DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3

Modelando problemas reales con grafos

14 de diciembre de 2018 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Jiménez, Gabriel	407/17	gabrielnezzg@gmail.com
Tasat, Dylan	29/17	dylantasat11@gmail.com
Buceta, Diego	001/17	diegobuceta35@gmail.com
Springhart, Gonzalo	318/17	glspringhart@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. El Trabajo

En este trabajo, vamos a analizar un tipo de algoritmos conocidos como “Heurísticas”.
Estos se caracterizan por un pequeño detalle, en muchos casos no dan la solución correcta al problema.
Y uno normalmente se preguntaría ¿Y para que nos sirven si no sabemos si la solución es correcta?.
Bueno, la respuesta es simple, es lo mejor que tenemos si queremos una complejidad razonable.
Resulta que los problemas que estámos intentando resolver son muy difíciles, esto quiere decir que no tenemos una solución de complejidad polinomial para resolverlos y por lo tanto tardaría mucho tiempo en darnos una respuesta en casos complejos (que en general son los que queremos resolver).
Entonces no nos queda otra opción que intentar llegar a una respuesta “razonablemente buena”, esto es lo que hacen las Heurísticas, son algoritmos que intentan darnos la mejor solución posible en el tiempo que tenemos para generarla.
Esto nos va a abrir un juego en donde tenemos que equilibrar dos variables, el tiempo que tarda y que tan buena es la solución.
Por otro lado tenemos las “Metaheuristicas” que se encargan de mejorar nuestra solución en cada iteración y así acercarnos tanto como querramos a la solución correcta.

2. Introducción al problema

El VRP problem o problema de ruteo de vehículos es un nombre que se le ha dado a toda una serie de problemas en las que se busca diseñar rutas para una flota de una clase de vehículos para satisfacer las demandas de un conjunto de clientes. La resolución de este problema es central para el reparto de bienes y servicios alrededor del mundo. Dentro de este problema, existen diferentes variantes que dependen de una serie de cosas tales como: los bienes a transportar, la calidad del servicio que se busca brindar, el tipo de clientes a satisfacer, el tipo de vehículos, la extensión física del servicio que se busca brindar, entre otras cosas.
Alguna de las complicaciones que pueden existir a la hora de diseñar estas rutas consisten en clientes sólo disponibles en determinados rangos horarios, planificación de rutas en múltiples días, incompatibilidad de un tipo de vehículo con algún cliente, diferentes cantidades de vehículos en cada depósito o punto de partida, etc.
Sin embargo, en todos los casos se busca resolver el problema, cualquiera sea su caso particular, al menor costo, donde el costo puede ser representado mediante alguna función objetivo que expresa el parámetro importante a minimizar al resolver el problema. En algunas situaciones esto podría ser el costo de k vehículos realizando rutas de entrega. En otras podría llegar a necesitarse minimizar la cantidad de vehículos que realizan las entregas y no el costo de las rutas en particular. Es decir, estamos ante un problemas que puede particularizarse para muchas situaciones en general.

Sin embargo, uno de los mas estudiados de esta familia de problemas es el de Ruteo de Vehículos con Capacidad (CVRP por sus siglas en inglés). En este caso, tenemos un conjunto de vehículos del mismo tipo, situados en un deposito y se necesita encontrar un conjunto de rutas para cada uno que permitan satisfacer la demanda de un conjunto de clientes con demandas. Cada vehículo solo puede realizar una ruta y tiene una capacidad de demandas,

de modo que sólo puede ir a un subconjunto de clientes cuya demanda sea inferior.

En este trabajo, recorreremos diferentes heurísticas para la resolución del CVRP problem.

3. Descripción formal del problema

Sea G un grafo no dirigido, G = (V,E), donde: V = $v_1, v_2, \cdots, v_n$, donde v_1 representa el depósito y todos los demás vértices representan clientes y E es el conjunto de aristas, las cuales tendrán asignado un costo no negativo, representando la distancia euclídea entre el par de vértices adyacentes.
Sea además H el conjunto de vehículos de un mismo tipo con una capacidad asociada.
Definimos una ruta como un circuito R que empieza y termina en v_1 y el costo de R como la suma de los costos de las aristas incidentes a los vértices de R, que además no supera a la capacidad asociada a los vehículos de H. Cada vehículo de H está asignado a sólo una ruta y cada ruta tiene asignado un vehículo diferente, de modo que el mapeo entre el conjunto de vehículos y rutas tiene el comportamiento de una función biyectiva. Cada Ruta r, excepto por el depósito, no tiene vértices repetidos, y la unión de todas las rutas contiene a todos los vértices de G(V).

Si recordamos el TSP problem o problema del viajante de comercio otro problema muy conocido, podremos encontrar una 'reducción' de uno de los problemas que surgieron de CVRP, que consiste en encontrar una determinada ruta.
El problema consiste en dada una lista de ciudades (interpretemos como vértices) y las distancias entre ellas (interpretemos como el valor de una arista, que representa la distancia entre sus vértices adyacentes) hay que encontrar la ruta más corta que sólo visita exactamente una vez cada ciudad y que finaliza en la misma ciudad que comenzó.
Este es un problema que pertenece a la categoría de NP-Hard y por lo tanto no tenemos una algoritmo que lo resuelva en tiempo polinomial. Sin embargo, podemos ver que si partimos de nuestro problema inicial y lo atacamos de determinada manera llegamos a un punto en donde existen una gran variedad de heurísticas con las cuales atacar este nuevo y bien conocido problema.

4. Técnicas y soluciones

Dado que es un problema ampliamente conocido por su rol dentro de los problemas que son de gran interés en la actualidad, recorreremos diferentes aproximaciones para atacar el problema. Cabe mencionar que como forma parte del conjunto de problemas NP-Hard, no existe un algoritmo exacto que permita resolver el problema en un tiempo computacional razonable, de modo que gran parte de lo que sigue son heurísticas y metaheurísticas que son lo único a lo que podemos apuntar en términos de solución para grandes instancias.

4.1. Branch and Bound

En esta aproximación se utiliza la técnica de divide and conquer para particionar el problema en subproblemas y optimizar cada uno de ellos por separado. La parte de 'Branch' se

corresponde con que se trata de conseguir subconjuntos del espacio total que se busca rutear. Cada uno de estos subconjuntos unidos forman el conjunto total y se juntan dentro de una lista de candidatos, próximos a analizar. Posteriormente se selecciona uno de ellos y se procesa. Los resultados pueden ser desde intercambiar la mejor solución por la actual, eliminarla porque no sirve o subdividir el subproblema actual y actualizar la lista de candidatos. En general el algoritmo termina después de haber vaciado toda la lista de candidatos del problema. Como se puede ver, es una solución del tipo fuerza bruta, en general combinada con ciertos requisitos para acercarse a una solución del tipo backtracking, pero en la teoría es una solución de tiempos computacionales no razonables.

4.2. Heurísticas

En este terreno comenzamos a encontrarnos con algoritmos aproximados, ya que las heurísticas son métodos o formas de atacar un problema buscando encontrar una 'buena' solución en un tiempo razonable pero que dado que suelen recorrer un espacio búsqueda 'limitado' de soluciones no necesariamente encontramos la solución exacta o mejor.

Dentro de este grupo podemos nombrar otros grupos

4.2.1. Constructivas

Consisten en ir construyendo una solución a partir de un criterio inicial. Alguno de las más conocidas son: Savings: Clark and Wright; Matching Based, etc.

4.2.2. Algoritmo de dos fases

Consiste en dividir el problema en otros dos: clusterizar el conjunto de clientes, de modo que cada cluster contenga un subconjunto de clientes que puedan ser recorridos por un vehículo de forma factible (la suma de demandas es menor o igual a la capacidad del vehículo) y posteriormente construir una ruta para cada cluster. También puede realziarse en orden inverso. Algunos ejemplos: Cluster first: route-second algorithm; Fisher and Jaikumar; The Sweep algorithm; Router-first:cluster-second; etc.

4.3. Metaheurísticas

El objetivo en este tipo consiste en un análisis o recorrido profundo por el espacio de soluciones, que por algún criterio, es el mejor para recorrer. Es razonable pensar que en este tipo se consiguen mejores resultados que en las heurísticas nombradas, básicamente porque podemos pensar como que dada una solución se trata de buscar otras mejores (optimizar) incursionando o buscando en zonas donde puede haber otras mejores.

Un criterio para recorrer otras soluciones puede ser ir a una que tenga un mejor valor objetivo, donde valor objetivo es una función que expresa lo interesante que puede ser una solución (depende del problema). En general suele denominarse a estos como búsqueda local. Y algunos ejemplos: Grasp, Simulated Annealing; Tabu Search; etc.

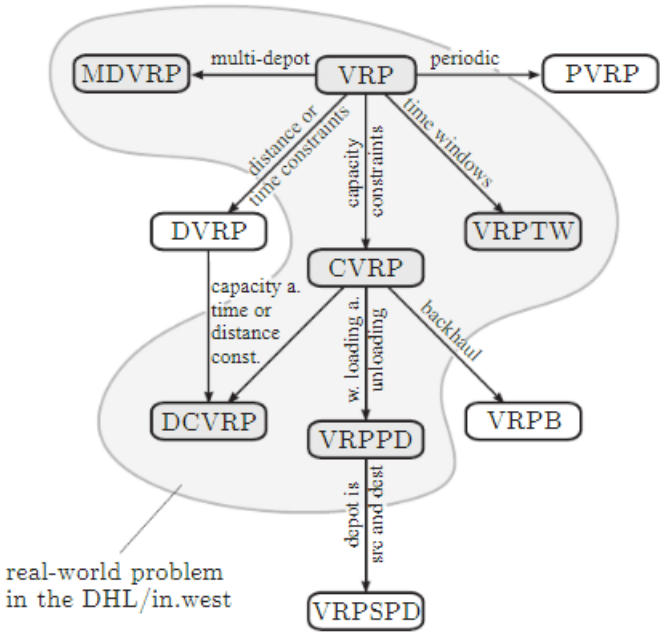
Por otro lado tenemos lo que son algoritmos genéticos y que a grandes rasgos buscan someter a las soluciones a detemrinados procesos 'evolutimos', similares, y por eso el nombre, a los procesos de mutación de la evolución biológica, combinaciones genéticas, procesos de selección (según algún criterio), y en función de los que 'sobreviven' son considerados los más

aptos mientras que las demás son descartadas. Algunos ejemplos son: Ant Algorithm; Genetic Algorithm.

5. Problemas de la vida real

Hemos mencionado la importancia del CVRP problem debido a la cantidad de problemas de la vida real que pueden modelarse utilizándolo.

Naturalmente en la vida real los problemas tienden a ser más difíciles de resolver debido a la cantidad de diferentes variables que intervienen en él. El VRP problem tiene muchos derivados y es que en la vida real las empresas suelen necesitar resolver varios tipos de este problema simultaneamente.
Imaginemos una empresa de logística y distribución como DHL, UPS, etc, es claro que tienen más de un depósito o lugar desde el que distribuyen; también utilizan diferentes tipos de vehículos (trenes, camiones, aviones, barcos); tienen clientes de diferentes categorías (desde una persona que envía un paquete a otra a una empresa que hace un envío de mercadería hacia otro país).
Es por esto que imaginamos que un diagrama de su logística de distribución podría verse como esto:



Es decir, de alguna manera tienen que subdividir con algún criterio su problema y encontrar la manera de utilizar las instancias del VRP que más se adapte a cada 'zona'.

Dentro del campos de la robótica, aunque para la construcción de circuitos en general, si se necesita unir ciertas componentes de la placa y se quiere reducir la cantidad de buses que van a unirlos, o la cantidad componentes que unir, se podría modelar el problema con alguna modificación del VRP.

Si pensamos en las agencias de turismo vemos que cuando un cliente solicita un paquete para recorrer determinas ciudades, participar de determinados eventos, etc, estamos frente a problemas fácilmente modelables con el VRP. Por un lado, recorrer N ciudades siendo un turísta podría ser casi el mismo problema que TSP, dado que el turísta probablemente quiera aprovechar al máximo los dias y no volver a ciudades que ya conocio para ir a otra. Además, podríamos tener variantes donde por determinadas razones hay más de un transporte a utilizar, o al contrario sólo uno, etc.

También podría pensarse en tareas a resolver y la necesidad de encontrar una forma de resolverlas de forma secuencial reduciendo el costo. Si una parte del proceso de producción de una empresa requiere del funcionamiento de una máquina de alto costo por hora y varias tareas necesitan de ella. O que determinadas tareas requieren de mayor personal. Es decir, es claramente un problema modelable como un TSP (una instancia particular del VRP, donde sólo hay un vehículo).

Las empresas que transportan personas, por ejemplo micros escolares o micros que llevan gente al trabajo es otra de las situaciones de la vida real que puede modelarse con VRP. El micro sale de una estación y tiene que ir hacia diferentes puntos recogiendo personas. En este caso la estación inicial podría no coincidir con la de llegada o mismo el costo de los viajes podría ser diferente a la mañana que en otra parte del día. Claramente no es un problema sencillo pero es modelable con VRP.

Otra situación podría ser los censos. Podemos modelar a las manzanas como vértices y las personas como vehículos. Habría varios depósitos ya que cada barrio o comuna tendría como un lugar de salida, pero seguro que al final del día las personas que realizaron el censo deben volver hacia el depósito desde el que salieron a entregar los datos relevados.

6. Heurística de Savings de Clarke-Wright

6.1. Introducción

Para nuestra primer aproximación del problema de Capacitated Vehicle Routing Problem introduciremos una heurística constructiva que utiliza fuertemente el concepto de "Savings" ideado por Clark-Wright y luego publicado en "Scheduling of vehicles from a central depot to a number of delivery points" (1964). Entonces, si consideramos un Grafo G con representación en el plano en el cual cada nodo (en este caso cliente) tiene dos coordenadas, y en el cual el nodo 1 es el depósito central, definimos d_{ij} como la distancia euclidea del nodo i al nodo j, en particular en este problema tendremos que $d_{ij} = d_{ji}\ \forall i, j$ ya que no hay obstáculos entre ningún nodo en ninguna dirección.

Diremos que el saving entre i y j notado como s_{ij} será la diferencia que hay entre la suma del peso las rutas en las cuales solo voy a i o a j, y la ruta en la que paso por i y por j. Es decir si lo pensamos como una fórmula sería de la forma:

$$s_{ij} = peso(\{1, i, 1\}) + peso(\{1, j, 1\}) - peso(\{1, i, j, 1\})$$

Lo cual también se puede traducir a:

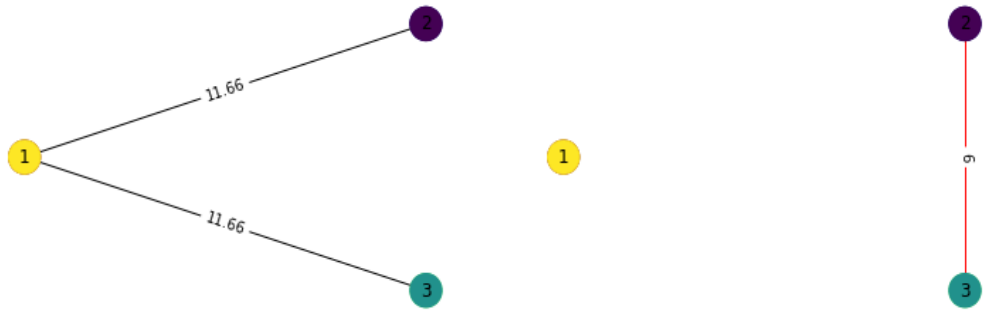
$$s_{ij} = 2 * d_{1i} + 2 * d_{1j} - (d_{1i} + d_{ij} + d_{1j})$$

Que se puede reducir a:

$$s_{ij} = d_{1i} + d_{1j} - d_{ij}$$

Es decir, el saving entre i y j sería lo que ahorraríamos considerando la ruta en la que vamos de i a j respecto de las rutas en las que solo voy a i y a j.

Para poder entender mejor el concepto de saving consideraremos el siguiente ejemplo:



Observación: para este ejemplo tomaremos las coordenadas 1 = (0,0), 2 = (5,3) y 3 = (5,-3)

En particular podemos ver que de tomar los caminos C1 = {1,2,1} y C2 = {1,3,1} tendremos un peso de 11.66 por camino y un peso total de 23.32, y que la distancia entre 2 y 3 será 6

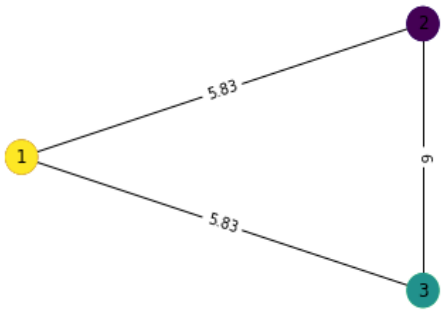
Entonces si calculamos el saving s_{23}:

$$s_{23} = d_{12} + d_{13} - d_{23}$$

$$s_{23} = 5{,}83 + 5{,}83 - 6$$

$$s_{23} = 5{,}66$$

Se puede observar que al unir caminos pasando por 2 y 3 en un único camino se ahorrará 5.66 del peso total con respecto de los caminos individuales. Entonces el camino resultado de tener en cuenta el saving s_{23} será el siguiente



Y su peso total será de 17.66.

Una mención importante sobre los savings será que estos **siempre serán positivos.** Esto ocurrirá por una razón: como vimos antes, calcular el saving es hacer $d_{1i} + d_{1j} - d_{ij}$, por lo tanto si pensamos en el triangulo que tiene a d_{1i}, d_{1j}, d_{ij} como lados, entonces 1,i,j serán sus vértices y como sabemos cumplen estar en el plano euclídeo. Esto será importante porque podremos decir que los lados de este triángulo cumplen la desigualdad triangular, es decir en particular se verifica que:

$$d_{1i} + d_{1j} \geq d_{ij}$$

De lo cual podemos deducir:

$$d_{1i} + d_{1j} - d_{ij} \geq 0$$

Y podemos ver que lo que esta a la izquierda es nuestra fórmula para calcular el saving s_{ij}, por lo tanto podremos afirmar que es positivo.

Ya introducido el concepto de saving se pasará a explicar el algoritmo/heurística y su metodología.

6.2. El Algoritmo

La idea general del algoritmo será la siguiente: Primero se calcularán los savings entre todos los nodos del sistema y se los guardará en un vector. Ese vector será ordenado de manera decreciente de manera de tener los 'mayores savings' al principio del vector.

Luego se generarán n rutas de la forma $C_i = \{1,i,1\}$ $\forall i = 2...n$, es decir rutas en las que solo se visita 1 cliente.

A partir de aquí se empezarán a combinar las rutas respetando el siguiente procedimiento:

Desde el principio del vector de savings hasta el final se tomará el saving actual s_{ij} y se determinará si existen dos rutas que pueden ser unidas:

- Una ruta que empiece con $\{1,j\}$
- Una ruta que termine con $\{i,1\}$

De existir tales rutas y que la suma de las capacidades hasta ahora de ambas no sea mayor que la capacidad de un camión, entonces se las combinará en una sola ruta que será de la forma $\{1,j,...,i,1\}$

Dicho procedimiento se corresponde con el siguiente pseudocódigo:

```
HeuristicaCW(listaNodos Gn){
    CapacidadMax <- Gn.getCapacity()
    savings <- vector()

    for(int i = 2...n){
        for(int j = i+1...n){
            Sij <- calcularSaving(getNodo(Gn,i),getNodo(Gn,j))
            savings.pushBack(Sij)
        }
    }

    ordenarDecrecientemente(savings)
    Rutas <- vector()

    for(int i = 2...n){
        rutaIndividual <- agregarCliente(getNodo(Gn,i))
        Rutas.pushback(rutaIndividual)
    }

    for(sij : savings){
        k <-0
        while(no encuentre ruta que termine en el nodo i en Rutas){
            k++
        }
        l <-0
        while(no encuentre ruta que empieze en el nodo j en Rutas){
            l++
        }
        if(esValido(i) && esValido(j) &&
        peso(Rutas[i])+peso(Rutas[j])<= CapacidadMax){
            unirRutas(Rutas,Rutas[i],Rutas[j])
        }
    }
    return Rutas
}
```

6.3. Análisis de Complejidad

Para realizar el cálculo de complejidad se detallarán todos los pasos realizados y el costo asociado a cada uno de ellos.

- Costo de calcular y guardar los savings: en particular tanto el costo de calcular el saving entre dos nodos y almacenarlos en un vector es $\mathcal{O}(1)$, y como realizamos esto en un doble

ciclo en n la complejidad de este paso será $\mathcal{O}(n^2)$.

- Costo de armar rutas singulares: esto será $\mathcal{O}(1)$ por cada ruta costando un total de $\mathcal{O}(n)$.
- Costo de combinar las rutas: El ciclo principal se realizará $\mathcal{O}(n^2)$ veces ya que esa es la cota de la cantidad de savings. Luego para las operaciones dentro de este ciclo tendremos que:
  - tanto encontrar la ruta que termine en i como la que empieze en j es $\mathcal{O}(n)$
  - combinar rutas si es posible es $\mathcal{O}(n)$, ya que a lo sumo se termina combinando en la ruta en la que están todos los clientes.

  Por lo tanto el costo de este procedimiento será de $\mathcal{O}(n^3)$

Entonces la complejidad total del algoritmo será: $\mathcal{O}(n^2) + \mathcal{O}(n) + \mathcal{O}(n^3) = \mathcal{O}(n^3)$

6.4. Peores Instancias

Para este algoritmo y sus variantes de TSP, aún no se conocen peores instancias [2], por lo cual, someteremos la pregunta **¿Qué tan malo puede ser nuestro algoritmo respecto de la solucón óptima?** a la experimentación.

7. Heurística Constructiva

7.1. Introducción

Para nuestra segunda aproximación del problema utilizaremos dos heurísticas constructivas golosas distintas, con el fin de tener un abanico de posibles soluciones más amplio.

A tales fines se presentan las siguientes Heurísticas:

- Heurística Más Cercano al Depósito
- Heurística Vecino Más Cercano

Como se puede interpretar de los nombres que les pusimos, la primera consta de recorrer los nodos en un orden decreciente respecto de su distancia al deposito.
Esta heuristica simplemente arma una lista con los vecinos y los ordena por su distancia al deposito, utilizaremos una priority queue para optimizar la ejecucion del algoritmo, de esta manera podemos ir sacando elementos de la cola de prioridad y agregarlos en ese orden hasta que se acabe la capacidad del camion y si se acaba la capacidad empezamos a agregar al proximo camion.
Se puede esperar que esta sea la peor de las dos Heurísticas, ya que en general los primeros caminos pueden ser buenos, con distancias muy cortas ya que son los puntos mas cercanos al deposito, pero los ultimos caminos podrian recorrer de una punta del mapa a la otra.

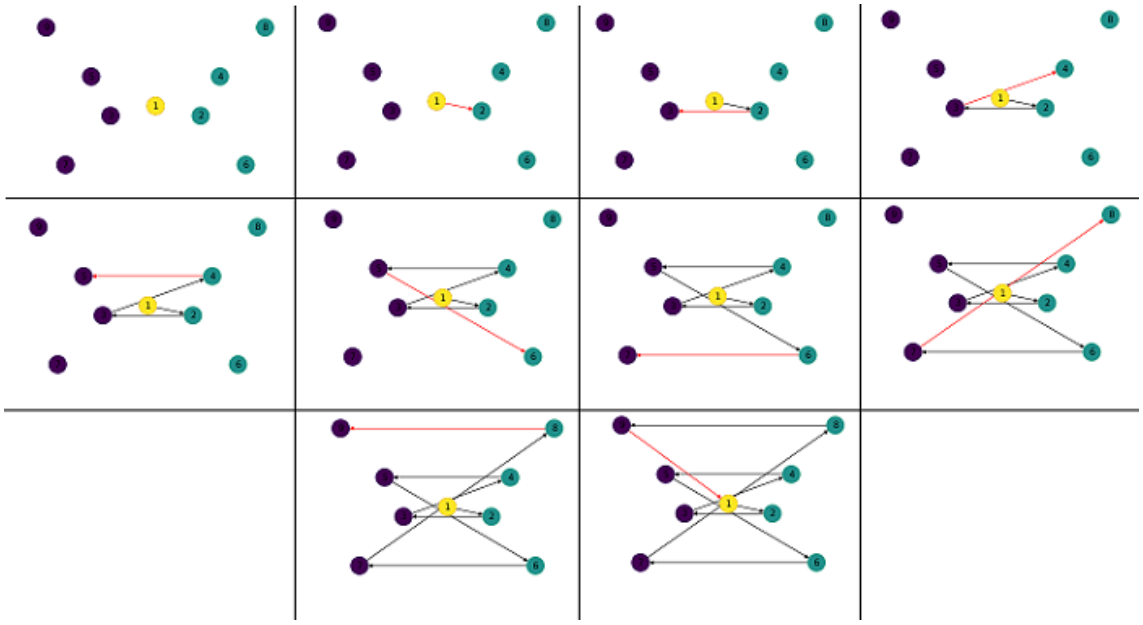


Figura 1: Ejemplo resuelto por H. Más Cercano al Depósito en el cual solo se necesita un camión.

Esto nos deja con una heuristica bastante mala por parte del resultado, pero es tan simple que nos deja una complejidad muy buena de la cual vamos a hablar en la seccion de complejidad.
Por otro lado tenemos la “Heuristica del Vecino Más cercano”, esta heuristica es un poco

más elaborada pero nos va a permitir obtener soluciones mejores a la anterior (en general) y manteniendo una cota de complejidad polinomial aunque peor que en la otra.
Esta heuristica consta de agregar siempre el nodo más cercano al ultimo nodo que visitamos con el camion actual. De esta manera vamos a recorrer siempre distancias cortas excepto en la ultima arista (cuando se lleno el camion) ya que podría suceder que cuando se acabe la capacidad nos encontremos en un nodo muy lejano al deposito, y como no tenemos más capacidad, no nos queda otra opcion que volver al deposito por la arista que lleva de ese nodo al deposito que puede ser muy costoso y arruinar el camino.
Sin embargo por ejemplo si recorrés 50 nodos con distancias muy cercanas y despues te toca volver al deposito por un camino largo, esto no debería dejar tan mala la suma total que es lo que nos importa.
Para este algoritmo agregamos un poco de la lógica del algoritmo anterior, tambien utilizaremos una priority queue ordenanda por los que están mas cerca del deposito, esta queue nos va a dar el primer nodo del camino, y luego vamos a recorrer todos los nodos para obtener el más cercano a este y así consecutivamente hasta que el más cercano al nodo actual supere la capacidad, esto va a cerrar el camino y volver a comenzar tomando otro nodo de la cola de prioridad (uno que no haya sido visitado).

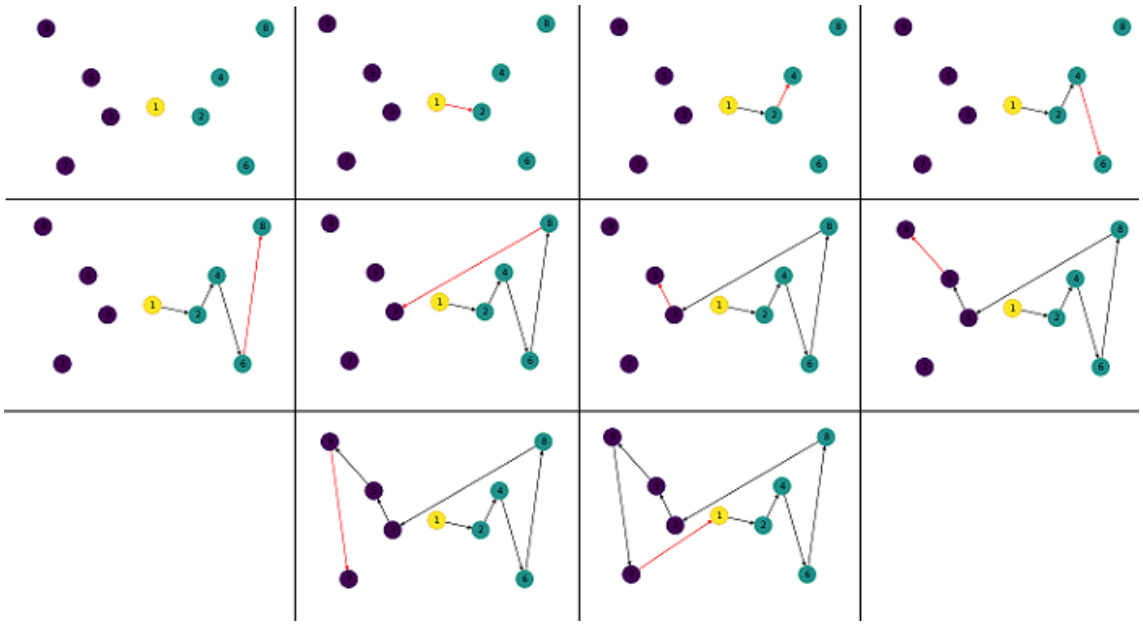


Figura 2: Mismo ejemplo que el anterior resuelto por H. Vecino Más Cercano.

Esto nos va a dejar una complejidad un poco más cara que el algoritmo anterior pero se puede ser optimista de que las soluciones nos van a dar mucho mejor en la mayoría de los casos. Así tenemos dos heuristicas constructivas para probar, que seguramente van a dar distintos resultados y en complejidades muy buenas, en la vida real podríamos correr varias heuristicas en paralelo y quedarnos con el mejor resultado.

7.2. El Algoritmo

Ya contámos un poco de como es la idea del algoritmo, ahora vamos a desarrollar un pseudocodigo para poder tener una idea mas clara y poder analizar la complejidad de los algoritmos.
Vamos a comenzar con la Heuristica Mas Cercano al Deposito:

```
HeuristicaMasCercanoAlDeposito(listaNodos Gn) {
    Cola <- heapify(Gn.getNodos())

    Solucion <- []
    CamionActual <- []

    while (!vacia?(Cola)) {
        NodoActual <- SacarUltimo(Cola)

        if (SumaDeDemandas(CamionActual) + Demanda(NodoActual) > Gn.getCapacity()) {
            Solucion.agregar(CamionActual)
            CamionActual <- []
        }

        CamionActual.agregar(NodoActual)
        MarcarComoVisitado(NodoActual)
    }
}
```

Se puede ver como esta heuristica es super simple y no es dificil implementarla, esto, combinado con su notablemente buena complejidad (de la que hablaremos en la seccion de complejidad), nos dan razones para tenerla en cuenta en algunos casos donde necesitemos una solucion muy rapido y de poco costo, a pesar de que es evidente que los resultados no van a ser muy certeros en la mayoría de los casos.
Ahora vamos a continuar con el pseudocódigo de la Heuristica del Vecino Más Cercano:

```
HeuristicaDelVecinoMasCercano(listaNodos Gn) {

    Solucion <- []
    CamionActual <- [Deposito]

    while (Hay nodos sin visitar) {
        MenorDistancia <- infinito

        for (nodo in Gn.getNodos()) {
            if (nodo == Ultimo(CamionActual) continue;
            if (Visitado?(nodo)) continue;
            if () continue;

            dist <- CalcularDistancia(nodo, Ultimo(CamionActual))
            if (dist < menorDistancia) {
```

```
                MenorDistancia <- dist
                NodoActual <- nodo
            }
        }

        if (MenorDistancia es infinito ||
        SumaDeDemandas(CamionActual) + Demanda(NodoActual) > Gn.getCapacity()) {
            CamionActual.agregar(NodoActual)
            MarcarComoVisitado(NodoActual)
        } else {
            Solucion.agregar(CamionActual)
            CamionActual <- [Deposito]
        }
    }
}
```

Aqui se puede ver que este algotitmo es a penas un poco más complejo que el anterior, pero contiene un aumento en complejidad, esto puede sonar mal pero se podrá observar que las soluciones encontradas por este algoritmo tienen mucha más precisión.

7.3. Análisis de Complejidad

Vamos a mirar los pseudocódigos para poder descubrir la complejidad teórica de cada uno de los algoritmos.
En la H. Mas Cercano al Depósito se puede observar que al comienzo usamos la funcion $HEAPIFY$ que tiene una complejidad $O(N)$ y luego se utiliza un loop que va sacando elementos del heap hasta que se quede vacío, esto por propiedades del Heap cuesta $O(N * logN)$ todo lo demás es $O(1)$ por lo que tendríamos una complejidad total de $O(N * logN)$ teorica en peor caso para este algoritmo.

En la H. del Vecino Más Cercano estámos recorriendo todos los nodos, pero en este caso, para cada uno de ellos tenemos que recorrer todos de nuevo para encontrar el más cercano, como todas las operaciónes dentro de estos loops son $O(1)$ se puede deducir que la cota de complejidad teórica en peor caso es de $O(N^2)$.

De esta manera, tenemos dos heuristicas constructivas que devuelven soluciones viables para el problema pero en general no van a devolver la solución optima, pero a causa de esto pudimos ganar mucho en la complejidad ya que la solución optima se obtiene con complejidad logaritmica, y estas heuristicas tienen complejidades polinomiales y encima con polinomios razonablemente chicos.
En general cuando se trata de heuristicas, uno tiende a ganar complejidad perdiendo precisión, esto lo vamos a ver luego en la experimentación pero es de esperarse que la H. del Vecino Más Cercano se acerque más a la solución real que la H. del Más Cercano al Depósito por como están constuidas y por eso tiene una complejidad más alta.

8. Algoritmo de cluster-first, route second: sweep algorithm

8.1. Introducción

Como tercera aproximación al problema utilizaremos un algoritmo que divide el CVRP en dos partes, la **clusterización** y el **ruteo**. La idea del algortimo es separar a los clientes en distintos clusters donde luego se aplicara un algoritmo de TSP a cada uno para poder armar las rutas que recorrerán los camiones. Para la clusterización, los clusters se forman a partir de un "barrido" sobre los clientes según el ángulo que tengan respecto del depósito, de menor a mayor se van colocando en clusters, si los ángulos son iguales se verifica cuál es menor por la distancia. Si la demanda total de los clientes de un cluster excede la capacidad de los vehículos cuando se agrega un cliente nuevo, el cluster se "cierra" y se coloca a ese cliente en otro cluster vacío, que posteriormente se llenará de la misma forma. Así podemos ubicar cada cliente en un cluster distinto, existiendo la posibilidad de que haya sólo un cliente en un cluster.
En la siguiente imagen se puede ver un ejemplo de esta clusterización:

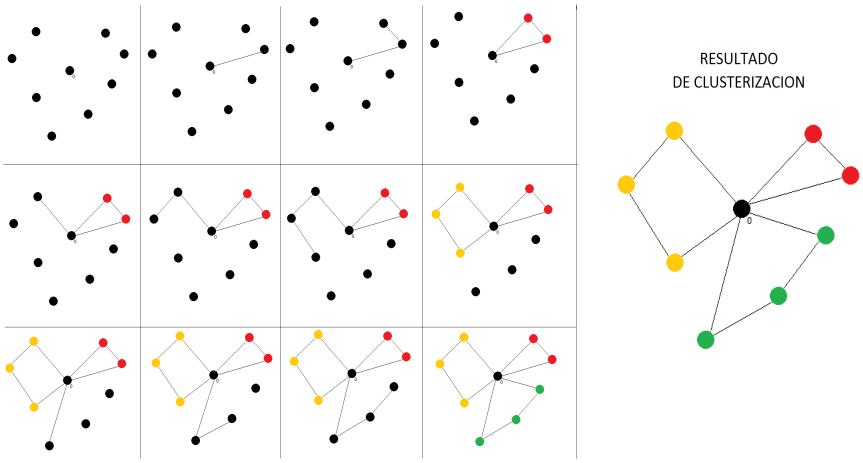


Figura 3: En la parte izquierda de la imagen, tenemos de izquierda a derecha y de arriba hacia abajo un ejemplo de como se podria generar una clusterización con este algoritmo, el punto con un 0 al lado indica que es el depósito, a la derecha de la imagen se encuentra el resultado de la clusterización agrandado para que sea más fácil verlo

Una vez conseguidos los clusters, se aplica un algoritmo TSP (que se explica posteriormente) a cada uno para poder generar los ruteos de los caminos. // Este algoritmo se puede hacer facilmente si se cambian las coordenadas de los clientes a coordenadas polares y se colocan en una cola de prioridad que los ordene según su ángulo y distancia, así esta parte del algoritmo puede ser simplificada a ir sacando elementos del principio de la cola y acomodarlos en clusters según lo dicho anteriormente. La cantidad de camiones necesarios va a estar dada por la cantidad de clusters que se formen, ya que por como se forman la suma de las demandas de los clientes de cada cluster no supera la capacidad de los camiones.

8.2. El Algoritmo

Algunas aclaraciones antes de presentar el algoritmo:

- A la hora de implementar el algortimo consideramos como parte del mismo la lectura de coordenadas de los nodos y la conversión de las mismas a coordenadas polares.
- ListaCorPol es una clase que contiene un Nodo depósito y una Cola de prioridad que ordena nodos según sus coordenadas polares, de los que tengan menor angulo a los que tengan mayor ángulo respecto del punto marcado como depósito. También asumimos que los parametros se pasaron por referencia para que la complejidad no sea afectada.
- obtenerDatos es una función que toma un vector de nodos y una capacidad, lee un archivo externo con la información de los nodos, carga el vector con los nodos, escribe el valor de la capacidad de los camiones en capacidad y setea el depósito en la lista de coordenadas Polares.

El algortimo presentado es el siguiente:

```
vector<vector<Nodo>> sweep(){ //O(n + 2*n*log(n) + n^2) = O(n^2)
    ListaCordPol lcp;
    Vector<Nodos> vn;
    float capacidad;

    cargarDatos(vn, lcp, capacidad); //O(n + n*log(n))

    vector<vector<Nodo>> solucion;
    vector<vector<Nodo>> clusters = generarClusters(lcp, vn, capacidad); //O(n*log(n))

    for(c : clusters){
        solucion.push_back(resolverTSP(c)); //O(n^2)
    }

    return solucion;
}

cargarDatos(vector<Nodo> vn, ListaCordPol lcp, float capacidad){
    obtenerDatos(vn, capacidad, lcp); //O(n)

    for(Nodo n : vn){//O(n*log(n))
        if(n != lcp.getNodoBase()){
            lcp.agregarNodo(n) //O(log(n))
        }
    }
}

vector<vector<Nodo>> generarClusters(ListaCordPol lcp, vector<Nodo> vn, float capacidad){/
    vector<vector<Nodo>> vecClusters;
    float capActual = 0;
    vector<Nodo> vecActual;
    vecActual.push_back(lcp.getNodoBase());
```

```
    while(!lcp.vacia()){ //O(n*log(n))
        if(capActual + lcp.siguiente().demanda <= capacidad){
            capActual += lcp.siguiente().demanda; //O(log(n))
            vecActual.push_back(vn[lcp.siguiente().id - 1]); //O(1)
            lcp.pop();//O(log(n))
        }
        else{
            capActual = 0; //O(1)
            vecClusters.push_back(vecActual); //O(1)

            vecActual.clear(); //
            vecActual.push_back(lcp.getNodoBase()); //O(1)
        }
    }

    vecClusters.push_back(vecActual);

    return vecClusters;
}
```

8.3. Análisis de Complejidad

Siendo n el número de clientes, cargarDatos en $O(n)$ llena el vector vn con la información de los nodos (su posición, id y demanda) y obtiene la capacidad de los camiones, y en $O(n * \log(n))$ inserta los nodos en la lista de coordenadas polares, al insertarlos se convierten en $O(1)$ los nodos a nodos de coordenadas polares. La generación de los clusters se hace en generarClusters, ya que consiste en ir tomando de a uno los elementos de la lista, la complejidad de esa función es $O(n * \log(n))$ ya que las operaciones que manejan que cluster se llena son $O(1)$. Finalmente con los clusters ya formados, se hace TSP sobre los nodos, que tiene una complejidad de $O(n^2)$

Entonces la complejidad total $O(n + 2 * n * \log(n) + n^2) = O(n^2)$.

9. Algoritmo de cluster-first, route second: mediante alternativa de sweep algorithm

9.1. Introducción

En este tipo de técnicas la forma de atacar el problema es primero clusterizar (con algún criterio) los puntos en el espacio y posteriormente aplicar un algoritmo de resolución de TSP. Si clusterizamos en función de los nodos más cercanos entre sí y que además la suma de las demandas no supere la capacidad del camión, entonces ese es un cluster válido. Si logramos encapsular de esta manera todos los vértices y además tratando de alcanzar el mínimo número de clusters entonces estaremos encontrando soluciones de buena calidad. Para resolver cada uno de los cluster simplemente es aplicar algún algoritmo de resolución de TSP.

9.2. El Algoritmo

Empecemos la explicación de la clusterización:
La idea es utilizar un método de agrupamiento llamado k-means, que tiene como finalidad particionar un conjunto en k conjuntos y donde se cumple que los elementos que están en un determinado grupo cumplen ser los que están más cercanos al valor medio de ese conjunto (de ahora en más cluster).
Para esto vamos a definir a k como el número que resulta de dividir la demanda total de los vértices por el promedio de las demandas y será la primera cantidad de camiones con la que intentaremos realizar los ruteos. Los pasos que realiza nuestro algoritmo son:

- Lo primero es definir las posiciones de los k cluster como posiciones aleatorias delimitadas por los valores máximos de posición que toman los vértices. La posición de un cluster se llamará centroide y será lo que usemos para comparar las distancias con los vértices. Complejidad: O(k)

- Para cada par de los k cluster se iterará sobre los vértices buscando asignar cada uno al cluster más cercano. En caso de que el cluster no permita por capacidad ingresar otro nodo se analizará si el nodo (perteneciente) más lejano al cluster está más distante que el nodo que queremos ingresar y además si sacar ese nodo permite (por demanda) que ingrese el nuevo. En caso afirmativo sacamos ese nodo y guardamos el otro. En caso negativo nos queda libre el nodo. En ambos casos queda un nodo sin asignar. Complejidad: $\approx$ O(k^2 * cantidad-nodos) $\cdot$ O(TamCluster).

- Después de iterar cada nodo con cada cluster, se recorren los cluster y se re-calcula su centroide en función del promedio de posiciones de sus integrantes. En caso de que la nueva posición coincida con la actual y esto además se cumpla con todos los clusters entonces pasamos a la siguiente fase. En caso de que esto no pase se repite el paso anterior. Además tenemos un flag llamado tope-intentos-cluster-particular que está regulado por el usuario y permite fijar un número máximo de iteraciones con un mismo cluster antes de ir a la siguiente fase. Complejidad: O(k).

- Una vez finalizada una clusterización (ya sea porque se cumplieron la coincidencia de los centroides o porque se llegó al tope de iteraciones) verificamos si la clusterización es válida chequeando que todos los nodos estén asignados y que las demandas de los clusters están acotadas por la capacidad del camión. Complejidad: O(k).

- En caso de ser una clusterización inválida se repite el proceso desde el primer paso (es decir, considerando nuevos k clusters y asignando de nuevo posiciones aleatorias); en caso de ser válida se aplica TSP-GRASP a cada cluster, se analiza con la solución actual (si hay) y se actualiza en consecuencia. Después, en dependiendo de lo que permitan los flags, se repite desde el primer paso. Complejidad: O($cantidad - nodos^3$). Más adelante vamos a detallar TSP-GRASP y ahí queda mejor explicada la complejidad.

En los pasos significativos se aclara cuál es la complejidad para facilitar la comprensión del algoritmo y además de la complejidad total. Como observación a nivel implementación hay cosas que pueden mantenerse actualizadas constantemente en lugar de calcularlas cuando se las necesite. Un ejemplo de esto es para ver cuántos nodos están asignados en cada cluster o la suma de las demandas de un cluster: no hace falta recorrerlos a todos para calcular esto

sino que mantenemos actualizado su valor. Sin embargo sí se recorrerá el cluster para buscar el nodo más lejano debido a la propia dificultad de mantener actualizado este campo por el funcionamiento del algoritmo.

9.3. Análisis de Complejidad

El proceso de definir la complejidad de este algoritmo es algo complicado debido a las ideas sobre las que está construído: hay varios refinamientos controlados por el usuario y gran parte de la solución está basada sobre clustering, que como vimos en el trabajo práctico 2, la idea entre la percepción del ser humano respecto de puntos en el espacio y las decisiones que puede tomar un algoritmo en función de ciertos parámetros no siempre se correlacionan. Adicional a esto, este es un problema aún más complejo, debido a que la construcción de los clusters está definida además por las demandas de los clientes y las cargas que pueden soportar los clusters, de modo que al final del día son muchas las cosas a tener en cuenta.
Entonces como no podemos saber si para un determinado k <n vamos a poder encontrar solución, la manera en que vamos a establecer la complejidad es haciendo mención a cómo fijamos determinados parámetros de control y tomando como peor caso que lo único que podemos afirmar con total seguridad es que dados n clientes, cada uno con capacidad menor o igual a la capacidad de los cluster hay solución con k = n. Claramente esto es un resultado sin sentido a nivel práctico, pero es la forma de establecer la complejidad en peor caso.

Si definimos a los flags de control como constantes del problema y calculando la complejidad en peor caso, donde peor caso ≈ k = n. Las complejidades intermedias quedan:

- Repartir al cluster más cercano O(k^2 * cantidad-nodos) y contemplar las posibles extracciones de nodos ya asignados: O(TamCluster), total : O(k^2 * cantidad-nodos) * O(TamCluster)
- Aplicar TSP a la solución es: O($(cantidad - nodos)^3$)

Observemos que TSP se realiza con cada uno de los cluster y que la sumatoria de todos ellos es la cantidad total de vértices (cantidad-nodos). La complejidad entonces de aplicar TSP a todos es : O($(cantidad - nodos)^3$) dado que el tamaño de cada cluster es menor a la cantidad-nodos y podemos acotarlo. Total: O(k^2 * cantidad-nodos) * O(TamCluster) + O($(cantidad - nodos)^3$)
Si estamos en el caso en que no se encuentra solución para k <n en particular queda: O($(cantidad - nodos)^4$). Para clarificar más acerca de los flags de control:

1. tope-intentos-sin-aumentar-cluster: determina cuantas veces se puede repetir el proceso descrito arriba de armado de los cluster(que consiste particularmente en generar k cluster de posición aleatorias, iterar cada nodo para cada par de cluster y asignar por cercanía y verificar si es un cluster válido)
2. tope-intentos-mismo-cluster: determina cuantas veces se puede repetir el proceso de asignación de nodos a cluster más cercano mientras no se consiga que todos los cluster tengan su centroide coincidente con el promedio de posiciones de sus integrantes.
3. tope-intentos-optimizar: determina cuantas soluciones encontrar antes de terminar.

4. seguir-buscando-sol-con-mayor-k: una vez que encontramos solución podríamos seguir buscando otra si el flag tope-intentos-optimizar lo permite. En caso de que se alcance el tope de intentos sin aumentar cluster, ahora el algoritmo estaría buscando nuevas soluciones pero con más cluster. Este flag permite o no que esto suceda.

9.4. Algoritmo de resolución de problema de TSP

Este problema lo vamos a resolver mediante la técnica de vecinos más cercanos. Sin embargo le vamos a agregar un refinamiento. Básicamente utilizamos un método de búsqueda local de recorrido de soluciones conocido como GRASP. Nuestra solución inicial será una formada por procedimientos estricamente golosos y para las siguientes vamos a agregar a nuestro vecino más cercano una cuota de probabilidad, donde con probabilidad p se elegirá el vecino más cercano y con probabilidad (1-p) se elegirá otro nodo, dentro de lo que podríamos definir un algoritmo goloso aleatorizado, tomando a veces la mejor decisión y a veces otra cosa.
El valor de la probabilidad inicialmente será 1 y para las demás será producto de restarle 0.1 a p mientras sea mayor a 0.5.
Para la búsqueda local de soluciones utilizaremos una técnica de optimización conocida como 2-opt. La idea de este algoritmo consiste en dada una solución y dos nodos i y j del camino, se modificará la solución hacia una que cumpla: entre i y j el orden del sub-camino se invierte mientras que para los caminos de posición entre 0 e i y entre j+1 y el último nodo se mantienen igual. Esto lo vamos a repetir entre cada posición del camino solución y comparar la solución obtenida con este método con la mejor actual y actualizar en consecuencia.

El funcionamiento será: cada vez que se termine de construir una clusterización correcta se aplicara el algoritmo de resolución de TSP a cada uno de los cluster. Sea V la lista de nodos de un cluster en particular. Sea p el valor de la probabilidad y shifter = 0.1 el valor con el que se reducirá p.

1. Calculamos solución golosa con la técnica de vecinos más cercanos usando V y p = 1 y sea C el camino retornado. Complejidad: O($|V|^2$).

2. Dada esta solución inicial realizamos lo siguiente, mientras p >= 0.5

   - Iteramos sobre cada par de posibles posiciones de C sin considerar la primera y la última que son el depósito. Sobre cada uno de ellas aplicamos la técnica 2-opt. O($|V|^2$ * 2-opt) ≈ O($|V|^2$ * |V|) ≈ O($|V|^3$).
   - Verificamos si la solución nueva obtenida aplicando 2-opt es mejor que la actual y en ese caso la actualizamos.
   - p = p - shifter
   - calculamos otra solución golosa (ahora randomizada) con el p actualizado.
   - Verificamos si esta nueva solución es mejor que la actual y en ese caso actualizamos la actual.
   - Repetimos desde (2).

3. Finalmente tenemos el camino y el costo de haber ruteado el cluster y haberlo optimizado.

La complejidad de todo esto es O($|V|^3$). Notar que |V| = |C|, dado que las variaciones entre la primera solución golosa y las demás no sacan o agregan nodos sino los mueven de lugar.

El funcionamiento del algoritmo de vecinos más cercanos es muy intuitivo: el primero en guardarse en el camino C es el depósito que además lo marcamos como el nodo actual y a partir de ahí mientras haya nodos realiza:

- Mientras haya nodos
    - Buscar nodo más cercano al actual que no haya sido visitado. Además en este mismo recorrido por los nodos guardamos uno no visitado para usar como opción aleatoria. En caso de que no se encuentre ninguno porque están todos visitados ponemos en true un flag de todos visitados.
- Calculamos valor aleatorio entre 0 y 1 y si es mayor que p y todos visitados es false, entonces guardamos en el camino el nodo anteriormente almacenado para esto y lo marcamos como visitado y como nodo actual. Si es mayor que p pero todos visitados es verdadero marcamos en false el flag de hay nodos, agregamos al camino el nodo actual y termina. En caso de que el valor aleatorio no sea mayor a p, haríamos lo equivalente pero con el vecino más cercano buscado en el paso anterior y seguiríamos sujetos a lo que nos permitan los flags.

La complejidad de esto entonces es cuadrática respecto del tamaño de la lista de nodos de entrada (tamaño del cluster).
El procedimiento de 2-opt es lineal en el tamaño de la lista de entrada porque simplemente mueve algunos elementos de lugar.

10. Algoritmo de Simulated Annealing

10.1. Introducción

El Simulated Annealing es una metaheurística que busca aproximar el óptimo global de una función. Su nombre esta inspirado en una técnica utilizada en la metalurgia para reducir las imperfecciones de los metales. La idea de la metaherística la siguiente, a partir de una solución inicial de la función que se quiere optimizar, se van recorriendo soluciones vecinas al azar, si la solución a la que se va a pasar es mejor, se toma la mejor como solución actual, y si la solución a la que se va a pasar es peor que la solución actual, una probabilidad decide si te toma como actual o no. De esta forma se evita caer el mínimos (o máximos) locales. En esencia esta heurística funciona como una combinación de las heurísticas de Hill Climbing y Scouting. La variable que controla la probabilidad de llama **temperatura**, que tiende a 0 durante la ejecución del algoritmo, mientras mas alta sea esta más probabilidad hay de que se pase a un estado de peor costo. Para la vecindad de soluciones decidimos tomar lo siguiente, una solución es vecina de otra si:

- Todos los caminos son iguales entre las soluciones excepto por uno que tiene dos clientes adyacentes cambiados de orden, similar a la vecindad SWAP.
- El último nodo de un camino es cambiado con el primer nodo de otro, y los demás caminos siguen iguales con respecto a la solución original.

Esta vecindad se eligió ya que alterar el orden de los elementos de algún camino puede resultar en mejoras de costo, mientras que la parte de cambia el último nodo de un camino con el primero de un camino adyacente va mezclando los caminos para ver si esa mejora se puede conseguir entre nodos. Se elige un vecindario que cambia clientes de caminos adyacentes ya que las clusterizaciones tienden a poner cerca caminos que estén cercanos en el grafo. La probabilidad en nuestra implementación del algoritmo fue reducida dado al alto número de pasajes a soluciones malas que se ocasionan, haciendo que sea muy difícil que mejore el resultado de la solución.

10.2. El Algoritmo

El algoritmo implementado en el trabajo es el siguiente, partiendo de una solución, la cantidad de ciclos que se van a realizar, una temperatura máxima y un coeficiente de enfriamiento:

- NOTA: En las complejidades n es la cantidad de clientes o nodos del grafo.
- Se inicializan una solución final (que va a ser la mejor alcanzada) y una solución actual que se usa para avanzar entre las soluciones vecinas. Para copiar la solución se tienen que copiar todos los caminos, como eso involucra copiar a todos los clientes, la complejidad de ambas copias es $O(n)$. Luego se entra al ciclo principal del algoritmo.
- Se genera el vecindario de la solución actual, la complejidad de esta parte se es $O(n^3)$ y se justifica luego.
- Se elige uno de los elementos del vecindario al azar, esto puede ser implementado en $O(1)$, se copia a la solución tentativa en $O(n)$.
- Se pide un numero al azar entre 0 y 1 inclusives y se asigna a una variable r, que va a ser la que decida si la probabilidad de pasar de una solución a otra se da o no.
- Se calculan los costos de la solución actual y la solución tentativa y la diferencia entre ambos en $O(1)$.
- Si el costo de la solución tentativa es mejor que el actual se copia en la solución actual con una complejidad de $O(n)$.
- Sino se pasa a la solución tentativa según la probabilidad, si se pasa de solución esto cuesta $O(n)$.
- Si la solución actual es mejor que la solución final entonces se actualiza la solución final en $O(n)$.
- Se reduce la temperatura multiplicandola por el coeficiente de enfriamiento, esto es $O(1)$.
- Luego continúa el ciclo hasta que se llegue a la cantidad de pasos ingresada.

En forma de código sería asi:

```
    solucionProb simulatedAnnealing(solucionProb& solucionInicial,
    int cantPasosMax, float temperaturaMax, float coefEnfriamiento){
    float tempActual = temperaturaMax;
    solucionProb solFinal = solucionInicial;
    solucionProb solActual = solucionInicial;

    int cantPasos = 0;
    while(cantPasos < cantPasosMax){

        vector<solucionProb> vecindario = generarVecindario(solActual);

        int index = elegirUno(vecindario);

        solucionProb solTentativa = vecindario[index];

        //Calculo la probabilidad de pasar a esta solucion
        float r = random(0, 1);

        float prob = -1;

        float energiaEstadoActual = solActual.getCostoSol();
        float energiaNuevoEstado = solTentativa.getCostoSol();
        float deltaE = energiaEstadoActual - energiaNuevoEstado;

        if(energiaNuevoEstado < energiaEstadoActual){
            solActual = solTentativa;
        }
        else{
            prob = exp(((deltaE) / tempActual));
        }

        if(prob - 0.5 >= r){
            solActual = solTentativa;
        }

        //Guardamos la mejor
        if(solActual.getCostoSol() < solFinal.getCostoSol()){
            solFinal = solActual;
        }

        tempActual *= coefEnfriamiento;
        cantPasos++;
    }

    return solFinal;
}
```

Generar el vecindario consiste en dos partes:

- Generar los vecinos que tienen todos los caminos iguales salvo por dos clientes adyacentes de un camino cambiados de orden, esto lo hacemos copiando los caminos de la solución original y usando generarCaminos para crear las variaciones de este tipo de todos los caminos.
- Luego generamos los vecinos donde los caminos entre la solución original y la vecina son iguales salvo por dos caminos que intercambiaron el último nodo de uno con el primer nodo de otro. Esto se hace creando una copia de los caminos y cambiando el último nodo del i-ésimo camino con el primer nodo del $i+1$-ésimo camino hasta que se hayan cambiado todos.

```
vector<solucionProb> generarVecindario(solucionProb& base){
    vector<solucionProb> sol;

    for(int i=0; i < base.getCaminos().size(); i++){
        vector<vector<Nodo>> variaciones = generarCambios(base.getCaminos()[i]);
        vector<vector<Nodo>> caminos = base.getCaminos();

        for(int j=0; j < variaciones.size(); j++){
            caminos[i] = variaciones[j];
            sol.push_back(solucionProb(base.getCapacidad(), caminos));
        }
    }

    for (int i = 0; i < base.getCaminos().size()-1; ++i) {
        vector<vector<Nodo>> nueva = base.getCaminos();
        Nodo n = nueva[i][nueva[i].size()-2];
        nueva[i][nueva[i].size()-2] = nueva[i+1][1];
        nueva[i+1][1] = n;
        if (capacidadAlcanza(base.getCapacidad(), nueva[i]) && capacidadAlcanza(base.getCa
            sol.push_back(solucionProb(base.getCapacidad(), nueva));
    }

    return sol;
}
```

Generar cambios toma un camino y crea los caminos que pertenecen a la vecindad SWAP del mismo.

```
vector<vector<Nodo>> generarCambios(vector<Nodo>& base){
    vector<vector<Nodo>> sol;
    for(int i=1; i < base.size() - 2; i++){
        vector<Nodo> nuevo = base;
        nuevo[i] = base[i+1];
        nuevo[i+1] = base[i];
        sol.push_back(nuevo);
```

```
    }
    return sol;
}
```

10.3. Análisis de Complejidad

Para generarVecindario, veamos la complejidad de crear el vecindario de soluciones:

- crear la parte del vecindario que se parece al SWAP necesita que se recorran los caminos, por cada camino se crea una copia de los caminos originales, esto se realiza en $O(n)$.

- generarCambios crea los cambios posibles de un camino en $O(|c|^2)$ siendo c el camino a partir del cuál se van a crear los cambios. Para un camino c generarCambios crea $|c|-1$ caminos distintos, estos caminos, de a uno reemplazan al camino c y se agregan a la vecindad. Reemplazar el camino c por uno de los posibles caminos cuesta $|c|$ ya que tienen el mismo largo, entonces crear una solución nueva para agregarla a la vecindad cuesta $O(n)$ debido a que tiene que copiar los clientes y calcular el costo de la misma (que también es $O(n)$ ya que necesitas recorrer a los clientes de todos los caminos para calcular el costo de la solución).

Siendo CAM la cantidad de caminos, hasta el momento tenemos $O(CAM*(|c|-1+n+|c-1|*(|c|+n)))$, veamos la segunda parte de la generación de vecindad:

Para cada camino, se copian los caminos, que cuesta $O(n)$, luego en $O(1)$ se reemplaza el último nodo del i-ésimo camino con el primer nodo del i+1-ésimo camino, esto es una copia de clientes y es $O(1)$. Se comprueba si las demandas de los caminos con los nodos cambiados son válidas en $|c|$ y si lo son, en $O(n)$ se agrega la nueva solución a la vecindad, y tendríamos hasta el momento $O(CAM*(n+1+1+|c|+n)) = O(CAM*(n+|c|+n))$

Entonces la complejidad de la generación del vecindario sería $O(CAM*(|c|-1+n+|c-1|*(|c|+n)))+O(CAM*(n+|c|+n)) = O(CAM*(|c|-1+n+|c-1|*(|c|+n))+CAM*(n+|c|+n)) = O(CAM*(|c|-1+n+|c-1|*(|c|+n)+(n+|c|+n))$ y como $|c|$ puede ser acotado por n ya que a lo sumo un camino tiene n clientes $O(CAM*(|c|-1+n+|c-1|*(|c|+n)+(n+|c|+n)) = O(CAM*(n+n+n*(n+n)+(n+n+n)) = O(CAM*(n+n^2+(n)) = O(CAM*(n^2))$ y como la cantidad de caminos CAM a los sumo también se puede acotar por n, $O(CAM*(n^2)) = O(n^3)$ Por lo tanto generarVecindario es $O(n^3)$.

En la función de simulated annealing entonces tenemos con lo planteado en la sección del algoritmo una complejidad de $O(n+K*(n^3+n+n+n+n)) = O(K*(n^3)+n) = O(K*(n^3))$ donde K es la cantidad de pasos que puede realizar la función

Concluimos, que la complejidad total del algoritmo es $O(K*(n^3))$.

11. Experimentación: Heurísticas constructivas

11.1. Parte 1: Primeras ideas sobre la performance

En esta sección en particular, se analizará la performance de cada algoritmo por separado, verificando complejidades y calculando un techo de ineficiencia cuando resulte de interés, para los distintos algoritmos presentados. En cuanto a complejidad, para verificar esto, vamos a realizar unos gráficos que contemplen muchas instancias de los algoritmos, de distintos tamaños y algunas caracteristicas que puedan hacer que algún algoritmo funcione más rapido o más lento.
Vamos a verificar los algoritmos de interés por separado, graficando el tiempo en función del tamaño de la entrada, esto nos debería dar una idea de que complejidad tiene, pero este gráfico nos va ayudar a distinguir si la complejidad es lineal, polinomial o exponencial, en general es difícil distinguir en estos gráficos si la función es por ejemplo $O(n^2)$ o $O(N^3)$.
Para poder distinguir esto, vamos a hacer gráficos del tiempo dividido la complejidad esperada en función al tamaño de la entrada. Por ejemplo, si tenemos un algoritmo que tarda $O(n^5)$ vamos a graficar t/n^5 en función de n siendo t el tiempo que tarda el algoritmo en resolver la instancia de tamaño n. Este tipo de gráficos nos va a ayudar a verificar que la complejidad teórica que deducimos para cada algoritmo, es correcta y certera respecto a lo que tarda en realidad,esto va a ser así si el gráfico resultante tiene una tendencia a ser constante.

11.2. Parte 2: Algoritmo de Savings

Como primera observación relevante para la experimentación, podremos decir que en particular el algoritmo siempre tardará lo mismo en sus dos primeros pasos para instancias del mismo tamaño, es decir, no habrá un peor o mejor caso en el que una de ellas tarde más o menos en esta parte de la ejecución: calcular los savings y armar las rutas individuales. En donde se podrá observar una diferencia de tiempo (la cual en nuestra opinión no será muy notable) entre instancias del mismo tamaño será en la parte de combinar rutas, ya que el tiempo que lleve encontrar la ruta que termine en i como la que empiece en j en cada iteración no siempre será igual. Los análisis sobre los tiempos de ejecución de Savings serán analizados en conjunto en la experimentación final.

11.2.1. Ineficiencia de Savings respecto del Óptimo

A continuación se evaluará el algoritmo con algunos sets de instancias para observar cualitativamente la solución obtenida respecto de la solución óptima. Para realizar el análisis correspondiente también se mencionará la cantidad de camiones utilizados en dichas soluciones a fines de contemplar varios aspectos de las mismas. En base a esto se podrá observar **qué tan mala puede llegar a ser la solución**, y se procederá a calcular un promedio con el objetivo de poder hacer una generalización de los resultados obtenidos.

Con todo esto en mente, se procede al primer de los experimentos:

11.2.2. Primer set de Instancias: Set A (Augerat, 1995)

Datos de las instancias que nos interesarán para este experimento:

- Cantidad de clientes, lo cual llamaremos 'n' irá de 31 a 79.

- Capacidad de los camiones: en particular para este set de instancias la capacidad será fija y será 100.
- Las coordenadas de los clientes son generadas aleatoriamente dentro de un cuadrado de lado = 100.
- La demanda promedio será de 15. Las demandas son seleccionadas de una distribución uniforme U(1,30), además n/10 de esas demandas son multiplicadas por 3.

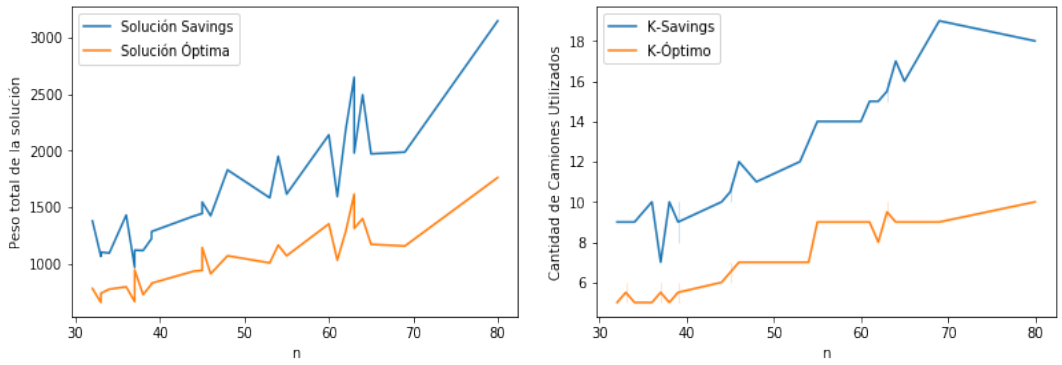


Figura 4: Izquierda: comparación de los pesos finales. Derecha: comparación de la cantidad de camiones utilizados.

Aquí, podemos ver que la curva de peso de las soluciones de Savings será muy parecida a la de la solución óptima pero con una constante de por medio. Y que la cantidad de camiones utilizados en Savings siempre se mantiene a una distancia mayor a 2 del óptimo lo cual hará que este aspecto de la solución se aleje bastante de lo deseado.

Si bien esta información es relevante, se ampliarán los detalles de la comparación con el fin de obtener un porcentaje de cuánto más se utiliza en la solución provista:

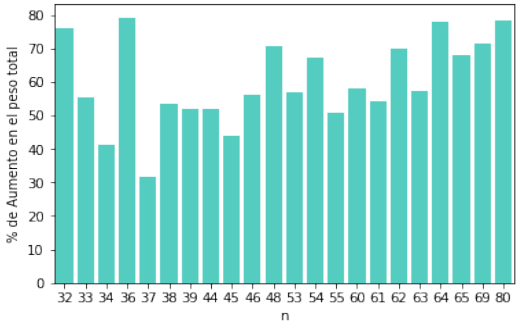


Figura 5: Aumento en porcentaje del peso de Savings respecto de la solución óptima

Y lo que este gráfico nos dirá es que para esta instancia, el algoritmo de Savings puede llegar a ser hasta un 80 % peor que la solución óptima.

Notar que este tipo de comparaciones no tendrán el mismo significado e importancia si se lo analiza en cuanto a la cantidad de camiones ya que ese no es el aspecto principal a analizar.

También podemos mencionar que para esta instancia, si sacamos un promedio de cuán peor se puede llegar a ser, este dará que es del 57.80 %, lo cual nos indicaría que nuestra solución en promedio es por lo menos una mitad más de lo que se espera.

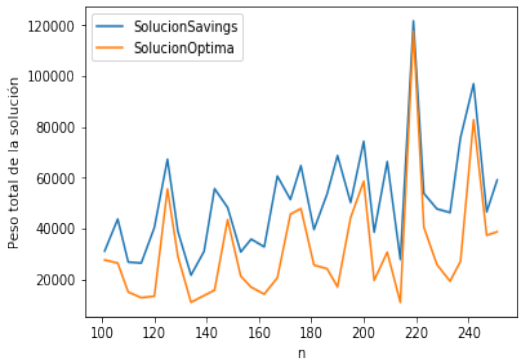
Para continuar con esta experimentación se pasará a analizar un set de instancias aún más grande:

11.2.3. Segundo set de Instancias: Set Uchoa (2014)

Datos de las instancias que nos interesarán para este experimento:

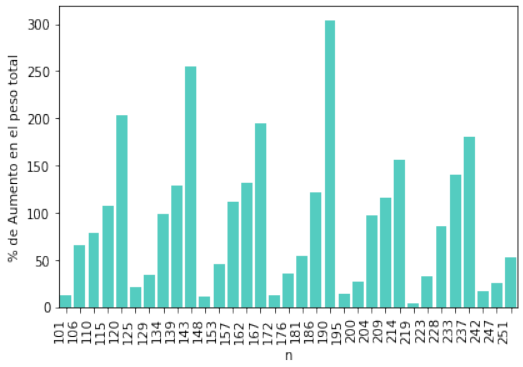
- Cantidad de clientes, lo cual llamaremos 'n' irá de 100 a 250 (En particular el set tiene instancias de más clientes pero solo tomamos las primeras consecutivas que su solución óptima es conocida).
- Capacidad de los camiones: para este set de instancias esta será variable.
- Las demandas irán de 1 a 10.
- Las posiciones de los clientes son generadas aleatoriamente.

A continuación la comparación de pesos:



Al igual que con el anterior set de instancias, aquí se podrá observar que la curva obtenida por las soluciones del algoritmo será parecido a su par óptimo pero con una constante de por medio, la cual en alguno casos será mínima. Por ejemplo para la instancia de 220 clientes podemos observar que nuestra solución sólo cuesta un 4111 más, lo que en porcentaje será un 3.49 % más.

Ahora, si se observa el gáfico con las diferencias porcentuales



se puede ver que el algoritmo de Savings puede llegar a ser hasta un **300 % peor** que lo esperado, lo cual nos indica que la solución obtenida puede llegar a ser 3 veces más grande que el óptimo, lo cual también supera ampliamente al 80 % obtenido por la instancia anterior.

11.2.4. Conclusión

Como conclusión final se recordará el 300 % obtenido previamente el cual apuntamos obtener para saber lo ineficiente que puede llegar a ser la solución.

Para poder hacer una generalización se tendrá en cuenta el promedio de cuán peor puede llegar a ser la solución brindada, para el cual se tendrán en cuenta todas las instancias de ambos sets mencionados previamente.

Dicho promedio arroja un 75.54 % de aumento relacionado al peso, este número no sería aceptable en cuanto a distancia relativa de un resultado óptimo, pero como se verá después, este algoritmo tendrá un tiempo de cómputo mucho menor en todos los casos a las heurísiticas que poseen una optimización, lo cual hará que dicho algoritmo posea un trade-off en cuanto a la relación costo Solución - tiempo de ejecución.

11.3. Parte 3: Nuestras Heurísticas Constructivas

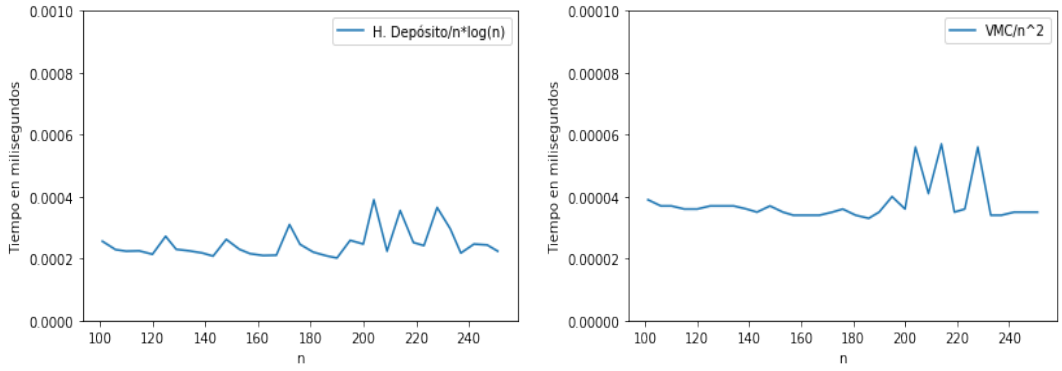
Ahora nos toca experimentar con las heuristicas que se nos ocurrieron a nosotros, como dijimos anteriormente, desarollamos dos heuristicas, una que recorre por los más cercanos al deposito, y otra que busca al vecino más cercano hasta que se agote la capacidad y despues vuelve.

Como recordatorio, habíamos dicho que la complejidad de la H. más cercano al deposito tenía una complejidad de $O(n * log(n))$ y la heuristica del VMC (vecino más cercano) tenía una complejidad de $O(n^2)$.

Ahora vamos a intentar verificar la complejidad calculada anteriormente con las mismas técnicas que usamos anteriormente en Savings.

11.3.1. Verificando las complejidades

Hasta ahora venimos viendo a ojo que los gráficos de los tiempos se parecen a las funciones de complejidad estipuladas, para verificar esto con un poco más de seriedad, vamos a tomar un set de datos y vamos a graficar los algoritmos divididos por la funcion de complejidad esperada, como ya dijimos anteriormente, si esto nos da constante se puede esperar que nuestra



aproximación teórica se acerque bastante a la realidad. En particular para este experimento se tendrán en cuenta el Set de Instancias Uchoa con n de 100 a 250.

Podemos ver que los gráficos tienen una dirección constante, a pesar de algunos picos que pueden ser por distintas razones. Por lo tanto podemos suponer que nuestras cotas de complejidad son bastante certeras aunque obviamente no lo podemos confirmar completamente sin una verificación formal.

11.3.2. Ineficiencia de las Heurísticas Golosas respecto del Óptimo

Aquí vamos a comprobar que tán cerca están nuestras Heurísticas de la solución óptima. Esto lo vamos a hacer con la misma técnica que utilizamos en el algoritmo de Savings, utilizaremos gráficos donde se muestra nuestra solución y la solución óptima en función del tamaño de la entrada en distintos set de datos. Y luego se observarán los gráficos obtenidos de las comparaciones porcentuales para arribar a una conclusión.

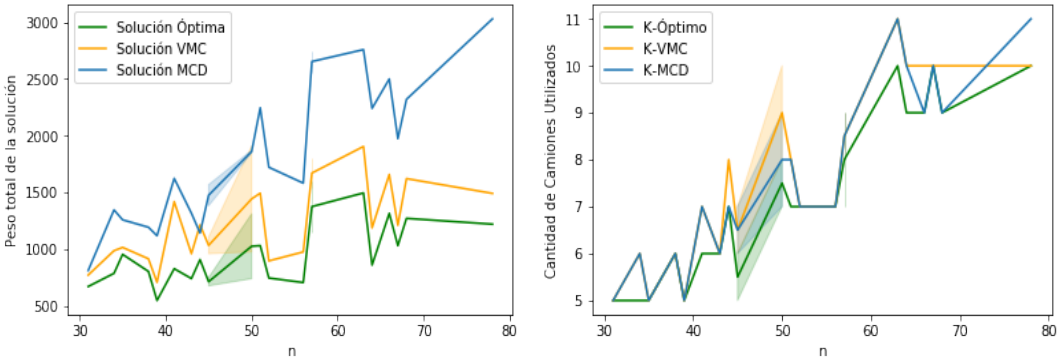
Para comenzar con la experimentación se introducirá el primer set de instancias a analizar.

11.3.3. Set de Instancias: Set B (Augerat, 1995)

Datos de las instancias que nos interesarán para este experimento:

- Cantidad de clientes, lo cual llamaremos 'n' irá de 31 a 78.
- Capacidad de los camiones: en particular para este set de instancias la capacidad será fija y será 100.
- Las coordenadas son generadas de tal manera que se crean NC regiones con K $\leq$ NC
- Las demandas son seleccionadas de una distribución uniforme U(1,30), además n/10 de esas demandas son multiplicadas por 3.

A continuación gráficos que resultarán interesantes a la hora de analizar la "*calidad*" de la solución de ambas heurísticas:

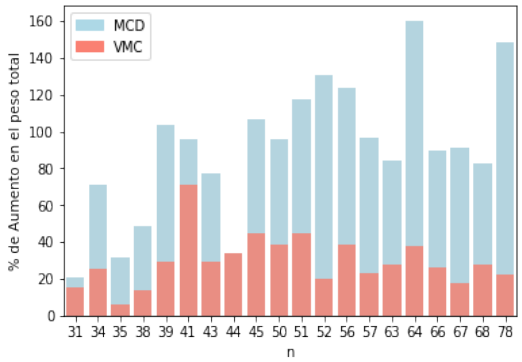


Observación: las áreas del gráfico donde hay sombreado se deben interpretar como varias muestras de igual tamaño con distintas soluciones.

Como se puede observar, en cuanto a cantidad de camiones utilizados, ambas heurísticas mantendrán números muy cercanos al óptimo y esto se deberá a su componente "*golosa*" ya que estas deciden cerrar un recorrido solo cuando el próximo cliente a visitar hace que se exceda la capacidad total del camión. Esto en la mayoría de las casos hará que la diferencia entre la cantidad de camiones del resultado óptimo y la de las heurísticas no sea tan grande.

Ahora en cuanto al peso total de la solución, se podrá notar que VMC tendrá en general un mejor resultado que MCD, lo cual va en línea con el trade-off antes mencionado de las complejidades: MCD tiene una complejidad menor que VMC ($\mathcal{O}(nlog(n))$ vs $\mathcal{O}(n^2)$) pero sus resultados serán relativamente peores. Eso no querrá decir que MCD no pueda dar resultados cercanos al óptimo, pero sí que esto no ocurre tan frecuentemente. Inclusive en esta muestra se puede observar un ejemplo cuando n es 44 en el cual MCD devuelve un resultado de peso 1143.32 el cual es mejor que el de VMC (1214.27) y esta sólo a un 25.77 % del óptimo (909).

Luego, si se analizaran estas diferencias respecto del óptimo porcentualmente, se obtendría el siguiente gráfico



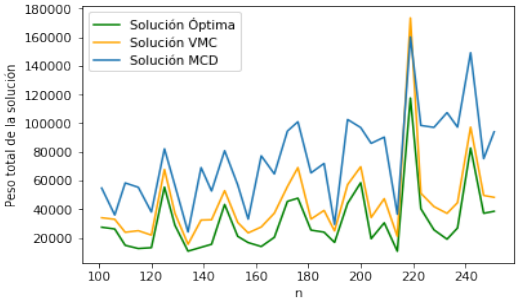
Del cual se puede observar que MCD puede llegar a ser un 160.21 % peor que el óptimo ($n = 64$) y que VMC lo será hasta en un 71.18 % ($n = 41$). En comparación al algoritmo de savings, estos gráficos varían mucho más al variar el tamaño de la entrada, pero igual mantienen una tendencia a un espacio constante entre los resultados, lo que es interesante.
No conformes con el resultado anterior, aumentaremos el muestreo para obtener más resultados de nuestro interés:

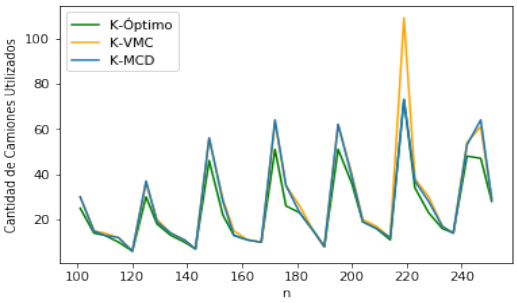
11.3.4. Segundo set de Instancias: Set Uchoa (2014)

Datos de las instancias que nos interesarán para este experimento:

- Cantidad de clientes, lo cual llamaremos 'n' irá de 100 a 250 (En particular el set tiene instancias de más clientes pero solo tomamos las primeras consecutivas que su solución óptima es conocida).
- Capacidad de los camiones: para este set de instancias esta será variable.
- Las demandas irán de 1 a 10.
- Las posiciones de los clientes son generadas aleatoriamente.

A continuación los gráficos que nos resultarán de interés:

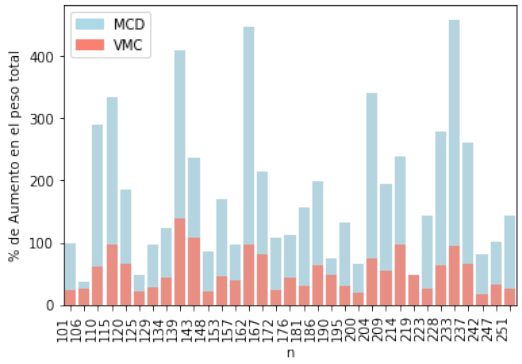




Tal como en el primer experimento para el set Augerat B, se puede observar que la cantidad de camiones utilizados es casi la óptima salvo en un caso de VMC ($n = 219$) en el cual este usa 36 camiones más que la solución óptima.

En cuanto a lo importante, que será el peso total de las soluciones, se podrá observar la misma tendencia que se anunció para el set anterior: los resultados de MCD son peores que los de VMC salvo en un caso ($n = 219$) por lo que se podría inferir que MCD serviría como una suerte de cota superior a VMC.

Con respecto a las diferencias porcentuales, se obtuvieron los siguientes resultados:



De los cuales se puede ver que MCD puede llegar a ser un **458.65 % peor** ($n = 233$) y que VMC tendrá como techo de mal desempeño un 139.73 % ($n = 139$). En el primer caso se notifica que MCD podrá llegar a ser hasta casi 5 veces más que la respuesta óptima lo cual signfica que su trade-off costo Solución-tiempo de cómputo podrá resultar muy caro a la hora de rutear vehículos.

11.3.5. Conclusión

Como parte de las conclusiones de esta experimentación se menciona que:

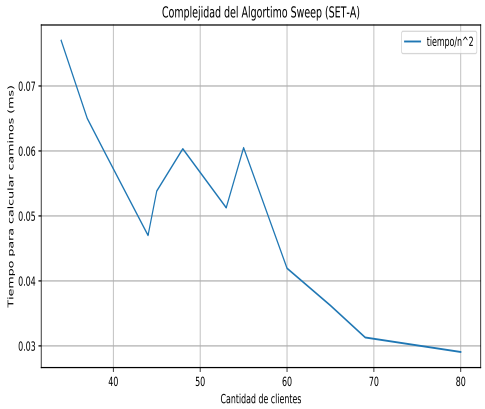
- Solo en el 3.57 % de los casos (2 de 56) MCD dio un resultado mejor que VMC.
- Como se mencionó antes, MCD puede llegar a ser hasta un 458.65 % peor y VMC un 139 %.
- En promedio MCD arroja un resultado un 144.47 % mayor y VMC un %43.83 siendo el último de estos un muy buen promedio para su complejidad teórica($\mathcal{O}(n^2)$)

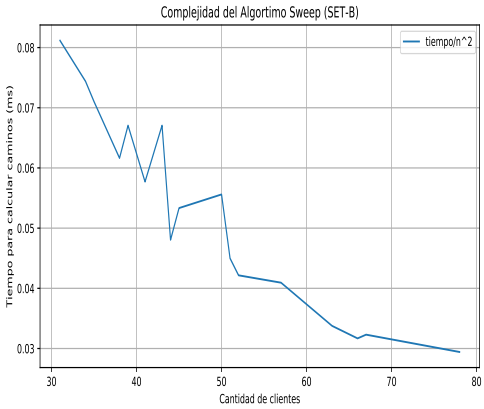
12. Experimentación: Heurísticas constructivas de cluster-first, route-second

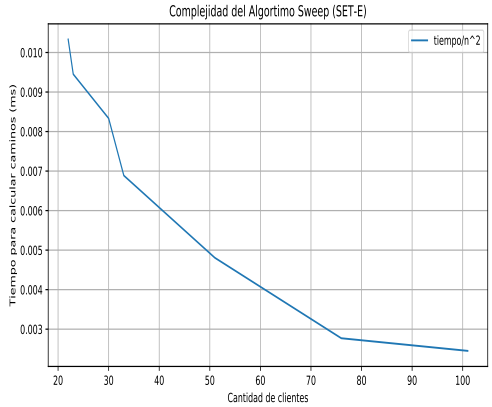
12.1. Parte 1: Sweep Algorithm

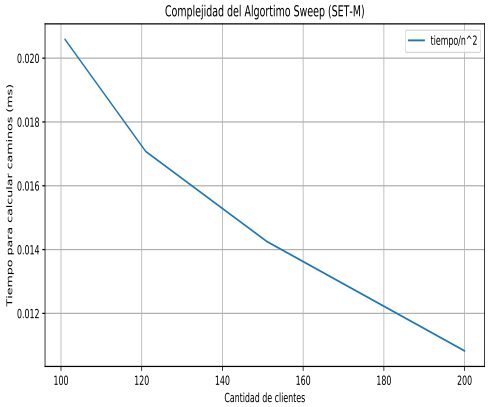
12.1.1. Verificando las complejidades

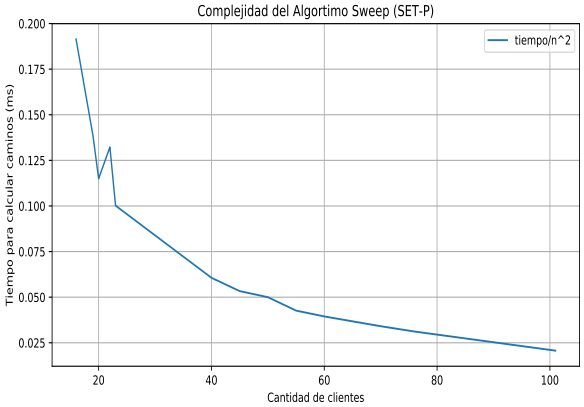
Para corroborar que la complejidad calculada anteriormente del algoritmo de sweep sea la correcta tomamos los tiempos (en milisegundos) que tardo en resolver los Sets usados en la experimentación anterior y los dividimos por la complejidad que supusimos que tiene el algortimo. Los resultados fueron los siguientes graficos:





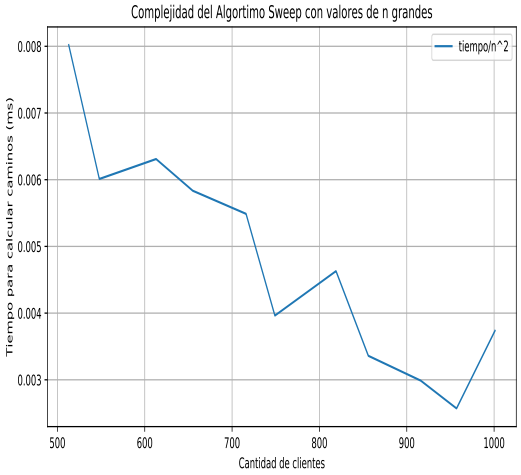






Como todas los graficos de función dividido complejidad resultaron en funciones que tienden a decrecer a 0, estamos inclinados a creer que la complejidad $O(n^2)$ es correcta.

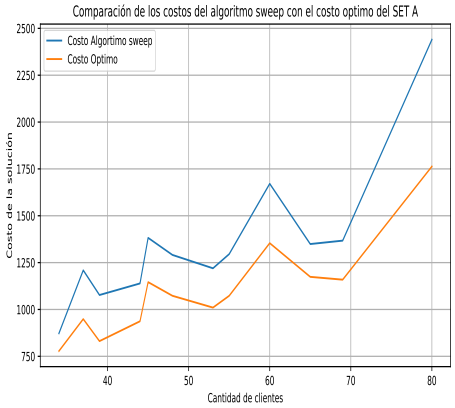
Para poder verlo mejor vamos a realizar otro experimento de este tipo con valores de n más grandes para ver si el comportamiento asintotico es el mismo que en los casos anteriores. Tomamos unos problemas del Set X con n entre 500 y 1000, resultando en el siguiente gráfico:

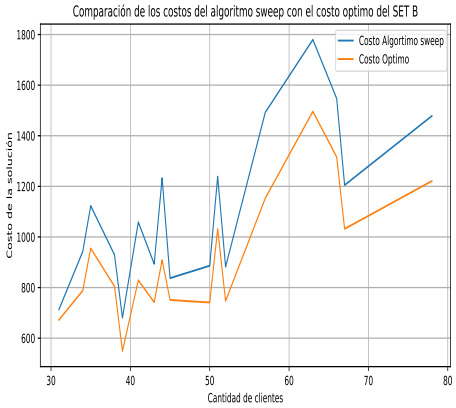


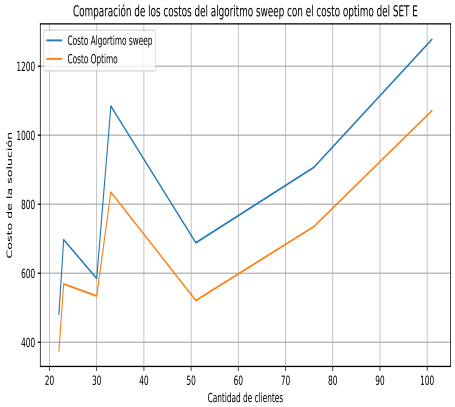
Como en los casos anteriores, el resultado fue un grafo con una funcion que tiende a decrecer a 0, por lo que podemos suponer que la complejidad $O(n^2)$ es correcta.

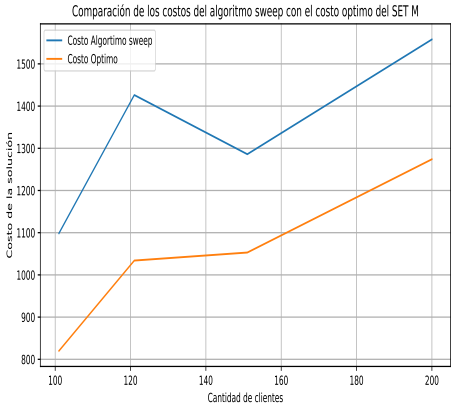
12.1.2. Categorización de la solución

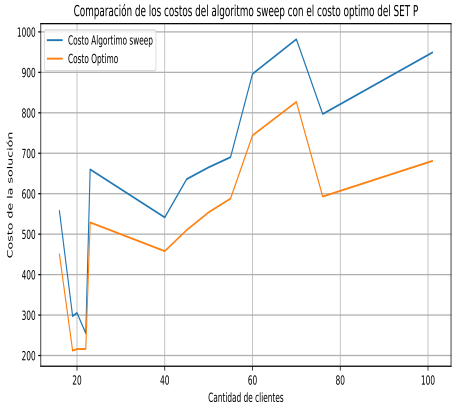
Vamos a ver como se comparan las soluciones del algortimo sweep con las soluciones óptimas de varios Sets de problemas para poder ver la presición del algoritmo. Se eligieron varios problemas de los Sets A, B, E, M y P de los cuales se produjeron los siguientes graficos:











Se puede ver que en todos los graficos el costo del camino generado por sweep esta cerca del costo del camino optimo, pareciendo diferir en un valor casi constante salvo en ciertas deformindades, por lo que la solución de sweep no parece ser muy mala con respecto a la optima. También, a partir de estos graficos podemos deducir que el tamaño de la entrada no influye en el costo del camino producido por el algoritmo.

12.2. Parte 2: Algoritmo alternativo al Sweep Algorithm

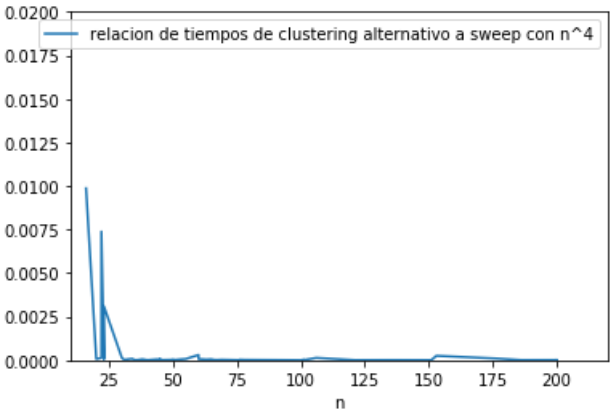
Como mencionamos durante la explicación del algoritmo fijaremos los flags de control en constantes fijas para los grupos de instancias. A priori el flag de seguir buscando solución con más cluster la vamos a deshabilitar, la cantidad de soluciones que queremos obtener para optimizar es 1, la cantidad de iteraciones hasta aumentar de cluster la fijamos en 15 y la cantidad de iteraciones para intentar formar un cluster en 5. Proyectamos experimentar con casos de entre 20 a 200 nodos y nos parece adecuada la proporción entre el tope de repeticiones fijado y los tamaños de los sets aunque en busca de experimentar acerca de la funcionalidad de estos flags los iremos modificando a lo largo de la experimentación. Los casos de test que utilizaremos son un subconjunto de los de A, B, E, F, M , P y X tratando de conseguir una

distribución lo más uniformemente posible entre los valores de n.

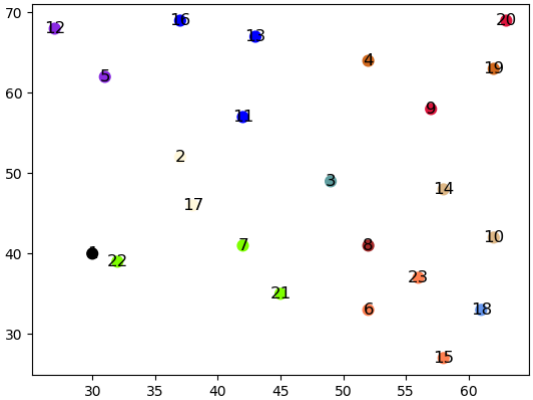
Algunas de las preguntas que nos surgen son: cuán alejado de la solución óptima esta nuestra solución; cuánto afecta la relación entre la cantidad de clientes y la cantidad de cluster que se necesitarán; cómo afecta el promedio de demandas y su relación con la capacidad máxima son algunas de ellas.

12.2.1. Verificando las Complejidades

Comenzamos con un gráfico que muestra la relación de complejidad calculado anteriormente.



Podemos notar que en el primer intervalo de n hay determinados saltos hasta que termina por asentarse. Analizando qué podía producir esto encontramos que los casos problemáticos eran:P-n16-k8, P-n22-k8, P-n23-k8. En los tres casos el algoritmo no pudo encontrar el ruteo con la cantidad de vehículos óptimo. Incluso para el caso n22-k8 necesitó de dos más. A su vez encontramos que en estos casos las demandas de los clientes están cercanas a las capacidades de los vehículos [0,31] y 31;[0,2500] y 3000;[0,30] y 40 respectivamente.
Clusterización de P-n23-k8

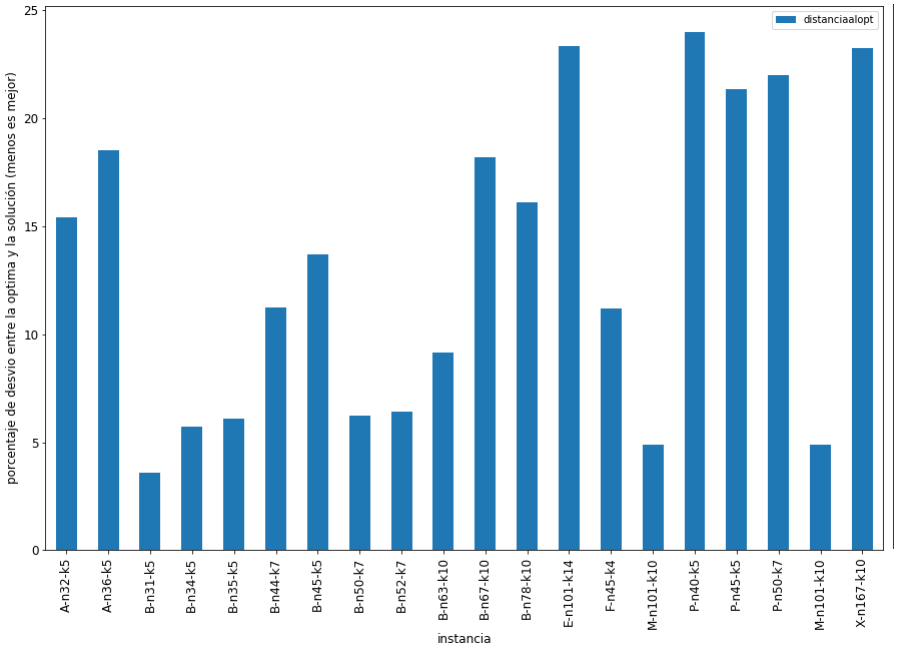


Si tenemos en cuenta que los flags de control y que la cantidad de clusters en estos casos están muy cercanos al valor de n y que gráficamente los puntos están distribuidos de forma que no ayuda a la clusterización estamos frente a casos en donde aplicar este tipo de técnicas no brinda soluciones de calidad.

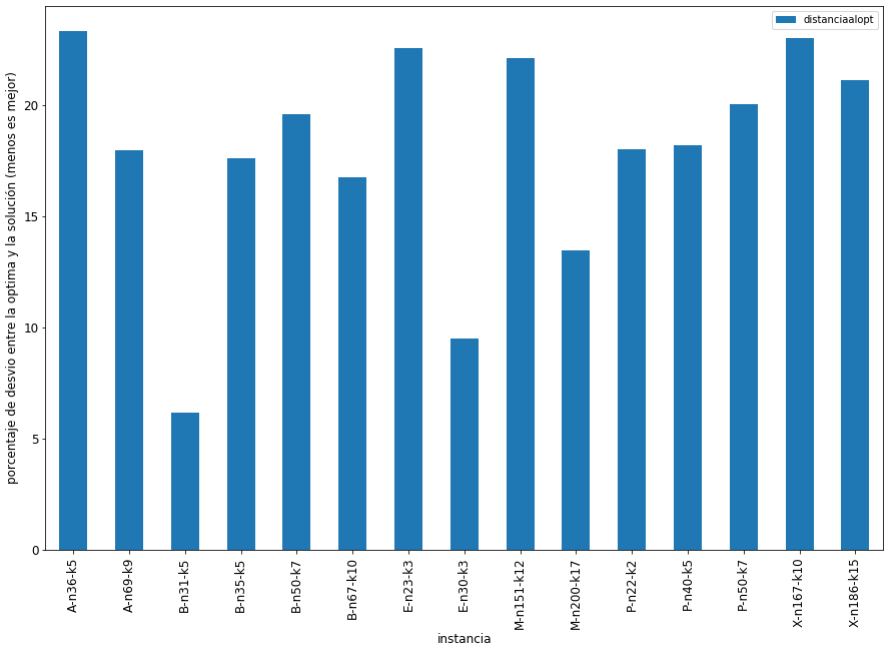
12.3. Sweep vs Alternativa: Comparando Soluciones

12.3.1. Comparando con el Óptimo

Centrémonos en las soluciones de buena calidad producidas por el algoritmo. Primero definamos que vamos a tomar como buena calidad: debe haberse ruteado con la cantidad de camiones óptima y no superar en 25 % el costo total del ruteo.
Algoritmo de alternativa de Sweep:



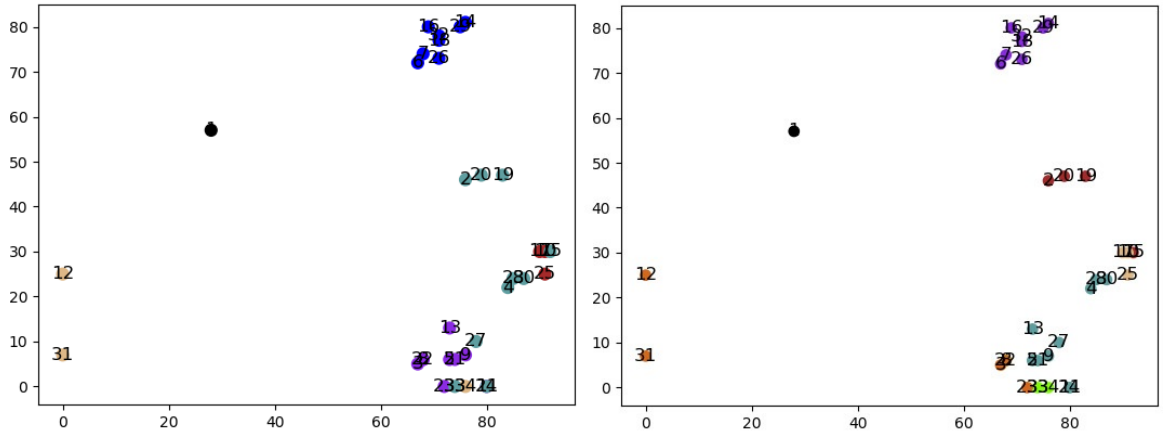
Algoritmo de Sweep:



Los anteriores gráficos de barras describen para cada instancia el porcentaje de desvío que tiene respecto de la solución óptima, es decir, cuánto más 'costoso' son los caminos ruteados. En este gráfico menos es mejor. La información relevante que brinda son aspectos que podemos reconocer: si bien corrimos instancias hasta alrededor de 200 clientes las instancias de solución óptima del gráfico sólo alcanzan hasta alrededor de 100 clientes, además para el mismo set de instancias Sweep consiguió un porcentaje de 36 % contra 40 % del algoritmo alternativa de efectividad para resolver el problema de forma óptima. Por otro lado no parece haber una relación estricta entre mayor cantidad de clientes y mayor desvío dado que el test M-n101-k10 tiene un n considerablemente más grande que la demás instancias y sin embargo se ruteó de forma cercana a la óptima. Por ejemplo el test B-n34-k5 Sweep no logra rutearlo con 5 vehículos y en cambio lo hace con 6.

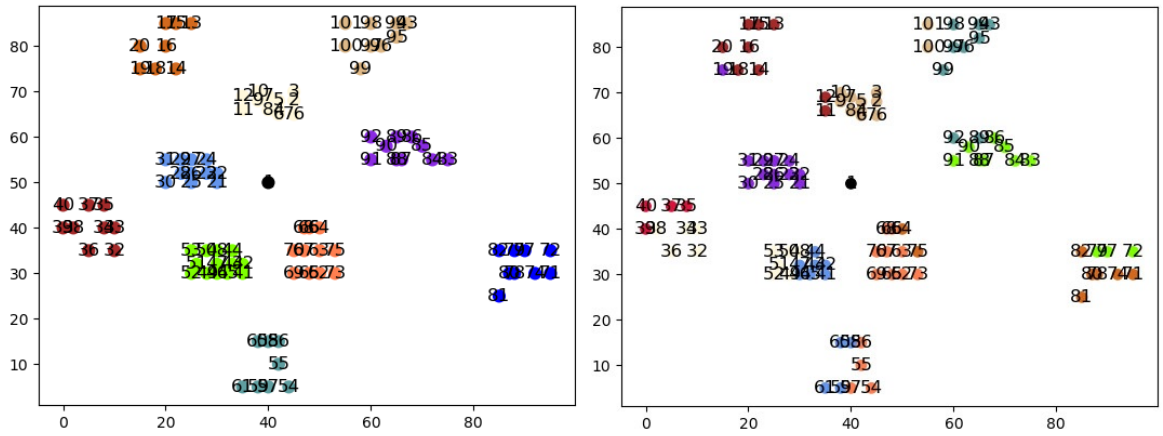
12.3.2. Problemas en la Clusterización

Clusterización con algoritmo alternativa a Sweep a la izquierda y Sweep a la derecha.

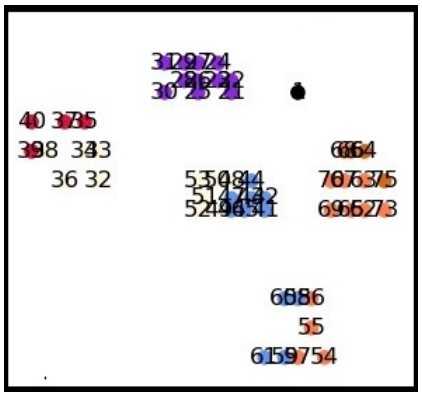


En este tipo de casos al no tener un refinamiento entre nodos asignados y nodos por asignar como sí tiene el algoritmo alternativa sucede que no se consigue completar lo máximo posible las demandas de los vehículos. En estos casos sucede que se necesitan agregar nuevas rutas para satisfacer clientes que en caso de haber hecho algún tipo de pasada adicional o intentado mover clientes previamente asignados se hubiera descubierto que no era necesario.

Uno de los casos que parece interesante analizar es el M-n101-k10. La alternativa de Sweep lo resolvio de forma óptima mientras que Sweep necesitó un vehículo más. Al anaizar en detalle encontramos lo siguiente:



Naturalmente este caso forma parte de los que intuitivamente es bueno para utilizar este tipo de algoritmo: los clientes están organizados en grupos idénticos a los que podrían clusterizarse para resolver el problema con TSP. Sin embargo Sweep no consigue clusterizar de la mejor manera y por lo tanto requiere incrementar los vehículos. La idea de picking de vértices de Sweep podría explicarlo: estamos frente a un algoritmo que trata de 'barrer' con un mismo cluster todos los nodos que queden a su alcance mientras las demandas lo permitan. La propia variación de demandas de los clientes podría generar una 'desincronización' entre el momento óptimo para cerrar un cluster por capacidad o por lejanía de los nodos.



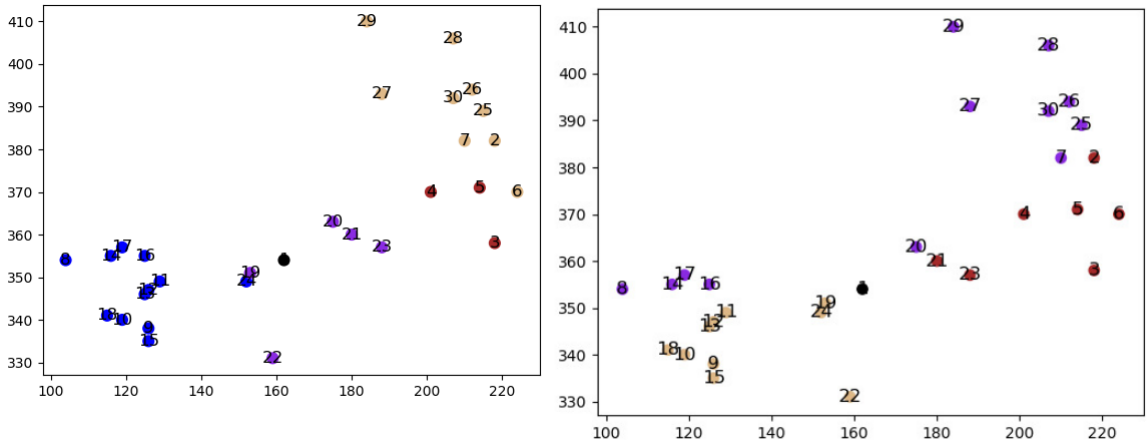
Esa porción de la clusterización condice bastante con la explicación dada y con lo verificado paso a paso en la corrida del test por Sweep: cuando termina de armar el cluster blanco

es porque no puede ingresar ningún nodo (de los que posteriormente serán rojos) dado que excederían la capacidad. Los rojos pasan a estar en este nuevo cluster que queda medio vacío debido a que los nodos violetas ya habían sido barridos y por lo tanto no queda nada que agregar.
Respecto de la alternativa a Sweep la clusterización es idéntica a la percepcion humana y es probable que estemos consiguiendo la mejor forma de organizar a los clientes y las variaciones en los costos respecto de la solución óptima se deban a la heurística que utilizamos para TSP.

12.3.3. Fortalezas y Debilidades

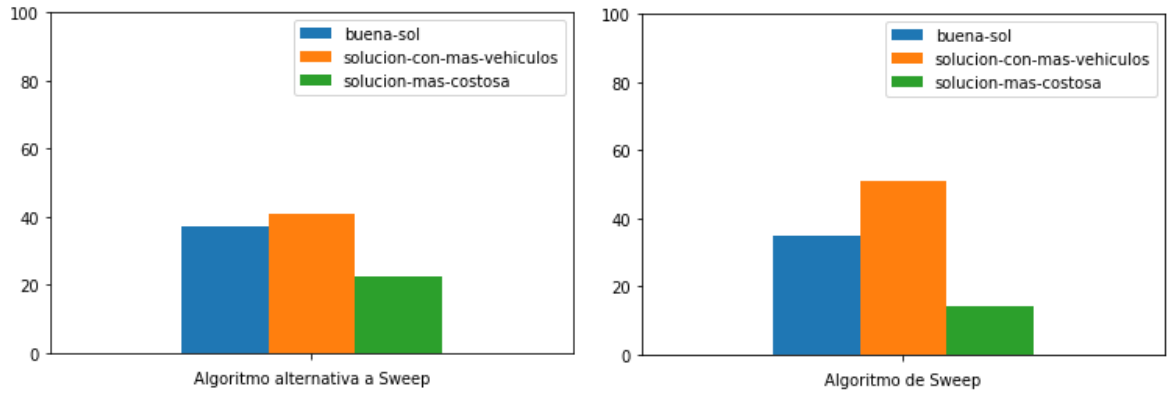
Otra de las observaciones que podemos hacer es que nuestros algoritmos se ven afectados en gran medida cuando los casos de test tienen intervalos de demandas de clientes cercanas a las capacidades del camión y sobre todo cuando hay muchos de estos puntos problemáticos (demandas iguales o muy cercanas a la capacidad). Sobre todo en la alternativa a Sweep los tiempos ejecución son mucho más elevados y las soluciones no suele consiguen rutear con la misma capacidad de vehículos ni un costo cercano. Y en relación con lo anterior mencionado si bien distribuciones de clientes en grupos similares a una clusterización ayudan al algoritmo aún ayuda más que no existan este tipo de puntos problemáticos y que la cantidad de clusters sea considerablemente más baja que n.

El hecho de que existan puntos con demandas tan cercanas al tope genera que el algoritmo necesite una cantidad de clusters cercana al valor de n y por otro lado que mientras se asigna los nodos a los clusters es altamente probable que las asignaciones necesiten sacar uno de los nodos del cluster para poder hacerse y esto genera más iteraciones para poder armar los clusters, mayor cantidad de recorridos de los cluster, mayor cantidad de clusters, etc. Por este motivo este es un caso que este algoritmo no podrá dar soluciones de calidad.
Para visibilizar esta problemática presentamos los siguientes casos: B-n35-k5 y E-n30-k3. Son instancias de casi la misma cantidad de nodos y clusters sin embargo el primero tiene una distribución de demandas de [0,69] con un tope de 100 y el tercero de [0,3100] y 4500 de tope. Con alternativa a Sweep conseguimos un resultado de 5 % de diferencia respecto del costo óptimo y con la misma cantidad de vehículos y en el segundo sólo conseguimos rutearlo con un vehículo más y con un 14 % de desvío. Con Sweep logramos resolver a ambos con la cantidad de vehículos óptima pero con un desvío mayor: 17 % y 9 % respectivamente.
Veamos por qué sólo Sweep consigue solución de buena calidad en E-n30-k3.



Lo interesante de este caso es que si bien anticipamos que casos con puntos problemáticos eran un problema para este tipo de técnicas encontramos un caso en donde la distribución de los puntos favorece a Sweep. El barrido que tiene utiliza esta técnica permite que dos grupos de nodos en esquinas opuestas sean del mismo cluster: esto nunca pasaría en la alternativa y en este caso en particular donde hay 3 zonas de mayor convergencia de clientes (y en particular cada zona tiene un cliente con una demanda muy cercana a la capacidad, es decir, un punto problemático) se pone más en evidencia sus falencias. En este caso lo que sucede es que los puntos problemáticos quedan fuera del cluster azul pero lo suficientemente cercanos como para que el cluster azul sólo tenga a ellos como alternativa a intercambiar por nodos propios pero esto nunca pasa porque el nodo más lejano del azul está incluso más cerca que esos nodos. Entonces el problema es que los nodos que quedan fuera del cluster azul no pueden clusterizarse en dos clusters y por lo tanto se necesita uno más. Como idea final podemos decir que cuando tenemos esta distribución de puntos Sweep es posible que se favorezca (recordemos el caso anterior B-n34-k5, de similar distribución de clientes, en donde Sweep no se beneficiaba por la ubicación de los puntos problemáticos). Pensemos que si esta instancia no tendría puntos de ese tipo posiblemente la alternativa habría logrado una solución de buena calidad o si la instancia tuviera puntos más uniformes en el plano aunque existieran no lo afectarían tanto.

En las siguientes gráficas podemos ver de forma más estructurada los puntos resaltados anteriormente. Lo que se representa son las soluciones de buena calidad (definadas previamente), las soluciones con más vehículos y las soluciones que están ruteadas con la cantidad de vehículos óptima pero con un costo asociado mayor al 25 %.

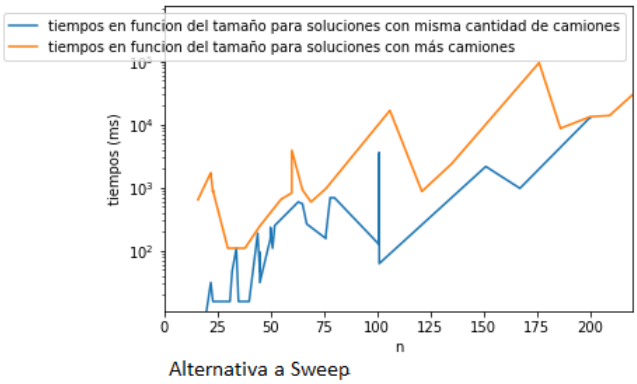


Ambos gráficos tienen un porcentaje similar de soluciones óptimas y las diferencias más notorias están con las 'no' soluciones óptimas. Es evidente que los refinamientos que no tiene Sweep provocan que tenga un mayor porcentaje de soluciones ruteadas con más vehículos, mientras que la alternativa que sí tiene este tipo de complementos parece evitar esto en algunas instancias pero con un costo asociado: posiblemente la forma en la que termina por agrupar los clientes no le permiten posteriormente crear rutas lo más óptimas posibles.

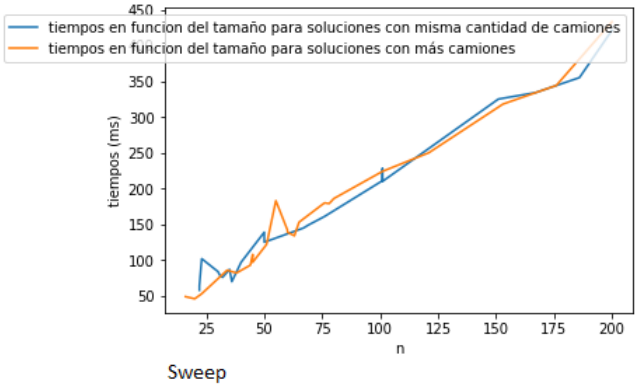
12.3.4. Tiempos de Ejecución

Hasta ahora estuvimos haciendo un análisis comparativo de calidad de soluciones de ambos algoritmos y las menciones de los tiempo de ejecución no se mencionaron o enfatizaron.

Durante el análisis los algoritmos supieron tener marcadas diferencias en sus tiempos: Sweep tiene un crecimiento en relación al tamaño y la alternativa también aunque condicionado con el tipo de instancia que esté evaluando. Al ser un algoritmo con refinamientos definidos por el usuario podría complejizarse (desde el punto de vista de los tiempos) tanto como uno quiera, debido a que sería como agrandar la fuerza bruta que analiza las posibles soluciones. Como introducción a todo esto podemos anticipar que los tiempos de Sweep son mejores y que la alternativa puede obtener soluciones de calidad en instancias considerablemente grandes siempre y cuando la instancia no contenga punto problemáticos o una combinación de muchos clientes con demandas muy chicas respecto de la capacidad (clusters muy grandes) y con una distribución uniforme (similar a un spray) a lo largo de todo el plano (provocando que dado un nodo está muy cerca de muchos clusters y por lo tanto se generan movimientos excesivos de asignaciones y re-asignaciones de nodos).



Alternativa a Sweep



Sweep

El gráfico anterior ilustra los conceptos que se vienen desarrollando. Lo que hicimos fue analizar los tiempos de ejecución en función del tamaño de entrada y separando en el tipo de solución conseguida. En naranja están los tiempos en función del tamaño de entrada cuando sólo se pudo conseguir soluciones con una cantidad de camiones mayor a la solución óptima. En azul están las que sí se pudo conseguir la óptima cantidad de camiones. Por razones de escala la alternativa a Sweep está en escala logarítmica.
El algoritmo de Sweep tiene una estructura que depende completamente del tamaño de entrada sus tiempos de cómputo, mientras que la alternativa es un método que depende no sólo de tamaño de la entrada sino también del tipo de instancia. Esto es algo muy visible en los gráficos: saltos pequeños hay en el de Sweep mientras que en el otro hay más saltos y teniendo en cuenta la escala de considerable mayor tamaño. Con esto podemos hablar acerca de lo malo que puede ser la alternativa a Sweep cuando no es una buena instancia: incluso en caso de poder llegar a la solución óptima los tiempos pueden ser mucho mayores a Sweep y estamos experimentando con instancias de menos de 200 clientes.

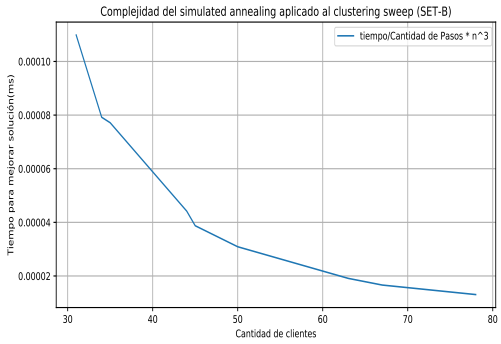
13. Experimentación: Metaheurística de Simulated Annealing

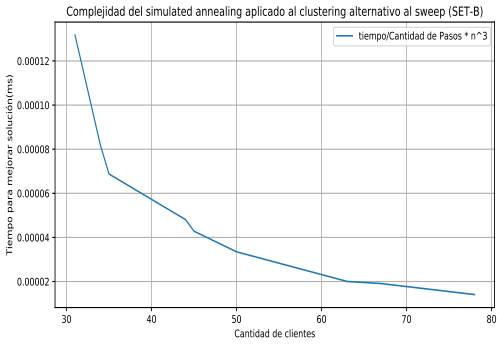
13.1. Verificación de las complejidades

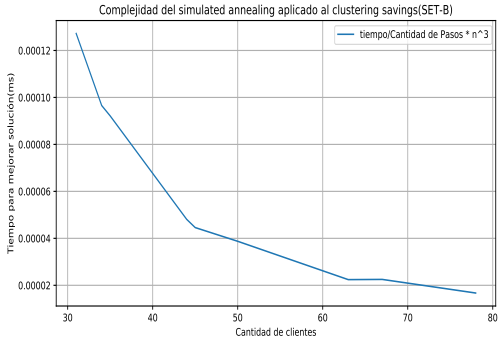
Para verificar que la complejidad calculada para el simulated annealing sea correcta tomamos los tiempos provenientes de aplicárselo a soluciones producidas por los varios algoritmos de clustering que fuimos desarrollando en el informe sobre el SET A y se dividió ese tiempo por la complejidad calculada. Los parámetros de simulated annealing tomados para este experimento fueron:

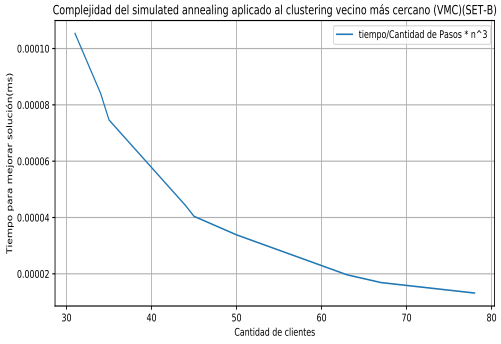
- Cantidad de pasos: 1000
- Temperatura Máxima: 20
- Coeficiente de enfriamiento: 0.99

Y los gráficos producidos fueron los siguientes:

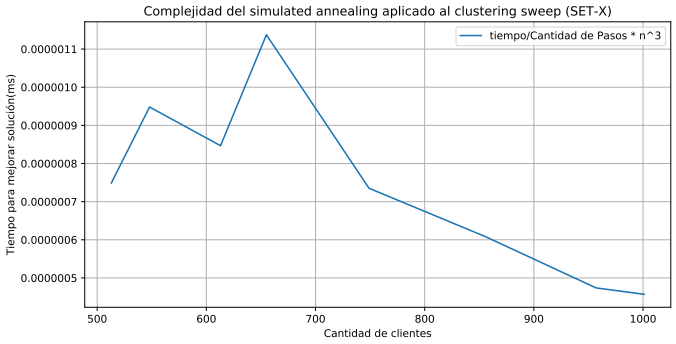








También se tomaron los tiempos de aplicar simulated annealing sobre el SET X con una gran cantidad de clientes, los parametros del simulated annealing fueron los mismos salvo la cantidad de pasos, que fue reducida a 500, el resultado fue el siguiente gráfico:



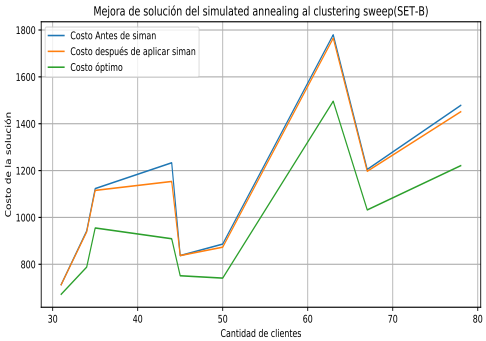
En todos los grafos hasta el momento, se puede ver que el resultado de dividir el tiempo tardado en realizar el simulated annealing por la complejidad calculada nos da funciones que tienden a 0 a medida que crece la cantidad de clientes, por lo que podemos creer que la complejidad es correcta.
Sin embargo el grafico no es una funcion constante, esto indicaría que la cota de complejidad calculada es precisa, por lo tanto podemos suponer que la complejidad del algoritmo es incluso mejor que lo que calculamos teoricamente.

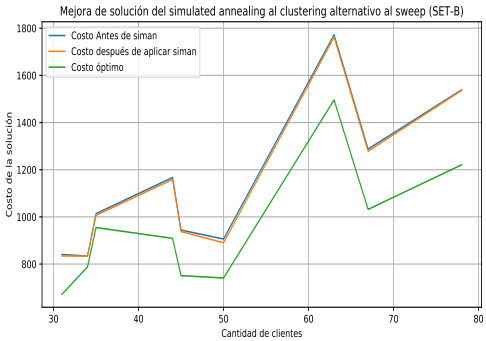
13.2. Mejora de solución

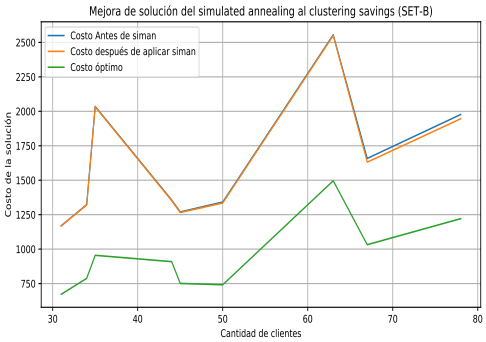
Mejorar una solución a través de simulated annealing puede provocar resultados muy variados dependiendo de varios factores, por ende nos interesaría saber si alguna de las heurísticas anteriores produce soluciones que sean mejor punto de partida para empezar a mejorarlas con simulated annealing, para poder ver esto corrimos las otras heurísticas para generar soluciones del SET B y las mejoramos con simulated annealing, tomando los siguientes parámetros:

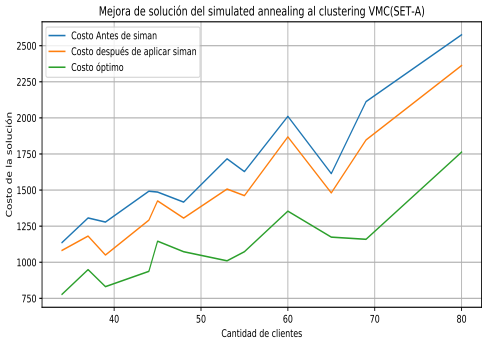
- Cantidad de pasos: 1000
- Temperatura Máxima: 20
- Coeficiente de enfriamiento: 0.99

La temperatura máxima es baja debido a que las diferencias de costos entre estados vecinos lo son también, si se eligiesen temperaturas más altas la solución mejoraría menos (o incluso no mejoraría). Los graficos generados fueron los siguientes:



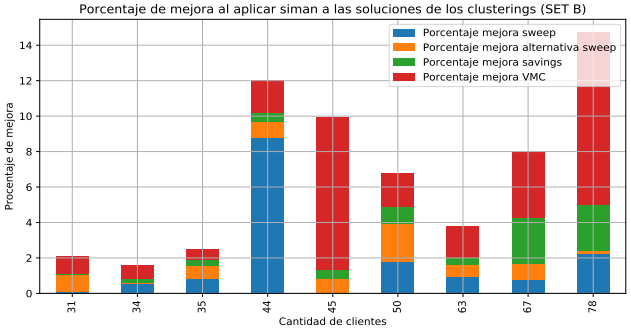






Podemos ver que de las heurísticas usadas, la heurística golosa del vecino más cercano forma caminos que mejoraron más en porcentaje que los de las demás heurísticas.
Esto puede ser por muchos factores, tal vez los parametros que utilizamos no son muy buenos o el vecindario utilizado tiende a arreglar los errores cometidos en VMC.
El algoritmo está abierto a ir probando con distintos parametros para así encontrar los que mejor se adaptan a tu algoritmo. Nosotros luego de un periodo de pruebas decidimos utilizar estos pero probablemente haya mejores opciones a utilizar.

En el siguiente gráfico podemos ver la comparación entre los porcentajes de mejora:



Se puede ver como la mejora es muy variable, hay casos donde no mejora nada, y otros donde mejora muchisimo, lo importante son estos segundos, la idea de la metaheuristica es la posibilidad de adaptarla a un caso particular para intentar mejorar la solución lo más posible y con un costo relativamente bajo de tiempo (complejidad).
A parte de esto la metaheuristica nos agrega la posibilidad de volver a ejecutar el algoritmo las veces que querramos consecutivamente, incluso variando los parametros, debido a que la solucion que devuelve nuestro algoritmo de Simulated Annealing es siempre igual o mejor que la solucion inicial y el algoritmo tiene una parte aleatoria por lo tanto puede que con el mismo parametro devuelva dos soluciones distintas.
Se puede suponer que si el vecindario te da la posibilidad de recorrer todas las soluciones entonces con el tiempo suficiente y la configuracion correcta, el algoritmo podría llegar a devolver una respuesta muy cercana al optimo, incluso empezando desde una respuesta que no sea muy buena.

14. Comparando performances

Comenzaremos analizando algunos casos específicos de datos, comparando todas las heurísticas con distintos métodos y gráficos para hacernos una idea de cual funciona mejor y para que circunstancias.

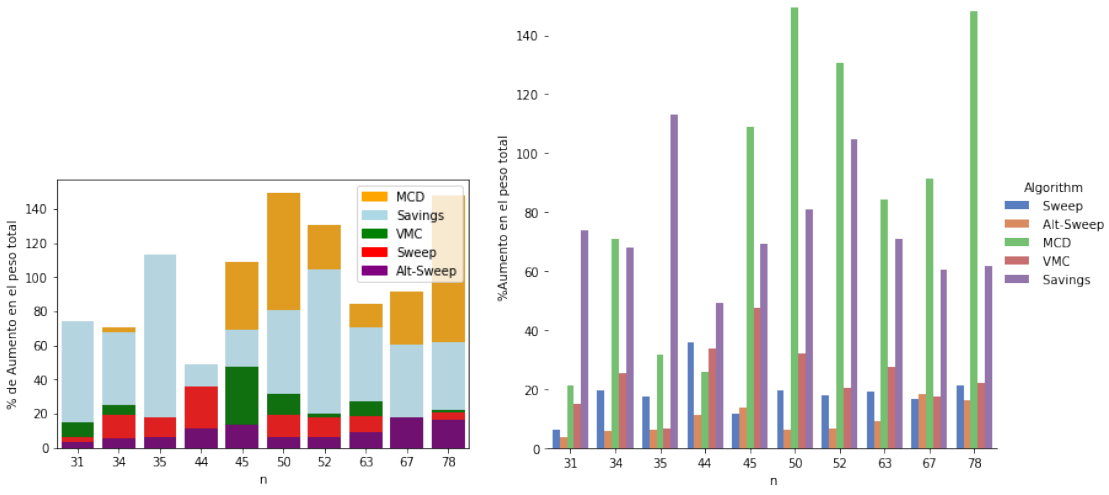
Notar que para las comparaciones de tiempo de ejecución se utilizará escala logarítmica, ya que las variaciones entre nuestros algoritmos serán grandes y de esta manera se podrán apreciar mejor.

14.1. Analisis en Set B (Augerat)

Datos de las instancias que nos interesarán para este experimento:

- Cantidad de clientes, lo cual llamaremos 'n' irá de 31 a 78.
- Capacidad de los camiones: en particular para este set de instancias la capacidad será fija y será 100.
- Las coordenadas son generadas de tal manera que se crean NC regiones con K(cantidad óptima de camiones) $\leq$ NC
- Las demandas son seleccionadas de una distribución uniforme U(1,30), además n/10 de esas demandas son multiplicadas por 3.

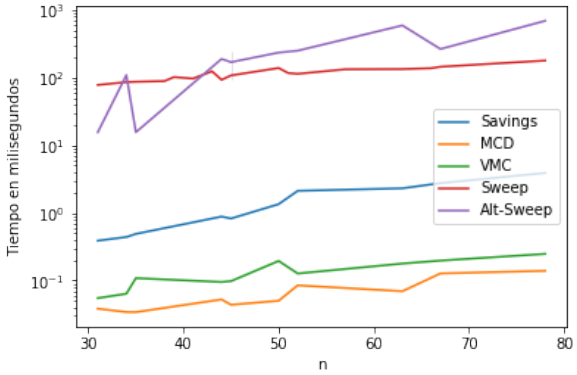
Vamos a tomar este set de datos y vamos a ejecutar unas pruebas para ver como reacciona cada algoritmo en distintas situaciones.
Comenzaremos analizando los siguientes gráficos que muestra el porcentaje de desviación al óptimo para distintos valores de N en el set de datos:



Se puede ver que en general los algoritmos de Savings y MCD son notablemente peores que los demás.
Otra observación que se puede hacer es que en este set de datos los algoritmos de clusterización parecen ser los mejores, lo que tiene sentido porque son los algoritmos más complejos que suelen dar mejores resultados porque sino no tendría sentido estudiarlos.

Por otro lado podemos destacar a la heuristica de VMC, es una heuristica golosa y con una idea muy básica, al mismo tiempo que es muy fácil de programar, y a pesar de esto logró muy buenos resultados, casi tán buenos como los algoritmos de clusterización. De echo en la práctica se podría considerar utilizar esta heuristica junto con unas pasadas del Simulated Annealing (que vimos que da buenos resultados en esta heuristica) y así obtener soluciones muy buenas en un tiempo razonable y con muy bajo costo de programación.

Ahora si tenemos los tiempos de ejecución en cuenta, podemos ver que es inversa proporcionalmente a la calidad de la solución:



Es decir, los algoritmos que suelen dar mejores soluciones respecto del óptimo, son los que más tardan en computar el ruteo. También se manifestará un cambio respecto de los órdenes: Savings tiene orden cúbico y Sweep tiene orden cuadrado y esto se verá reflejado de otra forma en la práctica. Esto es importante de aclarar ya que las complejidades teóricas suelen ser cotas groseras.

Otro aspecto importante a analizar, será la cantidad de buenas, promedio y malas soluciones que proporcionará cada algoritmo. Si consideramos las mejores soluciones para todas las instancias de este set, los porcentajes de los algoritmos que las dieron serán

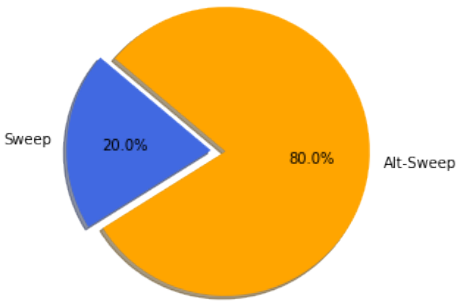


Figura 6: % de mejores soluciones por algoritmo

Los cuales como se dijo antes, serán los más costosos. Luego, teniendo en cuenta que son 5 heurísticas las que proporcionan las soluciones, si analizamos la cantidad de veces que un algoritmo brinda la "tercer mejor solución" a lo que nosotros llamaremos la solución promedio podemos ver que

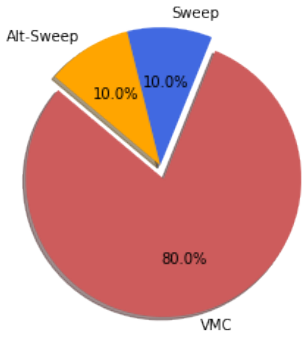


Figura 7: % de soluciones promedio por algoritmo

VMC será el algoritmo que más brinde estas soluciones, y esto será interesante, ya que compensaría muy bien con su complejidad teórica y su tiempo real de ejecución. Es decir, si se quisisera obtener una solución relativamente buena (en este caso las de VMC son a lo sumo 55 % peores) en poco tiempo, el algoritmo de VMC sería la mejor opción.

Como última instancia, si se analiza la cantidad de peores soluciones brindadas por estos algoritmos se ve que:

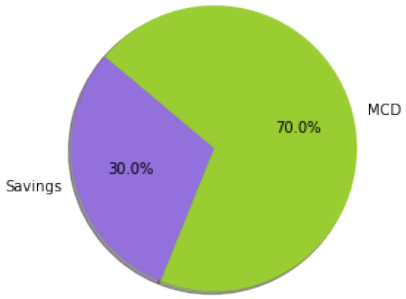


Figura 8: % de peores soluciones por algoritmo

los algoritmos de Savings y MCD son los que las brindan. Notar que la mayoría de estas serán brindadas por MCD, que al igual que antes, si analizamos su trade-off será de cierta manera "justo": este es el algoritmos que menos tarda para todas las instancias de este set, pero al mismo tiempo el que mayor cantidad de peores soluciones brinda.

14.2. Analisis en Set A (Augerat)

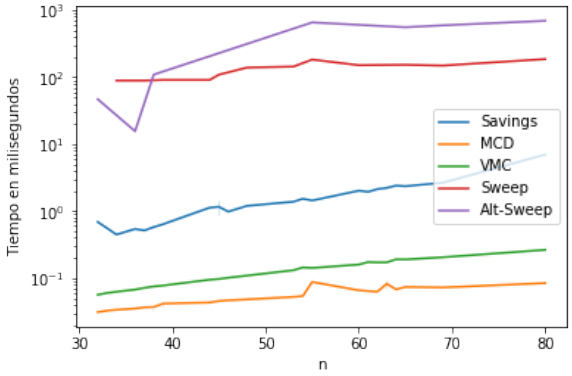
Datos de las instancias que nos interesarán para este experimento:

- Cantidad de clientes, lo cual llamaremos 'n' irá de 31 a 79.
- Capacidad de los camiones: en particular para este set de instancias la capacidad será fija y será 100.
- Las coordenadas de los clientes son generadas aleatoriamente dentro de un cuadrado de lado = 100.

- La demanda promedio será de 15. Las demandas son seleccionadas de una distribución uniforme U(1,30), además n/10 de esas demandas son multiplicadas por 3.

Para poder seguir con el análisis de performance de nuestros algoritmos se tomará este set de Instancias.

Si se observan los tiempos de ejecución de todos los algoritmos se obtiene el siguiente gráfico:



De los cuales podemos seguir viendo la misma tendencia que se venía viendo.

Ahora en cuanto a la calidad de la solución respecto del peso, si lo analizamos en las muestras más grandes para este set, se puede ver que:

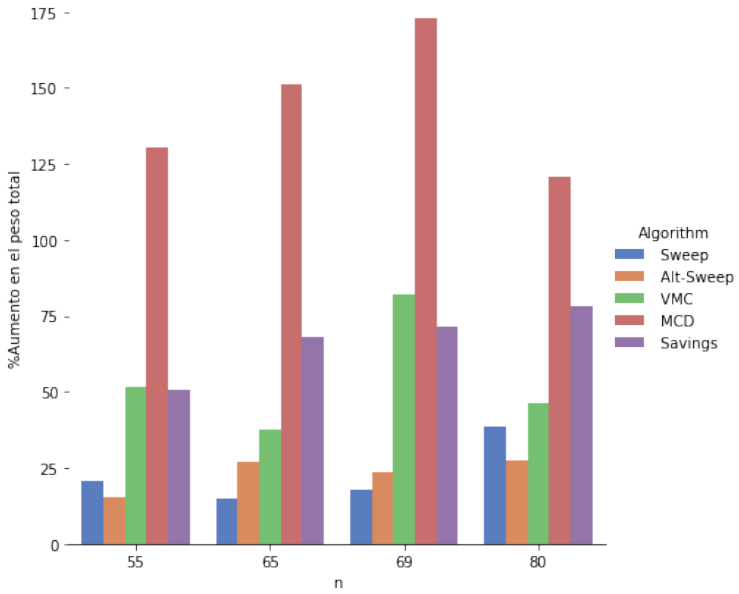


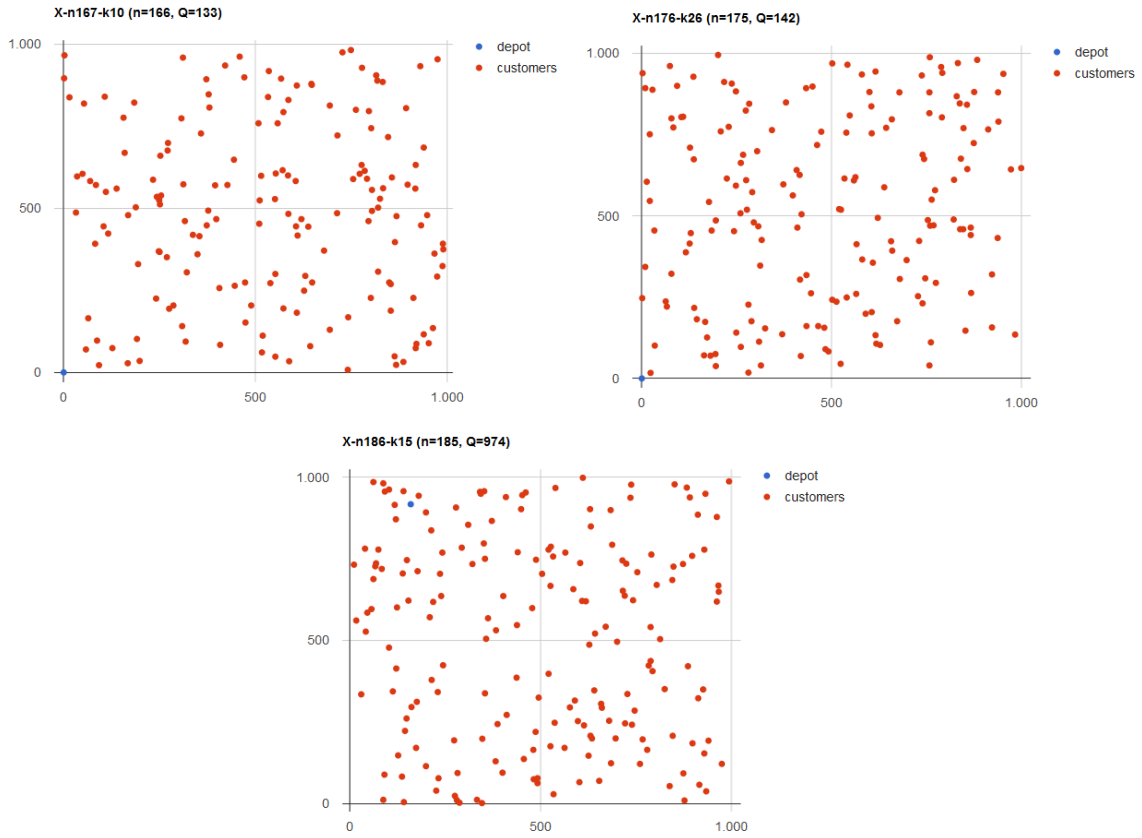
Figura 9: % Aumento respecto del Óptimo

El algoritmo de Savings mejorará relativamente en cuanto a la solución brindada, tal es así, que tendrá varias soluciones que serán incluso mejores que las de VMC, el cual como se citó antes, es uno de los algoritmos más balanceados en tiempo-costo Solución.

Otra tendencia clara que se repite respecto del anterior experimento será la incidencia de MCD en la cantidad de peores soluciones y la buena calidad de ambos algoritmos Sweep y Alt-Sweep en donde "buena" aquí será no valer más que un 35 % del óptimo.

14.3. Probando con instancias más grandes

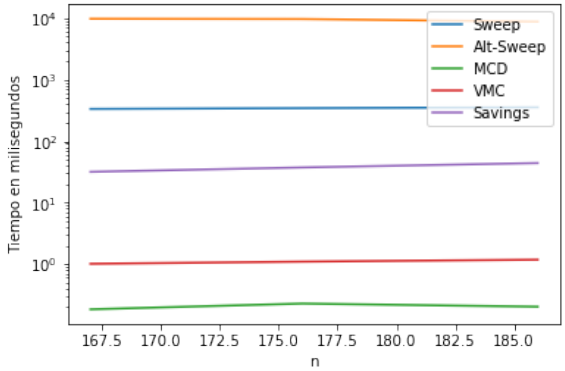
Teniendo en cuenta que tenemos algoritmos que pueden llegar a tardar mucho con instancias más grandes, se analizará la performance de estos algoritmos para 3 instancias grandes del set Uchoa, las cuales serán:



En particular tenemos que:

- Capacidad de los camiones: esta será Q.
- Las demandas irán de 1 a 10 en X-n167 y de 1 a 100 en X-n176, X-n186.
- Las posiciones de los clientes son generadas aleatoriamente.
- La posición del depósito es generada aleatoriamente

Con eso en mente, si se analizan los tiempos de ejecución se tiene que



Las heurísticas golosas tienen un tiempo de ejecución bastante menor inclusive para estas instancias más grandes, después le seguirá Sweep y por último Alt-Sweep el cual en estos casos se diferenciará mucho de los demás, y esto se deberá en gran parte a las optimizaciones que realiza el mismo, lo cual para instancias más densas se puede ver que tendrá un fuerte impacto en los tiempos de ejecución.

En cuanto a la calidad de la solución se puede observar

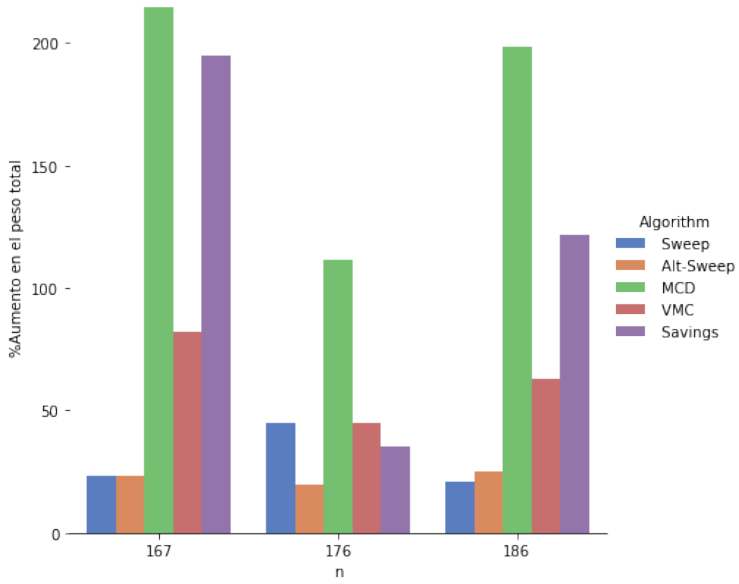


Figura 10: % Aumento respecto del Óptimo

que para instancias grandes se sigue manteniendo la tendencia antes mencionada: Sweep y Alt-Sweep serán los mejores y MCD el peor, con Savings y VMC con variaciones interesantes disputando el tercer puesto en cuanto a mejores soluciones obtenidas.

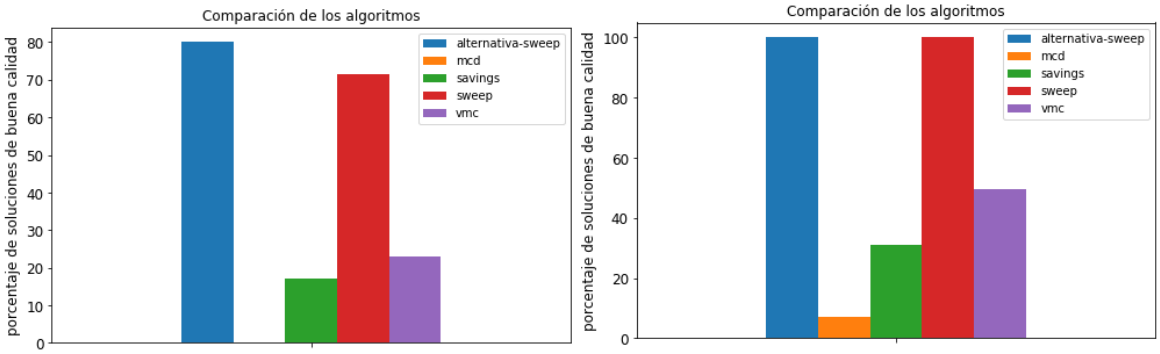
14.3.1. Conclusión parcial

Como conclusión de estos experimentos no se puede dejar de mencionar lo siguiente:

- Las tendencias en cuanto a tiempo de ejecución están bien marcadas sin variaciones entre instancias y las mismas respetarán el orden de las complejidades teóricas salvo en el caso de Savings y Sweep por lo que si se ordernaran de mayor tiempo de ejecución a menor tendriamos: Alt-Sweep, Sweep, Savings, VMC, MCD.
- En cuanto al costo en la solución que implica ser más rápido o lento para calcular el ruteo, se puede ver que VMC será el que menos sufre una penalización por esta relación: se mantienen tiempos de cómputo muy bajos y se obtienen soluciones que en general no superan muy ampliamente el óptimo. En cambio con los otros algoritmos puede verse que dar una mejor solución implica mayor tiempo de ejecución y lo mismo con los que tardan poco como Savings y MCD pero con soluciones relativamente muy malas por lo que diremos que estos algoritmos mantienen comportamientos más extremos respecto del trade-off previamente mencionado.
- Las soluciones de MCD serán generalmente las más costosas de todas y las de Sweep y Alt-Sweep serán de las mejores de los 5 algoritmos.

14.4. Resultados generales

Las comparaciones entre los algoritmos de cluster-first establecieron estándares de calidad para solución relativamente altos respecto de las conseguidas por otras heurísticas. Sin embargo para un análisis más general entre los diferentes algoritmos será necesario por lo menos al inicio reducir los criterios de calidad.



En el gráfico anterior se puede comparar las soluciones brindadas por los algoritmos para un conjunto de instancias de entre 20 y 250 clientes los cuales serán tomados de los set de instancias: A, B, P de Augerat y X de Uchoa con una calidad de solución definida. En el primer gráfico se considera una buena solución cuando el costo obtenido no supera en 30 % al de la solución óptima. Un detalle a considerar es que para el set de instancias ninguna pudo resolver en esta configuración el algoritmo de más cercano al depósito. En la segunda se considera una buena solución a las que no superen en 50 % al costo de la óptima. En este caso se puede apreciar que el algoritmo MCD aparece en escena aunque muy distante respecto de los demás. Como se mencionó reiteradas veces previamente, se puede observar fuertemente en este gráfico que mientras más alta sea la complejidad teórica, más alto será el porcentaje de soluciones de "buena calidad".

15. Conclusiones finales

En este trabajo tuvimos la oportunidad de experimentar con algoritmos heuristicos en un problema que muchas veces surge en la vida real, pudimos notar la importancia de estos algoritmos ya que en primera instancia parece mala idea tener algoritmos que no den una solucion correcta en la mayoría de los casos pero las heuristicas nos dan una flexibilidad a la hora de elegir cual vamos a utilizar y con que parametros, que las vuelve muy importantes a la hora de solucionar problemas en la vida real.
Las heuristicas nos ofrecen la posibilidad de adaptarnos, hacernos las preguntas "¿Cual es la máxima complejidad que nos podemos permitir?" o "¿Que tán buena tiene que ser la solución?". Esto es algo que nunca nos hubiesemos imaginado debido a que en general uno como programador recibe un problema de su jefe y busca una forma la mejor manera de resolverlo, pero de esta manera se le puede dar más flexibilidad a la respuesta y resolver el problema en función a las condiciones del trabajo.

Pudimos probar varias heuristicas conocidas y tambien desarrollar las nuestras, es notorio la diferencia de complejidad de código entre las distintas heuristicas, tambien puede ser algo a consideración, hay algunas que son muy simples de pensar y de programar pero en general llegan a soluciones no tan buenas como los algoritmos más complejos, de esta manera tambien podemos decidir cual heuristica utilizar si es que tenemos un tiempo limitado para el desarrollo del algoritmo.
Algo que nos llamó mucho la atención es que en general en las distintas heuristicas se mantiene como una relación de compensación entre las ventajas y las desventajas de cada una, por ejemplo en general si una heuristica da resultados muy cercanos al óptimo, podemos suponer automáticamente que tiene alguna desventaja como una complejidad muy alta o una complejidad de código muy alta, esto se debe a que cada una de estas heuristicas tiene sus ventajas y desventajas, porque si tuvieramos una heuristica que sea mejor que las demás en todo concepto, utilizaríamos esta siempre y no tendría siquiera sentido estudiar las demás.

La parte de clusterización fue muy interesante ya que es una idea que no es tán simple, y no se le ocurriría a uno como una primer idea para resolver este problema, sin embargo llegamos a resultados muy precisos y con una complejidad razonablemente baja. Pudimos reutilizar parte de lo aprendido en el trabajo práctico anterior para hacer frente a un ejercicio más complejo y así obtener muy buenos resultados.
En cambio la parte de heuristicas golosas nos dejo abierta la posibilidad de pensar como lo resolveríamos nosotros si tuvieramos bajo presupuesto para desarrollar el algoritmo, probamos con la heuristica del Más Cercano al Depósito que intuitivamente es bastante mala, para demostrar que las heuristicas golosas, si no están bien pensadas, pueden llegar a resultados pésimos. Pero nos llevamos la sorpresa con la heuristica del Vecino Más Cercano, ya que dio resultados bastante buenos en una grán cantidad de casos, y aunque hubo casos donde fueron bastante malos los resultados, para ser una idea tán simple como es " ir siempre al que esté más cerca mientras pueda" , dio resultados razonablemente buenos.

Referencias

[1] Some important Heuristics for the TSP.
https://ocw.mit.edu/courses/civil-and-environmental-engineering/1-203j-logistical-and-transportation-planning-methods-fall-2006/lecture-notes/lec16.pdf

[2] The Savings Algorithm
http://neo.lcc.uma.es/vrp/solution-methods/heuristics/savings-algorithms/