TP 2: Heurísticas para el Problema de Ruteo de Vehículos con Capacidad (CVRP)

Integrantes:
Camus Sol
Luchetti Olivia
Martinez Kiara

8 de julio de 2025

Índice

1. Introducción

El Problema de Ruteo de Vehículos con Capacidad (CVRP, por sus siglas en inglés) es un problema clásico de optimización combinatoria con aplicaciones directas en logística, distribución y transporte. El objetivo principal es diseñar rutas para una flota de vehículos idénticos, cada uno con una capacidad limitada, que deben atender a un conjunto de clientes dispersos geográficamente, partiendo y regresando a un depósito central. Cada cliente tiene una demanda conocida y debe ser atendido exactamente una vez, sin exceder la capacidad de los vehículos.

El CVRP es un problema NP-hard, lo que implica que no existen algoritmos exactos eficientes para resolver instancias de gran tamaño en tiempo razonable. Por esta razón, se recurre a heurísticas y metaheurísticas que permiten obtener soluciones de buena calidad en tiempos computacionales aceptables. En este trabajo, se exploran e implementan diferentes enfoques heurísticos para abordar el CVRP, analizando su desempeño y las características de las soluciones obtenidas.

1.1. Definición Formal

Dado un grafo $G = (V, E)$, donde $V = \{0, ..., n\}$ es el conjunto de nodos (0 es el depósito) y E es el conjunto de aristas. Cada cliente $i \in V \setminus \{0\}$ tiene una demanda q_i, y la distancia entre nodos i y j es d_{ij}. El objetivo es encontrar un conjunto de rutas de mínimo costo total para una flota de vehículos de capacidad Q, tal que cada cliente sea visitado exactamente una vez.

2. Heurísticas Constructivas Implementadas

2.1. Algoritmo de Ahorros de Clarke & Wright

Es una de las heurísticas más conocidas. Parte de una solución inicial donde cada cliente es servido individualmente y luego fusiona rutas de forma iterativa si esto genera un .[a]horro.[en] la distancia total.

2.1.1. Concepto de Ahorros

El ahorro s_{ij} al conectar los clientes i y j se calcula como: $s_{ij} = d_{0i} + d_{0j} - d_{ij}$. Representa la distancia que se ahorra al no tener que volver al depósito entre las visitas a i y j.

2.1.2. Análisis de Complejidad

- **Cálculo de ahorros:** Se calculan para cada par de nodos, resultando en una complejidad de $O(n^2)$.
- **Ordenamiento de ahorros:** Ordenar la lista de ahorros tiene una complejidad de $O(n^2 \log n)$.
- **Construcción de rutas:** El bucle principal itera sobre la lista de ahorros. Con estructuras de datos optimizadas (como un mapa para localizar nodos), las operaciones internas son rápidas. La complejidad total está dominada por el ordenamiento: $O(n^2 \log n)$.

2.2. Algoritmo del Vecino Más Cercano (Nearest Neighbor)

Es una heurística voraz que construye rutas secuencialmente. Desde el último nodo añadido a una ruta, se selecciona el cliente no visitado más cercano que no viole la restricción de capacidad.

2.2.1. Variante Implementada

Nuestra implementación evalúa una lista restringida de los 5 candidatos más cercanos para añadir a la ruta, en lugar de solo el más cercano, para diversificar la construcción.

2.2.2. Análisis de Complejidad

En cada paso de la construcción de una ruta, se busca el vecino más cercano entre los nodos no visitados. En el peor caso, esto se repite para cada uno de los n nodos, resultando en una complejidad de $O(n^2)$.

3. Operadores de Búsqueda Local

3.1. Descripción de los Operadores

Con el objetivo de mejorar las soluciones iniciales generadas por las heurísticas constructivas, implementamos operadores de búsqueda local clásicos, específicamente los operadores de reubicación (relocate) e intercambio (swap).

El operador de **relocate** consiste en seleccionar un cliente de una ruta y moverlo a otra posición, ya sea dentro de la misma ruta o en una ruta diferente, siempre que se respete la restricción de capacidad de los vehículos. En nuestra implementación, se exploran todas las posibles reubicaciones y se aplica la primera que produce una mejora en el costo total de la solución. Este proceso se repite de manera iterativa hasta que no se encuentran más movimientos beneficiosos.

Por otro lado, el operador de **swap** selecciona dos clientes, que pueden pertenecer a rutas distintas o a la misma ruta, y los intercambia de lugar. Antes de realizar el intercambio, se verifica que las demandas resultantes en ambas rutas no superen la capacidad máxima permitida. Si el intercambio resulta en una reducción del costo total, se realiza el cambio y se actualizan las rutas afectadas.

Ambos operadores se aplican de forma alternada y repetida, permitiendo así refinar progresivamente la solución inicial. Esta estrategia de búsqueda local nos permitió obtener soluciones de mejor calidad, acercándonos más a los óptimos de referencia y superando las limitaciones de las heurísticas constructivas puras. La implementación se diseñó para ser eficiente, priorizando la rapidez en la evaluación de movimientos y la actualización de las rutas tras cada modificación.

Impacto en los resultados: La incorporación de estos operadores de búsqueda local tuvo un efecto significativo en la calidad de las soluciones obtenidas. En la mayoría de los casos, se logró reducir el costo total de las rutas y, en ocasiones, disminuir la cantidad de vehículos necesarios. Esto demuestra la importancia de complementar las heurísticas constructivas con técnicas de mejora local para abordar problemas complejos como el CVRP.

3.1.1. Análisis de Complejidad

El costo computacional de los operadores de búsqueda local depende principalmente del número de rutas (R) y del número máximo de clientes por ruta (N).

- **Relocate:** La complejidad por iteración es $O(R^2N^2)$, ya que para cada par de rutas se consideran todas las posiciones posibles para mover un cliente de una ruta a otra.
- **Swap:** La complejidad por iteración también es $O(R^2N^2)$, dado que se evalúan todos los pares posibles de clientes entre todas las rutas.

En la práctica, si las rutas están balanceadas, el número de clientes por ruta suele ser aproximadamente $N \approx n/R$, donde n es el número total de clientes. Sin embargo, la cota teórica general sigue siendo $O(R^2N^2)$. Además, como la implementación detiene la búsqueda al encontrar la primera mejora, el tiempo promedio de ejecución suele ser menor que el peor caso, aunque para instancias grandes el costo computacional puede seguir siendo significativo si se buscan mejoras exhaustivas en cada iteración.

4. Métodos Híbridos

Para la generación de soluciones iniciales, se implementó un método que combina una heurística constructiva clásica con una fase de mejora local básica. En particular, utilizamos el algoritmo de **Clarke & Wright** para construir una primera solución factible, aprovechando su eficiencia y capacidad para generar rutas de buena calidad en poco tiempo.

Una vez obtenida la solución inicial, se aplica de manera iterativa el operador de **relocate** como técnica de búsqueda local. Este proceso consiste en intentar mover clientes entre rutas para reducir el costo total, repitiendo la operación hasta que no se logran más mejoras o se alcanza un número máximo de iteraciones. De esta forma, se refina la solución constructiva y se incrementa la calidad de los resultados obtenidos.

Decidimos esta estrategia híbrida porque nos permite partir de una solución razonable y mejorarla rápidamente, combinando la rapidez de las heurísticas constructivas con la capacidad de ajuste fino de la búsqueda local. En la práctica, observamos que este enfoque logra soluciones competitivas en tiempos computacionales reducidos.

4.0.1. Análisis de Complejidad

- **Clarke & Wright:** La complejidad es $O(n^2 \log n)$, ya que se calcula la matriz de ahorros y se ordenan los ahorros.
- **Relocate:** La complejidad es $O(R^2N^2)$, ya que se evalúan todas las posiciones posibles para mover un cliente de una ruta a otra.

Complejidad total: $O(n^2 \log n + R^2N^2)$ (Clarke & Wright + Relocate)

5. Consideraciones de Implementación y Dificultades Encontradas

El proyecto fue desarrollado en C++11, priorizando la eficiencia en el manejo de memoria y el rendimiento computacional.

Una de las principales dificultades encontradas fue el procesamiento y parsing de las instancias en formato VRPLIB, que requirió el desarrollo de una clase específica para la lectura robusta de los datos. Además, la validación de las restricciones de capacidad y la correcta construcción de las rutas demandó una cuidadosa gestión de las estructuras de datos, especialmente al fusionar rutas o al seleccionar el siguiente cliente en las heurísticas.

Otra dificultad relevante fue la depuración y validación de los resultados obtenidos, ya que fue necesario comparar las soluciones generadas con resultados de referencia y analizar casos particulares donde las heurísticas no lograban cumplir todas las restricciones. Finalmente, se optimizaron ciertas operaciones críticas, como la búsqueda de vecinos más cercanos y la actualización de rutas, para reducir el tiempo de ejecución en instancias de mayor tamaño.

6. Comandos para compilar y ejecutar el proyecto

6.1. Usando make

```
# Compilar el proyecto
make

# Limpiar archivos compilados
make clean
```

6.2. Ejecutando el proyecto

```
./bin/cvrp_solver.exe <instancia>
```

Luego de la ejecución, se dispondrá un menú con las siguientes opciones:

- **1:** Ejecutar el algoritmo de Clarke & Wright.
- **2:** Ejecutar el algoritmo de Nearest Neighbor.
- **3:** Ejecutar el algoritmo de GRASP.

Si se selecciona alguna de las primeras dos opciones, se dispondrá nuevamente otro menú con las siguientes opciones:

- **0:** No aplicar búsqueda local
- **1:** Aplicar operador Relocate
- **2:** Aplicar operador Swap
- **3:** Aplicar ambos operadores (Relocate + Swap)

6.2.1. Ejemplos de ejecución

```
./bin/cvrp_solver.exe instances/E045-04f.vrp
```

6.3. Ejecutando los tests

```
./bin/test_local_search.exe  (o ./bin/test_solution_reader.exe)
```

6.4. Clases Principales

A continuación se describen las clases principales desarrolladas en el proyecto:

Clase	**Descripción**
VRPLIBReader	Responsable de leer y parsear las instancias desde archivos de formato VRPLIB, almacenando los datos relevantes como nodos, demandas, capacidades y la matriz de distancias.
Solution y Ruta	Modelan una solución al problema y las rutas individuales, permitiendo almacenar y manipular la secuencia de clientes, demandas, costos y otras métricas asociadas.
ClarkeWrightSolver	Implementa la heurística constructiva de ahorros de Clarke & Wright, generando soluciones iniciales eficientes a partir de los datos de la instancia.
LocalSearch	Contiene la implementación de los operadores de búsqueda local, como reubicación (relocate) e intercambio (swap), para mejorar soluciones existentes.
GRASP	Implementa el metaheurístico GRASP (Greedy Randomized Adaptive Search Procedure), que construye soluciones de manera aleatorizada y aplica búsqueda local intensiva para refinar los resultados.
initial_methods	Función que combina la heurística de Clarke & Wright con búsqueda local, utilizada para generar soluciones iniciales de buena calidad de forma eficiente.

Cuadro 1: Clases principales desarrolladas en el proyecto y su función.

6.5. Casos de Test

Utilizamos ChatGPT para generar los tests unitarios para la clase **LocalSearch** e implementamos tests para la clase **Solution**. La idea principal de los tests para Solution es utilizarla para facilitar la comparación entre soluciones generadas y las soluciones dadas. Además, dispusimos de la IA para que, durante la experimentación, se registren los tiempos, costos y métodos de ejecución de cada instancia en un archivo .csv para luego facilitar el análisis de los resultados.
Link a la conversación con la IA

7. Experimentación y Resultados

Para obtener parámetros analizables, cuando se trataba de instancias con más de una solución óptima provista, decidimos tomar como referencia la solución con menor costo. Además, solo consideramos instancias con soluciones válidas a la hora de la experimentación.

7.1. Resultados de heurísticas constructivas / combinadas

Cuadro 2: Promedios sobre las 15 instancias

Método	**Coste/Óptimo**	**Gap (%)**	**Tiempo (ms)**	**Rutas**
Inicial	1.164	16.36	2.18	12.47
Clarke & Wright	1.383	38.27	0.63	12.47
Nearest Neighbor	1.359	35.93	0.45	9.73
Nearest Neighbor + Relocate	1.191	19.11	0.93	9.73
Nearest Neighbor + Híbrido	1.177	17.72	1.09	9.73

Coste/Óptimo. Observamos dos grupos bien definidos. Por un lado, *Inicial*, NN-R y NN-H mantienen el coste medio entre un 17 % y un 19 % por encima del óptimo, mientras que NN y Clarke & Wright presentan desviaciones cercanas al 36–38 %. El heurístico *Inicial* resulta ser competitivo—e incluso el mejor en 9 de 15 instancias—debido a la información a priori de carga y capacidad que incorpora la generación to-do-facto de rutas.

Gap porcentual. El patrón anterior se confirma al analizar la columna *Gap (%)*: NN-H es ligeramente mejor que NN-R (17.7 % frente a 19.1 %), ambos superando con claridad a NN y Clarke & Wright.

Número de rutas. Los métodos basados en NN generan de media tres rutas menos que *Inicial* y Clarke & Wright. Esta reducción no siempre se traduce en menor coste, lo que sugiere que, al optimizar la distancia, perdemos capacidad de consolidación de carga.

Tiempo de cómputo. Los cinco algoritmos resuelven cada instancia en milisegundos. Clarke & Wright y NN son los más rápidos (medio $\approx$ 0,5 ms); *Inicial* es el más costoso en CPU (2.2 ms) porque ejecuta comprobaciones adicionales de factibilidad.

8. Análisis detallado por algoritmo

Clarke & Wright

Es competitivo en tiempo, pero su heurística de ahorro *greedy* introduce rutas largas y un gap medio cercano al 40 %. Resulta subóptimo en todas las instancias, sobre todo en las de gran tamaño (`E200`), donde paga un sobrecoste del 65 %.

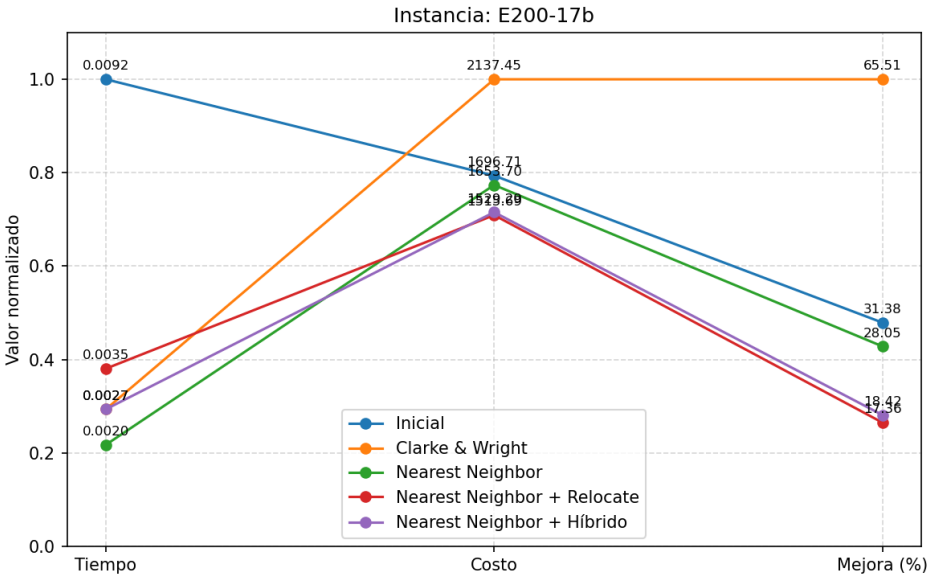


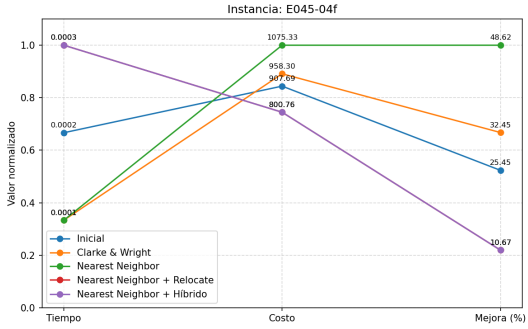
Figura 1: Tiempo, costo y mejora para la instancia E200-17b

Nearest Neighbor

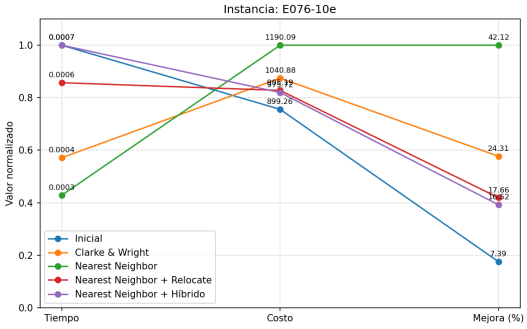
Produce menos rutas, pero incurre en "ciclos" de servicio ineficientes, elevando el coste total; el gap del 35–46 % lo sitúa sistemáticamente por detrás de NN-R y NN-H.

NN + Relocate

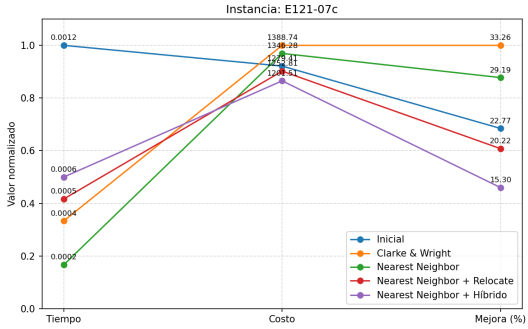
La simple operación de *1-relocate* sobre la solución de NN reduce el gap promedio a 19 %, recortando hasta 16 p.p. respecto a NN con un costo de tiempo marginal (< 1 ms). En 4 instancias (`E045`, `E076-10e`, `E121-07c`, `E200-17c`) obtiene el menor coste absoluto. (figura 2**??**)



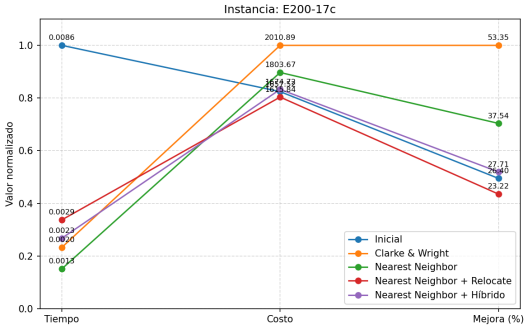
((a)) Instancia E045-04f



((b)) Instancia E076-10e



((c)) Instancia E121-07c



((d)) Instancia E200-17c

Figura 2: Comparación de rendimiento en diferentes instancias

NN + Híbrido

Integra *relocate* y *two-opt* ligeros. Cronométricamente se mantiene bajo 1.1 ms y mejora 1.4 p.p. adicionales frente a NN-R. Destaca en instancias densas (`E101-14u`, `E101-08e`) donde la inversión de aristas corrige bien los circuitos.

Inicial

Sorprende su buen desempeño (gap medio 16 %) pese a ser una construcción "naïve". En la mitad de las instancias es imbatible, lo cual confirma que la heurística interna de partición de carga es eficaz cuando la demanda está justamente balanceada.

9. Conclusiones

- **Eficacia global.** NN-H ofrece el mejor compromiso calidad/tiempo: recorta el gap casi a la mitad de NN con sólo el doble de tiempo (1 ms).
- **Sobrecoste de Clarke & Wright.** Aunque rápido, su política de ahorros resulta ineficiente en escenarios de alta densidad de clientes.
- **Calidad del arranque.** Un buen algoritmo inicial (como *Inicial*) puede igualar o superar a mejoras locales ligeras; la calidad de la semilla importa más que el refinamiento posterior cuando las restricciones de capacidad son ajustadas.

10. Posibles mejoras

1. **Metaheurísticas simples.** Implementar *Variable Neighborhood Search* (VNS) que combine *relocate*, *swap* y *2-opt* permitiría explorar vecindarios más amplios con coste temporal todavía aceptable (¡10 ms).

2. **Paralelización.** Dado que cada movimiento local es independiente, podemos paralelizar la evaluación de vecinos sobre GPU o hilos CPU, reduciendo la latencia a microsegundos en instancias medianas.

3. **Inicial adaptativo.** Ajustar dinámicamente el parámetro de carga máxima por ruta en la construcción inicial ("capacitated sweep") podría reducir los 12.5 rutas medias a valores comparables con NN-H, manteniendo el bajo gap observado.

4. **Aprendizaje de parámetros.** Un *tuning* automático mediante búsqueda Bayesiana sobre pesos de ahorro (Clarke & Wright) y probabilidades de selección de vecinos (NN) puede cerrar la brecha de 15–20 p.p. detectada.

11. Observaciones adicionales

- Los tiempos de cómputo son tres órdenes de magnitud menores que el tiempo operativo típico de un planificador logístico, por lo que priorizamos calidad sobre velocidad.

- La dispersión del gap se amplifica con el tamaño de la instancia: en `E200`, la diferencia máxima entre algoritmos supera los 35 p.p., mientras que en `E045` no llega a 38 p.p.

- Al compartir el mismo número medio de rutas, la diferencia entre NN-R y NN-H se atribuye casi exclusivamente a la fase *two-opt*, validando su utilidad aun con un tiempo
  ll 2 ms.

12. Conclusión

En este trabajo se implementaron y evaluaron diferentes heurísticas para el problema CVRP, demostrando que la combinación de métodos constructivos con búsqueda local produce resultados significativamente mejores que los algoritmos puramente constructivos. Los resultados muestran que NN-H ofrece el mejor compromiso entre calidad y tiempo de ejecución.