ChatGPT

Implementación de un Modelo de Ruteo Multiobjetivo con ALNS en Python

1. Contexto y Objetivos del Problema

La planificación de rutas de vehículos (VRP, *Vehicle Routing Problem*) consiste en asignar un conjunto de demandas de entrega a una flota de vehículos y determinar el orden en que cada vehículo atenderá sus entregas, buscando típicamente minimizar el costo total (por ejemplo, la distancia recorrida o el costo operativo) [1] [2]. En este caso, además del costo, se desean **rutas con "atractivo visual"**, es decir, rutas fáciles de entender y que luzcan lógicas sobre el mapa. En particular, la empresa quiere rutas que:

- **No tengan tramos que se crucen** dentro de la misma ruta.
- **No se solapen zonas entre rutas diferentes**, evitando que dos rutas recorran los mismos sectores de la ciudad.
- **No atraviesen demasiadas comunas distintas** en una sola ruta, fomentando rutas más compactas geográficamente.
- **Estén balanceadas en carga y duración** (todas las rutas de longitud o número de entregas similares).

Estas preferencias de "atractivo visual" buscan que las nuevas rutas no sean demasiado distintas a las que los planificadores están acostumbrados, facilitando la adopción de la solución automática. Sin embargo, agregar estas consideraciones estéticas puede aumentar el costo operativo (por ejemplo, usar más kilómetros o más vehículos), creando un **problema de optimización multiobjetivo** con dos objetivos en conflicto: **minimizar el costo** y **maximizar el atractivo visual** de las rutas. No existe una única solución óptima, sino un conjunto de soluciones de compromiso (trade-off) conocidos como soluciones **Pareto-óptimas**. Una solución se considera Pareto-óptima si **no es posible mejorar uno de los objetivos sin empeorar al menos otro** [3]. El conjunto de todas las soluciones no-dominadas forma la llamada **frontera (o frente) de Pareto**, que ilustra explícitamente el trade-off entre costo y estética [4].

Objetivo: Implementaremos paso a paso un modelo heurístico basado en **Adaptive Large Neighborhood Search (ALNS)** para generar rutas que minimicen el costo operativo y, simultáneamente, maximicen el atractivo visual. Al final, obtendremos múltiples soluciones representativas a lo largo de la frontera de Pareto (al menos 5 alternativas), desde la solución más económica hasta la más "estética", incluyendo sus rutas detalladas y visualizaciones en mapa y gráficas comparativas.

2. Herramientas y Librerías Recomendadas

Para resolver este problema en Python de manera eficiente, se recomienda apoyarse en las siguientes herramientas y librerías:

- **Librerías de optimización/heurísticas**: Una opción es utilizar la librería `ALNS` (Adaptive Large Neighborhood Search) de Python [5], que proporciona una implementación general de ALNS ya probada y flexible. Esta librería permite definir operadores de destrucción/reparación y criterios

de aceptación de forma modular, facilitando la experimentación. Se puede instalar fácilmente con `pip install alns` [6]. Alternativamente, puede implementarse ALNS manualmente con Python básico si se desea más control o aprendizaje. Para validaciones en instancias pequeñas, se puede complementar con un solver como **OR-Tools** de Google o **PuLP** para optimización entera mixta, aunque estos solvers manejan un solo objetivo (generalmente costo).

- **Manejo de datos**: Usaremos **pandas** para cargar y manipular los datos de entradas (demandas, vehículos, matrices de distancia/tiempo), y **numpy** para estructuras de datos numéricas (por ejemplo, matrices de distancia) que se necesiten para cálculos rápidos.

- **Cálculo geométrico (opcional)**: Para evaluar criterios geométricos como cruces de rutas, puede ser útil la librería **Shapely** o simplemente implementar funciones geométricas ad hoc (por ejemplo, verificar intersección de segmentos de línea). Sin embargo, en una primera instancia se puede evitar complejidad geométrica explícita mediante métricas aproximadas de "cruces" (discutido más adelante).

- **Visualización**: Utilizaremos **Matplotlib** para generar gráficos, incluyendo la visualización de las rutas sobre un plano cartesiano (latitud/longitud) y la gráfica de la frontera de Pareto. Si se desea una visualización más realista sobre mapas, se puede emplear **Folium** (para mapas interactivos con folletos) o la API de Google Maps, pero para nuestro instructivo nos enfocaremos en representaciones estáticas con Matplotlib por simplicidad.

- **Otras utilidades**: Python estándar ofrece todo lo necesario para manejar listas de rutas, cálculos de costos, etc. Se aprovechará la experiencia en Python del lector para manipular estructuras de datos (listas, diccionarios, tuplas) y escribir funciones de evaluación. No se requiere experiencia previa en metaheurísticas; iremos explicando el ALNS paso a paso.

3. Representación de los Datos del Problema

Antes de desarrollar el algoritmo, definiremos cómo representaremos la información relevante en Python de forma clara:

- **Conjuntos de nodos (clientes y depot)**: Tenemos un único **centro de distribución** (depot) y múltiples **demandas** de clientes. Podemos asignar a cada punto un índice o identificador. Por ejemplo, índice 0 para el depot, y 1…N para los N clientes. También es práctico manejar los nombres/IDs originales (por ejemplo "D_1", "D_2", etc.) en algún diccionario para referencias legibles, pero internamente usaremos índices numéricos para eficiencia (por ejemplo, una lista de demandas donde el índice en la lista representa al cliente).

- **Demandas (clientes)**: Cada demanda se caracteriza por:

- Coordenadas de destino (`latitude`, `longitude`).
- Cantidad a entregar o tamaño (`size`) que consume capacidad del vehículo.
- Tiempo de servicio (`stop_time`) en ese destino (ej: minutos que toma la entrega).
- Ventana de tiempo de atención (`tw_start`, `tw_end`) si la hay, que indica el intervalo del día en que se puede realizar la entrega (por ejemplo, 08:00–11:00). Convertiremos estos tiempos a una escala numérica común (por ejemplo minutos desde inicio de la jornada) para facilitar cálculos.

Una forma conveniente es guardar las demandas en un **DataFrame de pandas** o en una lista de diccionarios/tuplas. Por ejemplo, podríamos definir una clase simple `Demand` con atributos, o usar un DataFrame `df_demands` donde cada fila tiene las columnas mencionadas. Para acceder por índice, podríamos crear un dict `{id: index}` para mapear IDs a índices de array.

- **Vehículos**: Sabemos que el número de vehículos es fijo (por ejemplo 28 vehículos disponibles, según los datos proporcionados). Cada vehículo puede tener atributos:
- Capacidad de volumen (`capacity`).
- Tipo o ID (si hay diferentes tipos con distintas capacidades).
- Costo de uso, si aplica (en este problema, parece haber un **costo fijo por usar un vehículo** y posiblemente un costo variable según su capacidad; por simplicidad podríamos asumir que todos los vehículos tienen igual costo fijo de salida, o incorporar las diferencias si se proveen).

Podemos almacenar vehículos en una lista `vehicles` donde cada elemento es un dict o objeto con `capacity` y otros atributos. También tendremos el **número total de vehículos disponibles**. En este problema, la solución no puede usar más vehículos que los disponibles, pero podría usar menos si no son necesarios (lo que ahorra costo fijo).

- **Matrices de distancia y tiempo**: Disponemos de matrices pre-calculadas de distancias y tiempos de viaje entre cada par de puntos (depot y demandas) en los archivos provistos. Estas matrices pueden cargarse en estructuras como:
- Un **diccionario anidado**: `distance[i][j]` = distancia desde el punto i al punto j.
- O bien como **arrays NumPy** de dimensión NxN (donde N = número de clientes + 1 depot). Por ejemplo, `dist_matrix[i, j]` y `time_matrix[i, j]`. Dado que tenemos 364 puntos (1 depot + 363 demandas), las matrices serían de 364x364. Manejar esto en NumPy es eficiente y permite indexación directa por índices numéricos.

Al cargar de CSV, es posible que tengamos que mapear coordenadas a índices. Una estrategia es primero leer las demandas y el depot para crear la lista ordenada de nodos, luego construir las matrices en ese mismo orden. (En los datos, la matriz viene dada como tabla larga con columnas de coordenadas origen/destino; habría que hacer el `merge` o correspondencia de coordenadas a IDs. Para no complicar el instructivo, supondremos que generamos correctamente `dist_matrix` y `time_matrix` alineadas con el orden de nuestros nodos).

- **Horizonte de tiempo (jornada)**: Tenemos horas de inicio y fin de jornada (por ejemplo 07:20–16:00 según `overview.csv`). Esto determina que ninguna ruta puede salir antes de 7:20 ni regresar después de las 16:00. Conviene convertir estas horas a minutos desde un origen (7:20 podría ser minuto 0 de la jornada, 16:00 el minuto 520 de la jornada, por ejemplo) para manejar tiempos fácilmente. Cada ruta deberá respetar que el tiempo total de recorrido + servicio no exceda la jornada.

- **Representación de una ruta (solución parcial)**: Una ruta puede representarse simplemente como una **lista ordenada de índices de demanda** que el vehículo atiende en secuencia, comenzando y terminando en el depot implícitamente. Por ejemplo, una ruta `[0, 5, 2, 8, 0]` (usando `0` para denotar el depot al inicio y fin) indica que el vehículo sale del depot, va al cliente 5, luego 2, luego 8, y retorna al depot. Internamente, durante la búsqueda, podemos omitir listar el depot en ambos extremos si siempre asumimos que calcular el costo incluye salida y retorno al depot. Otra posibilidad es almacenar la ruta sin el depot (p.ej. `[5, 2, 8]`) pero tener claro en funciones de costo que se agrega el tramo desde y hacia el depot.

- **Solución completa (conjunto de rutas)**: Podemos representarla como una lista de rutas. Por ejemplo `solution = [[ruta1], [ruta2], ..., [rutaK]]` para K rutas usadas. Es importante que esta estructura sea fácil de manipular (añadir/quitar clientes de rutas, mover clientes entre rutas, etc.). Un enfoque común es también tener una estructura inversa para saber qué vehículo atiende cada demanda (un arreglo donde la posición i indica la ruta asignada de la demanda i), pero eso podemos derivarlo a partir de las listas de rutas.

Nota: Asegúrate de inicializar todas estas estructuras correctamente antes de ejecutar el algoritmo. Por ejemplo, leyendo los CSV con pandas:

```
import pandas as pd
df_demands = pd.read_csv('demands.csv')
df_vehicles = pd.read_csv('vehicles.csv')
dist_matrix = ... # construir a partir de distances.csv
time_matrix = ... # construir a partir de times.csv
```

y luego creando listas/arrays a partir de estos DataFrames. Mantener consistencia en los índices es clave (que la fila i de `df_demands` corresponda a la fila i+1 de la matriz de distancia si consideramos fila 0 de la matriz como el depot, etc.).

4. Funciones Objetivo: Costo Operativo vs. Atractivo Visual

Definiremos a continuación las métricas para los dos objetivos que queremos optimizar:

a) Costo operativo (a minimizar): Este puede incluir varios componentes según el caso: - *Distancia total recorrida* por todos los vehículos, convertida a costo monetario (por ejemplo, kilómetros * costo por km). Según los datos, podría usarse la matriz de distancias y un factor de costo por metro recorrida (en `costs.csv` se indica `cost_per_meter = 1.5`, lo que sugiere 1.5 unidades monetarias por metro). - *Costo fijo por uso de vehículo*: si se usa un vehículo, se incurre en un costo fijo (por ejemplo 7500 monetarios por vehículo en `costs.csv`). Esto incentiva utilizar menos vehículos si es posible. - *Otros costos variables*: en algunos problemas podría haber costo por tiempo (ej: salarios) o penalizaciones por entregas retrasadas. En nuestro caso nos centraremos en distancia y vehículos usados.

Entonces, la función de costo podría ser: $$ \text{Costo}(\text{solución}) = \sum_{\text{rutas}} \Big(C_f + \sum_{\text{tramos }(i\to j)\text{ en ruta}} \text{dist}(i,j) \cdot C_d \Big) $$ donde C_f es el costo fijo por ruta (por ejemplo 7500) y C_d es el costo por metro (1.5). La sumatoria recorre cada tramo de cada ruta, incluyendo los tramos de salir del depot al primer cliente y del último cliente regresar al depot.

Además, **restricciones** como no exceder la capacidad del vehículo o las ventanas de tiempo deben ser respetadas al evaluar una solución (incumplirlas haría la solución inviable). En ALNS típicamente trabajamos **solo con soluciones factibles**, así que la función de costo asume que ya se cumplen las restricciones duras (capacidad, tiempo máximo, ventanas). Si por algún motivo consideramos soluciones temporales infactibles, habría que incorporar penalizaciones muy grandes al costo para desincentivarlas.

b) Atractivo visual (a maximizar): Este objetivo es menos trivial de cuantificar, pues abarca **varios criterios cualitativos**. Propondremos crear una **función de "penalización estética"** que sea más baja

para rutas visualmente atractivas y más alta para rutas desordenadas. Como nuestro algoritmo ALNS por defecto minimiza una función (busca menor "costo"), conviene transformar *maximizar atractivo* en *minimizar falta de atractivo* (penalizaciones). Algunos componentes que podemos incluir:

- **Cruces en la misma ruta:** Si una ruta se cruza a sí misma geométricamente, se añade una penalización. Detectar cruces requiere revisar la secuencia de puntos: para cada par de tramos no adyacentes, comprobar si las líneas (A->B) y (C->D) se intersectan. Esto se puede hacer con fórmulas de intersección de segmentos (por ejemplo, usando la orientación de puntos) o con ayuda de la librería `shapely`. Cada cruce detectado podría sumar, por ejemplo, X puntos de penalización (X grande, ya que se desea evitar totalmente). Con ALNS podemos incluso prohibir esas soluciones rechazándolas, pero es preferible penalizar fuertemente para que el algoritmo tienda a evitarlas.

- **Solapamiento de rutas en la misma zona:** Si dos rutas distintas cubren la misma zona o sector de la ciudad, considerarlo anti-estético. Una forma de medir esto: si disponemos de la **comuna** o región de cada cliente, podemos contar cuántas rutas distintas entran en la misma comuna. Por ejemplo, si la comuna X es atendida por 2 rutas, eso implica un solapamiento. Se podría penalizar por cada comuna atendida por más de una ruta. Otra aproximación más continua es calcular un indicador de superposición geográfica, como el área común cubierta por las rutas (esto es complejo). Para simplificar: **penalicemos cada comuna duplicada**. Por cada comuna que aparece en más de una ruta, sumamos penalización P_{solap}. Si todas las rutas están bien separadas, cada comuna aparece en solo una ruta y esta penalización es cero.

- **Cantidad de comunas por ruta:** Rutas más concentradas geográficamente son más fáciles de seguir. Entonces, por cada ruta podemos contar el número de comunas distintas que atraviesa. Cuanto mayor ese número, mayor penalización. Por ejemplo, una ruta que pasa por 5 comunas es menos deseable que una que pasa por solo 2 comunas. Podemos sumar una penalización $P_{\text{comuna}} \times (\text{comunas en ruta} - 1)$ (suponiendo 1 comuna es ideal, no penaliza, cada comuna adicional penaliza). O alternativamente, fijar un umbral (ej: más de 3 comunas ya es mucho).

- **Desbalance entre rutas:** Queremos que las rutas tengan longitudes similares (en distancia o en número de entregas), para evitar que una sea muy larga y otra muy corta. Una métrica típica es el **desvío estándar** del número de entregas por ruta, o de las distancias por ruta. Podemos calcular la desviación estándar σ de las distancias recorridas por cada vehículo y penalizarla (o penalizar la diferencia máxima entre rutas). Otra opción es penalizar directamente la **diferencia** entre la ruta más larga y la más corta. Por ejemplo, si la ruta más larga tiene 50 km y la más corta 10 km, esa dispersión de 40 km es mala; podríamos penalizar proporcionalmente a esa brecha. Para número de entregas, lo mismo: si una ruta lleva 20 paquetes y otra solo 2, la distribución es muy desigual, penalizar. En resumen, introduciremos un término de penalización $P_{\text{bal}} \times \sigma_{\text{rutas}}$ (tras normalizar unidades, quizá).

En conjunto, la **función de penalidad estética** podría ser: $$ \text{Penalidad}_\text{est} = w_1 (\text{#Cruces}) + w_2 (\text{#Solapamientos}) + w_3 (\text{#Comunas\ totales en todas las rutas}) + w_4 (\text{Desbalance}) $$ donde w_1, w_2, w_3, w_4 son pesos o factores de importancia para cada criterio. Ajustar estos pesos es delicado y puede requerir prueba y error. Por ejemplo, w_1 (cruces) podría ser muy alto para prácticamente prohibir cruces. w_2 (solapamiento) también alto. w_3 más moderado. w_4 podría ser más bajo, porque cierto desbalance podría tolerarse más que ver dos rutas mezcladas en la misma zona.

Un enfoque alternativo: Definir directamente un **puntaje de atractivo visual** en escala 0 a 100, donde 100 es lo más atractivo. Se podría empezar en 100 y restar puntos por cada "falta": -50 por cualquier cruce (muy grave), -20 por cada comuna extra por ruta, -30 por cada comuna servida por rutas múltiples, etc., con mínimos en 0. Sin embargo, para la metaheurística es más fácil manejar **minimización**, por lo que es común trabajar con penalizaciones (lo que equivale a minimizar la *falta de atractivo*).

Nota: Dado que finalmente debemos optimizar **ambos** objetivos, no es estrictamente necesario colapsarlos en una sola función. Podemos manejarlo con técnicas multiobjetivo (ver sección 7). Pero a efectos de implementación, muchas veces se combina linealmente costo y penalidades en una **función de fitness** única si se usa el método de suma ponderada. Por ejemplo: $$ \text{Fitness} = \text{Costo} + \lambda \times \text{Penalidad}_\text{est} $$ donde λ es un peso que equilibra unidades de costo y penalidad. Exploraremos más esta idea en la sección de multiobjetivo. Por ahora, tenemos definidas las métricas necesarias para evaluar cualquier solución en términos de costo y atractivo.

5. Generación de una Solución Inicial Feasible

El algoritmo ALNS necesita arrancar con **una solución inicial** (factible) sobre la cual empezar a iterar [5]. Esta solución no tiene que ser óptima; puede ser obtenida mediante una heurística constructiva sencilla. Presentamos algunas estrategias:

- **Heurística del vecino más cercano (Nearest Neighbor):** Históricamente la empresa usaba un método así, que consiste en formar rutas tomando siempre la siguiente demanda más cercana al último punto añadido [7]. Para adaptarla aquí, podríamos: empezar en el depot, luego ir al cliente más cercano, luego al siguiente más cercano desde allí, hasta que agregar otro cliente violaría la capacidad del vehículo o alguna ventana de tiempo; en ese punto, cerrar la ruta y comenzar una nueva ruta con algún cliente no atendido (por ejemplo, el más cercano al depot entre los pendientes) y repetir. Esta heurística es **greedy** por distancia, tiende a producir rutas cortas, aunque no considera balance ni otras métricas.

- **Algoritmo de barrido (Sweep Algorithm):** Ordena los clientes angularmente alrededor del depot (por ejemplo, calculando el ángulo polar de cada punto respecto al depot) y luego parte esa lista ordenada en segmentos que formen cada ruta (respetando capacidades). La idea es que puntos cercanos en ángulo estarán geográficamente cercanos, formando rutas naturalmente compactas. Esta técnica puede generar rutas más atractivas visualmente (menos zigzag), aunque no garantiza mínimos costos globales.

- **Clarke & Wright (Savings) Heuristic:** Es un método clásico donde se comienza con cada cliente atendido por una ruta individual, luego iterativamente se combina rutas si hacerlo ahorra distancia (usando los "ahorros" de combinar dos rutas en una, siempre y cuando no se violen capacidades ni ventanas). Este método tiende a reducir número de vehículos y costo total de manera eficaz, aunque podría crear algunas rutas menos compactas.

- **Uso de solvers exactos en subproblemas pequeños:** Si el número de demandas fuera pequeño, uno podría resolver un VRP simplificado con un ILP (ej. usando OR-Tools o Gurobi) [8] [9]. Pero con cientos de demandas esto no es viable directamente. Sin embargo, se podría dividir el problema en clústeres (por zona) y resolver cada clúster de forma exacta para arrancar con rutas "óptimas locales" por zona. En nuestro caso, tal subdivisión podría alinearse con el atractivo visual (rutas por comuna).

Para nuestro instructivo, implementemos un esquema simple combinando ideas: **Greedy por distancia respetando capacidad**: 1. Comenzar con todas las demandas sin asignar y una lista vacía de rutas. 2. Seleccionar una demanda no asignada como inicio de una nueva ruta (por ejemplo la más cercana al depot, o simplemente la siguiente en la lista). 3. Ir agregando a la ruta actual la demanda no asignada *más cercana al último punto de la ruta*, siempre que quepa en capacidad y no viole la ventana de tiempo (si la siguiente demanda más cercana no cabe, intentar la segunda más cercana, etc.). Continuar hasta que no se pueda agregar más sin violaciones (o hasta un máximo de paradas si quisiéramos limitar). 4. Cerrar la ruta (regresar al depot) y marcar las demandas de esa ruta como asignadas. 5. Repetir hasta asignar todas las demandas o quedarse sin vehículos disponibles.

Esta estrategia produce rápidamente una solución factible. Podría dejar algunas rutas con muy pocos puntos si esos puntos eran muy "lejanos" (lo cual suele suceder con NN puro). Por ejemplo, supongamos 20 pedidos y 4 vehículos de capacidad suficiente: la heurística greedy podría asignar 18 pedidos en 3 rutas y dejar 2 en la última ruta. Eso produce desbalance (dos rutas de 9 entregas y una ruta con 2 entregas, por ejemplo).

Para mejorar un poco la estética inicial, otra variante: **Agrupar por proximidad geográfica primero**. Por ejemplo, se puede aplicar *k-means* clustering sobre las coordenadas de las demandas, con k = número de vehículos que planeamos usar. Así obtenemos grupos de clientes cercanos. Luego, para cada grupo, construir la ruta (ordenando internamente por nearest neighbor o resolviendo TSP dentro del clúster). Esto garantiza rutas bien separadas por zona, aunque quizás use más vehículos que el mínimo.

Ejemplo (ilustrativo): Supongamos un subconjunto de 20 demandas y 4 vehículos disponibles. Una solución inicial puramente orientada a costo (distancia) podría usar solo 3 vehículos llenándolos casi al máximo y dejando un vehículo sin usar, resultando en rutas más largas y mezcladas. En cambio, una solución inicial orientada a atractivo podría usar los 4 vehículos, cada uno atendiendo un sector distinto. Las figuras a continuación muestran esquemáticamente la diferencia:

Ejemplo de solución inicial orientada a minimizar el costo. En esta solución, se usaron 3 vehículos en lugar de 4, combinando más entregas por ruta. Cada color representa la ruta de un vehículo distinto, partiendo y retornando al depósito (marcado con ●).* La solución minimiza la distancia total y el número de vehículos, pero puede notarse que las rutas (por ejemplo, la roja y la verde) cubren zonas que se superponen parcialmente y los recorridos son más largos (una ruta cubre desde la parte superior derecha hasta el sur, atravesando varias comunas).

Ejemplo de solución orientada a maximizar el atractivo visual. Aquí se utilizaron 4 vehículos para mantener las rutas más compactas geográficamente. Cada ruta cubre una zona diferente sin traslaparse: por ejemplo, la ruta roja cubre la parte derecha, la verde la izquierda, la azul el extremo superior, etc. Las rutas están más balanceadas en número de entregas (cada vehículo lleva un número similar de puntos) y evitan cruzar sus recorridos con otros vehículos.* (Obsérvese que las rutas son más cortas individualmente, aunque el costo total podría ser algo mayor por usar un vehículo extra).

Estas soluciones iniciales sirven como puntos de partida. En la práctica, elegir una buena solución inicial puede acelerar la convergencia del ALNS. Es válido comenzar con una solución "feasible cualquiera" y dejar que ALNS mejore, pero si podemos incorporar cierta lógica inicial (por ejemplo, usando el método de clúster para reducir solapamientos de entrada), estaremos más cerca de la frontera de Pareto buscada desde el inicio.

6. Algoritmo ALNS: Búsqueda de Vecindarios a Gran Escala Adaptativa

Ahora entramos en el núcleo: la implementación del **Adaptive Large Neighborhood Search (ALNS)**. Este es un algoritmo iterativo que alterna entre **destruir** parcialmente la solución actual (removiendo algunas entregas de sus rutas) y luego **repararla** (reinsertando esas entregas en las rutas, potencialmente en diferentes posiciones o vehículos), con el fin de explorar distintas configuraciones. Lo de "Adaptativa" se refiere a que el algoritmo ajusta dinámicamente la frecuencia con que usa cada tipo de movimiento según su desempeño durante la búsqueda [10] .

Estructura general del ALNS:

1. **Inicialización:** Partir de la solución inicial construida en el paso 5. Esta es nuestra solución *actual* y también la *mejor conocida* hasta ahora. Definir parámetros iniciales: por ejemplo, una temperatura inicial T si usamos criterio de aceptación simulando recocido simulado (ver más abajo), un límite de iteraciones sin mejora, etc. También inicializar **pesos de operadores** (cada operador de destrucción y reparación tiene un puntaje o probabilidad de ser elegido, que iremos ajustando). Todos los operadores pueden empezar con pesos iguales.

2. **Iteración principal:** Repetir hasta cumplir criterio de parada (por ejemplo, 10000 iteraciones o X minutos de cómputo):

3. **Selección de operadores:** Elegir aleatoriamente **un operador de destrucción** y **uno de reparación** de acuerdo a sus pesos actuales (por ejemplo, tenerlos en una ruleta de probabilidades proporcional al peso). Estos determinarán cómo modificaremos la solución.
4. **Aplicar destrucción:** Usando el operador de destrucción elegido, remover un subconjunto de entregas de la solución actual. Por ejemplo, quitar 5 entregas de sus rutas (las seleccionadas según cierta lógica del operador).
5. **Aplicar reparación:** Con el conjunto de entregas removidas, usar el operador de reparación seleccionado para reinsertarlas (asignarlas a alguna ruta y posición). Esto produce una **solución candidata** (hopefully factible; los operadores de reparación deben intentar respetar todas las restricciones al reinsertar).
6. **Evaluar la solución candidata:** Calcular su costo total y penalidad estética (o el fitness combinado, según estemos optimizando uno o múltiples objetivos en este run; ver sección 7).
7. **Criterio de aceptación:** Decidir si la solución candidata reemplaza a la solución actual. Podemos usar un criterio **greedy** (aceptar solo si es mejor que la actual en el objetivo que estemos optimizando) o un criterio **probabilístico (Simulated Annealing)** que ocasionalmente acepte soluciones peores para escapar de óptimos locales. Un ejemplo típico: si $\Delta = \text{Objetivo}*{cand} - \text{Objetivo}$ (diferencia en valor de función a minimizar), entonces si $\Delta \le 0$ (candidato no peor) aceptarlo siempre; si $\Delta > 0$ (candidato peor), aceptarlo con probabilidad $p = \exp(-\Delta / T)$, donde T es una "temperatura" que va disminuyendo con las iteraciones (alto T al inicio permite más exploración, T baja al final endurece la aceptación). En caso de multiobjetivo, el criterio se complica – se podría aceptar si el candidato **no es dominado** por la actual, etc., pero en un primer approach, manejemos un solo objetivo compuesto para la decisión de aceptación.
8. **Actualizar la solución actual:** Si se aceptó el candidato, éste pasa a ser la *solución actual*. Si además es mejor que la *mejor solución conocida* (global) en términos de nuestro objetivo primario, actualizar la mejor conocida.

9. **Actualizar pesos de operadores:** Aquí entra la adaptatividad. Se lleva un registro de cómo resultó la aplicación de los operadores elegidos:

    - Si el operador produjo una nueva mejor solución global, otorgarle una **alta recompensa** (por ejemplo +5 puntos).
    - Si produjo una solución que fue aceptada pero no global óptima, darle una **recompensa menor** (+1 punto).
    - Si no se aceptó (o fue rechazada por peor), quizás 0 puntos. Luego actualizar los pesos de esos operadores (por ejemplo, $peso_{nuevo} = \rho \times peso_{actual} + (1-\rho) \times \text{recompensa}$, con ρ un factor de decaimiento para que el efecto de recompensas recientes sea prominente). De esta forma, operadores que frecuentemente llevan a buenas mejoras tendrán pesos más altos y serán elegidos más a menudo conforme avanza la búsqueda, adaptándose a la instancia [10].

10. **Retornar la mejor solución encontrada** una vez se cumple el criterio de parada. Esto podría ser tras X iteraciones sin mejora (estancamiento) o un tiempo límite.

En pseudocódigo simplificado con lista numerada:

- **Entrada:** Solución inicial S, lista de operadores de destrucción D_i y reparación R_j, parámetros.
- **Inicializar:** $S_{actual} \leftarrow S$; $S^* \leftarrow S$; pesos $w(D_i), w(R_j) \leftarrow 1$.
- **Para iteración = 1 hasta MaxIter:**
- Seleccionar D_i y R_j aleatoriamente con prob. $\propto w(D_i), w(R_j)$.
- $S' \leftarrow$ aplicar D_i a S_{actual} (remover algunas entregas).
- $S_{cand} \leftarrow$ aplicar R_j a S' (reinsertar entregas removidas).
- Si S_{cand} no es factible: (p.ej. violó ventana de tiempo) opcionalmente descartar o penalizar fuertemente y continuar (goto next iter).
- Calcular $f(S_{cand})$ y $f(S_{actual})$ (valor de la función objetivo en cada).
- **Si** $f(S_{cand}) < f(S_{actual})$ (mejor) **entonces** aceptar;
  sino aceptar con probabilidad $\exp(-(f(S_{cand})-f(S_{actual}))/T)$.
- Si aceptado: $S_{actual} \leftarrow S_{cand}$.
- Si además $f(S_{cand}) < f(S^)$*: actualizar mejor solución $S*^ \leftarrow S_{cand}$.
- Actualizar pesos $w(D_i), w(R_j)$ según éxito:
    - Si S_{cand} es nuevo S^* (global best): recompensa grande a D_i,R_j.
    - Si S_{cand} fue aceptado pero no global: recompensa pequeña.
    - Si no se aceptó: recompensa cero.
- Reducir gradualmente T (en enfoques con recocido simulado).
- **Retornar** S^* como mejor solución encontrada.

Operadores de Destrucción (Remoción): Es clave definir varios métodos diferentes de seleccionar qué entregas quitar en cada iteración. Algunos ejemplos útiles en VRP: - *Remoción aleatoria:* quita, por ejemplo, un % aleatorio de clientes (p.ej. 5%) de la solución, elegidos uniformemente al azar. Es simple y da diversidad. - *Remoción de peor costo:* quita aquellos clientes cuya remoción daría la mayor reducción de costo inmediato. Por ejemplo, podemos calcular para cada cliente cuánto ahorra (en distancia) sacarlo de su ruta (conectar sus vecinos directos entre sí). Quitar los "más costosos" puede abrir oportunidad a reinserciones en otra ruta más eficientes. - *Remoción por clúster:* selecciona un cliente aleatorio y luego también quita algunos de sus vecinos cercanos geográficamente. Esto elimina un grupo local para reordenar posiblemente de otra forma. Ayuda a reconfigurar subrutas en una zona. - *Remoción secuencial:* elige una ruta entera (o un gran tramo de una ruta) para quitar completamente. Por ejemplo, eliminar todos los clientes de la ruta más larga. Después en la reparación, estos clientes podrían reubicarse en otras rutas más cortas, mejorando balance. - *Remoción basada en tiempo:* (si

hubiera holguras de ventanas) quitar clientes que tienen ventanas de tiempo muy restrictivas o que van últimos en su ventana (candidatos conflictivos). - *Remoción por atractivo visual:* podríamos diseñar operadores específicos, p.ej., quitar clientes que causan que una ruta entre a una comuna lejana aislada de las demás entregas de esa ruta (outliers geográficos), o quitar clientes de rutas que se están solapando con otras (para permitir recolocarlos en otra ruta y así reducir solape).

La cantidad de clientes a remover puede ser fija (por ejemplo 5 clientes cada vez) o aleatoria dentro de un rango (e.g. entre 5% y 10% de los clientes). A veces se liga a la temperatura: mayor destrucción al inicio (exploración amplia) y menos al final.

Operadores de Reparación (Inserción): Una vez tenemos un conjunto de clientes removidos, debemos reinsertarlos uno por uno en alguna ruta (existente o una nueva si tenemos vehículos libres). Estrategias comunes: - *Inserción greedy por costo:* Para cada cliente removido, inserta donde incremente menos el costo (distancia) de ruta. Esto implica probar ubicarlo entre cada par de entregas existente en cada ruta y encontrar el lugar con menor costo extra (considerando también tiempo y capacidad). Es el más básico. - *Inserción por "regret" (remordimiento):* En lugar de insertar un cliente inmediatamente en su mejor posición, se evalúa su costo de inserción en las **m** mejores posiciones (por ejemplo 2 mejores), y se calcula el "remordimiento" = diferencia de costo entre la mejor y la segunda mejor posición. Luego se inserta primero aquel cliente cuyo remordimiento es mayor (es decir, aquel para el cual si no lo insertas ahora en su mejor sitio, luego podría salir mucho más caro ponerlo en su segunda opción). Esto mejora decisiones miopes. - *Inserción orientada a estética:* Podríamos modificar el criterio de "costo" para incluir penalizaciones estéticas. Por ejemplo, al calcular el costo de insertar un cliente X en una ruta Y, además del aumento en kilómetros, sumarle un término si X pertenece a una comuna nueva para esa ruta Y (penalizar) o si causaría que la ruta Y se traslape con otra (difícil de evaluar localmente). Una idea: favorecer insertar clientes en rutas que ya atienden esa comuna, para no aumentar comunas distintas por ruta. O insertar un cliente en la ruta cuyo "centroide" esté más cerca de ese cliente (mantener clusters). - *Inserción en nueva ruta:* Si hay vehículos disponibles sin usar, considerar si conviene iniciar una ruta nueva con alguno de los clientes removidos (por ejemplo, si estaban causando mucho desbalance en su ruta original). Esto requiere tener en cuenta el costo fijo de abrir una ruta.

Implementar los operadores requiere escribir funciones que manipulen la representación de la solución (listas de rutas) de forma cuidadosa pero es relativamente local: remover es quitar elementos de listas; insertar es agregar en una lista. Hay que recalcular costo incremental rápidamente – aquí la estructura de matriz de distancia es útil: se puede obtener el costo de insertar cliente k entre i y j como `dist[i,k] + dist[k,j] - dist[i,j]`. También hay que recalcular tiempos de llegada para chequear ventanas: tras inserciones, actualizar la hora de llegada a cada cliente en la ruta para verificar que no se violen las `tw_end`. Si una inserción rompe la factibilidad temporal, se descarta esa posición.

Criterio de aceptación y enfriamiento: Usar recocido simulado (Simulated Annealing) es común en ALNS [10]. Inicialmente, ponemos una temperatura T alta para permitir aceptar incluso soluciones algo peores (escapar de óptimos locales), y gradualmente la bajamos (por ejemplo, cada cierta cantidad de iteraciones do `T := 0.999 * T`). También podemos reiniciar la búsqueda si parece atascada (por ejemplo, si en 1000 iteraciones no mejora la mejor solución, podemos resetear la solución actual a la mejor global conocida, o aumentar temperatura de nuevo).

Ejemplo de iteración ALNS: Supongamos tenemos una solución actual con costo+penalidad = 520000. Seleccionamos un operador de destrucción "remoción aleatoria de 5 clientes" y uno de reparación "inserción greedy". Quitamos 5 clientes (se liberan huecos en sus rutas). Luego los reinsertamos uno por uno en la mejor posición posible. Obtenemos una solución candidata de valor 530000 (un poco peor). Si T es alta al inicio, podríamos aceptar esta solución a pesar de ser peor (hay un cierto chance). Si aceptamos, actualizamos la solución actual. Como no fue mejora global, los operadores reciben poca

recompensa (pero al menos fueron aceptados). En otra iteración, quizá removiendo un tramo entero de una ruta y reinsertando, logremos 510000, mejorando el global; entonces esos operadores obtienen alta recompensa (haciendo más probable que repitamos ese movimiento u otros similares). Así el algoritmo irá explorando diversas permutaciones de entregas entre rutas.

El ALNS ha demostrado ser muy efectivo para VRP y variantes, encontrando soluciones de alta calidad en menos tiempo que métodos exactos para instancias grandes [11]. Al combinarlo con criterios de aceptación tipo recocido, se evitan quedarnos atrapados en subóptimos rápidamente.

7. Extensión a la Optimización Multiobjetivo

Hasta ahora describimos ALNS como si optimizara una sola función objetivo (por ejemplo, el costo más una penalización combinada). Sin embargo, nuestro problema es intrínsecamente multiobjetivo: queremos minimizar costo **y** maximizar atractivo visual. En optimización multiobjetivo, generalmente no se reduce a una única solución óptima, sino a un **conjunto de soluciones óptimas de Pareto** (cada una equilibra los objetivos de manera distinta) [4]. ¿Cómo incorporamos esto en el algoritmo? Existen un par de enfoques:

A. Suma ponderada de objetivos (método escalar): Como mencionamos, podemos crear una sola función de *fitness* que combine costo y penalizaciones mediante un peso λ. Variando ese peso en diferentes ejecuciones, obtenemos diferentes énfasis: - Si $\lambda = 0$, la función es solo el costo (ignora atractivo visual). ALNS nos dará la solución mínima en costo (posiblemente con rutas locas visualmente). - Si λ es muy grande, la penalidad estética dominará la función, y ALNS buscará casi exclusivamente maximizar la estética a expensas del costo (resultado: rutas muy bonitas pero tal vez largas o usando muchos vehículos). - Valores intermedios de λ producen soluciones compromiso.

El método sería: fijar λ en un cierto valor y correr todo el ALNS de principio a fin optimizando $f = \text{Costo} + \lambda \times \text{Penalidad}_\text{est}$. Al final obtenemos una solución $S(\lambda)$. Repetimos con varios valores de λ (por ejemplo 5 valores distintos que cubran desde 0 hasta un valor alto) para obtener al menos 5 soluciones distintas. Luego, recopilamos todas y filtramos las dominadas para quedarnos con la frontera de Pareto aproximada.

Ventajas: simple de implementar usando esencialmente el mismo código ALNS, solo cambiando el valor de λ.
Desventajas: requiere múltiples corridas del metaheurístico, y la elección de λ adecuados puede ser delicada (la relación de intercambio no es lineal generalmente). Además, la suma ponderada no puede encontrar soluciones Pareto no-convexas en el espacio objetivo, pero en la práctica con objetivos de este tipo suele funcionar razonablemente.

B. Enfoque de restricciones ε: Otra técnica es fijar un objetivo como primario y convertir otro en restricción. Por ejemplo: minimizamos costo pero imponiendo que la penalidad estética $\leq \varepsilon$. Variando ε conseguimos distintas soluciones. Esto se podría integrar p. ej. en ALNS penalizando fuertemente cualquier solución que exceda cierta penalidad máxima (para mantenerlo factible en sentido de la restricción). Es similar a la ponderación pero a veces más intuitivo: "encuentra la mejor ruta con penalidad estética no peor que X".

C. ALNS multiobjetivo (Pareto local search): En teoría podríamos modificar ALNS para **mantener un conjunto de soluciones** en paralelo, e intentar progresar ese conjunto según dominancia de Pareto. Por ejemplo, en lugar de una solución actual, manejar un *conjunto actual* de varias soluciones y aplicar

destrucción/reparación a alguna de ellas (o a combinaciones) generando nuevos candidatos, incorporándolos si amplían la frontera de Pareto. El criterio de aceptación aquí es: aceptar cualquier nueva solución que **no sea dominada** por las ya existentes (y desechar las que resulte dominadas por la nueva). Con el tiempo se va llenando un conjunto Pareto. Este enfoque es más complejo de implementar, pues requiere almacenar y comparar muchas soluciones constantemente y diversificar bien las búsquedas para distintas zonas del espacio objetvo. A veces se usan híbridos con algoritmos evolutivos (por ejemplo, usar ALNS como operador de mutación en un esquema tipo NSGA-II que mantiene población). Dado que esto rebasa la complejidad esperada para alguien nuevo en optimización avanzada, lo mencionamos solo conceptualmente.

En este instructivo adoptaremos el enfoque (A) de **múltiples corridas con distintos pesos** para obtener múltiples soluciones. Es más sencillo y explícito para generar la frontera pedida. Así, definiremos $\text{Fitness}\lambda(S) = Costo(S) + \lambda \cdot \text{Penalidad}\text{est}(S)$ y en cada corrida ALNS usaremos esa como función objetivo a minimizar.

Durante cada corrida, el ALNS considerará una solución mejor que otra si su Fitness_λ es menor. Fíjese que eso implica hacer compromisos internamente también: por ejemplo, con λ grande, el algoritmo preferirá sacrificar bastante costo si logra mejoras notables en estética (porque la función ponderada así lo valora).

Elección de los pesos λ: Una forma de elegirlos es: - $\lambda = 0$ (ignorar estética) → solución puramente costo mínimo. - λ ligeramente mayor para pequeñas consideraciones estéticas. - Un valor intermedio que balancee ambos objetivos aproximadamente igual. - Un valor alto donde la estética importe mucho más que el costo. - Un valor extremadamente alto que fuerce la estética casi a ultranza (sirve para encontrar el extremo Pareto enfocado en estética).

No hay valores absolutos claros; podríamos normalizar las escalas (por ejemplo, dividir el costo por un valor típico y penalidades por otro) para que ambos estén del mismo orden antes de ponderar. Suponiendo penalidades están en una escala de 0 a 1000 y costos en cientos de miles, λ necesitaría ser bastante grande para equiparar la influencia. Alternativamente, se pueden calibrar realizando corridas de prueba.

Criterio de parada en multi-corridas: Cada corrida ALNS con peso distinto se puede ejecutar con los mismos parámetros de iteraciones. También es posible **inicializar cada corrida con la solución óptima de la corrida anterior** para acelerar convergencia. Por ejemplo, podríamos primero obtener la solución óptima en costo (λ=0). Luego, para λ=0.25, iniciar ALNS desde la solución anterior (aunque esa solución no es buena en estética, el ALNS la irá modificando). O al revés: empezar de una solución muy estética para λ altos y luego bajar λ. Sin embargo, para evitar sesgo, puede ser mejor iniciar cada corrida desde cero (o desde la misma solución inicial básica) y dejar que cada una explore libremente.

Salida de cada corrida: Guardaremos la mejor solución obtenida S^*_λ con sus métricas: - Costo total (distancia recorrida + costos fijos). - Penalidad estética total (o, podemos derivar a partir de ella valores como número de cruces, etc., para interpretación). - Estos dos forman un par $(Costo, PenEst)$ que podremos plotear.

Una vez tenemos las soluciones de las 5 (o más) corridas con distintos λ, construiremos la **frontera de Pareto aproximada** tomando aquellas soluciones que no sean dominadas por otra. Dado que elegimos λ de forma creciente, es probable que $S^_{\lambda_0}$ tenga menor costo y peor estética que $S^_{\lambda_4}$, etc., formando ya un frente ordenado. Pero no se garantiza – podríamos encontrar que un λ intermedio produjo una solución dominada por otra. En tal caso, la descartamos. Lo

importante es terminar con un conjunto de soluciones donde **al movernos de una a otra, mejorar un objetivo implica empeorar el otro**, reflejando el compromiso.

8. Generación de la Frontera de Pareto y Selección de Soluciones

Procedamos entonces a generar las soluciones con distintos pesos y ensamblar la frontera de Pareto:

1. **Definir escenarios/pesos:** Por ejemplo, tomemos 5 escenarios: $\lambda \in {0, 0.5, 1.0, 2.0, 5.0}$ (estos valores son ilustrativos; podrían ser necesarios valores mayores dependiendo de la escala de penalidades). Aquí λ=0 ignora estética, λ=5 la prioriza fuertemente.

2. **Ejecutar ALNS para cada λ:** Para cada valor, ejecutar el algoritmo descrito en sección 6, usando la función objetivo Fitness_λ. Conviene dar suficientes iteraciones para que converja o mejore sustancialmente (por ejemplo 5000 iteraciones cada uno, o hasta X segundos). Registrar la mejor solución encontrada en cada caso.

3. **Recopilar resultados:** Supondremos obtenemos cinco soluciones $S_0, S_{0.5}, S_{1.0}, S_{2.0}, S_{5.0}$, con:

4. Costos: $C(S_i)$.
5. Penalidades: $P_{est}(S_i)$. Podemos tabular estos resultados, por ejemplo:

Escenario (λ)	Costo [monedas]	Penalidad estética	Características de la solución
0.0 (solo costo)	**X** (mínimo)	Alta	Rutas largas, mezcladas; uso de pocos vehículos.
0.5	X2	P est.2	… (un poco más estética)
1.0	X3	P est.3	… (balance medio)
2.0	X4	P est.4	… (más restricción estética, se nota en rutas)
5.0	X5 (máx)	**Baja** (mejor estética)	Rutas muy regionales, balanceadas, mayor costo.

(Reemplazar X por números reales obtenidos).

1. **Identificar Pareto:** Ordenamos por costo ascendente, y eliminamos soluciones que sean dominadas. Por ejemplo, si para λ=2.0 y λ=5.0 se obtuvieron penalidades similares pero λ=5.0 costó más, la solución de λ=5.0 sería dominada por la de λ=2.0 (hipotético). En general, con monótono aumento de λ es raro que ocurra dominancia cruzada, pero hay que revisarlo. Las que queden forman la frontera.

2. **Seleccionar 5 soluciones representativas:** Idealmente, nuestras 5 corridas ya son 5 puntos de la frontera. Si uno no lo fuera, podemos o ignorarlo o hacer otra corrida con otro λ para llenar ese hueco. Queremos cubrir desde el extremo de costo mínimo hasta el extremo de estética máxima. **Ejemplo:** S_0 (mínimo costo), $S_{0.5}, S_{1.0}, S_{2.0}, S_{5.0}$ (máximo atractivo) podría ya estar bien distribuidos.

Cada una de estas soluciones consiste en un conjunto completo de rutas asignadas. Llegados a este punto, conviene **presentar numéricamente cada solución**: podríamos listar, para cada vehículo usado, la secuencia de entregas. Por ejemplo, la solución S_{0} (costo mínimo) quizás use 20 vehículos y podríamos mostrar 20 rutas. Para no sobrecargar, podríamos resumir: indicar cuántos vehículos y destacar diferencias: - S_0: 18 vehículos usados, distancia total 300 km, penalidad estética 200 (2 cruces en rutas, 5 comunas duplicadas, etc). - $S_{5.0}$: 25 vehículos usados, distancia total 370 km, penalidad estética 50 (0 cruces, 0 comunas duplicadas, rutas balanceadas ±2 entregas).

El detalle de cada ruta se puede dar en anexos o visualmente en mapas, más que en texto plano, debido a la cantidad de puntos. Aun así, para completitud, se podría listar una ruta de ejemplo para cada solución: - En S_0, tal ruta quizás: Depot -> D5 -> D2 -> D17 -> ... -> Depot. - En $S_{5.0}$, rutas más cortas: Depot -> D5 -> D12 -> Depot (ejemplo, muy local).

El **punto central** es analizar el **impacto**: a medida que disminuye la penalidad (mejora estética), cómo crece el costo. Esta comparación se ilustra bien en la gráfica de Pareto.

Frontera de Pareto estimada que muestra el trade-off entre costo operativo y atractivo visual. Cada punto representa una solución distinta generada con ALNS para un peso λ diferente. En el eje horizontal está el costo total (por ejemplo en miles de unidades monetarias, incluyendo distancia y costos fijos) – hacia la izquierda es mejor (menor costo). En el eje vertical está una métrica de **atractivo visual** (por ejemplo un puntaje inverso de penalidad donde más arriba es más atractivo). La curva delineada por los puntos negros es la frontera de Pareto: obsérvese que para mejorar la estética (subir en el eje vertical) se tiene que aceptar un costo mayor (moverse a la derecha). Ninguna solución es claramente superior a las demás en ambos criterios: cada punto es óptimo en el sentido de Pareto (no dominado) [12] . Los decisores pueden elegir, por ejemplo, la solución del medio si desean un equilibrio, o una de los extremos si priorizan absolutamente uno de los objetivos.

9. Visualización de las Rutas y Resultados

Finalmente, una parte fundamental es **comunicar los resultados** de manera visual e intuitiva. Recomendamos dos formas de visualización:

a) Rutas dibujadas en un mapa o plano: Tomando las coordenadas de cada punto (depot y entregas) podemos trazar los recorridos de los vehículos. Lo ideal es usar un mapa de fondo de la ciudad (por ejemplo con Folium podemos superponer polilíneas sobre un mapa de OpenStreetMap). Sin embargo, si no se dispone de mapas interactivos, una gráfica cartesiana sirve: usar latitud vs longitud y dibujar líneas de ruta. En las figuras anteriores ya mostramos ejemplos comparando una solución de mínimo costo vs. una estética. Para un informe final, sería bueno graficar **cada una de las 5 soluciones seleccionadas** en mapas separados o subplots, cada uno con rutas en colores distintos. Así se puede apreciar cómo, conforme avanza la preferencia estética, las rutas van cambiando: - Menos cruces: los caminos de cada ruta se ven más “ordenados”, sin volverse sobre sí mismos. - Menos solapamiento: en el mapa se distingue que las áreas cubiertas por diferentes rutas están separadas (cada color ocupa una región del mapa diferente). - Comunas por ruta: podríamos dibujar los límites de comunas en el mapa si se tienen, para mostrar que en soluciones estéticas, cada ruta cubre 1-2 comunas, versus la solución de costo que quizás una ruta atraviese varias zonas distintas. - Balance: En mapas no se ve directamente, pero podemos anotar en la leyenda la longitud o número de entregas de cada ruta para mostrar que en la solución estética esas cifras están más cercanas entre rutas.

Al plotear, marcar el **centro de distribución** claramente (p. ej. un cuadrado o estrella negra) y usar colores contrastantes para rutas. Añadir quizás números o etiquetas a las rutas (Vehicle 1, 2, ...). Si el

plano está muy cargado (363 puntos es bastante), se puede mostrar solo un ejemplo de cada tipo. En informes escritos, a veces se muestran 2-3 soluciones representativas en el mapa: por ejemplo la solución puramente económica vs la puramente estética, para resaltar diferencias extremas.

b) Gráfica de la Frontera de Pareto: Como la insertada arriba, es imprescindible para entender cuantitativamente el trade-off. En el gráfico, cada solución está marcada; podríamos etiquetar cada punto con algo como "Cost = \$X, VisualScore = Y" o simplemente con el λ usado o un identificador de escenario. Esto ayuda a elegir. Por ejemplo, la curva podría mostrar que para mejorar el atractivo de, digamos, 60 a 80 (escala hipotética), el costo sube solo 5%, pero de 80 a 90 requiere un salto de 15% en costo. Esa información permitiría decir "lograr rutas *muy* estéticas es significativamente más costoso, quizás no vale la pena más allá de cierto punto".

Presentación de resultados numéricos: Además de las gráficas, un cuadro resumen de los 5 escenarios finalistas con sus métricas clave es útil (como se bosquejó antes). Incluir: - # de vehículos usados en cada solución. - Distancia total recorrida. - Costo total (distancia * costo_km + costos fijos). - # promedio de entregas por vehículo (y desviación). - # de cruces detectados. - # de comunas distintas por ruta (promedio o máximo). - % de solapamiento (por ejemplo, cuántos clientes estaban en comunas donde operan 2 vehículos).

Este resumen cuantifica el "atractivo visual" logrado en cada caso. Por ejemplo, en la solución extrema estética: cruces = 0, solapamiento = 0, max comunas/ruta = 2, desviación entregas = baja; comparado con la solución de costo: cruces = 3, solapamiento en 4 comunas, max comunas/ruta = 5, etc. Así, el equipo podrá entender qué se sacrifica o gana con cada modelo.

10. Conclusiones y Recomendaciones Finales

Siguiendo este instructivo, habrás implementado un algoritmo ALNS multiobjetivo en Python capaz de **generar rutas de vehículos optimizando dos criterios**. Repasemos los puntos principales y algunas recomendaciones finales:

- Empleamos ALNS debido a su flexibilidad y potencia para VRP grandes, donde algoritmos exactos no escalan [11] . Partimos de una solución inicial heurística y aplicamos operadores de destrucción/reparación adaptativos para explorar numerosas configuraciones de rutas.
- Incorporamos las preferencias de **atractivo visual** mediante penalizaciones en la función objetivo, penalizando rutas con características indeseadas (cruces, dispersiones, solapes).
- Para manejar la **naturaleza multiobjetivo**, corrimos el algoritmo con distintos pesos λ en la función de fitness combinada. Esto nos dio un conjunto de soluciones de compromiso, de las cuales extrajimos 5 representativas en la frontera de Pareto. Cada una equilibra costo vs. estética de forma distinta, permitiendo al usuario escoger la más adecuada a sus prioridades.
- Visualizamos los resultados dibujando las rutas en el mapa, lo cual facilita verificar empíricamente el atractivo visual (por ejemplo, se puede *ver* que no haya rutas cruzándose ni sobre la misma zona). La gráfica de Pareto complementa esto mostrando cuantitativamente el costo de imponer dichas consideraciones estéticas.
- En una transición real, probablemente se optaría por un modelo "intermedio", no el más extremo en estética ni el más barato, de modo que se ahorren costos con respecto al método antiguo pero manteniendo rutas razonables para los transportistas. Gracias a la frontera obtenida, se puede **cuantificar** cuántos pesos (dinero, kms, vehículos extra) cuesta lograr cierta mejora visual. Esto es invaluable para la toma de decisiones informada.

Recomendaciones: Al implementar, presta atención a la calibración de parámetros: - Tamaño de remoción (porcentaje de clientes removidos en cada destrucción) y tipos de operadores: influye en la

eficacia de búsqueda. - Parámetros del criterio de aceptación (temperatura inicial, factor de enfriamiento) para equilibrar exploración/explotación. - Pesos de penalización estética: si son demasiado bajos, ALNS ignorará estética; si son demasiado altos, puede sacrificar excesivo costo. Busca un punto donde la penalización estética de una solución malísima (muchos cruces, etc.) sea del orden del incremento de costo que deseas evitar. - Tiempo de cómputo: ALNS es estocástico; se recomienda ejecutar varias veces si posible y escoger la mejor, o al menos usar una semilla aleatoria fija para reproducibilidad al comparar escenarios.

Por último, aunque nuestro enfoque fue específico, la estructura se puede extender o modificar. Si en el futuro se quisiera agregar otro objetivo (ej. minimización de emisiones de CO2) o más restricciones (ej. vehículos con distintas velocidades, tráfico variable), se pueden ajustar las funciones de costo/penalidad e incluso incluir nuevos operadores especializados (por ejemplo, para cambiar asignación de tipos de vehículo a rutas).

Con esto concluye el instructivo. Tendrás ahora en tus manos un conjunto de rutas optimizadas y equilibradas: un **modelo de ruteo multiobjetivo con ALNS** que proporciona soluciones eficientes en costo y operativamente amigables en el terreno visual. ¡Buena suerte con la implementación y la puesta en marcha de estas ideas!

---

1 2 8 9 11 Solving the Vehicle Routing Problem (VRP) in Python with Gurobi & Adaptive Large Neighborhood Search (ALNS) | by Jewis.Y, PhD | Medium
https://medium.com/@jyopt/solving-the-vehicle-routing-problem-vrp-with-gurobi-alns-from-exact-optimization-to-322bcc57a33f

3 4 12 Eficiencia de Pareto - Wikipedia, la enciclopedia libre
https://es.wikipedia.org/wiki/Eficiencia_de_Pareto

5 6 10 — ALNS 7.0.0 documentation
https://alns.readthedocs.io/en/latest/

7 P5-Ruteo+con+atractivo+visual.pdf
file://file-8dvNKCqLTL4PmZT1osuhZd