Informe Técnico: Heurísticas y Metaheurísticas para el Problema de Ruteo de Vehículos con Capacidad (CVRP)

Daniel Puyol
Mateo Tejera
Agustín Rodríguez

8 de Julio de 2025

Tecnología Digital V: Diseño de Algoritmos
Licenciatura en Tecnología Digital
Primer Semestre 2025

Índice

1. Introducción al Problema

El Problema de Ruteo de Vehículos con Capacidad (CVRP) es un problema de optimización combinatoria fundamental y ampliamente estudiado, con profundas implicaciones en la logística y la gestión de la cadena de suministro. Su objetivo principal es diseñar un conjunto de rutas óptimas para una flota de vehículos que parten y regresan a un depósito central, con el fin de satisfacer las demandas de un conjunto de clientes geográficamente dispersos. Cada vehículo posee una capacidad máxima uniforme, y la suma de las demandas de los clientes asignados a una ruta no debe exceder dicha capacidad. Además, cada cliente debe ser visitado exactamente una vez. El objetivo final es minimizar el costo total de las rutas, que generalmente se traduce en la distancia total recorrida.

El CVRP es conocido por ser un problema NP-hard, lo que implica que encontrar la solución óptima para instancias de gran tamaño es computacionalmente inviable en un tiempo razonable. Debido a esto, se recurre a heurísticas y metaheurísticas, con el fin de buscar soluciones que se aproximen al óptimo de una forma mucho menos costosa computacionalmente y más eficiente.

El trabajo se basa en la implementación, combinación y evaluación de diversas heurísticas y una metaheurística para resolver el CVRP.

2. Descripción de los Algoritmos Implementados

Para resolver el CVRP, se han implementado distintas estrategias, clasificadas en heurísticas constructivas, operadores de búsqueda local y una metaheurística.

2.1. Heurísticas Constructivas

Mediante este método se busca generar una solución inicial desde cero.

2.1.1. Heurística del Vecino Más Cercano (VMC)

- **Es una heurística *greedy* que construye rutas de forma secuencial.** Desde el nodo actual, siempre selecciona el cliente no visitado más cercano que cumpla con la restricción de capacidad del vehículo.
- **Funcionamiento:**
  1. Se inicializa una ruta en el depósito.
  2. Mientras queden clientes sin visitar y la capacidad del vehículo lo permita, se busca el cliente no visitado más cercano al nodo actual.
  3. Si se encuentra un cliente factible, se añade a la ruta, se actualiza la capacidad restante del vehículo y se marca como visitado.
  4. Si no se pueden añadir más clientes a la ruta actual (porque no hay clientes cercanos o porque la capacidad se ha agotado), la ruta se cierra (regresando al depósito) y se añade a la solución.
  5. Si aún quedan clientes sin visitar, se inicia una nueva ruta desde el depósito con un nuevo vehículo.

6. El proceso se repite hasta que todos los clientes han sido visitados.

- **Complejidad:** La complejidad principal radica en la búsqueda del vecino más cercano para cada cliente en cada paso. Si hay N clientes, en el peor caso, se realizan $\mathcal{O}(N)$ búsquedas, y cada búsqueda puede implicar $\mathcal{O}(N)$ comparaciones. Por lo tanto, la complejidad es aproximadamente $\mathcal{O}(N^2)$.

2.1.2. Heurística de Savings de Clarke y Wright (CAW)

- **Esta heurística busca reducir la distancia total al combinar rutas.** Comienza con rutas individuales para cada cliente y las fusiona de manera que se maximice el ahorro de distancia.
- **Funcionamiento:**
  1. Se crean rutas individuales para cada cliente, donde cada ruta va del depósito al cliente y regresa al depósito.
  2. Para cada par de clientes (i, j), se calcula el ahorro que se obtendría al fusionar sus rutas individuales en una sola ruta que los conecte directamente.
  3. Los ahorros se ordenan en forma decreciente.
  4. Se itera a través de la lista de ahorros ordenados:
     - Para cada par (i, j), se identifican las rutas actuales que contienen a i y j (donde i y j deben ser adyacentes al depósito en sus respectivas rutas).
     - Si i y j están en rutas diferentes y la fusión es factible (la suma de las demandas de ambas rutas no excede la capacidad del vehículo), se fusionan las dos rutas. La fusión implica eliminar el depósito intermedio y unir las rutas.
     - Las rutas fusionadas se actualizan y las rutas originales se eliminan.
  5. El proceso continúa hasta que no se pueden realizar más fusiones factibles.
- **Complejidad:** El cálculo de ahorros es $\mathcal{O}(N^2)$. El ordenamiento de los ahorros es $\mathcal{O}(N^2 \log N^2) = \mathcal{O}(N^2 \log N)$. La fase de fusión implica iterar sobre los ahorros y buscar rutas, lo que puede ser $\mathcal{O}(N^2 \cdot N) = \mathcal{O}(N^3)$ en el peor caso si la búsqueda de rutas es lineal. En general, es más eficiente que $\mathcal{O}(N^3)$ con estructuras de datos adecuadas, pero sigue siendo polinomial.

2.2. Operadores de Búsqueda Local

Los operadores de búsqueda local ayudan a mejorar una solución existente realizando pequeños cambios en ella.

2.2.1. Operador Swap

Intenta mejorar una solución intercambiando dos clientes entre dos rutas diferentes.

Funcionamiento:

1. Se itera sobre todos los pares de rutas en la solución.

2. Para cada par de rutas, se consideran todos los pares de clientes (uno de cada ruta, excluyendo los depósitos).
3. Para cada posible intercambio, se verifica si es factible (es decir, si las demandas de las rutas resultantes no exceden la capacidad del vehículo).
4. Si es factible, se calcula la diferencia de costo que resultaría del intercambio.
5. Si se encuentra un intercambio que mejora la solución, se realiza el intercambio y se actualiza el costo total de la solución.

Complejidad: Si hay K rutas y N clientes en total, y cada ruta tiene en promedio N/K clientes, la complejidad de explorar todos los intercambios es aproximadamente:

$$\mathcal{O}\left(K^2 \cdot \left(\frac{N}{K}\right)^2\right) = \mathcal{O}(N^2)$$

2.2.2. Operador Relocate

- **Intenta mejorar una solución moviendo un cliente de una ruta a otra.**
- **Funcionamiento:**
  1. Se itera sobre todos los pares de rutas en la solución.
  2. Para cada par de rutas (una ruta de origen y una ruta de destino), se considera cada cliente en la ruta de origen (excluyendo el depósito).
  3. Se verifica si mover el cliente de la ruta de origen a la ruta de destino es factible (si la demanda del cliente cabe en la capacidad restante de la ruta de destino).
  4. Si es factible, se consideran todas las posibles posiciones de inserción en la ruta de destino.
  5. Se calcula la diferencia de costo que resultaría del movimiento.
  6. Se aplica una estrategia "Best Improvement": se busca el movimiento que genere la mayor reducción de costo.
  7. Si se encuentra un movimiento que mejora la solución, se realiza el movimiento y se actualiza el costo total de la solución.
- **Complejidad:** Es similar al operador Swap, la complejidad es aproximadamente $\mathcal{O}(N^2)$ en el peor caso.

2.3. Metaheurística

2.3.1. GRASP (Greedy Randomized Adaptive Search Procedure)

- **Concepto:** GRASP es una metaheurística iterativa que combina una fase *greedy* aleatorizada con una fase de búsqueda local.
- **Funcionamiento:**
  1. **Fase Constructiva Aleatorizada:**

- En lugar de elegir el mejor cliente de forma puramente *greedy* (como en VMC), se construye una Lista de Candidatos Restringidos (RCL). La RCL contiene los k mejores clientes que son factibles para añadir a la ruta actual.
- Se selecciona de forma randomizada un cliente de la RCL para añadirlo a la ruta. Esto introduce la aleatorización y permite explorar diferentes soluciones iniciales.
- En nuestra implementación, la fase constructiva se basa en una versión aleatorizada de la heurística del Vecino Más Cercano, donde `rcl` define el tamaño de la RCL.

2. **Fase de Búsqueda Local:**
   - Una vez que se ha construido una solución inicial, se aplica un operador de búsqueda local (en nuestro caso, el operador Swap) para mejorarla. La búsqueda local explora el vecindario de la solución actual hasta encontrar un óptimo local.

   Iteraciones: Las fases constructiva y de búsqueda local se repiten un número predefinido de `iters` veces. **Mejor Solución:** GRASP mantiene un registro de la mejor solución encontrada a lo largo de todas las iteraciones. La solución final es la mejor solución global y la que se devuelve.

- **Complejidad:** La complejidad de GRASP dependerá de:
  - número de iteraciones
  - la complejidad de la fase constructiva (que incluye la construcción de la RCL)
  - la complejidad de la búsqueda local.

  Si la fase constructiva es $\mathcal{O}(N^2)$ y la búsqueda local es $\mathcal{O}(N^2)$, entonces la complejidad total es $\mathcal{O}(N^2)$.

3. Decisiones de Diseño

Para la resolución del problema del CVRP se adoptó un enfoque de programación modular y orientado a objetos utilizando el lenguaje `C++`. Las decisiones de diseño se centraron en garantizar la legibilidad, reutilización del código y facilidad de mantenimiento.

- **Modularidad y orientación a objetos:** Desde el inicio se definió una estructura basada en clases, asignando a cada componente del algoritmo (rutas, soluciones, operadores, heurísticas, etc.) una entidad propia con responsabilidad bien definida.
- **Separación entre lógica y control:** Se pensó el sistema dividiendo claramente entre los algoritmos (encapsulados en clases propias) y el flujo principal del programa, que se encarga de configurar, ejecutar y registrar los resultados.
- **Adaptabilidad del sistema:** Uno de los objetivos clave del diseño fue permitir la fácil incorporación de nuevas heurísticas, operadores o metaheurísticas en el futuro, manteniendo una interfaz coherente y reutilizable.
- **Configurabilidad:** Desde el diseño se decidió permitir la modificación de parámetros críticos (como iteraciones, tamaño de RCL, tipo de heurística base o operador local) sin alterar el código fuente, lo cual se logró usando funciones parametrizadas desde el `main`.
- **Validación y robustez:** Se estableció como criterio de diseño que cada módulo controle la validez de sus propios datos: por ejemplo, que las rutas no excedan la capacidad, que todos los clientes sean visitados una vez, o que las soluciones sean evaluables en cualquier momento.

4. Implementación

La implementación concreta del sistema refleja las decisiones de diseño descritas anteriormente. Se utilizó el lenguaje `C++` y se dividió el código en archivos de cabecera (`.h`) y de implementación (`.cpp`) para separar las declaraciones de las definiciones funcionales.

- **Estructura de archivos:** El proyecto se organiza en los siguientes módulos:
  - `Ruta.h / Ruta.cpp`: clase que modela una ruta individual.
  - `Solucion.h / Solucion.cpp`: representa una solución completa del problema.
  - `CAW.cpp` y `VMC.cpp`: implementan heurísticas constructivas.
  - `swap.cpp` y `relocate.cpp`: operadores de búsqueda local.
  - `GRASP.h / GRASP.cpp`: encapsulan la metaheurística GRASP.
  - `Archivos main.cpp(sin incluir el provisto por la cátedra)`: se usaron para probar y ejecutar los códigos y luego realizar la experimentación
- **Lectura y procesamiento de datos:** Los datos de entrada se leen desde archivos externos y se procesan para generar matrices de distancias y estructuras con coordenadas, demandas y capacidades. Esta información es utilizada luego por los algoritmos.
- **Ejecución de algoritmos:** Desde el `main` se configura el algoritmo deseado y se ejecuta el procedimiento correspondiente. Cada componente está diseñado para funcionar de manera autónoma y compatible con el resto del sistema.
- **Salida y análisis:** Los resultados se imprimen por pantalla y se escriben en un archivo `.csv`, permitiendo su posterior análisis. Este archivo incluye métricas como el costo total, el número de rutas y la composición de cada una.
- **Parámetros configurables:** El tamaño de la lista restringida (RCL), la cantidad de iteraciones, el tipo de operador y la heurística base son ajustables desde el flujo de ejecución principal, sin necesidad de modificar las clases internas.

Quien desee ejecutar los algoritmos debe realizarlo con el siguiente comando por terminal, incluyendo todos los archivos .h en su código `main.cpp`:

```
g++ VRPLIBReader.cpp Ruta.cpp Solucion.cpp VMC.cpp CAW.cpp swap.cpp relocate.cpp
GRASP.cpp main.cpp -o main
```

Se deben declarar la instancia VRPLIBReader, el algoritmo a usar y la clase Solucion. Si se usa GRASP se deben pasar el valor RCL y las iteraciones deseadas. Ejemplo:

```
VRPLIBReader instance(ruta del archivo);

VMC algoritmo;

Solucion sol = algoritmo.resolver(instance);
```

5. Experimentación

Como base para evaluar los distintos métodos, se procedió a ejecutarlos para comparar los resultados (costos) que devolvería cada combinación de algoritmos y el tiempo en que lo harían.

Esto se hizo dentro del mismo archivo `main2.cpp` probando las distintas combinaciones y los resultados obtenidos son:

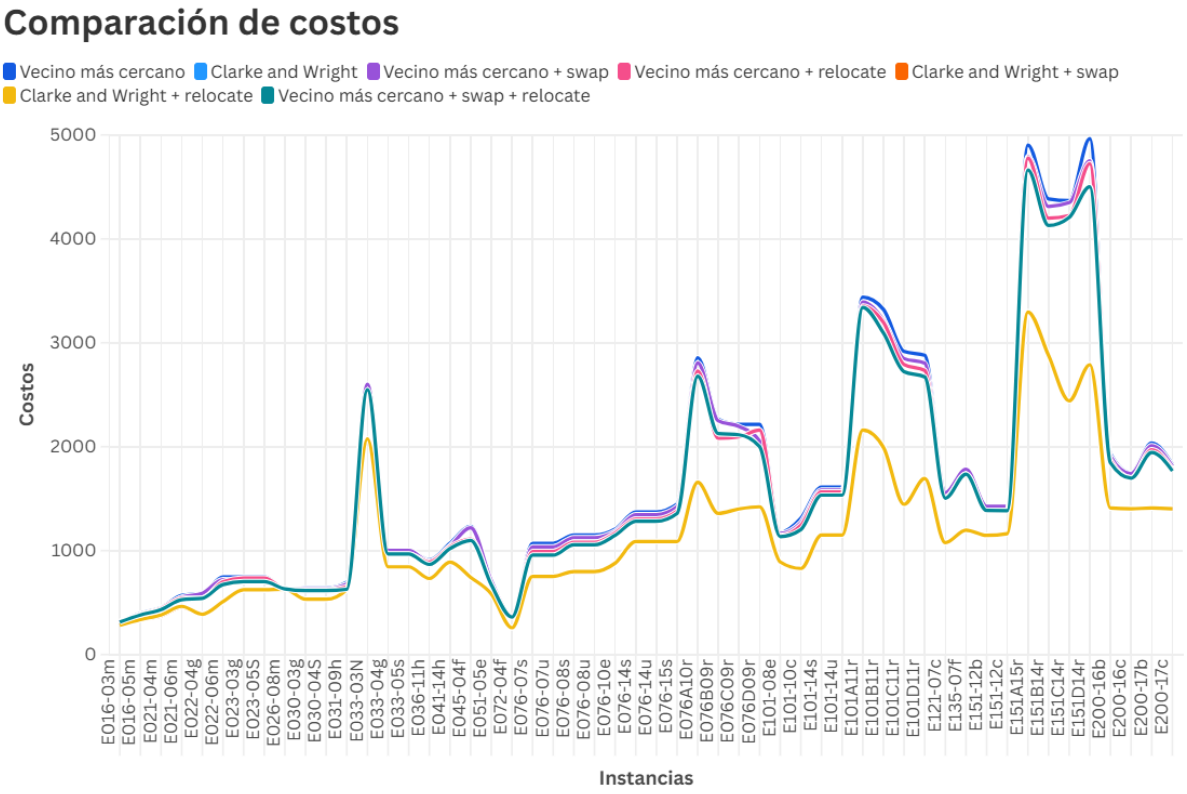

Figura 1: Comparación de costos de las soluciones obtenidas.

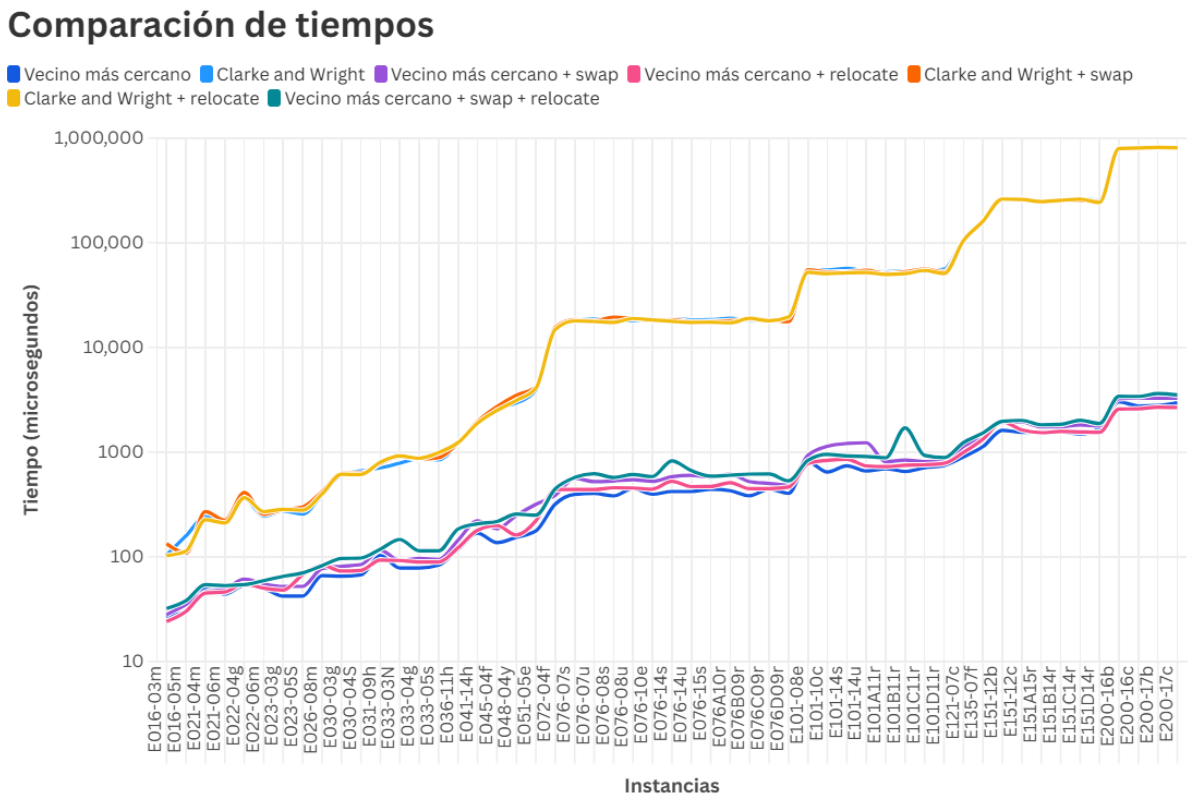

Figura 2: Comparación de tiempos de ejecución entre métodos.

Puede apreciarse una relación lógica entre los resultados obtenidos y los tiempos de ejecución. Clarke and Wright, con y sin operadores de búsqueda local, devuelve soluciones de mejor calidad a costo de tardar un tiempo visiblemente mayor que Vecino más Cercano. Ambos aumentan el tiempo de cómputo a medida que el tamaño de las instancias crecen, pero Clarke and Wright termina siempre con mejores soluciones que Vecino más Cercano, incluso usando operadores de búsqueda local con este último.

Aún así, se notó que VMC mejora siempre al usar un operador de búsqueda, mientras que CAW obtiene mejoras despreciables. Esto hizo que se plantee la posibilidad de que, si se aplican varias veces, VMC podría obtener soluciones de mayor calidad e incluso, casi igualando a las de CAW. Por esto, se planteó un experimento en el que a las soluciones de VMC se le aplicarían 100 veces operadores locales. Esta decisión de iteraciones fue arbitraria, pero se hizo con el fin de confirmar la hipótesis. Se obtuvieron los siguientes resultados:

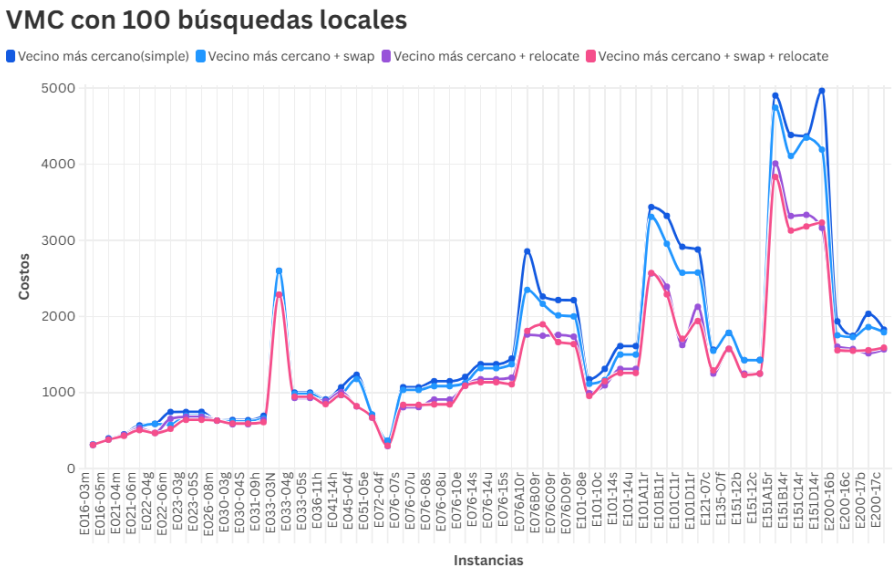

Figura 3: Comparación de costos de VMC al hacer 100 búsquedas locales

Puede comprobarse que efectivamente si se realizan muchas búsquedas locales, las soluciones de VMC mejoran notablemente que si no se hacen o si se realizan en escasa cantidad. Incluso una combinación de arroja resultados de buena calidad.

Para finalizar, se deseaba poner a prueba la efectividad de la metaheurística GRASP, para determinar qué tan útil y confiable resulta la randomización para escapar de óptimos locales en vecindarios de operadores de búsqueda local. Para esto, se fijó el valor de RCL en 4 (decisión arbitraria) e ir variando la cantidad de iteraciones en 10,100,1000 y 10.000. Se obtuvieron los siguientes resultados.

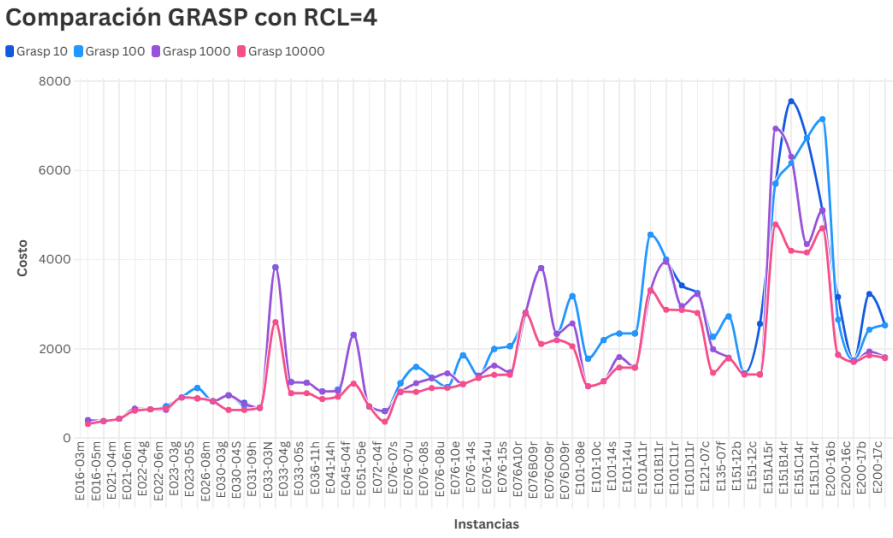

Figura 4: Costos con distinta cantidad de iteraciones en GRASP

Figura 5: Tiempos para las distintas iteraciones de GRASP

Puede apreciarse una vez más, que al aumentar la cantidad de ejecuciones se obtienen soluciones de mejor calidad a cambio de un mayor tiempo de cómputo, notándose la amplia diferencia de tiempo en 10.000 iteraciones. Esto puede deberse a que al aumentar la cantidad de intentos de obtener una solución buena, es mayor la posibilidad de obtener una y no dejar que la randomización influya en la búsqueda de la misma.

No obstante, las soluciones obtenidas no superaban las de Clarke and Wright, por lo que en un intento de lograrlo, se optó por añadir otra vez muchas búsquedas locales usando `relocate`, que ha mostrado obtener mejores soluciones que `swap`. Los resultados obtenidos, con dos cantidades de relocate, fueron los siguientes:

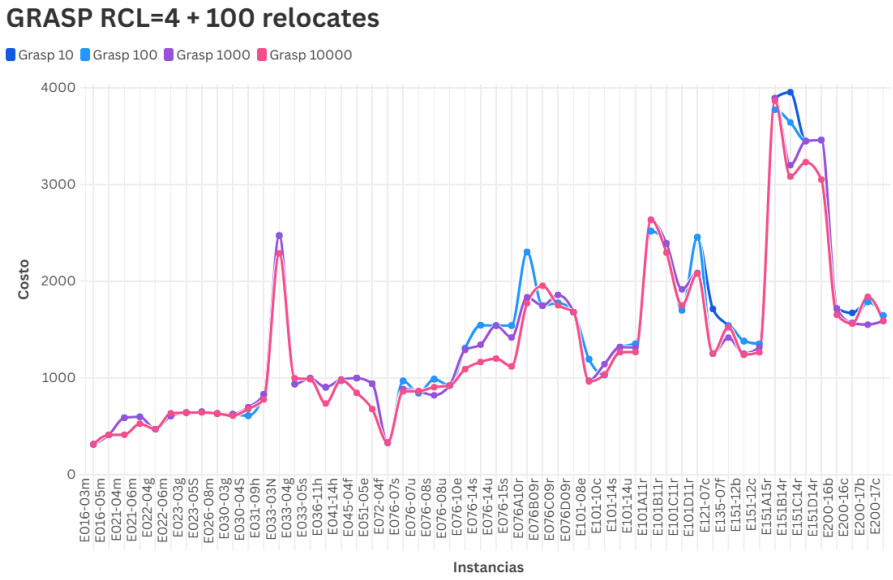

Figura 6: Aplicar relocate 100 veces a las soluciones obtenidas en GRASP

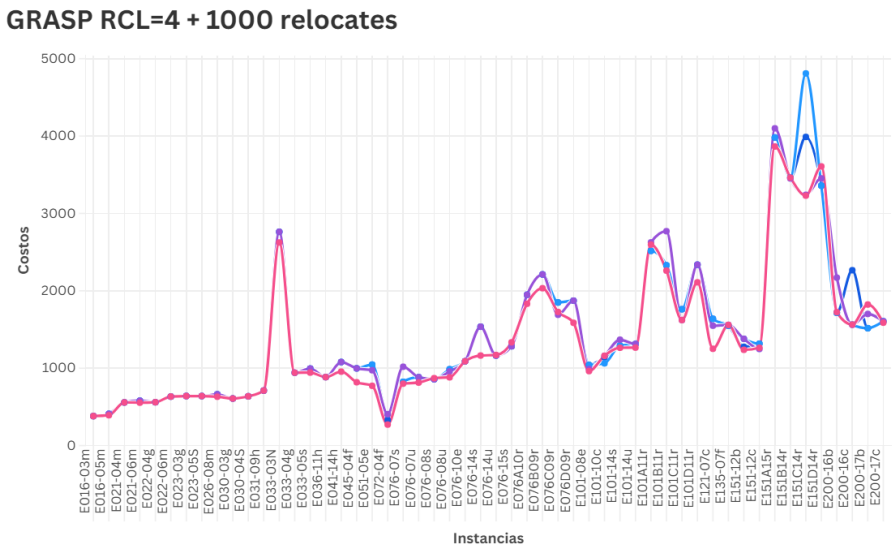

Figura 7: Aplicar relocate 1000 veces a las soluciones obtenidas en GRASP

Por lo observado, las soluciones al aplicar `relocate` obtienen notables mejoras, ayudando a GRASP en la búsqueda de respuestas de calidad. Aún así, todavía parece que la randomización se hace notar en algunos momentos, afectando el objetivo de encontrar la mejor solución posible.

6. Conclusiones

A lo largo de este trabajo se abordó el Problema de Ruteo de Vehículos con Capacidad (CVRP) mediante la implementación y evaluación de diversas heurísticas constructivas, operadores de búsqueda local y la metaheurística GRASP. Los resultados obtenidos permiten extraer varias conclusiones relevantes:

- **Calidad vs. Tiempo de Cómputo:** Se observó una relación consistente entre la calidad de las soluciones y el tiempo de ejecución. Las heurísticas más simples, como el Vecino Más Cercano (VMC), producen soluciones rápidamente pero de menor calidad. Por el contrario, métodos como Clarke and Wright (CAW) ofrecen soluciones de mejor calidad, aunque requieren tiempos de ejecución más elevados.
- **Importancia de la Búsqueda Local:** La aplicación de operadores de búsqueda local, como `swap` y `relocate`, mejora significativamente la calidad de las soluciones. Esto es especialmente notorio en el caso de VMC, donde aplicar múltiples iteraciones de búsqueda local permite obtener soluciones comparables e incluso cercanas a las de CAW.
- **Efectividad de GRASP:** La metaheurística GRASP demostró ser una herramienta eficaz para diversificar la búsqueda y escapar de óptimos locales. Se confirmó que incrementar el número de iteraciones mejora la calidad de las soluciones, aunque a costa de un aumento considerable en los tiempos de ejecución.
- **GRASP y Búsqueda Local Intensiva:** Al aplicar intensivamente operadores como `relocate` sobre las soluciones generadas por GRASP, se logró una mejora sustancial en la calidad de las rutas obtenidas. Esta combinación permitió alcanzar soluciones de buena

calidad, aunque en algunos casos la aleatorización inherente a GRASP seguía afectando los resultados finales.

- **Balance entre Exploración y Explotación:** Los experimentos evidenciaron la importancia de equilibrar la exploración (randomización y diversidad de soluciones) con la explotación (búsqueda local intensiva). Si bien las soluciones puramente greedy pueden ser rápidas, las soluciones obtenidas mediante procesos iterativos y mejoradas localmente son claramente superiores en calidad.
- **Limitaciones y Perspectivas Futuras:** A pesar de los avances, algunas soluciones no lograron superar a las generadas por CAW. Esto sugiere que, en futuras investigaciones, podría explorarse la hibridación de métodos, la incorporación de nuevas metaheurísticas o la optimización de los parámetros de GRASP para obtener mejores resultados sin incurrir en tiempos de cómputo excesivos.

En definitiva, el trabajo demostró cómo la aplicación inteligente y combinada de heurísticas, búsqueda local y metaheurísticas permite abordar de forma efectiva un problema complejo como el CVRP, logrando soluciones de buena calidad con tiempos de ejecución razonables según los objetivos y restricciones del contexto.

A tener en cuenta:

- Para la comparación de soluciones, la instancia E048-04y fue omitida dado que, para todos los algoritmos, su costo era particularmente alto y no permitía tener una buena visualización para comparar los resultados.
- Dado que Clarke and Wright no toma en cuenta la restricción de la cantidad de automóviles, la misma tuvo que ser relajada en las siguientes instancias: E016-03m; E016-05m; E021-06m; E022-06m; E026-08m; E030-03g; E031-09h; E036-11h; E041-14h; E051-05e; E072-04f; E076-14s; E076-14u; E076B09r; E076C09r; E076D09r; E101A11r; E101D11r; E151A15r; E151C14r; E151D14r; E200-16b; E200-16c.