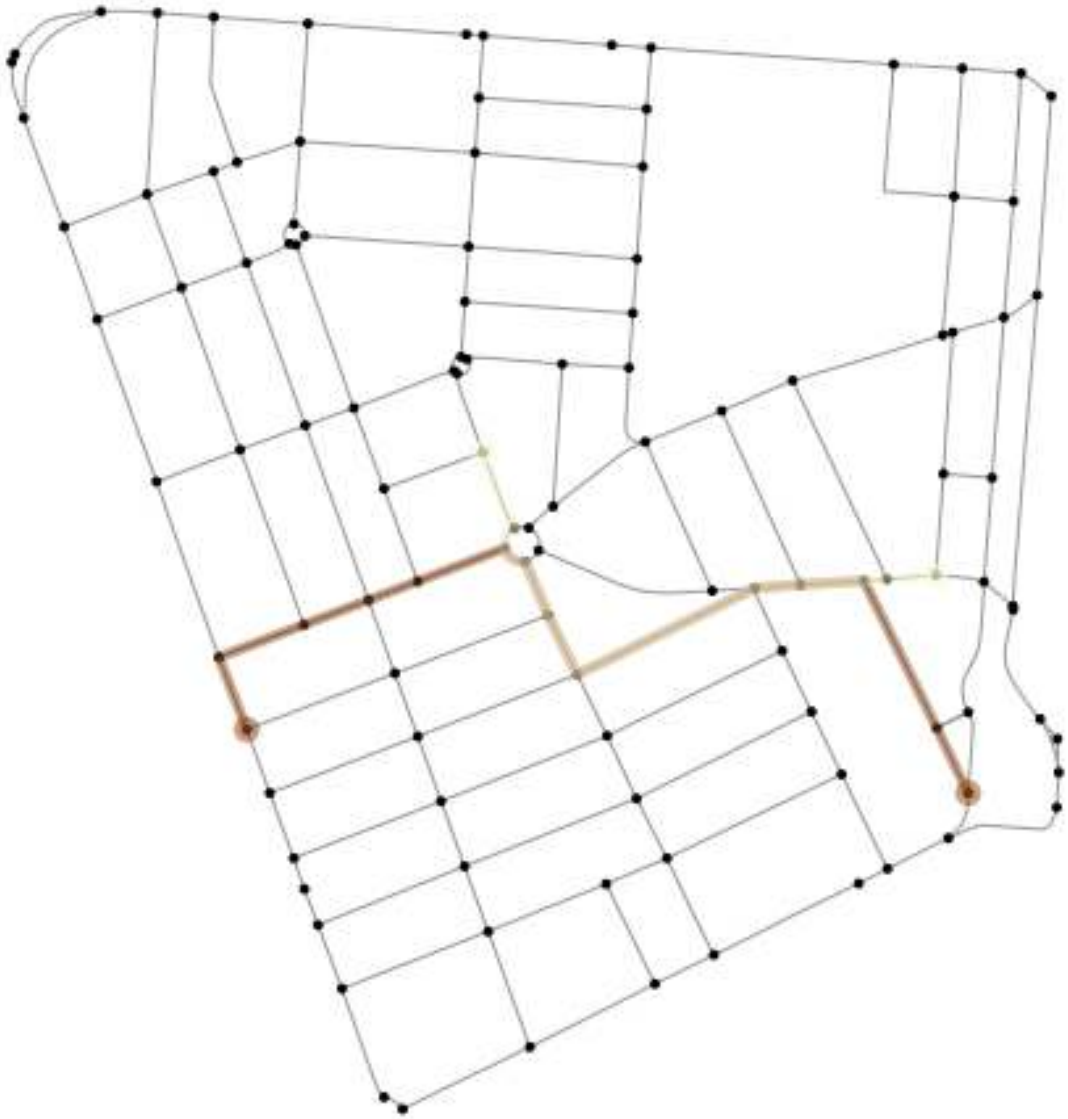
```
Número de usuarios en el grupo 1 :  3
```



Usuarios asignados al grupo 2.

[53]:
```
plot_cluster(2, clusters, Graph)
```

```
Número de usuarios en el grupo 2 :  2
```



2.7. Problema 1 - Algoritmo Evolutivo: Caso mono-objetivo

2.7.1. Representación de las soluciones

Debido a cómo trabajan los operadores de cruza y mutación para permutaciones en los AEs, veremos una solución como una permutación de los índices de una solución base. A continuación se muestra cómo crear la solución base.

[54]:
```
base_solution = initial_permutation(myUsersA, K_A, myUsersB, K_B)
print(base_solution, type(base_solution))
```

```
[ 0  1  1  2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1] <class 'numpy.ndarray'>
```

Cada individuo será una instancia de la siguiente clase:

[55]:
```
class Individual:
    global base_solution

    def __init__(self, f, permutation, myUsersA, K_A, myUsersB, Graph):
        self.permutation = permutation.copy()
        self.labels = base_solution[permutation][:K_A]
        self.f = f(self.labels, myUsersA, K_A, myUsersB, Graph)

    def __repr__(self):
        mystr = "\nIndexes: " + str(self.permutation)
        mystr += "\nG_A: " + str(self.labels)
        mystr += "\nf: " + str(self.f)
        return mystr

    def __lt__(self, other):
        if self.f < other.f:
            return True
        return False
```

Donde *permutation* es una permutación de las posiciones del arreglo, *labels* son las etiquetas asignadas dada la permutación de las posiciones y la solución base, y f es el valor en la función objetivo.

[56]:
```
np.random.seed(0)
indexes = np.array([i for i in range(len(base_solution))])
print(indexes)
np.random.shuffle(indexes)
print(indexes)
x = Individual(f1, indexes, myUsersA, K_A, myUsersB, Graph)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13]
[ 8  6  4 11  2 13  9  1  7 10  3  0  5 12]
```

[57]:
```
np.random.shuffle(indexes)
print(indexes)
y = Individual(f1, indexes, myUsersA, K_A, myUsersB, Graph)
```

```
[13  4 11  2  0  8 10 12  1  9  6  3  5  7]
```

[58]:
```
print("Solución x: ", x)
print("Solución y: ", y)
```

```
Solución x:
Indexes: [ 8  6  4 11  2 13  9  1  7 10  3  0  5 12]
G_A: [-1 -1 -1 -1  1 -1 -1  1 -1 -1]
f: 249.5
Solución y:
Indexes: [13  4 11  2  0  8 10 12  1  9  6  3  5  7]
G_A: [-1 -1 -1  1  0 -1 -1 -1  1 -1]
f: 279.5
```

2.7.2. Población inicial

La siguiente función nos permite crear una población de n soluciones.

[59]:
```
def get_initial_population(n, f, myUsersA, K_A, myUsersB, Graph):
    pop = []
    indexes = np.array([i for i in range(len(base_solution))])

    for _ in range(n):
        np.random.shuffle(indexes)
        pop.append(Individual(f, indexes, myUsersA, K_A, myUsersB, Graph))

    return pop
```

[60]:
```
pop = get_initial_population(3, f1, myUsersA, K_A, myUsersB, Graph)
for e in pop:
    print(e)
```

```
Indexes: [11  1  7 13  2 12  6  5 10  0  3  4  9  8]
G_A: [-1  1 -1 -1  1 -1 -1 -1 -1  0]
f: 215.5

Indexes: [ 4  6  2 12  7  0  3 10  1 11  5  9  8 13]
G_A: [-1 -1  1 -1 -1  0  2 -1  1 -1]
f: 301.5

Indexes: [ 9 12  3  6  1 11  8  0  7  5  4 13 10  2]
G_A: [-1 -1  2 -1  1 -1 -1  0 -1 -1]
f: 323.5
```

Dado que en el método lt de la clase Individual indicamos que los objetos de esta clase se comparan a través de su atributo f, podemos ordenar la población

con respecto al valor de la función objetivo de la siguiente forma:

[61]:
```
pop.sort()
for e in pop:
    print(e)
```

```
Indexes: [11  1  7 13  2 12  6  5 10  0  3  4  9  8]
G_A: [-1  1 -1 -1  1 -1 -1 -1 -1  0]
f: 215.5

Indexes: [ 4  6  2 12  7  0  3 10  1 11  5  9  8 13]
G_A: [-1 -1  1 -1 -1  0  2 -1  1 -1]
f: 301.5

Indexes: [ 9 12  3  6  1 11  8  0  7  5  4 13 10  2]
G_A: [-1 -1  2 -1  1 -1 -1  0 -1 -1]
f: 323.5
```

2.7.3. Operador de cruza

[62]:
```
def position_based_crossover(p1, p2, n):
    size = len(p1)
    random_positions = np.random.choice(size, n, replace=False)

    h1 = np.array([-1]*size)
    h1[random_positions] = p1[random_positions]
    copied_elements = p1[random_positions]

    index_p = 0
    index_h = 0

    while index_p < size and index_h < size:
        if(p2[index_p] not in copied_elements):
            if(h1[index_h] == -1):
                h1[index_h] = p2[index_p]
                index_p += 1
            index_h += 1
        else:
            index_p += 1

    return h1
```

[63]:
```
print("x:", x)
print("y:", y)
h = position_based_crossover(x.permutation, y.permutation, 8)
print(h)
```

```
x:
Indexes: [ 8  6  4 11  2 13  9  1  7 10  3  0  5 12]
G_A: [-1 -1 -1 -1  1 -1 -1  1 -1 -1]
f: 249.5
y:
Indexes: [13  4 11  2  0  8 10 12  1  9  6  3  5  7]
G_A: [-1 -1 -1  1  0 -1 -1 -1  1 -1]
f: 279.5
[ 8 12  4 11  2 13  9  1  6 10  3  0  5  7]
```

2.7.4. Operador de mutación

[64]:
```
def mutation(p, n):
    new_p = p.copy()
    for _ in range(n):
        positions = np.random.choice(len(G_A), 2, replace=False)
        new_p[positions[0]], new_p[positions[1]] = new_p[positions[1]],␣
 ↪new_p[positions[0]]

    return new_p
```

[65]:
```
mutation(h,1)
```

[65]:
```
array([ 8, 12,  4, 11,  2, 13,  1,  9,  6, 10,  3,  0,  5,  7])
```

2.7.5. Selección de padres y sobrevivientes

Para este problema vamos a elegir las soluciones padres de manera aleatoria y los sobrevivientes se elegirán haciendo una selección +. Es decir, se unirán las poblaciones de padres e hijos y se seleccionarán a las mejores soluciones para pasar a la siguiente iteración.

2.7.6. Algoritmo completo

[66]:
```
def AE(num_generations, size_population, prob_crossover, prob_mutation,␣
 ↪f, n_positions, n_swaps,  myUsersA, K_A, myUsersB, K_B):
    global base_solution
    base_solution = initial_permutation(myUsersA, K_A, myUsersB, K_B)
```

```
    current_pop = get_initial_population(size_population, f, myUsersA,␣
↪K_A, myUsersB, Graph)

    for _ in range(num_generations):
        new_pop = []

        while len(new_pop) < size_population:
            p1, p2 = np.random.choice(size_population, 2, replace=False)
            if np.random.uniform(0,1) < prob_crossover:
                h_permutation = position_based_crossover(current_pop[p1].
↪permutation, current_pop[p2].permutation, n_positions)
            else:
                h_permutation = current_pop[p1].permutation.copy()

            if np.random.uniform(0,1) < prob_mutation:
                h_permutation = mutation(h_permutation, n_swaps)


            new_pop.append(Individual(f, h_permutation, myUsersA, K_A,␣
↪myUsersB, Graph))

        current_pop += new_pop
        current_pop.sort()
        current_pop = current_pop[:size_population]

    return current_pop[0].labels, current_pop[0].f
```

[67]:
```
AE(num_generations=100, size_population=100, prob_crossover=0.9,␣
↪prob_mutation=0.1, f=f1, n_positions=K_A//2, n_swaps=1, ␣
↪myUsersA=myUsersA, K_A=K_A, myUsersB=myUsersB, K_B=K_B)
```

[67]:
```
(array([ 0, -1, -1,  2, -1,  1, -1, -1, -1,  1]), 160.0)
```

2.7.7. Estadísticas

Al igual que en RS, evaluaremos el comportamiento de la metaheurística realizando M ejecuciones con los mismos parámetros de entrada.

[68]:
```
def get_statistics_AE(M, num_generations, size_population,␣
↪prob_crossover, prob_mutation, f, n_positions, n_swaps,  myUsersA, K_A,␣
↪myUsersB, K_B):
    sols = []
    for _ in range(M):
```

```
        sol = AE(num_generations, size_population, prob_crossover,␣
↪prob_mutation, f, n_positions, n_swaps,  myUsersA, K_A, myUsersB, K_B)
        sols.append(sol)

    sols.sort(key = lambda x: x[1])
    best = sols[0]
    worst = sols[-1]
    median = sols[M//2]
    fmean = np.mean([x[1] for x in sols])
    fstd = np.std([x[1] for x in sols])

    return best, worst, median, fmean, fstd
```

A continuación ejecutamos la metaheurística 100 veces utilizando 100 generaciones, un tamaño de población de 100, K_A//2 posiciones fijas para el operador de cruza, un intercambio en el operador de mutación, una probabilidad de cruza de 0.9 y una probabilidad de mutación de 0.1.

[69]:
```
M = 100
AE_results = get_statistics_AE(M, 100, 100, 0.9, 0.1, f1, K_A//2, 1,␣
 ↪myUsersA, K_A, myUsersB, K_B)
```

[70]:
```
results = pd.DataFrame()
results.index = ["AE"]
results["x_best"] = [AE_results[0][0]]
results["f_best"] = [AE_results[0][1]]

results["x_worst"] = [AE_results[1][0]]
results["f_worst"] = [AE_results[1][1]]

results["x_median"] = [AE_results[2][0]]
results["f_median"] = [AE_results[2][1]]

results["f_mean"] = [AE_results[3]]
results["f_std"] = [AE_results[4]]
results
```

[70]:

	x_best	f_best	\
AE	[0, -1, -1, 2, -1, 1, -1, -1, -1, 1]	160.0	

	x_worst	f_worst	\
AE	[-1, -1, -1, 2, -1, 1, -1, -1, -1, 1]	162.5	

	x_median	f_median	f_mean	f_std

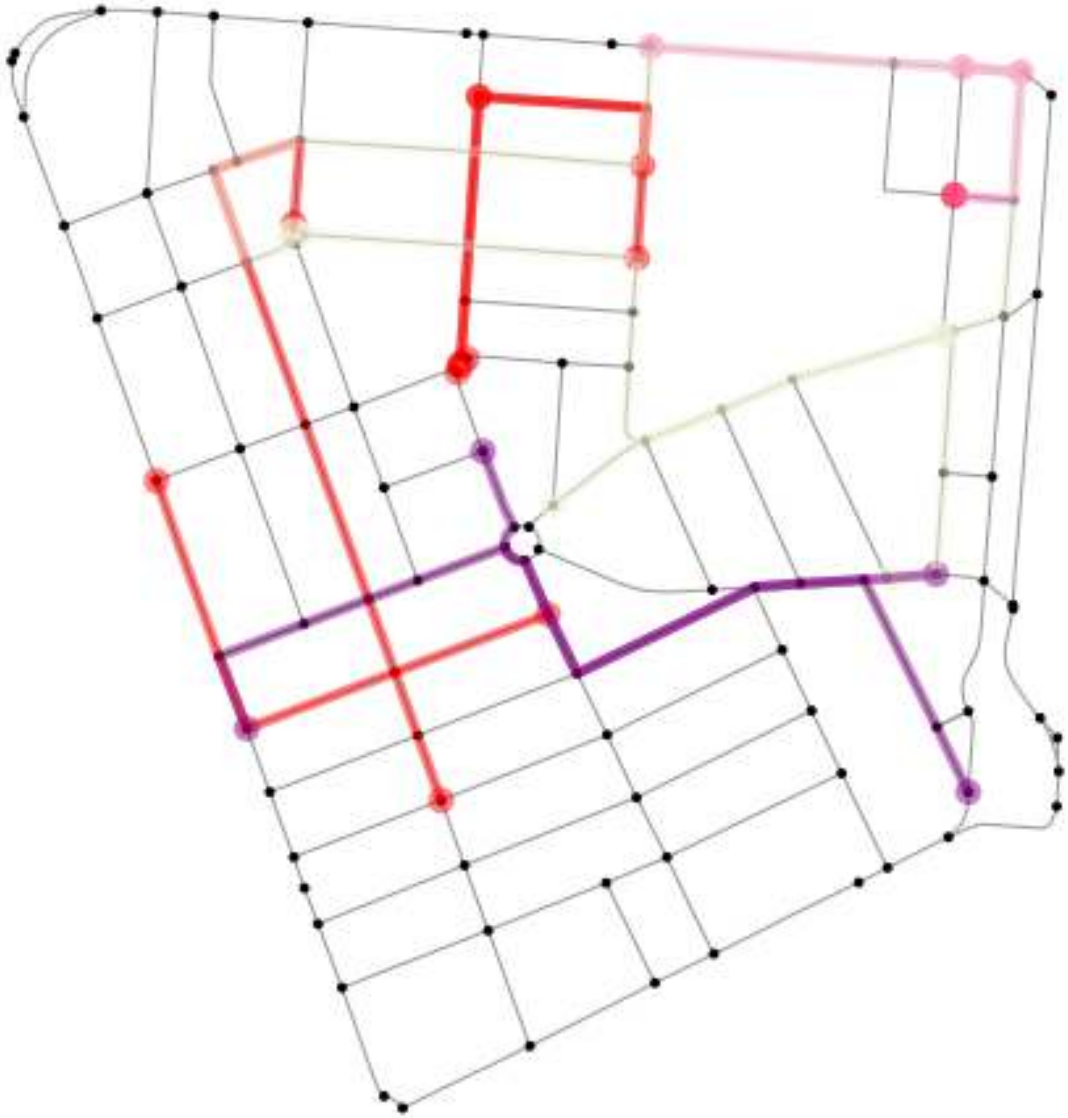
```
AE  [0, -1, -1, 2, -1, 1, -1, -1, -1, 1]     160.0  160.175  0.637868
```

Utilizamos la mejor solución para agrupar las rutas según las etiquetas asignadas.

[71]:
```
best = AE_results[0][0]
clusters = create_groups(best, myUsersA, K_A, myUsersB, K_B)
```

Graficamos los grupos: Las rutas pertenecientes a un grupo tienen el mismo color.

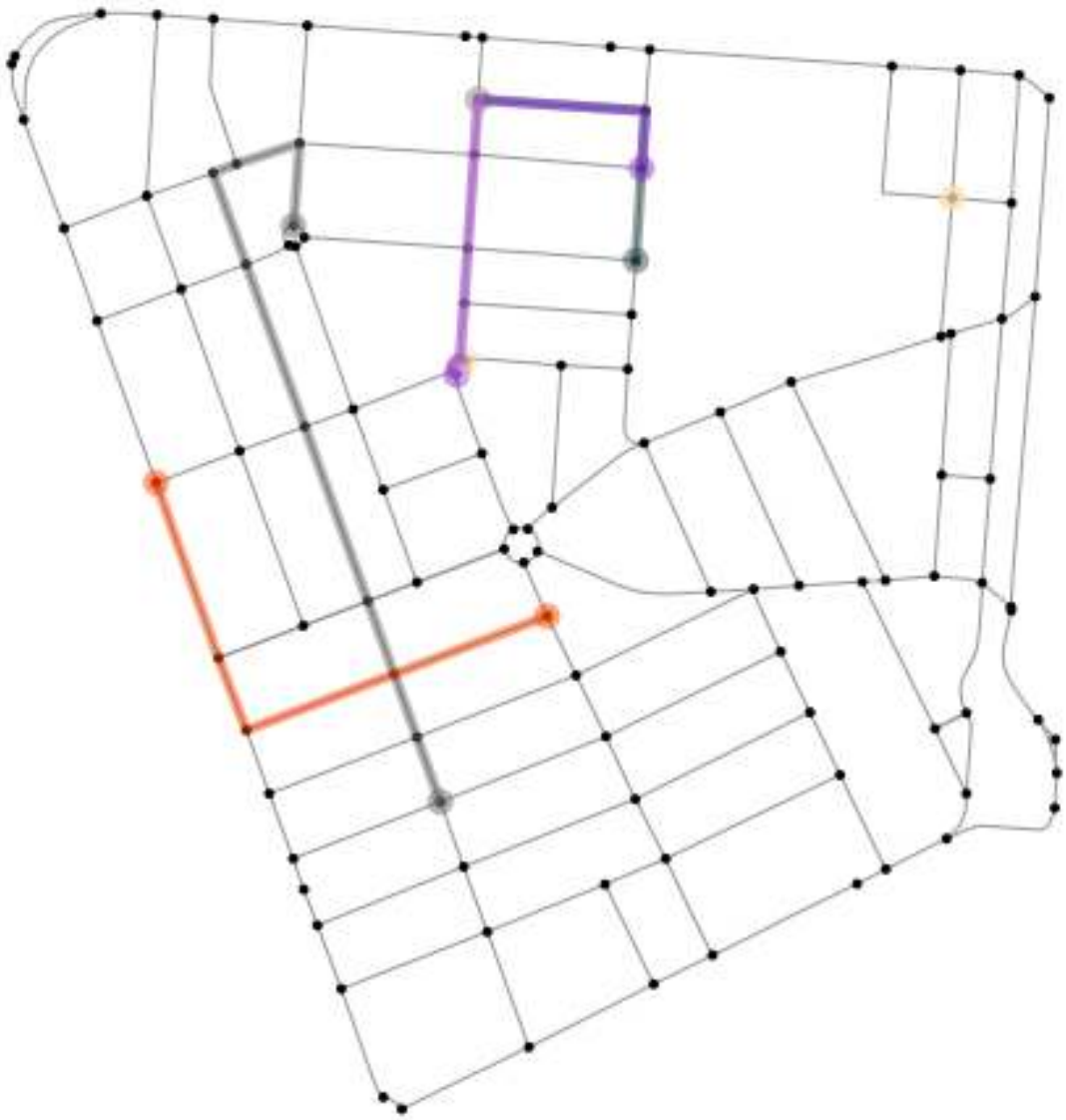
[72]:
```
plot_clustered_routes(clusters, get_colors(K_B), Graph)
```



Usuarios A que no tienen asignado un vehículo.

[73]:
```
plot_cluster(-1, clusters, Graph)
```

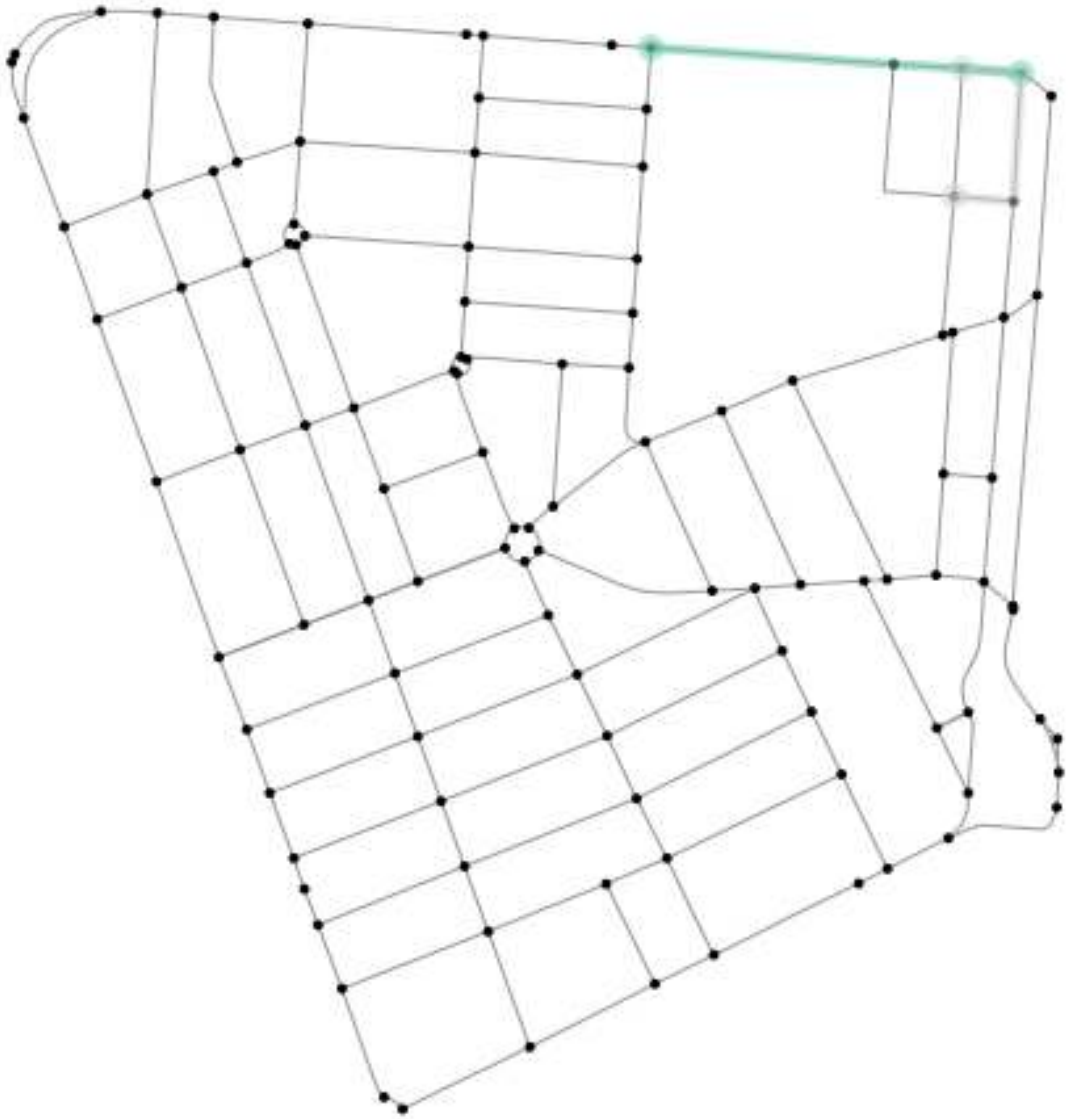
```
Número de usuarios en el grupo -1 :  6
```



Usuarios asignados al grupo 0.

[74]:
```
plot_cluster(0, clusters, Graph)
```

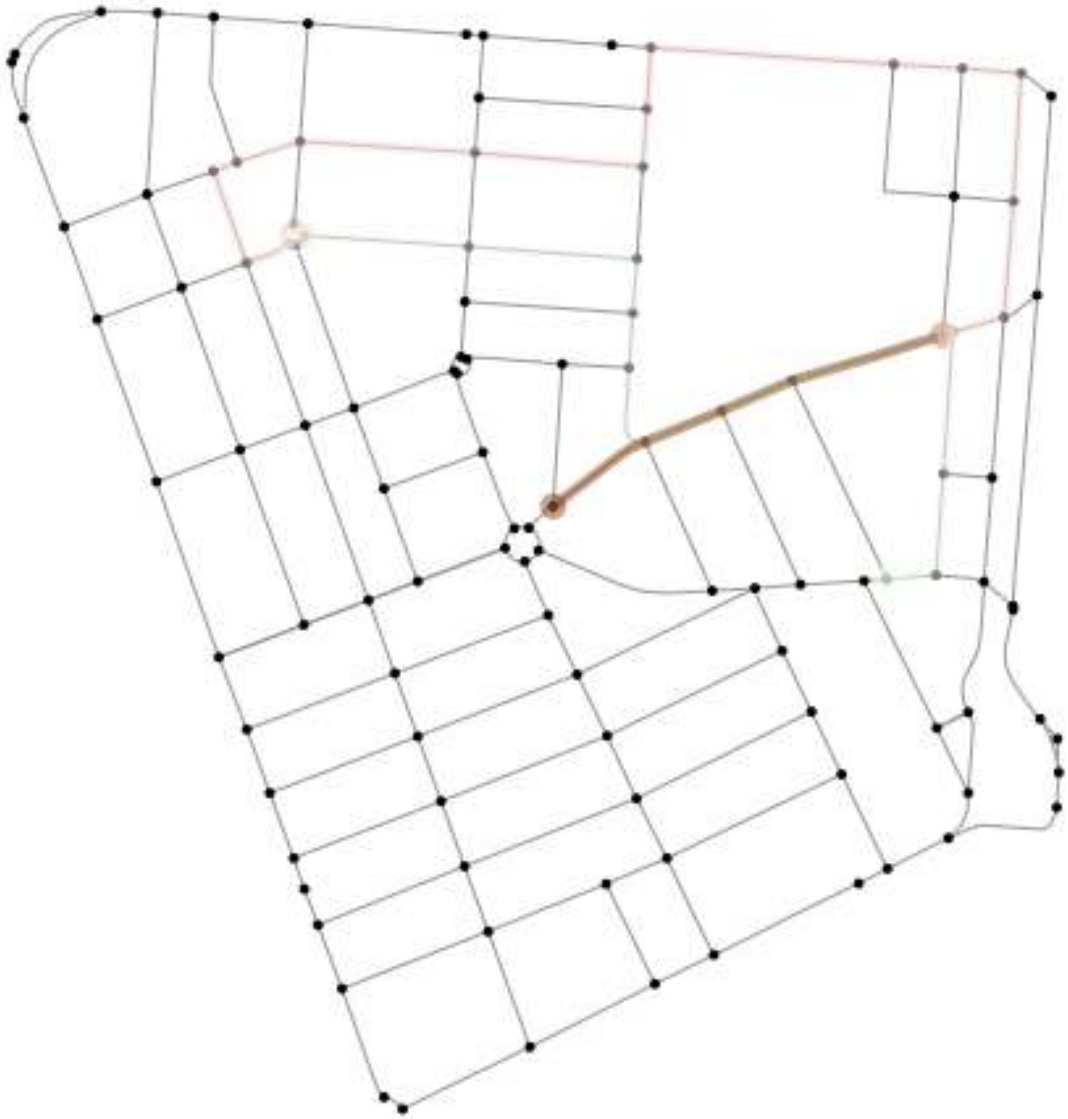
```
Número de usuarios en el grupo 0 :  2
```



Usuarios asignados al grupo 1**.**

[75]:
```
plot_cluster(1, clusters, Graph)
```

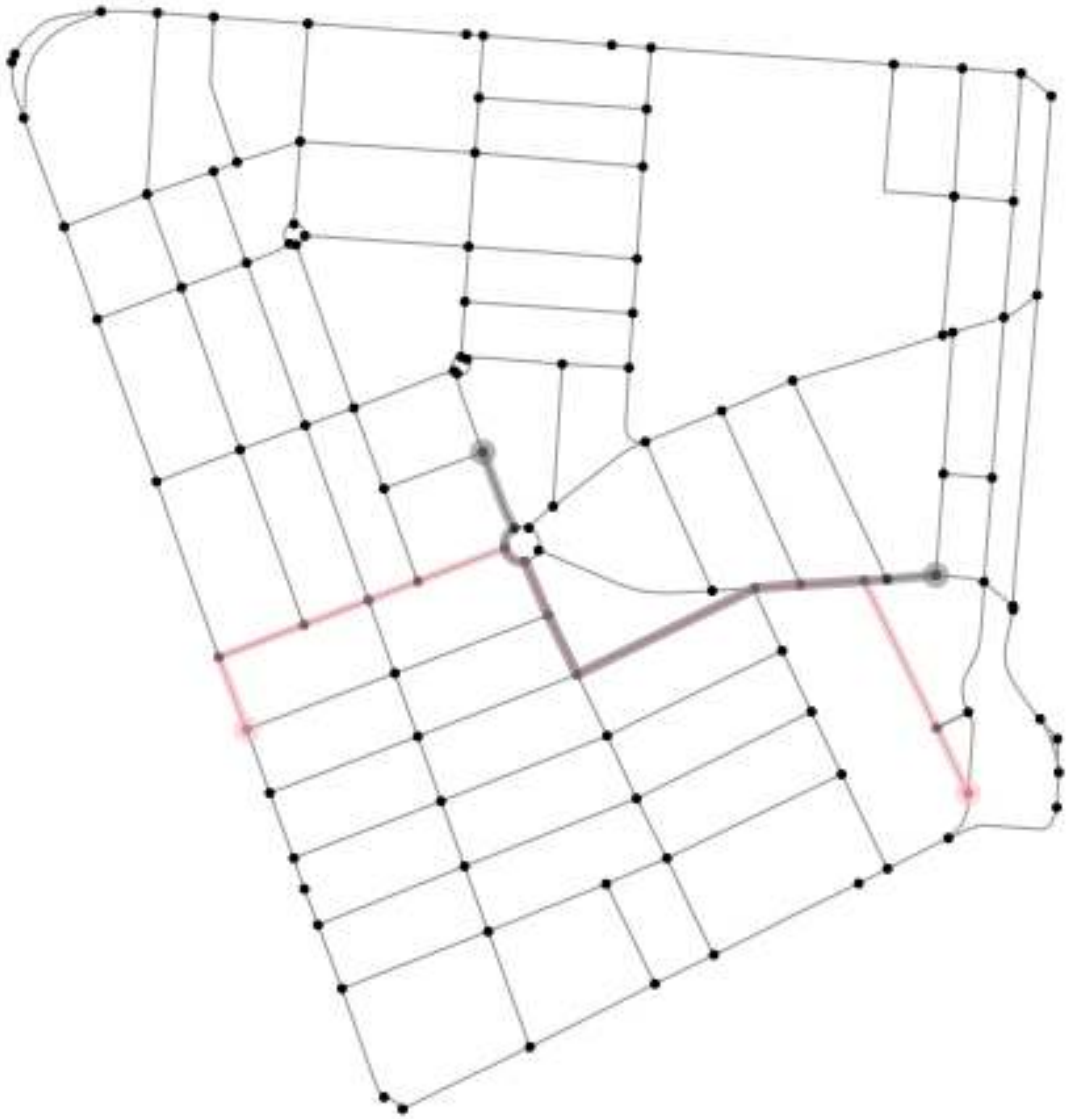
```
Número de usuarios en el grupo 1 :  3
```



Usuarios asignados al grupo 2.

[76]:
```
plot_cluster(2, clusters, Graph)
```

```
Número de usuarios en el grupo 2 :  2
```



Ejercicio 1: Generar una nueva instancia del problema, utilizando una ciudad que tenga un mayor número de nodos, generar 100 usuarios de tipo A y 20 usuarios de tipo B.

Ejercicio 2: Utilizar las metaheurísticas diseñadas para resolver el problema con la nueva instancia. Deberá presentar una tabla comparativa de ambas metaheurísticas considerando al menos tres conjuntos de parámetros diferentes.

Ejercicio 3: Graficar los grupos de la mejor solución encontrada por RS.

Ejercicio 4: Graficar los grupos de la mejor solución encontrada por el AE.

2.8. Problema 2 - Optimización multi-objetivo

A continuación definimos una función que nos permite verificar la dominancia entre dos soluciones. Dicha función regresa 1 si la solución x domina a la solución y, 0, si la solución y domina a la solución x y -1 si ambas soluciones son no dominadas entre sí.

[77]:
```
def dominates(x: np.array, y: np.array) -> int:
    n = len(x)

    #Si los puntos son iguales
    if (x==y).all():
        return -1

    #Si x domina a y
    if sum(x<=y) == n:
        return 1
    #Si y domina a x
    elif sum(y<=x) == n:
        return 0

    #Si son no dominadas entre si
    return -1
```

En el siguiente ejemplo las soluciones $x = [1, 2]$ y $y = [2, 1]$ son no dominadas entre sí porque x es mejor que y en la primera componente pero es peor en la segunda.

[78]:
```
dominates(np.array([1,2]), np.array([2,1]))
```

[78]:
```
-1
```

En el siguiente ejemplo las solución $x = [1, 2]$ domina a la solución $y = [1, 3]$ porque es mejor en ambas componentes.

[79]:
```
dominates(np.array([1,2]), np.array([1,3]))
```

[79]:
```
1
```

En el siguiente ejemplo las solución $x = [1, 2]$ es dominada por la solución $y = [0.5, 1]$ porque es peor en ambas componentes.

[80]:
```
dominates(np.array([1,2]), np.array([1,1]))
```

[80]:
```
0
```

2.9. Problema 2 - Algoritmo Evolutivo: Caso multi-objetivo

2.9.1. Funciones objetivo

La función f_1 es la misma que se utilizó en el caso mono-objetivo. La función f_2 regresa la cantidad de usuarios A que no tienen asignados un vehículo:

[81]:
```
def f2(G_A, K_A):
    total = 0
    for i in range(K_A):
        if G_A[i] == -1:
            total += 1
    return total
```

A continuación definimos el problema multi-objetivo F:

[82]:
```
def F(G_A, myUsersA, K_A, myUsersB, Graph):
    return np.array([f1(G_A, myUsersA, K_A, myUsersB, Graph), f2(G_A,␣
 ↪K_A)])
```

2.9.2. Representación de las soluciones

En el caso multi-objetivo utilizaremos la misma representación que en el caso mono-objetivo pero no definiremos el método lt en la clase *Individual*.

[83]:
```
class Individual:
    global base_solution

    def __init__(self, f, permutation, myUsersA, K_A, myUsersB, Graph):
        self.permutation = indexes.copy()
        self.labels = base_solution[permutation][:K_A]
        self.f = f(self.labels, myUsersA, K_A, myUsersB, Graph)

    def __repr__(self):
        mystr = "\nIndexes: " + str(self.permutation)
        mystr += "\nG_A: " + str(self.labels)
        mystr += "\nf: " + str(self.f)
        return mystr
```

2.9.3. Selección de sobrevivientes

A continuación implementaremos un método conocido como actualización continua que permite seleccionar el conjunto de soluciones no dominadas de un

conjunto de soluciones. La función regresa tanto el conjunto de soluciones dominadas como el de soluciones no dominadas.

[84]:
```
def get_non_dominated_solutions(P):
    non_dom = [0]

    for i in range(1, len(P)):
        j = 0
        i_is_non_dominated = True
        while j < len(non_dom):
            d = dominates(P[i].f, P[non_dom[j]].f)
            if d == 1:
                non_dom.pop(j)
            elif d == 0:
                i_is_non_dominated = False
                break
            else:
                j += 1

        if i_is_non_dominated:
            non_dom.append(i)

    new_P = []
    P_i = []
    for i in range(len(P)):
        if i in non_dom:
            P_i.append(P[i])
        else:
            new_P.append(P[i])

    return new_P, P_i
```

Creamos un conjunto de soluciones:

[85]:
```
np.random.seed(0)
pop = get_initial_population(5, F, myUsersA, K_A, myUsersB, Graph)
for e in pop:
    print(e)
```

```
Indexes: [13  4 11  2  0  8 10 12  1  9  6  3  5  7]
G_A: [-1 -1 -1 -1  1 -1 -1  1 -1 -1]
f: [249.5   8. ]

Indexes: [13  4 11  2  0  8 10 12  1  9  6  3  5  7]
```

```
G_A: [-1 -1 -1  1  0 -1 -1 -1  1 -1]
f: [279.5   7. ]

Indexes: [13  4 11  2  0  8 10 12  1  9  6  3  5  7]
G_A: [ 2 -1 -1 -1 -1 -1 -1 -1 -1 -1]
f: [284.5   9. ]

Indexes: [13  4 11  2  0  8 10 12  1  9  6  3  5  7]
G_A: [ 0 -1 -1 -1 -1 -1  1 -1 -1  2]
f: [233.   7.]

Indexes: [13  4 11  2  0  8 10 12  1  9  6  3  5  7]
G_A: [-1 -1  1 -1 -1  2  1 -1 -1 -1]
f: [264.5   7. ]
```

Obtenemos las soluciones dominadas y no dominadas:

[86]:
```
dom, non_dom = get_non_dominated_solutions(pop)
```

Soluciones dominadas:

[87]:
```
for e in dom:
    print(e)
```

```
Indexes: [13  4 11  2  0  8 10 12  1  9  6  3  5  7]
G_A: [-1 -1 -1 -1  1 -1 -1  1 -1 -1]
f: [249.5   8. ]

Indexes: [13  4 11  2  0  8 10 12  1  9  6  3  5  7]
G_A: [-1 -1 -1  1  0 -1 -1 -1  1 -1]
f: [279.5   7. ]

Indexes: [13  4 11  2  0  8 10 12  1  9  6  3  5  7]
G_A: [ 2 -1 -1 -1 -1 -1 -1 -1 -1 -1]
f: [284.5   9. ]

Indexes: [13  4 11  2  0  8 10 12  1  9  6  3  5  7]
G_A: [-1 -1  1 -1 -1  2  1 -1 -1 -1]
f: [264.5   7. ]
```

Soluciones no dominadas:

[88]:
```
for e in non_dom:
    print(e)
```

```
Indexes: [13  4 11  2  0  8 10 12  1  9  6  3  5  7]
G_A: [ 0 -1 -1 -1 -1 -1  1 -1 -1  2]
f: [233.   7.]
```

Finalmente creamos una función que nos permita seleccionar m soluciones de un conjunto de n soluciones:

[89]:
```
def get_survivors(pop, m):
    survivors = []

    while len(survivors) < m:
        dom, non_dom = get_non_dominated_solutions(pop)
        if len(survivors) + len(non_dom) <= m:
            survivors += non_dom
        else:
            remaining = m - len(survivors)
            survivors += list(np.random.choice(non_dom, remaining,␣
↪replace=False))

    return survivors
```

2.9.4. Algoritmo completo

[90]:
```
def AEMO(num_generations, size_population, prob_crossover, prob_mutation,␣
↪f, n_positions, n_swaps,  myUsersA, K_A, myUsersB, K_B):
    global base_solution
    base_solution = initial_permutation(myUsersA, K_A, myUsersB, K_B)
    current_pop = get_initial_population(size_population, f, myUsersA,␣
↪K_A, myUsersB, Graph)

    for _ in range(num_generations):
        new_pop = []

        while len(new_pop) < size_population:
            p1, p2 = np.random.choice(size_population, 2, replace=False)
            if np.random.uniform(0,1) < prob_crossover:
                h_permutation = position_based_crossover(current_pop[p1].
↪permutation, current_pop[p2].permutation, n_positions)
                if np.random.uniform(0,1) < prob_mutation:
                    h_permutation = mutation(h_permutation, n_swaps)
            else:
                h_permutation = current_pop[p1].permutation.copy()
```

```
            new_pop.append(Individual(f, h_permutation, myUsersA, K_A,␣
 ↪myUsersB, Graph))

        current_pop = get_survivors(current_pop+new_pop, size_population)


    dom, non_dom = get_non_dominated_solutions(current_pop)
    return non_dom
```

[91]:
```
aprox_Pareto = AEMO(num_generations=100, size_population=100,␣
 ↪prob_crossover=0.9, prob_mutation=0.1, f=F, n_positions=K_A//2,␣
 ↪n_swaps=1,  myUsersA=myUsersA, K_A=K_A, myUsersB=myUsersB, K_B=K_B)
```

[92]:
```
for e in aprox_Pareto[:5]:
    print(e)
```

```
Indexes: [13  4 11  2  0  8 10 12  1  9  6  3  5  7]
G_A: [ 0 -1 -1 -1  2 -1 -1 -1 -1  1]
f: [184.   7.]

Indexes: [13  4 11  2  0  8 10 12  1  9  6  3  5  7]
G_A: [-1 -1  1  2 -1 -1 -1 -1  1  0]
f: [207.5   6. ]

Indexes: [13  4 11  2  0  8 10 12  1  9  6  3  5  7]
G_A: [-1 -1 -1  2 -1 -1 -1 -1 -1  1]
f: [170.   8.]

Indexes: [13  4 11  2  0  8 10 12  1  9  6  3  5  7]
G_A: [ 0 -1 -1 -1  2 -1 -1 -1 -1  1]
f: [184.   7.]

Indexes: [13  4 11  2  0  8 10 12  1  9  6  3  5  7]
G_A: [-1 -1  1  2 -1 -1 -1 -1  1  0]
f: [207.5   6. ]
```

2.9.5. Gráfica de la aproximación del frente de Pareto

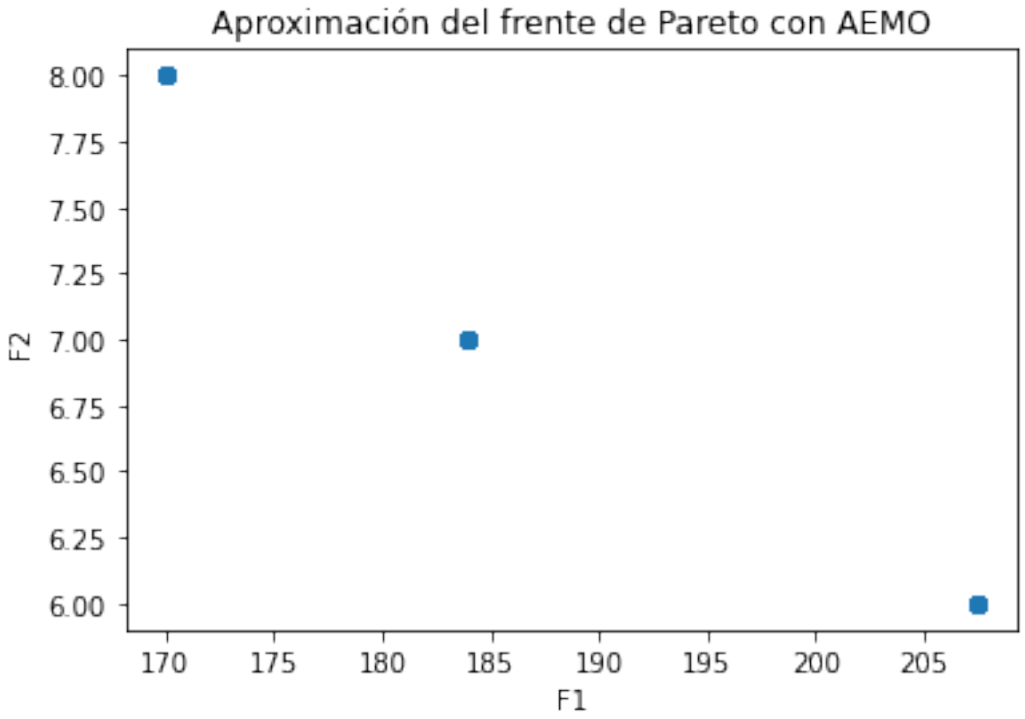
Para graficar la aproximación del frente de Pareto extraemos los vectores objetivo de una población.

[93]:
```
def get_objective_values(pop):
    return np.array([e.f for e in pop])
```

La siguiente función genera la gráfica del frente de Pareto.

[94]:
```
def plot_pareto_front(PF:np.array, alg:str):
    plt.scatter(PF[:,0],PF[:,1])
    plt.title(f"Aproximación del frente de Pareto con {alg}")
    plt.xlabel("F1")
    plt.ylabel("F2")
    plt.show()
```

[95]:
```
PF = get_objective_values(aprox_Pareto)
plot_pareto_front(PF, "AEMO")
```



2.9.6. Indicador de hipervolumen

Con la siguiente instrucción, importamos el indicador de hipervolumen que está implementado en el módulo pymoo.

[96]:
```
from pymoo.indicators.hv import HV
```

Generamos una solución que sea dominada por todas las soluciones del frente de Pareto:

[97]:
```
reference_point = np.amax(PF, axis=0)+0.1
reference_point
```

[97]:
```
array([207.6,   8.1])
```

Creamos un objeto de tipo *HV* que utiliza como punto de referencia el punto creado *reference_point*:

[98]:
```
indicator = HV(ref_point = reference_point)
```

Finalmente, calculamos el valor del hipervolumen para el frente de Pareto encontrado:

[99]:
```
print("Hypervolume: ", indicator(PF))
```

```
Hypervolume:  27.459999999999976
```

2.9.7. Estadísticas

Al igual que en el caso mono-objetivo, evaluaremos el comportamiento de la metaheurística realizando M ejecuciones con los mismos parámetros de entrada. Para evaluar cada ejecución se utilizará el indicador de hipervolumen y el punto de referencia debe ser dominado por todos los puntos de los frentes encontrados en cada ejecución.

[100]:
```
def get_reference_point(PFs):
    ref_points = []
    for pf in PFs:
        ref_points.append(np.amax(PF, axis=0))
    ref_points = np.array(ref_points)

    return np.amax(ref_points, axis=0)+0.1
```

[101]:
```
def get_statistics_AEMO(M, num_generations, size_population,␣
↪prob_crossover, prob_mutation, f, n_positions, n_swaps,  myUsersA, K_A,␣
↪myUsersB, K_B):
    PFs = []
    sols = []
    for _ in range(M):
        sol = AEMO(num_generations, size_population, prob_crossover,␣
↪prob_mutation, f, n_positions, n_swaps,  myUsersA, K_A, myUsersB, K_B)
        sols.append([sol])
        PF = get_objective_values(sol)
        PFs.append(PF)
```

```
    ref_point = get_reference_point(PFs)
    indicator = HV(ref_point = ref_point)
    for i in range(M):
        sols[i].append(indicator(PFs[i]))

    sols.sort(key = lambda x: x[1])
    best = sols[-1]
    worst = sols[0]
    median = sols[M//2]
    HVmean = np.mean([x[1] for x in sols])
    HVstd = np.std([x[1] for x in sols])

    return best, worst, median, HVmean, HVstd
```

A continuación ejecutamos la metaheurística 100 veces utilizando 100 generaciones, un tamaño de población de 100, K_A//2 posiciones fijas para el operador de cruza, un intercambio en el operador de mutación, una probabilidad de cruza de 0.9 y una probabilidad de mutación de 0.1.

[102]:
```
AE_results = get_statistics_AEMO(M=100, num_generations=100,␣
↪size_population=100, prob_crossover=0.9, prob_mutation=0.1, f=F,␣
↪n_positions=K_A//2, n_swaps=1,  myUsersA=myUsersA, K_A=K_A,␣
↪myUsersB=myUsersB, K_B=K_B)
```

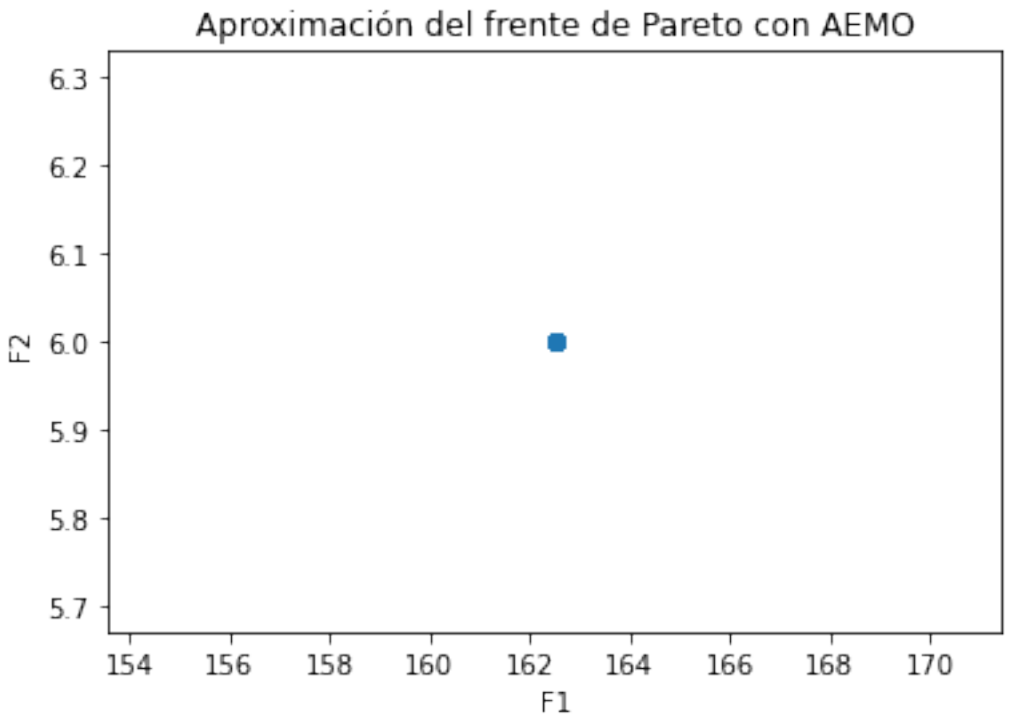
[103]:
```
results = pd.DataFrame()
results.index = ["AEMO"]
results["HV_best"] = [AE_results[0][1]]
results["HV_worst"] = [AE_results[1][1]]
results["HV_median"] = [AE_results[2][1]]
results["HV_mean"] = [AE_results[3]]
results["HV_std"] = [AE_results[4]]
results
```

[103]:

	HV_best	HV_worst	HV_median	HV_mean	HV_std
AEMO	94.71	3.76	41.36	41.6145	19.757017

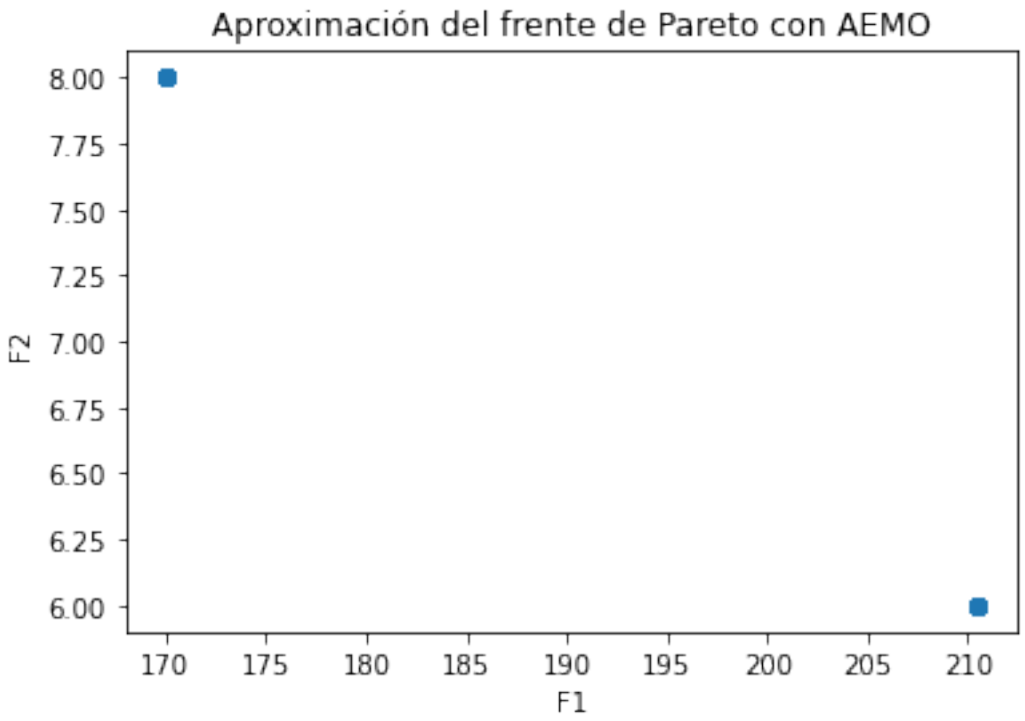
Graficamos el frente de Pareto que logra obtener un mejor hipervolumen.

[104]:
```
best_PF = get_objective_values(AE_results[0][0])
plot_pareto_front(best_PF, "AEMO")
```



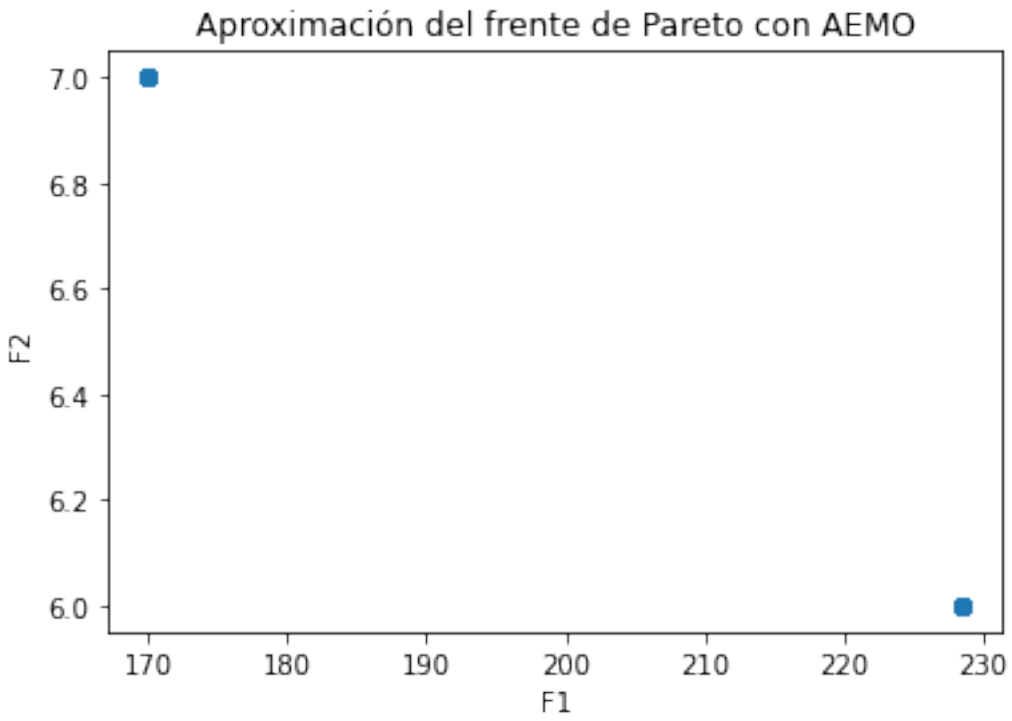
Graficamos el frente de Pareto que logra obtener un peor hipervolumen.

[105]:
```
worst_PF = get_objective_values(AE_results[1][0])
plot_pareto_front(worst_PF, "AEMO")
```



Graficamos el frente de Pareto que está en la mediana, considerando el indicador de hipervolumen.

[106]:
```
median_PF = get_objective_values(AE_results[2][0])
plot_pareto_front(median_PF, "AEMO")
```



2.9.8. Grupos creados

Utilizamos el frente de Pareto correspondiente al mejor valor en el indicador de hipervolumen y buscamos la solución del frente que logra dejar la mejor cantidad de usuarios tipo A sin vehículo.

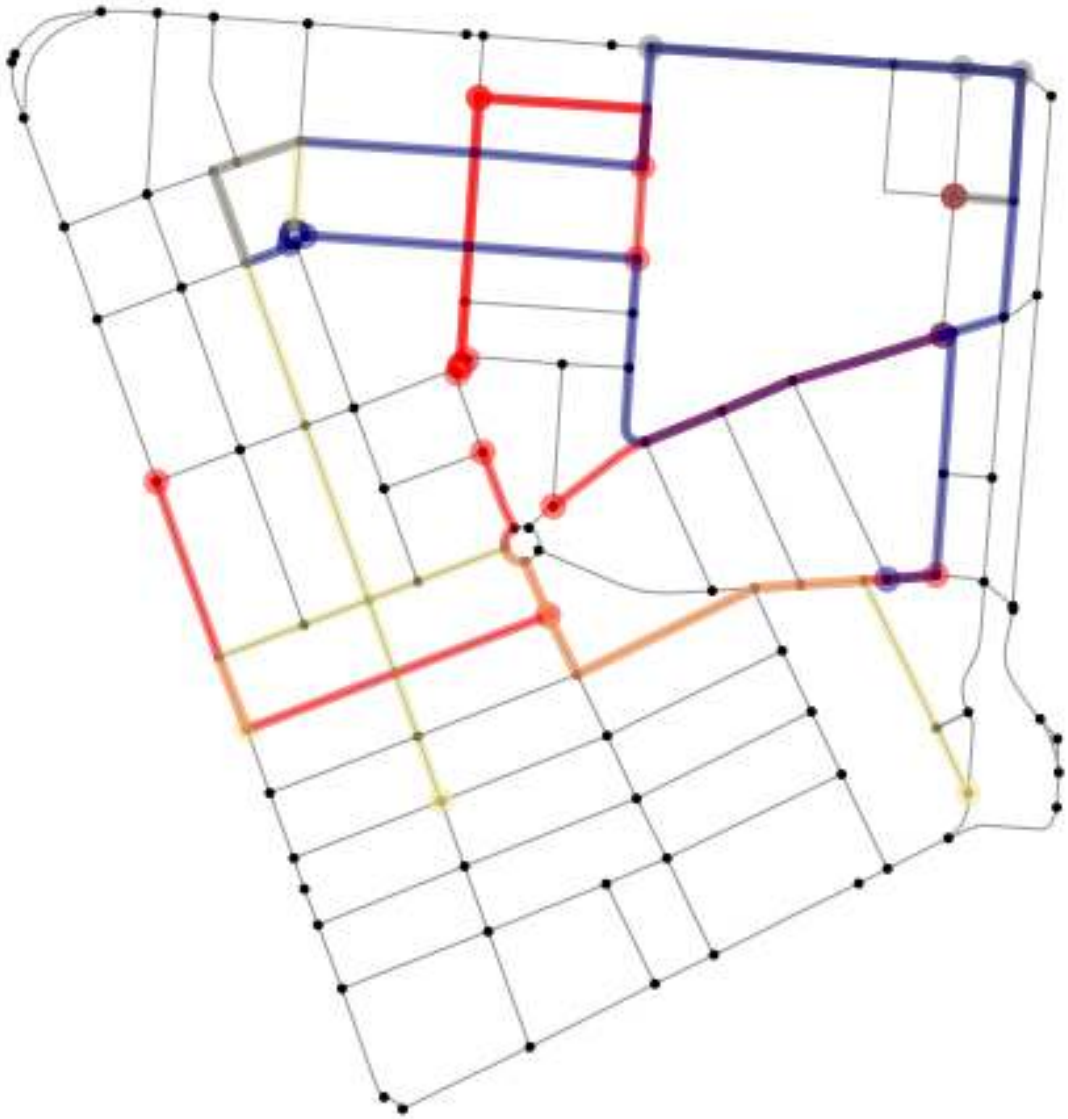
[107]:
```
index = np.argmin(best_PF[:,1])
```

Obtenemos los grupos:

[108]:
```
G_A = aprox_Pareto[index].labels
clusters = create_groups(G_A, myUsersA, K_A, myUsersB, K_B)
```

Graficamos los grupos: Las rutas pertenecientes a un grupo tienen el mismo color. Las rutas de los usuarios a los que no se les asignó un vehículo están de color rojo.

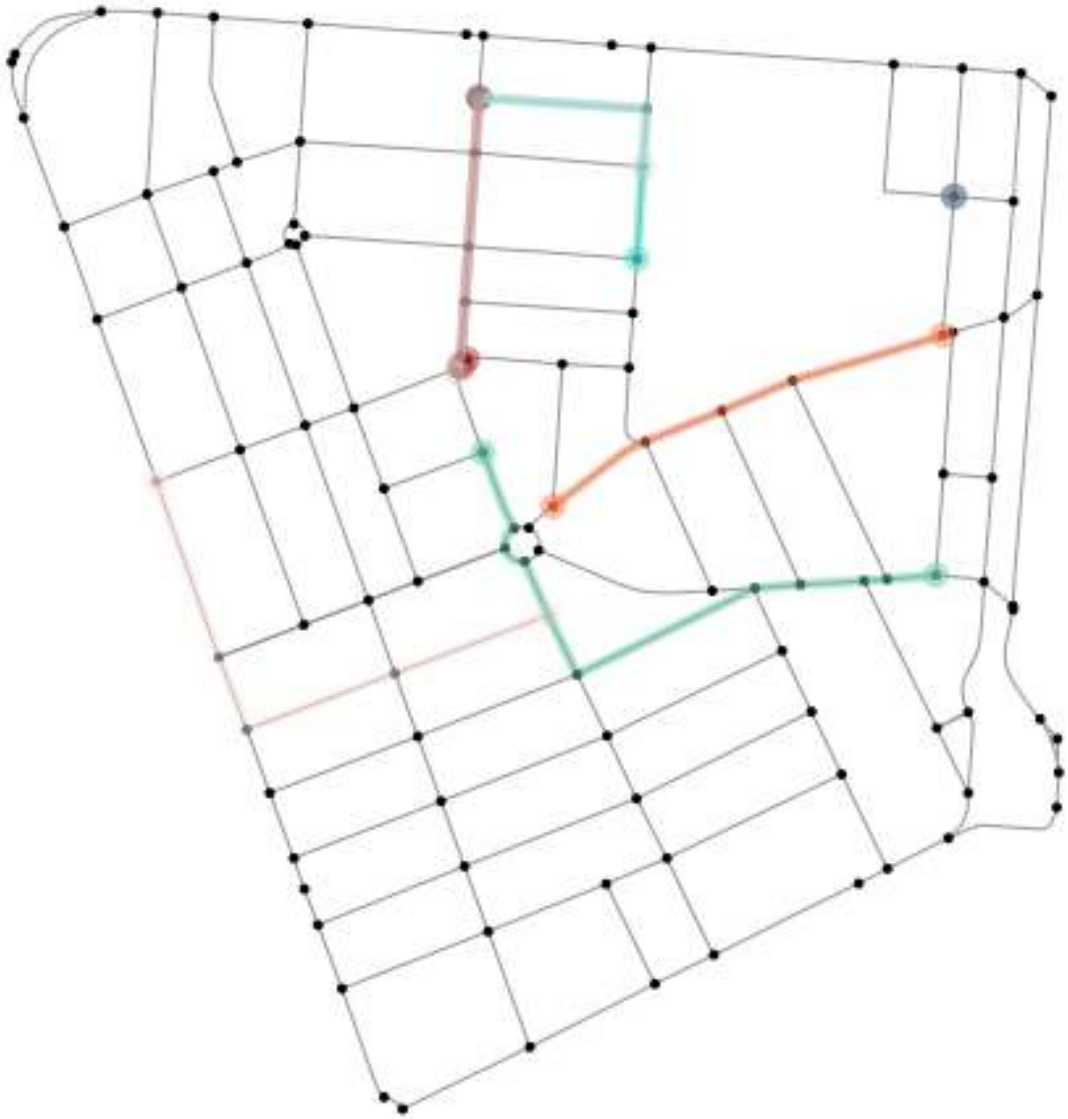
[109]:
```
plot_clustered_routes(clusters, get_colors(K_B), Graph)
```



Usuarios A que no tienen asignado un vehículo.

[110]:
```
plot_cluster(-1, clusters, Graph)
```

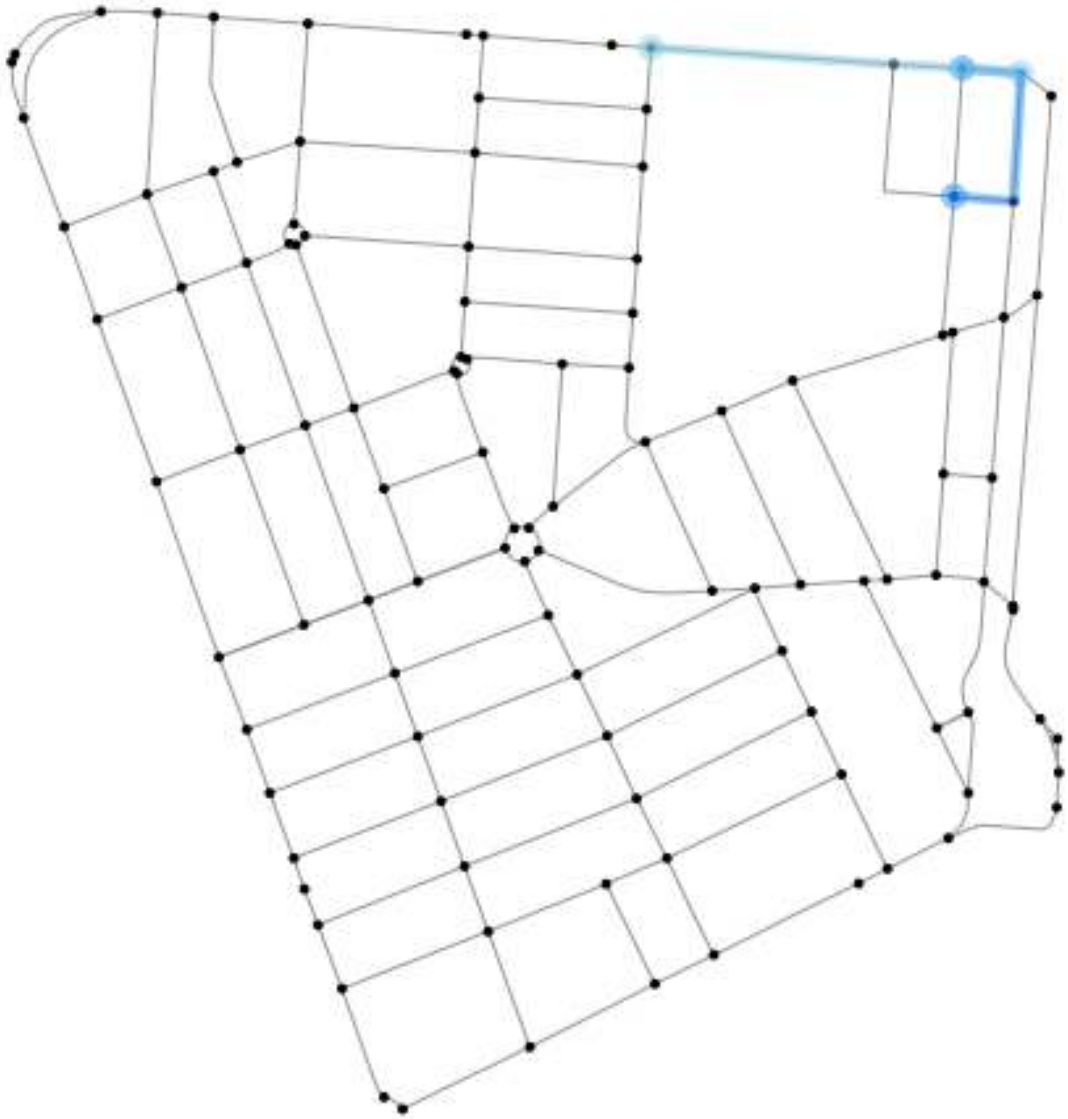
```
Número de usuarios en el grupo -1 :  7
```



Usuarios asignados al grupo 0.

[111]:
```
plot_cluster(0, clusters, Graph)
```

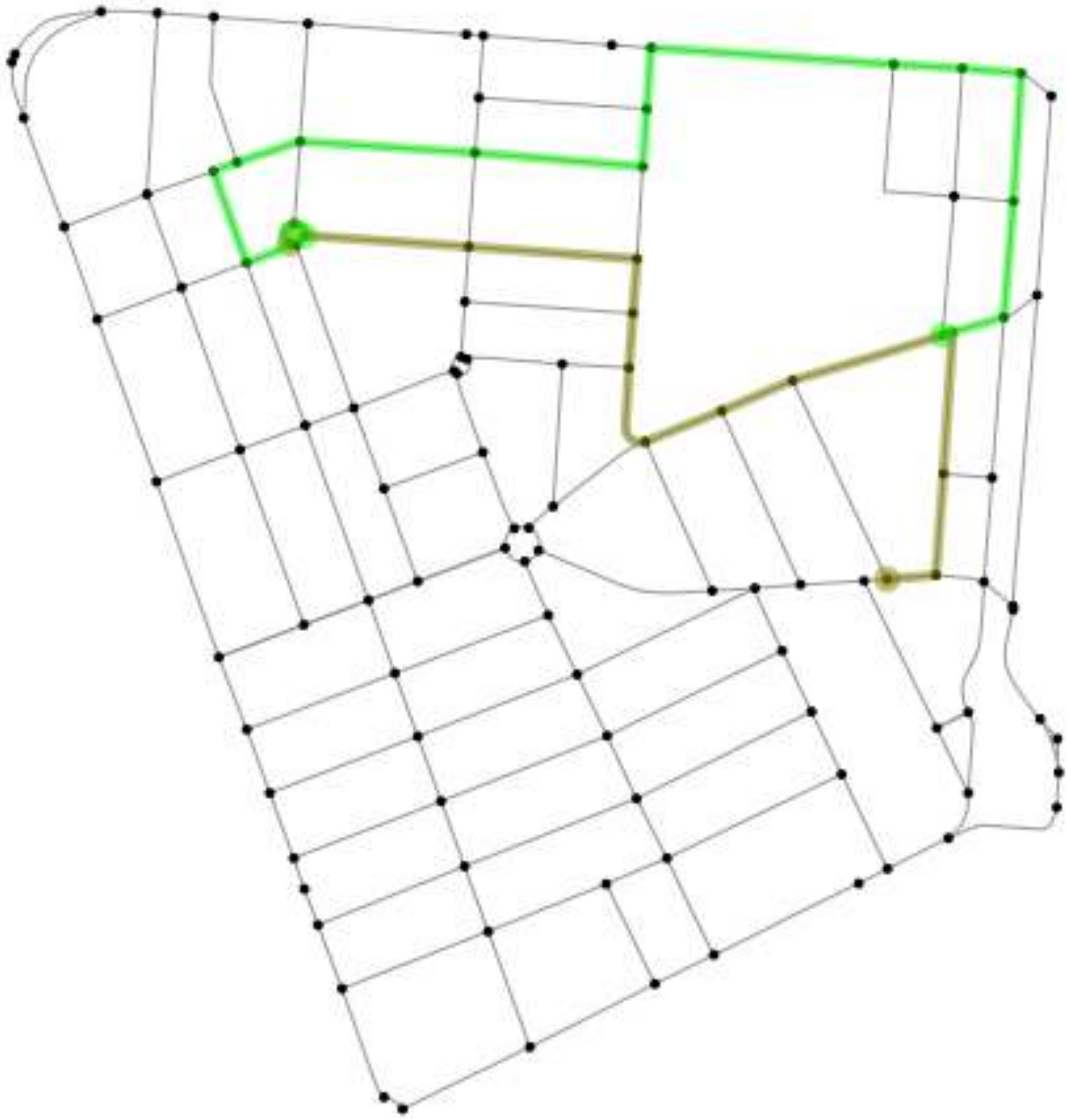
```
Número de usuarios en el grupo 0 :  2
```



Usuarios asignados al grupo 1**.**

[112]:
```
plot_cluster(1, clusters, Graph)
```

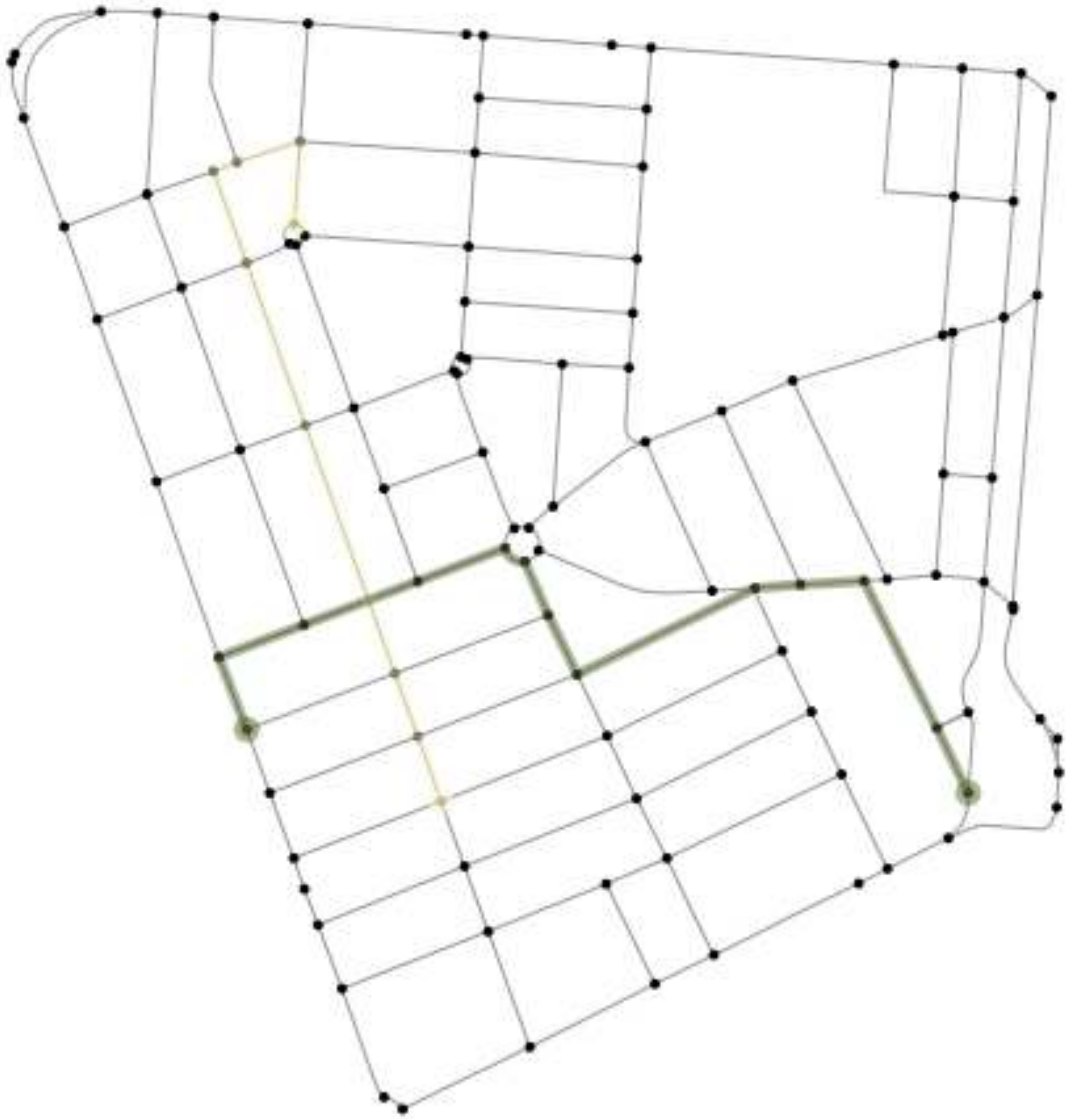
```
Número de usuarios en el grupo 1 :  2
```



Usuarios asignados al grupo 2.

[113]:
```
plot_cluster(2, clusters, Graph)
```

```
Número de usuarios en el grupo 2 :  2
```



Ejercicio 5: Resolver la instancia generada en el *ejercicio 1* con el algoritmo evolutivo multi-objetivo propuesto. Se debe hacer un estudio estadístico, graficar el frente de Pareto correspondiente al mejor valor de hipervolumen y las agrupaciones generadas por las soluciones que puedan ser interesantes para el tomador de decisiones.

Ejercicio 6: Combinar la técnica de jerarquización de Pareto con una técnica que mantenga diversidad en las soluciones a lo largo del frente de Pareto. Utilizar dicha combinación en el algoritmo evolutivo diseñado y usarlo para resolver la instancia generada en el *ejercicio 1*. Se debe graficar el frente de Pareto obtenido y las agrupaciones generadas por las soluciones que puedan ser interesantes para

el tomador de decisiones.

Ejercicio 7: Utilizar una técnica de cruza diferente y resolver la instancia del *ejercicio 1*. Se debe graficar el frente de Pareto obtenido y las agrupaciones generadas por las soluciones que puedan ser interesantes para el tomador de decisiones.