DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3

Heurísticas para el problema de enrutamiento de vehículos con capacidad.

21 de Noviembre de 2019 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Julian Ezequiel Arnesino	147/18	julianezequielarnesino@gmail.com
Pablo Manuel Grinberg	780/16	pablogrinberg@hotmail.com
Juan Junqueras	804/16	junquerasjuan@gmail.com
Tomás Caneda	490/17	tomy.3698@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - Pabellón I
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Argentina
Tel/Fax: (54 11) 4576-3359
http://exactas.uba.ar

Índice

1. Introducción al Problema

1.1. Descripción del problema.

El problema del enrutamiento de vehículos, o VRP (por sus siglas en inglés), es un problema de optimización combinatoria. Se puede pensar como una generalización del problema del viajante de comercio (TSP). En este último se desea recorrer todos los puntos dados en un espacio euclídeo exactamente una vez, minimizando la distancia recorrida. En VRP, debemos resolver el mismo problema. Las diferencias están en que VRP tiene un punto de partida dado en el mapa, y no hay un solo "viajante de comercio". Se tienen varios viajantes − desde ahora vehículos − que comienzan del punto de partida dado − desde ahora depósito.

Hay variaciones de muchos tipos para este problema. Se puede tener una flota limitada de vehículos que puede ser homogénea o no, puede haber más de un depósito, puede limitarse la distancia máxima a recorrer por vehículo y hasta se puede limitar la cantidad de puntos − desde ahora, clientes − a abastecer por cada uno.

El sub-problema abordado en el presente trabajo es el enrutamiento de vehículos con capacidad, o CVRP. Dado un conjunto de clientes con demandas asociadas, un depósito con una flota de vehículos no acotada y posiciones asociadas a todos ellos, se quiere encontrar el conjunto óptimo (de menor distancia total) de rutas para abastecer a todos los clientes desde el depósito. Los vehículos tienen todos una misma capacidad dada y los clientes no pueden tener demandas individuales que superen la capacidad de un vehículo.

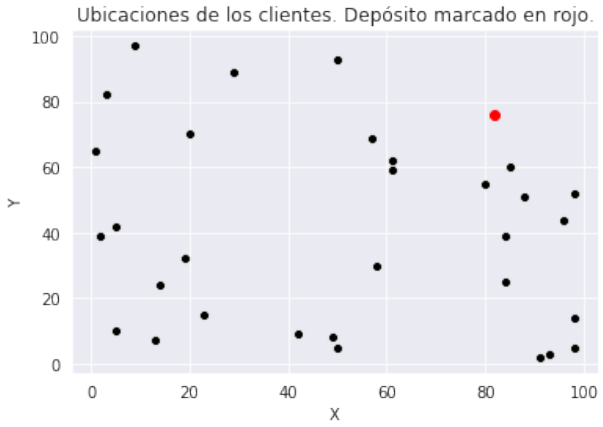
Un mismo vehículo, entonces, puede abastecer a una cantidad limitada de clientes, que será determinada por la capacidad del vehículo y las demandas de los clientes que se quiere abastecer. Como la flota de vehículos disponibles no está acotada, se permite considerar la cantidad de vehículos a usar a discreción. En todos los casos, trabajaremos sobre un espacio euclídeo bidimensional.

Está demostrado que es un problema del tipo NP-Completo, y aún no existe un algoritmo de complejidad temporal polinomial para resolverlo. Es por eso que se desarrollaron varias heurísticas y metaheurísticas para abordarlo. Éstas pueden proveer una solución cuyo costo es relativamente cercano al de la solución óptima, sin necesariamente ser el mismo. Como ventaja, las complejidades temporales de las heurísticas son polinomiales.

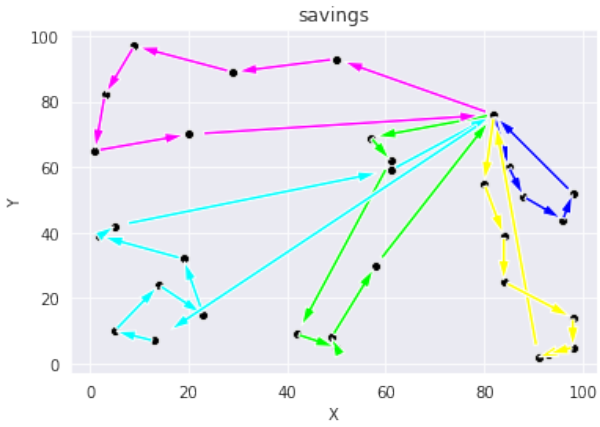
Para el desarrollo y la implementación de estas heurísticas, vamos a modelar el problema usando grafos. Tanto el depósito como los clientes serán representados por nodos. Las rutas disponibles se corresponderán a las diagonales de menor distancia euclídea conectando cada par de nodos como aristas. El grafo resultante es un grafo completo. Una solución factible será un conjunto de caminos no orientados de clientes, cada uno representando el circuito no orientado desde el depósito hacia uno de lo extremos de la ruta, y de vuelta al depósito. De todas maneras, para nuestros intereses, un conjunto de caminos orientados es, también, una solución factible, pues basta invertir el sentido de todas las aristas del conjunto para encontrar el camino en dirección opuesta.

1.2. Ejemplos y soluciones.

A continuación, se listan dos instancias del problema, con una solución factible para cada una. No son necesariamente las soluciones óptimas.

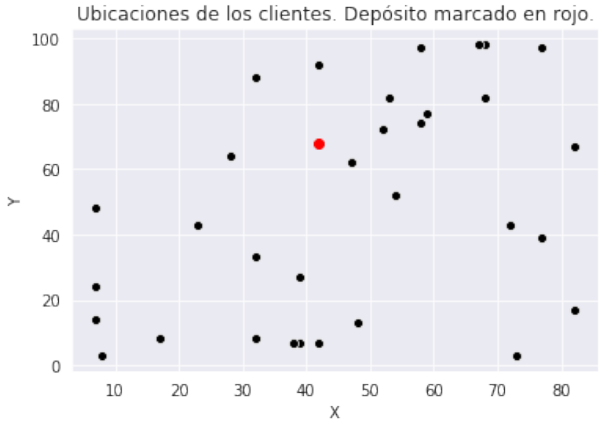


(a) Instancia del problema.

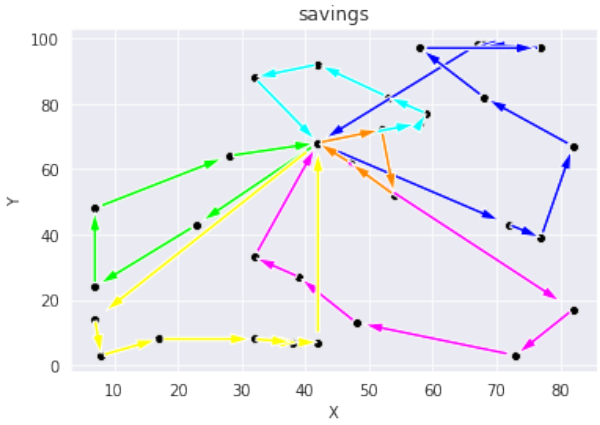


(b) Solución factible con caminos orientados.

Figura 1: Ejemplos de CVRP.



(a) Instancia del problema.



(b) Solución factible con caminos orientados.

Figura 2: Más ejemplos de CVRP.

En las figuras 1(b) y 2(b) se representan las rutas elegidas para los vehículos con distintos colores. La solución fue calculada usando una heurística explicada en las siguientes secciones que, por supuesto, no garantiza optimalidad.

1.3. Aplicaciones.

La aplicación más intuitiva es la que sugiere el nombre del problema: el enrutamiento de vehículos con capacidad. Sea una empresa de logística con sus clientes, resolver una instancia del CVRP con una heurística puede dar un plan de rutas para sus vehículos que reduzca los costos de combustible tanto como se pueda. Cada vez que se agregue un cliente, hayan nuevos requisitos por parte de un cliente existente, se cambien las rutas o se modifique la flota, habría que hacer el cálculo de nuevo.

También puede servir para una empresa recolectora de residuos. Si se piensan las cuadras de un barrio como nodos de un grafo y las intersecciones como aristas, pasar por todos los nodos con una flota de vehículos recogiendo una cantidad estimada de basura por casa es un problema que se puede

plantear como uno del tipo CVRP.

Una instalación eléctrica − o análogamente con otro servicio − es fácil de pensar como una instancia de CVRP. Se puede pensar como un problema de flujo, pero con CVRP también se puede resolver ahorrando cable y sección de cable.

Una empresa de transporte con varios destinos − paradas de colectivo, por ejemplo − que tenga las demandas de cada uno de ellos en tiempo real puede tener interés en modelar su situación diaria como instancias de este problema, resolviendo lo más rápido posible caminos poco costosos entre destinos.

Evidentemente, es un tipo de problema que no surge desde un interés exclusivamente científico, sino que se presenta constantemente en la cotidianeidad.

2. Desarrollo de Heurísticas

2.1. Heurística constructiva Nearest Neighbour. Método goloso.

2.1.1. Explicación del algoritmo.

Como se busca optimizar un recorrido, se puede pensar un algoritmo que desarrolle rutas de manera constructiva cliente por cliente, eligiendo cada vez al cliente que de el mayor y más evidente beneficio inmediato (el menor costo). Este tipo de algoritmos trabajan con un paradigma llamado *goloso* − también *greedy* por su denominación angloparlante −, y son los más intuitivamente comprensibles. En este problema, nos interesa el costo total (distancia total) de las rutas generadas y la demanda total de las mismas, evitando que supere la capacidad de un vehículo. Por eso, el algoritmo goloso que elegimos tiene ambas cosas en consideración.

La heurística constructiva golosa creada se llama *nearest neighbour* − vecino más cercano. Es una heurística constructiva porque selecciona cliente por cliente para agregar a una ruta generada. Como su nombre sugiere, partiendo desde el depósito y en cada paso realizado en la ruta se elige al cliente vecino más cercano al último de la ruta hasta ahora. Como modelamos el problema con un grafo completo, todos los clientes son vecinos entre ellos. Sin embargo, también verifica que la elección de cliente sumada a la de la ruta construida hasta el momento no supere la capacidad de un vehículo. En el momento en que la supera, el nodo elegido no se recorre en la ruta y, en su lugar, se termina la misma en el depósito. Se repiten esos pasos generando nuevas rutas hasta que no haya clientes pendientes por visitar.

Algoritmo 1: Nearest neighbour.

Data: Grafo euclídeo completo (clientes y depósito), demandas de clientes y capacidad de vehículos.
Result: Lista de rutas que abastecen a todo cliente.

1 Comenzar desde el depósito e ir al cliente más cercano e éste. Marcarlos como visitados.; /* $\mathcal{O}(n)$ */
2 Mientras haya clientes sin visitar, repetir el siguiente proceso:; /* $\mathcal{O}(n)$ iteraciones. */
3 **1.** Elegir el cliente **p** más cercano a la posición actual entre los no visitados. ; /* $\mathcal{O}(n)$ */
4 **2. Si** la demanda de **p** sumada a la demanda total de la ruta es menor o igual que la capacidad de un vehículo: ; /* $\mathcal{O}(1)$ */
5 - Marcar **p** como posición actual y como cliente visitado. Sumar su demanda a la demanda total de la ruta. ; /* $\mathcal{O}(1)$ */
6 **Sino:** ; /* $\mathcal{O}(1)$ */
7 - No ir a **p**, sino al depósito. Terminar la ruta actual. ; /* $\mathcal{O}(1)$ */
8 **3.** Si se revisaron todos los clientes (la cantidad de clientes visitados es igual a la cantidad total), entonces termina el algoritmo. ; /* $\mathcal{O}(1)$ */
9 Return **Lista de rutas.**

Como se aprecia en la facilidad del pseudocódigo, es un algoritmo simple e intuitivo. Esta es una de las ventajas de los algoritmos golosos en general.

2.1.2. Complejidad temporal en el peor caso.

Hay muchos pasos del algoritmo que son reducibles a una cantidad constante de operaciones elementales. Sin embargo, los pasos de las líneas 1, 2 y 3 no respetan ese esquema.

En la línea 1, buscamos el cliente no visitado con distancia mínima a la posición actual. Como se saben las posiciones de todos los clientes y es posible averiguarlas en tiempo constante según la implementación de grafos utilizada, hacer el cálculo de distancia euclídea para los $n-1$ clientes que pueden no ser visitados y encontrar el que resulte con la mínima es una operación realizable en, a lo sumo, n iteraciones.

En la línea 2, hay un ciclo que se repite mientras haya clientes sin visitar. Manteniendo un contador de clientes visitados en los pasos de las líneas 1 y 5, se ve si se visitaron todos los clientes con una operación elemental, comparando el contador de clientes visitados con la cantidad total de los mismos. En el peor de los casos, el algoritmo elige una ruta por cada cliente. Es decir que, a lo sumo, recorre una vez la línea 7 por cada vez que recorre la línea 5. En ese caso, hay $\mathcal{O}(n)$ iteraciones del ciclo.

Por último, la línea 3 repite la operación de la línea 1 en cada una de las iteraciones del ciclo de la línea 2.

La complejidad temporal total en el peor caso es, con un poco de abuso de notación, la siguiente:

$$\mathcal{O}(n) + \mathcal{O}(n) * (\mathcal{O}(n) + \mathcal{O}(1)) = \boxed{\mathcal{O}(n^2)}$$

2.1.3. Instancias poco adecuadas.

En casi toda instancia, la solución heurística golosa presentada resultará en un conjunto de rutas de mayor costo que el óptimo.

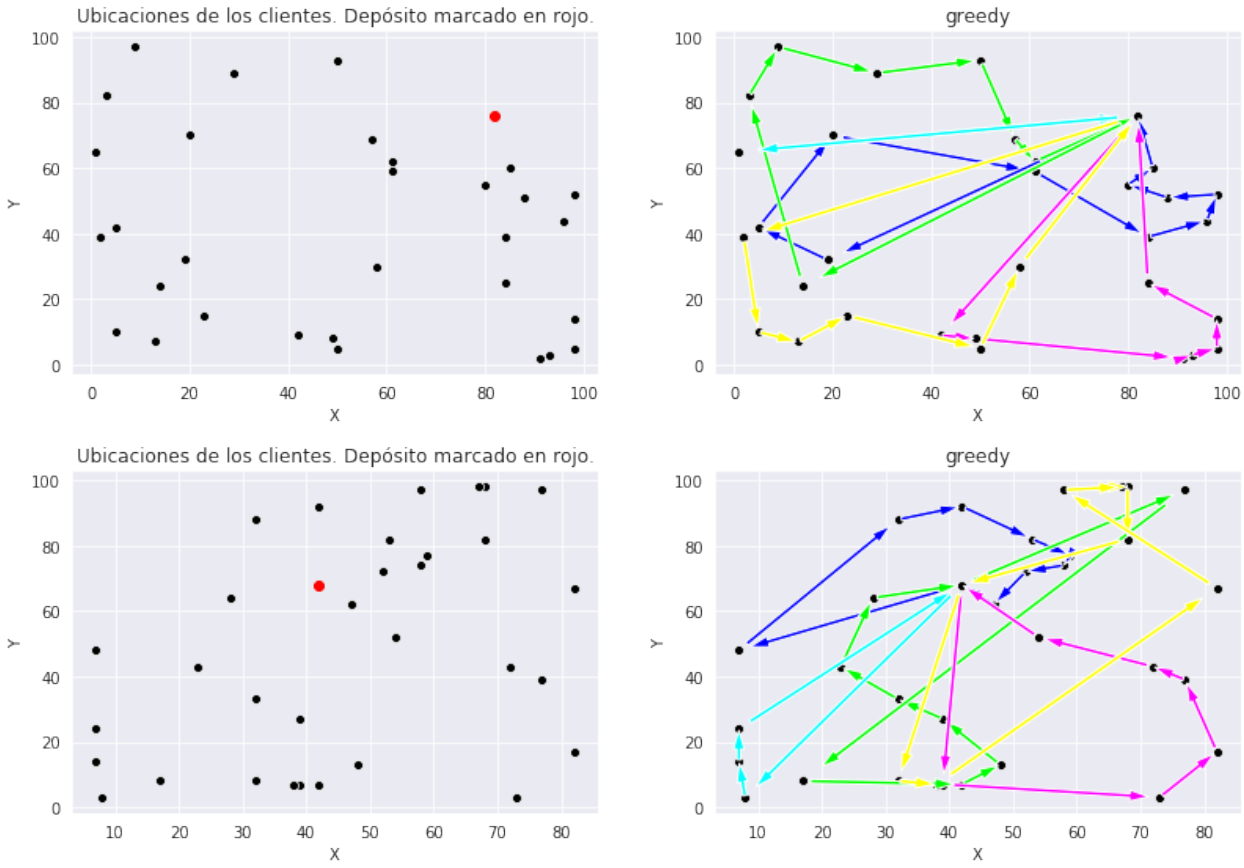


Figura 3: Rutas para las instancias graficadas.

Para ambas instancias de la figura 3, por ejemplo, la heurística Greedy no provee la solución óptima. El costo de la solución encontrada es de 1133 unidades, mientras que en la solución óptima es de 784 unidades. Para la segunda instancia, la solución generada y la óptima tienen costos de 965 y 661, respectivamente.

2.2. Heurística constructiva Savings.

Dado que en el contexto de este trabajo el valor de una arista cualquiera equivale a la distancia euclídea entre sus extremos, la heurística de *savings* consiste en el aprovechamiento de un conveniente teorema de la geometría euclidiana: la desigualdad triangular. La misma establece, en términos vulgares, que **la suma de las longitudes de dos lados de un triángulo es mayor a la longitud del lado restante**.

Como consecuencia de lo antedicho, cualquier par de rutas $R_1 = (d, v_1, ..., v_n, d)$ y $R_2 = (d, w_1..., w_m, d)$ (con d representando el único depósito y $v_{i/1\leq i\leq n}, w_{j/0\leq j\leq m}$ los vértices o clientes internos a las rutas R_1 y R_2 respectivamente) puede unirse a través de sus extremos (los vértices vecinos a los depósitos en las rutas) y **reducir la longitud total recorrida sin perder ningún cliente**. Concretamente, dado que la rutas R_1 y R_2 tienen a lo sumo dos extremos v_1, v_n y w_1, w_m respectivamente, hay cuatro formas de conectar las rutas a través de ellos, como se ilustra en la figura a continuación[1]:

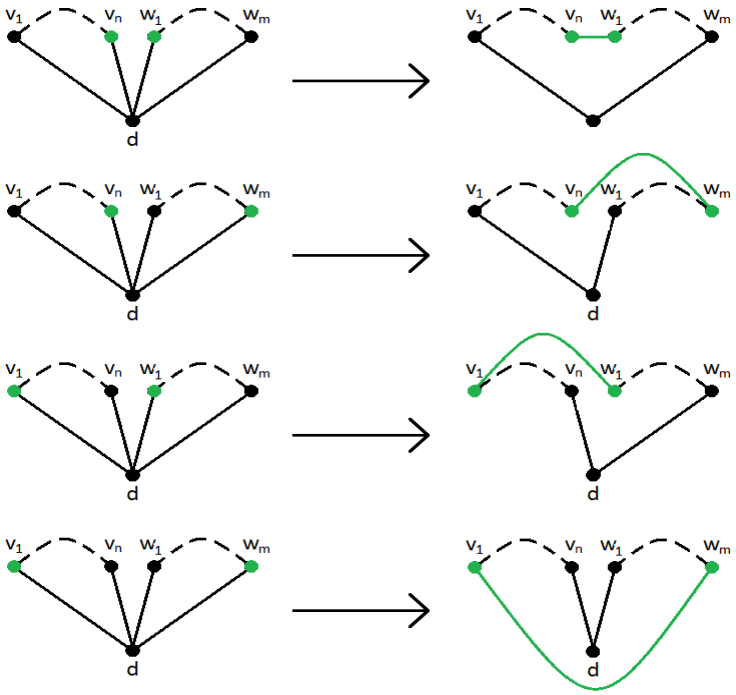


Figura 4: Posibles conexiones de dos rutas con dos extremos

En cada caso, del triángulo definido por los tres puntos d, e_{R_1} y e_{R_2} (siendo e_{R_i} algún extremo de la ruta R_i) se quitan los lados (d, e_{R_1}) y (d, e_{R_2}) originalmente presentes para reemplazarlos por (e_{R_1}, e_{R_2}), **de longitud menor a la suma de los removidos**[2], logrando así llegar a los mismos clientes recorriendo una menor distancia. Resulta entonces que mediante este procedimiento se consigue un *saving* o *ahorro* -para cada par de rutas que lo permitan[3]- de valor $s = \delta(d, e_{R_1}) + \delta(d, e_{R_2}) - \delta(e_{R_1}, e_{R_2}) > 0$ (donde $\delta(a, b)$ representa la distancia euclídea entre los puntos a y b), así denominado pues esta mag-

[1] El caso en que ambas rutas tienen un sólo nodo interno (y en consecuencia un sólo extremo cada una) se resuelve de forma idéntica, considerando que hay una sola conexión posible.

[2] Tener en cuenta que, de la Figura 1, sólo la conexión mostrada al tope de la misma representa con fidelidad la longitud del lado (e_{R_1}, e_{R_2}) mediante la línea verde, pues para el resto se optó por el uso de un arco del mismo color que cubre una mayor distancia que la necesaria para facilitar la lectura de la imagen.

[3] Ya que dos rutas pueden conectarse sólo si el camión que debe recorrer su conexión tiene la capacidad para hacerlo.

nitud positiva puede sustraerse a la suma de las longitudes de R_1 y R_2[4] (distancia que se cubre si se recorren estas rutas por separado) para llegar a una longitud **menor** correspondiente a la ruta que las conecta por sus extremos (como se muestra en la Figura de arriba).

Como se verá a continuación, el algoritmo que implementa esta heurística consistirá en la creación de rutas iniciales simples que cubran todos los clientes, para luego combinarlas como se mostró anteriormente (en tanto no se superen las capacidades de los camiones) con el propósito de reducir las distancias recorridas una cantidad determinada de veces [5].

2.2.1. Explicación del algoritmo.

El algoritmo que implementa la heurística de *savings* sigue un esquema más bien simple:

1. Se crea una serie de rutas iniciales que cubren todos los clientes disponibles. En este trabajo esto se logrará, si los n clientes se representan con la secuencia $< v_1, v_2, ..., v_n >$, mediante la generación de n rutas básicas $R_1 =< d, v_1, d >,\ R_2 =< d, v_2, d >,\ ...,\ R_n =< d, v_n, d >$, cada una de las cuales visita a un cliente distinto partiendo del depósito para luego volver a él. Es necesario también que cada ruta inicial R_i (con $1 \leq i \leq n$) tenga acceso a su longitud, que será $L(R_i) = \delta(d, v_i) + \delta(v_i, d) = 2 * \delta(d, v_i)$ (donde, como ya fue mencionado previamente, $\delta(a, b)$ representa la distancia euclídea entre los puntos a y b, por lo que $\delta(a, b) = \delta(b, a)$).

2. Para cada par de nodos (clientes) distintos se calcula el *saving* correspondiente a una conexión a través de ellos, asumiendo que ambos son extremos de las rutas en que se encuentran (esto es, que ambos tienen como vecino al depósito). La fórmula para el *saving* entre los clientes v_i y v_j (siendo $i \neq j$) es $s_{ij} = \delta(d, v_i) + \delta(d, v_j) - \delta(v_i, v_j)$[6]. Estos valores se almacenarán en un vector al que se denominará `savings`.

3. Se ordena el vector `savings` de mayor a menor, de modo que puedan accederse en las primeras posiciones a los *ahorros* de mayor valor.

4. Se recorre el vector `savings` desde su primer elemento hasta el último. Si para un *saving* determinado s_{ij}[7] se cumple que:

   I Los clientes v_i y v_j pertenecen a rutas distintas,

   II la capacidad de un vehículo no es excedida si los clientes de las rutas de v_i y v_j se combinan en una sola, y

   III ambos v_i y v_j son extremos de sus respectivas rutas (son vecinos del depósito),

   entonces las rutas de v_i y v_j se conectan a través de ellos, con la longitud de la ruta resultante equivalente a la suma de las longitudes de susodichas rutas menos el valor del *saving* correspondiente (s_{ij}).

[4] Lo cual es consistente con lo planteado previamente: al restar s a la suma de las longitudes de R_1 y R_2 se sustrae la longitud de los lados (d, e_{R_1}) y (d, e_{R_2}) (pues se quitan), mientras que se suma la del lado (e_{R_1}, e_{R_2}) (pues se agrega). El resto de las aristas que componen las rutas se conservan inalteradas al conectarlas (como puede comprobarse en la Figura 1), y por ende la operación antedicha no las involucra.

[5] Es importante aclarar que el paper de Clarke y Wright del que origina esta heurística de *savings* da en realidad una perspectiva más amplia y general de la técnica que no se ve reflejada en este trabajo, el cual se limita a ofrecer sólo una adaptación de la misma.

[6] Observar que $s_{ij} = s_{ji}$, por lo que será innecesario calcular el segundo si ya se conoce el valor del primero (y viceversa).

[7] Aquí los subíndices no aluden a un orden dentro del vector, sino que se utilizan para identificar a los clientes involucrados en el *ahorro* en cuestión.

5. Una vez recorrido el vector `savings` en su totalidad, se registran las rutas conseguidas a lo largo del procedimiento junto a sus longitudes para poder eventualmente imprimirlas por salida estándar.

Es relevante aclarar que el paso 3 ordena el vector `savings` de mayor a menor para que los primeros *ahorros* que se apliquen sean los más beneficios (esto es, para que se prioricen las conexiones entre rutas que más longitud pierden). Colocarlos en cualquier otra posición menos prioritaria incrementaría el riesgo de que estos mejores *ahorros* no puedan aplicarse, pues a medida que el algoritmo avanza más son los caminos combinados y menores las chances de que la capacidad de los vehículos no sea excedida por ellos, o de que las rutas de dos clientes sean distintas (ambas condiciones necesarias para poder hacer efectivo un *saving*, como se mostró anteriormente).

A continuación se presenta el pseudocódigo que implementa la heurística como fue descripta previamente:

Algoritmo 2: Savings

Data: Grafo euclídeo completo (clientes y depósito) G, demandas de clientes d y capacidad de vehículos c.
Result: Lista de rutas que abastecen a todo cliente.

```
1  n ← |V(G)|, Rutas ← ∅ ;                                                    /* O(1) */
2  for cada v_i ∈ V(G) do                                                     /* O(n) */
3  |  Rutas ∪ {< d, v_i, d >} ;                                               /* O(1) */
4  end
5  savings ← M_{n×n}/M[i][j] = −1  ∀(0 ≤ i, j < n) ;                          /* O(n^2) */
6  for todo i ∈ [0, n − 1] do                                                 /* O(n) */
7  |  for todo j ∈ [i + 1, n − 1] do                                          /* O(n) */
8  |  |  savings[i][j] ← δ(d, v_i) + δ(d, v_j) − δ(v_i, v_j) ;               /* O(1) */
9  |  end
10 end
11 OrdenarMatrizMayorAMenor(savings) ;  /* O(n^2 * log(n^2)) = O(n^2 * 2 * log(n)) = O(n^2 * log(n)) */
12 for todo i ∈ [0, n − 1] do                                                 /* O(n) */
13 |  for todo j ∈ [0, n − 1] do                                             /* O(n) */
14 |  |  s_ij ← savings[i][j] ;                                              /* O(1) */
15 |  |  if s_ij ≠ −1 then                                                   /* O(1) */
16 |  |  |  R_i ← RutaDe(v_i, Rutas), R_j ← RutaDe(v_j, Rutas) ;   /* O(n) amortizado. */
17 |  |  |  D(R_i) ← DemandaTotalDe(R_i, d), D(R_j) ← DemandaTotalDe(R_j, d) ;   /* O(n) */
18 |  |  |  if R_i ≠ R_j ∧ c ≥ D(R_i) + D(R_j) ∧ (EsExtremo(v_i, R_i) ∧ EsExtremo(v_j, R_j) then
   |  |  |   /* O(n) */
19 |  |  |  |  UnirRutasPorExtremos(Rutas, v_i, v_j) ;   /* O(n) amortizado. */
20 |  |  |  end
21 |  |  end
22 |  end
23 end
24 return Rutas ;
```

Observar que se decidió representar al vector `savings` como una matriz (cuyas celdas inicialmente tienen el valor -1[8]) para simplificar el código. Por otra parte, se utilizan las siguientes funciones:

- *OrdenarMatrizMayorAMenor(matriz)*, que ordena la *matriz* (pasada por referencia) como si estuviera desplegada en forma de vector (leída de izquierda a derecha y desde arriba hacia abajo).

[8]Esto permite al código completar la matriz sólo desde su diagonal (que conecta el extremo superior izquierdo con el inferior derecho) hacia arriba, pues dicha matriz de *savings* es simétrica al cumplirse la propiedad que $s_{ij} = s_{ji}$. Las celdas de la mitad inferior de la matriz quedan entonces con valor -1, lo cual más tarde resulta conveniente para poder identificarlas y consecuentemente ignorarlas (el ordenamiento de mayor a menor que se lleva a cabo no cambia este hecho).

Esto puede lograrse transformando dicha *matriz* a vector en $O(n^2)$, ejecutando un algoritmo de ordenamiento como `sort` de la STL de C++ en $O(n^2 * log(O(n^2)))$, y finalmente copiando el resultado hacia la *matriz* pasada como argumento en $O(n^2)$ nuevamente.

- *RutaDe(cliente,rutas)*, que devuelve la ruta de *rutas* en que el *cliente* se encuentra presente. Esto puede lograrse recorriendo todas los elementos presentes en *rutas* hasta hallar el buscado, lo cual toma tiempo *O(n)* amortizado, puesto que entre todos las rutas presentes en el conjunto aparecen todos los n clientes del grafo, ni más ni menos.

- *DemandaTotalDe(ruta, demandas)*, que calcula la suma de las demandas de los clientes presentes en la *ruta*. Se determina en *O(n)* recorriendo toda la ruta cliente por cliente y accediendo en *O(1)* a la demanda correspondiente a cada uno de ellos en el vector de *demandas*.

- *esExtremo(cliente,ruta)*, que determina si el *cliente* es extremo (vecino del depósito) en la *ruta*. Se logra en *O(n)* recorriendo toda la *ruta*, aunque si está implementada como vector puede reducirse a *O(1)* evaluando sólo a los vecinos del primer y último elemento (ya que ambos representan al depósito).

- *UnirRutasPorExtremos(rutas,cliente1,cliente2)*, que combina las rutas en que se encuentran presentes *cliente1* y *cliente2* a través de ellos, reflejando los cambios en el conjunto de *rutas*. Esto es, primero se identifica por medio de los clientes a las dos rutas individuales pasadas por parámetro en dicho conjunto, lo cual demora *O(n)* amortizado (empleando la misma metodología que la función *RutaDe*) para luego borrarlas (en a lo sumo *O(n)*) y agregar la ruta combinada en *rutas*, que se obtiene en tiempo *O(n)*.

Tener en cuenta que estas funciones se proveen sólo como una conveniencia para abstraer el pseudocódigo y evitar el uso de objetos más complejos como punteros y estructuras de datos personalizadas. Como consecuencia de esta medida, el procedimiento presentado en esta sección **no refleja con alta fidelidad el algoritmo implementado en lenguaje C++** (`savings.cpp`), incluido en la entrega de la que forma parte este informe.

Como comentario final, notar que esta heurística constructiva genera únicamente soluciones factibles (proporciona rutas válidas que parten de y vuelven al depósito y que pueden ser recorridas por los vehículos sin superar su capacidad), ya que antes de unir rutas por sus extremos se evalúa que se cumplan las tres condiciones necesarias para hacerlo sin perder factibilidad.

2.2.2. Complejidad temporal en el peor caso.

Como se evidencia en el pseudocódigo provisto en la subsección previa, la complejidad en peor caso del algoritmo *Savings* puede determinarse fácilmente sumando las complejidades de todas las operaciones y ciclos involucrados, que aparecen comentadas a la derecha de dicho código:

$$O(1) + O(n) * O(1) + O(n^2) + O(n) * O(n) * O(1) + O(n^2 log(n)) + O(n) * O(n) * O(n) =$$

$$O(n) + O(n^2) + O(n^2) + O(n^2 log(n)) + O(n^3) = \boxed{O(n^3)}$$

La implementación en lenguaje C++, `savings.cpp`, comparte la complejidad temporal con el algoritmo *Savings*. Esto se debe, tal como ocurre en el pseudocódigo, al ciclo que itera sobre los elementos del vector `savings`, puesto que para cada uno de los $O(n^2)$ de ellos se puede ejecutar la inversión de alguna de las rutas, de complejidad $O(n)$, lo que resulta en una complejidad total $O(n^2) * O(n) = \boxed{O(n^3)}$. El resto de los ciclos no demoran más que $O(n^2)$, y la única operación que

destaca es el ordenamiento del vector `savings`, lo cual nuevamente demora $O(n^2 * log(n))$, pero todos estos otros costos son subsumidos por el ciclo de complejidad $O(n^3)$ mencionado al principio.

2.2.3. Instancias poco adecuadas.

Esta heurística resulta beneficiada cuanto mayores son los valores de los *ahorros* o *savings*, pero visualizar las instancias que resultan en soluciones de baja calidad no es tan trivial como la simpleza del algoritmo sugiere. Quizás resulte entonces más fácil hipotetizar instancias que produzcan buenas soluciones, para luego tomar medidas opuestas a las necesarias para llegar a ellas. Este ejercicio podría no conducir a las peores instancias, pero resulta razonable para alejarse de las mejores (o buenas).

Se propone entonces que las instancias que darán soluciones de buena calidad serán aquellas en que los clientes estén alejados del depósito (d) pero cerca entre sí. Con esta configuración, resulta sensato pensar que los valores de los *savings* calculados por el algoritmo serán relativamente altos, ya que los triángulos presentes en el grafo definidos por los puntos d, v_i y v_j (clientes cualquiera estos últimos dos) tendrán un lado (v_i, v_j) muy pequeño comparado a los otros dos, (d, v_i) y (d, v_j). De este modo se maximiza la desigualdad triangular, y con ella el valor del *saving* asociado a los clientes en cuestión: $s_{ij} = \delta(d, v_i) + \delta(d, v_j) - \delta(v_i, v_j))$, que ya se conocía era positivo. Si $\delta(d, v_i), \delta(d, v_j) >> \delta(v_i, v_j)$, entonces $s_{ij} >> 0$ y por ende **mayor el valor del *ahorro* al unir rutas a través de estos clientes**.

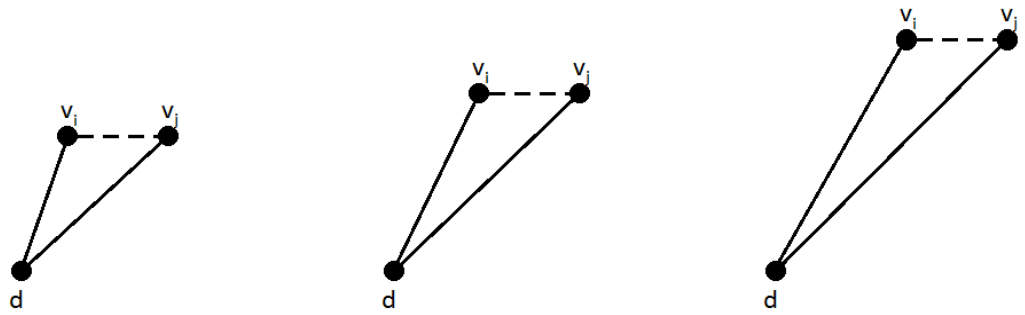


Figura 5: Ejemplos de triángulos generados entre dos clientes y el depósito. La distancia entre v_i y v_j (longitud de la línea punteada) permanece inalterada en los tres casos. s_{ij} crece al aumentar la distancia entre el depósito y los clientes, por lo que será máximo en el triángulo a la derecha.

Considerando el análisis previo puede sugerirse que, **de reducirse la distancia entre el depósito y los clientes, es probable que las soluciones conseguidas sean de una calidad menor**, puesto que los *savings* que serán calculados no serán tan grandes. Puede tomarse luego como hipótesis que **los grafos que presenten distancias pequeñas entre los clientes y el depósito serán parte del cuerpo de instancias poco adecuadas**.

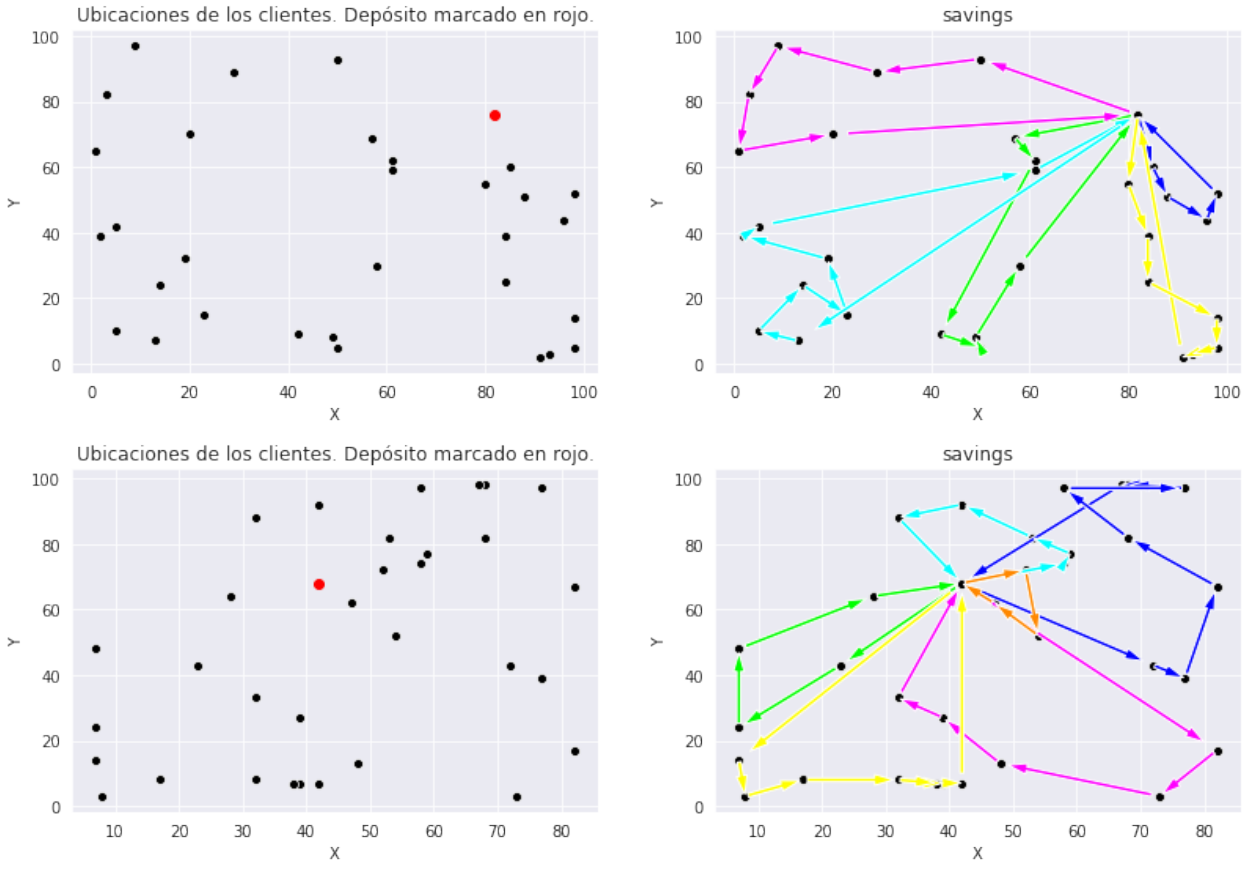


Figura 6: Rutas para las instancias graficadas.

En la figura 6 presentamos dos instancias tales que la aplicación de la heurística Savings no resulta en una ruta óptima. En la primera, la solución generada es de 1123, contra una óptima de 784. En la segunda, la solución generada cuesta 792 unidades, contra 661 de la óptima.

2.3. Heurística por cluster first (Sweep), route second (Nearest to depot).

Las heurísticas por *cluster first, route second* consisten en separar en grupos los nodos del grafo y luego aplicar otra heurística a cada uno de esos grupos.

La idea del *clustering* por **sweep** es agrupar los nodos de manera que cada cluster contenga nodos que se encuentren en un mismo camino *lógico* desde el depósito. Para definir estos clusters, se toma como criterio el ángulo de cada nodo con respecto a un cero arbitrario sobre el cual está el depósito.

De esta manera, aunque dos nodos estén a gran distancia entre sí, si sus ángulos con respecto al depósito son similares, es razonable pasar por uno de camino al otro.

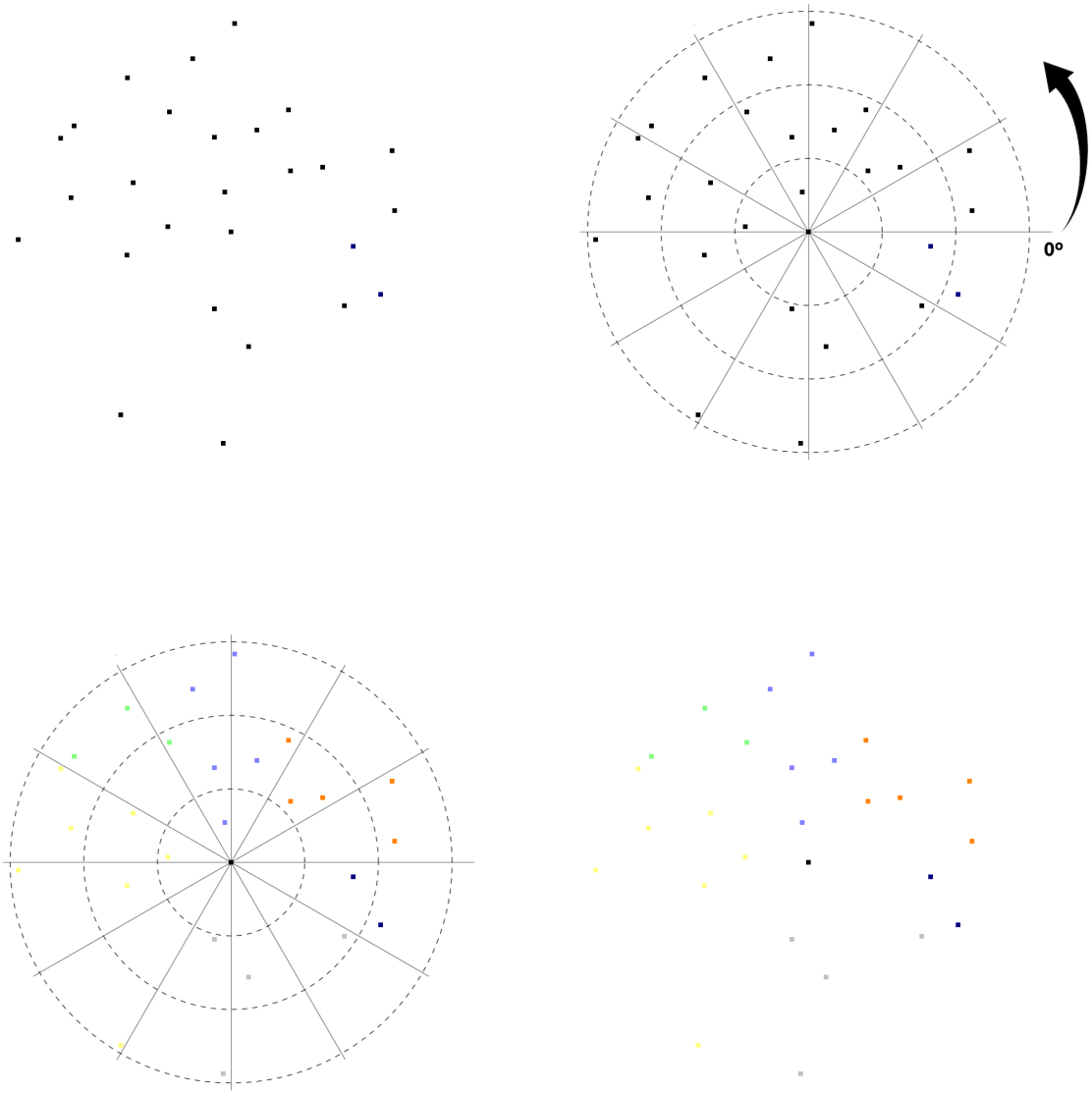


Figura 7: Proceso de clustering por sweeping

Una vez que todos los clusters están definidos, para cada uno se aplica la heurística de *nearest to*

depot, que consiste en recorrer los nodos del cluster crecientemente respecto a su distancia al depósito.

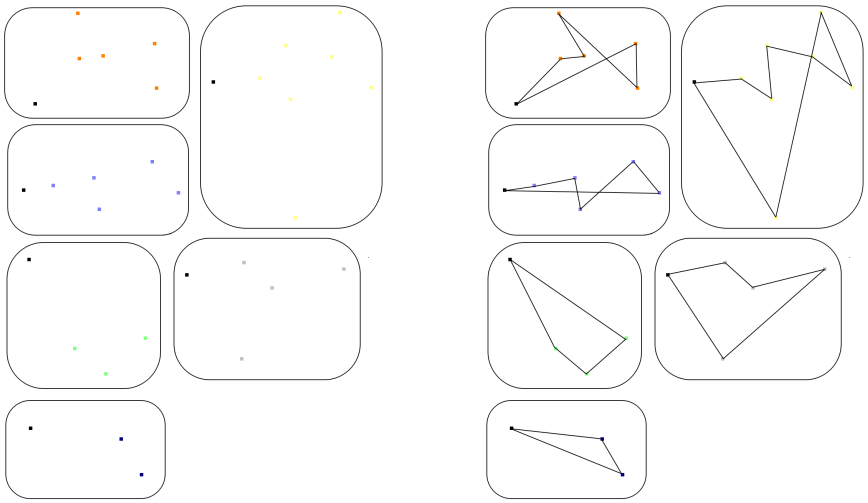


Figura 8: Proceso de routing

2.3.1. Explicación del algoritmo.

El primer paso del algoritmo es el ordenamiento de los nodos según el criterio antes mencionado. Para esto se calcula el θ de sus coordenadas polares. Es necesario "preprocesar"la información para que considerar el depósito como la coordenada $(0, 0)$, lo cual se logra restando a todas las coordenadas los valores de las del depósito.

Luego de ordenar los nodos de manera creciente según el θ se realiza el barrido circular, de la siguiente manera: se recorren los nodos, ordenados, y se los va agrupando en un cluster mientras la demanda acumulada no sobrepase la capacidad de los camiones. Cuando se llega a esta situación, el cluster se cierra, y se comienza a formar uno nuevo.

Al finalizar el barrido, si quedaron nodos en un cluster que todavía tiene capacidad, el cluster se cierra también. Es importante destacar que se incluye al depósito en todos los clusters.

De esta manera queda dividido el problema original en subproblemas equivalentes al de viajante de comercio, porque se elimina el factor de la capacidad: por cómo fueron construidos los clusters, sabemos que un camión alcanza para recorrer cada uno íntegramente.

En este punto se aplica otra heurística, una golosa de cercanía al depósito. Para cada cluster, se calcula la distancia euclídea del depósito a cada nodo, y se ordenan los nodos de manera creciente según esta. Este ordenamiento es equivalente a una ruta.

Combinando la solución de cada cluster se obtiene una lista de rutas que es la solución al problema original.

Algoritmo 3: Sweep

```
1  rutas ← ∅
2  angulos ← ∅
3  depósito ← G.nodes.first
4  for todo nodo que no es depósito do
5  |  angulo ← anguloPolar(nodo.X − depósito.X, nodo.Y − depósito.Y)
6  |  angulos.add(nodo, angulo)
7  end
8  sort(angulos)
9  clusters ← ∅
10 cluster ← ∅
11 demandasCluster ← ∅
12 demandaCluster ← 0
13 for a en angulos do
14 |  if clusterDemanda + demandas[a.nodo.id] > c then
15 |  |  cluster.addFirst(depósito)
16 |  |  clusters.add(cluster)
17 |  |  cluster ← ∅
18 |  |  demandasCluster ← ∅
19 |  |  demandaCluster ← 0
20 |  end
21 |  cluster.add(a.node)
22 |  demandasCluster.add(demandas[a.nodo.id])
23 |  demandaCluster+ = demandas[a.nodo.id]
24 end
25 if cluster == ∅ then
26 |  cluster.addFirst(depósito)
27 |  clusters.add(cluster)
28 |  cluster ← ∅
29 |  demandasCluster ← ∅
30 |  demandaCluster ← 0
31 end
32 for cluster en clusters do
33 |  ruta ← nearestTodepósito(cluster)
34 |  rutas.add(ruta)
35 end
36 return rutas
```

2.3.2. Complejidad temporal en el peor caso.

El cálculo de coordenadas polares tiene un costo de $\mathcal{O}(1)$, y recorrer los nodos cuesta $\mathcal{O}(n)$. Ordenarlos tiene un costo de $\mathcal{O}(n \cdot logn)$. Iterar sobre los ángulos para crear los clusters también cuesta $\mathcal{O}(n)$, y luego aplicar nearest to depot tiene un costo de $\mathcal{O}(k \cdot logk)$ donde k es la cantidad de nodos del cluster correspondiente.

Si hay z clusters, la complejidad de aplicar esa heurística sobre todos ellos se puede calcular como $\sum_{i=0}^{z} |C_i| \cdot log(|C_i|)$, con $\sum_{i=0}^{z} |C_i| = n$. Es decir que es $\mathcal{O}(n) \cdot \sum_{i=0}^{z} log(|C_i|)$. Por propiedades de logaritmos, podemos decir que la sumatoria es igual a $log(\prod_{i=0}^{z}(|C_i|))$, lo cual está acotado por $log(n^n)$, que es $\mathcal{O}(n \cdot logn)$.

Entonces la complejidad total del algoritmo es $\mathcal{O}(n \cdot logn)$.

2.3.3. Instancias poco adecuadas.

Es posible que dos nodos que están muy cerca entre sí (y que sería lógico visitar consecutivamente) queden en clusters distintos, porque luego de agregar el primero, la capacidad restante del camión es menor que la demanda del segundo nodo.

Además es necesario considerar las instancias poco adecuadas para la heurística golosa. Por ejemplo, si un nodo a está cerca de otro nodo b, ambos alejados de uno nodo c, y la distancia al depósito de c es mayor a la de a y menor a la de b, la ruta devuelta por este algoritmo sería $a \to c \to b$, aunque solución óptima sería $a \to b \to c$.

Sin embargo, dado que, suponiendo distribución uniforme de coordenadas y de demandas, a medida que la cantidad de nodos crece, es lógico concluir que el rango angular comprendido por cada cluster se achique, por lo que estas situaciones se vuelven cada vez más infrecuentes (si el ángulo en el que están todos los nodos con respecto al depósito es chico, es razonable pensar que si dos nodos tienen una distancia similar al depósito, la distancia entre ellos es pequeña).

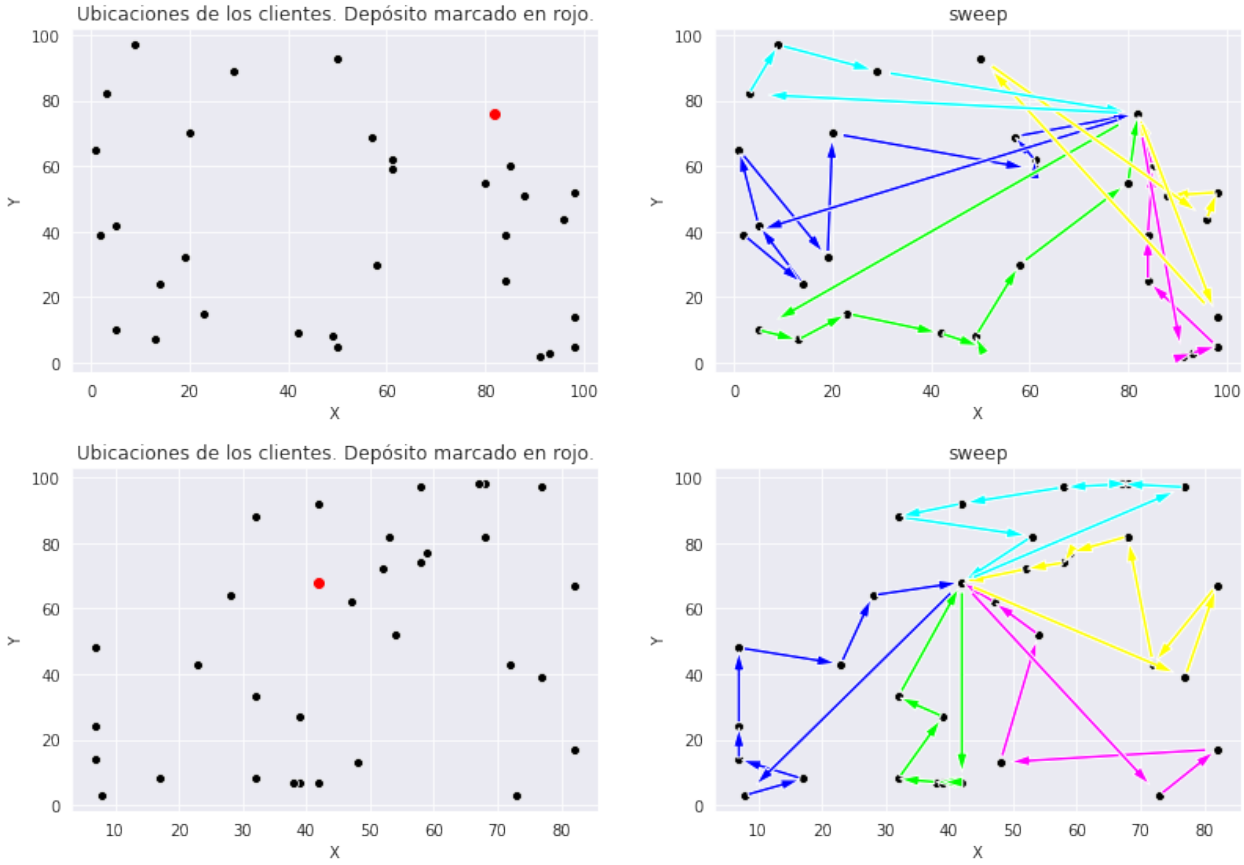


Figura 9: Rutas para las instancias graficadas.

En la figura 9 vemos como el algoritmo genera rutas para las instancias presentadas. En el primer caso, la solución generada cuesta 852 unidades, contra 784 de la óptima. En el segundo, la solución generada cuesta 758 unidades, contra 661 que cuesta la óptima.

2.4. Heurística por cluster first (K-Means), route second (Nearest Insertion).

2.4.1. Explicación del algoritmo.

Una alternativa a la heurística anterior es una es otra que agrupa los nodos con el método K-Means, y luego resuelve cada grupo por separado con el método Nearest Insertion.

El método K-means de clusterización se basa en el uso de puntos distribuidos aleatoriamente en el espacio de clusterización. Se generan K de estos puntos, llamados **medias** (o means), lo que le da el nombre al algoritmo. La cantidad de puntos que se generan será correspondiente a la máxima cantidad de clusters que resultarán del algoritmo. Como una primera estimación, el valor de K será la suma total de demandas sobre la capacidad de un vehículo. Luego se calcula, para cada nodo, la media más cercana. Se cambia la posición de la media a la posición promedio de sus nodos cercanos. Se repite este proceso una cantidad fija de veces y, al terminar, un conjunto de nodos que tienen la misma media más cercana componen un cluster. A priori, las medias pueden generarse en una posición inicial aleatoria, y es es lo que hacemos en nuestra implementación.

Como el problema se basa en ruteo de vehículos con capacidad, y la idea es resolver un cluster como una sola ruta, hay que asegurarse de que la demanda total de un cluster no supere la capacidad de un vehículo. Para eso, los clusters que sí la superen se separan de manera golosa, tal que cada uno pueda ser abastecido por un sólo vehículo.

Para cada cluster, se piensa su ruta como una instancia del problema del viajante de comercio, buscando un camino hamiltoniano en el mismo (o un circuito hamiltoniano, considerando el depósito). La heurística que empleamos para esto es Nearest Insertion. Se comienza con un camino de de dos ciudades, una aleatoria y luego la más cercana a ella. Del camino obtenido elegimos un par (k,j) de ciudades tal que k no pertenezca al camino y j sí, y su distancia sea la menor de todos tales pares. Seleccionamos la ciudad i del camino que minimiza el costo de insertar k entre i y j, y se hace la inserción. Este proceso de selección e inserción se repite mientras haya ciudades fuera del camino.

Una vez resuelto cada cluster, se tiene un conjunto de rutas cuyo cardinal será la cantidad de camiones empleados y que, entre todas sus rutas, abastecen a todos los clientes respetando la capacidad de los vehículos. A continuación, el pseudocódigo de la heurística.

Algoritmo 4: Cluster first (K-Means), route second (Nearest Insertion).

Data: Grafo euclídeo completo (clientes y depósito), demandas de clientes y capacidad de vehículos.
Result: Lista de rutas que abastecen a todo cliente.

`/* Clusterizar con K-Means (da resultados similares pero diferentes cada vez, por ser randomizado). */`

1 Estimar K como la suma total de demandas sobre la capacidad de un camión.; `/*` $\mathcal{O}(n)$ `*/`
2 Inicializar K puntos llamados "means.[en] el espacio aleatoriamente. Se debe respetar el rango de los puntos a clusterizar.; `/*` $\mathcal{O}(K)$ `*/`
3 Repetir el siguiente proceso una cantidad fija de veces:; `/* Iteraciones` $\in \mathcal{O}(1)$ `*/`
4 **1.** Calcular, para cada punto a clusterizar, a qué mean es más cercano.; `/*` $\mathcal{O}(n * K)$ `*/`
5 **2.** Actualizamos las posiciones de las means como la media de las posiciones de los puntos que eran más cercanos a él.; `/*` $\mathcal{O}(K)$ `*/`
6 El resultado serán K clusters. El cluster al que pertenece cada punto es la mean al que es más cercano.
7 Si algún cluster tiene más demanda total que la que puede abastecer un camión, separarlo en clusters de manera golosa.; `/*` $\mathcal{O}(n)$ `*/`

`/* Resolver los problemas tipo TSP restantes con la heurística de nearest insertion. Llamo d(k,j) a la distancia entre dos ciudades k y j. Iteraciones` $\in \mathcal{O}(K)$ `*/`

8 Comenzar con un camino de dos ciudades. Una aleatoria y luego la más cercana a ella.; `/*` $\mathcal{O}(n)$ `*/`
9 Para cada cluster, repetir el siguiente proceso mientras falten clientes:; `/*` $\mathcal{O}(n^2)$ `amortizado. */`
10 **1.** Elegir el par de ciudades (k,j) tal que:;
11 - La ciudad k no pertenezca al camino, y j sí.
12 - Se minimiza d(k,j) entre todos los pares.
13 **2.** Encontrar el vecino i de j tal que (i,j) en el camino tal que se minimiza (d(k,i) + d(k,j) - d(i,j)). ; `/*` $\mathcal{O}(1)$ `*/`
14 **3.** Remover (i,j) del camino. Agregar (k,i) y (k,j). ; `/*` $\mathcal{O}(n)$ `en vectores. */`
15 Return **Lista de rutas.**

2.4.2. Complejidad temporal en el peor caso.

Repasamos, entonces, las complejidades de cada parte del algoritmo descripto arriba.

Estimar K como la suma total de demandas sobre la capacidad del camión tiene complejidad temporal $\mathcal{O}(n)$, pues se suman las n demandas y se dividen por la capacidad. Clusterizar los n puntos usando K-Means tiene complejidad temporal $\mathcal{O}(n * K)$. Esto se debe a que, para cada uno de los n puntos, se evalúa la distancia hacia las K medias disponibles para encontrar la más cercana.

Realizar *nearest insertion* para todos los clusters tiene complejidad temporal $\mathcal{O}(n^2)$. Intuitivamente esta conclusión puede entenderse siguiendo el comportamiento de este ciclo: si bien en el cluster inicialmente se debe agregar el vecino más cercano a una ruta compuesta por un sólo nodo (lo cual cuesta *O(n')*, siendo n' los nodos fuera de la ruta), una vez hecho esto deben computarse las distancias de los $n' - 1$ nodos restantes (en *O(n'-1)*) al añadido, por lo que en la iteración siguiente se hará un trabajo de costo *O(n'+n'-1)*. Eventualmente se habrán agregado todos los nodos posibles, y se habrá hecho un trabajo de costo $O(n'+(n'+n'-1)+(n'+n'-1+n'-2)+...) = O(n' * n') = O(n'^2)$. Repetir este trabajo sobre todos los clusters, que en total contienen a los n nodos del grafo, resulta en un costo total $\mathcal{O}(n^2)$.

Realizando el cálculo completo, el algoritmo tiene complejidad temporal $\mathcal{O}(n^2 + K * n)$. Como la cantidad de clusters K resulta ser menor o igual que n, entonces la complejidad temporal final pertenece a $\boxed{\mathcal{O}(n^2)}$

2.4.3. Instancias poco adecuadas.

Como es un algoritmo basado en clusterización, si las instancias presentadas no poseen clusters marcados, entonces no va a haber una buena agrupación de nodos en clusters por parte del algoritmo. Esto puede resultar en nodos muy cercanos que podrían ser abastecidos por un mismo vehículo, pero que no lo sean por estar en clusters diferentes. Sucede, de hecho, en muchas instancias aleatorias generadas para este trabajo.

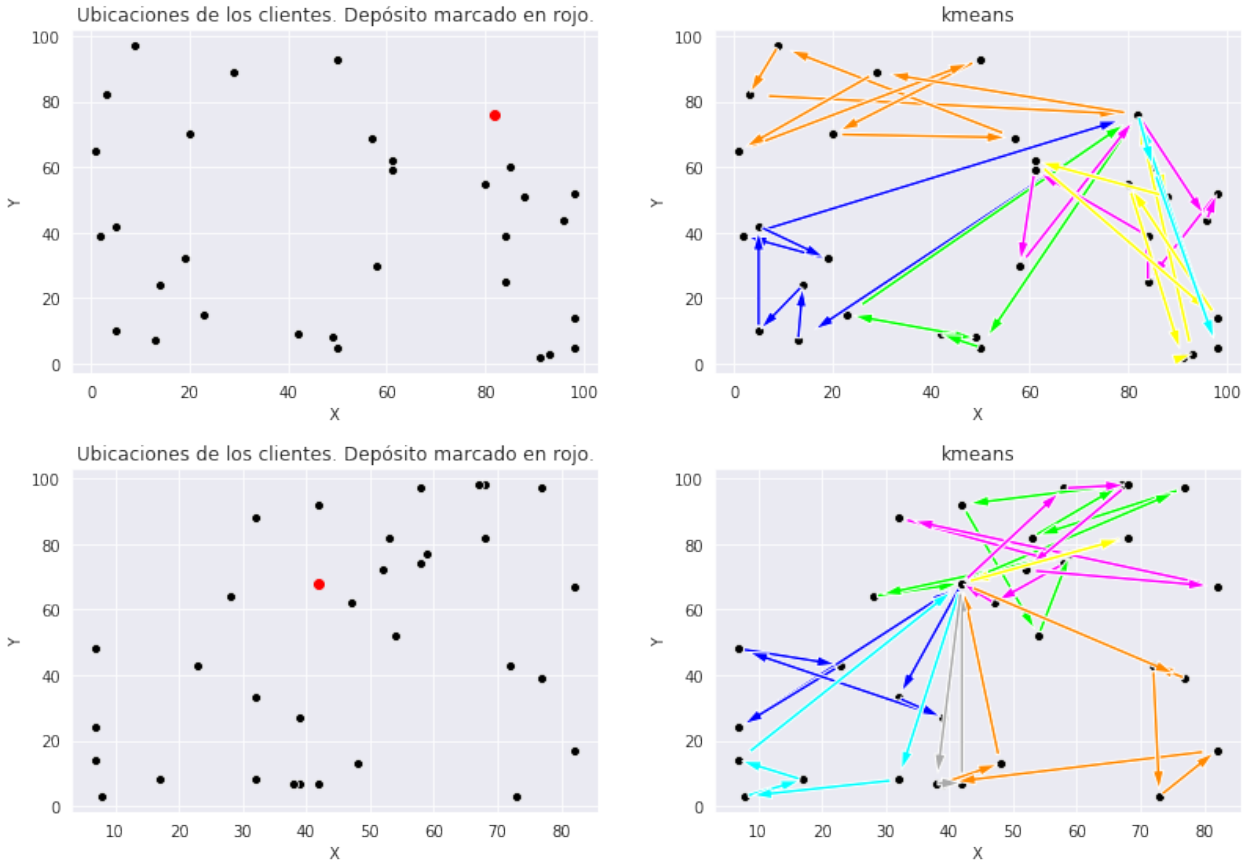


Figura 10: Rutas para las instancias graficadas.

En la figura 10 se ven las rutas construidas por el algoritmo. Más adelante abordaremos la "desprolijidad" y la calidad de dichas rutas, pero ahora veremos su relación con las óptimas. En el caso de la primera instancia, aplicar K-Means resulta en una ruta de costo 1400, contrastando con el costo de 784 de la óptima, casi duplicando la distancia total recorrida. En el caso de la segunda instancia también está cerca de duplicarse, siendo 1176 el costo de la solución encontrada ante 661 de la óptima.

2.5. Metaheurística basada en Simulated Annealing.

2.5.1. Explicación del algoritmo.

Simulated annealing es un algoritmo de búsqueda meta-heurística para problemas de optimización global. Las metaheurísticas se pueden aplicar a cualquier problema que se pueda reformular en términos heurísticos, por ejemplo el problema de este trabajo, y suelen ser menos eficientes que las heurísticas específicas, en varios órdenes de magnitud, en problemas que aceptan este tipo de heurísticas puras.

El nombre e inspiración vienen del proceso de fundición del acero, una técnica que consiste en calentar y luego enfriar lentamente el material para moldearlo en diferentes formas. El calor aumenta la energía de los átomos y permite el desplazamiento de los mismos respecto de sus posiciones iniciales. Por otro lado, el enfriamiento les da mayores probabilidades de quedarse en un lugar. Fue descripto independientemente por Scott Kirkpatrick, C. Daniel Gelatt y Mario P. Vecchi en 1983, y por Vlado Černý en 1985. El método es una adaptación del algoritmo Metropolis-Hastings, un método de Montecarlo utilizado para generar muestras de estados de un sistema termodinámico.

El algoritmo funciona de la siguiente manera: inicialmente se obtiene, mediante otra heurística, una solución aproximada S, la cual se intentará mejorar. Se obtiene su vecindario de soluciones, permutando de a pares cada nodo. Luego, con S y su vecindario se calcular algunos parámetros para el enfriamiento posterior (la máxima diferencia entre el costo de S y algún vecino será la temperatura inicial T_s y cinco veces la temperatura final T_f, y la cantidad de vecinos factibles por la cantidad de vértices del grafo será α. Estos parámetros se inicializan con los valores planteados en el paper [4], con excepcion de T_f para el cual encontramos que para nuestras instancias se comportaba mucho mejor si T_f era la parte entera de T_s /5. Esto se debe a que si en el vecindario de soluciones, hay dos soluciones con el mísmo costo, el valor de T_f es cero (porque el trabajo de Osman y Cristófides proponía que su valor sea la diferencia mínima entre todas las diferencias de costos del vecindario tomados de a dos), entonces cuando se va a valuar la guarda del ciclo, la temperatura baja a cero en una iteración, prueba uno sólo de los vecinos y termina, esto sucedía muy frecuentemente con nuestras instancias por lo que se decidió esta modificación.

Mientras T_s sea mayor a T_f, se obtiene el vecindario de S y se elige un vecino al azar, llamado S' Luego se calcula la diferencia entre los costos de S y S', y con ese valor y T_k se calcula el valor de la función de aceptación en ese momento. Si la función devuelve un resultado mayor o igual que θ (que es un número generado aleatoriamente entre 0 y 1), S' se convierte en el nuevo S. Si, además, su costo es menor que el de $S*$, que es el menor hasta el momento, también se convierte en el nuevo $S*$.

Si S se modificó, en la siguiente iteración se calculará el vecindario de un nuevo nodo; en caso contrario se volverá a calcular el vecindario del mismo nodo, pero al elegir un vecino al azar es muy probable que se elija a otro. Las temperatura se reduce utilizando distintas fórmulas según si S fue modificado o si no, pero en ambos casos se reducen.

Finalmente, si durante 10 iteraciones en las que S fue modificado no se encontró ninguna solución que mejorara a $S*$, se hace un reset mediante el cual S vuelve a valer $S*$. Eso se hace para que si el algoritmo tomó un camino alejándose de $S*$ por el cual no parece haber ganancia, pueda volver a una situación mejor y buscar por otro camino.

Al llegar T_k a ser igual a T_f, el algoritmo termina, y devuelve $S*$, la mejor solución encontrada.

Algoritmo 5: Simulated Annealing

```
1  S ← heuristica(G)
2  V ← vecindario(S)
3  params ← parametros(G, S, V)
4  vueltasSinMejorar ← 0
5  maxCantVueltas ← 10
6  S* ← S
7  while T_k > T_f do
8  |  V ← vecindario(S)
9  |  S' ← V[random]
10 |  δ ← abs(costo(S') − costo(S))
11 |  θ ← random(0, 1)
12 |  if funcAceptacion(δ, T_k) ≥ θ then
13 |  |  S ← S'
14 |  |  if costo(S) ≤ costo S* then
15 |  |  |  S* = S
16 |  |  |  T_b = T_k
17 |  |  |  vueltasSinMejorar = 0
18 |  |  else
19 |  |  |  vueltasSinMejorar + +
20 |  |  end
21 |  |  T_k ← actualizarHabiendoAceptado(T_k)
22 |  else
23 |  |  T_k ← actualizarSinAceptar(T_k)
24 |  end
25 |  k + +
26 |  if vueltasSinMejorar = maxCantVueltas then
27 |  |  S = S*
28 |  |  vueltasSinMejorar = 0
29 |  end
30 end
```

El algoritmo para calcular el vecindario de una solución es

```
1  vecindario ← ∅
2  for i en S do
3  |  for e en S[i] do
4  |  |  for j en S[i,] do
5  |  |  |  for f en S[j] do
6  |  |  |  |  vecino ← copy(S)
7  |  |  |  |  elem1 ← rutas[i][e]
8  |  |  |  |  elem2 ← rutas[j][f]
9  |  |  |  |  vecino[i][e] ← elem2
10 |  |  |  |  vecino[j][f] ← elem1
11 |  |  |  end
12 |  |  end
13 |  end
14 |  return vecino
15 end
```

2.5.2. Complejidad temporal en el peor caso.

Lo primero que surge del análisis de complejidad es que hay una cota inferior dada por la heurística que se usa para obtener la solución inicial.

Además de eso, la obtención del vecindario de soluciones tiene un costo de $\mathcal{O}(n^2)$, pero a la hora de calcular cuántas veces se realiza esto, nos encontramos con un problema: la cantidad de ejecuciones del ciclo principal no está determinada por el tamaño de la entrada.

Sin contar esto, el ciclo de ejecución de Simulated Annealing depende de las variables de temperatura (Tk y Ts) que no dependen del tamaño de la entrada, sino que dependen de la distancia entre los nodos, es decir que dependen del tipo de entrada y no de su tamaño.

Por todo esto, es evidente que plantear un análisis asintótico de la complejidad basado en el tamaño de la entrada no tiene sentido. Sin embargo,es posible una solución alternativa: utilizar un contador que restrinja la cantidad de iteraciones máximas del algoritmo, esto permitiría proponer una cota superior a la complejidad del algoritmo, idealmente, este contador no sería una constante (porque la complejidad podría reducirse a O(1), según el álgebra de órdenes), sino que dependería del tamaño de la entrada. Si bien esta solución permitiría una cota superior a la complejidad de Simulated Annealing, fue considerada una solución muy forzada, sobre todo porque la implementación del algoritmo de este tabajo termina en tiempos razonables, por lo que se optó por explicar las dificultades del análisis de complejidad y plantear una solución posible.

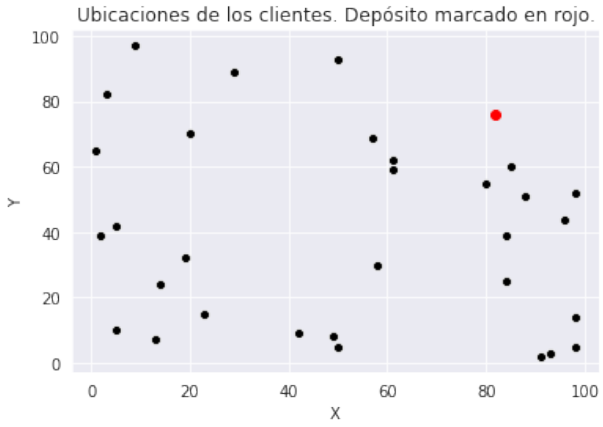
2.5.3. Instancias poco adecuadas.

Al ser una metaheurística, las instancias poco adecuadas tienen relación directa con las instancias poco adecuadas para las heurísticas que utiliza.

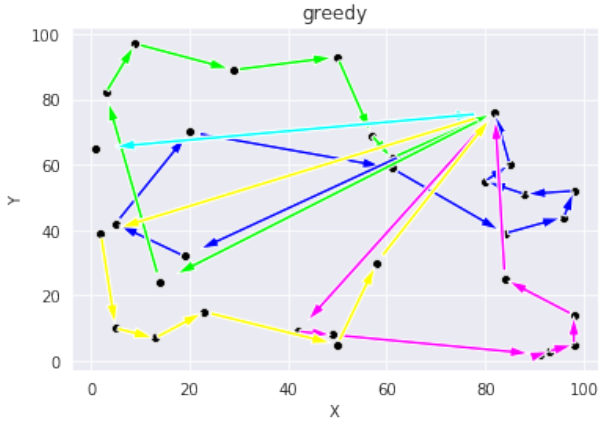
3. Experimentación

3.1. Control cualitativo de rutas generadas

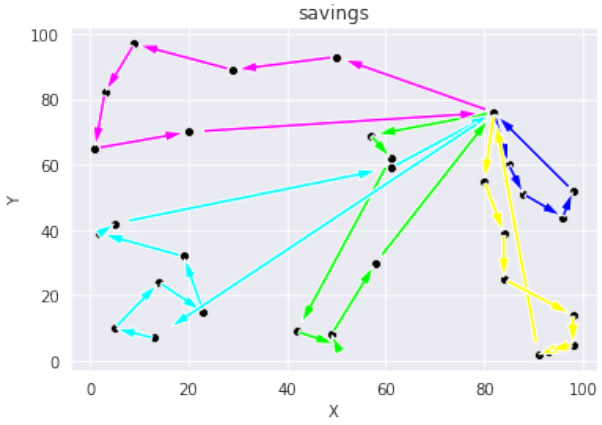
Comenzamos la experimentación con un control cualitativo simple de las rutas que se generan. Queremos ver si hay muchos cruces en una misma ruta, y si falta coherencia en las soluciones presentadas por los varios algoritmos.



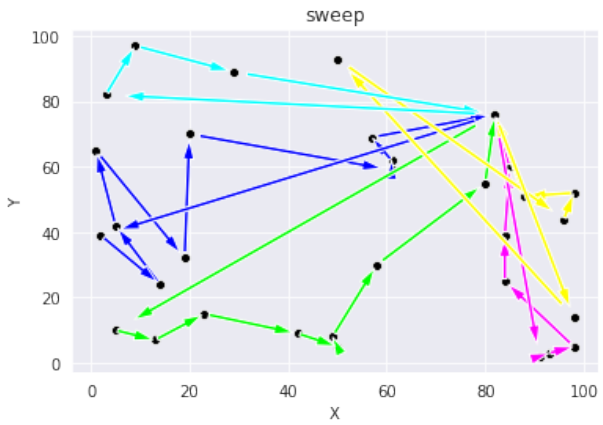
(a) Instancia del problema.



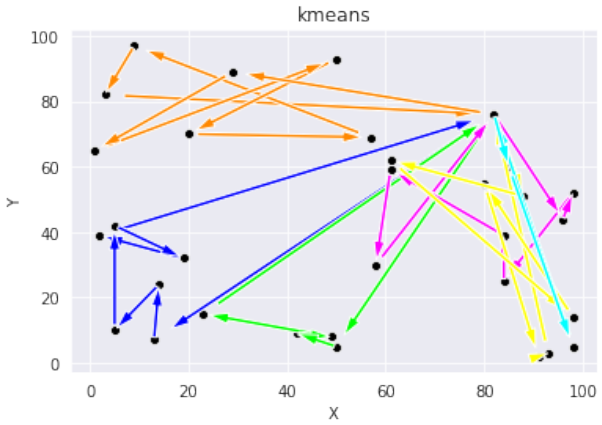
(b) Greedy.



(c) Savings.

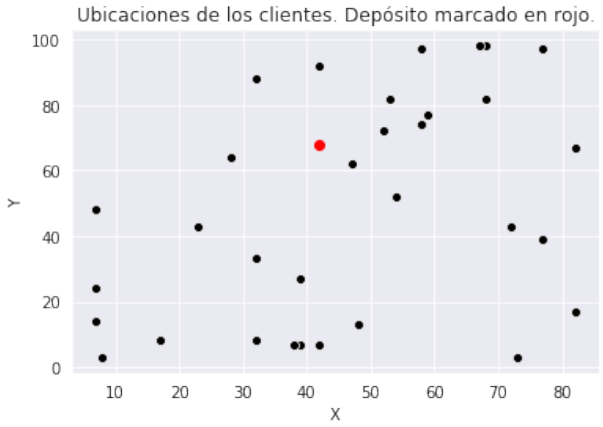


(d) Sweep.

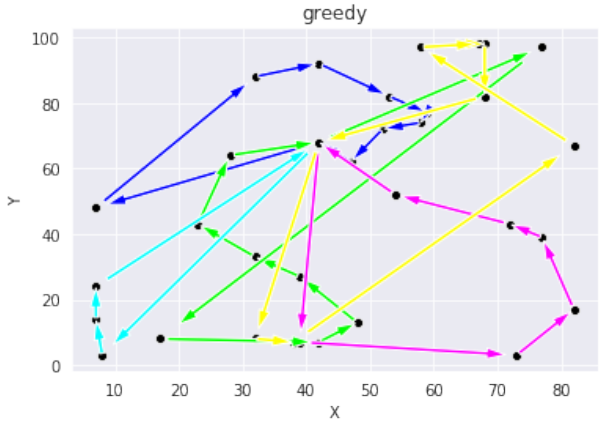


(e) K-Means.

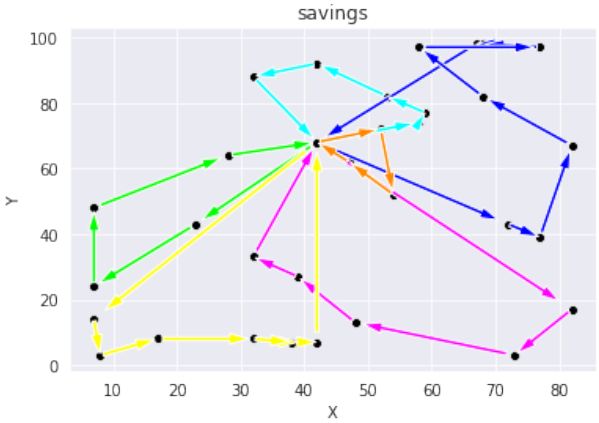
Figura 11: Rutas generadas por las heurísticas.



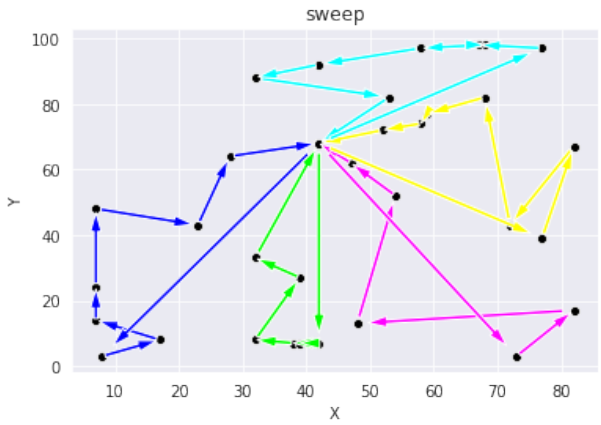
(a) Instancia del problema.



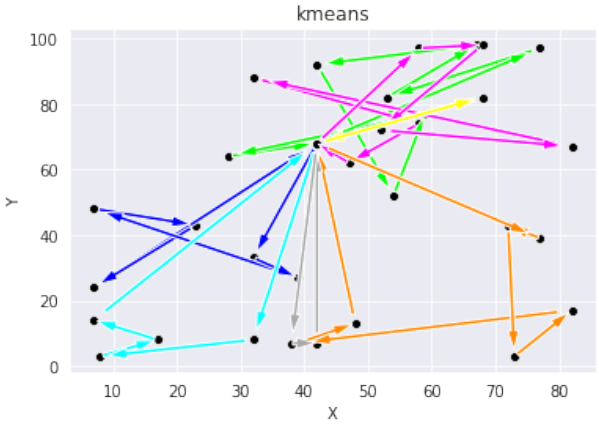
(b) Greedy.



(c) Savings.



(d) Sweep.



(e) K-Means.

Figura 12: Rutas generadas por las heurísticas.

En las figuras 11 y 12 se muestran las rutas generadas por las heurísticas en dos instancias separadas del problema. Para facilitar la apreciación visual sin trivializar el problema, la primera instancia tiene 32 nodos totales y la segunda tiene 33.

Las heurísticas Savings y Sweep dan un conjunto de rutas que no se cruzan mucho entre ellas. En especial Sweep, por su característica clusterización. Savings tiene cruces entre rutas, pero no tantos

como el algoritmo Greedy. Es esperable que Greedy sufra de cruces y otras ineficiencias, dada su naturaleza golosa.

Por otro lado, K-Means parece no estar haciendo un muy buen trabajo en separar las rutas. De hecho, hasta tiene una ruta más que los otros algoritmos en ambos casos. Esto último se debe a como estima K, el número que determina la mínima cantidad de rutas a utilizar. Su mala clusterización se debe a la falta de *clusters* o *conglomerados* reales presentes en las instancias. El algoritmo Sweep no trabaja con una clusterización tradicional. De hecho, hasta es discutible que los clusters en los que divide a los puntos no siempre son clusters cualitativamente correctos. En otras palabras, un humano no distinguiría los clusters a simple vista de la manera en la que los distingue el algoritmo Sweep. K-Means, por su parte, sí trabaja de una manera más tradicional. Si los clusters son bien definidos a simple vista, las medias tenderán a separarse más. Sino, se mezclarán desprolijamente, resultando en casos como los de la figura 11(e) y 11e.

3.2. Tiempo tardado en función de la cantidad de nodos

Todas las heurísticas tienen complejidad temporal dependiente de la cantidad de nodos presentes en el grafo instancia. Parece un buen experimento evaluar la influencia de esta cantidad en los tiempos que tardan las heurísticas en resolver el problema.

Una instancia de n nodos se generó con nodos ubicados aleatoriamente entre las unidades 1 y 100 en ambos ejes de un espacio euclídeo, según una distribución uniforme para cada eje. El depósito corresponde al primer nodo generado aleatoriamente. La capacidad de los vehículos es 50 en todos los casos. Las demandas de cada nodo que no corresponde al depósito se eligieron según una distribución uniforme entre 1 y 25. Se evaluaron instancias de 1 a 4000 nodos, con saltos de a 10 nodos entre instancias. Cada una de ellas se ejecutó 10 veces, promediando el tiempo tardado para dar un resultado final por instancia y graficarlo.

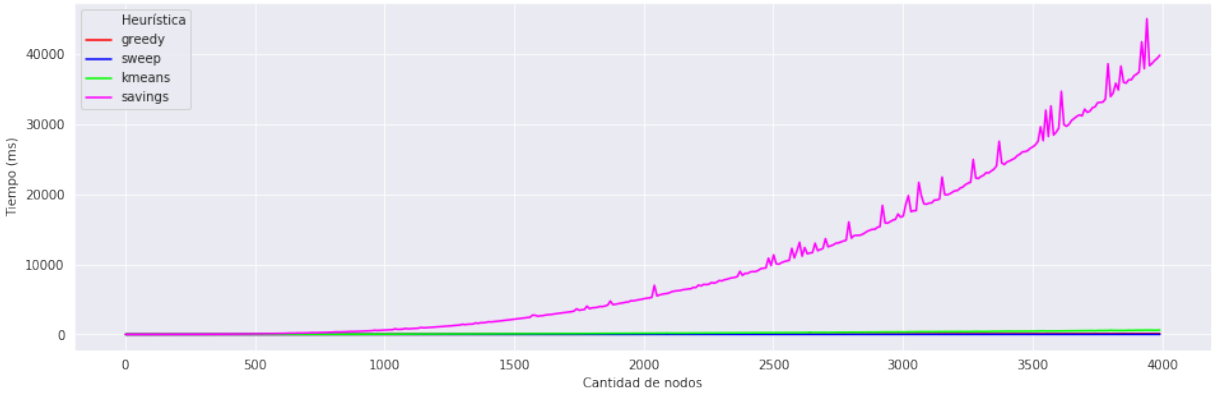


Figura 13: Tiempo tardado en resolver una instancia aleatoria según su cantidad de nodos.

Parece ser, según la figura 13, que el algoritmo Savings tarda notablemente más en resolver casos aleatorios que los otros tres. Este comportamiento, sin embargo, es esperado. La complejidad del algoritmo Savings es la única que está en el orden polinomial cúbico. Las otras tres heurísticas tienen complejidad temporal en $\mathcal{O}(n^2)$. Entonces, esta diferencia no es ninguna sorpresa.

Hay mucha variación en los tiempos que tarda Savings, con muchos picos en varios lugares. Hipotetizamos que esto se debe a la inconsistencia de funcionamiento del sistema operativo en el que se ejecutó el *script* del experimento. Al tardar tanto con más de 2000 nodos, la heurística Savings es

propensa a ser afectada por cuestiones intrínsecas del sistema operativo que generan valores atípicos, y afectan el valor promediado final.

Tampoco es sorpresa que, al menos en el caso de Savings, la figura sea evidencia de que la cantidad de nodos influye en el tiempo tardado. Sin embargo, no se pueden apreciar los casos de las otras heurísticas, por lo que volvemos a graficar, esta vez ignorando Savings.

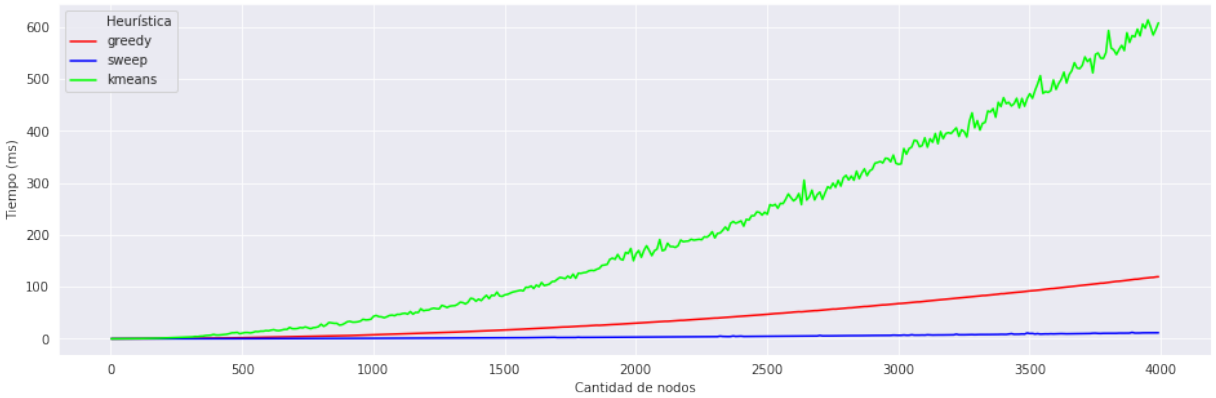


Figura 14: Tiempo tardado en resolver una instancia aleatoria según su cantidad de nodos.

En la figura 14 sí se puede ver cómo el tiempo tardado por las heurísticas K-Means y Greedy es afectado por la cantidad de nodos. K-Means tiene inconsistencias esperadas, pues las medias se generan aleatoriamente cada vez que se corre el algoritmo. En un nuevo gráfico, queremos ver con más claridad el tiempo tardado por el algoritmo Sweep.

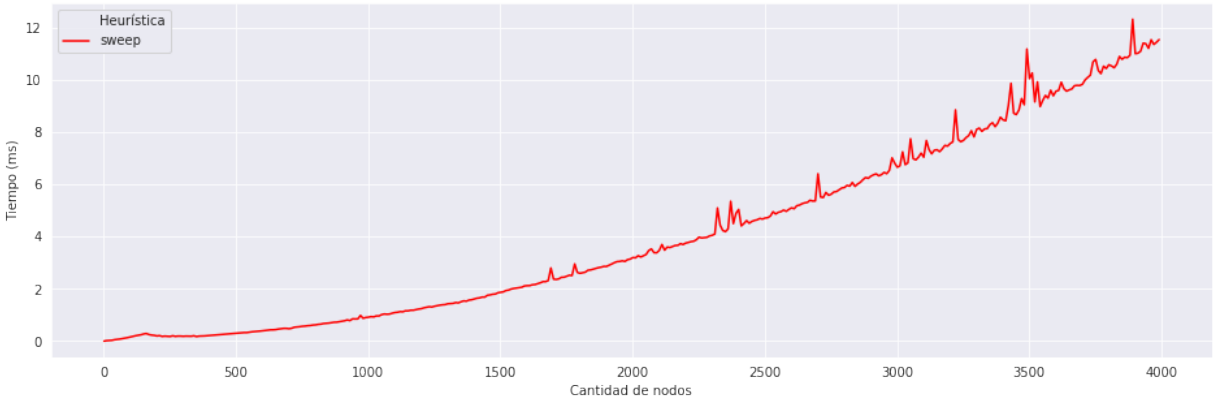


Figura 15: Tiempo tardado en resolver una instancia aleatoria según su cantidad de nodos.

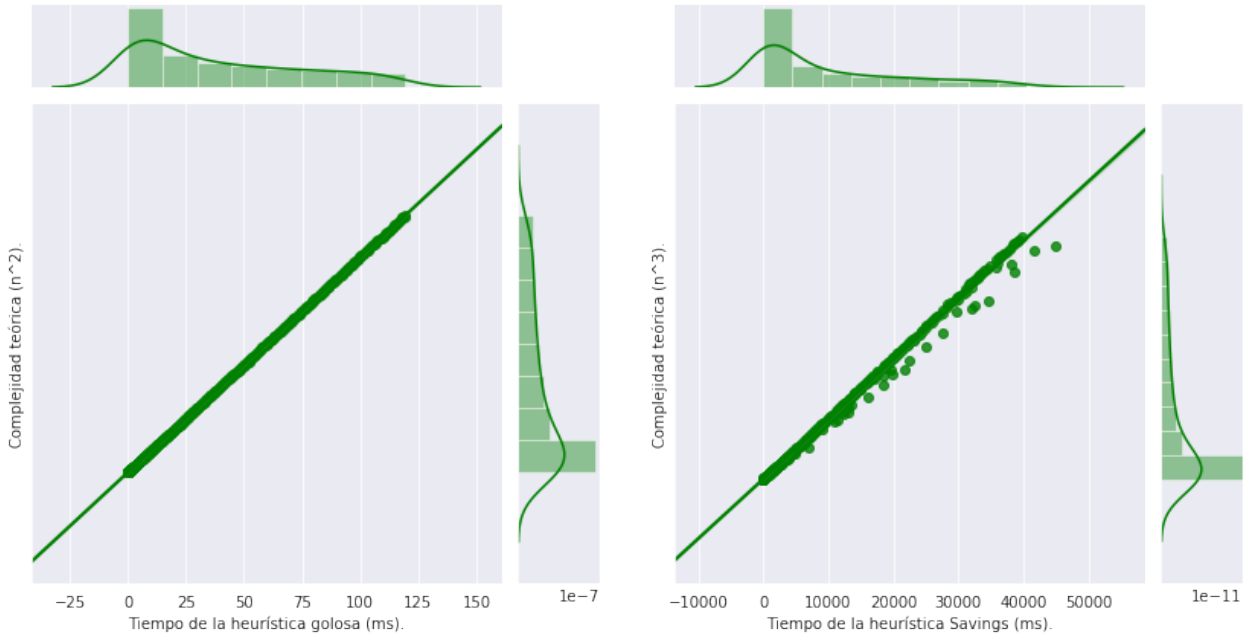
Viendo como la heurística Sweep también es afectada por la cantidad de nodos, entonces podemos decir que nuestra hipótesis de que los tres algoritmos son afectados por la cantidad de clientes es válida. Al menos para las instancias generadas aleatoriamente como fue descripto.

3.3. Relación entre los resultados empíricos y teóricos

El experimento de la sección anterior nos sugiere que los resultados empíricos no se distancian mucho de los teóricos. Queremos, ahora, convencernos con más certeza de que la complejidad teórica

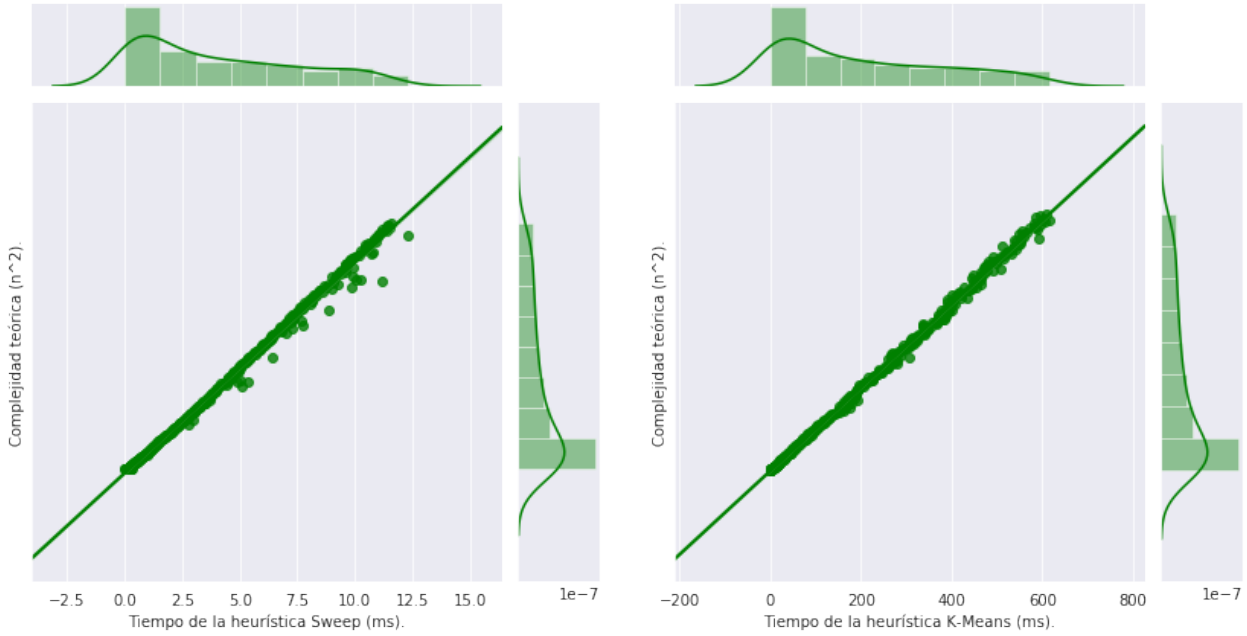
fue correctamente calculada.

Para esto, utilizaremos los tiempos tardados para las instancias utilizadas en la sección anterior. Compararemos los tiempos tardados según la cantidad de nodos con la complejidad teórica en función de la cantidad de nodos y veremos que tipo de relación hay entre ellas.



(a) Greedy.

(b) Savings.



(c) Sweep.

(d) K-Means.

Figura 16: Relación entre la complejidad teórica y los tiempos tardados en instancias aleatorias.

La figura 16, una vez más, nos indica una fuerte relación proporcional entre la complejidad teórica de los algoritmos y los resultados obtenidos por los algoritmos en instancias aleatorias. Pero queda

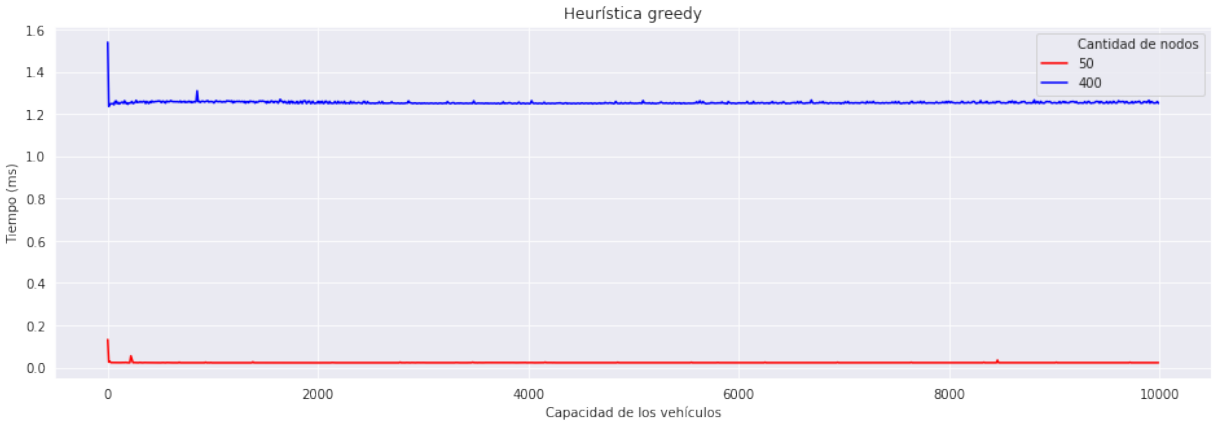
la pregunta de cuán fuerte es esa relación. Para medir esa fuerza, nos valdremos del *coeficiente de correlación de Pearson* [1] [2].

En los cuatro casos, el coeficiente de correlación de Pearson resultante de los experimentos realizados es mayor a 0.9978. Consecuentemente, es razonable concluir que la relación entre los tiempos tardados para resolver CVRP y la complejidad teórica para las cuatro heurísticas mencionadas es evidente y muy fuerte en las instancias descriptas.

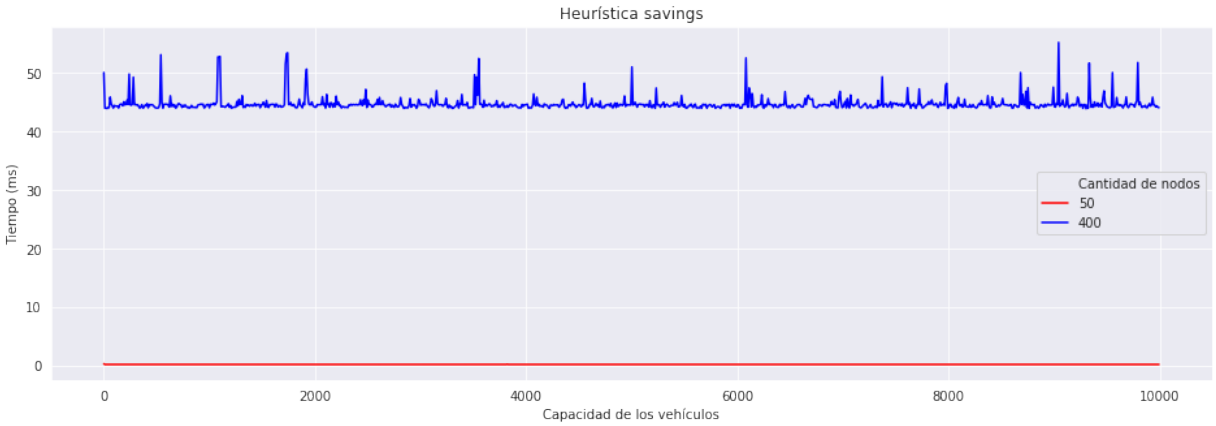
3.4. Tiempo en función de la capacidad de los vehículos

Como la capacidad de los vehículos no influye de manera directa en la complejidad teórica de las heurísticas, queremos ver que el tiempo de cálculo no se ve afectado por ella.

Las instancias sobre las que se experimenta se generan con dos cantidades de clientes, 50 y 400. De nuevo, los clientes se ubican entre las posiciones 1 y 100 de cada eje, según una distribución uniforme. El depósito corresponde al primer nodo generado aleatoriamente. Las demandas de los clientes serán elegidas según una distribución uniforme entre 1 y la mitad de la capacidad de la instancia en cuestión. Las capacidades se evaluarán entre 2 y 10000, con saltos de a 10. Otra vez, para cada instancia, se promedian diez ejecuciones de cada heurística.

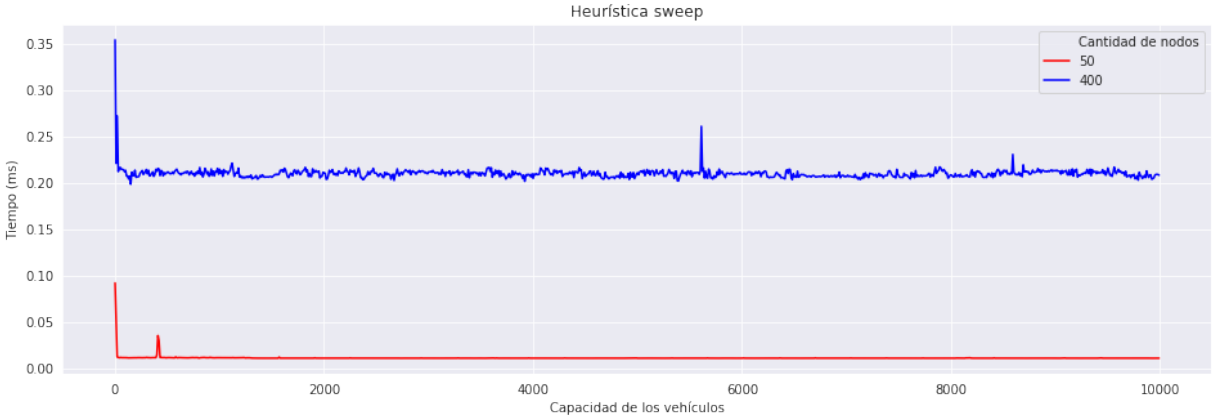


(a) Greedy.

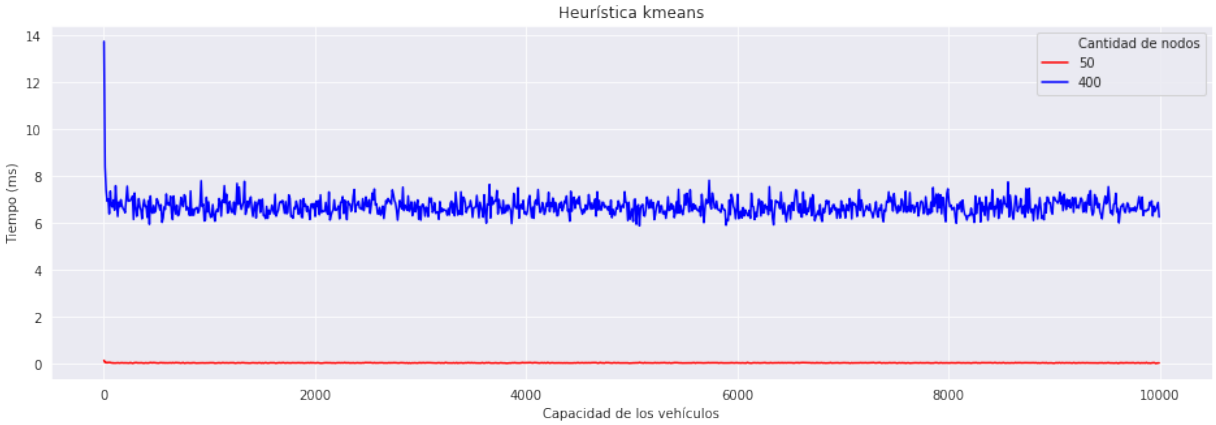


(b) Savings.

Figura 17: Tiempo tardado por las heurísticas en función de la capacidad de los vehículos, para cantidades fijas de clientes.



(a) Sweep.



(b) K-Means.

Figura 18: Tiempo tardado por las heurísticas en función de la capacidad de los vehículos, para cantidades fijas de clientes.

En las figuras 17 y 18 se muestra que, para las instancias generadas, la capacidad de los vehículos no afecta el tiempo que tarda una heurística de las presentadas para entregar una solución factible.

De nuevo, para 400 nodos, las heurísticas son propensas a ser afectadas por cuestiones intrínsecas del sistema operativo que generan valores atípicos, y afectan el valor promediado final, como se puede ver en las figuras 17 y 18.

3.5. Costo de la solución en función de la capacidad de los vehículos

Si mantenemos fija la demanda de los clientes y aumentamos la capacidad de los camiones, creemos que deberíamos ver cierta variación en el costo de las rutas. Como los algoritmos terminan una ruta cuando están cerca de maximizar la capacidad de un vehículo, entonces las longitudes de las rutas y la cantidad de rutas deberían ser afectadas por esa capacidad. En el caso del algoritmo Greedy, por ejemplo, se corta la búsqueda del vecino más cercano cuando éste tiene una demanda que, sumada a la demanda de la ruta construida hasta ahora, supera la capacidad del vehículo.

Generaremos instancias aleatorias con 400 nodos entre los puntos 1 y 100 de cada eje, según una distribución uniforme. El depósito será el primero de ellos. Las demandas de los clientes serán elegidas según una distribución uniforme entre 1 y el mínimo entre 10 y la mitad de la capacidad de un vehículo. Las capacidades se evaluarán entre 2 y 5000, con saltos de a 10. Al estar evaluando costos,

no se promedian los resultados obtenidos.

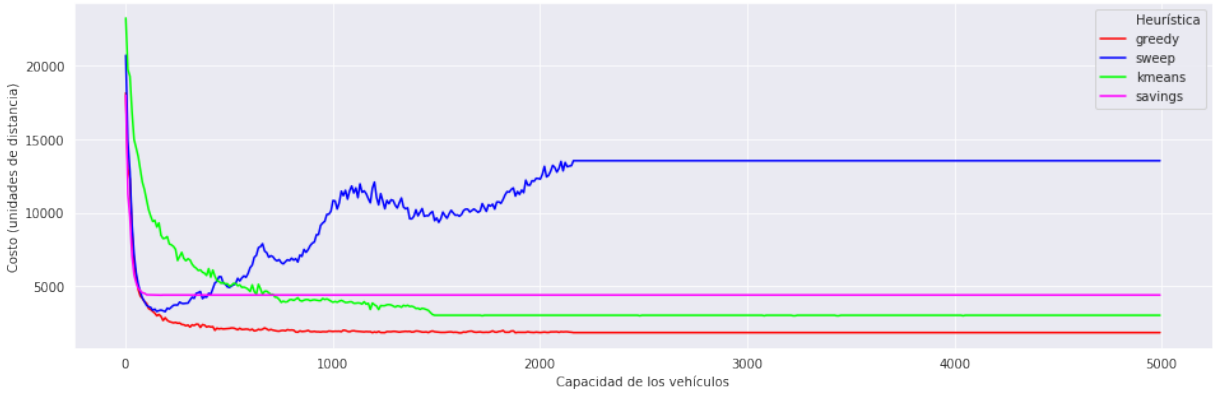


Figura 19: Costo de la solución según la capacidad de los vehículos.

En la figura 19, nos encontramos con diferentes comportamientos según la heurística. Primero, abordaremos el caso del algoritmo Sweep. Éste resulta en rutas con costo fuertemente variable según la capacidad de los vehículos, y se estabiliza una vez la capacidad alcanza cierto valor.

El gráfico resultante de los costos generados por la heurística Sweep parece tener sentido por como se eligen los clusters. Según la capacidad de un camión, se hace un barrido en ángulo polar respecto del depósito mientras que todos los clientes dentro del ángulo generado sumen una demanda total menor a la capacidad del vehículo. Si la capacidad de éste aumenta, el ángulo generado puede crecer aún más para cada ruta, generando menos rutas. Sin embargo, esto puede tener efectos contraproducentes. Si se trata a muchos clientes con el mismo camión y, por ejemplo, el cluster es de un ángulo tan grande que tiene nodos al norte y al sur del depósito, el camión hará recorridos innecesarios. Como no usa una versión pura de nearest neighbour, elige los nodos en el orden en que son más cercanos al depósito. Hay una gran probabilidad de que haya nodos cercanos en todas las direcciones del depósito, entonces podrá generar rutas que recorran un nodo arriba del depósito, luego uno abajo, luego uno arriba y así causando recorridos muy costosos. Sería menos costoso usar dos camiones que atiendan a los nodos norte y sur por separado, por lo que la clusterización es ineficiente. Sin embargo, ese criterio de elección de recorridos no es tan *naïve* cuando los ángulos de clusterización son más chicos.

La explicación de por qué se estabiliza el costo de la solución a partir de cierta capacidad es relativamente simple. Cuando la capacidad de los vehículos es lo suficientemente grande como para superar la suma de demandas de todos los clientes, el algoritmo Sweep agrupará a todos los clientes dentro de un mismo cluster. Este cluster será abastecido por el único camión, con capacidad suficiente. Si se aumenta la capacidad de los vehículos, seguirá formando un único cluster abastecido por un único vehículo. Como el método de resolución de cada cluster para Sweep es determinístico, entonces siempre resulta en una misma ruta con un mismo costo.

Por otro lado, queda evaluar el comportamiento de las otras heurísticas. Para eso, quitamos Sweep del gráfico y lo volvemos a evaluar para menores valores de capacidad en la figura 20.

Resulta que los tres algoritmos tienen un comportamiento similar. A medida que aumenta la capacidad de los vehículos, se requieren menos de ellos para abastecer a los clientes y, por ende, menos rutas que vuelvan al depósito. Esta distancia se ahorra hasta cierto punto, en el que la capacidad de un camión supera la suma de las capacidades totales de los clientes, y un sólo vehículo los abastece a todos.

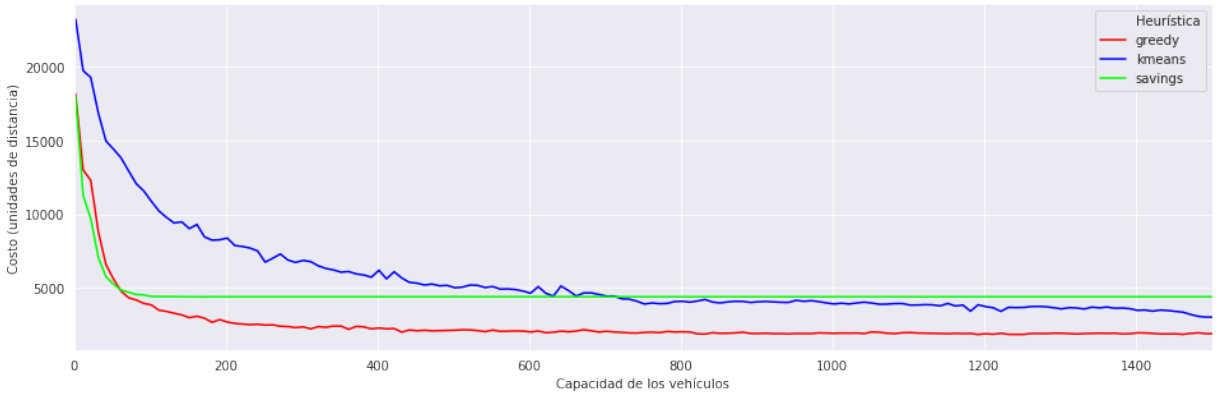


Figura 20: Costo de la solución según la capacidad de los vehículos.

3.6. Costo de la solución de las diferentes heurísticas para instancias aleatorias

Para evaluar el funcionamiento de las diferentes heurísticas presentadas, generaremos varias instancias aleatorias en las que no hay clusters "tradicionales" (clusters no angulares), y veremos el costo de las soluciones generadas. Las instancias sobre las que se trabaja se generan tal cual es explicado en la sección de tiempo en función de cantidad de nodos. Sin embargo, evaluaremos los costos de instancias hasta 500 nodos, y no más que eso. Al estar evaluando costos, no se promedian los resultados obtenidos.

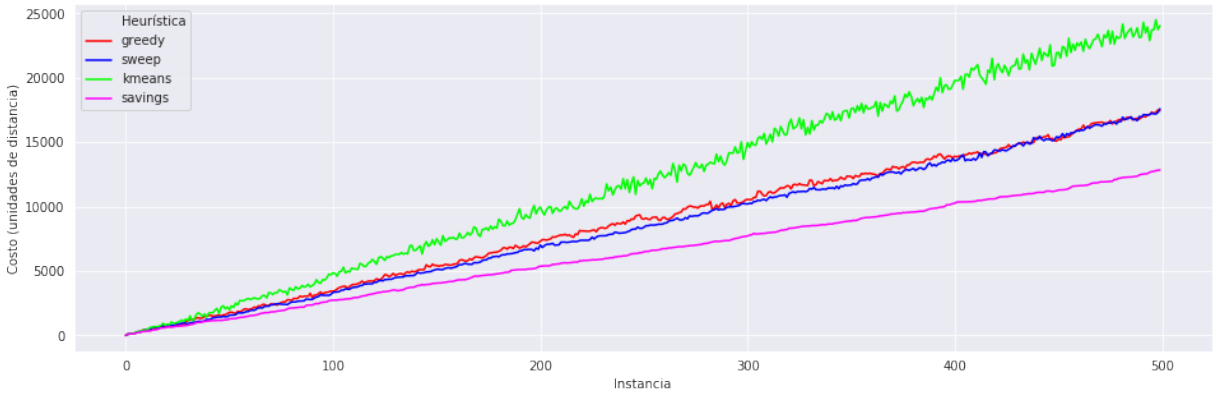


Figura 21: Costo de la solución para una instancia aleatoria i con i nodos.

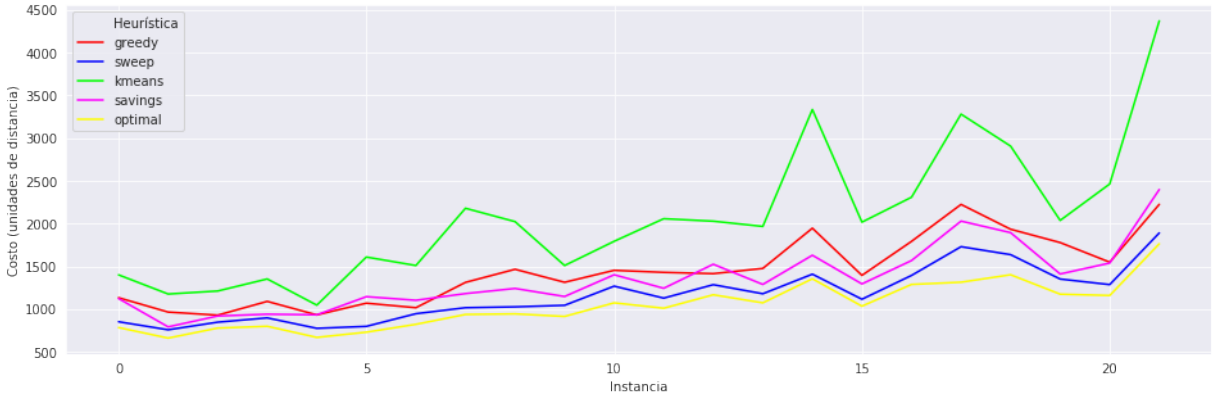
La figura 21 nos muestra el costo de las soluciones generadas por los tres algoritmos. La heurística K-Means parece ser la que da soluciones más costosas. Hipotetizamos que esto es por la falta de clusters "tradicionales" en las instancias generadas, y por la naturaleza errática del algoritmo Nearest Insertion que se utiliza en la parte "route second" del algoritmo para resolver cada cluster.

Los algoritmos Sweep y Greedy parecen comportarse de manera similar. Uno podría discutir que es razonable implementar cualquiera de los dos, por la simpleza en sus implementaciones. Sweep, recordemos, no utiliza Nearest Neighbour para resolver sus clusters, sino que recorre los clusters según más cerca estén del depósito.

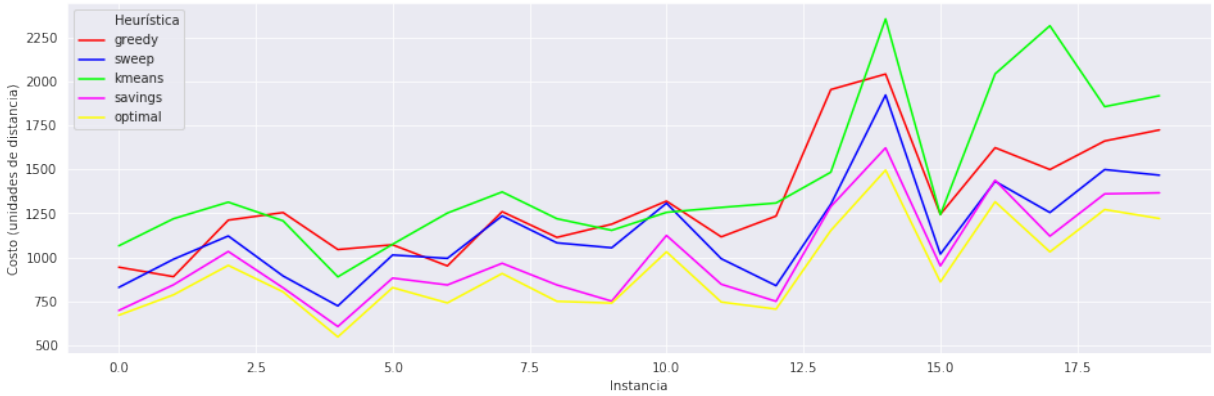
El algoritmo Savings parece estar dominando la categoría de instancias aleatorias. A priori, con la información presentada hasta ahora, parece ser el que provee rutas más razonables y eficientes.

3.7. Costo de la solución de las diferentes heurísticas para instancias conocidas

Ahora pasaremos a evaluar los costos sobre instancias de las cuales tenemos información sobre la solución óptima. Las instancias pueden encontrarse en el *Set A* de [3]. Al estar evaluando costos, no se promedian los resultados obtenidos.



(a) Set A

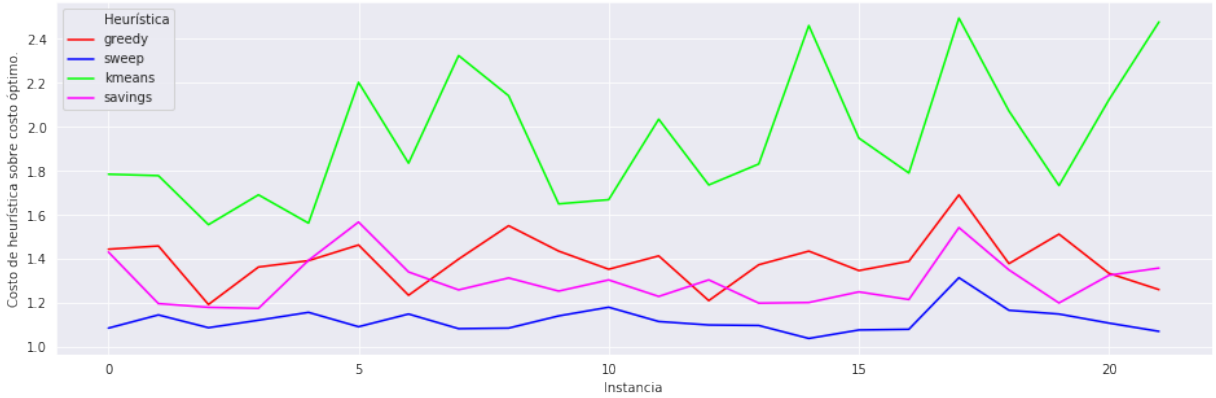


(b) Set B

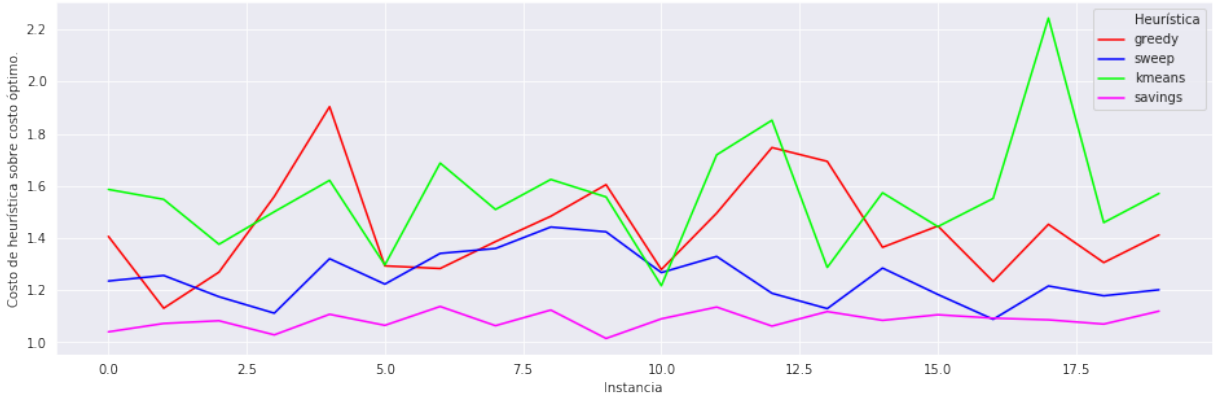
Figura 22: Costo de la solución para instancias conocidas.

Como es de esperarse, la figura 22 muestra que ninguna heurística devuelve la solución óptima, y K-Means vuelve a ser la peor de ellas. Aún así, hay un cambio respecto a las instancias aleatorias, pues el algoritmo Sweep es el que más se acerca a la solución óptima en el set A.

En el set B, Savings vuelve a mostrar su calidad de construcción de rutas. Como los clientes de este set están relativamente lejos del depósito, el algoritmo Savings se ve beneficiado a la hora de hacer los ahorros combinando rutas. K-Means parece también acercarse más a los costos de las demás heurísticas en el set B. A continuación, veremos las relaciones entre los costos obtenidos y los óptimos.



(a) Set A



(b) Set B

Figura 23: Relación entre el costo de la solución heurística y óptima.

En la figura 23 vemos qué tan parecidas son las soluciones en función a la óptima. La relación costo heurístico sobre costo óptimo se mantiene siempre por encima de 1. Ninguna heurística encontró, en estos casos, la solución óptima.

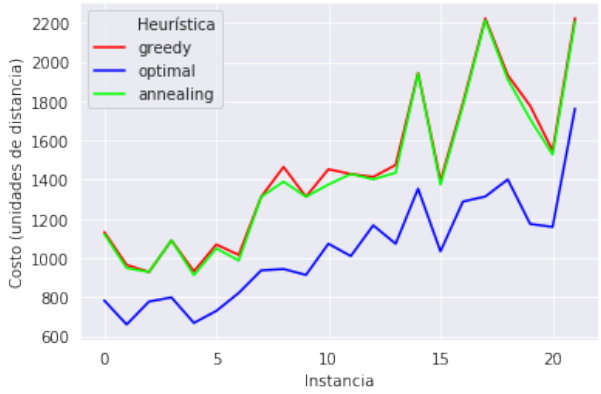
3.8. Mejoras en costo con Simulated Annealing.

El propósito de una metaheurística es mejorar las soluciones dadas por una heurística inicial. Queremos ver si Simulated Annealing, en efecto, mejora las soluciones heurísticas con las que comienza.

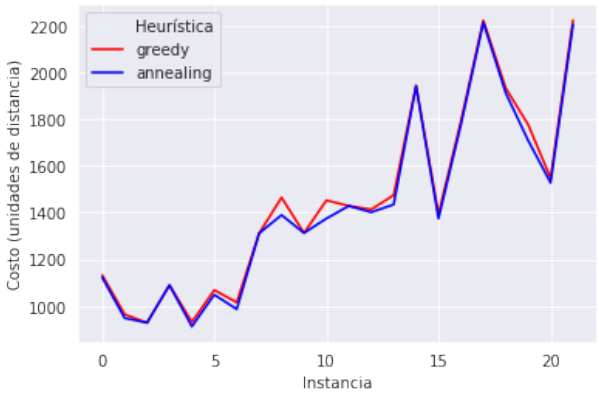
En la figura 24, se ve que, en efecto, la metaheurística mejora los resultados obtenidos en la solución inicial. Hasta en el caso de Savings, que es discutible considerar como la mejor heurística en promedio hasta ahora, se reducen aún más los costos.

No se llega al costo óptimo en ninguno de los casos probados. Sin embargo, la mejora es el objetivo que se impuso, y se cumplió. Se pueden cambiar los parámetros con más consideración según las instancias, y ver qué combinación de ellos resulta en las soluciones menos costosas.

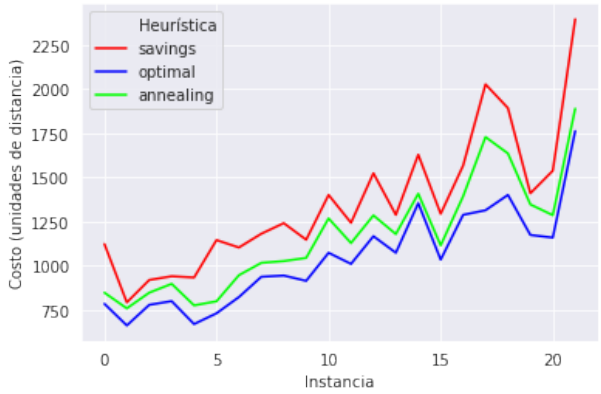
Como Simulated Annealing devuelve los mejores resultados partiendo de una solución dada por la heurística Savings, tomaremos esta implementación como la definitiva en este trabajo.



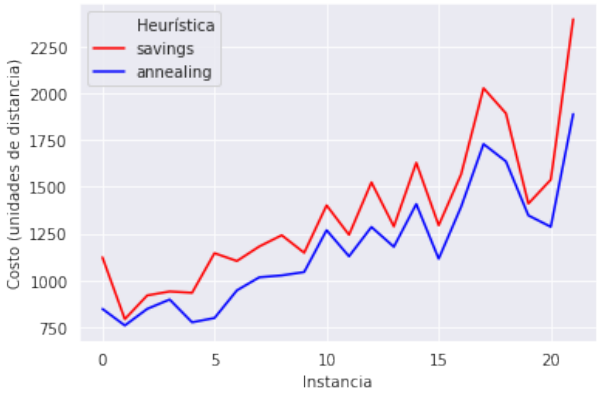
(a) Simulated Annealing inicializada con heurística Greedy.



(b) Sin el costo óptimo.



(c) Simulated Annealing inicializada con heurística Savings.



(d) Sin el costo óptimo.

Figura 24: Costos de las rutas generadas por los algoritmos.

3.9. Parámetros de Simulated Annealing y su influencia en el costo total

El objetivo de este experimento era, originalmente, presentar un gráfico mostrando como influyen los parámetros de temperatura y *alpha* variaban los costos de las soluciones generadas [4]. Sin embargo, nos encontramos con que variar los parámetros de maneras suficientemente drásticas para afectar los costos finales resulta en un incremento multiplicativo del tiempo de ejecución del algoritmo. Tanto así, que en algunos casos con pocos nodos éste tardaba horas en terminar de trabajar sobre una instancia que previamente resolvía en segundos. A pesar de tanta tardanza extra, los resultados cambiaban en menos del 1 % del valor anterior y, por lo general, para peor.

Hipotetizamos que se debe a las características intrínsecas de nuestra implementación. Decidimos, entonces, que la mejor configuración de parámetros es la descripta por el trabajo citado en [4]. Creemos que, si tuviéramos la intención de ver, con más detalle, la influencia de estos parámetros en las soluciones, deberíamos cambiar por completo la filosofía conceptual de nuestra implementación − al menos a nivel código, no pseudocódigo − y revisar las diferentes maneras en las que se pueden elegir los parámetros en función del vecindario inicial disponible de similares maneras al trabajo citado en [4].

4. Conclusiones

En base al estudio de las heurísticas llevado a cabo en las secciones previas, pueden trazarse las siguientes conclusiones:

- La heurística *Greedy* tiene un desempeño temporal razonable, que no superó por mucho margen al del resto de las heurísticas cuando fue comparada contra ellas en los tests realizados, siendo incluso mejor que algunas de ellas (en particular *Savings*). Sin embargo, tanto para instancias aleatorias como para instancias específicas del problema demuestra que la calidad de sus soluciones es desafortunadamente bastante inferior a la del resto (a excepción de *K-Means*).
- La heurística de *Savings* tiene uno de los peores rendimientos temporales asintóticos comparado al resto por su complejidad de peor caso, pero ofrece soluciones cuya calidad fluctúa entre buena y excepcional, superada sólo por la heurística *Sweep* para uno de los sets usados (set A) en la experimentación.
- La heurística *Sweep* ofrece tanto uno de los mejores desempeños temporales (amén de su complejidad en peor caso), como buena calidad en sus soluciones (aunque no escala tan bien como el resto con el aumento de la capacidad de vehículos). Es una de las heurísticas que mejor balancea las dos métricas estudiadas (tiempo y calidad).
- La heurística *K-Means* tiene una performance temporal razonable (superior a la de *Savings*), pero sufre mucho en la calidad de sus soluciones, que se encuentran frecuentemente entre las peores de todas las estudiadas.

Por su parte,la metaheurística de *Simulated Annealing* demostró ser un método efectivo de mejora de soluciones iniciales, como fue expuesto en la experimentación. En particular, puede recomendarse el uso de esta metaehurística incluso cuando las soluciones iniciales son provistas por heurísticas tardicionalmente "lentas" (relativas al resto) como *Savings*.

En base a lo expuesto se estima que las heurísticas *Sweep* y *Savings* ofrecen el mejor *trade-off* entre calidad de soluciones y tiempo insumido. Si bien la segunda puede tomar una cantidad no menos significativa de tiempo extra, también tiene la ventaja de poder ofrecer soluciones superiores en casos aleatorios y para ciertos otros casos particulares que pueden darse con buena frecuencia (clientes distantes al depósito), que son además de fácil detección a ojo.

Finalmente, las heurísticse cas *Greedy* y *K-Means*, aunque moderadamente veloces, no ofrecen soluciones de buena calidad como para considerarse una alternativa valiosa a las antedichas. Sin embargo, escalan bien a medida que aumentan las capacidades de los vehículos, lo cual no ocurre con la misma intensidad para *Savings*, por lo que podrían eventualmente resultar atractivas (en especial considerando que esta última cuenta con la peor performance temporal).

En retrospectiva, nos habría gustado tener más tiempo para hacer experimentaciones más detalladas y profundas que las realizadas. Sin embargo, no fue posible. Entre otras cosas, este trabajo nos mostró las características de la investigación científica en un nivel en que otros trabajos no lo hicieron. El enfrentamiento ante lo desconocido de estos algoritmos fue realmente marcado en relación a todo lo hecho anteriormente en la cursada, debido a la poca relación que tuvimos con heurísticas en general. Esto logró establecer un clima más realista en esta ocasión que los experimentados previamente.

Referencias

[1] Karl Pearson: Notes on regression and inheritance in the case of two parents, Proceedings of the Royal Society of London, 1895.
[2] SPPS: Pearson Correlation.
[3] Instancias conocidas: http://vrp.atd-lab.inf.puc-rio.br/index.php/en/
[4] Ibrahim Osman, Nicos Christofides: Capacitated Clustering Problems by Hybrid Simulated Annealing and Tabu Search, 1994.