DOI: 10.24275/uama.6749.9387

Posgrado en Optimización

CBI DIVISIÓN DE CIENCIAS BÁSICAS E INGENIERÍA
UAM - Azcapotzalco

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Planeación de rutas de distribución óptimas con abastecimiento de combustible

Tesis presentada a la
Universidad Autónoma Metropolitana Azcapotzalco
como requerimiento parcial para obtener el grado de
Maestro en Optimización
por

Ing. Carlos Aurelio Mendieta Robles

Asesores:

Dr. Francisco Javier Zaragoza Martínez
Departamento de Sistemas, UAM Azcapotzalco

Dr. Rodrigo Alexander Castro Campos
Departamento de Sistemas, UAM Azcapotzalco

Ciudad de México, México
24 de agosto de 2020

Dedicatoria

A mis padres y a mi hermana, y a su amor incondicional.

Agradecimientos

A mis padres por su apoyo absoluto, a mi hermana por su ánimo constante y a mis asesores por su paciencia y dedicación.

Resumen

El problema de ruteo de vehículos es un problema central en la distribución de bienes, en donde se tiene un conjunto de solicitudes de clientes geográficamente dispersos y se dispone de una flota de vehículos de capacidad limitada que se encuentra estacionada inicialmente en un centro de distribución. El objetivo es asignar solicitudes a vehículos y calcular sus itinerarios, para que el costo total de distribución sea mínimo y la ejecución del plan sea factible.

En esta tesis presentamos un problema multiobjetivo nuevo en la literatura, en el que además de considerar el gasto y el reabastecimiento de combustible durante los recorridos, se busca minimizar tanto la distancia total recorrida por los vehículos (proporcional al combustible consumido por los mismos), como el tiempo total de recorrido (considerando la existencia de tiempos de espera variables en las distintas gasolineras) y también el costo monetario derivado de la compra del combustible (considerando la variación del precio de la gasolina en las distintas gasolineras). Formulamos este problema con un modelo de programación entera mixta para resolver un conjunto de instancias realistas mediante el solucionador lineal Gurobi. Posteriormente presentamos adaptaciones novedosas de heurísticas clásicas del problema de ruteo, con las que calculamos soluciones iniciales factibles para acelerar el proceso de resolución del modelo. El solucionador lineal también es asistido por un algoritmo de búsqueda local para intentar mejorar las soluciones obtenidas durante el proceso de optimización.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

La distribución de bienes es una actividad económicamente relevante para la industria y el comercio. Una gestión hábil de la distribución puede constituir una fuente importante de ahorro en los costos globales de operación de una empresa. La toma de decisiones en el proceso de planeación de la distribución puede ser asistida por herramientas de optimización que integran sistemas de información geográfica para diseñar rutas vehiculares eficientes. Además, su implementación permite automatizar y estandarizar el proceso de planeación e incluso facilita la generación de un conjunto de escenarios con diferentes relaciones costo-beneficio para elegir la opción más conveniente con respecto a la política de cada organización.

El problema de ruteo de vehículos (VRP por las siglas del inglés *Vehicle Routing Problem*) es un problema central en la gestión de la distribución en el que se tiene un conjunto de solicitudes de transporte de clientes geográficamente dispersos y se dispone de una flota de vehículos de capacidad limitada que se encuentra estacionada inicialmente en un centro de distribución. El objetivo es asignar solicitudes a vehículos y detallar sus itinerarios, para que el costo total de distribución sea mínimo y la ejecución del plan sea factible. De este problema ha surgido una familia numerosa de problemas de optimización y se ha convertido en uno de los temas más estudiados en el campo de la investigación de operaciones.

En general, el estudio de VRP ha desatendido el papel fundamental que juega el consumo de combustible y la necesidad de abastecimiento en la planificación de rutas, que representa entre 40 % y 60 % del gasto de operación de empresas transportistas [27]. En la vida real, el abastecimiento de combustible puede implicar una decisión sobre la gasolinera más conveniente a visitar con base en la cercanía, precios ofertados y tiempo de espera.

En esta tesis proponemos una nueva variante del VRP con abastecimiento de combustible, considerando tres funciones objetivo: minimizar la distancia total recorrida por los vehículos (proporcional al combustible consumido por los mismos), minimizar el tiempo total de recorrido (considerando la existencia de tiempos de espera variables en las distintas gasolineras) y minimizar el costo monetario incurrido por la compra de combustible durante viajes (considerando la variación del precio de la gasolina en las distintas gasolineras). Para ello, presentamos un modelo de programación entera mixta con el que resolvemos un conjunto de instancias haciendo uso del solucionador lineal Gurobi. Posteriormente presentamos dos heurísticas que calculan soluciones factibles a usarse como soluciones iniciales en la resolución del modelo por parte del solucionador, el cual también fue asistido por un algoritmo de búsqueda local para intentar mejorar las soluciones obtenidas durante la ejecución.

1.1. Preliminares de programación entera

El problema de programación lineal es un problema de optimización que pretende encontrar el valor de un conjunto de variables continuas $(x_1, x_2, \ldots, x_n)$ que maximizan o minimizan una función objetivo lineal, sujeta a un conjunto de restricciones lineales. El problema de programación entera es un problema de programación lineal en el que al menos una de sus variables sólo puede tomar valores enteros [49]. Un modelo que consiste únicamente de variables enteras es un modelo de programación entera puro. Frecuentemente los modelos contienen tanto variables enteras como variables continuas; a esto se le conoce como programación lineal entera o programación entera mixta (MIP por las siglas del inglés *Mixed Integer Programming*). Un programa entero mixto se define matemáticamente como:

$$\text{mín} \quad \sum_{j}^{n} c_j x_j + \sum_{k}^{p} d_k y_k \tag{1.1}$$

$$\text{sujeto a} \quad \sum_{j}^{n} a_{ij} x_j + \sum_{k}^{p} g_{ik} y_k \geq b_i \qquad (i = 1, 2, \ldots, m) \tag{1.2}$$

$$x_j \geq 0 \qquad (j = 1, 2, \ldots, n) \tag{1.3}$$

$$y_k \in \mathbb{Z} \qquad (k = 1, 2, \ldots, p) \tag{1.4}$$

La programación entera mixta puede modelar una gran cantidad de problemas prácticos. En particular, las variables enteras binarias (con valores 0 o 1) permiten modelar decisiones del tipo "sí o no". Con programación lineal entera se puede modelar la negación $(\neg x)$, la

conjunción $(x_1 \wedge x_2)$ y la disyunción $(x_1 \vee x_2)$ de variables binarias x, x_1, x_2. Para esto, definiremos nuevas variables binarias $x_\neg$, $x_\wedge$ y $x_\vee$ restringidas de forma tal que tomen los valores de $\neg x$, de $x_1 \wedge x_2$ y de $x_1 \vee x_2$ respectivamente.

$$x_\neg = 1 - x \tag{1.5}$$

$$x_\wedge \leq x_1 \qquad x_\wedge \leq x_2 \qquad x_\wedge \geq x_1 + x_2 - 1 \tag{1.6}$$

$$x_\vee \geq x_1 \qquad x_\vee \geq x_2 \qquad x_\vee \leq x_1 + x_2 \tag{1.7}$$

Resolver un programa lineal entero es un problema $\mathcal{NP}$-Duro, aún cuando las variables sean binarias y sólo se busque determinar factibilidad [19]. Se cree que no existen algoritmos polinomiales para ningún problema $\mathcal{NP}$-Duro [51].

1.2. Preliminares de optimización multiobjetivo

La solución de la mayoría de los problemas en el mundo real implica satisfacer distintos criterios, los cuales suelen estar en conflicto entre sí. La optimización multiobjetivo trata de resolver problemas matemáticos que involucran la optimización simultánea de dos o más funciones objetivo, posiblemente en conflicto. El problema general de optimización multiobjetivo se define formalmente de la siguiente manera:

$$\text{mín} \quad \mathbf{F}(x) \coloneqq (f_1(x), f_2(x), \ldots, f_m(x)) \tag{1.8}$$

$$\text{sujeto a} \quad g_i(x) \geq 0 \qquad \forall i \in \{1, 2, \ldots, o\} \tag{1.9}$$

$$h_j(x) = 0 \qquad \forall j \in \{1, 2, \ldots, p\} \tag{1.10}$$

$$x_k \in \mathcal{X}_k \qquad \forall k \in \{1, 2, \ldots, n\} \tag{1.11}$$

Donde el espacio de búsqueda está restringido por o restricciones de desigualdad (1.9) y p restricciones de igualdad (1.10), mientras que el vector solución $x^* \in \mathcal{X}$ minimiza las funciones objetivo que conforman a la función vectorial (1.8). A diferencia de lo que pasa en la optimización mono-objetivo, en la optimización multiobjetivo puede que no exista una única solución que optimice al mismo tiempo todas las funciones objetivo, sino que usualmente existen varias soluciones que representan los mejores compromisos entre tales objetivos. Al conjunto de estas mejores soluciones se le denomina conjunto de óptimos de Pareto y su representación gráfica se conoce como frente de Pareto.

Un vector de variables de decisión $x^* \in \mathcal{X}$ es un óptimo de Pareto si no existe otro vector $x \in \mathcal{X}$ tal que $f_i(x) \leq f_i(x^*)$ para toda $i = 1, \ldots, m$ y $f_j(x) < f_j(x^*)$ para al menos una j. Los vectores que conforman el conjunto de óptimos de Pareto se llaman no dominados. Para que una solución "domine" a otra, no debe ser peor en ninguno de los objetivos y debe ser estrictamente mejor en al menos uno de ellos.

Existe software comercial de optimización que provee al usuario de una interfaz de optimización multiobjetivo con la que es posible tratar los múltiples objetivos como una suma ponderada u optimizarlos en orden jerárquico. En el primer caso, el usuario asigna un peso a cada función objetivo para combinarlas en un solo término. En el segundo caso, se establecen prioridades sobre las funciones objetivo para que al optimizar una de ellas nunca se degrade el valor de soluciones calculadas para objetivos con mayor prioridad. En particular, la manipulación de los pesos en una suma ponderada de objetivos puede usarse para intentar aproximar el frente de Pareto [18].

1.3. Preliminares del problema de ruteo de vehículos

El problema de ruteo de vehículos con límite de capacidad (CVRP por las siglas del inglés *Capacitated Vehicle Routing Problem*) es la versión clásica de VRP y su exposición sirve como introducción para el estudio de versiones más complejas del problema. El problema consiste en diseñar rutas de distribución (o recolección) de costo mínimo para satisfacer la demanda de un conjunto N de n clientes geográficamente dispersos. Para ello se cuenta con una flota K de k vehículos estacionados en un almacén central (denotado por $0 \notin N$) donde los vehículos comienzan y finalizan su recorrido. Todos los vehículos cuentan con la misma capacidad máxima de carga Q y tienen costos de operación idénticos. Un vehículo que viaja del punto i al punto j incurre en un costo no negativo de transporte c_{ij}, el cual puede representar distancia, tiempo o cualquier atributo relacionado con algún recurso que se consume o se utiliza durante el desplazamiento de los vehículos. La demanda del cliente $i \in N$ está dada por un escalar $q_i \geq 0$.

La información anterior puede estructurarse mediante gráficas. Una gráfica no dirigida $G = (V, E)$ es una pareja donde V es un conjunto finito de vértices (o nodos) y E es un conjunto finito de aristas, que son pares no ordenados de vértices. Una gráfica dirigida $D = (V, A)$, también llamada digráfica, es una pareja donde V es un conjunto finito de vértices y A es

un conjunto finito de arcos, que son pares ordenados de vértices. Ambos casos son parejas de conjuntos disjuntos. Una gráfica es completa si todas las parejas de vértices se encuentran unidas por una arista o un arco, según sea el caso. En el CVRP, si el costo de viajar de cualquier vértice i a cualquier otro vértice j es el mismo que el costo de viajar de j a i, el problema tiene costos simétricos y se puede estructurar como una gráfica completa no dirigida. Por el contrario, si existe al menos un par de vértices con costos asimétricos, entonces el problema se asocia con una digráfica completa.

Un vehículo que atiende un subconjunto de clientes $S \subseteq N$, comienza en el almacén y visita una vez cada cliente en S antes de volver al almacén para finalizar su ruta. Por lo tanto una ruta es una secuencia $r = (i_0, i_1, i_2, \ldots, i_s, i_{s+1})$ donde $i_0 = i_{s+1} = 0$. La ruta r tiene costo $c(r) = \sum_{p=0}^{s} c_{i_p, i_{p+1}}$ y es factible si respeta la restricción de capacidad $\sum_{i \in S} q_i \leq Q$ y ningún cliente se visita más de una vez. Para un subconjunto arbitrario de clientes $S \subseteq N$, $r(S)$ es el número mínimo de vehículos necesarios para atender S y se puede calcular resolviendo el problema de encontrar el número mínimo de mochilas con capacidad Q para almacenar n artículos con pesos q_i donde $i \in N$. La solución de CVRP consiste de k rutas factibles donde los k subconjuntos de clientes forman una partición de N, cada ruta visita al menos un cliente y la suma del costo de todas las rutas es mínima. El problema de calcular el costo mínimo para atender un subconjunto de clientes usando un vehículo es equivalente al problema del agente viajero (TSP por las siglas del inglés *Traveling Salesman Problem*), el cual busca determinar un circuito hamiltoniano de costo mínimo. Como el problema del agente viajero es $\mathcal{NP}$-Duro [25], entonces el CVRP también es un problema $\mathcal{NP}$-Duro [25].

Formulación del problema como flujo vehicular

Sea x_{ij} una variable binaria que indica si se realiza o no un viaje directo del vértice i al vértice j y sea $S \subseteq V$ un subconjunto arbitrario de vértices. Dada una digráfica $D = (V, A)$, los *conjuntos de corte*, *arco-entrada* y *arco-salida* son conjuntos de aristas con exactamente un extremo en S y se definen como $\delta^-(S) = \{(i,j) \in A : i \notin S, j \in S\}$ y $\delta^+(S) = \{(i,j) \in A : i \in S, j \notin S\}$, respectivamente. Un modelo para CVRP con costos asimétricos fue propuesto por Laporte, Mercure y Norbert en [22] y es el siguiente:

$$\text{mín} \quad \sum_{(i,j) \in A} c_{ij} x_{ij} \tag{1.12}$$

$$\text{sujeto a} \quad \sum_{j \in \delta^+(i)} x_{ij} = 1 \qquad \forall i \in N \tag{1.13}$$

$$\sum_{i \in \delta^-(j)} x_{ij} = 1 \qquad \forall j \in N \tag{1.14}$$

$$\sum_{j \in \delta^+(0)} x_{0j} = k \tag{1.15}$$

$$\sum_{(i,j) \in \delta^+(S)} x_{ij} \geq r(S) \qquad \forall S \subseteq N, S \neq \emptyset \tag{1.16}$$

$$x_{ij} \in \{0, 1\} \qquad \forall (i,j) \in A \tag{1.17}$$

El objetivo (1.12) es la minimización del costo total de ruteo. Las restricciones (1.13) y (1.14) establecen que cada cliente sólo se visita una ocasión. La restricción (1.15) asegura que se construyan exactamente k rutas. Si se cuenta con más vehículos de los necesarios, se puede intercambiar esta igualdad por una desigualdad $\leq$. La restricción (1.16) es una restricción de capacidad que además prohíbe la formación de recorridos que no contengan al almacén. Debido a que en esta tesis no se consideran otras versiones de la familia de VRP, en el resto del documento nos referimos al CVRP como VRP.

Capítulo 2

Antecedentes

En este capítulo se mencionan brevemente los planteamientos propuestos en la literatura del VRP que sirvieron como punto de referencia en la construcción de la formulación matemática que se expone en el siguiente capítulo, así como en la adaptación de métodos heurísticos usados para calcular soluciones factibles del problema en estudio. También se citan investigaciones que estudian el papel que juega el combustible en problemas de ruteo en general y en el VRP en particular. Cabe resaltar que la consideración del consumo y abastecimiento de combustible en el estudio del VRP es un campo pobremente explorado.

2.1. El problema de ruteo de vehículos

El VRP ha sido materia de investigación activa en el diseño de algoritmos desde que fue propuesto en 1959 por George Dantzig y John Ramser [10]. Su vigencia se debe, por un lado, a que el diseño de rutas eficientes de distribución y servicio a domicilio es económicamente relevante para la industria, el comercio y la administración pública. Más aún, muchos de los métodos exactos y heurísticos desarrollados para la versión más simple del problema pueden adaptarse para incluir diversas reglas de operación y restricciones administrativas, físicas o geográficas presentes en la práctica. Por otro lado, se ha mantenido el interés teórico en el VRP porque éste generaliza al TSP, el cual figura como uno de los problemas más representativos de la optimización combinatoria, si bien en la práctica el VRP es mucho más difícil de resolver [21]. Muchos de los métodos exactos y heurísticos desarrollados para el VRP provienen del extenso trabajo realizado previamente para resolver el TSP.

2.1.1. Modelos matemáticos

Debido al gran interés que se tiene por el VRP, se han desarrollado una gran cantidad de modelos matemáticos tanto para el problema clásico como para multitud de variantes con importancia en la práctica. Destacan algunos modelos de programación entera mixta planteados desde diferentes perspectivas, a partir de los cuales han surgido diversos métodos de solución.

La formulación de flujo vehicular probablemente sea el modelo al que más se recurre en la literatura de VRP. Está basada en la formulación clásica de TSP propuesta por Dantzig, Fulkerson y Johnson [9] y se distinguen dos planteamientos distintos de este tipo de formulación. El primer planteamiento es llamado formulación de flujo vehicular de dos índices, en alusión a los subíndices de sus variables binarias de decisión x_{ij} que indican si la solución óptima incluye o no un viaje directo entre el vértice i y el vértice j. El modelo para VRP con costos simétricos fue propuesto por Laporte, Norbert y Desrochers [23], mientras que el modelo con costos asimétricos fue propuesto por Laporte, Mercure y Norbert [22] y fue expuesto en el capítulo anterior. Ambos planteamientos tienen un número polinomial de variables con respecto al número de clientes n y al número de vehículos k y un número exponencial de restricciones con respecto a n. Una formulación popular de VRP que usa un número cuadrático de variables y restricciones fue propuesto por Miller, Tucker y Zemlin [32] para la versión asimétrica del problema y se reproduce a continuación:

$$\text{mín} \quad \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{2.1}$$

$$\text{sujeto a} \quad \sum_{j\in\delta^+(i)} x_{ij} = 1 \qquad \forall i \in N \tag{2.2}$$

$$\sum_{i\in\delta^-(j)} x_{ij} = 1 \qquad \forall j \in N \tag{2.3}$$

$$\sum_{j\in\delta^+(0)} x_{0j} = k \tag{2.4}$$

$$u_i - u_j + Qx_{ij} \leq Q - q_j \qquad \forall (i,j) \in A \mid i,j \in N \tag{2.5}$$

$$q_i \leq u_i \leq Q \qquad \forall i \in N \tag{2.6}$$

$$x_{ij} \in \{0,1\} \qquad \forall (i,j) \in A \tag{2.7}$$

El objetivo (2.1) es la minimización del costo total de ruteo. Las restricciones (2.2) y (2.3) establecen que cada cliente sólo tiene un vértice sucesor y un vértice antecesor, respectivamente. La restricción (2.4) asegura que se construyan exactamente k rutas. La variable u_i es

la demanda acumulada que ya ha sido distribuida por el vehículo una vez que éste atiende al cliente i. La restricción (2.5) asegura que cada vez que la solución utiliza un arco x_{ij} entre dos clientes, la diferencia de sus demandas acumuladas coincida con q_j. Esta restricción prohibe la formación de subrutas $(i, j, \ldots, i)$ que no contengan al almacén, ya que de otro modo se llegaría a la contradicción $u_i < u_j < \ldots < u_i$. Un inconveniente del modelo es que su relajación lineal produce una cota significativamente más débil que la producida mediante la formulación original.

La formulación de flujo vehicular de dos índices tiene la desventaja de modelar la flota de vehículos sólo de manera implícita. Su solución muestra los arcos o aristas que forman las rutas óptimas pero el recorrido individual de cada vehículo se debe construir interpretando esta información. Por esta razón, la formulación de dos índices no es adecuada para modelar otra versiones de VRP donde los vehículos tienen características específicas, o donde varios vehículos pueden visitar un mismo punto intermedio en repetidas ocasiones. Una estrategia para modelar múltiples visitas a un subconjunto de vértices es aumentar la gráfica con vértices artificiales ubicados en la misma posición de los vértices originales, agregando un vértice artificial por cada potencial visita adicional [12]. Esta misma idea fue propuesta inicialmente para convertir el VRP con un número fijo de k vehículos a un TSP mediante la inclusión de k almacenes artificiales ubicados en la misma posición del almacén original. De esta forma, el conjunto de k rutas operadas por k vehículos se vuelve una sola ruta que visita $n+k+1$ puntos, sujeta a restricciones de capacidad para cada subconjunto de clientes entre almacenes. Sin embargo, esta estrategia de solución fue rápidamente abandonada por la complejidad que resulta de un planteamiento del TSP con restricciones de capacidad impuestas por el VRP.

El segundo planteamiento, conocido como formulación de flujo vehicular de tres índices, fue propuesto por Golden, Magnanti y Nguyen [16]. Este modelo hace uso de dos tipos de variables binarias de decisión: x_{ijr} que indica si el vehículo r recorre el arco $(i, j) \in A$ y y_{ir} que indica si el vehículo r visita el vértice i. Desafortunadamente, a pesar de modelar con suficiente detalle la ruta que sigue cada vehículo, este planteamiento no es práctico porque genera soluciones redundantes que resultan de permutar los índices de los vehículos. Cualquier solución tiene $k!$ soluciones equivalentes, por lo que la dimensión del espacio de búsqueda puede ser demasiado grande, especialmente si la flota de vehículos es numerosa.

Otro tipo de planteamiento que también ha sido ampliamente estudiado, el cual aquí se menciona únicamente por su relevancia teórica, es la formulación extensa basada en partición

de conjuntos o cobertura de conjuntos propuesta por Balinski y Quandt [1]. Esta formulación resuelve el problema partiendo de la construcción de todas las particiones factibles de los clientes según las restricciones de capacidad. A pesar de que la aplicación directa de este modelo no es práctica por la gran cantidad de particiones posibles y porque el cálculo de rutas óptimas implica resolver TSP un número prohibitivo de veces, algunos de los métodos exactos para VRP más exitosos hacen uso parcial de esta formulación.

Actualmente, los métodos de ramificación y corte propuestos por Lysgaard *et al.* [31] y Pecin *et al.* [35] son los más efectivos para la versión clásica del problema y algunas variantes sencillas del mismo, ya que logran resolver instancias hasta con 300 clientes, especialmente si la instancia tiene características favorables para estos métodos [36]. Entre los distintos documentos que recopilan la variedad de métodos exactos propuestos para el VRP se encuentran los trabajos de Semet *et al.* [42] y Poggi y Uchoa [36].

2.1.2. Métodos heurísticos

A pesar del enorme progreso conseguido en la resolución de instancias de VRP, los algoritmos exactos más destacados apenas logran optimalidad en instancias relativamente pequeñas con cientos de clientes y el tiempo de cómputo puede ser impredecible. La necesidad de resolver instancias prácticas de distribución que involucran miles de solicitudes de transporte en un tiempo predecible, vuelve inviable cualquier método exacto conocido. Los métodos heurísticos sacrifican optimalidad a cambio de obtener soluciones de calidad razonable y reducciones significativas en el tiempo de cómputo. La consideración de métodos heurísticos en esta tesis se hace con la finalidad de contar con métodos rápidos para calcular soluciones factibles que sirvan como punto de partida en el cálculo de la solución exacta por parte de un solucionador entero. De manera general, los métodos heurísticos para VRP se dividen en heurísticas clásicas y metaheurísticas. En esta tesis nos limitamos a la adaptación de ejemplos destacados de heurísticas clásicas, pues aunque la calidad de las soluciones calculadas por metaheurísticas es mayor, no fueron consideradas por su dificultad de calibración y por el incremento en el tiempo de cómputo. Gendreau *et al.* presentan una recopilación de implementaciones metaheurísticas destacadas para VRP de recocido simulado, búsqueda tabú, algoritmos genéticos y colonia de hormigas, entre otras [14].

Las heurísticas clásicas se desarrollaron principalmente entre las décadas de 1960 y 1990 y son capaces de calcular buenas soluciones en un tiempo de cómputo moderado por medio

de exploraciones relativamente limitadas del espacio de búsqueda. A diferencia de las metaheurísticas, las heurísticas clásicas no admiten el deterioro de la función objetivo de una iteración a otra, lo que las hace propensas a quedar atrapadas en óptimos locales. Otra diferencia con respecto a las metaheurísticas, es la facilidad con la que las heurísticas clásicas pueden adaptarse para satisfacer restricciones de factibilidad impuestas en otras versiones del problema, por lo que son ampliamente usadas en paquetes comerciales. Un inconveniente es que la mayoría de las heurísticas clásicas manejan el tamaño de la flota como una variable de respuesta [24]. Se ha reportado muy poca experiencia computacional sobre heurísticas clásicas aplicadas al caso asimétrico [24]. Un trabajo notable es el de Rodríguez y Ruíz, quienes realizaron un conjunto de experimentos exhaustivos para determinar el efecto de diversos factores presentes en instancias realistas de VRP en la calidad de las soluciones obtenidas y el tiempo de cómputo empleado [39]. Según la estrategia empleada, las heurísticas clásicas se clasifican en heurísticas constructivas, heurísticas de dos fases y heurísticas de mejora [24].

Heurísticas constructivas

Estas heurísticas construyen gradualmente una solución factible usando dos estrategias básicas: uniendo rutas existentes tal que esta unión genere un ahorro y asignando iterativamente vértices a rutas. La heurística de este tipo más famosa para VRP es el algoritmo glotón de ahorros propuesto por Clarke y Wright [6]. Este método constructivo es fácil de implementar y produce buenas soluciones en muy poco tiempo de cómputo. El concepto de "ahorro" se refiere a la reducción del costo de distribución que resulta de unir dos rutas independientes en una sola ruta. Dados dos clientes i y j que representan el primer y el último cliente de sus respectivas rutas, el ahorro asociado se define como:

$$s_{ij} = c_{i0} + c_{0j} - c_{ij} \tag{2.8}$$

En la expresión (2.8), la unión de las rutas ahorra el viaje de regreso desde el cliente i al almacén, así como el viaje de ida desde el almacén al cliente j, pero agrega el costo del viaje entre los clientes i y j. La figura 2.1 ejemplifica esta situación. Como el ahorro producido por la unión de dos rutas con clientes extremos i y j se conoce de antemano, es posible precalcular el ahorro entre todo par de clientes de la gráfica y ordenarlos de forma no ascendente para su posterior procesamiento. El algoritmo inicia con una ruta independiente para cada cliente de ida y vuelta al almacén. En cada iteración posterior, el algoritmo evalúa si es factible unir las rutas a las que pertenecen los dos clientes implicados en el mayor ahorro disponible. En la

versión asimétrica del problema, la unión de dos clientes es factible si no pertenecen a la misma ruta, si la suma de las demandas de las rutas a las que pertenecen no excede la capacidad del vehículo y si ambos clientes ocupan el primer y el último lugar en la secuencia de clientes de sus respectivas rutas. El algoritmo termina cuando es imposible seguir uniendo rutas o cuando ya se tiene la cantidad de rutas deseadas. Por la naturaleza glotona del algoritmo, el número de vehículos en la solución calculada por este método es una variable de respuesta.

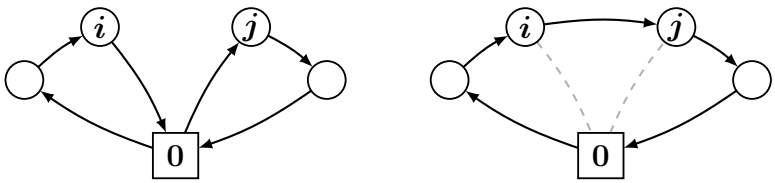


Figura 2.1: Unión de dos rutas independientes.

Las heurísticas constructivas basadas en inserción asignan gradualmente vértices a rutas con base en el mejor costo de inserción disponible, el cual puede ser afectado por diferentes parámetros que favorecen ciertas características de las rutas en construcción. Dos métodos representativos de inserción secuencial son los propuestos por Mole y Jameson [34] y Christofides *et al.* [5]. La mayoría de las heurísticas constructivas empleadas en la práctica incorporan algunas técnicas de mejora adicionales durante las distintas etapas del proceso.

Heurísticas de dos fases

Algunas heurísticas descomponen el proceso de solución del problema en dos etapas, simulando una forma muy intuitiva de resolver el VRP de forma manual: primero encuentran subconjuntos de clientes que pueden ser atendidos por un mismo vehículo y después construyen rutas a partir de éstos. Existen métodos menos investigados que siguen el orden inverso: primero construyen una ruta que visita todos los vértices y después la segmentan en subconjuntos factibles [24]. En ambos casos las fases pueden retroalimentarse entre sí.

El algoritmo de barrido pertenece a la familia de heurísticas conocidas como *asignar primero – rutear después*. Este algoritmo es una heurística geométrica para instancias bidimensionales, comúnmente atribuida a Gillett y Miller [15]. Comienza calculando el ángulo polar que forma cada cliente con respecto al almacén para determinar un ordenamiento único

de los clientes. El empate de dos o más clientes con exactamente el mismo ángulo se rompe considerando menor al cliente más cercano al almacén. En cada iteración, se intenta asignar el cliente no asignado con menor ángulo al vehículo en turno. Si la demanda del cliente por asignar no puede ser atendida por el vehículo en turno, se asume que el vehículo está lleno y el cliente en cuestión se asigna a un nuevo vehículo vacío. Por esto, la aplicación directa de este método no restringe el número de vehículos de la solución final. En la figura 2.2 se ejemplifica la asignación de clientes a rutas por este algoritmo; del lado izquierdo, la demanda de los tres clientes asignados impide que el cuarto cliente en el ordenamiento sea atendido en el mismo vehículo, por lo que se asigna a una ruta distinta, la cual se muestra del lado derecho. Una vez que todos los clientes han sido asignados, se puede calcular la ruta óptima de cada vehículo, lo cual sería la única etapa del algoritmo con un costo computacional significativo.

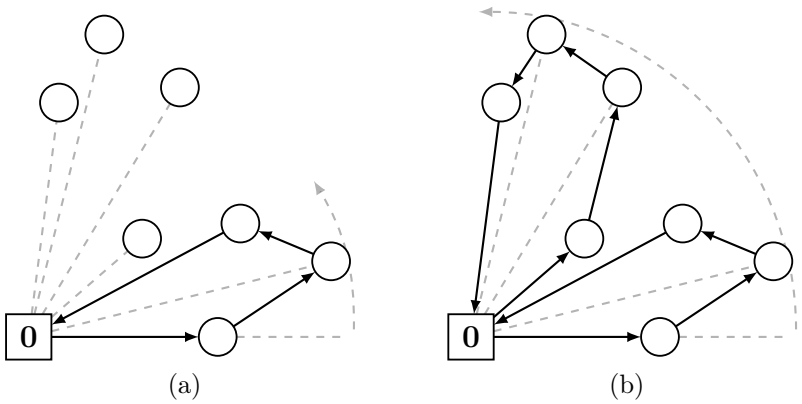


Figura 2.2: Asignación de clientes a rutas en la heurística de barrido.

El algoritmo puede ejecutarse n veces girando los ejes del plano cartesiano (con origen en el almacén) en el sentido contrario a las manecillas del reloj para así considerar una vez a cada cliente como el cliente con el menor ángulo. Si se repite el proceso completo descrito anteriormente intentando asignar al cliente con el mayor ángulo y girando los ejes del plano en el sentido de las manecillas del reloj, se obtienen en total $2n$ configuraciones de rutas, posiblemente distintas entre sí, de las cuales el algoritmo entrega la configuración de menor costo como solución final. Otros ejemplos de heurísticas pertenecientes a la familia *asignar primero – rutear después* son el algoritmo de pétalos [13], el algortimo de Fisher y Jiakumar [33] y el algoritmo de Bramel y Simchi-Levi [4].

Heurísticas de mejora

Se pueden aplicar dos estrategias heurísticas para intentar mejorar una solución factible de VRP: permutar el orden de visita de los clientes dentro de una misma ruta (por ejemplo, con alguna heurística para TSP) o bien, intercambiar uno o más clientes entre dos o más rutas. Es común combinar ambas estrategias en una misma heurística.

La mayoría de las estrategias usadas para intentar mejorar individualmente una ruta provienen del método de búsqueda local λ-opt propuesto para TSP por Lin [28]. Dependiendo de la versión que se implemente, este método puede seleccionar la primera permutación que mejore la solución o explorar todo el vecindario en busca de la mejor solución. En el segundo caso, la idea es encontrar la permutación con menor costo a la que se puede llegar remplazando sistemáticamente λ aristas, comenzando con una permutación o ruta inicial. Una ruta es λ-óptima si es imposible obtener otra ruta de menor costo remplazando λ aristas cualesquiera por otro conjunto de λ aristas. Una ruta que visita n clientes es una ruta óptima si y sólo si es n-óptima. Se puede comprobar si una solución es λ-óptima en tiempo de cómputo $\mathcal{O}(n^{\lambda})$.

Los movimientos simultáneos de mejora entre rutas normalmente consisten en remover uno o varios clientes de cierto número de rutas y reubicarlos. La mayoría de los métodos propuestos en esta dirección son casos especiales del esquema *b-cíclico, k-transferencia*, propuesto por Thompson y Psaraftis [48].

2.2. Problemas con abastecimiento de combustible

El primer software para asistir en la toma de decisiones con respecto al abastecimiento de combustible fue desarrollado a mediados de la década de 1990 por una compañía consultora especializada en transporte y establecida en Estados Unidos[44]. Estas herramientas utilizan información de bases de datos que se actualizan diariamente con el precio de combustible ofertado por gasolineras de una región determinada para decidir qué gasolineras visitar y cuánto combustible comprar para minimizar el costo de abastecimiento al viajar entre dos puntos. La idea básica es tomar ventaja de la variación de precio de combustible entre gasolineras, comprando más combustible en establecimientos donde es barato y comprando menos donde es caro. Desde entonces, el uso de este tipo de productos se ha extendido entre los transportistas de ese país, pues resultan en ahorros de $1200 dólares anuales por vehículo [45].

La literatura sobre problemas de ruteo con abastecimiento de combustible es escasa [20, 30, 44]. Uno de los primeros problemas planteados fue el problema de abastecimiento de combustible en una ruta preestablecida (FRVRP por las siglas de *Fixed-Route Vehicle Refueling Problem*, el cual no es una variante de VRP a pesar de la similitud de su nombre). Este problema describe una situación donde un vehículo con un nivel inicial de combustible debe recorrer una secuencia preestablecida de gasolineras con precios diferentes entre sí. El objetivo es encontrar una política óptima que especifique cuánto combustible se debe comprar en cada gasolinera, sin exceder la capacidad del tanque y minimizando el costo total al que se compra el combustible [30]. Se conocen algoritmos polinomiales para este problema [30, 20].

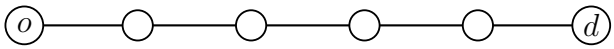


Figura 2.3: Diagrama de FRVRP.

Cuando la ruta es fija como en el FRVRP, la cantidad necesaria de combustible también es fija, por lo que el costo total de abastecimiento sólo depende del costo unitario al que se compra el combustible. Sin embargo, en la práctica la tasa de consumo de combustible depende de muchas otras variables. Suzuki *et al.* estudian el FRVRP desde un enfoque ambientalista que toma en cuenta el beneficio de mantener siempre espacio disponible en el tanque, lo que mejora la economía de combustible y disminuye las emisiones contaminantes [47]. Ellos proponen un programa no lineal que incorpora el efecto negativo del peso del combustible sobre la tasa de consumo del mismo y además varían el nivel mínimo de reserva de combustible en el tanque en función de qué tan densamente poblado de gasolineras está el trayecto próximo, lo que permite adquirir mayores cantidades de combustible en zonas donde se esperaría que los precios fueran menores por la competencia de mercado. Los autores desarrollan un método heurístico en el que relajan la no linearidad y la discontinuidad del programa propuesto y después utilizan un método estándar de programación lineal. Sus soluciones mejoran los costos obtenidos con métodos tradicionales para FRVRP y son amigables con el medio ambiente.

Tanto el software comercial como el planteamiento original de FRVRP ignoran las distancias que se deben recorrer cuando se visitan gasolineras que no se ubican a lo largo del trayecto y desatienden el inconveniente de tener que hacer demasiadas paradas para aprovechar los costos de combustible. Un modelo de programación entera mixta propuesto por Suzuki, desde una perspectiva más amplia de minimización de costos, investiga el efecto de considerar estos viajes y su repercusión en los costos de mantenimiento y depreciación de los vehículos, los cuales son funciones indirectas de la distancia recorrida [44]. La frecuencia

de abastecimiento se controla estableciendo una cota mínima en la cantidad de combustible que se recarga. Los resultados de sus experimentos muestran que las soluciones obtenidas resolviendo este modelo genérico por métodos convencionales de programación lineal difieren considerablemente de las soluciones calculadas por medio de software comercial de abastecimiento y sus costos son significativamente mejores.

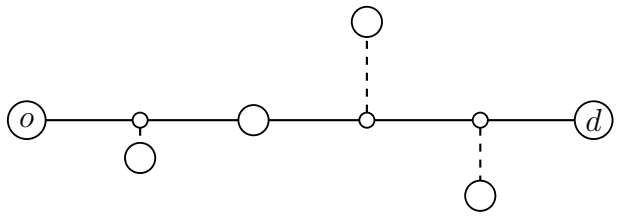


Figura 2.4: Diagrama de FRVRP con viajes fuera de ruta.

Khuller *et al.* desarrollaron algoritmos eficientes para calcular la solución de problemas de ruteo con abastecimiento de combustible que generalizan el problema de la ruta más corta [20]. Dada una gráfica completa de n vértices con oferta de combustible a diferentes precios y dada la distancia entre todo par de vértices, los autores muestran algoritmos exactos de complejidad polinomial basados en programación dinámica para encontrar la ruta y la política de abastecimiento óptimas entre un par de vértices en tiempo $\mathcal{O}(n^3)$ y entre todo par de vértices en tiempo $\mathcal{O}(n^4)$. A este problema se le conoce como el problema de abastecimiento en una ruta variable (VRVRP por las siglas de *Variable-Route Vehicle Refueling Problem*) [30, 46]. Lin propuso un algoritmo para este mismo problema basado en un análisis de la estructura combinatoria de las políticas de abastecimiento, con tiempos de ejecución $\mathcal{O}(n^2)$ y $\mathcal{O}(n^3)$, respectivamente [29]. El desempeño de los algoritmos mejora en ambos casos cuando se limita el número de paradas que puede hacer el vehículo.

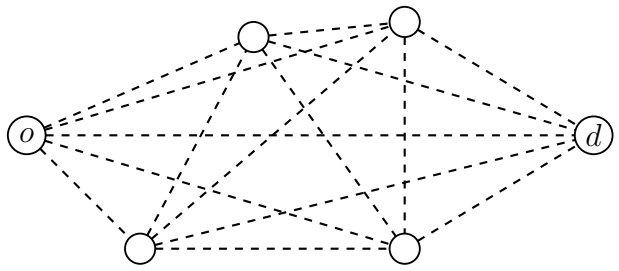


Figura 2.5: Diagrama de VRVRP.

El software comercial resuelve el VRVRP por medio de un procedimiento heurístico en el

que primero calcula la ruta más corta entre dos puntos, ignorando la necesidad de abastecimiento de combustible, y después corrige la ruta insertando paradas en gasolineras, lo cual no siempre produce soluciones óptimas [45]. En particular, cuando las inserciones buscan minimizar el gasto incurrido por la compra de combustible y no la distancia total recorrida, las soluciones generadas tienden a incrementar la longitud de las rutas para incluir gasolineras con precios bajos, lo que en la práctica puede representar un incremento inviable en el tiempo de viaje o en el consumo total de combustible. En vista del aparente conflicto entre objetivos, Suzuki y Dai desarrollaron un modelo exacto para instancias del VRVRP que sí toma en cuenta la distancia recorrida en viajes fuera de ruta y que adicionalmente usa un esquema multiobjetivo que produce un conjunto de soluciones pareto-óptimas que minimizan tanto el gasto en combustible como la distancia total de recorrido [44, 46]. Los autores utilizan instancias reales para resolver un modelo de programación entera mixta con el optimizador CPLEX y sus resultados superan el desempeño de productos comerciales en ambos objetivos, encontrando rutas significativamente más cortas que aquéllas halladas en la literatura y sin incurrir en incrementos importantes en el costo de combustible. Sus resultados demuestran que es recomendable la adopción de métodos exactos por parte del software comercial en términos de la calidad de la solución y el tiempo de cómputo [45].

2.3. El abastecimiento de combustible en VRP

La consideración del consumo y abastecimiento de combustible en el esquema del VRP es aún más escasa [3, 38]. Khuller *et al.* desarrollaron un algoritmo de aproximación en tiempo polinomial con un factor de aproximación $\frac{3}{2}$ $(\frac{1+\alpha}{1-\alpha})$, con α menor a 1, para una generalización del TSP donde un conjunto de los vértices que se deben visitar tienen oferta de combustible a precio uniforme [20]. Los algoritmos de aproximación para problemas de optimización son algoritmos eficientes que garantizan encontrar una solución cuyo valor nunca dista más de un factor α del valor de la solución óptima de cualquier instancia del problema [50]. Khuller *et al.* también resuelven otra versión donde varían los precios entre vértices utilizando el mismo algoritmo de aproximación pero pagando un factor β en la garantía de aproximación, donde β es la razón entre el precio más alto de combustible y el precio más bajo [20]. Finalmente, también resuelven el caso donde existe un único vértice con oferta de combustible en el que se encuentra el vehículo inicialmente y al cual debe regresar para recargar combustible y poder continuar las visitas a vértices. A este problema se le conoce como VRP con restricción de distancia (DVRP por las siglas de *Distance constrained Vehicle Routing Problem*).

Bousonville *et al.* presentan una extensión del problema de ruteo de vehículos con ventanas de tiempo (VRPTW por las siglas de *Vehicle Routing Problem with Time Windows*). Los autores integran un esquema de decisión de abastecimiento a una heurística constructiva específica de VRPTW, con la finalidad de investigar el efecto que tiene la variación de los precios de combustibles sobre la longitud de las rutas resultantes cuando se minimiza el costo de compra del combustible. En correspondencia con el conflicto de objetivos observado en VRVRP, sus resultados muestran que a medida que incrementa la variación de precios entre gasolineras, aumenta la distancia total de las rutas vehiculares [3].

Levy *et al.* proponen el problema de ruteo de vehículos heterogéneos no tripulados con múltiples depósitos [26]. En este problema existen múltiples depósitos en los que se encuentra estacionado un vehículo por depósito. La capacidad del tanque de combustible es diferente para cada vehículo, con la posibilidad de recargar combustible en cualquier depósito sin costo para completar su ruta. El objetivo del problema es encontrar el conjunto de rutas que visita un conjunto de clientes, minimizando el consumo del combustible. Levy *et al.* comparan el desempeño de dos metaheurísticas basadas en vecindarios variables que buscan encontrar buenas soluciones factibles para instancias grandes del problema. En ambos casos, los vecindarios usados provienen de las heurísticas clásicas de mejora 2-opt, 3-opt, almacén-opt y reubicación de vértices entre rutas. Sus resultados muestran una compensación muy clara entre el tiempo de cómputo del algoritmo y la calidad de las soluciones obtenidas, además sugieren que la inclusión de componentes probabilísticos para salir de óptimos locales no siempre produce mejores soluciones. Sundar *et al.* estudian este mismo problema con la consideración de una flota homogénea de vehículos convencionales y comparan empíricamente cuatro formulaciones de programación entera mixta [43].

Un planteamiento con un enfoque diferente es el presentado por Poonthalir *et al.* donde se calculan rutas de distribución que inician y terminan en un almacén central, minimizando el costo de abastecimiento de combustible y la cantidad de visitas a gasolineras [38]. El problema se modela como un problema de optimización multiobjetivo y se resuelve mediante un algoritmo de optimización por enjambre de partículas.

Existen muchas aplicaciones prácticas de problemas de ruteo en donde los vehículos deben seleccionar un punto a visitar de entre un conjunto de puntos disponibles, donde esto es un requisito para poder continuar con la ruta. El problema de ruteo de vehículos con

paradas intermedias considera los casos en donde los vehículos necesitan visitar almacenes intermedios o sitios de abastecimiento para deshacerse de la carga o para abastecerse de producto o combustible, según sea el caso [41]. A diferencia de los vértices que modelan clientes, no es necesario visitar cada uno de estos puntos. Schneider *et al.* proponen un modelo de programación entera mixta para este problema y también presentan un método de solución heurístico llamado búsqueda adaptativa en vecindarios variables, el cual emplean sobre un caso específico con vehículos eléctricos. La heurística propuesta combina ideas de búsqueda en vecindarios variables y búsqueda adaptativa en vecindarios grandes. Los distintos vecindarios reubican secuencias de vértices e intercambian vértices entre rutas de forma cíclica, donde cada vecindario en específico se selecciona según su desempeño previo. La efectividad de este método para calcular buenas soluciones en tiempo de cómputo aceptable se probó sobre instancias especiales del problema de ruteo de vehículos con paradas intermedias, disponibles en [40]. El algoritmo obtiene las mejores soluciones conocidas en menos de un minuto para un conjunto de instancias pequeñas, mientras que para un conjunto de instancias grandes obtiene buenas soluciones en pocos minutos.

Una línea de investigación que ha adquirido relevancia en los últimos años incorpora diferentes perjuicios ambientales a VRP bajo el nombre de problema de ruteo sustentable de vehículos (GVRP por las siglas de *Green Vehicle Routing Problem*) [2, 12, 27]. Estas variantes buscan reducir el impacto ambiental de los vehículos durante su operación al considerar aspectos como la emisión de gases contaminantes [11]. En este contexto, incluso importa conocer la velocidad de los vehículos en las diferentes etapas de sus recorridos [37]. Los inconvenientes que enfrentan la mayoría de los vehículos que usan energías limpias son un recorrido más corto en relación a los vehículos convencionales de combustión interna, capacidad limitada de almacenamiento de energía, así como escasa infraestructura de abastecimiento instalada. Esto los ha convertido en candidatos a ser estudiados en un marco teórico similar al del resto de los problemas de VRP con abastecimiento [12, 40].

Capítulo 3

VRP con abastecimiento de combustible

En este capítulo presentamos un problema nuevo en la literatura de VRP y un planteamiento de optimización multiobjetivo para el mismo. En las secciones siguientes describimos formalmente el problema propuesto y planteamos un programa entero mixto con una cantidad cúbica tanto de variables como de restricciones.

3.1. Descripción del problema

El VRP con abastecimiento de combustible consiste en diseñar rutas vehiculares óptimas para atender la demanda de cierto producto, el cual es solicitado por un conjunto de clientes geográficamente dispersos. Para ello se dispone de una flota homogénea de vehículos estacionados en un único centro de distribución, donde a su vez se encuentra almacenado el producto solicitado. Los vehículos consumen combustible mientras se trasladan de un lugar a otro y pueden reabastecerse de combustible en gasolineras también geográficamente dispersas, a un precio y tiempo de atención específico para cada una de ellas. El problema propuesto considera la optimización de las siguientes funciones objetivo: *i)* minimizar la distancia total recorrida (proporcional al combustible consumido); *ii)* minimizar el tiempo total de recorrido (considerando la existencia de tiempos de espera variables entre gasolineras); y *iii)* minimizar el costo monetario derivado de la compra de combustible (considerando la variación del precio de la gasolina en las distintas gasolineras).

Matemáticamente, definimos este problema sobre una gráfica dirigida $D = (V, A)$, donde $V = \{0, 1, \ldots, n + m\}$ es el conjunto de vértices y $A = \{(i, j) : i, j \in V, i \neq j\}$ es el conjunto de arcos. El vértice 0 representa el centro de distribución, del cual parte un conjunto K

de k vehículos idénticos que deben regresar al finalizar sus respectivas rutas. Cada vehículo atiende únicamente una ruta. El resto de los vértices corresponden con las ubicaciones de n clientes $N = \{1, 2, \ldots, n\}$ y m gasolineras $M = \{n+1, n+2, \ldots, n+m\}$ respectivamente. Los vértices que representan clientes sólo pueden ser visitados por un vehículo y en una sola ocasión, mientras que aquéllos que representan gasolineras pueden ser visitados más de una vez por un mismo vehículo o no ser visitados en absoluto. Cada cliente $i \in N$ tiene una demanda de producto $q_i > 0$. Cada gasolinera $g \in M$ cuenta con una oferta ilimitada de combustible, el cual despacha a un precio unitario p_g con un tiempo de atención constante t_g. Supondremos que es posible atender de manera simultánea cualquier cantidad de vehículos en cualquier gasolinera sin afectación al tiempo de atención. Todos los vehículos tienen la misma capacidad máxima de producto Q, la misma capacidad máxima de combustible F, el mismo nivel inicial de combustible f_0 y la misma tasa de rendimiento de combustible b.

Cada arco $\{i, j\} \in A$ tiene asociados una distancia $d_{ij} > 0$ y un tiempo de recorrido $t_{ij} > 0$ correspondientes al viaje directo de i a j. Supondremos que el consumo de combustible f_{ij} en cada arco es linealmente proporcional a d_{ij}, con $f_{ij} = \frac{d_{ij}}{b}$. Cada vehículo está asociado a una ruta $r = (0, v_1, \ldots, v_t, 0)$ que es una secuencia de vértices que inicia y termina en el centro de distribución y que atiende un subconjunto no vacío de clientes $S \subset N$, recargando combustible durante el camino en caso de ser necesario. Si r visita la gasolinera g entonces incurre en un tiempo de espera constante t_g y en un costo de compra de combustible p_g por unidad de combustible comprada. El costo total de r es una tripleta formada por la suma de las distancias de los arcos que recorre, la suma de los tiempos de recorrido y de espera, y la suma de los costos monetarios derivados de la compra de combustible. Una ruta r es factible si la suma de las demandas de los clientes que visita no sobrepasa Q, si el vehículo nunca se queda sin gasolina antes de poder reabastacerse, si la cantidad de combustible abastecido en una gasolinera más la cantidad de combustible que aún le quedaba al llegar no rebasa la capacidad máxima F, si el vehículo termina su ruta con al menos f_0 unidades de combustible y si ningún cliente se visita más de una vez. La figura 3.1 ejemplifica una instancia del VRP con abastecimiento de combustible con tres clientes y una gasolinera.

3.2. Modelo de programación lineal entera

El modelo exacto que proponemos en esta tesis para el VRP con abastecimiento de combustible se basa en representar una solución del problema como una secuencia de vértices que inicia y finaliza con el almacén y que contiene a todos los clientes en subsecuencias delimitadas por

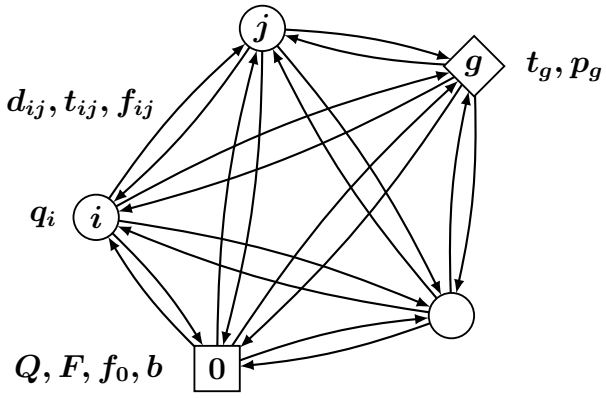
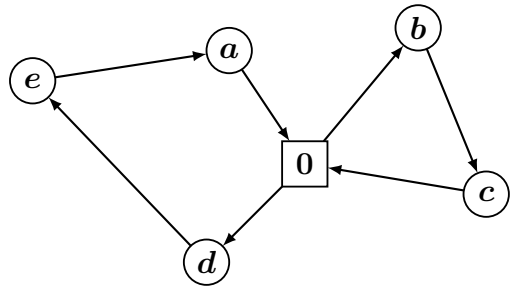


Figura 3.1: Diagrama del VRP con abastecimiento de combustible.

inclusiones intermedias del almacén, el cual representa simultáneamente el final de una ruta y el inicio de otra. Cada ruta debe atender al menos a un cliente, por lo que siempre debe aparecer un cliente entre cada dos apariciones del almacén. La figura 3.2 muestra la solución gráfica de una instancia de VRP con cinco clientes y dos vehículos y su secuencia correspondiente. El primer y el último elemento de la secuencia siempre corresponden al almacén. El número de subsecuencias de clientes consecutivos corresponde con la cantidad de vehículos disponibles en la instancia y cada subsecuencia denota los clientes que son atendidos por un vehículo en una misma ruta. Como los almacenes intermedios representan el final y el inicio de dos rutas adyacentes, el número de vértices a visitar en la solución de una instancia de VRP con n clientes y k vehículos es al menos $n + k + 1$.



0	d	e	a	0	b	c	0

Figura 3.2: Representación numérica de una solución de VRP.

El cálculo de los vértices necesarios para representar una solución del VRP con abastecimiento de combustible además debe considerar la posibilidad de visitar gasolineras para

abastecer combustible durante los viajes. Dada la distancia total de cualquier ruta calculada para las instancias de esta tesis y la capacidad considerada del tanque de combustible, supondremos que nunca es necesario visitar más de una gasolinera en una misma ruta, por lo que el número máximo de vértices a visitar es $n+2k+1$. Como cada vehículo debe terminar su ruta con al menos la cantidad de combustible con la que partió del almacén para garantizar que tenga suficiente combustible inicial en la siguiente jornada de trabajo, esto lo obliga a cargar gasolina al menos una vez para reponer la gasolina usada durante su ruta. Por lo tanto, una solución del modelo será una secuencia de exactamente $s = n+2k+1$ vértices. El cuadro 3.1 resume la notación de los conjuntos y constantes descritos en los párrafos anteriores y que son usados en la formulación del modelo propuesto.

V	: conjunto de $n+m+1$ vértices.
N	: conjunto de n clientes.
M	: conjunto de m gasolineras.
K	: conjunto de k vehículos disponibles.
0	: vértice que denota el centro de distribución o almacén.
Q	: capacidad máxima de carga de los vehículos.
F	: capacidad máxima del tanque de combustible de los vehículos.
b	: tasa de rendimiento de combustible.
f_0	: nivel inicial de combustible de cada vehículo.
q_v	: demanda de producto del cliente v.
t_g	: tiempo de servicio en la gasolinera g.
p_g	: precio de la gasolina en la gasolinera g.
d_{vu}	: distancia que se recorre al viajar directamente del vértice v al vértice u.
t_{vu}	: tiempo de viaje entre el vértice v y el vértice u.
f_{vu}	: consumo de combustible causado por viajar del vértice v al vértice u.
s	: cantidad de vértices a visitar en una secuencia solución.

Cuadro 3.1: Conjuntos y constantes del modelo.

Nuestra formulación hace uso de tres tipos de variables de decisión: *1)* variables de decisión binarias que determinan la secuencia de clientes y almacenes; *2)* variables de decisión enteras que modelan la capacidad restante de producto en los vehículos; y *3)* variables de decisión continuas que representan el nivel de combustible en los vehículos. El cuadro 3.2 reúne la definición de todas las variables de decisión. El modelo matemático propuesto es:

$$\text{mín } \Big(\sum_{(v,u)\in A} \sum_{i=1}^{s-1} d_{vu} x_{vui} \ , \ \sum_{(v,u)\in A} \sum_{i=1}^{s-1} t_{vu} x_{vui} + \sum_{g\in M} \sum_{i=1}^{s} t_g y_{gi} \ , \ \sum_{g\in M} \sum_{i=1}^{s} p_g r_{gi} \Big) \tag{3.1}$$

y_{vi} :	indica si el vértice v aparece en la posición i de la secuencia solución (binaria).
x_{vui} :	indica si el vértice v aparece en la posición i de la secuencia y el vértice u aparece en la posición $i+1$ de la misma (binaria).
n_i :	cantidad restante de producto en el vehículo tras atender la posible demanda del vértice que aparece en la posición i de la secuencia (entera).
r_{gi} :	cantidad de combustible a reabastecer en la gasolinera g cuando ésta aparece en la posición i de la secuencia (continua).
l_i :	nivel restante de combustible en el vehículo tras procesar la posible recarga hecha en el vértice que aparece en la posición i de la secuencia (continua).
e_i :	variable auxiliar para calcular l_i cuando el almacén aparece en la posición i de la secuencia (continua).

Cuadro 3.2: Variables de decisión del modelo.

sujeto a:

$$\sum_{v \in V} y_{vi} = 1 \qquad \forall i \in \{1, 2, \ldots, s\} \tag{3.2}$$

$$\sum_{i=1}^{s} y_{vi} = 1 \qquad \forall v \in N \tag{3.3}$$

$$y_{01} = 1 \tag{3.4}$$

$$y_{0s} = 1 \tag{3.5}$$

$$\sum_{v \in V \setminus \{0\}} \sum_{i=1}^{s-1} x_{v0i} = k \tag{3.6}$$

$$n_1 = Q \tag{3.7}$$

$$n_i \geq Q y_{0i} \qquad \forall i \in \{2, 3, \ldots, s\} \tag{3.8}$$

$$n_i \leq Q y_{0i} + n_{i-1} - \sum_{v \in N} q_v y_{vi} \qquad \forall i \in \{1, 2, \ldots, s\} \tag{3.9}$$

$$l_1 = f_0 \tag{3.10}$$

$$l_i \geq \sum_{(v,u) \in A} f_{vu} x_{vui} \qquad \forall i \in \{1, 2, \ldots, s-1\} \tag{3.11}$$

$$r_{gi} \leq F y_{gi} \qquad \forall g \in M, \forall i \in \{1, 2, \ldots, s\} \tag{3.12}$$

$$l_i - \sum_{v \in V \setminus \{0\}} f_{v0} x_{v0i} \geq f_0 y_{0i+1} \qquad \forall i \in \{1, 2, \ldots, s-1\} \tag{3.13}$$

$$e_i \leq l_{i-1} - \sum_{(v,u) \in A} f_{vu} x_{vui-1} + \sum_{g \in M} r_{gi} \qquad \forall i \in \{2, 3, \ldots, s\} \tag{3.14}$$

$$e_i \leq F(1 - y_{0i}) \qquad \forall i \in \{1, 2, \ldots, s\} \tag{3.15}$$

$$l_i = e_i + f_0 y_{0i} \qquad \forall i \in \{1, 2, \ldots, s\} \tag{3.16}$$

$$y_{vi} \in \{0,1\} \qquad \forall v \in V, \forall i \in \{1,2,\ldots,s\} \tag{3.17}$$

$$x_{vui} = y_{vi} \wedge y_{ui+1} \qquad \forall v,u \in V, \forall i \in \{1,2,\ldots,s-1\} \tag{3.18}$$

$$0 \leq n_i \leq Q \qquad \forall i \in \{1,2,\ldots,s\} \tag{3.19}$$

$$0 \leq l_i, r_{gi}, e_i \leq F \qquad \forall g \in M, \forall i \in \{1,2,\ldots,s\} \tag{3.20}$$

Las tres funciones objetivo consideradas en el planteamiento del problema se integran en una única función vectorial (3.1). El primer término denota la suma de las distancias de los arcos que describen el recorrido de todas las rutas que conforman la solución. Entre cada pareja de elementos sucesivos de la secuencia solución existe un traslado, el cual contribuye con la distancia asociada al arco correspondiente.

$$\sum_{(v,u)\in A} \sum_{i=1}^{s-1} d_{vu} x_{vui} \tag{3.1}$$

El segundo término representa el tiempo total de la solución, calculado como la suma de los tiempos de recorrido de los arcos que conforman las rutas y la suma de los tiempos de espera relacionados con el abastecimiento de combustible.

$$\sum_{(v,u)\in A} \sum_{i=1}^{s-1} t_{vu} x_{vui} + \sum_{g\in M} \sum_{i=1}^{s} t_g y_{gi} \tag{3.1}$$

El tercer término es la suma de los costos monetarios derivados de la compra de combustible en gasolineras.

$$\sum_{g\in M} \sum_{i=1}^{s} p_g r_{gi} \tag{3.1}$$

La ecuación (3.2) asegura que todo elemento de la secuencia corresponda con exactamente un vértice y la ecuación (3.3) verifica que cada vértice que corresponda con un cliente aparezca exactamente una vez en la secuencia.

$$\sum_{v\in V} y_{vi} = 1 \quad \forall i \in \{1,2,\ldots,s\} \tag{3.2}$$

$$\sum_{i=1}^{s} y_{vi} = 1 \quad \forall v \in N \tag{3.3}$$

Las igualdades (3.4) y (3.5) garantizan que el primer y el último elemento de la secuencia solución correspondan con el almacén.

$$y_{01} = 1 \tag{3.4}$$

$$y_{0s} = 1 \tag{3.5}$$

Cada vez que un vehículo llega al almacén procedente de un vértice distinto, ocurre el final de una ruta. La restricción (3.6) pide que cualquier solución factible tenga una cantidad de subsecuencias de clientes igual a la cantidad de vehículos disponibles.

$$\sum_{v \in V \setminus \{0\}} \sum_{i=1}^{s-1} x_{v0i} = k \tag{3.6}$$

Las restricciones (3.7), (3.8) y (3.9) imponen cotas de capacidad de producto en los vehículos dependiendo del tipo de vértice que se visita en cada posición de la secuencia solución. Si el vértice de la posición i corresponde con el almacén, el vehículo se llena completamente de producto por (3.8). De lo contrario, (3.9) especifica que la cantidad de producto que queda en el vehículo tras visitar el vértice de la posición i de la secuencia depende de la cantidad de producto que queda tras visitar el vértice anterior y de la demanda en la posición i.

$$n_1 = Q \tag{3.7}$$

$$n_i \geq Q y_{0i} \quad \forall i \in \{1, 2, \ldots, s\} \tag{3.8}$$

$$n_i \leq Q y_{0i} + n_{i-1} - \sum_{v \in N} q_v y_{vi} \quad \forall i \in \{2, 3, \ldots, s\} \tag{3.9}$$

Las restricciones de (3.10) a (3.16) tratan lo relacionado con el consumo y abastecimiento de combustible. La restricción (3.11) asegura que el nivel de combustible en el tanque del vehículo sea suficiente para llegar al destino próximo, mientras que la restricción (3.12) obliga a que sólo sea posible abastecer combustible en gasolineras.

$$l_1 = f_0 \tag{3.10}$$

$$l_i \geq \sum_{(v,u) \in A} f_{vu} x_{vui} \quad \forall i \in \{1, 2, \ldots, s-1\} \tag{3.11}$$

$$r_{gi} \leq F y_{gi} \quad \forall g \in M, \forall i \in \{1, 2, \ldots, s\} \tag{3.12}$$

El nivel de combustible al final de cada ruta debe ser mayor o igual al nivel inicial con el que los vehículos parten del almacén debido a (3.13). Esta restricción en el modelo garantiza que los vehículos tengan suficiente combustible al comienzo de la siguiente jornada de trabajo.

$$l_i - \sum_{v \in V \setminus \{0\}} f_{v0} x_{v0i} \geq f_0 y_{0i+1} \quad \forall i \in \{1, 2, \ldots, s-1\} \tag{3.13}$$

El valor de la variable de decisión e_i se define en (3.14) y (3.15) y sirve para determinar el valor de l_i en (3.16) según el tipo de vértice en la posición i de la secuencia. Si el vértice en la posición i corresponde con un cliente, el valor de e_i en (3.14) depende únicamente del nivel de combustible que el vehículo tenía en el vértice anterior, así como de la cantidad de combustible consumida en el arco recorrido. Si el vértice en la posición i corresponde con una gasolinera, el valor de e_i también toma en consideración la recarga a realizar en dicha gasolinera. Por otro lado, la desigualdad (3.15) obliga a que e_i valga 0 cuando el vértice en la posición i sea el almacén. Finalmente, la ecuación (3.16) determina el nivel de combustible del vehículo al llegar al vértice en la posición i de la solución, asignándole e_i si no se llega al almacén, o bien, el nivel inicial de combustible si sí se llega al almacén.

$$e_i \leq l_{i-1} - \sum_{(v,u) \in A} f_{vu} x_{vui-1} + \sum_{g \in M} r_{gi} \quad \forall i \in \{2, 3, \ldots, s\} \tag{3.14}$$

$$e_i \leq F(1 - y_{0i}) \quad \forall i \in \{1, 2, \ldots, s\} \tag{3.15}$$

$$l_i = e_i + f_0 y_{0i} \quad \forall i \in \{1, 2, \ldots, s\} \tag{3.16}$$

Las variables binarias que modelan el traslado entre vértices se definen en (3.18) como la conjunción de dos variables de decisión binarias. La solución usa una vez el arco entre el vértice v y el vértice u cuando estos vértices aparecen en las posiciones i e $i+1$ de la secuencia respectivamente.

$$y_{vi} \in \{0, 1\} \quad \forall v \in V, \forall i \in \{1, 2, \ldots, s\} \tag{3.17}$$

$$x_{vui} = y_{vi} \wedge y_{ui+1} \quad \forall v, u \in V, \forall i \in \{1, 2, \ldots, s-1\} \tag{3.18}$$

Finalmente, las cotas de las variables de decisión que modelan la oferta disponible de producto y el nivel de combustible están dadas por (3.19) y (3.20) respectivamente.

$$0 \leq n_i \leq Q \quad \forall i \in \{1, 2, \ldots, s\} \tag{3.19}$$

$$0 \leq l_i, r_{gi}, e_i \leq F \quad \forall g \in M, \forall i \in \{1, 2, \ldots, s\} \tag{3.20}$$

Capítulo 4

Instancias y métodos de resolución

En este capítulo detallamos las pruebas experimentales realizadas y presentamos los resultados obtenidos para el problema propuesto. En la primera sección del capítulo, describimos las características de un conjunto de instancias generadas a partir de información real del tráfico y de la oferta de diésel de gasolineras ubicadas en la Ciudad de México. En la segunda sección describimos la técnica básica de optimización multiobjetivo con la que se abordó la resolución del modelo matemático, así como la implementación de dos heurísticas que generan soluciones parciales a ser completadas por el solucionador para obtener soluciones iniciales factibles y acelerar el proceso de resolución. En esta misma sección describimos la implementación de un algoritmo de búsqueda local con el que se asistió al solucionador durante el proceso de resolución del modelo. En la tercera y última sección, analizamos los resultados obtenidos.

4.1. Instancias

A falta de instancias conocidas que se ajusten plenamente a la descripción del problema, se recabó información sobre las 137 gasolineras con venta de diésel que existen dentro de las coordenadas geográficas (19.3, -99.025) y (19.5, -99.25), zona que abarca diferentes alcaldías de la Ciudad de México y parte de algunos municipios del Estado de México. La figura 4.1 muestra la zona geográfica referida. Las ubicaciones de las gasolineras y los precios de venta del combustible fueron tomados del *Listado de precios comerciales de gasolina y diésel por estación de servicio* y del *Listado de estaciones de servicio con georreferencia* publicados por la Comisión Reguladora de Energía (CRE) en julio de 2019 [7, 8]. El tiempo de espera en cada gasolinera se estimó considerando que las gasolineras más céntricas o las que ofrecen precios más baratos son las de mayor tiempo de espera. Estos varían aproximadamente entre 2 y 12 minutos, y son independientes de la cantidad de combustible que se abastece.

Adicionalmente se generaron de manera aleatoria 263 coordenadas uniformemente distribuidas dentro de los mismos rangos para denotar ubicaciones de clientes y se usó el servicio de *Google Maps* para calcular la distancia y el tiempo de recorrido del viaje más corto entre todo par de coordenadas (tanto clientes como gasolineras) en ambos sentidos. La demanda de los clientes sigue una distribución geométrica que simula una situación en la que la mayoría de los clientes tienen una demanda pequeña y muy pocos clientes tienen una demanda grande.

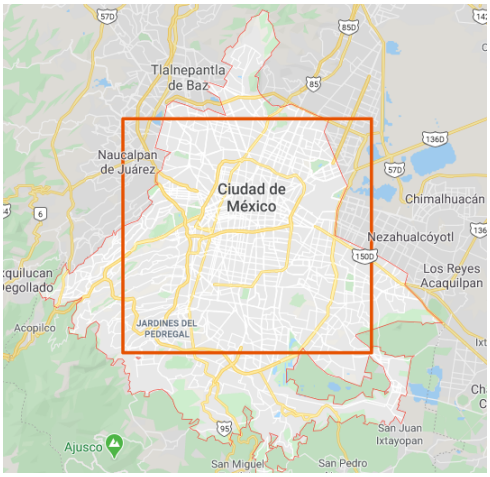
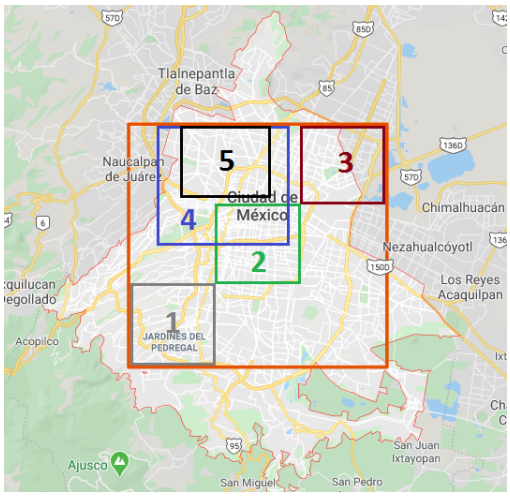


Figura 4.1: Enmarcación de una zona de la Ciudad de México.

Instancia	Contraesquinas	Vértices	Vehículos	Gasolineras	Diferencia máxima de precios
1	(+19.300, -99.175), (+19.366, -99.250)	22	2	5	$1.24
2	(+19.366, -99.100), (+19.433, -99.175)	45	4	16	$1.00
3	(+19.433, -99.025), (+19.500, -99.100)	32	3	10	$1.30
4	(+19.400, -99.110), (+19.500, -99.220)	107	8	48	$2.44
5	(+19.440, -99.130), (+19.500, -99.220)	44	3	21	$2.34

Cuadro 4.1: Características de las instancias generadas.

A partir de estos datos, se construyeron cinco instancias que abarcan diferentes cuadrantes de la región anteriormente descrita. La figura 4.2 muestra la disposición de estos cuadrantes y la distribución de los vértices de cada instancia. El cuadro 4.1 muestra las características de las instancias construidas. En todos los casos, se consideran flotas homogéneas de vehículos con capacidad de carga de 12 unidades de producto y 70 litros de capacidad en el tanque de combustible. Al momento de partir del almacén todos los vehículos cuentan con 3 litros de combustible en el tanque. Se considera que un litro de combustible rinde 12 kilómetros. El número de vehículos en cada instancia se calculó como el mínimo necesario con respecto a la demanda total a satisfacer, con excepción de la instancia 4 que cuenta con un vehículo extra.

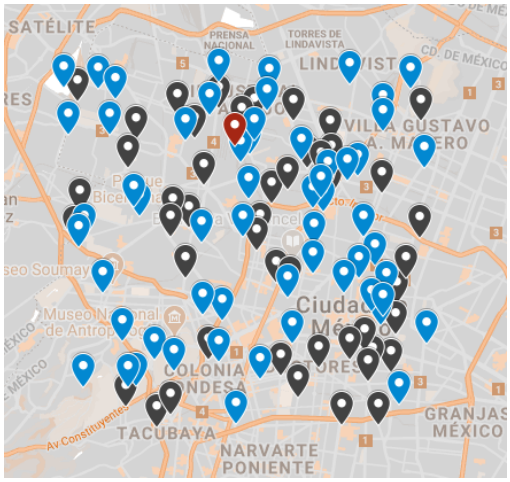


(a) Segmentación de la Ciudad de México en cuadrantes, correspondientes a las ubicaciones de las instancias generadas.

(b) Instancia 1.

(c) Instancia 2.

(d) Instancia 3.



(e) Instancia 4.

(f) Instancia 5.

Figura 4.2: Ubicación de las instancias y disposición de sus vértices. Los símbolos de ubicación rojos denotan el almacén en cada instancia, los azules denotan clientes y los negros, gasolineras con oferta de diésel.

4.2. Metodología de resolución

El problema de optimización multiobjetivo propuesto en esta tesis fue reducido a un problema de optimización mono-objetivo a través del método de sumas ponderadas. Este método asigna un peso a cada función objetivo y las combina en un solo término, volviendo el problema uno de optimización mono-objetivo sujeto a las mismas restricciones del problema original. El valor que se asigna a cada peso puede ser una medida de la importancia atribuida a la función objetivo correspondiente. La ecuación (4.1) muestra la combinación de las funciones objetivo del modelo descrito en el capítulo anterior, donde los coeficientes α, β y γ denotan los pesos asignados a cada término de la función objetivo resultante.

$$\alpha(\sum_{(v,u)\in A}\sum_{i=1}^{s-1} d_{vu}x_{vui}) + \beta(\sum_{(v,u)\in A}\sum_{i=1}^{s-1} t_{vu}x_{vui} + \sum_{g\in M}\sum_{i=1}^{s} t_g y_{gi}) + \gamma(\sum_{g\in M}\sum_{i=1}^{s} p_g r_{gi}) \tag{4.1}$$

El modelo debe resolverse múltiples veces para aplicar el método de sumas ponderadas, variando sistemáticamente los valores asignados a los pesos de las funciones objetivo, con la finalidad de intentar aproximar el frente de Pareto de las instancias generadas. Se usaron todas las combinaciones de α, β, γ donde los tres pesos son múltiplos de 10 y su suma vale 100. También se incluyó un caso donde los tres pesos valen lo mismo, por lo que cada conjunto de experimentos consiste de 67 ejecuciones para cada instancia, con un tiempo límite de 600 segundos de cómputo por ejecución. La implementación del modelo y del método de sumas ponderadas se realizó usando la interfaz de C++ de *Gurobi Optimizer*, versión 9.0 [17].

Con el propósito de acelerar el proceso de resolución del modelo, se seleccionaron dos heurísticas de VRP ampliamente conocidas y capaces de calcular buenas soluciones a un bajo costo computacional: el algoritmo glotón de ahorros y la heurística geométrica de barrido. Debido a que el problema planteado en esta tesis requiere un tamaño de la flota predeterminado donde cada vehículo visite al menos un cliente, ambos métodos tuvieron que ser adaptados porque las heurísticas originales no tienen control sobre el tamaño de la flota resultante. Las heurísticas también se adaptaron para calcular soluciones factibles de instancias realistas del VRP con abastecimiento (instancias donde surtir el tanque al inicio de la ruta permite completarla y regresar al almacén con el nivel mínimo requerido de combustible). Las soluciones producidas por las heurísticas son las rutas a usar, aunque desde el punto de vista del modelo entero están incompletas porque no asigna valores a todas las variables de decisión del mismo. Sin embargo, al entregarle al solucionador la colección de rutas, éste puede usarlas para construir una solución completa del modelo que sirva como solución inicial.

Cabe señalar que estas adaptaciones usan de una matriz de costos producida con el método de sumas ponderadas, obteniendo así una solución de calidad para la elección particular de pesos.

Nos referiremos a los métodos heurísticos propuestos en esta tesis como "Algoritmo glotón de ahorros para k vehículos con abastecimiento" y "Heurística geométrica de barrido para k vehículos con abastecimiento". La característica de poder entregar rutas con exactamente k vehículos fue implementada condicionando la asignación de clientes a vehículos a la determinación de factibilidad del problema asociado de empaquetado de contenedores (*Bin-Packing Problem*). Este problema busca determinar si es posible almacenar un conjunto de objetos de distintos tamaños en un conjunto de contenedores idénticos de capacidad limitada. El cuadro 4.2 muestra los conjuntos, las constantes y las variables de decisión relacionados a este problema de empaquetado, pero en términos de un problema de ruteo. También se expone un modelo lineal entero del problema referido, del cual sólo se busca determinar factibilidad:

N	: conjunto de n clientes.
K	: conjunto de k vehículos disponibles.
Q	: capacidad máxima de carga de los vehículos.
q_i	: demanda de producto del cliente i.
x_{ij}	: indica si el j-ésimo vehículo atiende al cliente i (binaria).
y_i	: indica el índice j del vehículo que atiende al cliente i (entera).

Cuadro 4.2: Componentes del problema de empaquetado de contenedores.

$$\sum_{j=1}^{k} x_{ij} = 1 \qquad \forall i \in N \tag{4.2}$$

$$\sum_{j=1}^{k} j x_{ij} = y_i \qquad \forall i \in N \tag{4.3}$$

$$\sum_{i \in N} x_{ij} \geq 1 \qquad \forall j \in \{1, 2, \ldots, k\} \tag{4.4}$$

$$\sum_{i \in N} q_i x_{ij} \leq Q \qquad \forall j \in \{1, 2, \ldots, k\} \tag{4.5}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i \in N, \forall j \in \{1, 2, \ldots, k\} \tag{4.6}$$

$$1 \leq y_i \leq k \qquad \forall i \in N \tag{4.7}$$

La ecuación (4.2) indica que cada cliente debe ser atendido por un solo vehículo. El cálculo de la variable y_i mediante la ecuación (4.3) es sumamente útil en el algoritmo glotón de ahorros para k vehículos, como se explica posteriormente. La restricción (4.4) obliga que ningún

vehículo sea ignorado en la asignación de clientes y (4.5) vigila que la suma de las demandas de los clientes asignados a cualquier vehículo nunca rebasen su capacidad máxima Q. Finalmente, (4.6) y (4.7) establecen el dominio de las variables de decisión. En la práctica, Gurobi resuelve la factibilidad del problema de empaquetado de contenedores de forma instantánea. Las heurísticas propuestas aprovechan esta situación y comprueban la factibilidad de este subproblema en cada iteración que intenta asignar un cliente a un vehículo.

En el algoritmo glotón de ahorros para k vehículos, la condición que usamos para unir las dos rutas que más conviene unir es la determinación de la factibilidad del problema de empaquetamiento al agregar una restricción $y_i = y_h$, donde i y h son clientes de tales rutas. Si el nuevo modelo es factible, entonces un mismo vehículo tiene capacidad para atender a los clientes de ambas rutas, de lo contrario se descarta la unión de ese par de clientes y se desecha la restricción agregada, repitiendo el proceso para el siguiente mayor ahorro.

En el caso de la heurística geométrica de barrido para k vehículos, el mecanismo implementado tiene una naturaleza distinta. Como se describió en el capítulo 2, la heurística geométrica de barrido asigna clientes a vehículos dependiendo del ángulo que forman con el almacén y de la capacidad restante del vehículo. En la heurística original, si la demanda del cliente por asignar no puede ser atendida por el vehículo en turno, se asume que el vehículo está lleno y el cliente en cuestión se asigna a un nuevo vehículo vacío. Esta estrategia puede ocasionar que exista una diferencia importante entre la capacidad de carga utilizada por los vehículos y su capacidad disponible Q, lo que genera soluciones que usan más vehículos de los necesarios. Por otro lado, si para alguna instancia ocurriera que la capacidad de carga de los primeros vehículos se usa casi en su totalidad, puede ser que éstos atiendan a todos o casi todos los clientes, resultando en vehículos inutilizados o en soluciones muy desbalanceadas. La adaptación propuesta primero calcula el nivel de utilización ideal dividiendo la suma de todas las demandas entre el número de vehículos disponibles. Cada vez que el algoritmo de barrido pretende asignar un cliente i, la heurística modificada verifica si la utilización del j-ésimo vehículo en turno iguala o supera el nivel de utilización ideal. Si esto es cierto, se intenta prohibir la asignación del cliente i al j-ésimo vehículo y se resuelve la factibilidad del problema de empaquetado de contenedores agregando la restricción $x_{ij} = 0$. Por otra parte, si la utilización del j-ésimo vehículo es menor al nivel de utilización ideal o si el modelo anterior resulta infactible, el algoritmo intenta asignarle el cliente i ahora usando la restricción $x_{ij} = 1$. De esta forma, siempre se obtienen soluciones que usan exactamente la cantidad deseada de vehículos, si esto es posible.

Otros aspectos menores en la implementación de estos métodos heurísticos incluyen el manejo de costos asimétricos y la inclusión de gasolineras en las rutas calculadas. La inserción de gasolineras se resolvió primero ignorando las gasolineras en la fase de construcción de rutas de ambas heurísticas y posteriormente insertando como primera parada de cada ruta la gasolinera que genera la menor desviación con respecto al trayecto que se recorre del almacén al primer cliente de la ruta. Esta forma de incluir visitas a gasolineras genera rutas sub-óptimas, pero suficientemente detalladas para que el solucionador las considere soluciones parciales válidas. De esta manera, se delegó al solucionador la tarea de calcular los valores de las variables de decisión no especificadas para así completar una solución inicial factible.

Ambas implementaciones también incluyen como subrutina a la heurística de búsqueda local 2-opt que intenta mejorar la calidad individual de las rutas obtenidas. Dada una ruta, el vecindario de búsqueda en 2-opt está formado por todas las rutas obtenidas al invertir el orden de visita de una subsecuencia de clientes de la ruta original. Una ruta 2-óptima también se conoce como una ruta óptima en relación a inversión. A su vez, se desarrolló una subrutina cooperativa de 2-opt para que el solucionador la invoque durante el proceso de optimización. Cada vez que el solucionador encuentra una nueva mejor solución, la heurística intenta mejorarla y, en caso de lograrlo, se la devuelve. La implementación de las tres heurísticas se realizó en C++ y la rutina para comprobar la factibilidad del problema de empaquetado de contenedores se implementó con la interfaz de C++ de *Gurobi Optimizer* versión 9.0.

4.3. Resultados computacionales

A continuación se muestran gráficas y tablas que presentan los resultados obtenidos por cada método de resolución. Para cada instancia, se grafican los valores de las soluciones para cada uno de los múltiples objetivos del problema, agrupados de la siguiente manera: *i)* distancia recorrida contra tiempo de recorrido, *ii)* distancia recorrida contra compra de combustible, *iii)* tiempo de recorrido contra compra de combustible, y *iv)* todos los objetivos. A su vez, los resultados obtenidos se separaron en dos grupos de gráficas. El primer grupo muestra los resultados obtenidos por la resolución del modelo exacto y por cada heurística individual, mientras que el segundo grupo compara resultados obtenidos por combinaciones de técnicas. Esto da un total de ocho gráficas por instancia. Las gráficas de ambos grupos se presentan intercaladas para facilitar la comparación entre los resultados obtenidos.

Las gráficas del primer grupo muestran cuatro series de datos. La primera serie corresponde con las soluciones obtenidas al resolver el modelo con el solucionador, limitado a 600 segundos de cómputo. La segunda serie muestra las soluciones obtenidas al resolver el modelo de forma cooperativa con la heurística 2-opt, también con tiempo límite de 600 segundos de cómputo. Las dos últimas series corresponden con la primera solución factible calculada por el solucionador a partir de la solución parcial obtenida por el algoritmo glotón de ahorros para k vehículos con abastecimiento y por la heurística geométrica de barrido para k vehículos con abastecimiento, respectivamente. La implementación de los métodos heurísticos escriben sus soluciones en archivos con formato MST (*MIP Start*) listando los valores para el subconjunto de las variables del modelo que especifican la secuencia solución. Cuando Gurobi importa un archivo en formato MST, éste intenta construir una solución inicial factible a partir de los valores especificados en el archivo. Los tiempos de ejecución de las heurísticas de ahorros y de barrido fueron menores a uno y cinco segundos respectivamente, mientras que el tiempo que tarda el solucionador en completar una solución factible fue menor a 15 segundos.

Las gráficas del segundo grupo muestran tres series de datos. La primera serie corresponde con las soluciones obtenidas al resolver el modelo de forma cooperativa con la heurística 2-opt. Las dos últimas series corresponden con los resultados obtenidos al resolver el modelo también en cooperación con 2-opt pero tomando como solución inicial la calculada por el algoritmo glotón de ahorros para k vehículos con abastecimiento y por la heurística geométrica de barrido para k vehículos con abastecimiento, respectivamente. El tiempo de ejecución de cada método se limitó a 600 segundos.

En general, las gráficas que muestran compra de combustible presentan un patrón que divide los resultados en dos grupos. Uno de ellos contiene soluciones con un alto costo derivado de la compra de combustible, las cuales corresponden a ejecuciones donde el peso asignado a minimizar la compra de combustible es cero, por lo que cada vez que un vehículo visita una gasolinera, llena el tanque completo e incurre en un gasto posiblemente innecesario. Estas soluciones fueron descartadas en las gráficas que contemplan los tres objetivos para mejorar la visualización de las soluciones con un costo monetario razonable. A su vez, la tabla de cada instancia resume los resultados numéricos obtenidos por todas las técnicas. Los resultados aparecen normalizados entre el mejor resultado obtenido para la combinación de pesos en cuestión. De este modo, el mejor resultado de cada combinación aparece como 1 y el resto de los resultados aparecen como la proporción en la que fueron peores que la mejor solución. El desempeño de cada técnica se resume como el promedio de sus resultados, igualmente

normalizado entre el promedio de la técnica con el mejor desempeño.

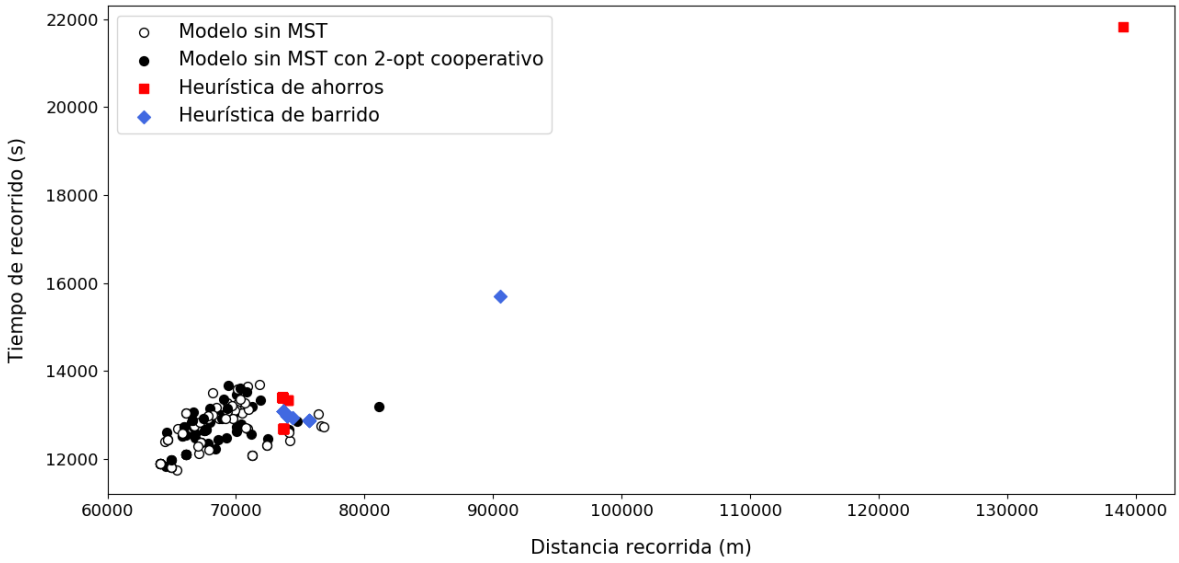


Figura 4.3: Instancia 1. Métodos de resolución independientes.

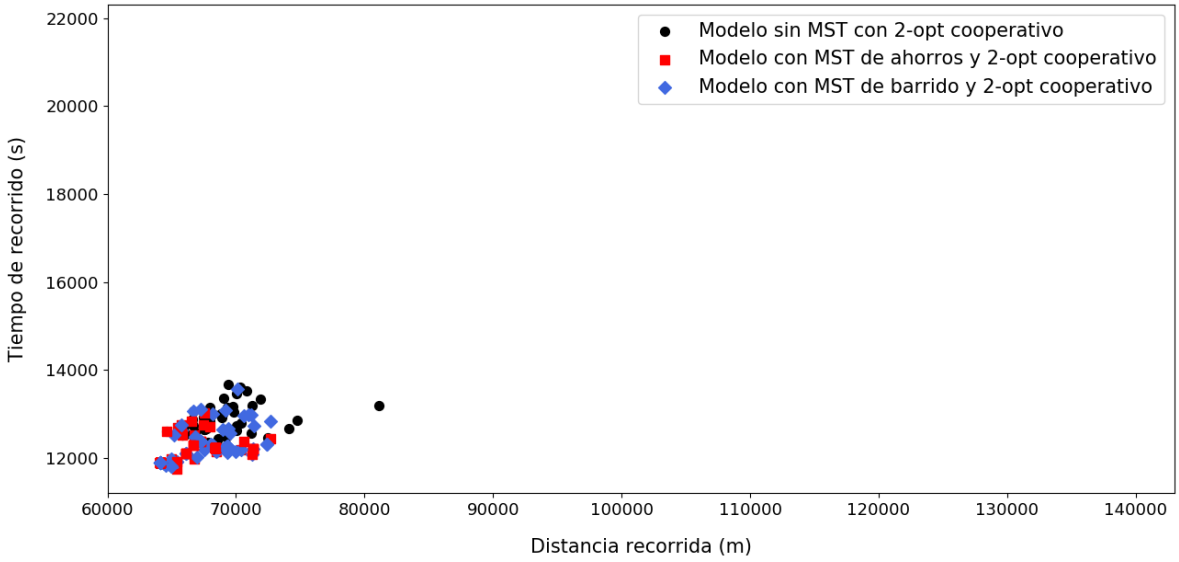


Figura 4.4: Instancia 1. Métodos de resolución combinados.

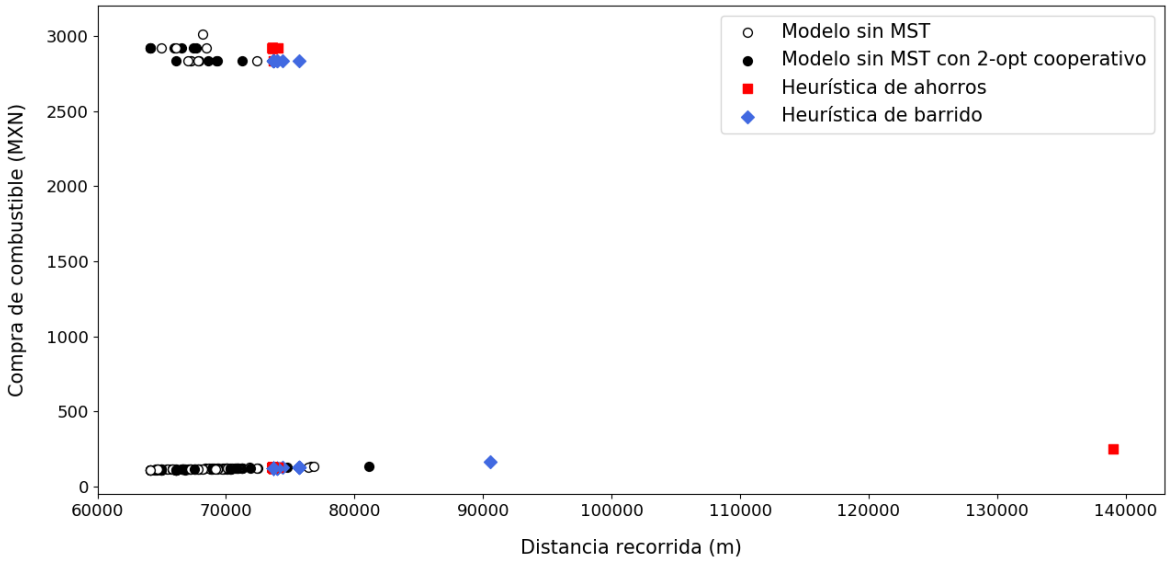


Figura 4.5: Instancia 1. Métodos de resolución independientes.

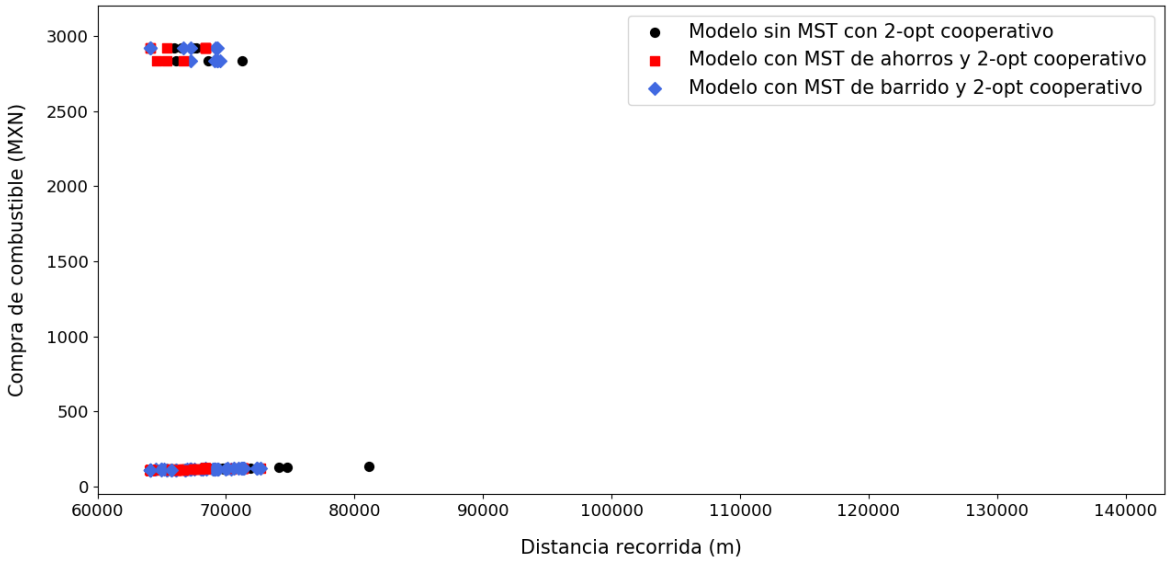


Figura 4.6: Instancia 1. Métodos de resolución combinados.

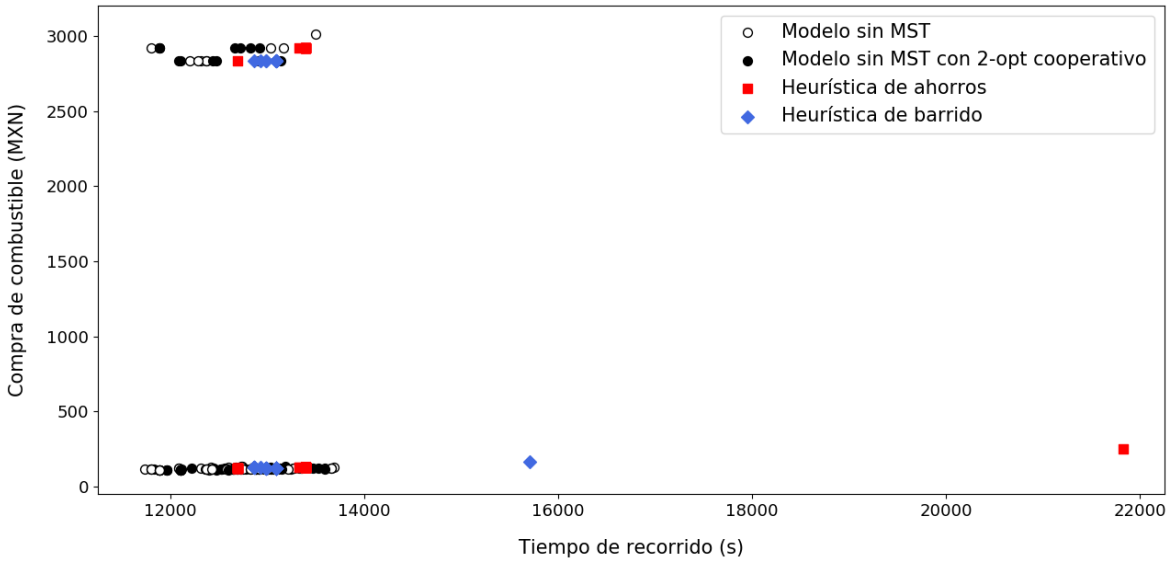


Figura 4.7: Instancia 1. Métodos de resolución independientes.

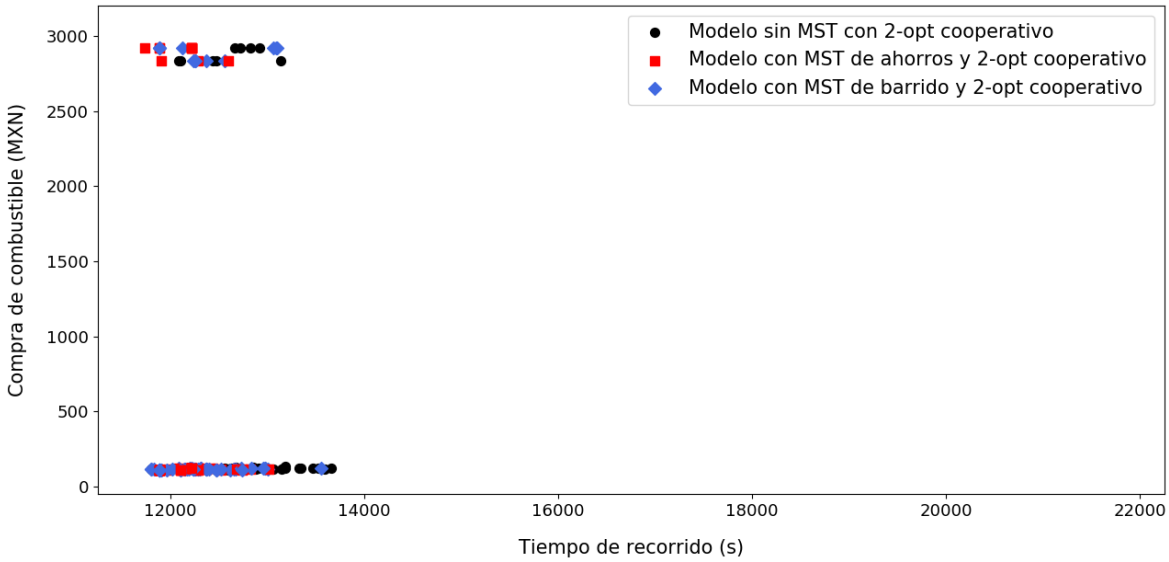


Figura 4.8: Instancia 1. Métodos de resolución combinados.

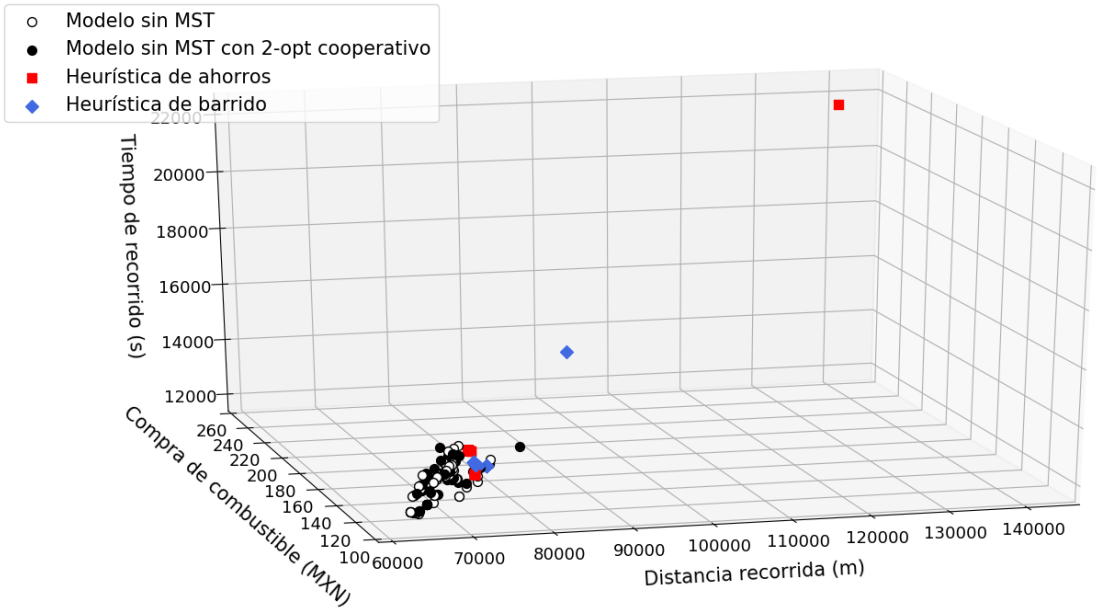


Figura 4.9: Instancia 1. Métodos de resolución independientes.

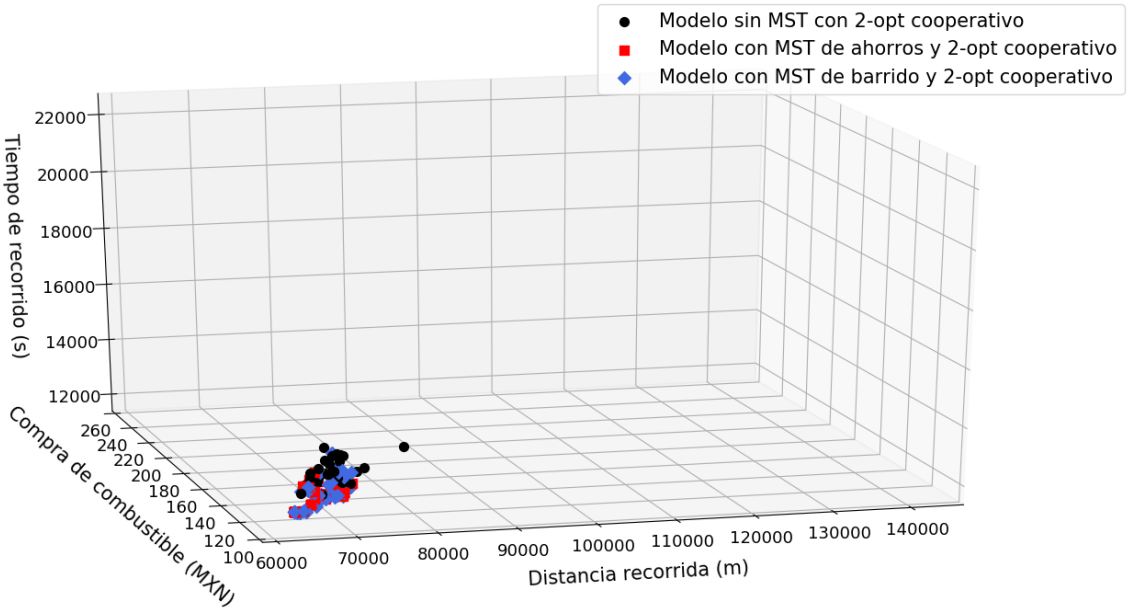


Figura 4.10: Instancia 1. Métodos de resolución combinados.

Cuadro 4.3: Comparación de los métodos de resolución para la instancia 1.

Pesos (d, t, p)	Modelo	Modelo con 2-opt	Ahorros	Barrido	Modelo con MST de ahorros y 2-opt	Modelo con MST de barrido y 2-opt
(1, 1, 1)	1.03	1.09	1.14	1.14	1	1.11
(0, 0, 100)	1	1	2.27	1.48	1	1
(0, 10, 90)	1.04	1.08	1.07	1.09	1.03	1
(0, 20, 80)	1.05	1.02	1.04	1.06	1	1.01
(0, 30, 70)	1	1.05	1.05	1.06	1.01	1.01
(0, 40, 60)	1.04	1.04	1.05	1.06	1	1.02
(0, 50, 50)	1.07	1.11	1.07	1.08	1.01	1
(0, 60, 40)	1.08	1.04	1.05	1.06	1.01	1
(0, 70, 30)	1	1.03	1.08	1.1	1.04	1.04
(0, 80, 20)	1.05	1.07	1.07	1.09	1.05	1
(0, 90, 10)	1.05	1.05	1.04	1.06	1	1
(0, 100, 0)	1.05	1.03	1.08	1.1	1	1.03
(10, 0, 90)	1.01	1.11	1.15	1.15	1	1.08
(10, 10, 80)	1.03	1	1.14	1.14	1.06	1.03
(10, 20, 70)	1.06	1.04	1.14	1.14	1	1.05
(10, 30, 60)	1.04	1	1.14	1.13	1.05	1.05
(10, 40, 50)	1.05	1.06	1.14	1.13	1	1.05
(10, 50, 40)	1.02	1	1.13	1.12	1.04	1.07
(10, 60, 30)	1.09	1.07	1.14	1.12	1.04	1
(10, 70, 20)	1.1	1	1.14	1.12	1.05	1.05
(10, 80, 10)	1.06	1	1.08	1.07	1.02	1
(10, 90, 0)	1.04	1.05	1.12	1.11	1	1.06
(20, 0, 80)	1.08	1	1.12	1.12	1.02	1.05
(20, 10, 70)	1.08	1.08	1.15	1.15	1	1.11
(20, 20, 60)	1.04	1.03	1.08	1.08	1	1.01
(20, 30, 50)	1.04	1.05	1.14	1.14	1	1.05
(20, 40, 40)	1.07	1	1.11	1.11	1.02	1.03
(20, 50, 30)	1.05	1.04	1.14	1.13	1.05	1
(20, 60, 20)	1.06	1.09	1.14	1.13	1	1.01
(20, 70, 10)	1	1.03	1.13	1.12	1.05	1.07
(20, 80, 0)	1	1.01	1.09	1.08	1	1.02
(30, 0, 70)	1.12	1.03	1.15	1.15	1	1.11
(30, 10, 60)	1	1.07	1.15	1.15	1	1.07
(30, 20, 50)	1.06	1	1.13	1.13	1.05	1.04
(30, 30, 40)	1.09	1.1	1.14	1.14	1	1.07
(30, 40, 30)	1.09	1.05	1.14	1.14	1	1.01
(30, 50, 20)	1.01	1	1.14	1.14	1.05	1.07
(30, 60, 10)	1.11	1.03	1.14	1.14	1	1.05
(30, 70, 0)	1.01	1	1.14	1.14	1.06	1.07
(40, 0, 60)	1.03	1.02	1.11	1.12	1.04	1
(40, 10, 50)	1.03	1.02	1.1	1.1	1.02	1
(40, 20, 40)	1	1.04	1.14	1.14	1.02	1.03
(40, 30, 30)	1.02	1.05	1.11	1.11	1	1.01
(40, 40, 20)	1	1.01	1.09	1.09	1.01	1
(40, 50, 10)	1.09	1	1.14	1.14	1.03	1.05
(40, 60, 0)	1.08	1.04	1.14	1.14	1.06	1
(50, 0, 50)	1.09	1.12	1.15	1.15	1	1.08

Continúa en la página siguiente.

Cuadro 4.3 – Continuación de la página anterior.

Pesos (d, t, p)	Modelo	Modelo con 2-opt	Ahorros	Barrido	Modelo con MST de ahorros y 2-opt	Modelo con MST de barrido y 2-opt
(50, 10, 40)	1.01	1.04	1.15	1.15	1.05	1
(50, 20, 30)	1.09	1.04	1.15	1.15	1	1.02
(50, 30, 20)	1.01	1	1.11	1.11	1.01	1.04
(50, 40, 10)	1	1	1.09	1.09	1.01	1.07
(50, 50, 0)	1	1.01	1.09	1.08	1.01	1.03
(60, 0, 40)	1.08	1.07	1.15	1.15	1.07	1
(60, 10, 30)	1.05	1.08	1.15	1.15	1.07	1
(60, 20, 20)	1.03	1.01	1.15	1.15	1.06	1
(60, 30, 10)	1.06	1.02	1.1	1.1	1	1.07
(60, 40, 0)	1.06	1.06	1.15	1.14	1	1.06
(70, 0, 30)	1.01	1.04	1.09	1.09	1	1.04
(70, 10, 20)	1.11	1.11	1.15	1.15	1	1
(70, 20, 10)	1.07	1.08	1.15	1.15	1.03	1
(70, 30, 0)	1	1.03	1.15	1.15	1	1
(80, 0, 20)	1.05	1	1.11	1.12	1.04	1.07
(80, 10, 10)	1	1.04	1.14	1.14	1.02	1.07
(80, 20, 0)	1.03	1.07	1.13	1.14	1	1.04
(90, 0, 10)	1	1.03	1.15	1.15	1.03	1.03
(90, 10, 0)	1.06	1	1.15	1.15	1.04	1.08
(100, 0, 0)	1	1.02	1.11	1.12	1.04	1.05
Promedio	**1.02**	**1.02**	**1.11**	**1.1**	**1**	**1.01**

Con respecto a la instancia 1, las figuras 4.3, 4.5, 4.7 y 4.9 muestran que los resultados obtenidos por el solucionador sin solución inicial no presentan una diferencia significativa cuando es o no asistido por 2-opt. A su vez, las soluciones calculadas por el solucionador superan los resultados obtenidos por los métodos heurísticos propuestos, a expensas de un tiempo de cómputo mucho más elevado. Para esta instancia pequeña, el tiempo total de ejecución de los métodos heurísticos son fracciones de segundo. En las figuras 4.4, 4.6, 4.8 y 4.10 las mejores soluciones obtenidas se calcularon con los métodos cooperativos que hacen uso de las soluciones heurísticas para completar soluciones iniciales. Cuando se compara el desempeño de los métodos heurísticos ejecutados de forma independiente con el desempeño de los métodos cooperativos que parten de una solución inicial heurística, se aprecia que las soluciones mejoran en términos de convergencia y dispersión. También ver el cuadro 4.3.

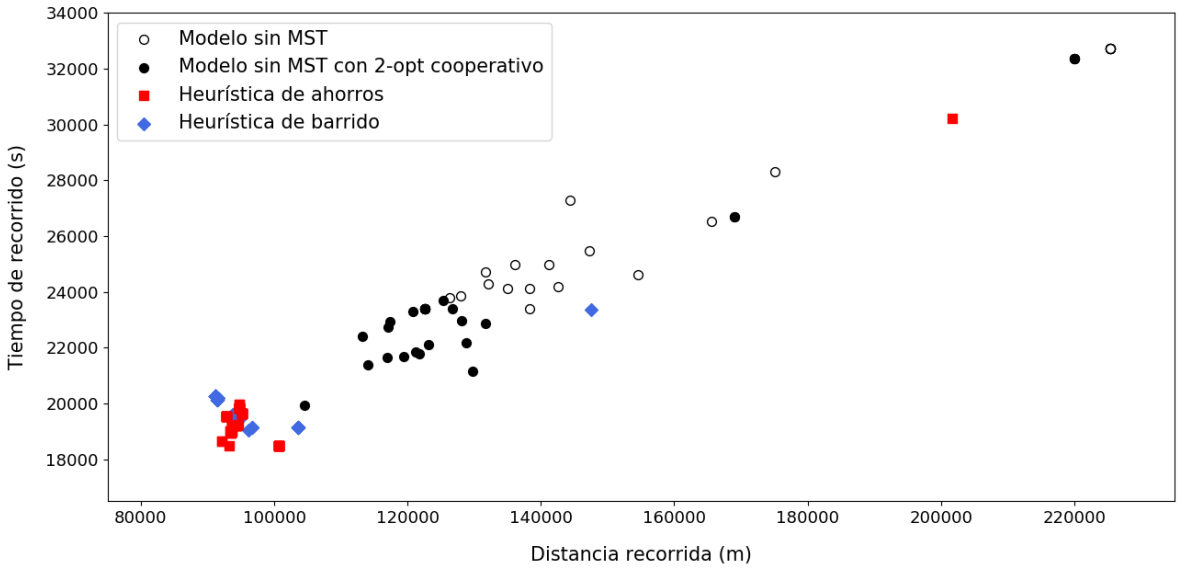


Figura 4.11: Instancia 2. Métodos de resolución independientes.

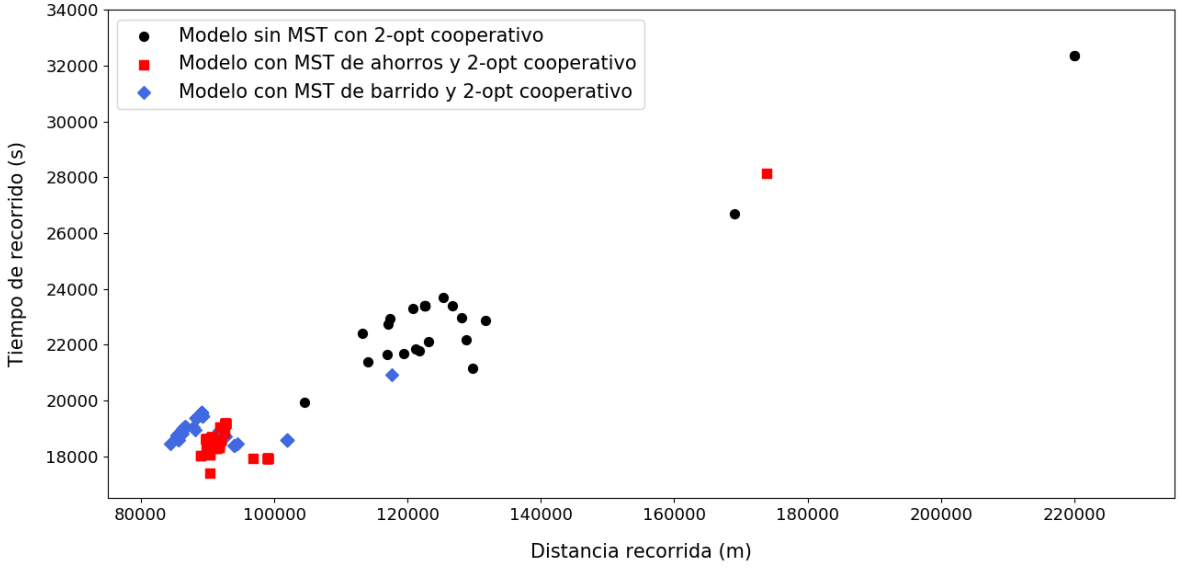


Figura 4.12: Instancia 2. Métodos de resolución combinados.

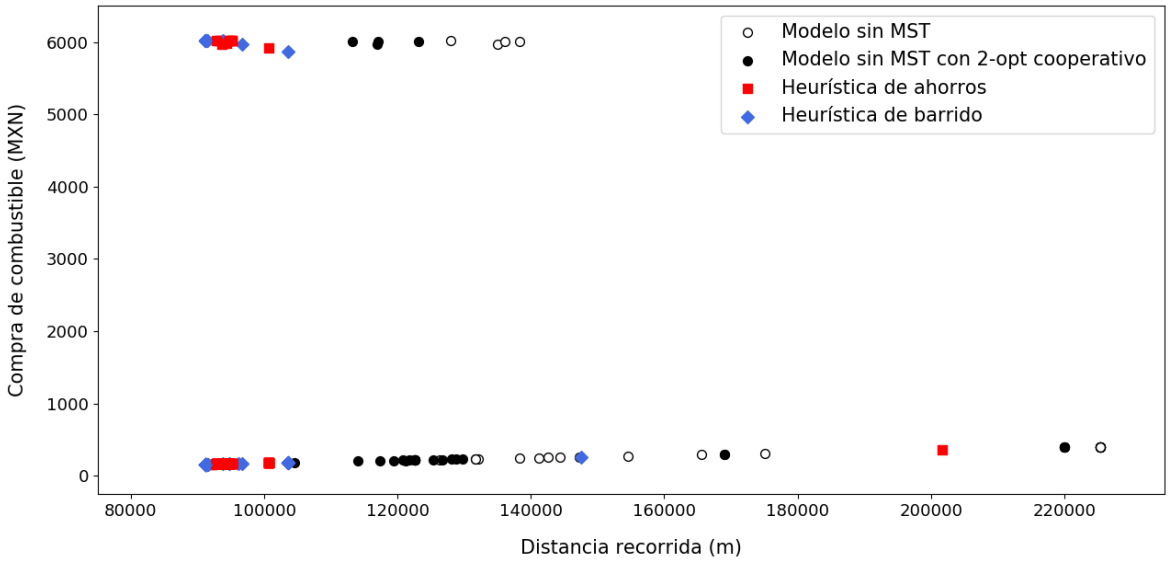


Figura 4.13: Instancia 2. Métodos de resolución independientes.

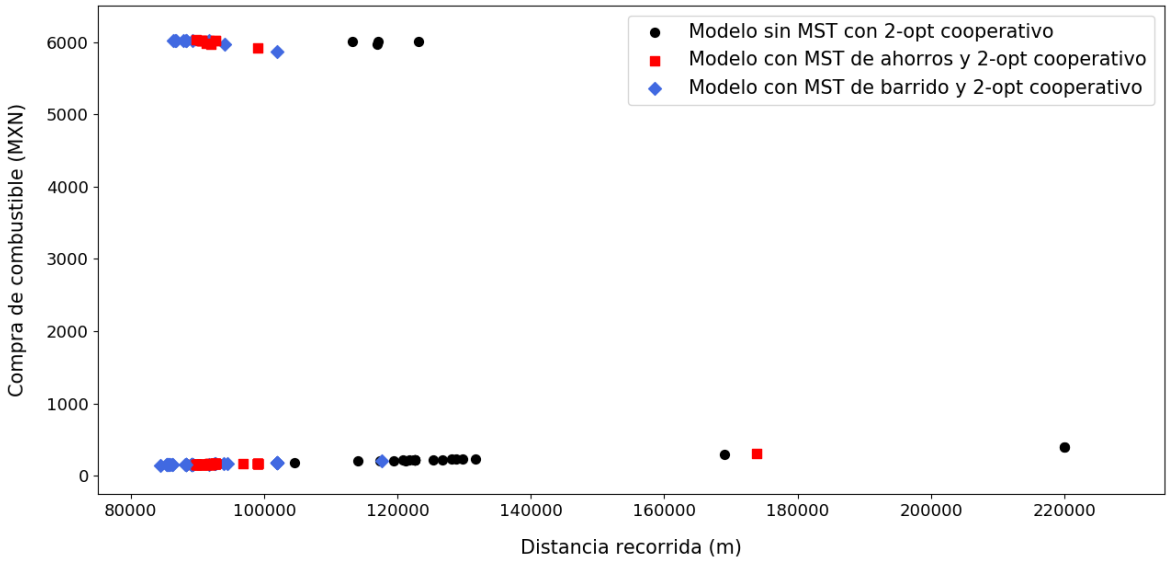


Figura 4.14: Instancia 2. Métodos de resolución combinados.

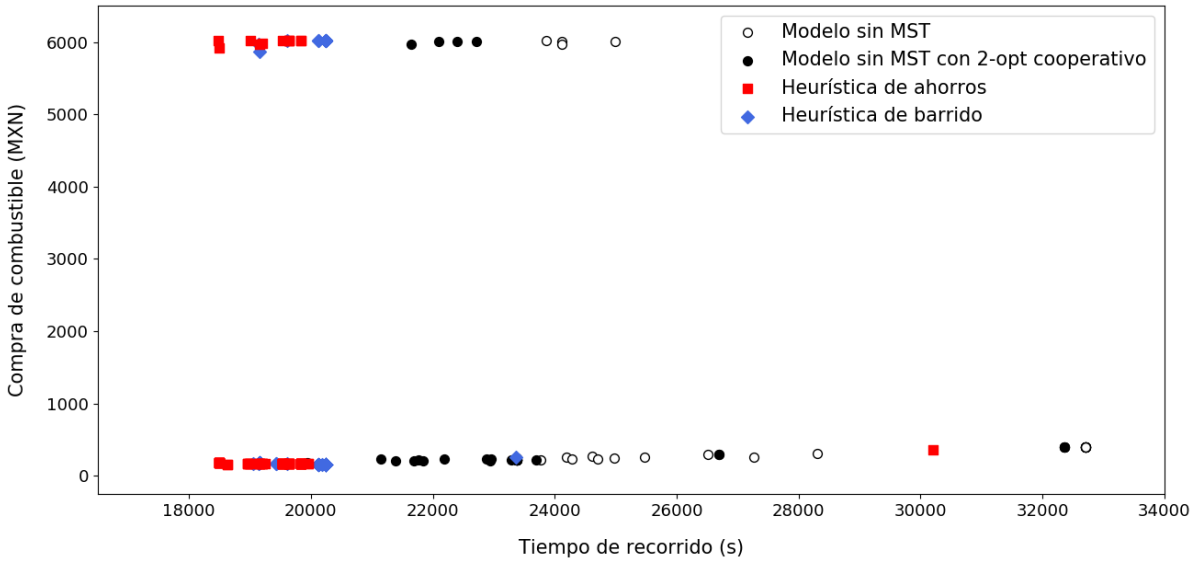


Figura 4.15: Instancia 2. Métodos de resolución independientes.

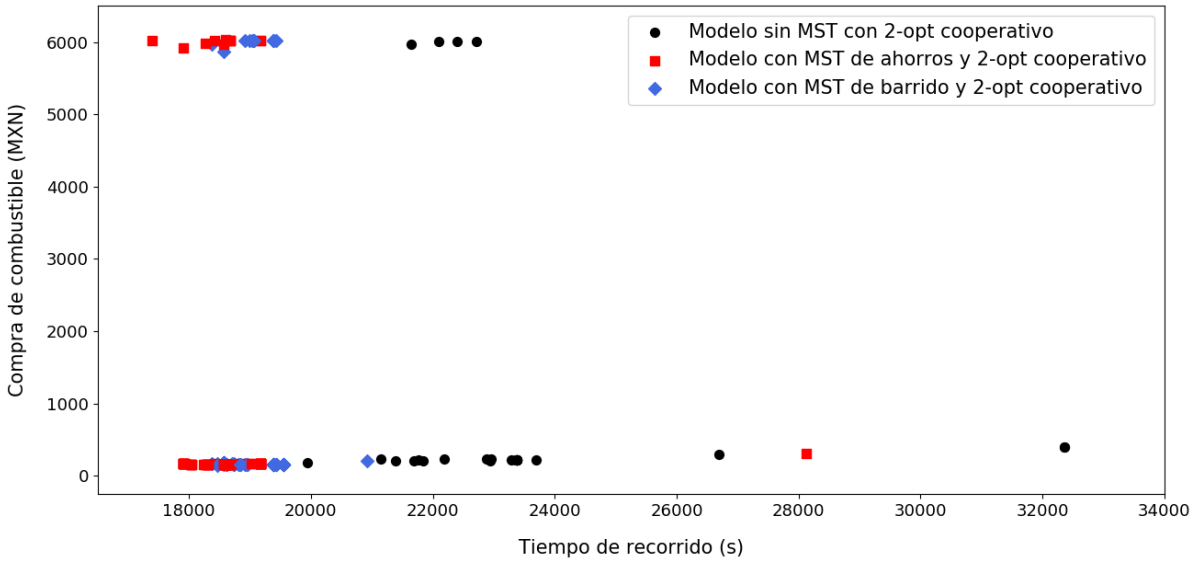


Figura 4.16: Instancia 2. Métodos de resolución combinados.

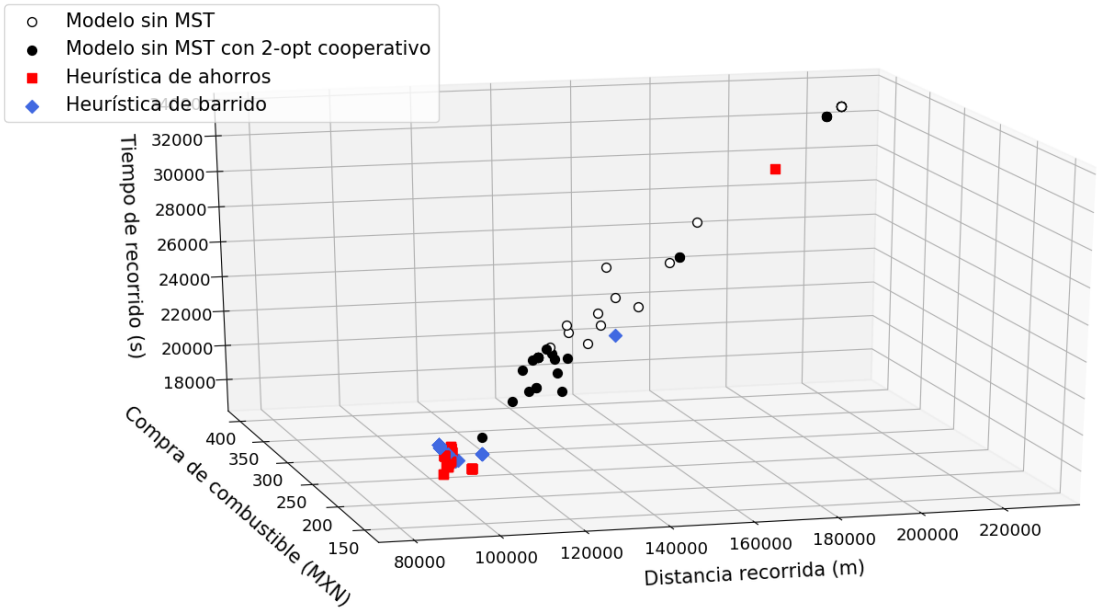


Figura 4.17: Instancia 2. Métodos de resolución independientes.

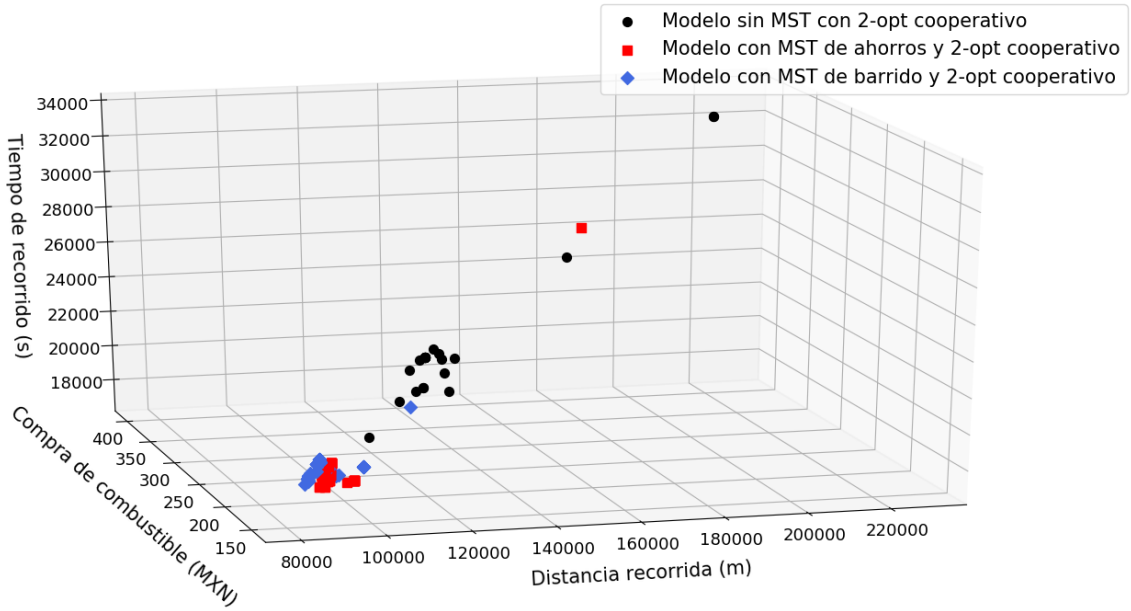


Figura 4.18: Instancia 2. Métodos de resolución combinados.

Cuadro 4.4: Comparación de los métodos de resolución para la instancia 2.

Pesos (d, t, p)	Modelo	Modelo con 2-opt	Ahorros	Barrido	Modelo con MST de ahorros y 2-opt	Modelo con MST de barrido y 2-opt
(1, 1, 1)	-	-	1.06	1.03	1.01	1
(0, 0, 100)	1.44	1.44	1.71	1.25	1.48	1
(0, 10, 90)	-	-	1.03	1.07	1	1.04
(0, 20, 80)	-	-	1.03	1.07	1	1.04
(0, 30, 70)	-	-	1.03	1.07	1	1.04
(0, 40, 60)	1.48	1.24	1.03	1.07	1	1.04
(0, 50, 50)	1.38	1.18	1.03	1.07	1	1.04
(0, 60, 40)	1.58	1.28	1.03	1.07	1	1.04
(0, 70, 30)	-	-	1.03	1.07	1	1.04
(0, 80, 20)	-	-	1.03	1.07	1	1.04
(0, 90, 10)	-	-	1.03	1.07	1	1.04
(0, 100, 0)	-	-	1.03	1.07	1	1.04
(10, 0, 90)	2.63	1.43	1.08	1.06	1.05	1
(10, 10, 80)	-	-	1.06	1.03	1.01	1
(10, 20, 70)	-	-	1.03	1.03	1.01	1
(10, 30, 60)	-	-	1.08	1.09	1.05	1
(10, 40, 50)	2.11	2.11	1.02	1.04	1	1.01
(10, 50, 40)	2.13	2.13	1.05	1.07	1	1.04
(10, 60, 30)	-	-	1.05	1.07	1	1.04
(10, 70, 20)	-	-	1.03	1.06	1	1.03
(10, 80, 10)	1.43	1.34	1.03	1.06	1	1.03
(10, 90, 0)	-	-	1.05	1.09	1	1.05
(20, 0, 80)	2.53	1.38	1.04	1.02	1.01	1
(20, 10, 70)	1.4	1.26	1.06	1.02	1.03	1
(20, 20, 60)	-	-	1.1	1.07	1.05	1
(20, 30, 50)	-	-	1.04	1.03	1	1
(20, 40, 40)	-	-	1.07	1.07	1.05	1
(20, 50, 30)	-	-	1.03	1.03	1	1
(20, 60, 20)	1.53	1.32	1.06	1.07	1.04	1
(20, 70, 10)	1.41	1.25	1.03	1.04	1	1.01
(20, 80, 0)	-	-	1.02	1.04	1	1.01
(30, 0, 70)	2.55	1.39	1.05	1.03	1.02	1
(30, 10, 60)	1.47	1.41	1.06	1.02	1.04	1
(30, 20, 50)	-	-	1.06	1.03	1.03	1
(30, 30, 40)	-	-	1.1	1.07	1.05	1
(30, 40, 30)	-	-	1.04	1.03	1	1
(30, 50, 20)	-	-	1.09	1.07	1.03	1
(30, 60, 10)	-	-	1.07	1.07	1.05	1
(30, 70, 0)	1.44	1.3	1.03	1.03	1.01	1
(40, 0, 60)	2.55	1.39	1.05	1.03	1.02	1
(40, 10, 50)	-	-	1.08	1.04	1.03	1
(40, 20, 40)	1.65	1.35	1.09	1.06	1.07	1
(40, 30, 30)	-	-	1.06	1.03	1.02	1
(40, 40, 20)	1.66	1.38	1.1	1.07	1.05	1
(40, 50, 10)	-	-	1.05	1.04	1	1.01
(40, 60, 0)	-	-	1.06	1.04	1.02	1
(50, 0, 50)	2.61	1.42	1.08	1.06	1.04	1

Continúa en la página siguiente.

Cuadro 4.4 – Continuación de la página anterior.

Pesos (d, t, p)	Modelo	Modelo con 2-opt	Ahorros	Barrido	Modelo con MST de ahorros y 2-opt	Modelo con MST de barrido y 2-opt
(50, 10, 40)	-	-	1.1	1.06	1.08	1
(50, 20, 30)	-	-	1.06	1.02	1.03	1
(50, 30, 20)	-	-	1.07	1.03	1.04	1
(50, 40, 10)	-	-	1.07	1.03	1.02	1
(50, 50, 0)	1.48	1.29	1.07	1.04	1.01	1
(60, 0, 40)	2.55	1.42	1.05	1.03	1.02	1
(60, 10, 30)	-	-	1.08	1.03	1.02	1
(60, 20, 20)	-	-	1.07	1.03	1.05	1
(60, 30, 10)	-	-	1.07	1.03	1.04	1
(60, 40, 0)	-	-	1.07	1.03	1.04	1
(70, 0, 30)	2.63	1.43	1.08	1.06	1.05	1
(70, 10, 20)	1.48	1.18	1.07	1.03	1.05	1
(70, 20, 10)	-	-	1.1	1.06	1.07	1
(70, 30, 0)	-	-	1.09	1.06	1.07	1
(80, 0, 20)	2.55	1.37	1.05	1.03	1.02	1
(80, 10, 10)	-	-	1.07	1.03	1.04	1
(80, 20, 0)	-	-	1.07	1.03	1.02	1
(90, 0, 10)	2.55	1.39	1.05	1.03	1.02	1
(90, 10, 0)	1.47	1.35	1.1	1.05	1.04	1
(100, 0, 0)	1.57	1.31	1.07	1.05	1.04	1
Promedio	**1.88**	**1.38**	**1.06**	**1.04**	**1.02**	**1**

Con respecto a la instancia 2, las figuras 4.11, 4.13, 4.15 y 4.17 muestran los resultados obtenidos por el primer grupo de métodos. En ellas se evidencia un mejor desempeño del solucionador cuando se asiste con la implementación cooperativa de 2-opt. Para esta instancia hubo combinaciones de pesos de las funciones objetivo para las que el solucionador no pudo encontrar una solución factible en el tiempo establecido, lo cual es contraintuitivo ya que el conjunto de restricciones siempre es el mismo y es independiente de la función objetivo. La implementación cooperativa del solucionador con 2-opt no resuelve esta situación debido a que esta heurística sólo se ejecuta cuando el solucionador encuentra una solución factible nueva. Las heurísticas glotona y de barrido encuentran una solución factible en todos los casos y sus soluciones son visiblemente mejores que las encontradas por el solucionador entero. Las figuras 4.12, 4.14, 4.16 y 4.18 son muy similares a las figuras anteriores. Al comparar los resultados de ambos grupos, se aprecia una mejora en los resultados obtenidos por las implementaciones cooperativas con soluciones iniciales heurísticas, en comparación con los resultados obtenidos por los métodos heurísticos individuales. También ver el cuadro 4.4.

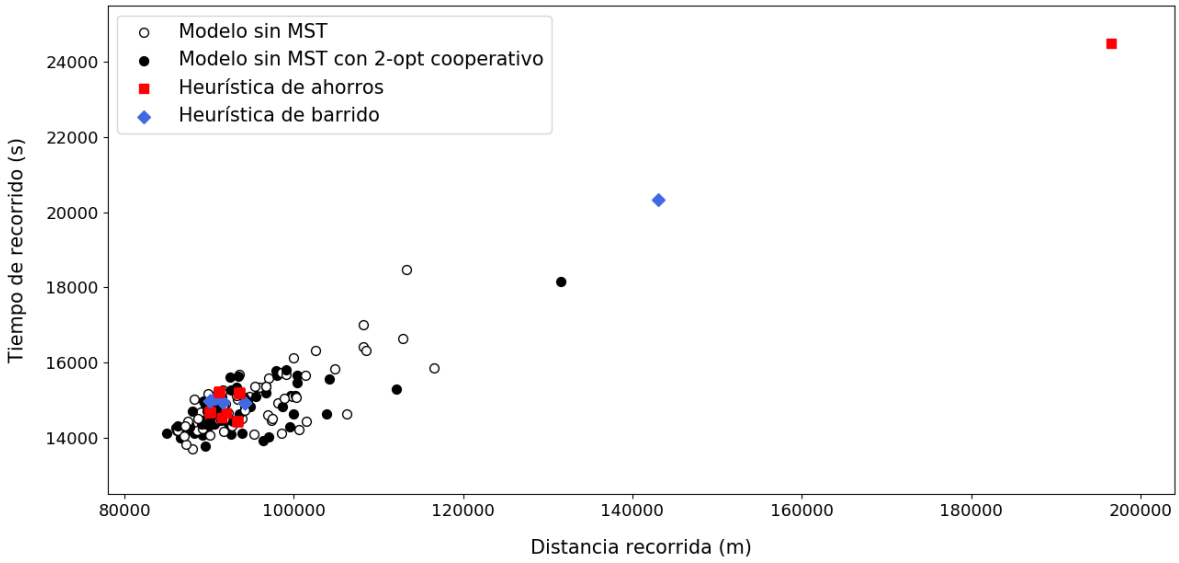


Figura 4.19: Instancia 3. Métodos de resolución independientes.

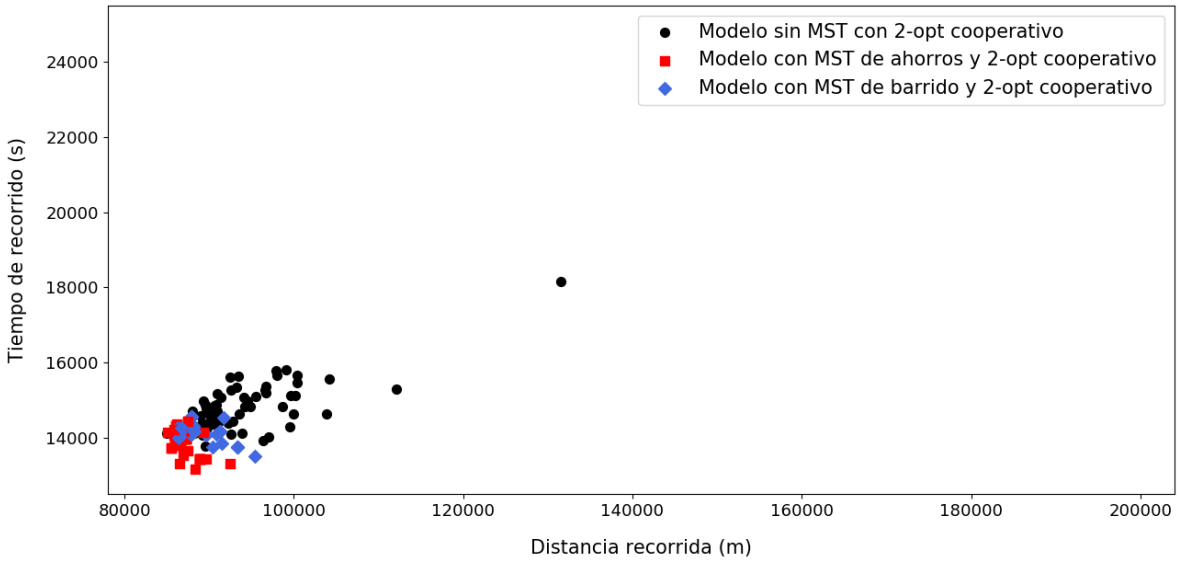


Figura 4.20: Instancia 3. Métodos de resolución combinados.

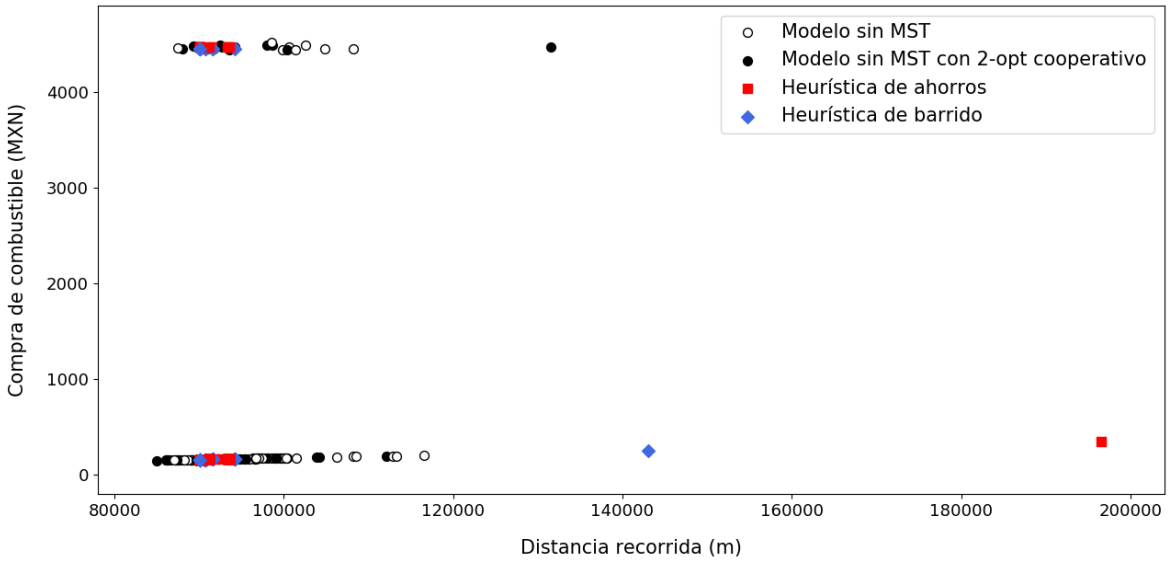


Figura 4.21: Instancia 3. Métodos de resolución independientes.

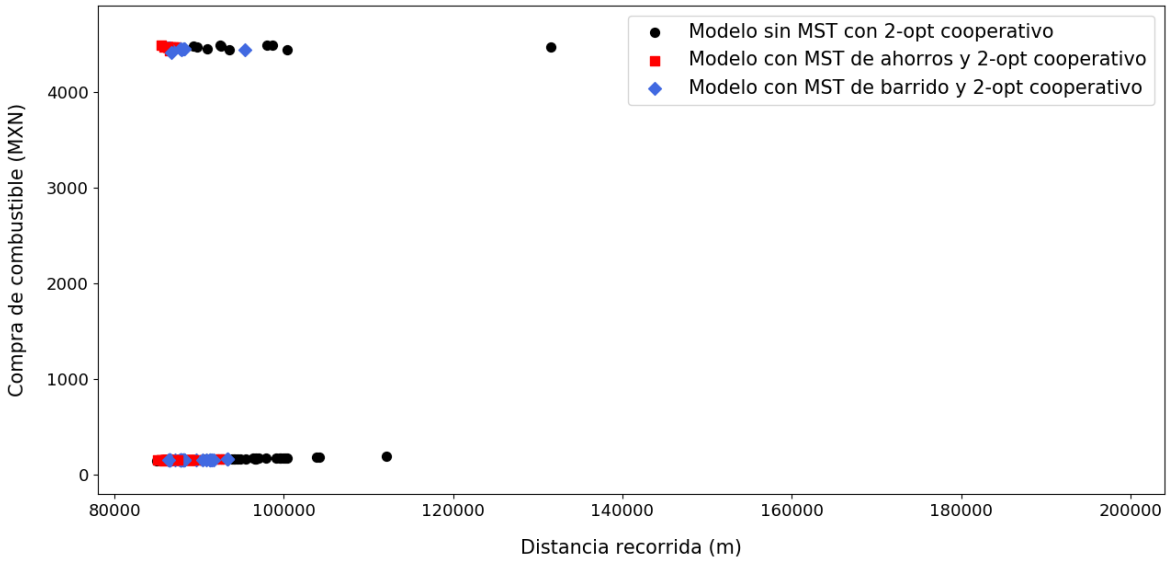


Figura 4.22: Instancia 3. Métodos de resolución combinados.

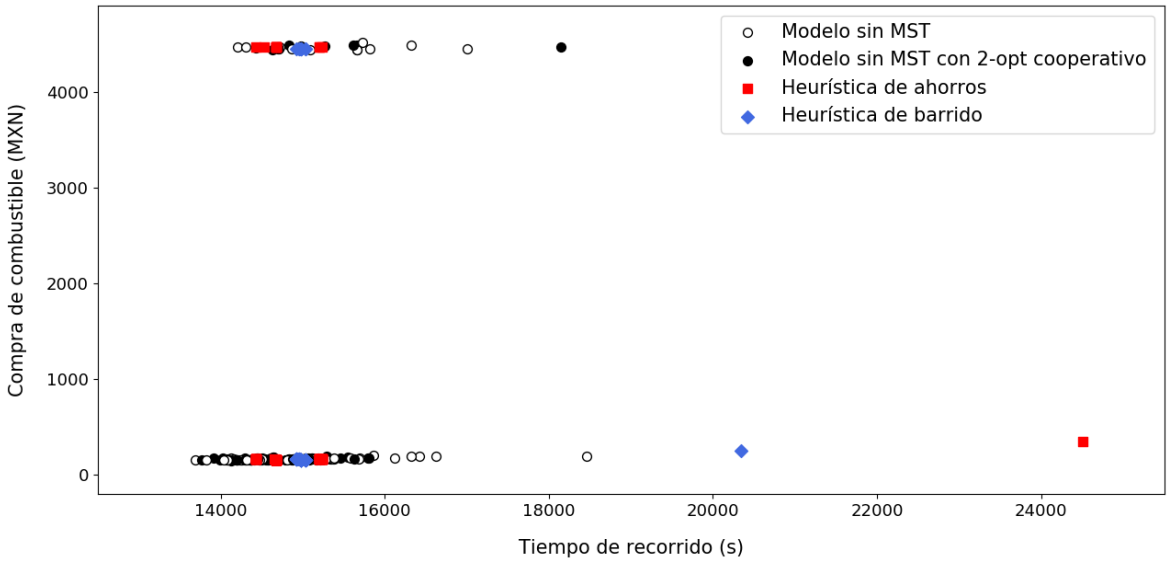


Figura 4.23: Instancia 3. Métodos de resolución independientes.

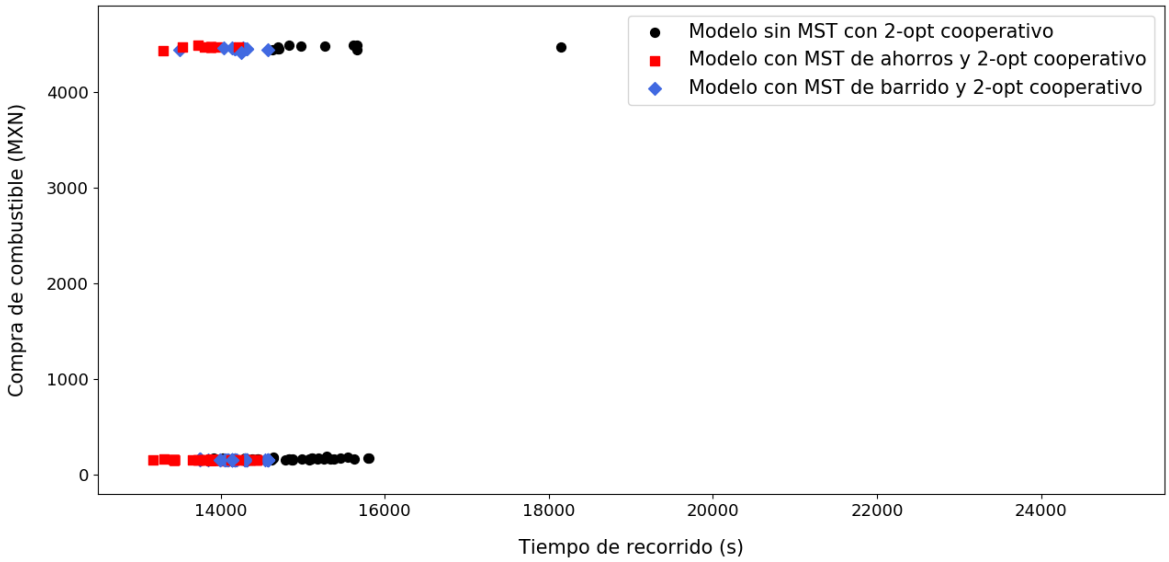


Figura 4.24: Instancia 3. Métodos de resolución combinados.

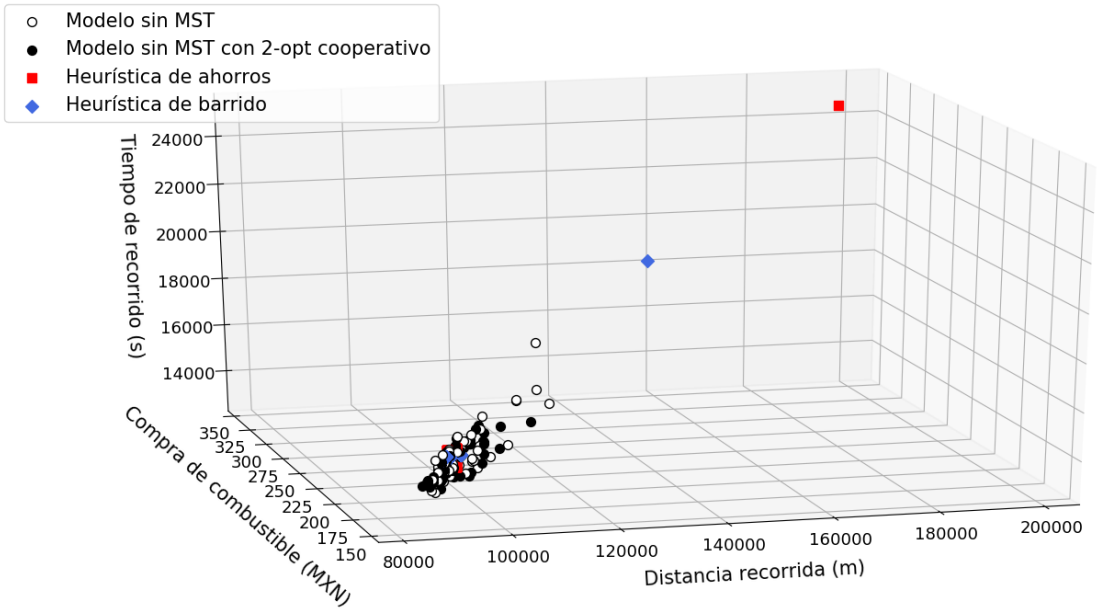


Figura 4.25: Instancia 3. Métodos de resolución independientes.

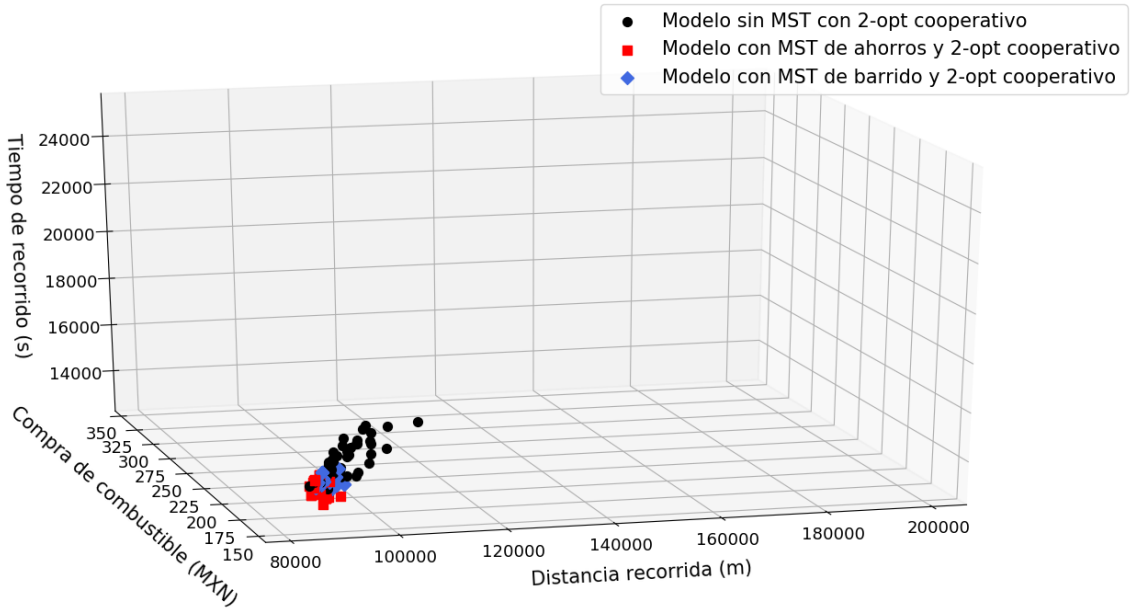


Figura 4.26: Instancia 3. Métodos de resolución combinados.

Cuadro 4.5: Comparación de los métodos de resolución para la instancia 3.

Pesos (d, t, p)	Modelo	Modelo con 2-opt	Ahorros	Barrido	Modelo con MST de ahorros y 2-opt	Modelo con MST de barrido y 2-opt
(1, 1, 1)	1.03	1.02	1.05	1.05	1	1.02
(0, 0, 100)	1.09	1.09	2.23	1.63	1	1.03
(0, 10, 90)	1.24	1.12	1.07	1.1	1	1.03
(0, 20, 80)	1.05	1.05	1.07	1.11	1	1.02
(0, 30, 70)	1.19	1.1	1.08	1.12	1	1.06
(0, 40, 60)	1.09	1.06	1.07	1.11	1	1.02
(0, 50, 50)	1.08	1.09	1.07	1.11	1	1.05
(0, 60, 40)	1.1	1.16	1.1	1.13	1	1.07
(0, 70, 30)	1.09	1.05	1.07	1.11	1	1.05
(0, 80, 20)	1.11	1.16	1.07	1.11	1	1.03
(0, 90, 10)	1.05	1.04	1.07	1.11	1	1.05
(0, 100, 0)	1.07	1.36	1.09	1.12	1	1.02
(10, 0, 90)	1.18	1	1.07	1.06	1.03	1.04
(10, 10, 80)	1.03	1	1.05	1.05	1	1
(10, 20, 70)	1.04	1.04	1.09	1.06	1	1.02
(10, 30, 60)	1	1.13	1.08	1.05	1.01	1.01
(10, 40, 50)	1.04	1.11	1.08	1.06	1	1.01
(10, 50, 40)	1.11	1.14	1.09	1.07	1	1.02
(10, 60, 30)	1.07	1.02	1.08	1.06	1	1.01
(10, 70, 20)	1.06	1	1.08	1.06	1	1.02
(10, 80, 10)	1.11	1.04	1.06	1.07	1	1.02
(10, 90, 0)	1.13	1.11	1.06	1.08	1	1.03
(20, 0, 80)	1.06	1	1.06	1.05	1.02	1.02
(20, 10, 70)	1.11	1.03	1.04	1.05	1	1.02
(20, 20, 60)	1.05	1.04	1.05	1.05	1	1
(20, 30, 50)	1.1	1.09	1.05	1.06	1	1.02
(20, 40, 40)	1.14	1.09	1.09	1.06	1	1.02
(20, 50, 30)	1.08	1.06	1.09	1.06	1	1.02
(20, 60, 20)	1.11	1.14	1.08	1.05	1	1.01
(20, 70, 10)	1.1	1.08	1.09	1.06	1	1.02
(20, 80, 0)	1.19	1.15	1.09	1.07	1	1.02
(30, 0, 70)	1.08	1.04	1.06	1.05	1	1.01
(30, 10, 60)	1.01	1.04	1.04	1.04	1	1
(30, 20, 50)	1.25	1.02	1.04	1.05	1	1
(30, 30, 40)	1.12	1.03	1.05	1.05	1	1
(30, 40, 30)	1.06	1.1	1.05	1.06	1	1.02
(30, 50, 20)	1.03	1.04	1.06	1.07	1	1.02
(30, 60, 10)	1.3	1.1	1.08	1.05	1	1.01
(30, 70, 0)	1.05	1.12	1.08	1.05	1	1.01
(40, 0, 60)	1.15	1.01	1.05	1.04	1.01	1
(40, 10, 50)	1.07	1.04	1.04	1.04	1	1.02
(40, 20, 40)	1.04	1.03	1.04	1.05	1	1.02
(40, 30, 30)	1.03	1.05	1.04	1.05	1	1.02
(40, 40, 20)	1.06	1.04	1.05	1.05	1	1
(40, 50, 10)	1.03	1.09	1.05	1.05	1	1.01
(40, 60, 0)	1.14	1.05	1.05	1.06	1	1.02
(50, 0, 50)	1.05	1	1.06	1.05	1	1.01

Continúa en la página siguiente.

Cuadro 4.5 – Continuación de la página anterior.

Pesos (d, t, p)	Modelo	Modelo con 2-opt	Ahorros	Barrido	Modelo con MST de ahorros y 2-opt	Modelo con MST de barrido y 2-opt
(50, 10, 40)	1	1.02	1.04	1.04	1	1.02
(50, 20, 30)	1.05	1.09	1.04	1.05	1	1.02
(50, 30, 20)	1.03	1.05	1.04	1.05	1	1.02
(50, 40, 10)	1.04	1.06	1.05	1.05	1	1.01
(50, 50, 0)	1.17	1.08	1.05	1.05	1	1
(60, 0, 40)	1.04	1.05	1.06	1.04	1	1.02
(60, 10, 30)	1.08	1.08	1.04	1.04	1	1.02
(60, 20, 20)	1.07	1.06	1.05	1.05	1	1.02
(60, 30, 10)	1.03	1.04	1.04	1.05	1	1.02
(60, 40, 0)	1.1	1.03	1.05	1.06	1	1.01
(70, 0, 30)	1.06	1.02	1.06	1.05	1	1.01
(70, 10, 20)	1.06	1.15	1.04	1.04	1	1.02
(70, 20, 10)	1.06	1.05	1.04	1.04	1	1
(70, 30, 0)	1.19	1.08	1.04	1.05	1	1.02
(80, 0, 20)	1.27	1.09	1.07	1.06	1	1.02
(80, 10, 10)	1.02	1.07	1.05	1.05	1	1.03
(80, 20, 0)	1.01	1.07	1.04	1.04	1	1
(90, 0, 10)	1.01	1.05	1.06	1.04	1.01	1
(90, 10, 0)	1.25	1.04	1.04	1.04	1	1.02
(100, 0, 0)	1.07	1.04	1.06	1.05	1	1.01
Promedio	**1.09**	**1.07**	**1.08**	**1.07**	**1**	**1.02**

Con respecto a la instancia 3, las figuras 4.19, 4.21, 4.23 y 4.25 muestran que el desempeño del solucionador mejora levemente cuando es asistido por 2-opt. Aunque algunos resultados obtenidos por el modelo superan a los métodos heurísticos, el desempeño de estos últimos es bastante bueno considerando la diferencia en el tiempo de cómputo. En las figuras 4.20, 4.22, 4.24 y 4.26 es evidente que las mejores soluciones también corresponden con las calculadas por los métodos cooperativos que utilizan las soluciones heurísticas como soluciones iniciales. También se puede observar que el método cooperativo encuentra mejores soluciones cuando parte de soluciones calculadas por el algoritmo de ahorros. También ver el cuadro 4.5.

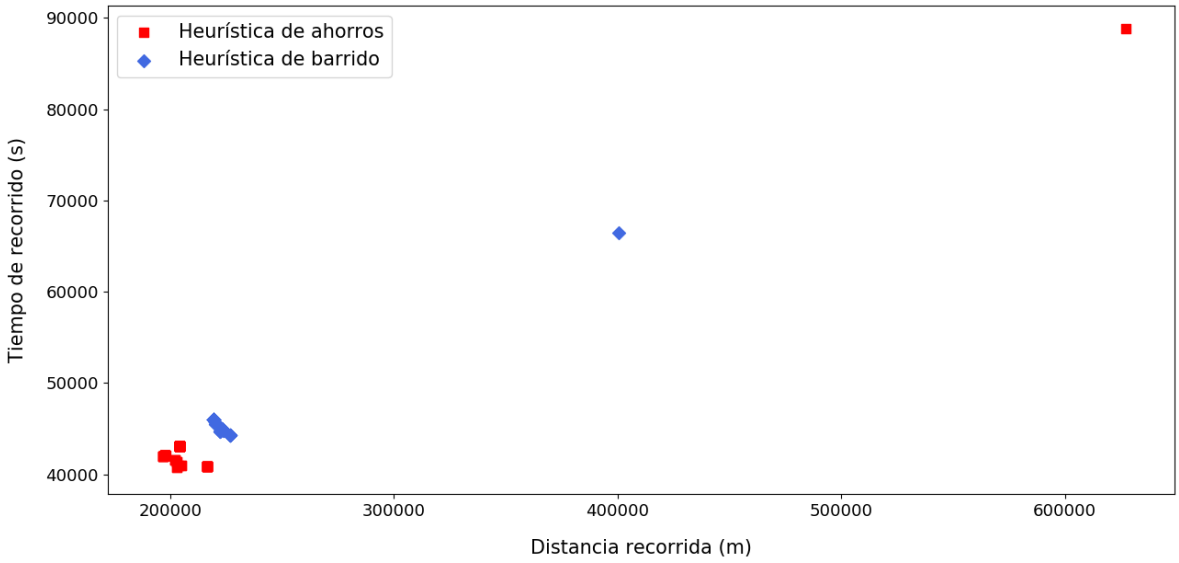


Figura 4.27: Instancia 4. Métodos de resolución independientes.

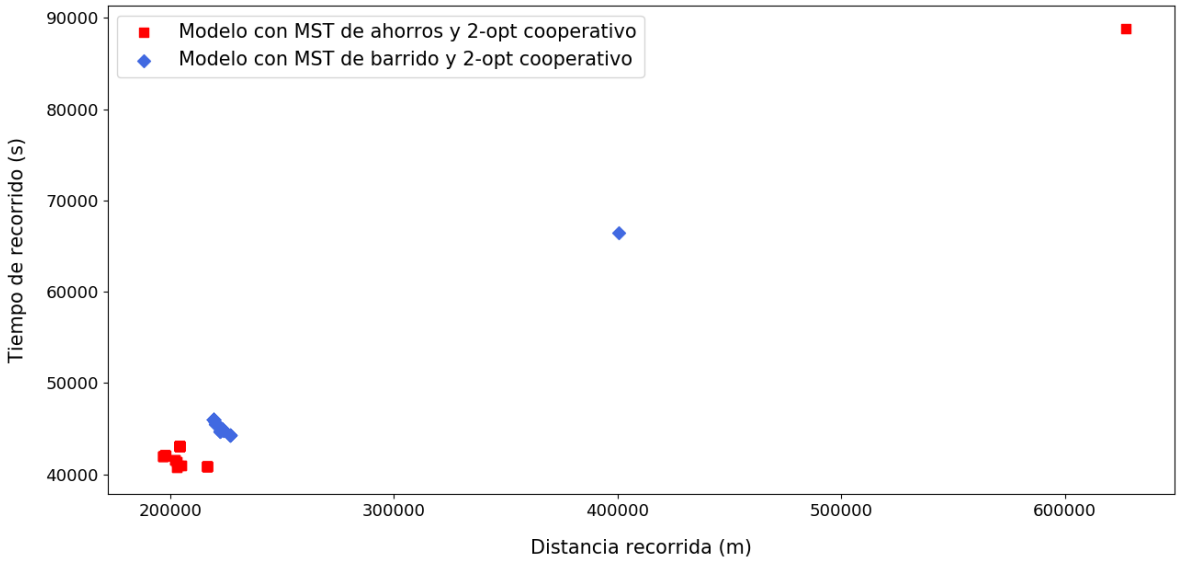


Figura 4.28: Instancia 4. Métodos de resolución combinados.

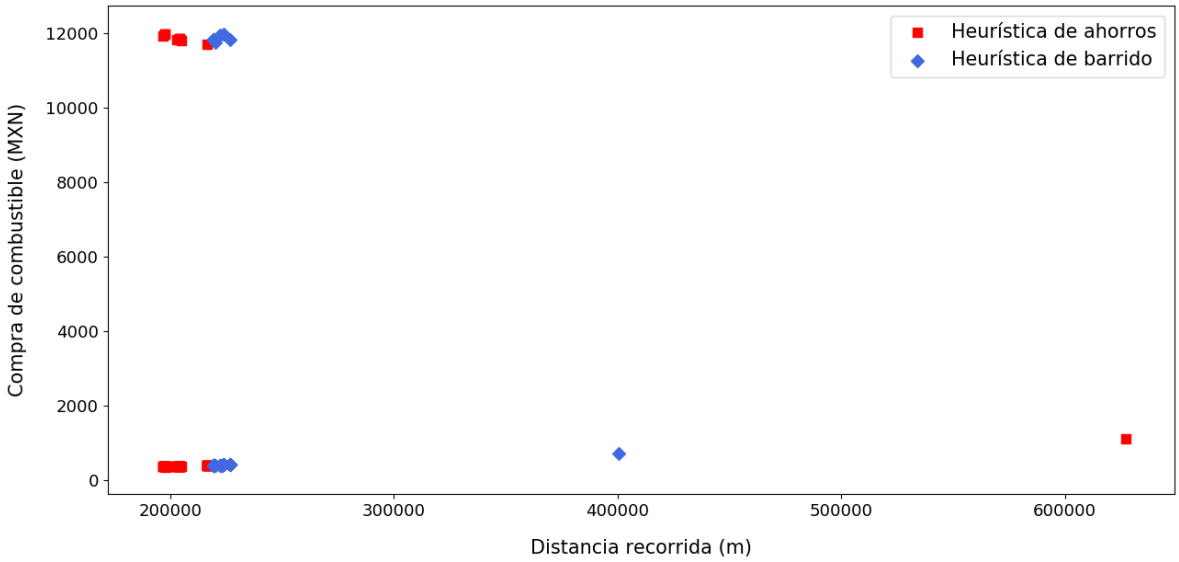


Figura 4.29: Instancia 4. Métodos de resolución independientes.

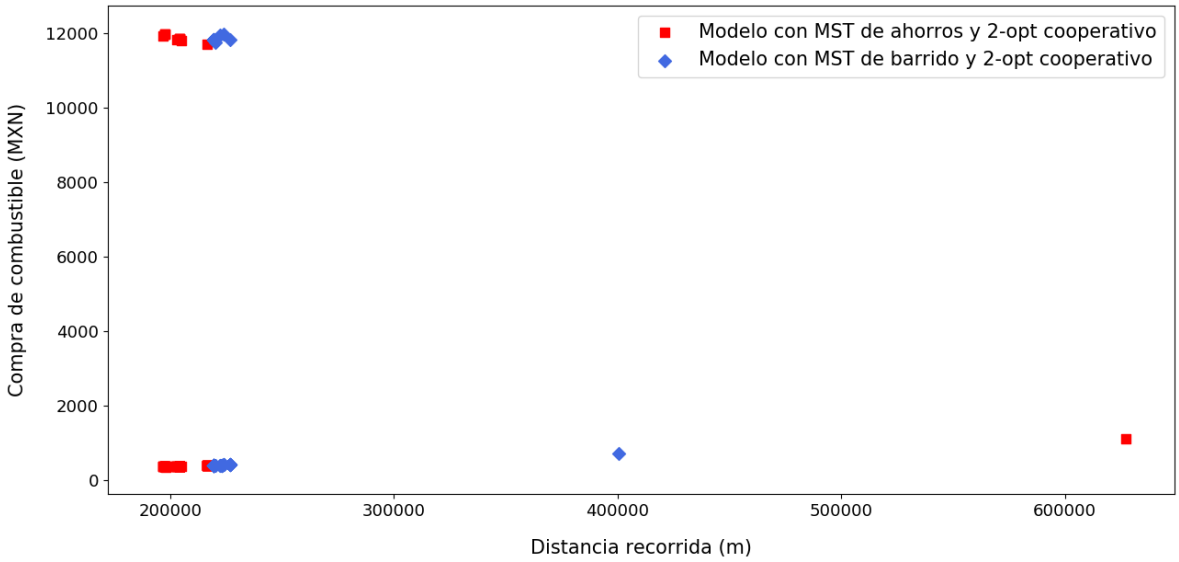


Figura 4.30: Instancia 4. Métodos de resolución combinados.

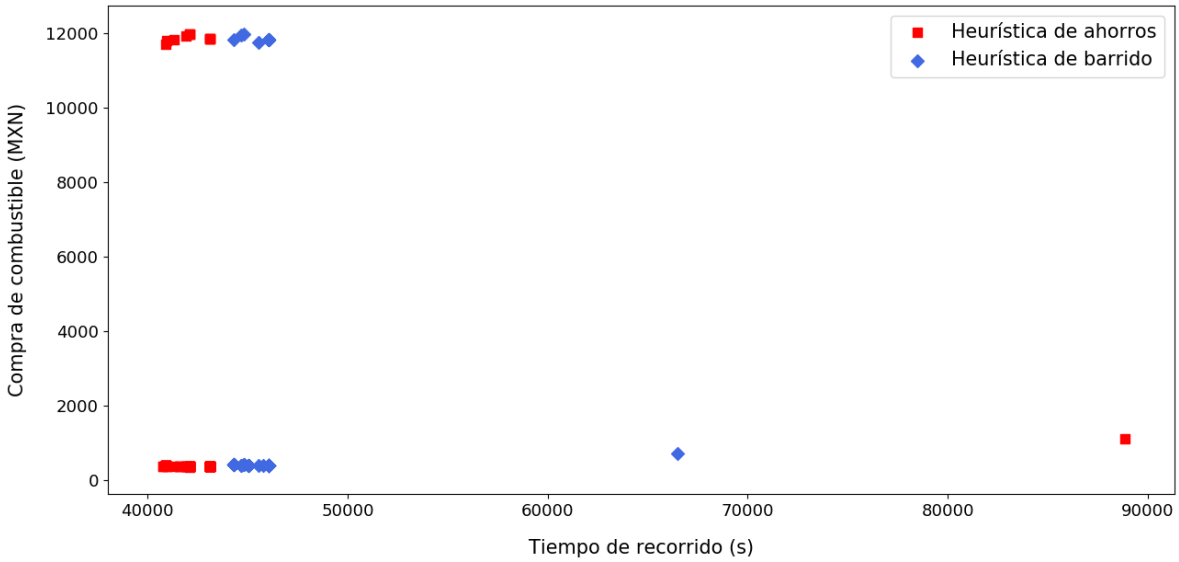


Figura 4.31: Instancia 4. Métodos de resolución independientes.

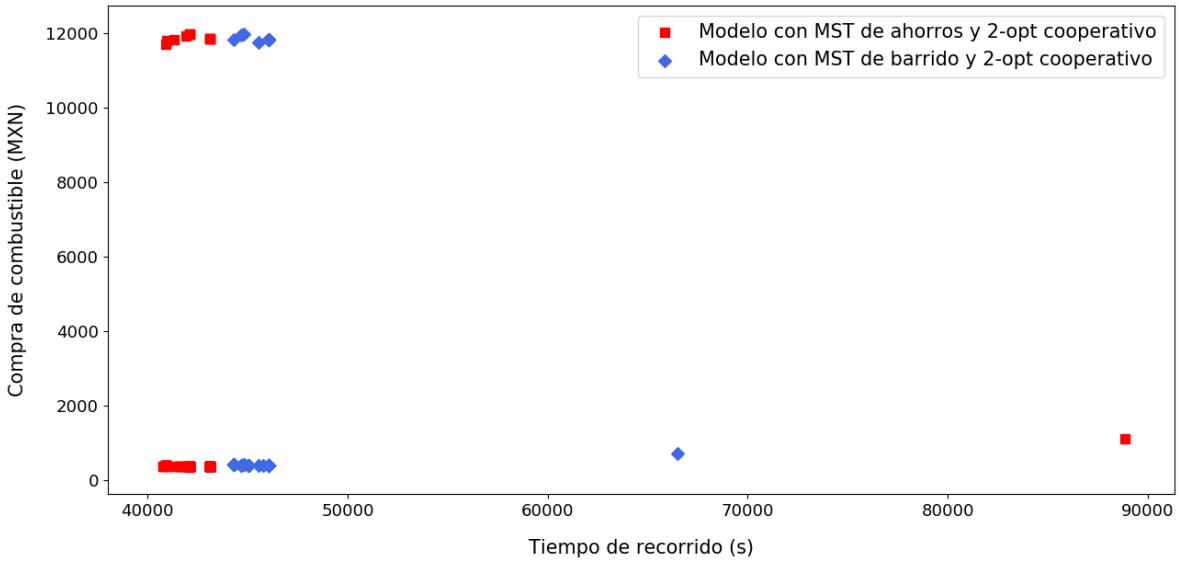


Figura 4.32: Instancia 4. Métodos de resolución combinados.

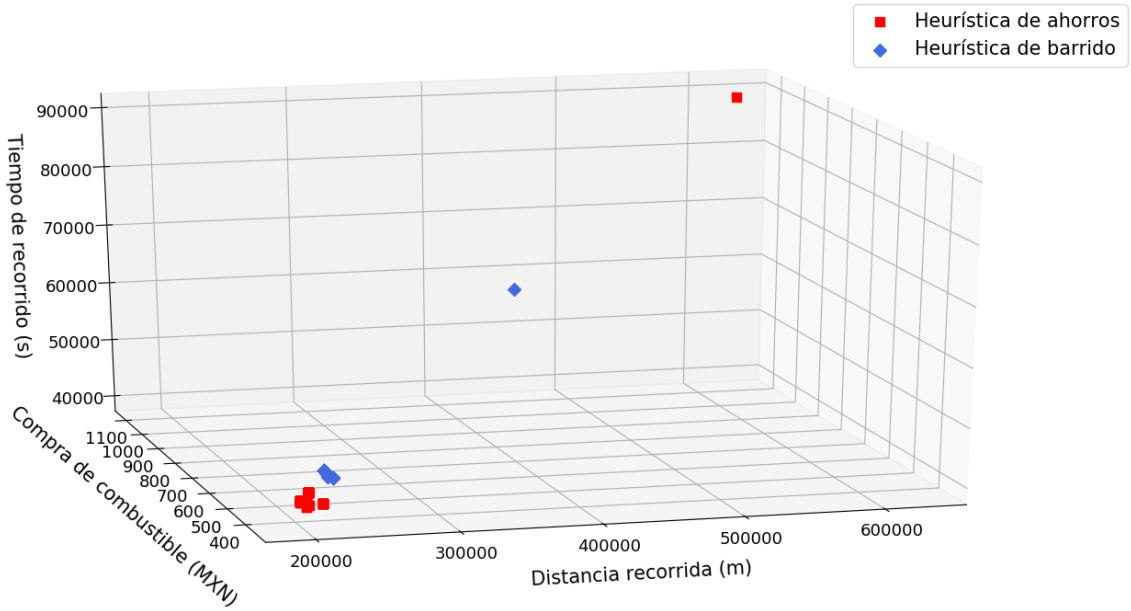


Figura 4.33: Instancia 4. Métodos de resolución independientes.

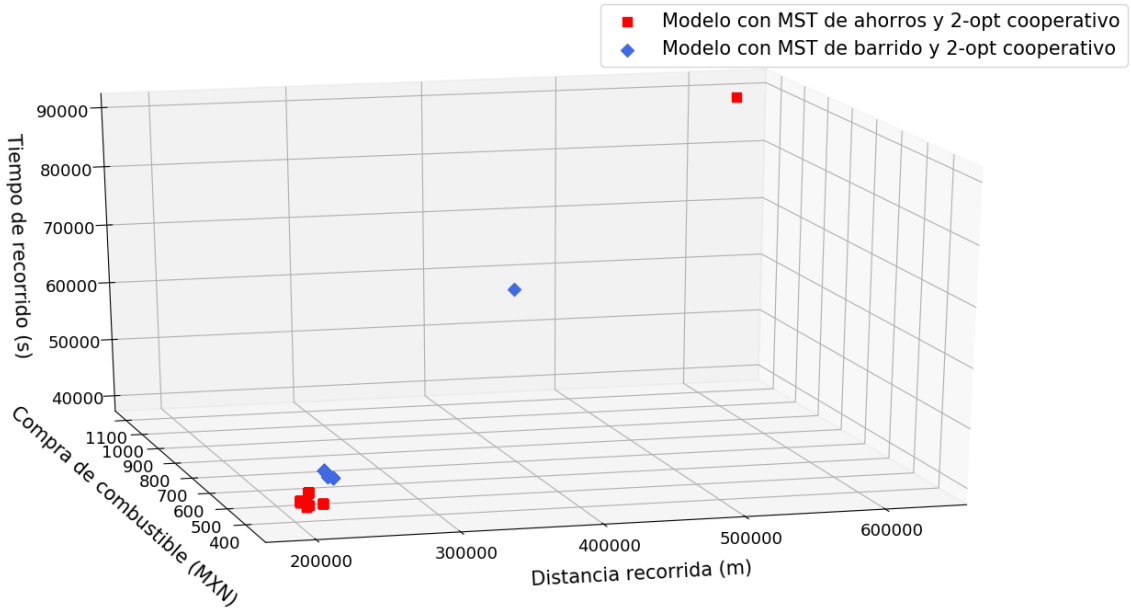


Figura 4.34: Instancia 4. Métodos de resolución combinados.

Cuadro 4.6: Comparación de los métodos de resolución para la instancia 4.

Pesos (d, t, p)	Modelo	Modelo con 2-opt	Ahorros	Barrido	Modelo con MST de ahorros y 2-opt	Modelo con MST de barrido y 2-opt
(1, 1, 1)	-	-	1	1.11	1	1.11
(0, 0, 100)	-	-	1.57	1	1.57	1
(0, 10, 90)	-	-	1	1.08	1	1.08
(0, 20, 80)	-	-	1	1.08	1	1.08
(0, 30, 70)	-	-	1	1.08	1	1.08
(0, 40, 60)	-	-	1	1.08	1	1.08
(0, 50, 50)	-	-	1	1.08	1	1.08
(0, 60, 40)	-	-	1	1.08	1	1.08
(0, 70, 30)	-	-	1	1.08	1	1.08
(0, 80, 20)	-	-	1	1.08	1	1.08
(0, 90, 10)	-	-	1	1.08	1	1.08
(0, 100, 0)	-	-	1	1.08	1	1.08
(10, 0, 90)	-	-	1	1.07	1	1.07
(10, 10, 80)	-	-	1	1.11	1	1.11
(10, 20, 70)	-	-	1	1.12	1	1.12
(10, 30, 60)	-	-	1	1.1	1	1.1
(10, 40, 50)	-	-	1	1.09	1	1.09
(10, 50, 40)	-	-	1	1.1	1	1.1
(10, 60, 30)	-	-	1	1.1	1	1.1
(10, 70, 20)	-	-	1	1.1	1	1.1
(10, 80, 10)	-	-	1	1.09	1	1.09
(10, 90, 0)	-	-	1	1.09	1	1.09
(20, 0, 80)	-	-	1	1.07	1	1.07
(20, 10, 70)	-	-	1	1.07	1	1.07
(20, 20, 60)	-	-	1	1.11	1	1.11
(20, 30, 50)	-	-	1	1.1	1	1.1
(20, 40, 40)	-	-	1	1.12	1	1.12
(20, 50, 30)	-	-	1	1.1	1	1.1
(20, 60, 20)	-	-	1	1.1	1	1.1
(20, 70, 10)	-	-	1	1.09	1	1.09
(20, 80, 0)	-	-	1	1.09	1	1.09
(30, 0, 70)	-	-	1	1.07	1	1.07
(30, 10, 60)	-	-	1	1.07	1	1.07
(30, 20, 50)	-	-	1	1.11	1	1.11
(30, 30, 40)	-	-	1	1.11	1	1.11
(30, 40, 30)	-	-	1	1.11	1	1.11
(30, 50, 20)	-	-	1	1.11	1	1.11
(30, 60, 10)	-	-	1	1.12	1	1.12
(30, 70, 0)	-	-	1	1.11	1	1.11
(40, 0, 60)	-	-	1	1.07	1	1.07
(40, 10, 50)	-	-	1	1.07	1	1.07
(40, 20, 40)	-	-	1	1.07	1	1.07
(40, 30, 30)	-	-	1	1.11	1	1.11
(40, 40, 20)	-	-	1	1.11	1	1.11
(40, 50, 10)	-	-	1	1.11	1	1.11
(40, 60, 0)	-	-	1	1.11	1	1.11
(50, 0, 50)	-	-	1	1.07	1	1.07

Continúa en la página siguiente.

Cuadro 4.6 – Continuación de la página anterior.

Pesos (d, t, p)	Modelo	Modelo con 2-opt	Ahorros	Barrido	Modelo con MST de ahorros y 2-opt	Modelo con MST de barrido y 2-opt
(50, 10, 40)	-	-	1	1.07	1	1.07
(50, 20, 30)	-	-	1	1.07	1	1.07
(50, 30, 20)	-	-	1	1.11	1	1.11
(50, 40, 10)	-	-	1	1.11	1	1.11
(50, 50, 0)	-	-	1	1.11	1	1.11
(60, 0, 40)	-	-	1	1.07	1	1.07
(60, 10, 30)	-	-	1	1.07	1	1.07
(60, 20, 20)	-	-	1	1.07	1	1.07
(60, 30, 10)	-	-	1	1.07	1	1.07
(60, 40, 0)	-	-	1	1.11	1	1.11
(70, 0, 30)	-	-	1	1.07	1	1.07
(70, 10, 20)	-	-	1	1.07	1	1.07
(70, 20, 10)	-	-	1	1.07	1	1.07
(70, 30, 0)	-	-	1	1.07	1	1.07
(80, 0, 20)	-	-	1	1.07	1	1.07
(80, 10, 10)	-	-	1	1.07	1	1.07
(80, 20, 0)	-	-	1	1.07	1	1.07
(90, 0, 10)	-	-	1	1.07	1	1.07
(90, 10, 0)	-	-	1	1.07	1	1.07
(100, 0, 0)	-	-	1	1.07	1	1.07
Promedio	**-**	**-**	**1**	**1.08**	**1**	**1.08**

La instancia 4 es la instancia con el mayor número de vértices. Para ella, el solucionador no logró encontrar soluciones factibles para ninguna de las combinaciones de los pesos de las funciones objetivo dentro del tiempo de cómputo establecido. Por otra parte, las figuras 4.27, 4.29, 4.31 y 4.33 muestran que las heurísticas propuestas encontraron soluciones factibles en todos los casos. El tamaño de la instancia afectó principalmente el tiempo de ejecución de la heurística de barrido, la cual se ejecuta en fracciones de segundo en las demás instancias pero tarda alrededor de cuatro segundos en calcular una solución para esta instancia. En las figuras 4.28, 4.30, 4.32 y 4.34 se muestra que el solucionador en cooperación con 2-opt tampoco logra mejorar las soluciones iniciales heurísticas dentro del tiempo de cómputo establecido. Por esta razón, las gráficas correspondientes a ambos grupos son idénticas. Los mejores resultados fueron calculados por el algoritmo de ahorros. También ver el cuadro 4.6.

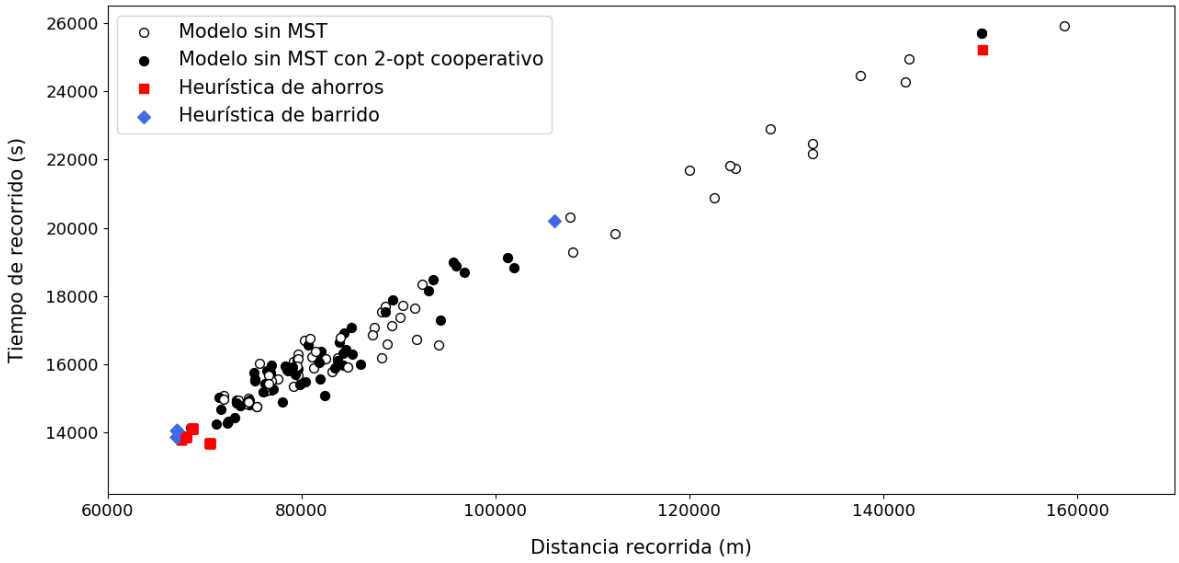


Figura 4.35: Instancia 5. Métodos de resolución independientes.

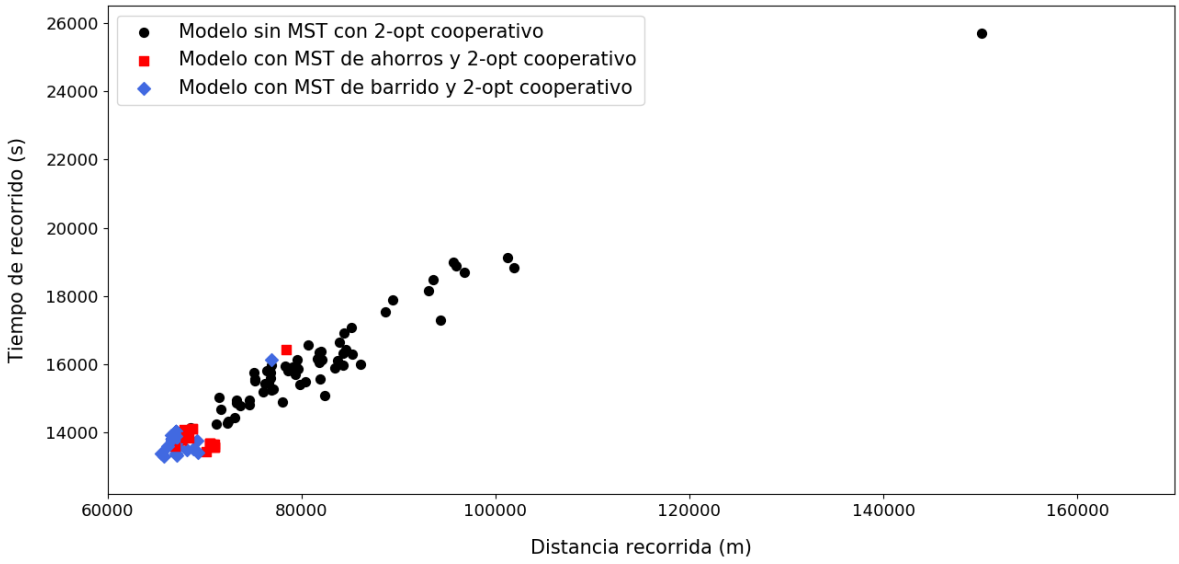


Figura 4.36: Instancia 5. Métodos de resolución combinados.

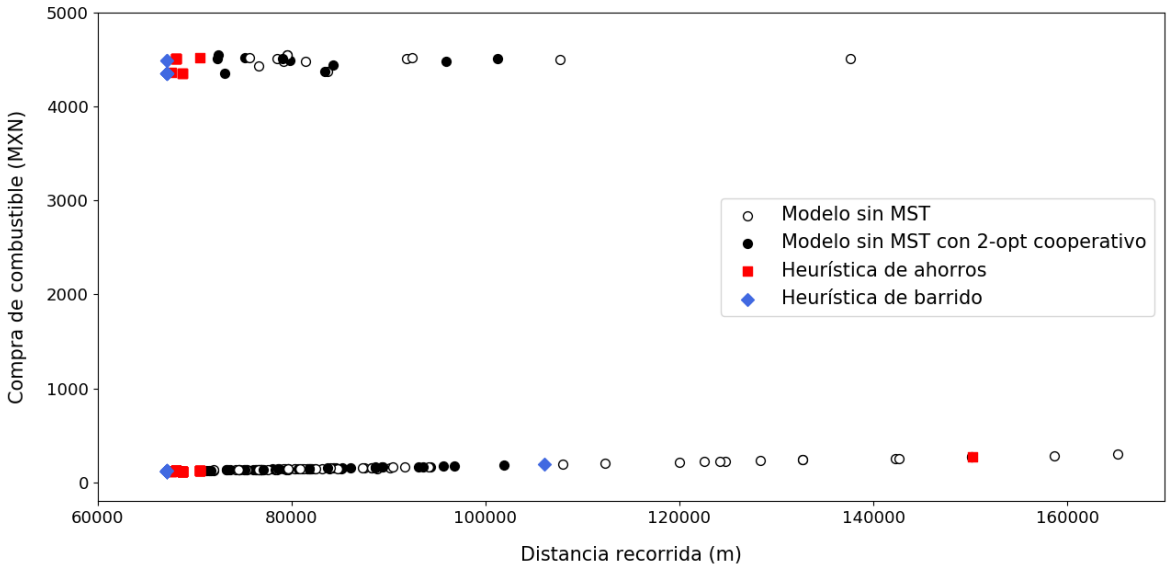


Figura 4.37: Instancia 5. Métodos de resolución independientes.

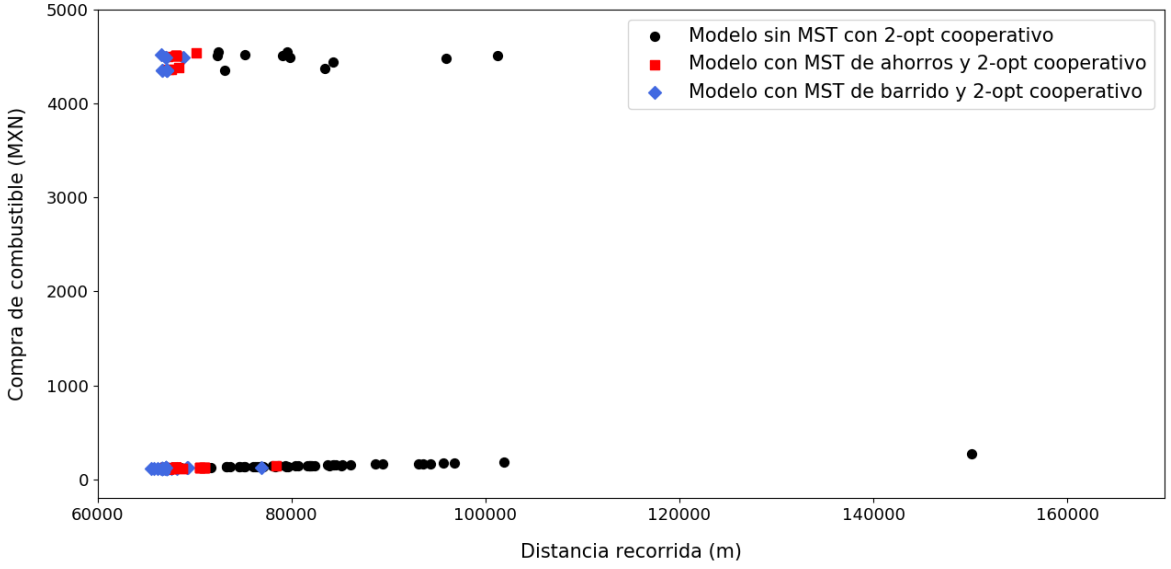


Figura 4.38: Instancia 5. Métodos de resolución combinados.

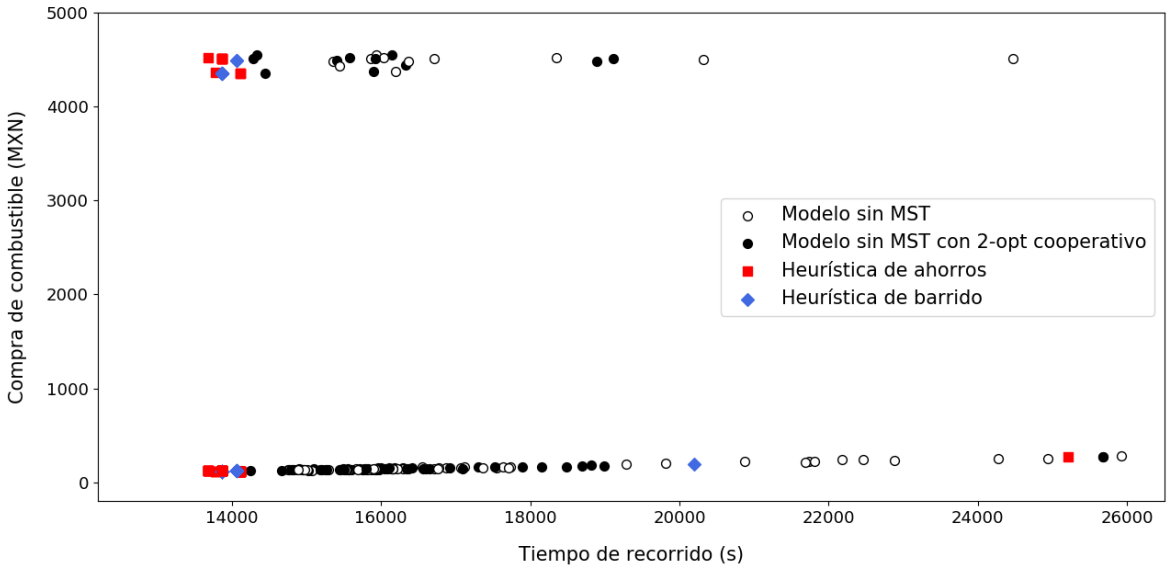


Figura 4.39: Instancia 5. Métodos de resolución independientes.

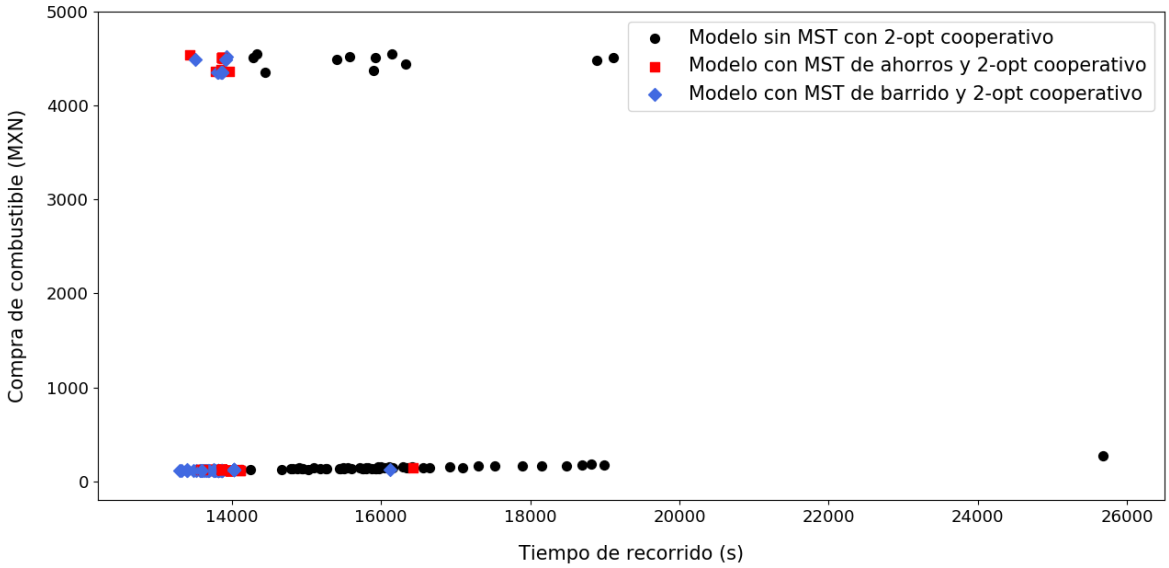


Figura 4.40: Instancia 5. Métodos de resolución combinados.

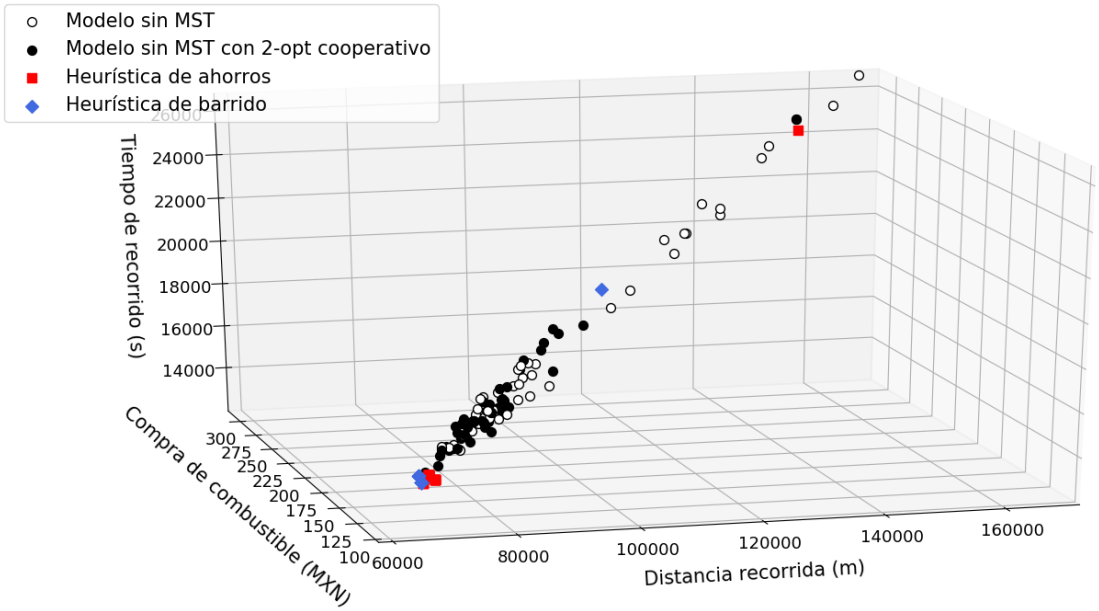


Figura 4.41: Instancia 5. Métodos de resolución independientes.

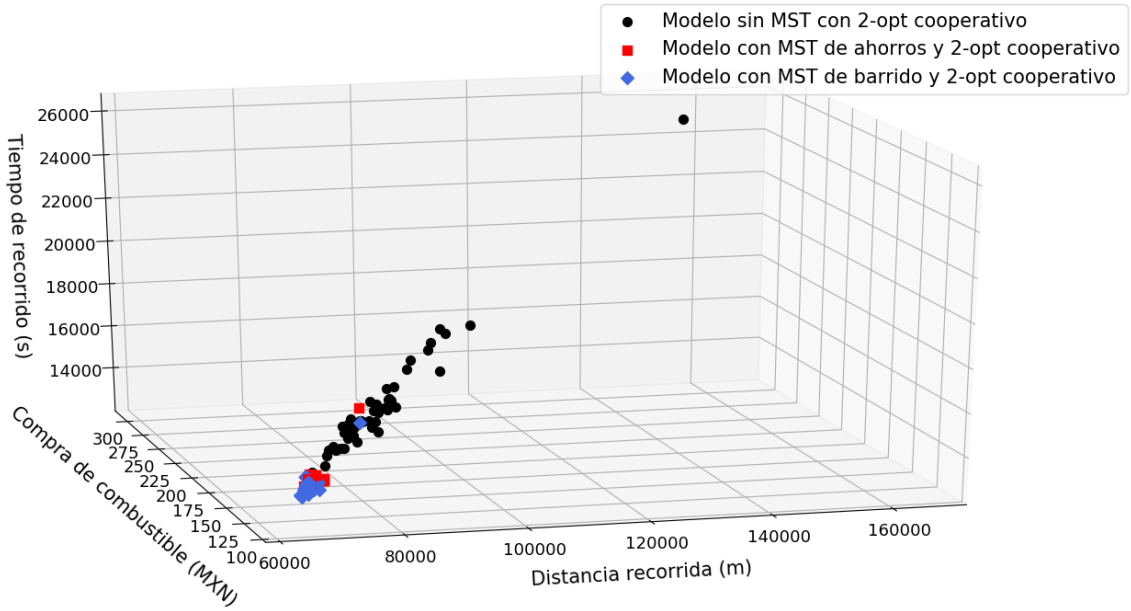


Figura 4.42: Instancia 5. Métodos de resolución combinados.

Cuadro 4.7: Comparación de los métodos de resolución para la instancia 5.

Pesos (d, t, p)	Modelo	Modelo con 2-opt	Ahorros	Barrido	Modelo con MST de ahorros y 2-opt	Modelo con MST de barrido y 2-opt
(1, 1, 1)	1.11	1.07	1.02	1.01	1.02	1
(0, 0, 100)	1.11	1.11	2.1	1.48	1.1	1
(0, 10, 90)	-	-	1.03	1.03	1.02	1
(0, 20, 80)	1.19	1.17	1	1.01	1	1
(0, 30, 70)	1.25	1.22	1.03	1.04	1.03	1
(0, 40, 60)	-	-	1.01	1.03	1.01	1
(0, 50, 50)	1.27	1.11	1.02	1.03	1.02	1
(0, 60, 40)	1.57	1.16	1.03	1.04	1.03	1
(0, 70, 30)	1.22	1.11	1.01	1.02	1	1.01
(0, 80, 20)	1.16	1.18	1.01	1.02	1	1.01
(0, 90, 10)	1.47	1.28	1.01	1.03	1.01	1
(0, 100, 0)	1.24	1.07	1.02	1.03	1	1.01
(10, 0, 90)	1.11	1.17	1.02	1	1.02	1
(10, 10, 80)	1.08	1.07	1.02	1	1.02	1
(10, 20, 70)	1.32	1.09	1.03	1.01	1.01	1
(10, 30, 60)	1.18	1.15	1.03	1.01	1.03	1
(10, 40, 50)	1.17	1.17	1.03	1.01	1.01	1
(10, 50, 40)	2.14	1.38	1.03	1.01	1.02	1
(10, 60, 30)	1.7	1.22	1	1	1	1
(10, 70, 20)	2.03	2.03	1.01	1.01	1.01	1
(10, 80, 10)	1.27	1.14	1.01	1.01	1.01	1
(10, 90, 0)	1.2	1.21	1	1	1	1
(20, 0, 80)	1.11	1.06	1.02	1	1.02	1
(20, 10, 70)	1.2	1.18	1.01	1	1.01	1
(20, 20, 60)	1.88	1.43	1.02	1.01	1.02	1
(20, 30, 50)	1.32	1.22	1.03	1.01	1.01	1
(20, 40, 40)	2.02	1.37	1.03	1.01	1.01	1
(20, 50, 30)	1.19	1.14	1.03	1.01	1.01	1
(20, 60, 20)	1.11	1.18	1.03	1.01	1.01	1
(20, 70, 10)	1.15	1.15	1.03	1.01	1.01	1
(20, 80, 0)	1.16	1.07	1.03	1.01	1.01	1
(30, 0, 70)	2.46	1.27	1.02	1	1.02	1
(30, 10, 60)	1.13	1.21	1.01	1	1.01	1
(30, 20, 50)	1.21	1.17	1.02	1.01	1.02	1
(30, 30, 40)	1.1	1.03	1.02	1.01	1.02	1
(30, 40, 30)	1.9	1.25	1.02	1	1.02	1
(30, 50, 20)	1.15	1.15	1.03	1.01	1.01	1
(30, 60, 10)	1.14	1.14	1.03	1.01	1.01	1
(30, 70, 0)	1.56	1.18	1.02	1	1	1
(40, 0, 60)	1.17	1.12	1.02	1	1.02	1
(40, 10, 50)	1.14	1.13	1.01	1	1.01	1
(40, 20, 40)	1.26	1.1	1.02	1.01	1.02	1
(40, 30, 30)	1.35	1.14	1.02	1.01	1	1
(40, 40, 20)	1.23	1.15	1.02	1.01	1.02	1
(40, 50, 10)	1.9	1.21	1.02	1	1	1
(40, 60, 0)	1.16	1.17	1.02	1	1.01	1
(50, 0, 50)	1.85	1.52	1.02	1	1.02	1

Continúa en la página siguiente.

Cuadro 4.7 – Continuación de la página anterior.

Pesos (d, t, p)	Modelo	Modelo con 2-opt	Ahorros	Barrido	Modelo con MST de ahorros y 2-opt	Modelo con MST de barrido y 2-opt
(50, 10, 40)	1.36	1.22	1.01	1	1.01	1
(50, 20, 30)	1.2	1.14	1.03	1.01	1.03	1
(50, 30, 20)	1.31	1.11	1.01	1	1.01	1
(50, 40, 10)	1.11	1.2	1.01	1	1.01	1
(50, 50, 0)	1.37	1.23	1.01	1	1.01	1
(60, 0, 40)	1.25	1.17	1.02	1	1.02	1
(60, 10, 30)	1.14	1.15	1.01	1	1.01	1
(60, 20, 20)	1.82	1.36	1.04	1.03	1.04	1
(60, 30, 10)	1.2	1.24	1.01	1	1.01	1
(60, 40, 0)	2.02	1.49	1.01	1	1.01	1
(70, 0, 30)	1.07	1.22	1.02	1	1.02	1
(70, 10, 20)	1.32	1.32	1.01	1	1.01	1
(70, 20, 10)	1.2	1.1	1.01	1	1.01	1
(70, 30, 0)	1.18	1.12	1.01	1	1.01	1
(80, 0, 20)	1.11	1.07	1.02	1	1.02	1
(80, 10, 10)	1.6	1.42	1.01	1	1.01	1
(80, 20, 0)	1.13	1.18	1.01	1	1.01	1
(90, 0, 10)	2.13	1.25	1.02	1	1.02	1
(90, 10, 0)	1.22	1.44	1.02	1.01	1.02	1
(100, 0, 0)	1.14	1.08	1.02	1	1.02	1
Promedio	**1.36**	**1.2**	**1.03**	**1.01**	**1.01**	**1**

Con respecto a la instancia 5, las figuras 4.35, 4.37, 4.39 y 4.41 muestran que los resultados presentan una mejora cuando el solucionador se ejecuta de forma cooperativa con 2-opt. También se evidencia el dominio de los métodos heurísticos sobre el solucionador sin soluciones iniciales. En las figuras 4.36, 4.38, 4.40 y 4.42 se observa que el solucionador en cooperación con 2-opt y con soluciones iniciales mejora la diversidad y la convergencia de las soluciones heurísticas. En este caso, las soluciones calculadas por la heurística de barrido son mejores en términos de la distancia y las soluciones calculadas por el algoritmo de ahorros son mejores en términos del tiempo de recorrido. Sin embargo las soluciones obtenidas por el solucionador con 2-opt son mejores tanto en distancia recorrida como en tiempo de recorrido cuando parten de una solución calculada por la heurística de barrido. También ver el cuadro 4.7.

Capítulo 5

Conclusiones

En esta tesis presentamos un problema multiobjetivo nuevo en la literatura al que llamamos problema de ruteo de vehículos con abastecimiento de combustible, el cual es de alto interés práctico pero también es computacionalmente muy difícil. Por lo mismo, exploramos diferentes métodos de resolución en los que combinamos novedosas adaptaciones heurísticas en cooperación con la resolución exacta del modelo matemático propuesto. Los resultados experimentales nos muestran que esta metodología está plenamente justificada y que es posible obtener soluciones buenas en un tiempo razonable. En particular, las soluciones obtenidas al resolver el modelo propuesto en combinación con heurísticas son superiores a las soluciones obtenidas por cada método de forma individual. Por un lado, la ejecución cooperativa de 2-opt junto con el solucionador genera soluciones con mejor convergencia que las calculadas por el solucionador solo y en el mismo tiempo de cómputo. Por otro lado, las soluciones obtenidas al usar una solución inicial calculada por las heurísticas adaptadas de VRP son muy superiores a las que no usan soluciones iniciales, esto sin incurrir en afectaciones importantes del tiempo de cómputo. Lo anterior indica que la combinación de los métodos de resolución seleccionados en esta tesis, en efecto aceleran el proceso de resolución del modelo propuesto.

Los métodos heurísticos desarrollados en esta tesis, el algoritmo glotón de ahorros para k vehículos con abastecimiento y la heurística geométrica de barrido para k vehículos con abastecimiento, son capaces de calcular buenas soluciones a un bajo costo computacional a pesar de la dificultad del problema propuesto, incluso en instancias grandes para las que el solucionador no logra encontrar soluciones factibles en el tiempo límite establecido. De hecho, para estas instancias, los métodos heurísticos son los únicos métodos que lograron encontrar soluciones factibles bajo las restricciones de tiempo. Además, las soluciones calculadas por los métodos heurísticos en instancias medianas superan fácilmente a las calculadas por el

solucionador y usan sólo una fracción de los recursos computacionales empleados por este último. En instancias pequeñas, el solucionador logra encontrar mejores soluciones que los métodos heurísticos a costa de un mayor tiempo de cómputo. Sin embargo, las soluciones heurísticas calculadas para estas instancias son cercanas a las soluciones calculadas por el solucionador.

Como trabajo futuro, contemplamos la posibilidad de estudiar una variante del problema con un horizonte de planeación mayor a un día. En el planteamiento actual existe una restricción que obliga a los vehículos a regresar al almacén con al menos la cantidad de combustible inicial, lo que por un lado garantiza que tengan suficiente combustible al comienzo de la siguiente jornada de trabajo, pero ocasiona que todas las rutas visiten una gasolinera. Esto podría resultar en un incremento de la distancia y el tiempo de recorrido debido a una frecuencia de abastecimiento que tal vez no ocurra en la vida real. El modelo propuesto podría modificarse contando con información real y detallada del uso de los vehículos de una empresa de distribución.

Apéndice A

Código fuente de los algoritmos implementados

A.1. Implementación del modelo de programación entera lineal utilizando la interfaz de programación de C++ de Gurobi

```
#include <algorithm>
#include <exception>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <string>
#include "gurobi_c++.h"
#include "matrix.h"

template<typename... T>
std::string formatted_string(const char* formato, const T&...
   variables) {
   char buffer[1024 + 1];
   std::sprintf(buffer, formato, variables...);
   return buffer;
}
```

```
struct cliente {
   int vertice;
   double demanda;
};

struct gasolinera {
   int vertice;
   double costo_metro, tiempo_servicio;
};

struct costo_arco {
   double distancia, tiempo;
};

int main(int argc, const char* argv[])
try {
   if (argc <= 1) {
      throw std::runtime_error(formatted_string("Uso: %s
         instancia", argv[0]));
   }

   std::string instancia = argv[1];
   std::ifstream entrada(instancia);
   if (!entrada.is_open( )) {
      throw std::runtime_error(formatted_string("Error al abrir %
         s", instancia.c_str( )));
   }

   int vertices, vehiculos; double cajuela, tanque, g0, kmpl;
   entrada >> vertices >> vehiculos >> cajuela >> tanque >> g0 >>
       kmpl;
   tanque *= kmpl * 1000, g0 *= kmpl * 1000;   // Convertir
      litros a metros
```

```
int almacen;
std::vector<cliente> clientes;
std::vector<gasolinera> gasolineras;
for (int v = 0; v < vertices; ++v) {
   double x, y, c1, c2; std::string direccion;
   entrada >> x >> y >> c1 >> c2;
   std::getline(entrada >> std::ws, direccion);
   if (c1 == -1 && c2 == -1) {
      almacen = v;
   } else if (c1 == -1) {
      clientes.push_back({ v, c2 });
   } else {
      gasolineras.push_back({ v, c1 / (kmpl * 1000), c2 });
   }
}

matrix<costo_arco, 2> matriz(vertices, vertices);
for (int v1 = 0; v1 < vertices; ++v1) {
   for (int v2 = 0; v2 < vertices; ++v2) {
      entrada >> matriz(v1, v2).distancia;
   }
}
for (int v1 = 0; v1 < vertices; ++v1) {
   for (int v2 = 0; v2 < vertices; ++v2) {
      entrada >> matriz(v1, v2).tiempo;
   }
}

// Construccion del modelo

GRBEnv ambiente;
ambiente.set(GRB_IntParam_Method, 3);
ambiente.set(GRB_IntParam_MIPFocus, 1);
ambiente.set(GRB_IntParam_NumericFocus, 1);
ambiente.set(GRB_IntParam_Presolve, 2);
```

```
ambiente.set(GRB_DoubleParam_TimeLimit, 600);
GRBModel modelo(ambiente);

// ——————————————— Restricciones de la estrategia general de
    espacios ———————————————

// Apartaremos espacio suficiente para que podamos visitar una
    gasolinera en cada ruta, ademas de que cada ruta empiece
   en el almacen
// (el almacen al final de una ruta es el inicio de otra, o
   bien, ademas tenemos un almacen final).
const int espacios = clientes.size( ) + 2 * vehiculos + 1;
const int traslados = espacios - 1;

// Cada espacio de la solucion puede contener cualquier
   vertice.
matrix<GRBVar, 2> vertice_en_espacio(vertices, espacios);
for (int v = 0; v < vertices; ++v) {
   for (int e = 0; e < espacios; ++e) {
      vertice_en_espacio(v, e) = modelo.addVar(0, 1, 0,
         GRB_BINARY, formatted_string("vertice_en_espacio[%d
         ][%d]", v, e));
   }
}

// Solo un vertice por espacio.
for (int e = 0; e < espacios; ++e){
   GRBLinExpr ex;
   for (int v = 0; v < vertices; ++v){
      ex += vertice_en_espacio(v, e);
   }
   modelo.addConstr(ex == 1);
}

// Cada cliente en solo un espacio.
```

```
for (cliente c : clientes) {
   GRBLinExpr ex;
   for (int e = 0; e < espacios; ++e) {
      ex += vertice_en_espacio(c.vertice, e);
   }
   modelo.addConstr(ex == 1);
}

// El primer y ultimo espacio contienen al almacen.
modelo.addConstr(vertice_en_espacio(almacen, 0) == 1);
modelo.addConstr(vertice_en_espacio(almacen, espacios - 1) ==
   1);

// Entre cada pareja de espacios existe el correspondiente
   traslado, que usa un arco. El k-esimo traslado corresponde
   a pasar del vertice en el espacio k al vertice en el
   espacio k+1.
matrix<GRBVar, 3> arco_en_traslado(vertices, vertices,
   traslados);
for (int v1 = 0; v1 < vertices; ++v1) {
   for (int v2 = 0; v2 < vertices; ++v2) {
      for (int t = 0; t < traslados; ++t) {
         arco_en_traslado(v1, v2, t) = modelo.addVar(0, 1, 0,
            GRB_BINARY, formatted_string("arco_en_traslado[%d
            ][%d][%d]", v1, v2, t));
         GRBVar extremos[] = { vertice_en_espacio(v1, t),
            vertice_en_espacio(v2, t + 1) };
         modelo.addGenConstrAnd(arco_en_traslado(v1, v2, t),
            extremos, 2);
      }
   }
}

// Ir de almacen a almacen indica no-actividad; cualquier
   llegada al almacen desde otro vertice es el fin de la ruta
```

```
   de un vehiculo. Tenemos un numero exacto de vehiculos a
   usar.
GRBLinExpr viajes_almacen;
for (int t = 0; t < traslados; ++t) {
   for (int v = 0; v < vertices; ++v){
      if (v != almacen) {
         viajes_almacen += arco_en_traslado(v, almacen, t);
      }
   }
}
modelo.addConstr(viajes_almacen == vehiculos);

// ———————————————— Restricciones de capacidad
   ————————————————

// A cada espacio se le asocia una variable entera del
   producto diponible en el vehiculo en ese punto de la ruta.
matrix<GRBVar, 1> producto_en_espacio(espacios);
for (int e = 0; e < espacios; ++e){
   producto_en_espacio(e) = modelo.addVar(0, cajuela, 0,
      GRB_INTEGER, formatted_string("producto_en_espacio[%d]",
       e));
}
modelo.addConstr(producto_en_espacio(0) == cajuela);

// Compara la cantidad de producto disponible en cada espacio
   vs el espacio anterior. Obliga a reiniciar el nivel si es
   un almacen.
for (int e = 1; e < espacios; ++e) {
   GRBLinExpr demanda;
   for (cliente c : clientes) {
      demanda += c.demanda * vertice_en_espacio(c.vertice, e);
   }
   modelo.addConstr(producto_en_espacio(e) <= cajuela *
      vertice_en_espacio(almacen, e) + producto_en_espacio(e -
```

```
        1) - demanda);
    modelo.addConstr(producto_en_espacio(e) >= cajuela *
       vertice_en_espacio(almacen, e));
}

// ---------------------- Restricciones de combustible
   ---------------------

// A cada espacio se le asocia una variable real del
   combustible en el vehiculo en ese punto de la ruta.
matrix<GRBVar, 1> combustible_en_espacio(espacios);
for (int e = 0; e < espacios; ++e){
    combustible_en_espacio(e) = modelo.addVar(0, tanque, 0,
       GRB_CONTINUOUS, formatted_string("combustible_en_espacio
       [%d]", e));      // <-- la cota superior es lo que nos
       permite obviar que no se recargue mas que el tanque
}
modelo.addConstr(combustible_en_espacio(0) == g0);

// La cantidad de combustible recargado por una gasolinera
   especifica en cada espacio (no podemos recargar si no
   visitamos la gasolinera).
matrix<GRBVar, 2> recarga_combustible_en_espacio(vertices,
   espacios);
for (gasolinera g : gasolineras) {
    for (int e = 0; e < espacios; ++e) {
        recarga_combustible_en_espacio(g.vertice, e) = modelo.
           addVar(0, tanque, 0, GRB_CONTINUOUS, formatted_string
           ("recarga_combustible_en_espacio[%d][%d]", g.vertice,
            e));
        modelo.addConstr(recarga_combustible_en_espacio(g.
           vertice, e) <= tanque * vertice_en_espacio(g.vertice,
            e));
    }
}
```

```
// El combustible gastado en un traslado debe ser suficiente
   para realizar el traslado y el combustible disponible es el
    restante mas la recarga. Obliga a reiniciar el nivel si es
    un almacen.
// El nivel de combustible al final de cada ruta debe ser al
   menos el nivel de combustible inicial.
for (int e = 1; e < espacios; ++e) {
   GRBLinExpr gasto;
   for (int v1 = 0; v1 < vertices; ++v1) {
      for (int v2 = 0; v2 < vertices; ++v2) {
         gasto += matriz(v1, v2).distancia * arco_en_traslado(
            v1, v2, e - 1);
      }
   }
   modelo.addConstr(combustible_en_espacio(e - 1) >= gasto);
   modelo.addConstr(combustible_en_espacio(e - 1) - gasto >=
      g0 * vertice_en_espacio(almacen, e));

   GRBLinExpr recarga;
   for (gasolinera g : gasolineras) {
      recarga += recarga_combustible_en_espacio(g.vertice, e);
   }

   GRBVar sobrante_efectivo = modelo.addVar(0, tanque, 0,
      GRB_CONTINUOUS, formatted_string("sobrante_efectivo[%d]"
      , e));
   modelo.addConstr(sobrante_efectivo <=
      combustible_en_espacio(e - 1) - gasto + recarga);
   modelo.addConstr(sobrante_efectivo <= tanque * (1 -
      vertice_en_espacio(almacen, e)));
   modelo.addConstr(combustible_en_espacio(e) ==
      sobrante_efectivo + (g0 * vertice_en_espacio(almacen, e)
      ));
}
```

```
// ——————————————— Objetivos ———————————————

GRBLinExpr distancia;
for (int v1 = 0; v1 < vertices; ++v1) {
   for (int v2 = 0; v2 < vertices; ++v2) {
      for (int t = 0; t < traslados; ++t) {
         distancia += matriz(v1, v2).distancia *
            arco_en_traslado(v1, v2, t);
      }
   }
}

GRBLinExpr precio;
for (gasolinera g : gasolineras) {
   for (int e = 0; e < espacios; ++e) {
      precio += g.costo_metro * recarga_combustible_en_espacio
         (g.vertice, e);
   }
}

GRBLinExpr tiempo;
for (int v1 = 0; v1 < vertices; ++v1) {
   for (int v2 = 0; v2 < vertices; ++v2) {
      for (int t = 0; t < traslados; ++t) {
         tiempo += matriz(v1, v2).tiempo * arco_en_traslado(v1
            , v2, t);
      }
   }
}
for (gasolinera g : gasolineras) {
   for (int e = 0; e < espacios; ++e) {
      tiempo += g.tiempo_servicio * vertice_en_espacio(g.
         vertice, e);
   }
```

```
}

// ---------------------- Resolucion del modelo ----------------------

auto pos = instancia.find_last_of('.');
if (pos != std::string::npos) {
   instancia.erase(pos);
}
modelo.write(formatted_string("%s.lp", instancia.c_str( )));
modelo.set(GRB_IntAttr_ModelSense, GRB_MINIMIZE);

auto optimiza_modelo = [&](int i, int j, int k) {
   std::cout << "*** OPTIMIZANDO MODELO CON PESOS " << i << "
      " << j << " " << k << "***\n";
   modelo.setObjectiveN(distancia, 0, 0, i);
   modelo.setObjectiveN(precio,    1, 0, j);
   modelo.setObjectiveN(tiempo,    2, 0, k);
   modelo.optimize( );
   if (modelo.get(GRB_IntAttr_Status) != GRB_INFEASIBLE) {
      std::string instancia_pesada = formatted_string("%s_%d_%
         d_%d", instancia.c_str( ), i, j, k);
      modelo.write(formatted_string("%s.sol", instancia_pesada
         .c_str( )));
      std::ofstream salida(formatted_string("%s.out",
         instancia_pesada.c_str( )));
      for (i = 0; i < 3; ++i) {
         modelo.set(GRB_IntParam_ObjNumber, i);
         salida << modelo.get(GRB_DoubleAttr_ObjNVal) << "\n";
      }
   } else {
      modelo.computeIIS( );
      modelo.write(formatted_string("%s.ilp", instancia.c_str(
          )));
      throw std::runtime_error(formatted_string("Instancia %s
         infactible", instancia.c_str( )));
```

```
      }
   };

   try {
      optimiza_modelo(1, 1, 1);
   } catch (...) {
      std::cout << "***␣SIN␣SOLUCION␣EN␣TIEMPO␣PARA␣MODELO␣CON␣
         PESOS␣1␣1␣1␣***\n";
   }
   for (int i = 0; i <= 100; i += 10) {
      for (int j = 0; j <= 100; j += 10) {
         for (int k = 0; k <= 100; k += 10) {
            if (i + j + k == 100) {
               try {
                  optimiza_modelo(i, j, k);
               } catch (...) {
                  std::cout << "***␣SIN␣SOLUCION␣EN␣TIEMPO␣PARA␣
                     MODELO␣CON␣PESOS␣" << i << "␣" << j << "␣"
                     << k << "***\n";
               }
            }
         }
      }
   }

} catch (const GRBException& ex) {
   std::cout << ex.getMessage( ) << "\n";
} catch (const std::exception& ex) {
   std::cout << ex.what( ) << "\n";
}
```

A.2. Implementación cooperativa del modelo de programación entera lineal utilizando la interfaz de programación de C++ de Gurobi

```
#include <algorithm>
#include <cmath>
#include <cfloat>
#include <exception>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <iterator>
#include <string>
#include "gurobi_c++.h"
#include "matrix.h"

template<typename... T>
std::string formatted_string(const char* formato, const T&...
   variables) {
   char buffer[1024 + 1];
   std::sprintf(buffer, formato, variables...);
   return buffer;
}

struct cliente {
   int vertice;
   double demanda;
};

struct gasolinera {
   int vertice;
   double costo_metro, tiempo_servicio;
};
```

```
struct costo_arco {
   double distancia, tiempo;
};

double evaluar_vector(const std::vector<int>& ruta, const matrix<
   costo_arco, 2>& matriz, int peso_distancia, int peso_tiempo) {
   double costo_total = 0;
   for (int i = 1; i < ruta.size( ); ++i) {
      costo_total += peso_distancia * matriz(ruta[i - 1], ruta[i
         ]).distancia +
                     peso_tiempo * matriz(ruta[i - 1], ruta[i
                        ]).tiempo;
   }
   return costo_total;
}

std::pair<double, std::vector<int>> heuristica(const std::vector<
   int>& mejor_gurobi, const matrix<costo_arco, 2>& matriz, int
   almacen, int peso_distancia, int peso_tiempo) {
   std::pair<double, std::vector<int>> solucion = { 0,
      mejor_gurobi };

   for (int i = 0; i < solucion.second.size( ) - 1; ++i) {
      if (solucion.second[i] == almacen && solucion.second[i + 1]
          != almacen) {  // inicio ruta
         std::vector<int>::iterator fin_ruta = std::find(solucion
            .second.begin( ) + i + 1, solucion.second.end( ),
            almacen);
         std::vector<int> ruta;
         std::copy (solucion.second.begin( ) + i, fin_ruta + 1,
            std::back_inserter(ruta));
         double costo_ruta = evaluar_vector(ruta, matriz,
            peso_distancia, peso_tiempo);
         bool cambio;
         do {
```

```
            cambio = false;
            std::vector<int> mejor;
            double costo_mejor = DBL_MAX;
            for (int j = 1; j < ruta.size( ); ++j) {
               for ( int k = j + 1; k < ruta.size( ); ++k) {
                  std::vector<int> vecino = ruta;
                  std::reverse(vecino.begin( ) + j, vecino.begin(
                      ) + k);
                  double costo_vecino = evaluar_vector(vecino,
                     matriz, peso_distancia, peso_tiempo);
                  if (costo_mejor > costo_vecino) {
                     mejor = vecino;
                     costo_mejor = costo_vecino;
                  }
               }
            }
            if (costo_ruta > costo_mejor) {
               ruta = mejor;
               costo_ruta = costo_mejor;
               cambio = true;
            }
         } while (cambio);
         std::copy (ruta.begin( ), ruta.end( ), solucion.second.
            begin( ) + i);
         solucion.first += costo_ruta;
      }
   }
   return solucion;
}

struct retrollamada : GRBCallback {
   retrollamada(matrix<GRBVar, 2>& v_e, matrix<costo_arco, 2>& m,
       int v, int e, int a, int d, int t)
   : vertice_en_espacio(v_e), matriz(m), vertices(v), espacios(e)
      , almacen(a), peso_distancia(d), peso_tiempo(t),
```

```
    mejor_gurobi(DBL_MAX, std::vector<int>(espacios)),
    entregar_propuesta(false) {
}

void callback( ) {
    if (where == GRB_CB_MIPSOL) { // encontro una mejor
        solucion factible
        for (int v = 0; v < vertices; ++v) {
            for (int e = 0; e < espacios; ++e) {
                if (std::round(getSolution(vertice_en_espacio(v, e
                    )))) {
                    mejor_gurobi.second[e] = v;
                }
            }
        }
        mejor_gurobi.first = evaluar_vector(mejor_gurobi.second,
            matriz, peso_distancia, peso_tiempo);
        entregar_propuesta = true;
    } else if (where == GRB_CB_MIPNODE && entregar_propuesta) {
        // actualmente explorando un nodo
        std::pair<double, std::vector<int>> propuesta =
            heuristica(mejor_gurobi.second, matriz, almacen,
            peso_distancia, peso_tiempo);
        if (propuesta.first < mejor_gurobi.first) {
            std::cerr << "proponemos␣pasar␣de␣" << mejor_gurobi.
                first << "␣a␣" << propuesta.first << "\n";
            for (int e = 0; e < espacios; ++e) {
                setSolution(vertice_en_espacio(propuesta.second[e
                    ], e), true);
            }
        }
        entregar_propuesta = false;
    }
}
```

```
   int vertices, espacios, almacen;
   int peso_distancia, peso_tiempo;
   const matrix<GRBVar, 2>& vertice_en_espacio;
   const matrix<costo_arco, 2>& matriz;
   std::pair<double, std::vector<int>> mejor_gurobi;
   bool entregar_propuesta;
};

int main(int argc, const char* argv[])
try {
   if (argc <= 1) {
      throw std::runtime_error(formatted_string("Uso: %s
         instancia", argv[0]));
   }

   std::string instancia = argv[1];
   std::ifstream entrada(instancia);
   if (!entrada.is_open( )) {
      throw std::runtime_error(formatted_string("Error al abrir %
         s", instancia.c_str( )));
   }

   int vertices, vehiculos; double cajuela, tanque, g0, kmpl;
   entrada >> vertices >> vehiculos >> cajuela >> tanque >> g0 >>
       kmpl;
   tanque *= kmpl * 1000, g0 *= kmpl * 1000;   // Convertir
      litros a metros

   int almacen;
   std::vector<cliente> clientes;
   std::vector<gasolinera> gasolineras;
   for (int v = 0; v < vertices; ++v) {
      double x, y, c1, c2; std::string direccion;
      entrada >> x >> y >> c1 >> c2;
      std::getline(entrada >> std::ws, direccion);
```

```
      if (c1 == -1 && c2 == -1) {
         almacen = v;
      } else if (c1 == -1) {
         clientes.push_back({ v, c2 });
      } else {
         gasolineras.push_back({ v, c1 / (kmpl * 1000), c2 });
      }
   }

   matrix<costo_arco, 2> matriz(vertices, vertices);
   for (int v1 = 0; v1 < vertices; ++v1) {
      for (int v2 = 0; v2 < vertices; ++v2) {
         entrada >> matriz(v1, v2).distancia;
      }
   }
   for (int v1 = 0; v1 < vertices; ++v1) {
      for (int v2 = 0; v2 < vertices; ++v2) {
         entrada >> matriz(v1, v2).tiempo;
      }
   }

   // Construccion del modelo

   GRBEnv ambiente;
   ambiente.set(GRB_IntParam_Method, 3);
   ambiente.set(GRB_IntParam_MIPFocus, 1);
   ambiente.set(GRB_IntParam_NumericFocus, 1);
   ambiente.set(GRB_IntParam_Presolve, 2);
   ambiente.set(GRB_DoubleParam_TimeLimit, 600);
   GRBModel modelo(ambiente);

   // ---------- Restricciones de la estrategia general de
       espacios ----------

   // Apartaremos espacio suficiente para que podamos visitar una
```

```
    gasolinera en cada ruta, ademas de que cada ruta empiece
   en el almacen
// (el almacen al final de una ruta es el inicio de otra, o
   bien, ademas tenemos un almacen final).
const int espacios = clientes.size( ) + 2 * vehiculos + 1;
const int traslados = espacios - 1;

// Cada espacio de la solucion puede contener cualquier
   vertice.
matrix<GRBVar, 2> vertice_en_espacio(vertices, espacios);
for (int v = 0; v < vertices; ++v) {
   for (int e = 0; e < espacios; ++e) {
      vertice_en_espacio(v, e) = modelo.addVar(0, 1, 0,
         GRB_BINARY, formatted_string("vertice_en_espacio[%d
         ][%d]", v, e));
   }
}

// Solo un vertice por espacio.
for (int e = 0; e < espacios; ++e){
   GRBLinExpr ex;
   for (int v = 0; v < vertices; ++v){
      ex += vertice_en_espacio(v, e);
   }
   modelo.addConstr(ex == 1);
}

// Cada cliente en solo un espacio.
for (cliente c : clientes) {
   GRBLinExpr ex;
   for (int e = 0; e < espacios; ++e) {
      ex += vertice_en_espacio(c.vertice, e);
   }
   modelo.addConstr(ex == 1);
}
```

```
// El primer y ultimo espacios contienen al almacen.
modelo.addConstr(vertice_en_espacio(almacen, 0) == 1);
modelo.addConstr(vertice_en_espacio(almacen, espacios - 1) ==
   1);

// Entre cada pareja de espacios existe el correspondiente
   traslado, que usa un arco. El k-esimo traslado corresponde
   a pasar del vertice en el espacio k al vertice en el
   espacio k+1.
matrix<GRBVar, 3> arco_en_traslado(vertices, vertices,
   traslados);
for (int v1 = 0; v1 < vertices; ++v1) {
   for (int v2 = 0; v2 < vertices; ++v2) {
      for (int t = 0; t < traslados; ++t) {
         arco_en_traslado(v1, v2, t) = modelo.addVar(0, 1, 0,
            GRB_BINARY, formatted_string("arco_en_traslado[%d
            ][%d][%d]", v1, v2, t));
         GRBVar extremos[] = { vertice_en_espacio(v1, t),
            vertice_en_espacio(v2, t + 1) };
         modelo.addGenConstrAnd(arco_en_traslado(v1, v2, t),
            extremos, 2);
      }
   }
}

// Ir de almacen a almacen indica no-actividad; cualquier
   llegada al almacen desde otro vertice es el fin de la ruta
   de un vehiculo. Tenemos un numero exacto de vehiculos a
   usar.
GRBLinExpr viajes_almacen;
for (int t = 0; t < traslados; ++t) {
   for (int v = 0; v < vertices; ++v){
      if (v != almacen) {
         viajes_almacen += arco_en_traslado(v, almacen, t);
```

```
        }
    }
}
modelo.addConstr(viajes_almacen == vehiculos);

// ---------- Restricciones de capacidad ----------

// A cada espacio se le asocia una variable entera del
   producto diponible en el vehiculo en ese punto de la ruta.
matrix<GRBVar, 1> producto_en_espacio(espacios);
for (int e = 0; e < espacios; ++e){
    producto_en_espacio(e) = modelo.addVar(0, cajuela, 0,
        GRB_INTEGER, formatted_string("producto_en_espacio[%d]",
         e));
}
modelo.addConstr(producto_en_espacio(0) == cajuela);

// Compara la cantidad de producto disponible en cada espacio
   vs el espacio anterior. Obliga a reiniciar el nivel si es
   un almacen.
for (int e = 1; e < espacios; ++e) {
    GRBLinExpr demanda;
    for (cliente c : clientes) {
        demanda += c.demanda * vertice_en_espacio(c.vertice, e);
    }
    modelo.addConstr(producto_en_espacio(e) <= cajuela *
        vertice_en_espacio(almacen, e) + producto_en_espacio(e -
         1) - demanda);
    modelo.addConstr(producto_en_espacio(e) >= cajuela *
        vertice_en_espacio(almacen, e));
}

// ---------- Restricciones de combustible ----------
```

```
// A cada espacio se le asocia una variable real del
   combustible en el vehiculo en ese punto de la ruta.
matrix<GRBVar, 1> combustible_en_espacio(espacios);
for (int e = 0; e < espacios; ++e){
   combustible_en_espacio(e) = modelo.addVar(0, tanque, 0,
      GRB_CONTINUOUS, formatted_string("combustible_en_espacio
      [%d]", e));      // <-- la cota superior es lo que nos
      permite obviar que no se recargue mas que el tanque
}
modelo.addConstr(combustible_en_espacio(0) == g0);

// La cantidad de combustible recargado por una gasolinera
   especifica en cada espacio (no podemos recargar si no
   visitamos la gasolinera).
matrix<GRBVar, 2> recarga_combustible_en_espacio(vertices,
   espacios);
for (gasolinera g : gasolineras) {
   for (int e = 0; e < espacios; ++e) {
      recarga_combustible_en_espacio(g.vertice, e) = modelo.
         addVar(0, tanque, 0, GRB_CONTINUOUS, formatted_string
         ("recarga_combustible_en_espacio[%d][%d]", g.vertice,
          e));
      modelo.addConstr(recarga_combustible_en_espacio(g.
         vertice, e) <= tanque * vertice_en_espacio(g.vertice,
          e));
   }
}

// El combustible gastado en un traslado debe ser suficiente
   para realizar el traslado y el combustible disponible es el
    restante mas la recarga. Obliga a reiniciar el nivel si es
    un almacen.
for (int e = 1; e < espacios; ++e) {
   GRBLinExpr gasto;
```

```
      for (int v1 = 0; v1 < vertices; ++v1) {
         for (int v2 = 0; v2 < vertices; ++v2) {
            gasto += matriz(v1, v2).distancia * arco_en_traslado(
               v1, v2, e - 1);
         }
      }
      modelo.addConstr(combustible_en_espacio(e - 1) >= gasto);
      modelo.addConstr(combustible_en_espacio(e - 1) - gasto >=
         g0 * vertice_en_espacio(almacen, e));

      GRBLinExpr recarga;
      for (gasolinera g : gasolineras) {
         recarga += recarga_combustible_en_espacio(g.vertice, e);
      }

      GRBVar sobrante_efectivo = modelo.addVar(0, tanque, 0,
         GRB_CONTINUOUS, formatted_string("sobrante_efectivo[%d]"
         , e));
      modelo.addConstr(sobrante_efectivo <=
         combustible_en_espacio(e - 1) - gasto + recarga);
      modelo.addConstr(sobrante_efectivo <= tanque * (1 -
         vertice_en_espacio(almacen, e)));
      modelo.addConstr(combustible_en_espacio(e) ==
         sobrante_efectivo + (g0 * vertice_en_espacio(almacen, e)
         ));
   }

   // ------------------- Objetivos --------------------

   GRBLinExpr distancia;
   for (int v1 = 0; v1 < vertices; ++v1) {
      for (int v2 = 0; v2 < vertices; ++v2) {
         for (int t = 0; t < traslados; ++t) {
            distancia += matriz(v1, v2).distancia *
               arco_en_traslado(v1, v2, t);
```

```
      }
   }
}

GRBLinExpr precio;
for (gasolinera g : gasolineras) {
   for (int e = 0; e < espacios; ++e) {
      precio += g.costo_metro * recarga_combustible_en_espacio
         (g.vertice, e);
   }
}

GRBLinExpr tiempo;
for (int v1 = 0; v1 < vertices; ++v1) {
   for (int v2 = 0; v2 < vertices; ++v2) {
      for (int t = 0; t < traslados; ++t) {
         tiempo += matriz(v1, v2).tiempo * arco_en_traslado(v1
            , v2, t);
      }
   }
}
for (gasolinera g : gasolineras) {
   for (int e = 0; e < espacios; ++e) {
      tiempo += g.tiempo_servicio * vertice_en_espacio(g.
         vertice, e);
   }
}

// ------------------- Resolucion del modelo --------------------

auto pos = instancia.find_last_of('.');
if (pos != std::string::npos) {
   instancia.erase(pos);
}
modelo.write(formatted_string("%s.lp", instancia.c_str( )));
```

```
modelo.set(GRB_IntAttr_ModelSense, GRB_MINIMIZE);

auto optimiza_modelo = [&](int i, int j, int k) {
   std::cout << "***␣OPTIMIZANDO␣MODELO␣CON␣PESOS␣" << i << "␣
      " << j << "␣" << k << "***\n";
   modelo.setObjectiveN(distancia, 0, 0, i);
   modelo.setObjectiveN(precio,    1, 0, j);
   modelo.setObjectiveN(tiempo,    2, 0, k);

   modelo.update( );
   retrollamada cb(vertice_en_espacio, matriz, vertices,
      espacios, almacen, i, k);
   modelo.setCallback(&cb);

   try {
      modelo.read(formatted_string("%s_%d_%d_%d.mst",
         instancia.c_str( ), i, j, k));
   } catch (const GRBException& ex) {
      std::cout << "No␣hay␣archivo␣de␣solucion␣inicial\n";
   }

   modelo.optimize( );
   if (modelo.get(GRB_IntAttr_Status) != GRB_INFEASIBLE) {
      std::string instancia_pesada = formatted_string("%s_%d_%
         d_%d", instancia.c_str( ), i, j, k);
      modelo.write(formatted_string("%s.sol", instancia_pesada
         .c_str( )));
      std::ofstream salida(formatted_string("%s.out",
         instancia_pesada.c_str( )));
      for (i = 0; i < 3; ++i) {
         modelo.set(GRB_IntParam_ObjNumber, i);
         salida << modelo.get(GRB_DoubleAttr_ObjNVal) << "\n";
      }
   } else {
      modelo.computeIIS( );
```

```
            modelo.write(formatted_string("%s.ilp", instancia.c_str(
                )));
            throw std::runtime_error(formatted_string("Instancia␣%s␣
               infactible", instancia.c_str( )));
         }
      };

      try {
         optimiza_modelo(1, 1, 1);
      } catch (...) {
         std::cout << "***␣SIN␣SOLUCION␣EN␣TIEMPO␣PARA␣MODELO␣CON␣
            PESOS␣1␣1␣1␣***\n";
      }
      for (int i = 0; i <= 100; i += 10) {
         for (int j = 0; j <= 100; j += 10) {
            for (int k = 0; k <= 100; k += 10) {
               if (i + j + k == 100) {
                  try {
                     optimiza_modelo(i, j, k);
                  } catch (...) {
                     std::cout << "***␣SIN␣SOLUCION␣EN␣TIEMPO␣PARA␣
                        MODELO␣CON␣PESOS␣" << i << "␣" << j << "␣"
                        << k << "***\n";
                  }
               }
            }
         }
      }
   } catch (const GRBException& ex) {
      std::cout << ex.getMessage( ) << "\n";
   } catch (const std::exception& ex) {
      std::cout << ex.what( ) << "\n";
   }
```

A.3. Algoritmo glotón de ahorros para k vehículos con abastecimiento

```
#include <algorithm>
#include <cmath>
#include <cfloat>
#include <deque>
#include <exception>
#include <fstream>
#include <iostream>
#include <queue>
#include <string>
#include <sstream>
#include "gurobi_c++.h"
#include "matrix.h"

template<typename... T>
std::string formatted_string(const char* formato, const T&...
   variables) {
   char buffer[1024 + 1];
   std::sprintf(buffer, formato, variables...);
   return buffer;
}

struct cliente {
   int vertice;
   double x, y, demanda;
};

struct gasolinera {
   int vertice;
   double x, y, costo_metro, tiempo_servicio;
};
```

```
struct costo_arco {
   double distancia, tiempo;
};

struct ahorro_arco {
   int vertice_origen, vertice_destino;
   double ahorro;
   bool operator<(const ahorro_arco &o) const {
      return ahorro < o.ahorro;
   }
};

bool oraculo_gurobi(GRBModel& modelo, GRBVar z1, GRBVar z2, std::
   vector<GRBConstr>& fijas) {
   GRBConstr intentar = modelo.addConstr(z1 == z2);
   modelo.optimize( );
   if (modelo.get(GRB_IntAttr_Status) != GRB_INFEASIBLE) {
      fijas.push_back(intentar);
      return true;
   } else {
      modelo.remove(intentar);
      return false;
   }
}

void insertar_gasolinera_cercana(std::deque<int>& ruta, const
   matrix<double, 2>& matriz, const std::vector<gasolinera>&
   gasolineras) {
   int gasolinera_cercana; double distancia_menor = DBL_MAX;
   for (gasolinera g : gasolineras) {
      double distancia_g = matriz(ruta[0], g.vertice) + matriz(g.
         vertice, ruta[1]);
      if (distancia_g < distancia_menor) {
         gasolinera_cercana = g.vertice;
         distancia_menor = distancia_g;
```

```
      }
   }
   ruta.insert (ruta.begin( ) + 1, gasolinera_cercana);
}

double busqueda_local(int vertices, int almacen, std::vector<std
   ::deque<int>>& ruta, const matrix<double, 2>& matriz, const
   matrix<costo_arco, 2>& matriz_original, const std::vector<
   gasolinera>& gasolineras) {
   double costo_total = 0;
   for (int v = 0; v < vertices; ++v) {
      if (!ruta[v].empty( )) {
         ruta[v].push_front(almacen);
         ruta[v].push_back(almacen);
         double distancia_ruta = 0;
         for (int c = 1; c < ruta[v].size( ); ++c) {
            distancia_ruta += matriz(ruta[v][c - 1], ruta[v][c]);
         }
         bool cambio;
         do {
            std::deque<int> mejor;
            double distancia_mejor = DBL_MAX;
            cambio = false;
            for (int i = 1; i < ruta[v].size( ); ++i) {
               for ( int j = i + 1; j < ruta[v].size( ); ++j) {
                  std::deque<int> vecino = ruta[v];
                  std::reverse(vecino.begin( ) + i, vecino.begin(
                      ) + j);
                  double distancia_vecino = 0;
                  for (int c = 1; c < vecino.size( ); ++c) {
                     distancia_vecino += matriz(vecino[c - 1],
                        vecino[c]);
                  }
                  if (distancia_mejor > distancia_vecino) {
                     mejor = vecino;
```

```
                    distancia_mejor = distancia_vecino;
                }
            }
        }
        if (distancia_ruta > distancia_mejor) {
            ruta[v] = mejor;
            distancia_ruta = distancia_mejor;
            cambio = true;
        }
    } while (cambio);

    insertar_gasolinera_cercana(ruta[v], matriz, gasolineras
        );
    distancia_ruta += matriz(ruta[v][0], ruta[v][1]) +
        matriz(ruta[v][1], ruta[v][2]) - matriz(ruta[v][0],
        ruta[v][2]);
    costo_total += distancia_ruta;
  }
 }
 return costo_total;
}

int main(int argc, const char* argv[])
try {
 if (argc <= 1) {
    throw std::runtime_error(formatted_string("Uso: %s
        instancia", argv[0]));
 }

 std::string instancia = argv[1];
 std::ifstream entrada(instancia);
 if (!entrada.is_open( )) {
    throw std::runtime_error(formatted_string("Error al abrir %
        s", instancia.c_str( )));
 }
```

```
auto pos = instancia.find_last_of('.');
if (pos != std::string::npos) {
   instancia.erase(pos);
}

int vertices, vehiculos; double cajuela, tanque, g0, kmpl;
entrada >> vertices >> vehiculos >> cajuela >> tanque >> g0 >>
    kmpl;
tanque *= kmpl * 1000, g0 *= kmpl * 1000;   // Convertir
   litros a metros

int almacen;
std::vector<cliente> clientes;
std::vector<gasolinera> gasolineras;
for (int v = 0; v < vertices; ++v) {
   double x, y, c1, c2; std::string direccion;
   entrada >> x >> y >> c1 >> c2;
   std::getline(entrada >> std::ws, direccion);
   if (c1 == -1 && c2 == -1) {
      almacen = v;
   } else if (c1 == -1) {
      clientes.push_back({ v, x, y, c2 });
   } else {
      gasolineras.push_back({ v, x, y, c1 / (kmpl * 1000), c2
         });
   }
}

matrix<costo_arco, 2> matriz_original(vertices, vertices);
for (int v1 = 0; v1 < vertices; ++v1) {
   for (int v2 = 0; v2 < vertices; ++v2) {
      entrada >> matriz_original(v1, v2).distancia;
   }
}
```

```
for (int v1 = 0; v1 < vertices; ++v1) {
   for (int v2 = 0; v2 < vertices; ++v2) {
      entrada >> matriz_original(v1, v2).tiempo;
   }
}

GRBEnv ambiente;
GRBModel modelo(ambiente);
modelo.set(GRB_IntParam_OutputFlag, 0);

GRBVar cliente_en_vehiculo[vertices][vehiculos],
   vehiculo_de_cliente[vertices];
for (const auto& c : clientes) {
   for (int v = 0; v < vehiculos; ++v) {
      cliente_en_vehiculo[c.vertice][v] = modelo.addVar(0, 1,
         0, GRB_BINARY, formatted_string("cliente_[%d]
         _en_vehiculo_[%d]", c.vertice, v));
   }
   vehiculo_de_cliente[c.vertice] = modelo.addVar(0, vehiculos
      , 0, GRB_INTEGER, formatted_string("vehiculo_de_cliente_
      [%d]", c.vertice));
}
for (const auto& c : clientes) {
   GRBLinExpr ex1, ex2;
   for (int v = 0; v < vehiculos; ++v) {
      ex1 += cliente_en_vehiculo[c.vertice][v];
      ex2 += cliente_en_vehiculo[c.vertice][v] * v;
   }
   modelo.addConstr(ex1 == 1);
   modelo.addConstr(ex2 == vehiculo_de_cliente[c.vertice]);
}
for (int v = 0; v < vehiculos; ++v) {
   GRBLinExpr ex1, ex2;
   for (const auto& c : clientes) {
      ex1 += cliente_en_vehiculo[c.vertice][v];
```

```
            ex2 += cliente_en_vehiculo[c.vertice][v] * c.demanda;
        }
        modelo.addConstr(ex1 >= 1);
        modelo.addConstr(ex2 <= cajuela);
    }

    for (int i = 0; i <= 100; i += 10) {
        for (int j = 0; j <= 100; j += 10) {
            for (int k = 0; k <= 100; k += 10) {
                if (i + j + k == 100) {
                    int peso_distancia = i, peso_precio = j,
                        peso_tiempo = k;

                    matrix<double, 2> matriz(vertices, vertices);
                    for (int v1 = 0; v1 < vertices; ++v1) {
                        for (int v2 = 0; v2 < vertices; ++v2) {
                            matriz(v1, v2) = peso_distancia *
                                matriz_original(v1, v2).distancia +
                                                peso_tiempo *
                                                    matriz_original(v1, v2).
                                                    tiempo;
                        }
                    }

                    // ---------------------- Ahorros con bin-packing
                        ---------------------
                    // Se crea una ruta para cada cliente.
                    std::vector<std::deque<int>> ruta(vertices);
                    int ruta_de_cliente[vertices] = { };
                    double demanda_de_ruta[vertices] = { };
                    std::vector<GRBConstr> fijas;
                    for (const auto& c : clientes) {
                        ruta[c.vertice].push_back(c.vertice);
                        ruta_de_cliente[c.vertice] = c.vertice;
                        demanda_de_ruta[c.vertice] = c.demanda;
```

```
}
// Se calculan ahorros y se ordenan de forma no
   descendente.
std::priority_queue <ahorro_arco> q;
for (const auto& c1 : clientes) {
   for (const auto& c2 : clientes) {
      if (c1.vertice != c2.vertice) {
         double ahorro = matriz(c1.vertice,
            almacen) + matriz(almacen, c2.vertice)
             - matriz(c1.vertice, c2.vertice);
         q.push({ c1.vertice, c2.vertice, ahorro
            });
      }
   }
}

while (!q.empty( )) {
   ahorro_arco mayor = q.top( );
   int ruta_origen = ruta_de_cliente[mayor.
      vertice_origen];
   int ruta_destino = ruta_de_cliente[mayor.
      vertice_destino];
   if (ruta[ruta_origen].back( ) == mayor.
      vertice_origen &&
      ruta[ruta_destino].front( ) == mayor.
         vertice_destino &&
      ruta_origen != ruta_destino &&
         demanda_de_ruta[ruta_origen] +
         demanda_de_ruta[ruta_destino] <= cajuela
         &&
      oraculo_gurobi(modelo, vehiculo_de_cliente[
         mayor.vertice_origen],
         vehiculo_de_cliente[mayor.vertice_destino
         ], fijas)
      ) {
```

```
            // Aceptar ahorro, unir rutas y actualizar
               arreglos.
            for (int vertice : ruta[ruta_destino]) {
               ruta[ruta_origen].push_back(vertice);
               ruta_de_cliente[vertice] = ruta_origen;
            }
            ruta[ruta_destino].clear( );
            demanda_de_ruta[ruta_origen] +=
               demanda_de_ruta[ruta_destino];
            demanda_de_ruta[ruta_destino] = 0;
         }
         q.pop( );
      }

      double costo_total = 0;
      costo_total = busqueda_local(vertices, almacen,
         ruta, matriz, matriz_original, gasolineras);

      std::ofstream salida(formatted_string("%s_%d_%d_%d
         .mst", instancia.c_str( ), peso_distancia,
         peso_precio, peso_tiempo));
      salida << "#␣Solucion␣de␣ahorros_mochila_reverse␣
         con␣costo:␣" << costo_total << "\n";
      int espacio = 0;
      for (int v = 0; v < vertices; ++v) {
         if (!ruta[v].empty( )) {
            for (int c = 0; c < ruta[v].size( ) - 1; ++c
               ) {
               salida << "vertice_en_espacio[" << ruta[v
                  ][c] << "][" << espacio << "]␣␣1\n";
               ++espacio;
            }
         }
      }
      salida << "vertice_en_espacio[" << almacen << "]["
```

```
                    << espacio << "]__1\n";

                for (auto actual : fijas) {
                    modelo.remove(actual);
                }
            }
        }
    }

} catch (const GRBException& ex) {
    std::cout << ex.getMessage( ) << "\n";
} catch (const std::exception& ex) {
    std::cout << ex.what( ) << "\n";
}
```

A.4. Heurística geométrica de barrido para k vehículos con abastecimiento

```
#include <algorithm>
#include <cmath>
#include <cfloat>
#include <exception>
#include <fstream>
#include <iostream>
#include <queue>
#include <string>
#include <sstream>
#include "gurobi_c++.h"
#include "matrix.h"

template<typename... T>
std::string formatted_string(const char* formato, const T&...
    variables) {
    char buffer[1024 + 1];
```

```
   std::sprintf(buffer, formato, variables...);
   return buffer;
}

struct cliente {
   int vertice;
   double x, y, demanda, angulo;
   bool operator<(const cliente& c) const {
      return angulo < c.angulo;
   }
};

struct gasolinera {
   int vertice;
   double x, y, costo_metro, tiempo_servicio;
};

struct costo_arco {
   double distancia, tiempo;
};

constexpr double PI = 3.14159265;

void insertar_gasolinera_cercana(std::vector<int>& ruta, const
   matrix<double, 2>& matriz, const std::vector<gasolinera>&
   gasolineras) {
   int gasolinera_cercana; double distancia_menor = DBL_MAX;
   for (gasolinera g : gasolineras) {
      double distancia_g = matriz(ruta[0], g.vertice) + matriz(g.
         vertice, ruta[1]);
      if (distancia_g < distancia_menor) {
         gasolinera_cercana = g.vertice;
         distancia_menor = distancia_g;
      }
   }
```

```
   ruta.insert (ruta.begin( ) + 1, gasolinera_cercana);
}

double busqueda_local(int vehiculos, std::vector<std::vector<int
   >>& ruta, const matrix<double, 2>& matriz, const matrix<
   costo_arco, 2>& matriz_original, const std::vector<gasolinera
   >& gasolineras) {
   double costo_total = 0;
   for (int v = 0; v < vehiculos; ++v) {
      double distancia_ruta = 0;
      for (int c = 1; c < ruta[v].size( ); ++c) {
         distancia_ruta += matriz(ruta[v][c - 1], ruta[v][c]);
      }
      bool cambio;
      do {
         cambio = false;
         std::vector<int> mejor;
         double distancia_mejor = DBL_MAX;

         for (int i = 1; i < ruta[v].size( ); ++i) {
            for ( int j = i + 1; j < ruta[v].size( ); ++j) {
               std::vector<int> vecino = ruta[v];
               std::reverse(vecino.begin( ) + i, vecino.begin( )
                  + j);
               double distancia_vecino = 0;
               for (int c = 1; c < vecino.size( ); ++c) {
                  distancia_vecino += matriz(vecino[c - 1],
                     vecino[c]);
               }
               if (distancia_mejor > distancia_vecino) {
                  mejor = vecino;
                  distancia_mejor = distancia_vecino;
               }
            }
         }
```

```
            if (distancia_ruta > distancia_mejor) {
               ruta[v] = mejor;
               distancia_ruta = distancia_mejor;
               cambio = true;
            }
         } while (cambio);

         insertar_gasolinera_cercana(ruta[v], matriz, gasolineras);
         distancia_ruta += matriz(ruta[v][0], ruta[v][1]) + matriz(
            ruta[v][1], ruta[v][2]) - matriz(ruta[v][0], ruta[v][2])
            ;
         costo_total += distancia_ruta;
      }
      return costo_total;
}

int main(int argc, const char* argv[])
try {
      if (argc <= 1) {
         throw std::runtime_error(formatted_string("Uso: %s
            instancia", argv[0]));
      }

      std::string instancia = argv[1];
      std::ifstream entrada(instancia);
      if (!entrada.is_open( )) {
         throw std::runtime_error(formatted_string("Error al abrir %
            s", instancia.c_str( )));
      }

      auto pos = instancia.find_last_of('.');
      if (pos != std::string::npos) {
         instancia.erase(pos);
      }
```

```
int vertices, vehiculos; double cajuela, tanque, g0, kmpl;
entrada >> vertices >> vehiculos >> cajuela >> tanque >> g0 >>
    kmpl;
tanque *= kmpl * 1000, g0 *= kmpl * 1000;   // Convertir
   litros a metros

int almacen; double x_almacen, y_almacen;
std::vector<cliente> clientes;
std::vector<gasolinera> gasolineras;
for (int v = 0; v < vertices; ++v) {
   double x, y, c1, c2; std::string direccion;
   entrada >> x >> y >> c1 >> c2;
   std::getline(entrada >> std::ws, direccion);
   if (c1 == -1 && c2 == -1) {
      almacen = v, x_almacen = x, y_almacen = y;
   } else if (c1 == -1) {
      clientes.push_back({ v, x, y, c2 });
   } else {
      gasolineras.push_back({ v, x, y, c1 / (kmpl * 1000), c2
         });
   }
}
for (auto& c : clientes) {
   c.angulo = std::atan2(c.y - y_almacen, c.x - x_almacen) *
      180 / PI;
   if (c.y < y_almacen) {
      c.angulo += 360;
   }
}

matrix<costo_arco, 2> matriz_original(vertices, vertices);
for (int v1 = 0; v1 < vertices; ++v1) {
   for (int v2 = 0; v2 < vertices; ++v2) {
      entrada >> matriz_original(v1, v2).distancia;
   }
```

```
    }
    for (int v1 = 0; v1 < vertices; ++v1) {
       for (int v2 = 0; v2 < vertices; ++v2) {
          entrada >> matriz_original(v1, v2).tiempo;
       }
    }

    // ------------------- Barrido con bin-packing --------------------
    std::sort(clientes.begin( ), clientes.end( ));
    std::vector<cliente> orden_ascendente(clientes.size( )),
       orden_descendente(clientes.size( ));
    double demanda_promedio = 0;
    for (int i = 0; i < clientes.size( ); ++i) {
       orden_ascendente[i] = clientes[i];
       orden_descendente[clientes.size( ) - i - 1] = clientes[i];
       demanda_promedio += clientes[i].demanda;
    }
    demanda_promedio /= vehiculos;

    GRBEnv ambiente;
    GRBModel modelo(ambiente);
    modelo.set(GRB_IntParam_OutputFlag, 0);

    GRBVar variables[vertices][vehiculos];
    for (const auto& c : clientes) {
       for (int v = 0; v < vehiculos; ++v) {
          variables[c.vertice][v] = modelo.addVar(0, 1, 0,
             GRB_BINARY);
       }
    }
    for (const auto& c : clientes) {
       GRBLinExpr ex;
       for (int v = 0; v < vehiculos; ++v) {
          ex += variables[c.vertice][v];
       }
```

```
      modelo.addConstr(ex == 1);
   }
   for (int v = 0; v < vehiculos; ++v) {
      GRBLinExpr ex1, ex2;
      for (const auto& c : clientes) {
         ex1 += variables[c.vertice][v];
         ex2 += variables[c.vertice][v] * c.demanda;
      }
      modelo.addConstr(ex1 >= 1);
      modelo.addConstr(ex2 <= cajuela);
   }

   for (int i = 0; i <= 100; i += 10) {
      for (int j = 0; j <= 100; j += 10) {
         for (int k = 0; k <= 100; k += 10) {
            if (i + j + k == 100) {
               int peso_distancia = i, peso_precio = j,
                  peso_tiempo = k;
               matrix<double, 2> matriz(vertices, vertices);
               for (int v1 = 0; v1 < vertices; ++v1) {
                  for (int v2 = 0; v2 < vertices; ++v2) {
                     matriz(v1, v2) = peso_distancia *
                        matriz_original(v1, v2).distancia +
                                       peso_tiempo *
                                          matriz_original(v1, v2).
                                          tiempo;
                  }
               }

               double costo_mejor = DBL_MAX;
               std::vector<std::vector<int>> mejor_asignacion(
                  vehiculos);

               std::pair<const char*, std::vector<cliente>&>
                  ordenes[] = {
```

```
        { "ascendente",  orden_ascendente },
        { "descendente", orden_descendente }
    };
    for (auto procesar : ordenes) {
        auto& mensaje = procesar.first;
        auto& barrido = procesar.second;

        for (int rotacion = 0; rotacion < clientes.size
            ( ); ++rotacion) {
            std::vector<GRBConstr> fijas;
            std::vector<std::vector<int>> ruta(vehiculos
                );
            bool cliente_asignado[vertices] = { };
            for (int v = 0; v < vehiculos; ++v) {
                double demanda_ruta = 0;
                ruta[v].push_back(almacen);
                for (const auto& c : barrido) {
                    if (demanda_ruta + c.demanda > cajuela
                        || cliente_asignado[c.vertice]) {
                        continue;
                    }

                    if (demanda_ruta >= demanda_promedio)
                        {
                        auto intentar = modelo.addConstr(
                            variables[c.vertice][v] == 0);
                        modelo.optimize( );
                        if (modelo.get(GRB_IntAttr_Status)
                            != GRB_INFEASIBLE) {
                            fijas.push_back(intentar);
                            continue;
                        } else {
                            modelo.remove(intentar);
                        }
                    }
```

```
                auto intentar = modelo.addConstr(
                    variables[c.vertice][v] == 1);
                modelo.optimize( );
                if (modelo.get(GRB_IntAttr_Status) !=
                    GRB_INFEASIBLE) {
                    cliente_asignado[c.vertice] = true;
                    ruta[v].push_back(c.vertice);
                    demanda_ruta += c.demanda;
                    fijas.push_back(intentar);
                    continue;
                } else {
                    modelo.remove(intentar);
                }
            }
            ruta[v].push_back(almacen);
        }

        double costo_total = busqueda_local(
            vehiculos, ruta, matriz, matriz_original,
             gasolineras);

        if (costo_mejor > costo_total) {
            costo_mejor = costo_total;
            for (int v = 0; v < vehiculos; ++v) {
                mejor_asignacion[v] = ruta[v];
            }
        }

        std::rotate(barrido.begin( ), barrido.begin(
             ) + 1, barrido.end( ));
        for (auto actual : fijas) {
            modelo.remove(actual);
        }
    }
```

```
                }

                std::ofstream salida(formatted_string("%s_%d_%d_%d
                   .mst", instancia.c_str( ), peso_distancia,
                   peso_precio, peso_tiempo));
                salida << "#␣Solucion␣de␣barrido␣con␣costo:␣" <<
                   costo_mejor << "\n";

                int espacio = 0;
                for (int v = 0; v < vehiculos; ++v) {
                   for (int c = 0; c < mejor_asignacion[v].size( )
                       - 1; ++c) {
                      salida << "vertice_en_espacio[" <<
                         mejor_asignacion[v][c] << "][" << espacio
                          << "]␣␣1\n";
                      ++espacio;
                   }
                }
                salida << "vertice_en_espacio[" << almacen << "]["
                    << espacio << "]␣␣1\n";
             }
          }
       }
    }
} catch (const GRBException& ex) {
    std::cout << ex.getMessage( ) << "\n";
} catch (const std::exception& ex) {
    std::cout << ex.what( ) << "\n";
}
```

Bibliografía

[1] Balinski, M. L. y R. E. Quandt: *On an integer program for a delivery problem.* Operations Research, 12(2):300–304, 1964.

[2] Bektaş, T. y G. Laporte: *The pollution-routing problem.* Transportation Research Part B: Methodological, 45(8):1232–1250, 2011.

[3] Bousonville, T., A. Hartmann, T. Melo y H. Kopfer: *Vehicle routing and refueling: the impact of price variations on tour length.* Herausforderungen, Chancen und Lösungen Band II, pág. 83, 2011.

[4] Bramel, J. y D. Simchi-Levi: *A location based heuristic for general routing problems.* Operations Research, 43(4):649–660, 1995.

[5] Christofides, N., A. Mingozzi y P. Toth: *The vehicle routing problem.* Combinatorial Optimization, págs. 315–338, 1979.

[6] Clarke, G. y J. W. Wright: *Scheduling of vehicles from a central depot to a number of delivery points.* Operations Research, 12(10):568–581, 1964.

[7] Comisión Reguladora de Energía: *Listado de estaciones de servicio con georreferencia*, 2019. `https://datos.gob.mx/busca/dataset/estaciones-de-servicio-gasolineras-y-precios-finales-de-gasolina-y-diesel/resource/6dd2613c-9e48-4db3-aa6b-3a0bb9a0dacf`, visitado el 2019-08-02.

[8] Comisión Reguladora de Energía: *Listado de precios comerciales de gasolina y diésel por estación de servicio*, 2019. `https://datos.gob.mx/busca/dataset/estaciones-de-servicio-gasolineras-y-precios-finales-de-gasolina-y-diesel/resource/492f3eb5-c42b-4326-a9bb-e5c52528b544`, visitado el 2019-08-02.

[9] Dantzig, G., R. Fulkerson y S. Johnson: *Solution of a large-scale traveling-salesman problem.* Journal of the Operations Research Society of America, 2(4):393–410, 1954.

[10] Dantzig, G. B. y J. H. Ramser: *The truck and dispatching problem.* Management Science, 6(10):80–91, 1959.

[11] Eglese, R. y T. Bektas: *Green Vehicle Routing.* En Toth, P. y D. Vigo (eds.): *Vehicle routing. Problems, methods and applications*, cap. 15, págs. 437–458. Mathematical Optimization Society and Society for Industrial and Applied Mathematics., Philadelphia, 2014.

[12] Erdoğan, S. y E. Miller-Hooks: *A green vehicle routing problem.* Transportation Research Part E: Logistics and Transportation Review, 48(1):100–114, 2012.

[13] Foster, B. A. y D. M. Ryan: *An integer programming approach to the vehicle scheduling problem.* Journal of the Operational Research Society, 27(2):367–384, 1976.

[14] Gendreau, M., G. Laporte y J. Y. Potvin: *Metaheuristics for the capacitated VRP.* En *The vehicle routing problem*, págs. 129–154. SIAM, 2002.

[15] Gillett, B. E. y L. R. Miller: *A heuristic algorithm for the vehicle-dispatch problem.* Operations Research, 22(2):340–349, 1974.

[16] Golden, B. L., T. L. Magnanti y H. Q. Nguyen: *Implementing vehicle routing algorithms.* Networks, 7(2):113–148, 1977.

[17] Gurobi Optimization, L.: *Gurobi Optimizer Reference Manual*, 2020. `http://www.gurobi.com`.

[18] Jahn, J.: *Vector optimization - theory, applications, and extensions.* Springer, 2004, ISBN 978-3-540-20615-6.

[19] Karp, R. M.: *Reducibility among combinatorial problems.* En *Complexity of Computer Computations*, págs. 85–103. Springer, 1972.

[20] Khuller, S., A. Malekian y J. Mestre: *To fill or not to fill: The gas station problem.* ACM Transactions on Algorithms (TALG), 7(3):1–16, 2011.

[21] Laporte, G.: *What you should know about the vehicle routing problem.* Naval Research Logistics (NRL), 54(8):811–819, 2007.

[22] Laporte, G., H. Mercure y Y. Nobert: *An exact algorithm for the asymmetrical capacitated vehicle routing problem.* Networks, 16(1):33–46, 1986.

[23] Laporte, G., Y. Nobert y M. Desrochers: *Optimal routing under capacity and distance restrictions.* Operations Research, 33(5):1050–1073, 1985.

[24] Laporte, G. y F. Semet: *Classical heuristics for the capacitated VRP.* En *The vehicle routing problem*, págs. 109–128. SIAM, 2002.

[25] Lenstra, J. K. y A. R. Kan: *Complexity of vehicle routing and scheduling problems.* Networks, 11(2):221–227, 1981.

[26] Levy, D., K. Sundar y S. Rathinam: *Heuristics for routing heterogeneous unmanned vehicles with fuel constraints.* Mathematical Problems in Engineering, 2014, 2014.

[27] Lin, J., W. Zhou y O. Wolfson: *Electric vehicle routing problem.* Transportation Research Procedia, 12:508–521, 2016.

[28] Lin, S.: *Computer solutions of the traveling salesman problem.* Bell System Technical Journal, 44(10):2245–2269, 1965.

[29] Lin, S. H.: *Finding optimal refueling policies in transportation networks.* En *International Conference on Algorithmic Applications in Management*, págs. 280–291. Springer, 2008.

[30] Lin, S. H., N. Gertsch y J. R. Russell: *A linear-time algorithm for finding optimal vehicle refueling policies.* Operations Research Letters, 35(3):290–296, 2007.

[31] Lysgaard, J., A. N. Letchford y R. W. Eglese: *A new branch-and-cut algorithm for the capacitated vehicle routing problem.* Mathematical Programming, 100(2):423–445, 2004.

[32] Miller, C. E., A. W. Tucker y R. A. Zemlin: *Integer programming formulation of traveling salesman problems.* Journal of the ACM (JACM), 7(4):326–329, 1960.

[33] ML, J. R. F.: *A generalized assignment heuristic for the vehicle routing problem.* Networks, 11:109–124, 1981.

[34] Mole, R. y S. Jameson: *A sequential route-building algorithm employing a generalised savings criterion.* Journal of the Operational Research Society, 27(2):503–511, 1976.

[35] Pecin, D., A. Pessoa, M. Poggi y E. Uchoa: *Improved branch-cut-and-price for capacitated vehicle routing.* Mathematical Programming Computation, 9(1):61–100, 2017.

[36] Poggi, M. y E. Uchoa: *Chapter 3: New exact algorithms for the capacitated vehicle routing problem.* En *Vehicle Routing: Problems, Methods, and Applications, Second Edition*, págs. 59–86. SIAM, 2014.

[37] Poonthalir, G. y R. Nadarajan: *A fuel efficient green vehicle routing problem with varying speed constraint (F-GVRP).* Expert Systems with Applications, 100:131–144, 2018.

[38] Poonthalir, G., R. Nadarajan y S. Geetha: *Vehicle routing problem with limited refueling halts using particle swarm optimization with greedy mutation operator.* RAIRO-Operations Research, 49(4):689–716, 2015.

[39] Rodríguez, A. y R. Ruiz: *A study on the effect of the asymmetry on real capacitated vehicle routing problems.* Computers & Operations Research, 39(9):2142–2151, 2012.

[40] Schneider, M., A. Stenger y D. Goeke: *The electric vehicle-routing problem with time windows and recharging stations.* Transportation Science, 48(4):500–520, 2014.

[41] Schneider, M., A. Stenger y J. Hof: *An adaptive VNS algorithm for vehicle routing problems with intermediate stops.* OR Spectrum, 37(2):353–387, 2015.

[42] Semet, F., P. Toth y D. Vigo: *Chapter 2: Classical exact algorithms for the capacitated vehicle routing problem.* En *Vehicle Routing: Problems, Methods, and Applications, Second Edition*, págs. 37–57. SIAM, 2014.

[43] Sundar, K., S. Venkatachalam y S. Rathinam: *Formulations and algorithms for the multiple depot, fuel-constrained, multiple vehicle routing problem.* En *2016 American Control Conference (ACC)*, págs. 6489–6494. IEEE, 2016.

[44] Suzuki, Y.: *A generic model of motor-carrier fuel optimization.* Naval Research Logistics (NRL), 55(8):737–746, 2008.

[45] Suzuki, Y. y J. Dai: *Reducing the fuel cost of motor carriers by using optimal routing and refueling policies.* Transportation Journal, 51(2):145–163, 2012.

[46] Suzuki, Y. y J. Dai: *Decision Support System of Truck Routing and Refueling: A Dual-Objective Approach.* Decision Sciences, 44(5):817–842, 2013.

[47] Suzuki, Y., F. Montabon y S. H. Lu: *DSS of vehicle refueling: A new enhanced approach with fuel weight considerations.* Decision Support Systems, 68:15–25, 2014.

[48] Thompson, P. M. y H. N. Psaraftis: *Cyclic transfer algorithm for multivehicle routing and scheduling problems.* Operations Research, 41(5):935–946, 1993.

[49] Williams, H. P.: *Model building in mathematical programming.* John Wiley & Sons, 2013.

[50] Williamson, D. P. y D. B. Shmoys: *The design of approximation algorithms*. Cambridge university press, 2011.

[51] Woeginger, G. J.: *Exact algorithms for NP-hard problems: A survey*. En *Combinatorial Optimization — Eureka, You Shrink!*, págs. 185–207. Springer, 2003.