DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3

Algoritmos y Estructuras de Datos III
Segundo Cuatrimestre de 2018

Integrante	LU	Correo electrónico
Alliani Federico	183/15	fedealliani@gmail.com
Imperiale Luca	436/15	luca.imperiale95@gmail.com
Raposeiras Lucas Damián	034/15	lucas.raposeiras@outlook.com

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

Índice

1. Introducción

El presente trabajo práctico consiste en comprender, analizar, resolver e implementar distintas soluciones aproximadas al *Problema de Ruteo de Vehículos con Capacidad* (*CVRP* por sus siglas en inglés).

El *CVRP* es un *Problema de Ruteo de Vehículos* (*VRP*) donde una flota de vehículos con determinada capacidad debe satisfacer las demandas de cierta cantidad de clientes desde un depósito en común recorriendo la menor distancia posible en total.

1.1. El Problema

Dado un conjunto de clientes posicionados en un plano euclídeo conectados entre sí y con un depósito D, con k_i la demanda asociada al i-ésimo cliente, y dado un conjunto no acotado de vehículos con capacidad uniforme C. Hallar un conjunto de rutas entre el depósito y los clientes de mínima longitud que cumplan:

- Cada cliente es visitado exactamente una vez por algún vehículo de la flota.
- Cada vehículo comienza y termina su ruta en el depósito.
- La suma de las demandas k_i de los clientes visitados por cada vehículo no puede exceder la capacidad C del mismo.

Modelelado como un problema de grafos:

Sea C un valor numérico que representa la capacidad uniforme de los vehículos. Sea $G = (V, E)$ un grafo no dirigido y completo tal que $\exists\ x \in V$ que representa al depósito, y para cada nodo $v \in V - \{x\}$, v representa un cliente con su correspondiente demanda asociada. Además, para cada arista $e = (u, v) \in E$ se define el $peso$ de e como la distancia euclídea entre los puntos que son representados por los nodos u y v en G. Sea una $flota$ no acotada de $vehículos$ con capacidad C. Hallar un conjunto de ciclos simples de mínimo peso asociado que cumplan:

- Para cada nodo $v \in V\ /\ v \neq x,\ v$ pertenece a un único ciclo.
- Cada ciclo comienza y termina en el nodo x.
- La suma de las demandas k_i de los nodos $v \neq x$ del ciclo no puede exceder el valor de C.

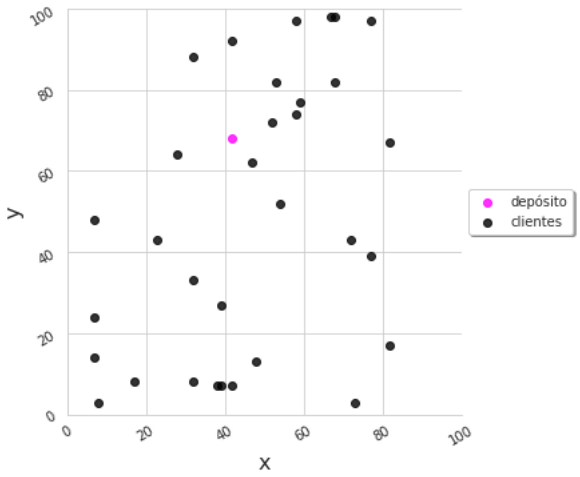

Figura 1: Conjunto de 32 clientes y un depósito de ejemplo

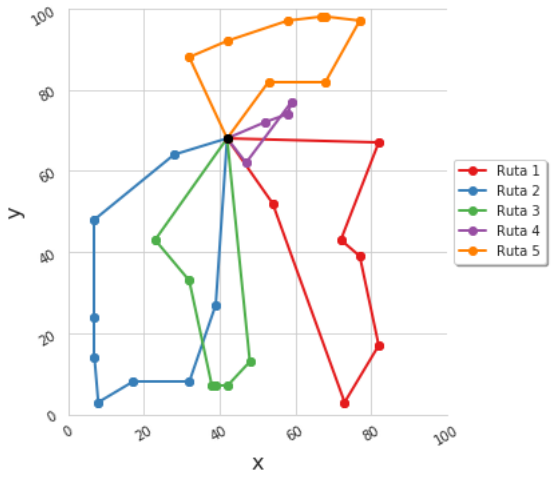

Figura 2: Conjunto de rutas de mínima longitud del conjunto de clientes y depósito de ejemplo mostrado en la Figura 1

Por ejemplo, para el conjunto de clientes y depósito de la Figura 1, donde los clientes poseen una

demanda en el rango $[0, 24]$ y la capacidad C de los vehículos es 100, el conjunto de rutas entre el depósito y los clientes de mínima longitud es el mostrado en la Figura 2.

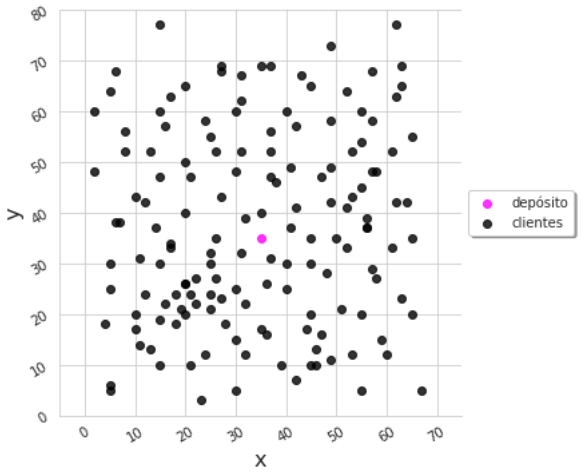

Figura 3: Conjunto de 150 clientes y un depósito de ejemplo

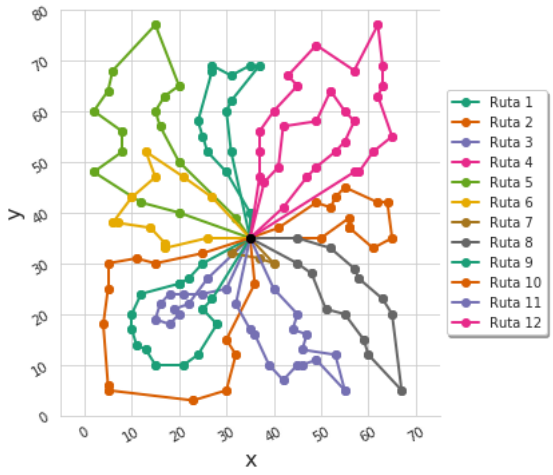

Figura 4: Conjunto de rutas de mínima longitud del conjunto de clientes y depósito de ejemplo mostrado en la Figura 3

Para el conjunto de clientes y depósito de la Figura 3, donde los clientes poseen una demanda en el rango $[0, 41]$ y la capacidad C de los vehículos es 200, el conjunto de rutas entre el depósito y los clientes de mínima longitud es el mostrado en la Figura 4.

El *CVRP*, como se vió en los ejemplos anteriores, **tiene** solución exacta. La forma más sencilla para hallar dicha solución podría ser mediante fuerza bruta, hallando cada combinación posible de ciclos y retornar la que minimize la distancia total recorrida. Como se puede intuir, esta técnica, implementada como un algoritmo de programación, no nos permite hallar una solución óptima en un tiempo de cómputo *razonable* para instancias donde la cantidad de clientes es lo suficientemente grande, ya que la cantidad de combinaciones a corroborar crece de forma exponencial en relación al tamaño de la instancia.

Esto nos dice que el *CVRP* es un problema computacionalmente difícil (NP-hard) y, precisamente, dejaremos de lado las técnicas que dan con la solución exacta al problema para concentrarnos en las heurísticas y metaheurísticas, de forma tal que, implementadas como algoritmos de programación, nos permitan hallar **soluciones aproximadas** del problema original a cambio de mejorar notablemente el tiempo de cómputo. Específicamente, vamos a presentar cuatro algoritmos heurísticos y un algoritmo metaheurístico.

2. Resolución heurística

2.1. Heurística de *savings*

2.1.1. Algoritmo

Comenzaremos definiendo una solución factible inicial al problema donde tenemos tantos vehículos como clientes, y cada vehículo únicamente satisface la demanda de un sólo cliente. Es decir, en esta solución inicial, para cada cliente i definimos la ruta *depósito – i – depósito* y se la asignamos a un único vehículo. Por ejemplo, para la instancia de la Figura 1, la solución factible inicial para la heurística de *savings* es la mostrada en la Figura 5.

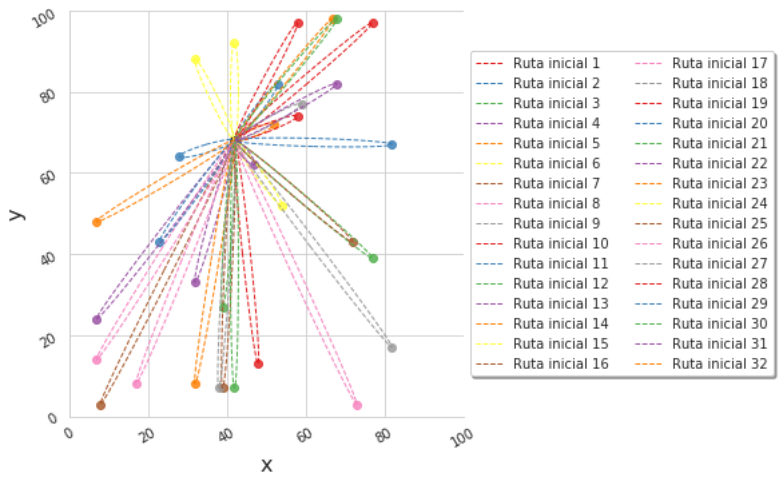

Figura 5: Solución factible inicial para la heurística de *savings* del conjunto de clientes y depósito de ejemplo mostrado en la Figura 1

Como estamos tratando con heurísticas, el simple hecho de hallar una solución factible implica que la misma, de por sí, ya es una solución aproximada al problema original. Si calculamos la distancia recorrida por cada vehículo teniendo en cuenta que cada uno de ellos debe "ir" y "volver" a su cliente, obtenemos que la distancia total recorrida por todos los vehículos es:

$$\sum_{i=1}^{n} 2 \times \|depósito \ - \ cliente_i\|$$

Una vez que construida la solución inicial, podemos calcular cuánta distancia podríamos ahorrar si consideramos unir dos rutas de dos vehículos distintos en una única ruta utilizando un sólo vehículo (siempre y cuando dicho vehículo pueda satisafacer la demanda de los clientes de los vehículos a los cuales está reemplazando). Llamaremos *saving* a dicho ahorro, y se calculará de la siguiente manera:

$$\begin{aligned} S(i,j) = 2 \ \times \|depósito \ - \ cliente_i\| \ + \ 2 \ \times \|depósito \ - \ cliente_j\| \\ - \Big(\|depósito \ - \ cliente_i\| + \|cliente_i \ - \ cliente_j\| + \|depósito \ - \ cliente_j\|\Big) \\ S(i,j) = \|depósito \ - \ cliente_i\| + \|depósito \ - \ cliente_j\| - \|cliente_i \ - \ cliente_j\| \end{aligned}$$

Calcularemos el *saving* para cada par de rutas y nos quedaremos con aquellos cuyo valor sea estrictamente mayor a cero y su unión de rutas correspondiente mantenga una solución factible, es decir, que no se exceda la capacidad del vehículo. Luego de calcular todos los *savings*, los ordenaremos decrecientemente, es decir, nos quedarán primeros aquellos *savings* que nos permitan generar un ahorro mayor.

Por último, iremos combinando las rutas que mayor ahorro nos genera y, luego de cada cambio, tendremos que actualizar los *savings* calculados, ya que por cada nueva unión habrá una ruta menos en la solución final. Esto lo haremos hasta que no haya más *savings* posibles.

A continuación se presenta el pseudocódigo para el algoritmo recién planteado.

function CVRP-SAVINGS($G =< V, E >$, *demandaDeLosClientes*, *capacidadVehículo*, *depósito*)
  rutas ← OBTENERSOLUCIÓNINICIAL(*G*, *demandaDeLosClientes*) ▷ $O(n)$
  savings ← COMPUTARSAVINGS(*G*, *rutas*, *capacidadVehículo*) ▷ $O(n^2)$
  ORDENARDECRECIENTEMENTE(*savings*) ▷ $O\big(n \times (log_2(n))^2\big)$
  while $savings.size() > 0$ **do** ▷ $O(n \times \ldots)$
    $rutaA \leftarrow savings_0.rutaA$ ▷ $O(1)$
    $rutaB \leftarrow savings_0.rutaB$ ▷ $O(1)$
    nuevaRuta ← COMBINARRUTAS(*G*, *rutaA*, *rutaB*) ▷ $O(n)$
    AGREGARRUTA(*rutas*, *nuevaRuta*) ▷ $O(1)$
    ELIMINARRUTA(*rutas*, *rutaA*) ▷ $O(n)$
    ELIMINARRUTA(*rutas*, *rutaB*) ▷ $O(n)$
    RECALCULARSAVINGS(*G*, *savings*, *rutas*, *nuevaRuta*, *rutaA*, *rutaB*, *capacidadVehículo*) ▷ $O(n^2)$
  end while

  costo ← CALCULARCOSTO(*rutas*) ▷ $O(n)$
  return $<$ *rutas*, *costo* $>$ ▷ $O(1)$
end function

2.1.2. Análisis de complejidad

La complejidad del algoritmo es $O(n^3)$, con n la cantidad de puntos en el plano incluyendo al depósito.

Analizaremos la complejidad del ciclo **while** principal y de las funciones OBTENERSOLUCIÓNINICIAL, COMPUTARSAVINGS, ORDENARDECRECIENTEMENTE, COMBINARRUTAS, AGREGARRUTA, ELIMINARRUTA, RECALCULARSAVINGS y CALCULARCOSTO.

La función OBTENERSOLUCIÓNINICIAL genera la solución inicial especificada en la sección 2.1.1. La complejidad temporal de crear cada ruta es $O(1)$, ya que cada ruta está formada únicamente por tres nodos (*depósito* – $cliente_i$ – *depósito*. Por lo tanto, como tenemos $n-1$ clientes y para cada uno se puede generar una ruta en $O(1)$, la función OBTENERSOLUCIÓNINICIAL obtiene una complejidad temporal total de $O(n)$.

Luego, se ejecuta la función COMPUTARSAVINGS que determina, para cada par de rutas, el ahorro generado si las mismas se unen en una sola ruta. Para calcular cada valor de *saving*, se utiliza la fórmula de $S(i, j)$ enunciada en la sección 2.1.1. Si el valor de *saving* es positivo decimos que se genera un ahorro, y agregamos dicho valor a la lista resultante de la función COMPUTARSAVINGS. Como el cálculo de *saving* se hace para cada par de rutas e inicialmente tenemos n rutas, la cantidad total de cálculos de *saving* será $O(n^2)$.

La función ORDENARDECRECIENTEMENTE se encarga de ordenar de manera decreciente la lista resultante de la función COMPUTARSAVINGS en base al ahorro obtenido. Para ello, ORDENARDECRECIENTEMENTE hace uso de la función `std::sort` de la librería estándar de `C++`, cuya complejidad es de $O(n \times log_2(n)^2)$.

A partir de allí, se ejecuta un ciclo **while** que itera sobre la cantidad de ahorros posibles y no tiene ningún tipo de condición extra que le permita terminar antes. Sin embargo, cada vez que unimos dos rutas, tendremos una ruta menos, entonces el bloque interno del ciclo se ejecutará como máximo n veces, ya que no es posible hacer más de n combinaciones de rutas. Por cada iteración del ciclo se combinan las rutas determinadas por el *saving* mediante la función COMBINARRUTAS, cuya complejidad temporal es $O(n)$, ya que la unión de las dos rutas puede resultar, en el peor caso, en una ruta con todos los clientes.

La función AGREGARRUTA agrega la ruta recién combinada al final de la lista de rutas. La lista de rutas está implementada con una lista doblemente enlazada, con lo cual la complejidad de AGREGARRUTA es $O(1)$.

La función ELIMINARRUTA elimina la ruta indicada en el segundo parámetro de la lista de rutas del primer parámetro. Para ello, la busca de forma secuencial hasta encontrarla y seguidamente la elimina. El costo de la búsqueda es $O(n)$ debido a que la lista de rutas está implementada con una lista doblemente enlazada. El costo de eliminarla es $O(1)$ ya que únicamente se debe que modificar una cantidad constante de punteros para que el invariante de la lista doblemente enlazada se siga cumpliendo.

```
function RecalcularSavings(G =< V, E >, savings, rutas, nuevaRuta,
                           rutaA, rutaB, capacidadVehiculo)
    savings ← SavingsQueNoContenganA(rutaA, rutaB)                          ▷ O(n^2)
    for 1 ≤ i ≤ rutas.size() do                                             ▷ O(n)
        if rutas_i ≠ nuevaRuta then                                         ▷ O(1)
            if rutas_i.capacidad + nuevaRuta.capacidad ≤ capacidadVehiculos then   ▷ O(1)
                rutaCombinada ← CombinarRutas(G, ruta_i, nuevaRuta)         ▷ O(n)
                if rutaCombinada.distancia < nuevaRuta.distancia + ruta_i.distancia then   ▷ O(1)
                    savingNuevo ← CrearSaving(G, rutas_i, nuevaRuta)        ▷ O(n)
                    Agregar(savingNuevo, savings)                           ▷ O(1)
                end if
            end if
        end if
    end for
end function
```

Por último, se recalculan los valores de *saving* nuevos que surgen de haber unido las dos rutas utilizando la función RecalcularSavings. Primero se recorre la lista de *savings* y se van agregando a una lista resultante aquellos que no contenían ninguna de las dos rutas recién combinadas con la función SavingsQueNoContenganA. El costo de agregar un *saving* a la lista resultante tiene complejidad $O(1)$, ya que dicha lista está implementada sobre una lista doblemente enlazada, y el costo de agregar un elemento en la misma es $O(1)$. Por lo tanto, la complejidad de SavingsQueNoContenganA es el costo de recorrerla que en el peor caso puede ser $O(n^2)$. Para verificar si se genera un nuevo valor de *saving* entre la ruta nueva y otra ruta cualquiera, utilizamos la función CombinarRutas, cuya complejidad es $O(n)$ como mencionamos anteriormente. Si la nueva ruta genera un ahorro utilizando la fórmula de $S(i, j)$ enunciada en la sección 2.1.1, entonces el algoritmo crea un nuevo valor de *saving* utilizando la función CrearSaving, cuya complejidad es $O(n)$ ya que debe copiar la ruta entera. Por último, debemos agregar dicho *saving* a la lista de *savings* utilizando la función Agregar, pero como la lista esta implementada sobre lista doblemente enlazada, la complejidad de Agregar es $O(1)$. Como estas funciones se realizan entre la ruta nueva y todas las otras rutas, en el peor caso se realiza $O(n)$ veces. Esto permite concluir que la complejidad temporal de la función RecalcularSavings es $O(n^2)$.

Entonces, como la función CVRP-Savings tiene un ciclo cuyo bloque interno se ejecuta $O(n)$ veces, y cada bloque interno tiene complejidad $O(n^2)$, se obtiene una complejidad total de $O(n^3)$.

2.2. Heurística golosa

2.2.1. Algoritmo

La heurística golosa que hemos elegido para obtener una solución aproximada es la del *vecino más cercano* (*Nearest Neighbor*). En esta técnica comenzaremos con un vehículo y el mismo se encargará de satisfacer al cliente más cercano al depósito, luego al cliente más cercano al último cliente visitado (siempre y cuando dicho cliente todavía no haya sido visitado), y así sucesivamente mientras el vehículo tenga la capacidad necesaria para satisfacer a dicho cliente y, en el caso de no poder satisfacerlo, volverá al depósito. Cuando un vehículo define correctamente su ruta, se procede a realizar el mismo método con un nuevo vehículo de la flota, buscando los clientes más cercanos entre sí que todavía no hayan sido visitados por otro vehículo de la flota. Esto se repite hasta que todos los clientes hayan satisfecho su demanda.

A continuación se presenta el pseudocódigo para el algoritmo recién planteado.

```
function CVRP-GOLOSA(G =< V, E >, demandaDeLosClientes, capacidadVehículo, depósito)
    rutas ← ∅
    loAgregué_i ← false                                                          ▷ O(n)
    while ¬TODOSTRUE(loAgregué) do                                               ▷ O(n × ...)
        capacidad ← 0                                                            ▷ O(1)
        ruta ← [ ]                                                               ▷ O(1)
        AGREGAR(ruta, depósito)                                                  ▷ O(1)
        clienteActual ← depósito                                                 ▷ O(1)
        for 0 <= i < n do                                                        ▷ O(n × ...)
            clienteMásCercano ← OBTENERMÁSCERCANO(G, capacidad, clienteActual, loAgregué)   ▷ O(n)
            if clienteMásCercano ≠ −1 then
                loAgregué[clienteMásCercano] ← true                              ▷ O(1)
                AGREGAR(ruta, clienteMásCercano)                                 ▷ O(1)
                capacidad ← capacidad + demandaDeLosClientes[clienteMásCercano]  ▷ O(1)
                clienteActual ← clienteMásCercano                                ▷ O(1)
            end if
        end for
        AGREGAR(ruta, depósito)                                                  ▷ O(1)
        rutas ← rutas ∪ {ruta}                                                   ▷ O(1)
    end while

    costo ← CALCULARCOSTO(rutas)                                                 ▷ O(n)
    return < rutas, costo >                                                      ▷ O(1)
end function
```

2.2.2. Análisis de complejidad

La complejidad del algoritmo es $O(n^3)$, con n la cantidad de puntos en el plano incluyendo al depósito.

Analizaremos la complejidad del ciclo **while**, del ciclo **for** interno, de las funciones TODOSTRUE, AGREGAR, OBTENERMÁSCERCANO y CALCULARCOSTO.

Como se explicó en la sección 2.2.1, la idea del algoritmo consiste en crear una solución donde se vayan satisfaciendo las demandas de los clientes más cercanos hasta que la capacidad del vehículo no pueda atender ningún cliente más.

El algoritmo comienza definiendo el diccionario *loAgregué* que indica, para cada cliente, si el mismo pertenece a alguna ruta. Como inicialmente ningún cliente pertenece a ninguna ruta, el diccionario se inicializa con todos sus significados en `false`. Cabe destacar que este diccionario está implementado sobre un arreglo convencional, con lo cual, declararlo e inicializarlo, tiene complejidad temporal $O(n)$ en total.

La función TODOSTRUE recibe un arreglo de `bool` y devuelve `true` si todos sus elementos son `true` y, en caso contrario, devuelve `false`. En este caso, como la función TODOSTRUE recibe a *loAgregué* como parámetro, y la longitud del mismo es $O(n)$, la complejidad temporal de la ejecución de la función TODOSTRUE es $O(n)$.

```
function OBTENERMÁSCERCANO(G =< V, E >, capacidadActual, clienteActual, loAgregué)
    mínimo ← −∞                                                                  ▷ O(1)
    clienteMásCercano ← −1                                                       ▷ O(1)
    for 0 <= i < n do                                                            ▷ O(n × ...)
        if ESVÁLIDO(G, clienteActual, i, loAgregué, mínimo, capacidadActual) then   ▷ O(1)
            clienteMásCercano ← i                                                ▷ O(1)
            mínimo ← peso(G, (clienteActual, i))                                 ▷ O(1)
        end if
    end for
    return clienteMásCercano                                                     ▷ O(1)
end function
```

```
function EsVálido(G, clienteActual, i, loAgregué, mínimo, capacidadActual)
    if i ≠ clienteActual then                                                    ▷ O(1)
        if ¬loAgregué_i then                                                     ▷ O(1)
            if peso(G, (clienteActual, i)) < mínimo then                         ▷ O(1)
                if demanda[i] + capacidadActual ≤ capacidadVehículo then         ▷ O(1)
                    return true                                                  ▷ O(1)
                end if
            end if
        end if
    end if
    return false                                                                 ▷ O(1)
end function
```

En el caso en el que la función TodosTrue devuelve `false`, es decir, cuando quedan clientes sin satisfacer, se ejecuta el bloque interno del **while**. En dicho bloque, se define una nueva ruta, la cual comienza en el depósito y, con la función ObtenerMásCercano, se obtiene un cliente que cumpla las siguientes propiedades:

1. El cliente no tiene que pertenecer a ninguna ruta actual.
2. La distancia euclídea del cliente al punto donde se encuentra el vehículo debe ser la mínima entre todos los clientes que no pertenecen a ninguna ruta.
3. La suma de la capacidad actual utilizada por el vehículo con la demanda del cliente no debe superar la capacidad máxima del vehículo.

La función EsVálido debe verificar las condiciones recién mencionadas. Para verificar la primer condición, es decir, que el cliente no pertenezca a ninguna ruta, alcanza con verificar que el arreglo *loAgregué*, evaluado en el índice del cliente, sea falso, cuya complejidad temporal es $O(1)$. Para verificar la segunda condición se debe comparar el peso de la arista que une al cliente con el punto donde se encuentra el vehículo con el peso de la arista que une al cliente más cercano encontrado hasta el momento y el punto donde se encuentra el vehículo. Obtener el peso de dichas aristas en una matriz de adyacencia y luego compararlas tiene una complejidad temporal de $O(1)$. Para verificar la tercer condición se debe sumar la demanda del cliente a la capacidad actual utilizada por el vehículo y comparar si es menor a la capacidad máxima del mismo, cuya complejidad temporal es $O(1)$. Por lo tanto, la función EsVálida, que es utilizada por la función ObtenerMásCercano, tiene una complejidad total de $O(1)$.

La función ObtenerMásCercano contiene un ciclo interno que itera en la cantidad de clientes. En la parte interna del ciclo se llama a la función EsVálida y se actualizan variables sin aportar complejidad. Por lo tanto, la función ObtenerMásCercano tiene complejidad $O(n)$.

Volviendo a la función CVRP-Golosa, el ciclo **for** interno itera en la cantidad de clientes y ejecuta la función ObtenerMásCercano y actualiza variables sin aportar complejidad. Por lo tanto, el **for** interno tiene complejidad $O(n \times n) = O(n^2)$.

Por último, el ciclo **while** externo de la función CVRP-Golosa itera hasta que todos los clientes hayan sido agregados a alguna ruta. En cada iteración del ciclo se agrega un cliente como mínimo, construyendo así la solución donde cada vehículo satisface la demanda de un sólo cliente. Sin embargo, es posible que en una única iteración se agreguen todos los clientes formando una única ruta que satisfaga la demanda de todos los clientes. Como realizamos análisis de peor caso, el ciclo **while** tendrá complejidad $O(n \times n^2) = O(n^3)$. Como conclusión, la complejidad total de la heurística es $O(n^3)$.

2.3. Heurística constructiva de *cluster-first, route-second* (1)

La heurística constructiva de *cluster-first, route-second* consiste en dos partes: primero, generar un conjunto de *clústers* disjuntos, donde agruparemos a los clientes por alguna noción de similaridad, y luego determinar la ruta de un sólo vehículo para cada *clúster*. Para poder obtener siquiera una solución factible usando este método, debemos asegurarnos que la suma de las demandas de todos los clientes de un *clúster* no exceda la capacidad máxima de un vehículo de la flota. Teniendo esto en cuenta, vemos que el problema se reduce a resolver un *TSP* en cada *clúster* o, más específicamente, encontrar un camino

hamiltoneano en el subgrafo inducido generado a partir de tomar los nodos pertenecientes al *clúster* en cuestión, ya que, al tratarse de una heurística, la solución computada es una solución aproximada del problema original.

2.3.1. Algoritmo

Para definir los *clústers* en esta primera heurística de *cluster-first*, *route-second*, utilizaremos el *algoritmo de sweep* (*sweep algorithm*), cuya aplicación tiene sentido en instancias del *CVRP* donde los clientes están ubicados en un plano euclídeo. Para determinar cada *clúster* se traza una línea imaginaria desde el depósito hacia un punto cualquiera, luego se rota dicha línea en sentido horario (o antihorario) y, cuando la línea se topa con un cliente, se lo agrega al último *clúster* sin cerrar. Si la demanda del cliente a agregar hace que la demanda total del *clúster* supere a la capacidad del vehículo entonces se crea un nuevo *clúster* con dicho cliente y se prosigue a rotar la línea y a agregar más clientes al nuevo *clúster*.

Por ejemplo, para la instancia de la Figura 1, una *clusterización* usando el *algoritmo de sweep* es la mostrada en la Figura 6, mientras que la Figura 7 muestra una *clusterización* de la instancia de la Figura 3 usando el mismo algoritmo.

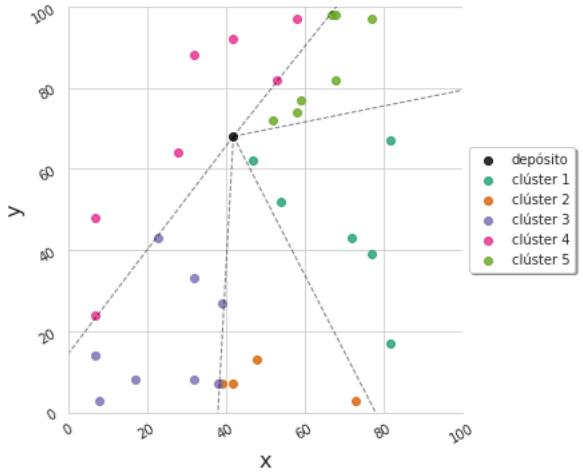

Figura 6: Posible *clusterización* del conjunto de clientes y depósito de ejemplo mostrado en la Figura 1 usando el *algoritmo de sweep*.

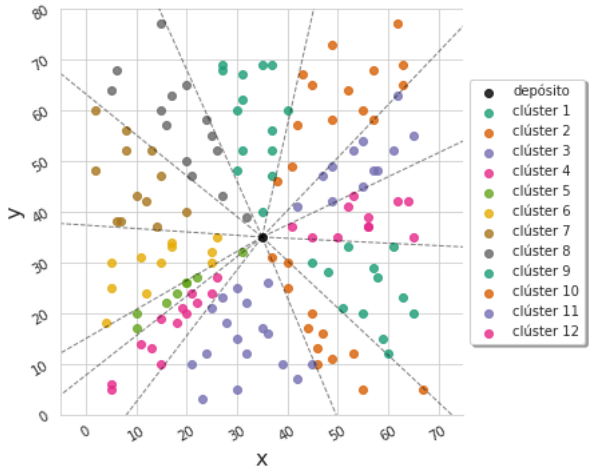

Figura 7: Posible *clusterización* del conjunto de clientes y depósito de ejemplo mostrado en la Figura 3 usando el *algoritmo de sweep*.

Una vez definidos los *clústers*, para cada uno de ellos, obtendremos una ruta de forma heurística utilizando el algoritmo de *vecino más cercano* (*Nearest Neighbor*) al igual que lo hicimos para la heurística

golosa de la sección 2.2. La única diferencia es que en este caso no debemos preocuparnos de la demanda de cada cliente para resolver el ruteo dentro de un *clúster*, ya que la misma *clusterización* nos asegura que todos los clientes de un mismo *clúster* pueden satisfacerse con un único vehículo.

A continuación se presenta el pseudocódigo para el algoritmo recién planteado.

---

function CVRP-CLUSTER-FIRST-ROUTE-SECOND-1($G =< V, E >$, *demandaDeLosClientes*, *capacidadVehículo*, *depósito*)
  rutas $\leftarrow \emptyset$ ▷ $O(1)$
  clientesPolares $\leftarrow$ GETCLIENTESENCOORDENADASPOLARES(V, *depósito*) ▷ $O\big(n \times log(n)\big)$

  clústers $\leftarrow$ GETCLÚSTERSSWEEPALGORITHM(*clientesPolares*, *demandaDeLosClientes*, *capacidadVehículo*) ▷ $O\big(n \times log(n)\big)$
  for *clúster* $\in$ *clústers* **do** ▷ $O(n \times \ldots)$
    ruta $\leftarrow$ NEARESTNEIGHBOR(G, *depósito*, *clúster*) ▷ $O(n^2)$
    rutas $\leftarrow$ *rutas* $\cup \{$*ruta*$\}$ ▷ $O(1)$
  end for

  costo $\leftarrow$ CALCULARCOSTO(*rutas*) ▷ $O(n)$
  return $<$ *rutas*, *costo* $>$ ▷ $O(1)$
end function

2.3.2. Análisis de complejidad

La complejidad del algoritmo es $O(n^3)$, con n la cantidad de puntos en el plano incluyendo al depósito.

Analizaremos la complejidad del ciclo **for** y de las funciones GETCLIENTESENCOORDENADASPOLARES, GETCLÚSTERSSWEEPALGORITHM y NEARESTNEIGHBOR.

En la función GETCLIENTESENCOORDENADASPOLARES se recorre la lista de clientes y, para cada uno de ellos, se calcula sus coordenadas polares en función de sus coordenadas cartesianas respecto del nodo depósito. Luego se inserta el elemento $<$ *cliente*, $r, \theta >$ en la cola de prioridad *clientes*, que está implementada con un *min heap* con lo cual, dicha inserción, tiene complejidad temporal $O\big(log(n)\big)$. La función de comparación del *min heap* simplemente compara los valores de θ en cada inserción de un elemento y mantiene en el *tope* aquel elemento con valor de θ mínimo entre todos y, de esta forma, aseguramos que los clientes siguen un orden que cumple lo especificado en la sección 2.3.1. Como la cantidad de clientes es $O(n)$ y por cada uno de ellos se realiza una operación de complejidad $O\big(log(n)\big)$ entonces la complejidad temporal final de la función GETCLIENTESENCOORDENADASPOLARES es $O\big(n \times log(n)\big)$.

---

function GETCLIENTESENCOORDENADASPOLARES(*clientes*, *depósito*)
  clientesPolares $\leftarrow \emptyset$ ▷ $O(1)$
  for *cliente* $\in$ *clientes* **do** ▷ $O(n \times \ldots)$
    xRelativo $\leftarrow$ *cliente.x* $-$ *depósito.x* ▷ $O(1)$
    yRelativo $\leftarrow$ *cliente.y* $-$ *depósito.y* ▷ $O(1)$
    $r \leftarrow \sqrt{(\textit{xRelativo})^2 + (\textit{yRelativo})^2}$ ▷ $O(1)$
    $\theta \leftarrow \arctan\left(\frac{\textit{yRelativo}}{\textit{xRelativo}}\right)$ ▷ $O(1)$
    clientesPolares$.push\big(<$ *cliente*, $r, \theta >\big)$ ▷ $O\big(log(n)\big)$
  end for
  return *clientesPolares* ▷ $O(1)$
end function

La función GETCLÚSTERSSWEEPALGORITHM realiza la propia *clusterización* de los clientes según el método explicado en la sección 2.3.1. En la misma se inicializa un diccionario *clústers* de tamaño $O(n)$ en 0 implementado con un arreglo, que representa, para cada cliente i, a qué número de *clúster* pertenece, siendo el primer *clúster* el número 1. Luego se recorre la cola de prioridad *clientes*, cuyo tamaño es $O(n)$, y se realizan una serie de operaciones cuya complejidad total es ▷ $O\big(log(n)\big)$. Finalmente se retorna el diccionario de *clústers*. Como la cantidad de clientes es $O(n)$ y por

cada uno de ellos se realiza una operación de complejidad $O(log(n))$ entonces la complejidad temporal final de la función GETCLÚSTERSSWEEPALGORITHM es $O(n \times log(n))$.

```
function GETCLÚSTERSSWEEPALGORITHM(clientes, demandaDeLosClientes, capacidadVehículo)
    clústers_i ← 0                                  ∀ 1 ≤ i ≤ n          ▷ O(n)
    últimoClúster ← 1                                                    ▷ O(1)
    capacidadRestante ← capacidadVehículo                                ▷ O(1)
    do                                                                   ▷ O(n × ...)
        cliente ← clientes.top()                                         ▷ O(1)
        if demandaDeLosClientes[cliente] > capacidadRestante then        ▷ O(1)
            últimoClúster ← últimoClúster + 1                            ▷ O(1)
            capacidadRestante ← capacidadVehículo                        ▷ O(1)
        end if
        clústers[cliente] ← últimoClúster                                ▷ O(1)
        capacidadRestante ← capacidadRestante − demandaDeLosClientes[cliente]  ▷ O(1)

        clientes.pop()                                                   ▷ O(log(n))
    while |clientes| > 0                                                 ▷ O(1)
    return clústers                                                      ▷ O(1)
end function
```

La función NEARESTNEIGHBOR determina una ruta para el clúster dado mediante la heurística de *Nearest Neighbor*. La misma cuenta con un ciclo que itera mientras queden nodos del *clúster* sin agregar a la ruta y, en caso afirmativo, se busca el nodo más cercano al último nodo de la ruta entre los nodos que aún no están en la misma mediante la función OBTENERMÁSCERCANOENCLÚSTER, cuya complejidad temporal es $O(n)$. Como la cantidad de clientes en un *clúster* es, en el peor caso, $O(n)$, y cada nodo obtenido mediante la función OBTENERMÁSCERCANOENCLÚSTER se contabiliza en la variable *nodosAgregados*, se concluye que el ciclo principal se ejecuta $O(n)$ veces. Luego, como el ciclo principal se ejecuta $O(n)$ veces y las operaciones del ciclo tienen complejidad $O(n)$, entonces la complejidad temporal final de la función NEARESTNEIGHBOR es $O(n^2)$.

```
function NEARESTNEIGHBOR(G, depósito, clúster)
    ruta ← [ ]                                                           ▷ O(1)
    visitados_i ← false                  ∀ 1 ≤ i ≤ |clúster|              ▷ O(n)
    nodosAgregados ← 0                                                   ▷ O(1)

    AGREGAR(ruta, depósito)                                              ▷ O(1)
    últimaVisita ← depósito                                              ▷ O(1)

    do                                                                   ▷ O(n × ...)
        clienteMásCercanoSinVisitar ← OBTENERMÁSCERCANOENCLÚSTER(G, últimaVisita
                                                          clúster, visitados)  ▷ O(n)
        AGREGAR(ruta, clienteMásCercano)                                 ▷ O(1)
        nodosAgregados ← nodosAgregados + 1                              ▷ O(1)
    while nodosAgregados < |clúster|                                     ▷ O(1)

    AGREGAR(ruta, depósito)                                              ▷ O(1)

    return ruta                                                          ▷ O(1)
end function
```

La función OBTENERMÁSCERCANOENCLÚSTER busca el cliente más cercano al nodo especificado que pertenezca al clúster indicado por parámetro y que no haya sido visitado, es decir, que no pertenezca a ninguna ruta actual. Dicha función está formada únicamente por un ciclo que itera en los elementos del clúster en cuestión y, por cada elemento, se realizan operaciones de complejidad temporal $O(1)$. Es

importante tener en cuenta que, para lograr una complejidad temporal de $O(1)$ al obtener el `peso` de una arista, el grafo G se representa con una *matriz de adyacencia*. Como la cantidad de clientes en un clúster es, en el peor caso, $O(n)$ y, por cada uno de ellos, se realiza una operación de complejidad $O(1)$, entonces la complejidad temporal final de la función OBTENERMÁSCERCANOENCLÚSTER es $O(n)$.

```
function ObtenerMásCercanoEnClúster(G, nodo, clúster, visitados)
    nodoMásCercano ← null                                              ▷ O(1)

    for cliente ∈ clúster do                                           ▷ O(n × ...)
        if ¬visitados[cliente] then                                    ▷ O(1)
            if nodoMásCercano = null then                              ▷ O(1)
                nodoMásCercano ← cliente                               ▷ O(1)
            else
                if peso(G, (nodo, cliente)) < peso(G, (nodo, nodoMásCercano)) then   ▷ O(1)
                    nodoMásCercano ← cliente                           ▷ O(1)
                end if
            end if
        end if
    end for
    return nodoMásCercano                                              ▷ O(1)
end function
```

Entonces, como la función CVRP-CLUSTER-FIRST-ROUTE-SECOND-1 tiene un ciclo cuyo bloque interno se ejecuta $O(n)$ veces, y cada bloque interno tiene complejidad $O(n^2)$, se obtiene una complejidad total de $O(n^3)$.

2.4. Heurística constructiva de *cluster-first, route-second* (2)

2.4.1. Algoritmo

Para definir los *clústers* en esta segunda heurística de *cluster-first, route-second*, utilizaremos una variante del algoritmo de *clusterización* especificado en la sección 2.2 de [1], cuya aplicación tiene sentido en instancias del *CVRP* donde los clientes están ubicados en un plano euclídeo. El algoritmo consiste en agrupar clientes según su ubicación en el plano, de forma tal que los *clústers* reprensenten grupos de clientes que son *cercanos* entre sí. Para lograr instancias factibles, agregaremos un paso extra al algoritmo mencionado, que consiste en *clusterizar* según capacidad de los vehículos, de forma tal que un *clúster* generado por el algoritmo será *sub-clusterizado* para que cada *sub-clúster* contenga clientes cuya demanda pueda satisfacerse con un único vehiculo.

Por ejemplo, para la instancia de la Figura 1, una *clusterización* usando el algoritmo recién presentado es la mostrada en la Figura 8, mientras que la Figura 9 muestra una *clusterización* de la instancia de la Figura 3 usando el mismo algoritmo.

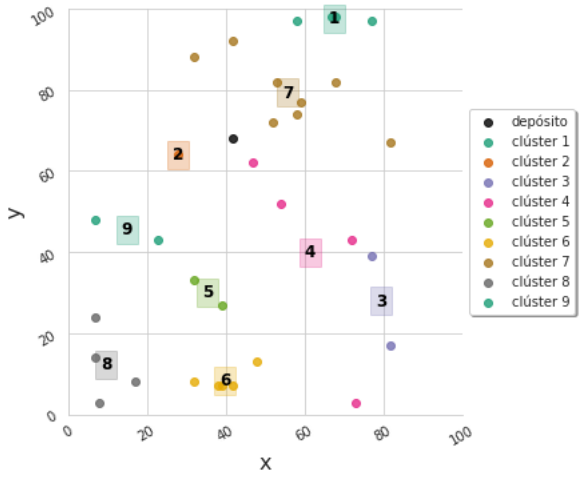

Figura 8: Posible *clusterización* del conjunto de clientes y depósito de ejemplo mostrado en la Figura 1 usando el algoritmo de *clusterización* presentado en [1] (con modificaciones para introducir capacidad de vehículos y demanda de los clientes) con la primer versión de *eje inconsistente* y un tamaño de *vecindad* y *exceso de desvío estándar* de 2 y 3 respectivamente.

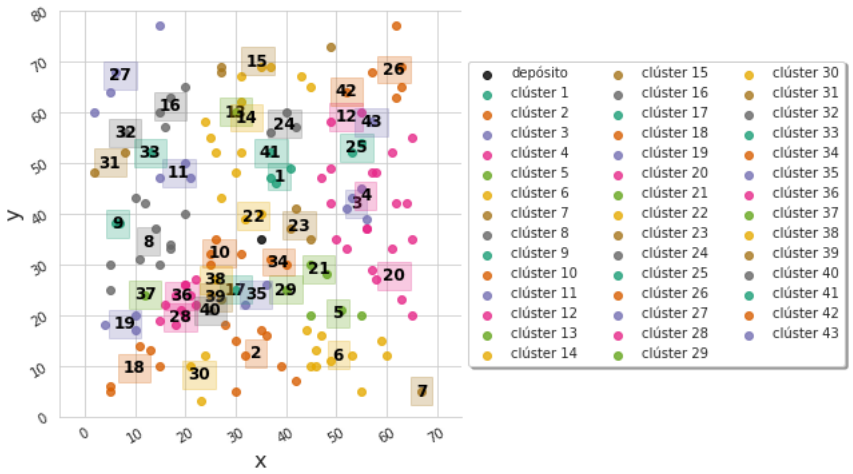

Figura 9: Posible *clusterización* del conjunto de clientes y depósito de ejemplo mostrado en la Figura 3 usando el algoritmo de *clusterización* presentado en [1] (con modificaciones para introducir capacidad de vehículos y demanda de los clientes) con la primer versión de *eje inconsistente* y un tamaño de *vecindad* y *exceso de desvío estándar* de 2 y 3 respectivamente.

Una vez definidos los *clústers*, para cada uno de ellos, utilizaremos un método heurístico para generar rutas, el cual consiste en simplemente tomar la primer solución factible que se encuentre. Notar que no debemos preocuparnos de la demanda de cada cliente para resolver el ruteo dentro de un *clúster*, ya que la misma *clusterización* nos asegura que todos los clientes de un mismo *clúster* pueden satisfacerse con un único vehículo.

A continuación se presenta el pseudocódigo para el algoritmo recién planteado.

function CVRP-CLUSTER-FIRST-ROUTE-SECOND-2($G =< V, E >$, *demandaDeLosClientes*, *capacidadVehículo*, *depósito*)
  rutas $\leftarrow \emptyset$ ▷ $O(1)$

  clústers $\leftarrow$ GETCLÚSTERSTP2ALGORITHM(V, *demandaDeLosClientes*, *capacidadVehículo*) ▷ $O(n^3)$
  for *clúster* $\in$ *clústers* **do** ▷ $O(n \times \ldots)$
    ruta $\leftarrow$ GETPRIMERSOLUCIÓNFACTIBLE(G, *depósito*, *clúster*) ▷ $O(n)$
    rutas $\leftarrow$ *rutas* $\cup \{$*ruta*$\}$ ▷ $O(1)$
  end for

  costo $\leftarrow$ CALCULARCOSTO(*rutas*) ▷ $O(n)$
  return $<$ *rutas*, *costo* $>$ ▷ $O(1)$
end function

2.4.2. Análisis de complejidad

La complejidad del algoritmo es $O(n^3)$, con n la cantidad de puntos en el plano incluyendo al depósito.

Analizaremos la complejidad del ciclo **for** y de las funciones GETCLÚSTERSTP2ALGORITHM y GETPRIMERSOLUCIÓNFACTIBLE.

La función GETCLÚSTERSTP2ALGORITHM es análoga a la función CLUSTERIZARCONPRIM mostrada en la sección 2.2 de [1]. La única diferencia es que GETCLÚSTERSTP2ALGORITHM verifica que los clientes pertenecientes a los *clústers* formados no excedan la capacidad total del vehículo. Esta comprobación se hace a través de un diccionario *demandas* que está implementado sobre un arreglo donde, para cada cliente, podemos saber su demanda en $O(1)$, mientras que, a medida que un *clúster* se va "llenando", contabilizaremos la capacidad restante en un contador, cuyas operaciones de suma y resta tienen complejidad $O(1)$. Luego, si la demanda de un cliente supera la capacidad restante del vehículo, se cierra el *clúster*, se incrementa la cantidad de *clústers* totales y se agrega el cliente al nuevo *clúster*. Estas operaciones tienen complejidad $O(1)$, ya que el diccionario *clústers*, que indica, para cada cliente, a qué *clúster* pertenece, se implementa sobre un arreglo, y el contador de *clústers* es un número entero que hace las veces de índice para el diccionario, con lo cual incrementar el contador tiene complejidad $O(1)$. Entonces, como la función CLUSTERIZARCONPRIM mostrada en la sección 2.2 de [1] tiene complejidad $O(n^3)$ y la función GETCLÚSTERSTP2ALGORITHM es una extensión de la primera agregando operaciones de complejidad $O(1)$, se concluye que la complejidad de GETCLÚSTERSTP2ALGORITHM es $O(n^3)$.

La función GETPRIMERSOLUCIÓNFACTIBLE obtiene la primer solución factible para un *clúster*, la cual consiste en recorrer la lista de clientes del *clúster* dado y agregar cada uno de ellos en el arreglo ruta. Como la lista de clientes de un *clúster* no contiene elementos repetidos y ninguno de ellos pertenece a otro *clúster*, se concluye que la solución generada es factible. Además, como la cantidad de clientes en un *clúster* es $O(n)$, el ciclo principal se ejecuta $O(n)$ veces. Luego, como el ciclo principal se ejecuta $O(n)$ veces y las operaciones del ciclo tienen complejidad $O(1)$, entonces la complejidad temporal final de la función GETPRIMERSOLUCIÓNFACTIBLE es $O(n)$.

function GETPRIMERSOLUCIÓNFACTIBLE(G, *depósito*, *clúster*)
  ruta $\leftarrow$ `[ ]` ▷ $O(1)$

  AGREGAR(*ruta*, *depósito*) ▷ $O(1)$
  for *cliente* $\in$ *clúster* **do** ▷ $O(n \times \ldots)$
    AGREGAR(*ruta*, *cliente*) ▷ $O(1)$
  end for
  AGREGAR(*ruta*, *depósito*) ▷ $O(1)$

  return *ruta* ▷ $O(1)$
end function

Entonces la función CVRP-CLUSTER-FIRST-ROUTE-SECOND-2 tiene complejidad temporal $O(n^3)$, que

surge de que GETCLÚSTERSTP2ALGORITHM tenga complejidad $O(n^3)$.

2.5. Metaheurística de Simulated Annealing

2.5.1. Algoritmo

El algoritmo de Simulated Annealing simula el proceso de templado del metal. Este proceso consiste en calentar el metal a una temperatura elevada y enfriarlo gradualmente hasta llegar a un estado de propiedades físicas preferibles. El calor causa que los átomos aumenten su energía y que puedan así desplazarse de sus posiciones iniciales (un mínimo local de energía), en cambio, al enfriarse los átomos, disminuirán su energía y comenzaran a desplazarse cada vez menos.

Si esta analogía la llevamos al problema actualmente estudiado, la idea de la metaheurística es que, cuando haya una temperatura elevada, se permita explorar la mayor cantidad de espacio posible en la vecindad de soluciones, para no quedarse en una solución mínima local y, a medida que la temperatura vaya disminuyendo, dejar de explorar vecindades cuyas soluciones no minimicen la distancia recorrida.

Comenzaremos obteniendo una solución inicial aproximada a partir de alguna heurística, por ejemplo la de *savings* presentada en la sección 2.1. Además inicializaremos los parámetros de temperatura de acuerdo a la función de temperatura que utilicemos.

A continuación se calculará una solución S' que sea parte de la vecindad de la solución inicial, y se computará la diferencia de costo entre ambas. Con ella y la temperatura actual, se determinará si la solución S' será aceptada o no y, si es aceptada, la solución actual será S' a partir de ese momento. También se guardará una solución S_m, que corresponderá con la mejor solución visitada.

Finalmente se actualizará la temperatura actual de acuerdo a la función de temperatura utilizada. Todo esto estará en un ciclo que terminará cuando la temperatura alcance el valor final que especifiquemos, o bien la misma sea lo suficientemente baja.

A continación se presenta el pseudocódigo para el algoritmo recién planteado.

---

function CVRP-SIMULATED-ANNEALING($G =< V, E >$, *demandaClientes*, *capacidadVehículo*, *depósito*)
  $S_{inicial} \leftarrow$ SOLUCIÓNINICIALSAVINGS(*G*, *demandaClientes*, *capacidadVehículo*, *depósito*) ▷ $O(n^3)$
  $Temp_{inicial}, Temp_{final} \leftarrow$ INICIALIZARTEMPERATURA(*S_inicial*) ▷ $O(1)$
  $Temp_{actual} \leftarrow Temp_{inicial}$ ▷ $O(1)$
  $S_{actual} \leftarrow S_{inicial}$ ▷ $O(1)$
  $S_{mejor} \leftarrow S_{inicial}$ ▷ $O(1)$
  while $Temp_{actual} > Temp_{final}$ **do** ▷ $O(\log C_i \times \ldots)$
    $S' \leftarrow$ OBTENERSOLUCIÓNDEVECINDAD(*G*, *demandaClientes*, *capacidadVehículo*, *depósito*) ▷ $O(n^3)$
    if FUNCIÓNDEACEPTACIÓN$\Big(C(S') - C(S), Temp_{actual}\Big)$ **then** ▷ $O(n)$
      $S_{actual} \leftarrow S'$ ▷ $O(1)$
      if $C(S_{actual}) < C(S_{mejor})$ **then** ▷ $O(n)$
        $S_{mejor} \leftarrow S_{actual}$ ▷ $O(1)$
      end if
      $Temp_{actual} \leftarrow$ ACTUALIZARTEMPERATURA$(Temp_{actual})$ ▷ $O(1)$
    end if
  end while
  return S_{mejor} ▷ $O(1)$
end function

Se puede ver que el algoritmo va a depender de que elijamos como funciones de VECINDAD, ACEPTACION y TEMPERATURA.

Vecindad :

La función de vecindad nos debe devolver una solución factible, que sea 'cercana' a nuestra solución actual. Para el trabajo implementamos el método de 1-Interchange, descripto en el paper provisto por la catedra [2].

El método de 1-Interchange lo que va a hacer es armar todas las soluciones posibles que se puedan generar a partir de la original realizando movimientos llamados $shifteos$ y $interchanges$. Un $shifteo$ va a poder ser (1,0),(0,1) y un $interchange$ (1,1). En los (1,0), la ruta sobre la cual se esta iterando de la

solución original va a conseguir un cliente mas, y este va a ser removido de la ruta que estaba. Los (0,1) van a ser el caso contrario, y los (1,1) van a ser los que las rutas se intercambien. Es importante también eliminar del espacio de busqueda las soluciones que se formen que no sean correctas, esto va a pasar cuando a una ruta se le agregue un cliente con el cual supera la capacidad de los camiones.

Temperatura :

La función de temperatura esta tomada del paper [3], en la cual el parámetro de temperatura inicial dependerá de la solución inicial, y luego se setea un coeficiente que va a ir reduciendo la temperatura actual hasta que llege a un valor fijo muy bajo.

Los parámetros se inicializaran de la siguiente manera

$$Temp_{Inicial} \leftarrow C(soluciónInicial) \times 0{,}2$$
$$Temp_{Final} \leftarrow 0{,}001$$
$$\alpha \leftarrow 0{,}95$$

Siendo $C(solución)$ el costo asociado a una solución. En cada paso la temperatura se actualizará simplemente haciendo

$$Temp_{Actual} \leftarrow Temp_{Actual} \times \alpha$$

Aceptación :

La función de aceptación utilizada fue:

$$e^{-\Delta/Temp_{actual}} > \theta$$

Donde θ es un número real aleatorio entre el 0 y el 1, y Δ es C(S')-C(S).

2.5.2. Análisis de complejidad

Podemos ver en el pseudocódigo que la complejidad de el algoritmo va a depender de las complejidades de la función de vecindad, así como la cantidad de iteraciones que genere la función de temperatura. Analizaremos la complejidad de cada una de estas partes, y veremos cual es la complejidad resultante.

A continación se presenta el pseudocódigo para la generación del vecindario utilizando 1-Interchange.

```
function VECINDAD-1-INTERCHANGE(solución, G =< V, E >, demandaDeLosClientes, capacidadVehículo, depósito)
    res10, res01, res11 ← soluciones[]
    for i = 0, i < solución.size() do
        for j = i + 1; j < solución.size() do
            for k = 1; k < solución[i].size() − 1 do
                for l = 1; l < solución[j].size() − 1 do                    ▷ O(n^2 * ...)
                    sol1 ← solución                                         ▷ O(n)
                    INSERTAREN(sol1[j][l], sol[i][k])                       ▷ O(n)
                    BORRAR(sol1[i][k])                                      ▷ O(n)
                    BORRARRUTASVACIASEINFACTIBLES(sol1)                     ▷ O(n)

                    sol2 ← solución                                         ▷ O(n)
                    INSERTAREN(sol2[i][k], sol[j][l])                       ▷ O(n)
                    BORRAR(sol2[j][l])                                      ▷ O(n)
                    BORRARRUTASVACIASEINFACTIBLES(sol2)                     ▷ O(n)

                    sol3 ← solución                                         ▷ O(n)
                    aux ← sol3[i].ruta[k]                                   ▷ O(n)
                    sol3[i].ruta[k] ← sol3[j].ruta[l]                       ▷ O(n)
                    sol3[j].ruta[l] ← aux                                   ▷ O(n)
                    BORRARRUTASINFACTIBLES(sol3)                            ▷ O(n)

                    AGREGARA(res01, sol1)                                   ▷ O(n)
                    AGREGARA(res10, sol2)                                   ▷ O(n)
                    AGREGARA(res11, sol3)                                   ▷ O(n)
                end for
            end for
        end for
    end for
    return CONCATENAR(sol01, sol10, sol11)                                  ▷ O(n)
end function
```

Por esto podemos decir que la complejidad de generar este vecindario va a ser de $O(n^3)$.

Por la parte de la temperatura, sabemos que se va a inicializar en un valor que depende de la solución inicial. Como para generarla usamos savings, un método ya descripto en la sección 2.1, podemos decir que en un peor caso el costo de la solución va a ser la sumatoria de todos los caminos depósito-cliente-depósito.

$$\sum_{i=1}^{n} dist(depósito - cliente_i - depósito) = \sum_{i=1}^{n} 2(depósito - cliente_i) \leq$$
$$\leq n \times 2 \times max(dist(depósito - cliente_i) \forall i \in n)$$

A esta cota del valor inicial la llamaremos C_i, y dependerá de la instancia del problema. Ahora analizaremos la cantidad de iteraciones que va a realizar el ciclo de temperatura. Sabemos que la ultima iteración será cuando la temperatura sea menor o igual a $0{,}001$, y en cada paso la temperatura actual se multiplicara por α, por lo que llamando i a la cantidad de iteraciones, podemos decir que cuando finalice el ciclo tendremos

$$0{,}001 \leq C_i \times 0{,}95^i \iff i \geq \frac{\log 0{,}001}{\log 0{,}95} - \frac{\log C_i}{\log 0{,}95}$$
$$i \geq k + a \log C_i$$

Con k y a siendo constantes positivas.

Viendo todo esto, podemos decir que la cantidad de veces que se ejecute el ciclo va a estar acotado por $O(logC_i)$, y la complejidad de ejecutar una iteración era $O(n^3)$, por lo que la complejidad final de algoritmo será $O(n^3 \times log(C_i))$.

3. Análisis de resultados

Para analizar los resultados de los algoritmos, se tomaron instancias que contienen entre 32 y 80 clientes, contando al depósito. Dichas instancias fueron tomadas del **Set A** de instancias recomendadas por la cátedra ([5]). Creemos que este set de instancias es apropiado para hacer análisis de los algoritmos heurísticos ya que cada instancia está generada al azar, con lo cual no deberían favorecer a ninguna heurística en particular. La capacidad de los vehículos es 100 para todas las instancias, y la demanda de cada cliente en el set utilizado sigue una distribución uniforme $U(1, 30)$.

3.1. Heurística de *savings*

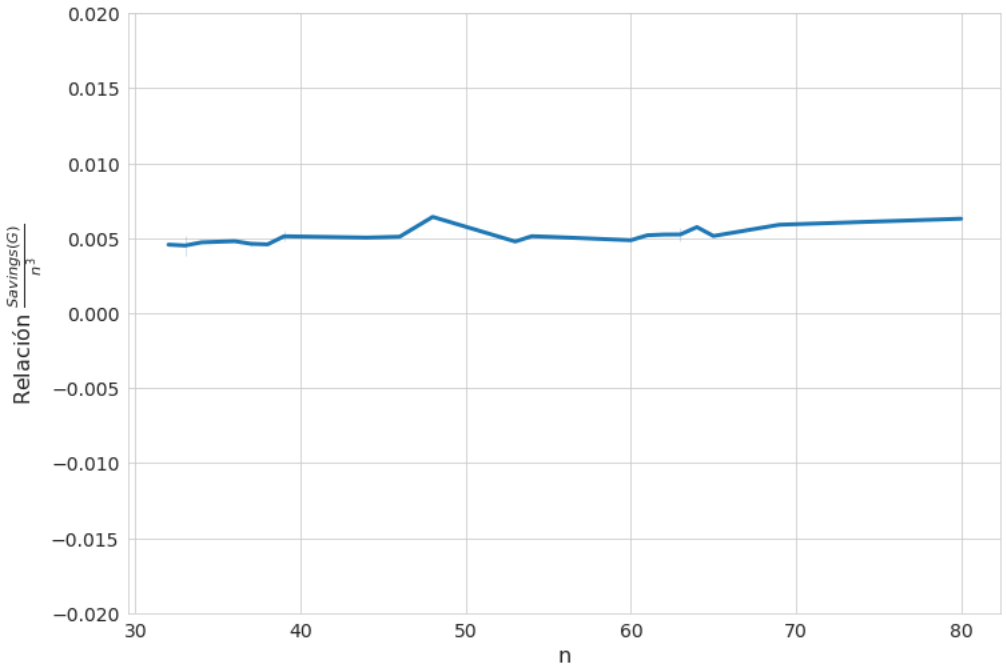

Figura 10: Relación entre el tiempo de ejecución de CVRP-SAVINGS y la función $f(n) = n^3$ para valores $32 \leq n \leq 80$

Notar que la función mostrada en la Figura 10 parece comportarse como una constante a medida que aumenta el valor de n con ciertas fluctuaciones, no tan apreciables, que podemos deducir que surgen de las diferencias que pueden existir entre las instancias del set escogido. A través de este gráfico se puede ver que ambas funciones están relacionadas entre sí, y el mismo permite visualizar que la complejidad temporal de CVRP-SAVINGS se comporta de forma similar a la función $f(n) = n^3$.

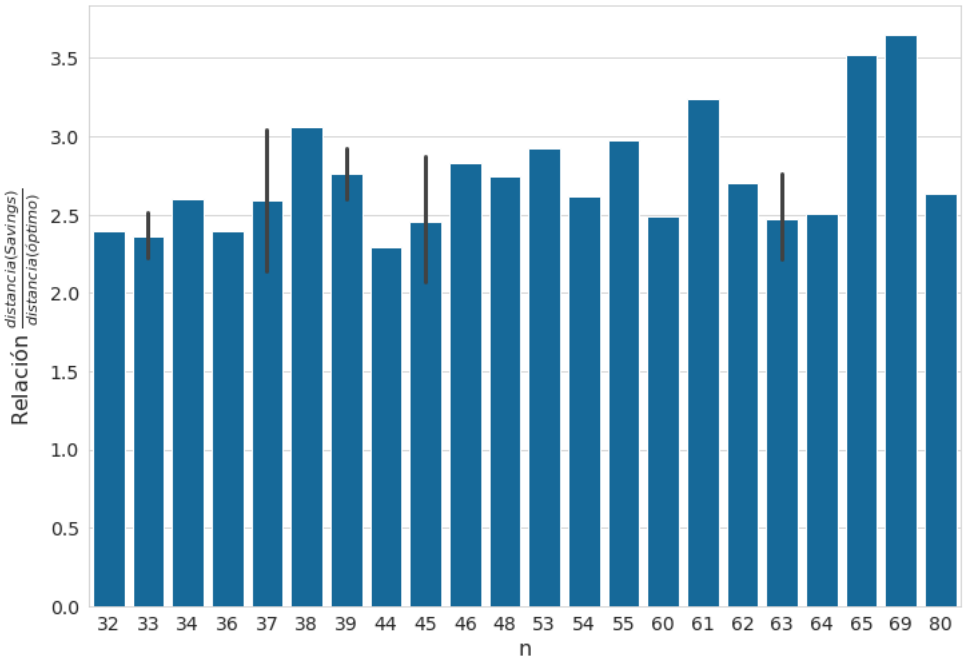

Figura 11: Relación entre los resultados de CVRP-SAVINGS
y los resultados óptimos para valores $32 \leq n \leq 80$

Por otro lado, en la Figura 11 podemos ver que los resultados de CVRP-SAVINGS se encuentran por encima de 2 veces el resultado óptimo para las instancias evaluadas, lo cual indica que la heurística o la implementación escogida no generan un resultado tan preciso.

3.2. Heurística *golosa*

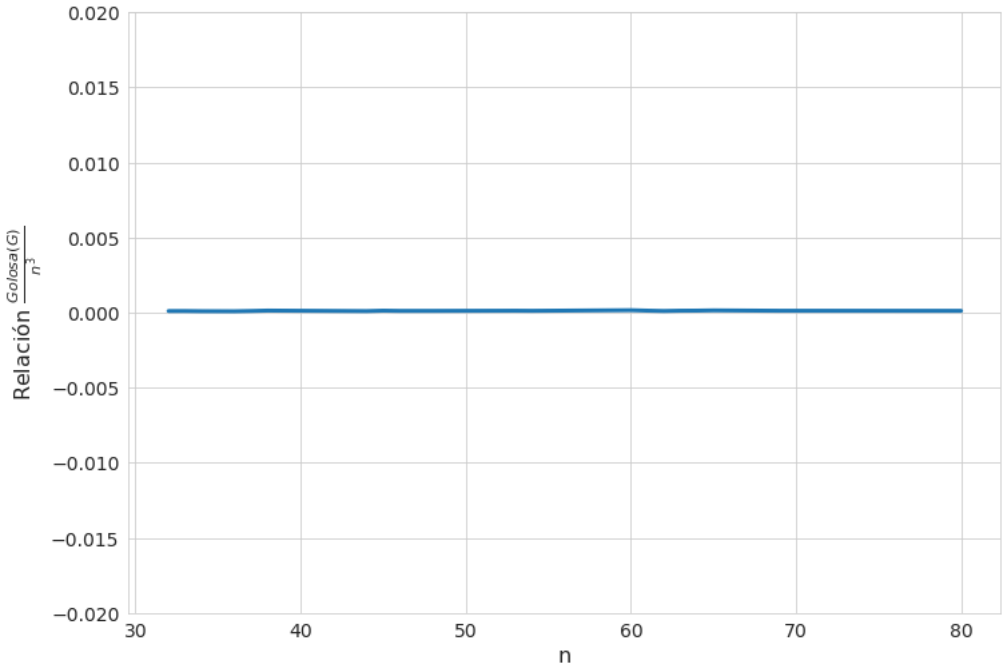

Figura 12: Relación entre el tiempo de ejecución de CVRP-GOLOSA
y la función $f(n) = n^3$ para valores $32 \leq n \leq 80$

Notar que la función mostrada en la Figura 12 tiende a ser constante a medida que aumenta el valor de n. A través de este gráfico se puede ver que ambas funciones están relacionadas entre sí, y el mismo

permite visualizar que la complejidad temporal de CVRP-GOLOSA se comporta de forma similar a la función $f(n) = n^3$.

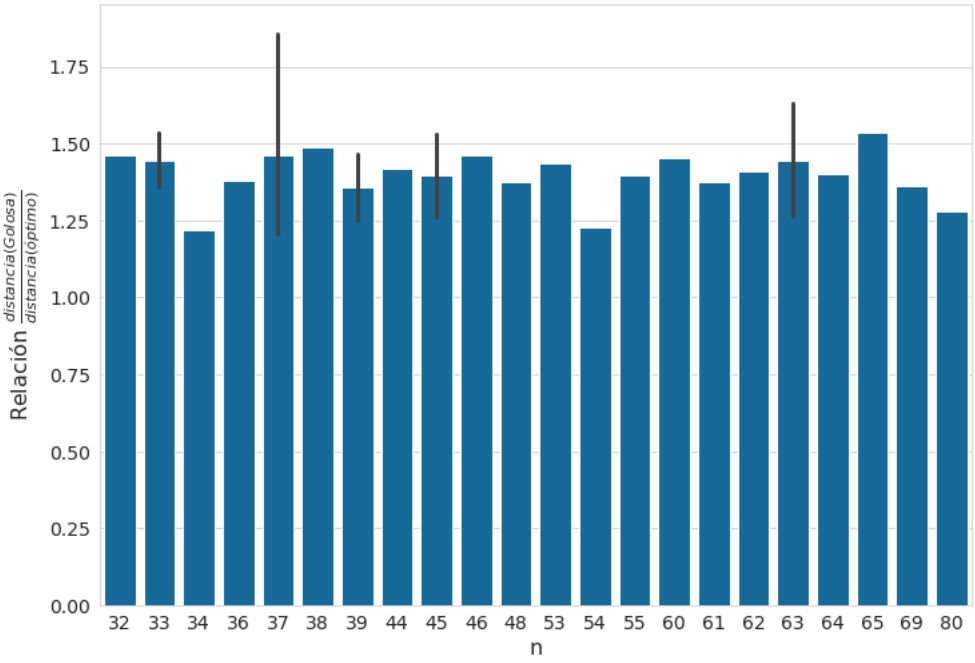

Figura 13: Relación entre los resultados de CVRP-GOLOSA
y los resultados óptimos para valores $32 \leq n \leq 80$

Por otro lado, en la Figura 13 podemos ver que los resultados de CVRP-GOLOSA se encuentran, en promedio, por debajo de $1,5$ veces el resultado óptimo para las instancias evaluadas.

3.3. Heurística constructiva de *cluster-first, route-second* (1)

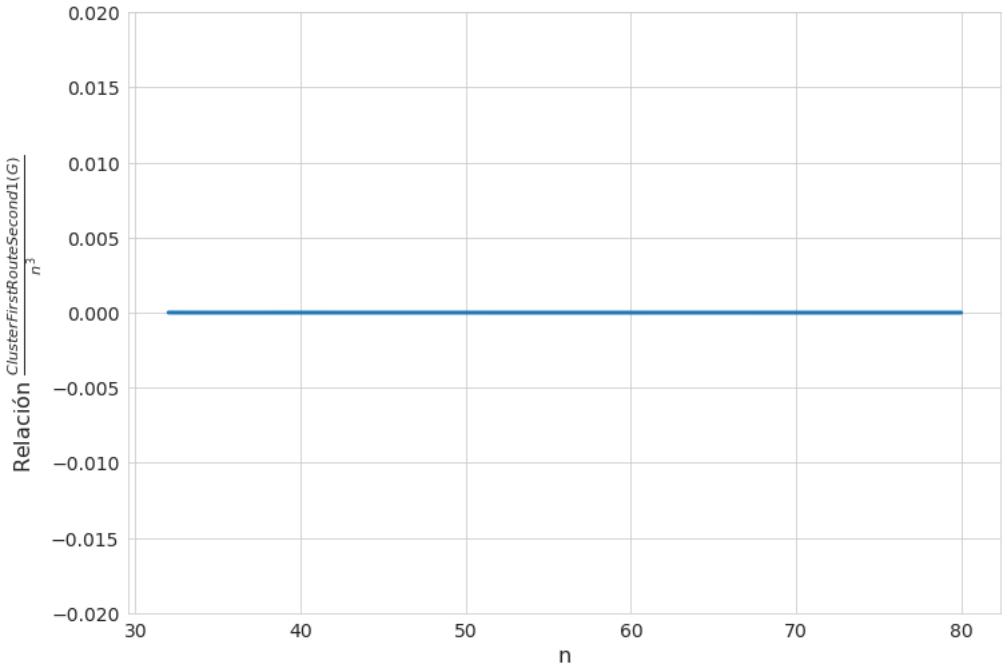

Figura 14: Relación entre el tiempo de ejecución de CVRP-CLUSTER-FIRST-ROUTE-SECOND-1
y la función $f(n) = n^3$ para valores $32 \leq n \leq 80$

Notar que la función mostrada en la Figura 14 tiende a ser constante a medida que aumenta el valor de n. A través de este gráfico se puede ver que ambas funciones están relacionadas entre sí, y el mismo permite visualizar que la complejidad temporal de CVRP-CLUSTER-FIRST-ROUTE-SECOND-1 se comporta de forma similar a la función $f(n) = n^3$.

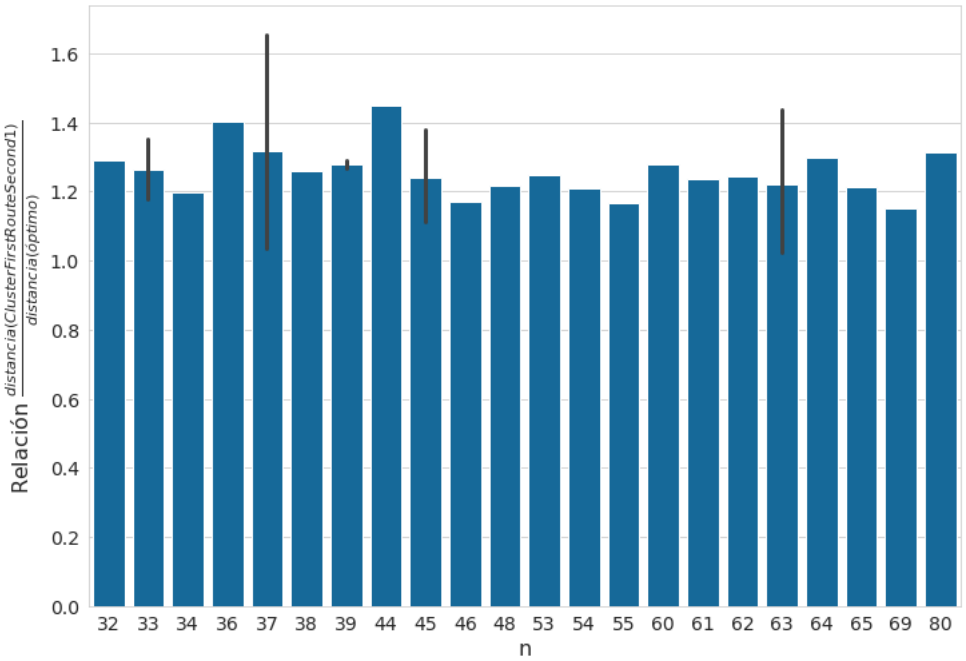

Figura 15: Relación entre los resultados de CVRP-CLUSTER-FIRST-ROUTE-SECOND-1 y los resultados óptimos para valores $32 \leq n \leq 80$

Por otro lado, en la Figura 15 podemos ver que los resultados de CVRP-CLUSTER-FIRST-ROUTE-SECOND-1 se encuentran, en promedio, por debajo de $1,4$ veces el resultado óptimo para las instancias evaluadas.

3.4. Heurística constructiva de *cluster-first, route-second* (2)

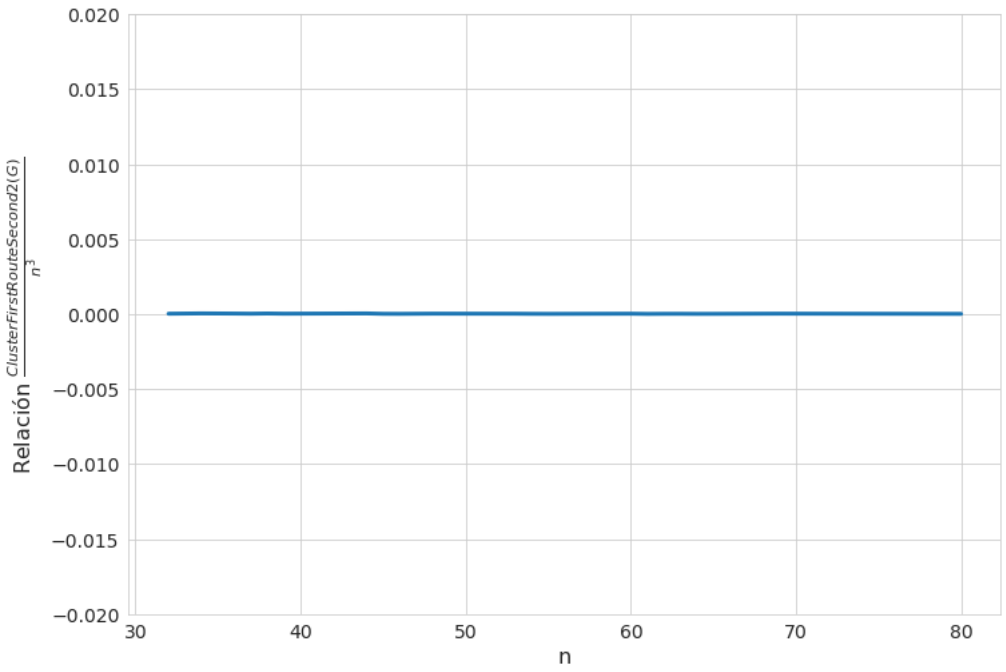

Figura 16: Relación entre el tiempo de ejecución de CVRP-CLUSTER-FIRST-ROUTE-SECOND-2 y la función $f(n) = n^3$ para valores $32 \leq n \leq 80$

Notar que la función mostrada en la Figura 16 tiende a ser constante a medida que aumenta el valor de n. A través de este gráfico se puede ver que ambas funciones están relacionadas entre sí, y el mismo permite visualizar que la complejidad temporal de CVRP-CLUSTER-FIRST-ROUTE-SECOND-2 se comporta de forma similar a la función $f(n) = n^3$.

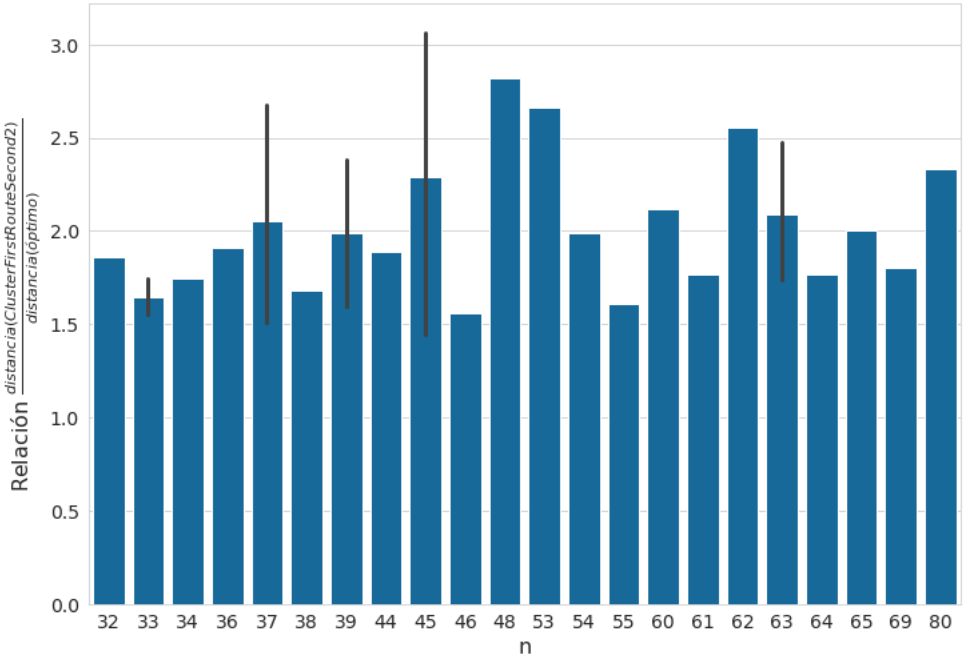

Figura 17: Relación entre los resultados de CVRP-CLUSTER-FIRST-ROUTE-SECOND-2 y los resultados óptimos para valores $32 \leq n \leq 80$

Por otro lado, en la Figura 17 podemos ver que los resultados de CVRP-CLUSTER-FIRST-ROUTE-

SECOND-2 varían considerablemente frente al resultado óptimo para las instancias evaluadas.

3.5. Metaheurística de Simulated Annealing

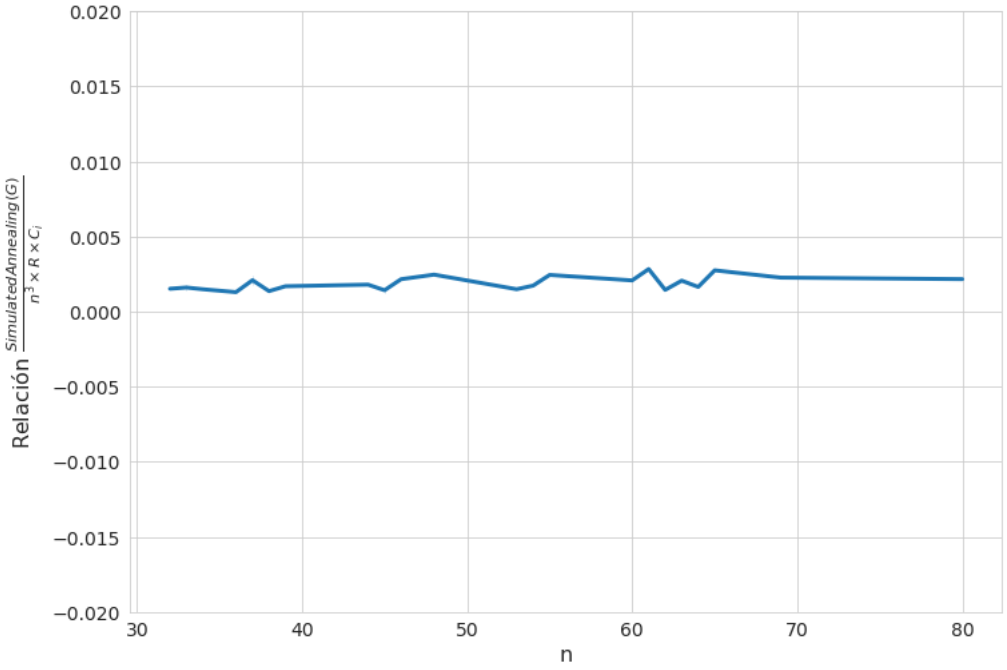

Figura 18: Relación entre el tiempo de ejecución de CVRP-SIMULATED-ANNEALING y la función $f(n) = n^3 \times R \times C_i$ para valores $32 \leq n \leq 80$

Notar que la función mostrada en la Figura 18 parece comportarse como una constante a medida que aumenta el valor de n con ciertas fluctuaciones que podemos deducir que surgen de la aleatoreidad propia que tiene el algoritmo. A través de este gráfico se puede ver que ambas funciones están relacionadas entre sí, y el mismo permite visualizar que la complejidad temporal de CVRP-SIMULATED-ANNEALING se comporta de forma similar a la función $f(n) = n^3 \times R \times C_i$.

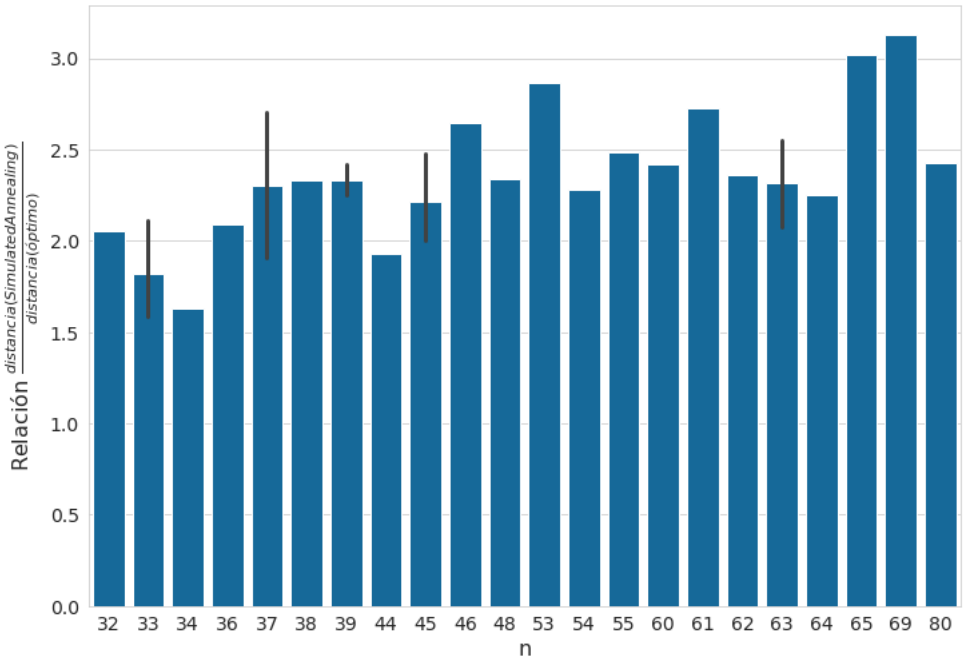

Figura 19: Relación entre los resultados de CVRP-SIMULATED-ANNEALING y los resultados óptimos para valores $32 \leq n \leq 80$

Por otro lado, en la Figura 19 podemos ver que los resultados de CVRP-SIMULATED-ANNEALING varían considerablemente frente al resultado óptimo para las instancias evaluadas.

4. Experimentación

4.1. Experimento 1: Instancias que representan casos reales

En el primer experimento vamos a comparar las heurísticas presentadas para instancias que representen casos reales. Los casos que vamos a abordar son aquellos donde el depósito representa una central de distribución de mercadería ubicada, aproximadamente, en el centro de los clientes, y los mismo se encuentran agrupados de forma tal que simulan ser distintas ciudades.

Estos casos podrían darse en las grandes empresas de envíos de mercadería como FedEx, DHL, UPS, etc.

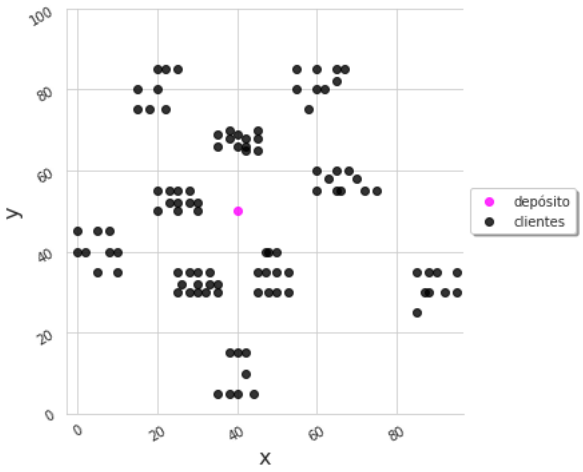

Figura 20: Instancia que representa clientes distribuidos en distintas ciudades.

Para la experimentación utilizaremos la instancia representada por la Figura 20 y, al final de la misma,

utilizaremos otra instancia similar para ver si el comportamiento es similar.

La instancia representada en la Figura 20 consiste en 10 "ciudades" con una cantidad considerable de clientes cada una. Por lo tanto, intuitivamente, creemos que obtendremos resultados más cercanos al óptimo si utilizamos una heurística de *Nearest Neighbor*, debido a que, al buscar siempre el cliente más cercano, el vehículo recorrerá primero una ciudad en su totalidad antes de desplazarse hacia otra, siempre y cuando la capacidad del vehículo lo permita. Nuestra hipótesis entonces será que tanto el algoritmo goloso. Nos interesará entonces estudiar el comportamiento de las heurísticas golosa y *cluster-first*, *route-second* (1) presentadas en las secciones 2.2 y 2.3 respectivamente y verificar que se comportan mejor que las demás heurísticas.

Por otro lado, la heurística de *savings* se comportará de manera ineficiente, debido a que el *saving* máximo será aquel que cumpla que la distancia del depósito a los clientes sea máxima, y la distancia entre los clientes de las rutas a unir sea mínima. Esto permitirá que se unan rutas donde haya clientes de más de una ciudad, pero que sean ciudades cercanas. Al permitir unir clientes de distintas ciudades es posible que más de un vehículo deba ir a una misma ciudad más de una vez, empeorando la distancia total obtenida.

La heurística de *cluster-first*, *route-second* (2) de la sección 2.4 no será tan eficiente como lo son la heurística de *Nearest Neighbor*, debido a que, si bien se *clusterizará* por áreas de clientes próximos entre sí, formando *clústers* similares a una ciudad entera si la capacidad lo permite, el algoritmo de ruteo de los *clústers* resultantes no utilizará ninguna técnica en particular; el mismo simplemente busca la primer solución factible que exista. Finalmente, esta heurística se comportará mejor que la de *savings*, ya que los vehículos no saldrán de su ciudad, con lo cual la distancia recorrida será menor.

En el caso de la metaheurística de *Simulated Annealing*, al depender de la heurística de *savings* para conseguir una solución inicial, esta va a estar muy alejada del óptimo por lo mencionado anteriormente. Luego, las soluciones generadas por la vecindad se van a intentar alejar de la incial, pero no van a lograr acercarse a las soluciones provistas por las otras heurísiticas.

4.1.1. Resultados

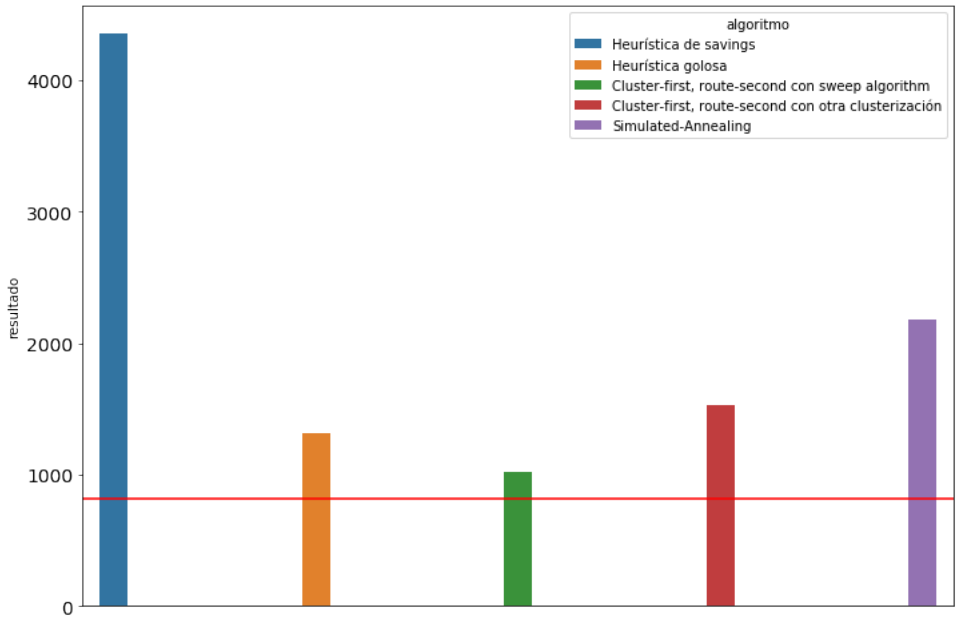

Figura 21: Resultados de la instancia representada en la Figura 20 para cada algoritmo. La línea roja representa el valor óptimo.

En la Figura 21 podemos ver que las hipótesis enunciadas parecen ser correctas para dicha instancia. Los resultados nos muestran que la heurística golosa y la heurística *cluster-first*, *route-second* (1) presentada en la sección 2.3 son las que mejor se comportan.

Por otro lado, la heurística de *cluster-first*, *route-second* (2) arrojó resultados cercanos a la heurística de *cluster-first*, *route-second* (1) tal como aventuramos en la hipótesis. También se puede ver cómo la me-

taheurística de Simulated Annealing logró mejorar significativamente la solución provista originalmente por *savings*, pero no logró estar a la par de las heurísticas mencionadas anteriormente.

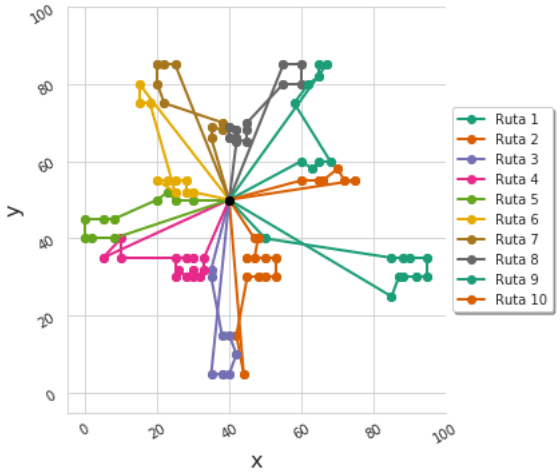

Figura 22: Resultados de la rutas generadas utilizando el algoritmo CVRP-CLUSTER-FIRST-ROUTE-SECOND-1 en la instancia de la Figura 20

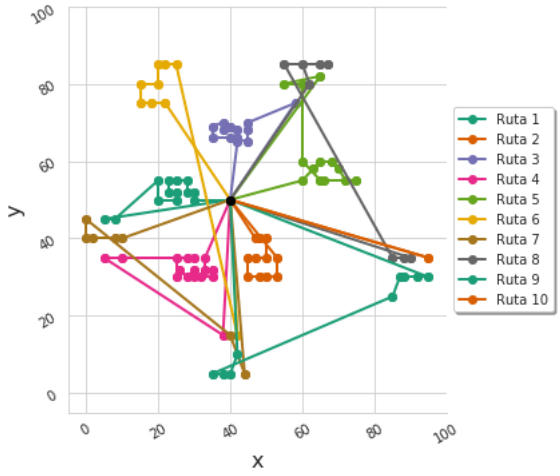

Figura 23: Resultados de la rutas generadas utilizando el algoritmo CVRP-GOLOSA en la instancia de la Figura 20

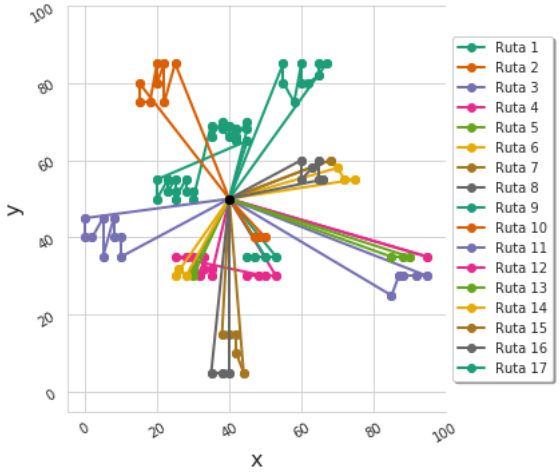

Figura 24: Resultados de la rutas generadas utilizando el algoritmo CVRP-CLUSTER-FIRST-ROUTE-SECOND-2 en la instancia de la Figura 20

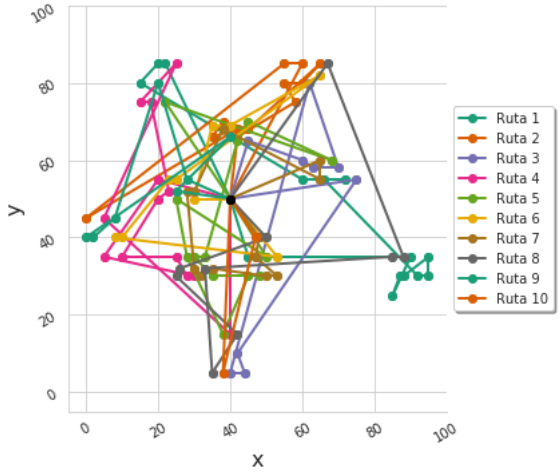

Figura 25: Resultados de la rutas generadas utilizando el algoritmo CVRP-SIMULATED-ANNEALING en la instancia de la Figura 20

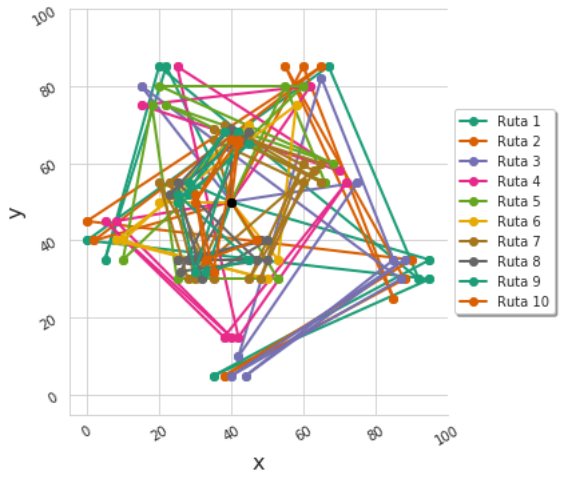

Figura 26: Resultados de la rutas generadas utilizando el algoritmo de CVRP-SAVINGS en la instancia de la Figura 20

En la figuras 23, 22 y 24 podemos ver que las rutas que se generaron priorizan los clientes de la misma ciudad tal como habíamos enunciado anteriormente. En cambio en las figuras 26 y 25 no hay una preferencia de clientes de las mismas ciudades, y es por esto que los valores están muy distantes del óptimo.

A continuación vamos a verificar que la misma hipótesis se cumple para una instancia similar, por ejemplo, para la instancia de la Figura 27, que cumple con las mismas condiciones que la instancia de la Figura 20. Realizaremos nuevamente el experimento en esta instancia para corroborar que realmente las hipótesis son correctas.

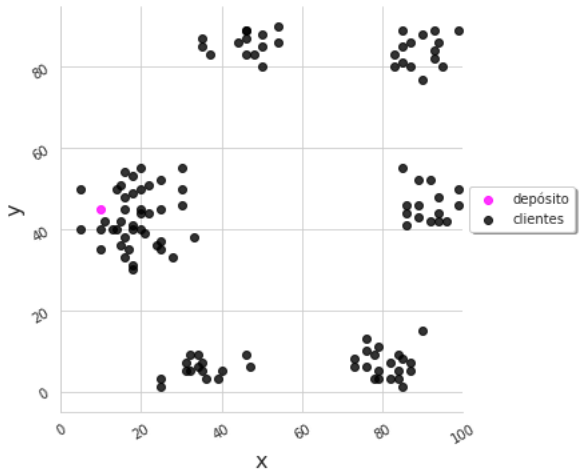

Figura 27: Instancia que representa clientes distribuidos en distintas ciudades.

Los resultados obtenidos son similares a los resultados anteriormente presentados, por lo tanto, podemos ratificar que las hipótesis enunciadas podrían ser correctas. Los resultados obtenidos de la instancia 27 se pueden ver en la Figura 28.

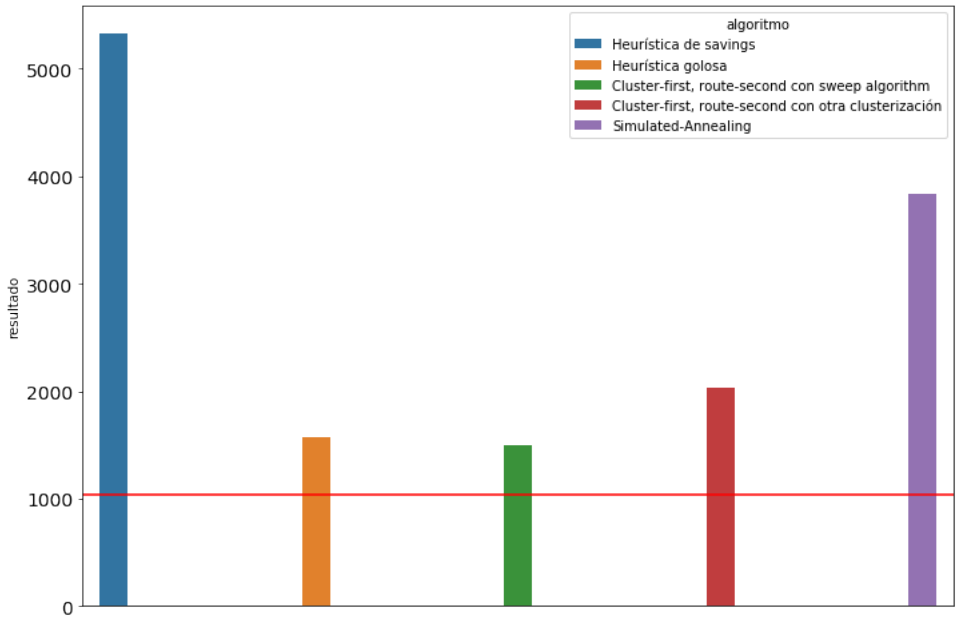

Figura 28: Resultados de la instancia representada en la Figura 27. La línea roja repesenta el valor óptimo.

4.2. Experimento 2: Instancias donde la heurística golosa no se comporta de manera adecuada

En el segundo experimento vamos a comparar las heurísticas en una familia de instancias donde creemos que la heurística golosa será la que peor se comportará. Las instancias que vamos a abordar son aquellas donde hay un depósito en el centro del plano y los clientes están ubicados simétricamente en el borde de un elipse. Los vehículos en dichas instancias tendrán capacidad C. Las demandas de los clientes estarán definidas de forma tal que los clientes ubicados en el primer y cuarto cuadrante tendrán demanda d (con $\frac{C}{2} < d < C$) y los clientes ubicados en el segundo y tercer cuadrante tendrán demanda $C - d$.

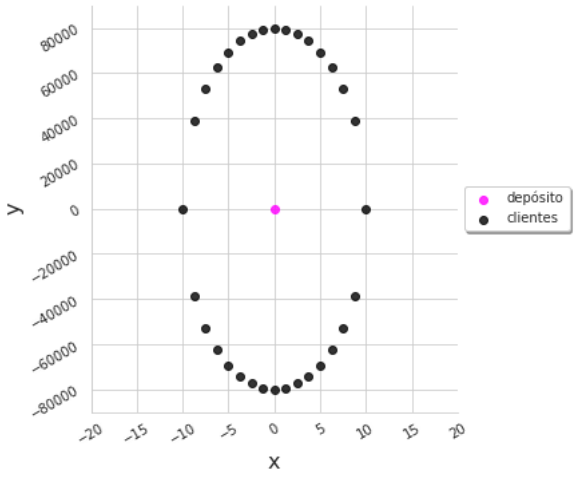

Figura 29: Instancia que pertenece a la familia de instancias analizadas en el experimento 2

Para la experimentación utilizaremos la instancia de la Figura 29 y, al final de la misma, utilizaremos otra instancia similar para ver si el comportamiento es similar.

La instancia representada en la Figura 29 contiene una elipse donde el eje mayor se corresponde con el eje vertical. En dicha elipse, los valores más cercanos al depósito son aquellos que se encuentran en el valor 0 del eje y, y los más lejanos son aquellos que se encuentran en el valor máximo y mínimo del eje y. Por otro lado, como todas los clientes de la parte superior del elipse tienen una demanda mayor a la

mitad de la capacidad del vehículo, necesariamente, para completar la capacidad del mismo, deberán ir a buscar un cliente que se encuentre en la mitad inferior del elipse si es que el algoritmo intenta llenar la capacidad del vehículo. Esto nos da una intuición de que esta instancia obligará a la heurística golosa a tener que ir desde la parte superior del elipse hasta la parte inferior del mismo en cada ruta, ya que la heurística busca el más cercano cuya capacidad no exceda el vehículo, y como todos los clientes de la parte superior del elipse excederán la capacidad del vehículo, necesariamente deberá ir a la parte inferior del elipse. Esto nos permite afirmar que dichas instancias funcionarán de manera ineficiente en la heurística golosa.

El motivo por el cual la heurística de *savings* funcionará mejor que la heurística golosa es que los ahorros que podrá generar utilizando la formula enunciada en la sección 2.1.1 serán aquellos tales que los clientes de sus rutas se encuentren cercanos. El problema es que los clientes cercanos de la parte superior del elipse no podrán unirse en *savings* porque exceden la capacidad del vehículo, entonces el algoritmo destinará un sólo vehículo para cada cliente de la parte superior. En cambio, para la parte inferior podrá unirlos ya que se puede satisfacer la demanda de varios clientes en un mismo vehículo. Esto permitirá que se ahorre tener que ir desde la parte superior a la parte inferior como sí lo debe hacer la heurística golosa, y además en la parte inferior del elipse, se ahorrará bastante distancia recorrida uniendo varios clientes en un sólo vehículo.

El motivo por el cual la heurística de *cluster-first*, *route-second* (1) presentada en la sección 2.3 funcionará mejor que la heurística golosa es que al *clusterizar* por ángulo se crearán *clústers* con varios clientes de la zona inferior del elipse, permitiendo que un sólo vehículo atienda a varios clientes de la misma zona. Sin embargo, en la parte superior del elipse se deberá crear una ruta por cada cliente, ya que la capacidad de los clientes de la parte superior del elipse tienen una demanda mayor a la mitad de la capacidad del vehículo. Aunque se deba generar una ruta por vehículo en la parte superior del elipse, se ahorrará tener que ir desde la parte superior a la parte inferior como sí lo debe hacer la heurística golosa, y además en la parte inferior del elipse, se ahorrará bastante distancia recorrida uniendo varios clientes en un sólo vehículo.

El motivo por el cual la heurística de *cluster-first*, *route-second* (2) presentada en la sección 2.4 funcionará mejor que la heurística golosa es el mismo que el de la heurística heurística de *cluster-first*, *route-second* (1) presentada en la sección 2.3.

En el caso de la metaheurística de Simulated Annealing, los resultados serán parecidos a los conseguidos con la heurística de *savings*. La solución inicial tendrá una ruta para cada cliente de la parte superior y algunas rutas que agrupan varios clientes en la parte inferior. La metaheurística explorará otras soluciones pero siempre se quedará con la mejor, por lo que intentará reducir el costo asociado con las rutas de la parte inferior.

4.2.1. Resultados

En la Figura 30 podemos ver que las hipótesis enunciadas son correctas para dicha instancia. Los resultados nos muestran que la heurística golosa es la que peor se comporta y que se encuentra muy lejos de las otras heurísticas. Esto nos permite afirmar la hipótesis de la ineficiencia de la heurística golosa para la instancia 29.

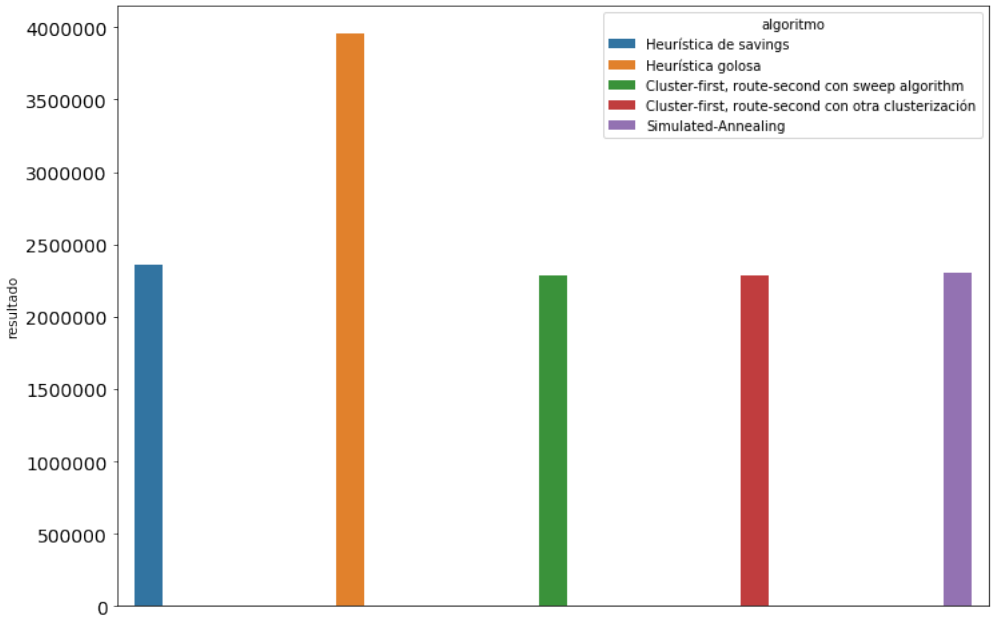

Figura 30: Resultados de la instancia representada en la Figura 29

Otra de las afirmaciones hechas es que la *clusterización* de los algoritmos *cluster-first*, *route-second* (1) y (2) presentados en las secciones 2.3 y 2.4 respectivamente permitirían agrupar varios clientes de la parte inferior del elipse en un mismo vehículo. Esto se puede ver claramente en las figuras 31 y 32

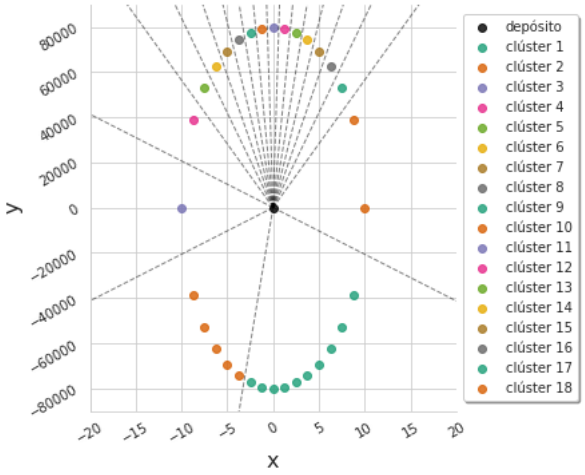

Figura 31: Clústers generados utilizando el algoritmo *cluster-first*, *route-second* (1) presentado en las sección 2.3 para la instancia de la Figura 29

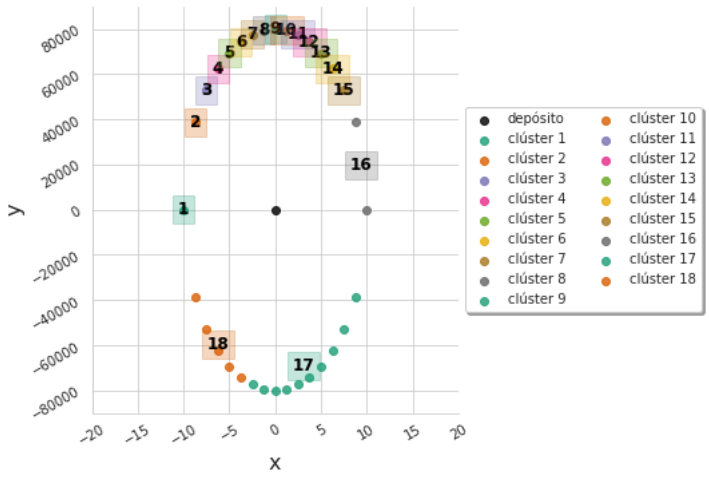

Figura 32: Clústers generados utilizando el algoritmo *cluster-first*, *route-second* (2) presentado en las sección 2.4 para la instancia de la Figura 29

Por último, tal como enunciamos anteriormente, las rutas de la heurística golosa estarán formadas en su mayoría por un cliente de la parte superior del elipse y un cliente de la parte inferior del elipse. Tal como dijimos, en la Figura 33 podemos ver que se ha cumplido lo esperado. Por otro lado, las otras heurísticas han aprovechado meter varios clientes de la parte inferior del elipse en un mismo vehículo. Las rutas se pueden ver en las figuras 34 , 35 , 36 y 37.

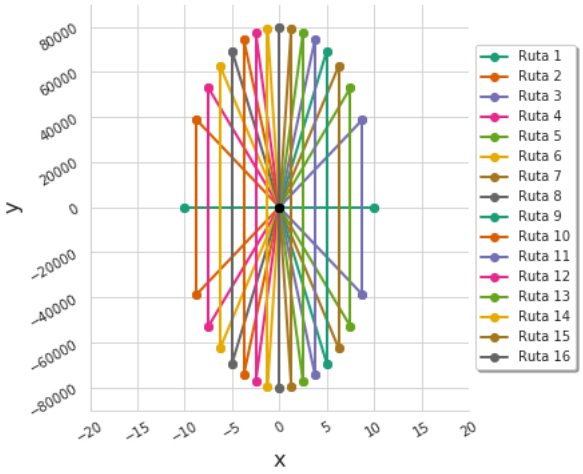

Figura 33: Resultados de la rutas generadas utilizando el algoritmo goloso en la instancia de la Figura 29

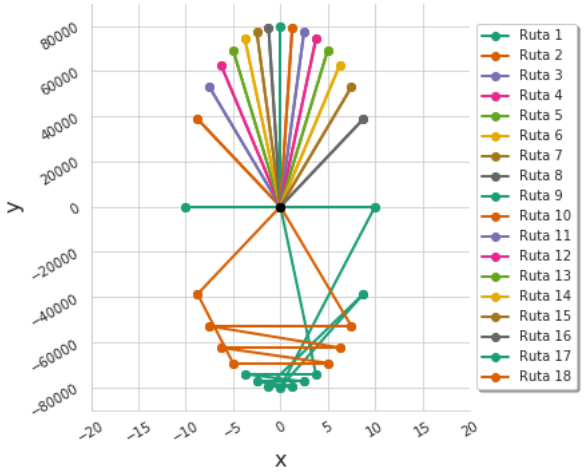

Figura 34: Resultados de la rutas generadas utilizando el algoritmo de *savings* en la instancia de la Figura 29

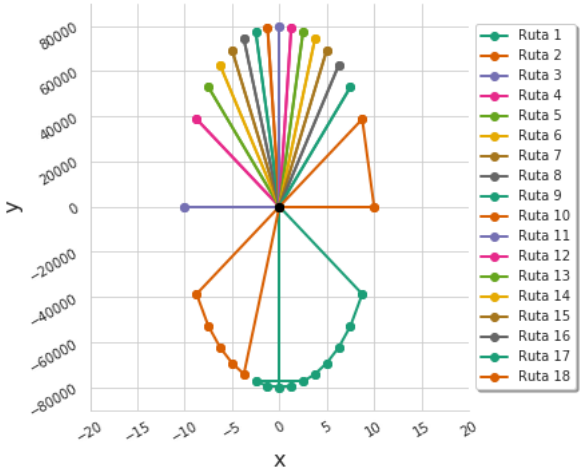

Figura 35: Resultados de la rutas generadas utilizando el algoritmo de *cluster-first*, *route-second* (1) presentado en las sección 2.3 en la instancia de la Figura 29

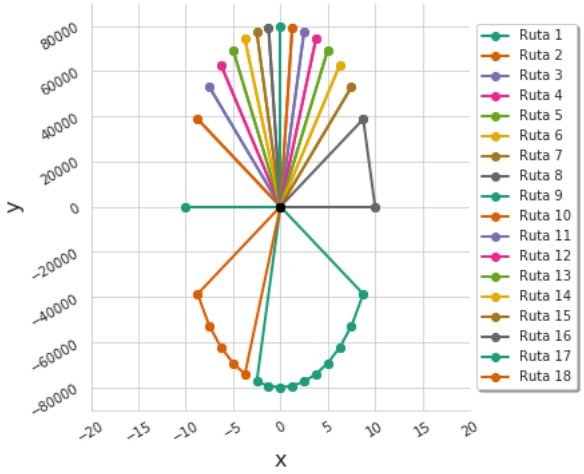

Figura 36: Resultados de la rutas generadas utilizando el algoritmo de *cluster-first*, *route-second* (2) presentado en las sección 2.4 en la instancia de la Figura 29

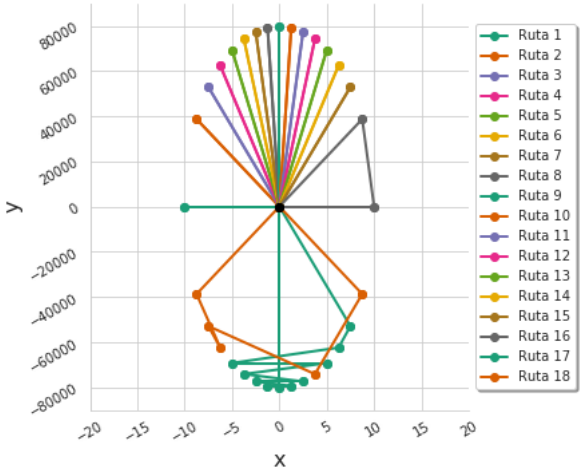

Figura 37: Resultados de la rutas generadas utilizando el algoritmo de Simulated Annealing en la instancia de la Figura 29

A continuación vamos a verificar que la misma hipótesis se cumple para una instancia similar, por ejemplo, para la instancia de la Figura 38, que cumple con las mismas condiciones que la instancia de la Figura 29. Realizaremos nuevamente el experimento en esta instancia para corroborar que realmente las hipótesis son correctas.

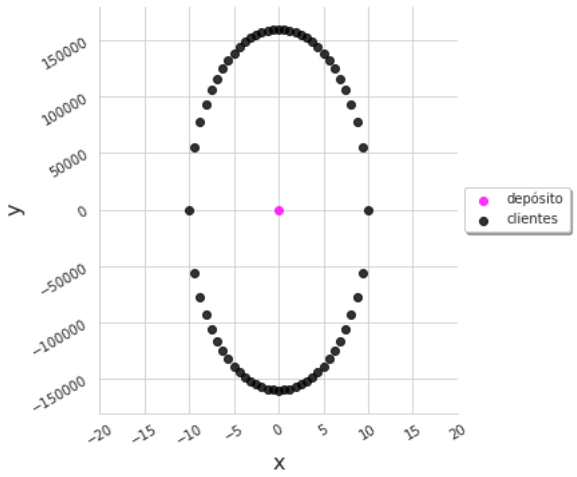

Figura 38: Instancia que pertenece a la familia de instancias analizadas en el experimento 2

Los resultados obtenidos son similares a los resultados anteriormente presentados, por lo tanto, podemos ratificar que las hipótesis enunciadas podrían ser correctas. Los resultados obtenidos de la instancia 38 se pueden ver en la Figura 39.

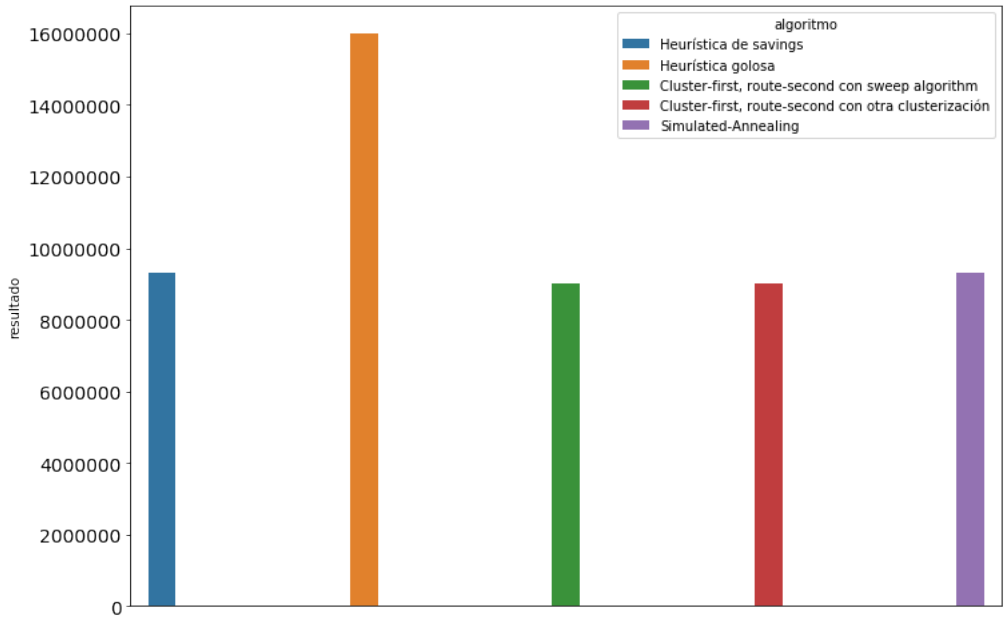

Figura 39: Resultados de la instancia representada en la Figura 38

4.3. Experimento 3: Utilizando distintas vecindades para la metaheurística Simulated Annealing

En este experimento vamos a ejecutar el algoritmo realizado para la metaheurística Simulated Annealing con 3 tipos de vecindades distintas. Las vecindades que utilizaremos serán 1INTERCHANGE, textproc1InterchangeRandom y MOVEHIGHESTAVERAGE. Luego de ejecutarlas analizaremos los resultados para ver qué vecindad produce una solución más cercana a la óptima para esta metaheurística.

El método de 1INTERCHANGE siempre se va a armar en un orden de $shifteos$ e $interchanges$ determinado, por lo que siempre se va a explorar la vecindad empezando por el mismo ”tipo”de soluciones. Podemos solucionar esto aplicando un random al orden en el cual son generadas, y a esto lo llamamos 1INTERCHANGERANDOM. Al hacer esto creemos que la exploración de soluciones va a ser más balanceada con respecto a qué operaciones realiza, y así podremos llegar a una solución más cercana al óptima.

También implementamos MOVEHIGHESTAVERAGE basados en [4]. Este método intenta construir una nueva solución utilizando información de la solución actual para no empeorarla, construyendo así en cada iteración una solución más cercana al óptima.

El método MOVEHIGHESTAVERAGE calcula para cada cliente la distancia promedio de su antecesor y su sucesor en la ruta que pertenece. La cuenta resultante es $d_i = \frac{d_{i-i}+d_{i-i}}{2}$. Una vez calculadas dichas distancias promedio para todos los vértices, se toma los 5 clientes cuya distancia d_i sean los mayores. Luego, se quita a dichos clientes de sus rutas y se pone a cada cliente en alguna ruta de manera aleatoria. Una vez seleccionada la ruta para poner al cliente, se lo insertará en la posición que minimice la distancia de la ruta.

Para analizar los resultados de los algoritmos, se tomaron instancias que contienen entre 32 y 80 clientes, contando al depósito. Dichas instancias fueron tomadas del **Set A** de instancias recomendadas por la cátedra ([5]), que ya hemos explicado anteriormente cómo están conformados.

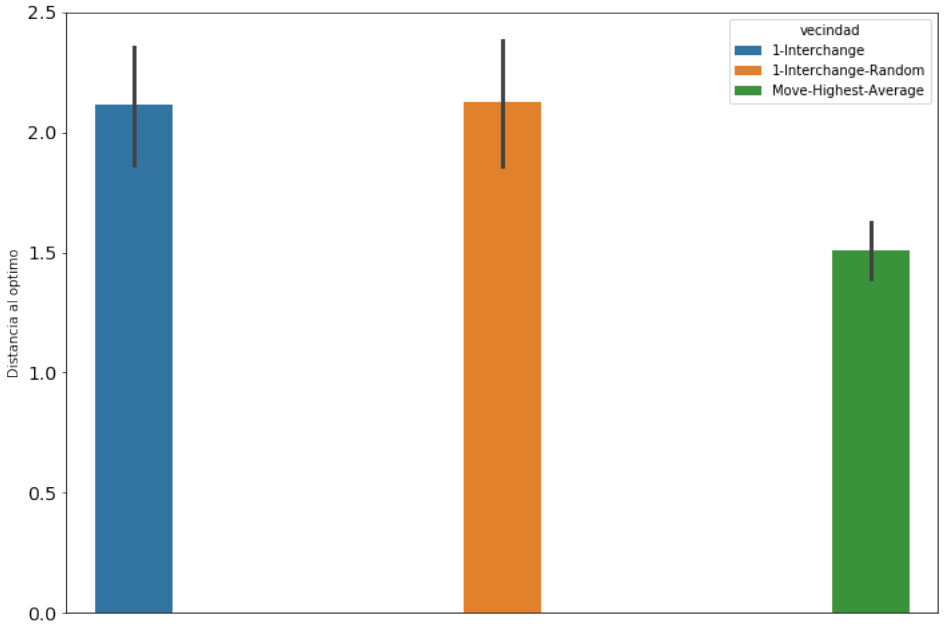

Figura 40: Resultados del experimento 3

Tomando como referencia el 1INTERCHANGE, no se notaron cambios sustanciales al reordenarlo aleatoriamente, por lo que no afecta el resultado final el orden en el que se explore dicho vecindario.

Por la parte de MOVEHIGHESTAVERAGE, sí se notó un mejora en el acercamiento a la solución óptima. Creemos que es porque, además de tener parte de exploración aleatoria muy necesaria en el Simulated Annealing, tiene otra en la que se aprovecha la solución ya dada e intenta utilizar la información anteriormente reunida. Esto hace finalmente que, en promedio, las soluciones se acerquen mas a la solución optima.

Referencias

[1] F. Alliani, L. Imperiale, L.D. Raposeiras, A.M. Ventura, Trabajo Práctico 2: Modelando problemas reales con grafos. *Algoritmos y Estructuras de Datos III*, oct 2018

[2] Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem, Ibrahim Hassan Osman, Institute of Mathematics and Statistics, The University, Canterbury, Kent CT2 7NF, UK

[3] METAHEURISTIC ALGORITHMS FOR THE VEHICLE ROUTING PROBLEM WITH TIME WINDOW AND SKILL SET CONSTRAINTS by Lu Han, Dalhousie University, Halifax, Nova Scotia, December 2016

[4] A Simulated Annealing Algorithm for The Capacitated Vehicle, Routing Problem, H. Harmanani, D. Azar, N. Helal W. Keirouz, Department of Computer Science,Lebanese American University, Byblos, 1401 2010, Lebanon

[5] http://vrp.atd-lab.inf.puc-rio.br/index.php/en/