1. Introducción

1.1. CVRP

Uno de los problemas más recurrentes en optimización combinatoria es el **problema de ruteo de vehículos**, conocido por su sigla inglesa **VRP**. Los problemas de esta índole, si bien dispares, suelen consistir en interrogantes logísticos donde se desea rutear vehículos con el propósito de entregar mercadería a una serie de puntos en el espacio, normalmente intentando minimizar la distancia recorrida. Entre las variantes de este problema existen casos donde la flota de vehículos con la que se cuenta para realizar el ruteo es homogenea, mientras que en otros no. A veces existe un depósito central desde donde se debe comenzar y a donde deben terminar las rutas. No siempre es esto así. Los vehículos de la flota pueden tener restricciones adicionales como una capacidad máxima de carga o una máxima distancia operativa.

De entre todo el abanico de problemas presentado, este trabajo focalizará el **problema de ruteo de vehículos con capacidad** (*CVRP*). El mismo consiste en lo siguiente: se tiene una serie de clientes distribuídos en un plano, cuya posición se determina por sus coordenadas X e Y. Cada cliente ha realizado un pedido de una determinada cantidad de mercancía (*demanda*) que habremos de suplir con nuestra flota de camiones, a priori infinita. Cada vehículo de nuestra flota, sin embargo, contará con una capacidad de carga máxima. Un punto extra, el *depósito* existe como comienzo y final del recorrido de nuestros camiones. Se desea minimizar la distancia recorrida en total, siempre asegurando que las demandas de los clientes sean cubiertas y que, además, cada cliente sea visitado una única vez.

Si bien es sabido que este problema pertenece a la categoría de problemas *NP*-hard, en el presente informe no buscaremos dar con la solución óptima sino con una heurística suficientemente eficiente como para obtener resultados de buena calidad tanto para instancias individuales como para instancias desconocidas.

TODO: Dar ejemplos de CVRP y sus soluciones (casos de test, no cosas de la vida cotidiana eh). Poner algún caso piola de la página con grafiquitos de la página y eso. Mostrar posibles caminos óptimos y subóptimos y algunos mejores que otros pero peores que el óptimo.

1.2. CVRP en la vida cotidiana

Si bien, contrario a lo que el computador promedio pueda creer, la gente común no suele aplicar optimización combinatoria por mano propia en su vida cotidiana, existen problemas *cotidianos* - o al menos lindantes a la cotidianeidad - que pueden modelarse como un CVRP y, por tanto, su solución puede ser aproximada con un análisis como el que llevaremos a cabo en las próximas secciones.

El problema más trivial es el de, como la sigla lo indica, ruteo de vehículos con capacidad. No es

un hecho esquivo a los ojos de empresas dedicadas al comercio y envío de mercadería que un transporte de producto a un punto alejado se traduce un mayor coste de infraestructura vehicular. No solo aumenta el gasto en combustible: un viaje longevo requerirá más tiempo, más paga para el conductor, mayor manútención del vehículo y presta oportunidad a que sucedan un sinfín de siniestros inesperados cuya probabilidad se incrementa con el tiempo en campo. Una ruta óptima con distancia minimal representaría un ahorro signicativo en las arcas de la entidad y, por tanto, mayor márgen de ganancia.

Un paquete turístico que consista en una única visita a una serie de lugares predefinidos también se vería beneficiado - en ganancia para el vendedor de paquetes como para el turista de cuyo bolsillo saldrá el viaje - por una heurística poderosa sobre el CVRP. Las ubicaciones podrían recorrerse en orden ahorrando tiempo y distancia, dejando libre tiempo al transeunte para disfrutar de su estadía y ahorrándole dinero.

Las posibilidades son infinitas: una tarde perdida de trámite en trámite de una punta a otra de la ciudad podría transformarse en una diligencia de unas pocas horas aplicando correctamente una heurística. Instalaciones como una cañería distribuidora de gas, una cloaca o una tubería de agua podrían ahorrar material, tiempo y esfuerzo implementativo si su trayecto se diseñase modelado como un CVRP. Y estas son tan solo algunas de las aplicaciones, en un caso hipotético. Empresas como **Rappi**, **Glovo**, **Gofer**, **MercadoEnvíos**, **Uber**, ruteadores de GPS (*"a continuación, gire a la derecha*), etc., probablemente lleven a cabo análisis como los que plantearemos a continuación para disminuir sus costos operacionales. Se trata de un problema no exclusivo al mundo de la abstracción científica, sino probablemente surgido - y tanto así aplicado - a partir de una necesidad palpable y material.

1.3. Problemas a resolver

Todos los experimentos presentados en este informe serán realizados sobre dos grupos de muestras/sets de problemas CVRP: instancias obtenidas de `http://vrp.atd-lab.inf.puc-rio.br/index.php/en`, denominadas en este documento como *"instancias de la cátedra"*, que comprenden una serie de casos de test, en su mayoría con soluciones óptimas ya encontradas (y, de lo contrario, con cotas mínimas conocidas), de diferentes características (distribución aleatoria, puntos que forman clusters, etc); y casos de test aleatorios generados por nosotros, de tamaño variable (hemos diseñado instancias desde tres hasta cien puntos) sobre los cuales testearemos nuestras heurísticas en busca de una minimización de coste sobre un posible "caso desconocido".

Cabe mencionar que nos parece importante el análisis de un caso aleatorio desconocido promedio ya que, muy probablemente, al analizar un caso en el mundo real no pueda saberse con exactitud cómo será siempre la muestra con la que se desee trabajar. Una heurística que se presente como un posible mínimo o "buena cota baja" para soluciones de problemas de este estilo permitiría, si se desea ir más allá de usar esta solución como tal, buscar mejores soluciones en base a un parámetro conocido, es decir, soluciones cuyo coste sea menor al descubierto por dicha heurística.

Los puntos de los casos generados se encuentran distribuidos aleatoriamente y hemos generado cien instancias para cada cantidad de puntos que se desee evaluar, a fin de poder obtener una suerte de "valor estimativo promedio" de entre una muestra potencialmente numerosa. Es posible que,

analizando una mayor cantidad de muestras, o regenerando las muestras, los valores obtenidos por nuestros experimentos cambien levemente. No se espera, sin embargo, un fuerte movimiento porcentual de la eficiencia obtenida por cada heurística llegado a darse este caso.

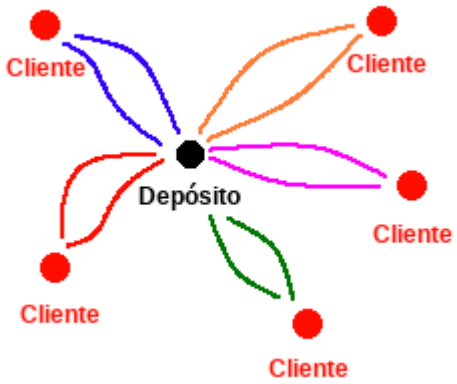
1.4. Objetivos

Partiendo de lo dicho en las secciones anteriores, puntualmente el **apartado 1.3**, en este análisis buscamos encontrar heurísticas *buenas* para el CVRP, es decir, que arrojen resultados cuyos costes puedan ser considerados como buenos. Fiel a la consigna, analizaremos cinco heurísticas de metodología dispar, de las cuales dos son de nuestra creación. Las compararemos entre sí, evaluaremos sus parámetros de entrada en busca de óptimos e intentaremos encontrar, tanto para las instancias de la cátedra - en conjunto e individualmente - como para nuestros casos aleatorios, heurísticas eficientes que permitan vislumbrar posibles soluciones válidas al problema presentado.

Nos parece importante recalcar que no solo nos interesa llevar a cabo un análisis sobre las instancias conocidas cuyos valores óptimos se tienen presentes, sino también sobre instancias generadas al azar de las cuales no se conoce nada. Creemos que un caso real podría tener condiciones similares (no saber nada) y, por lo tanto, es relevante el saber de una solución, aunque sea parcial, que pueda correr eficientemente sobre dichas situaciones. Claro es que no esperamos que una misma heurística sea la mejor para cada caso evaluado, pero sí esperamos encontrar una que, en promedio, se comporte mejor que todas las demás.

1.5. Caso canónico

Para facilitar la comunicación de nuestras ideas y, por lo tanto, la lectura de este documento, hemos definido el *caso canónico*. El caso canónico no es más que la solución trivial a un problema de CVRP, que consiste en tantas rutas como clientes haya, enviando un camión a cada cliente, que entrega la mercadería y regresa al depósito.



Ejemplo de caso canónico

La importancia del caso canónico radica en que es la peor solución posible a un problema de CVRP como los que estamos tratando y es además evidente. Nos intereza porque podemos utilizarla como parámetro para analizar la eficacia de una heurística sobre un caso del que no se sabe nada: podremos desconocer el óptimo, pero podemos calcular y afirmar que nuestra heurística es *n%* mejor que el caso canónico. Esto da entonces lugar a que una heurística absolutamente inútil sea *0%* mejor que el caso canónico (es decir que tardó el *100%* de lo que tardó el caso canónico), mientras que una muy buena pueda ahorrar un *99%* de la distancia del caso canónico (es decir, ejecutarse en el *1%* del tiempo que tarda la solución trivial).

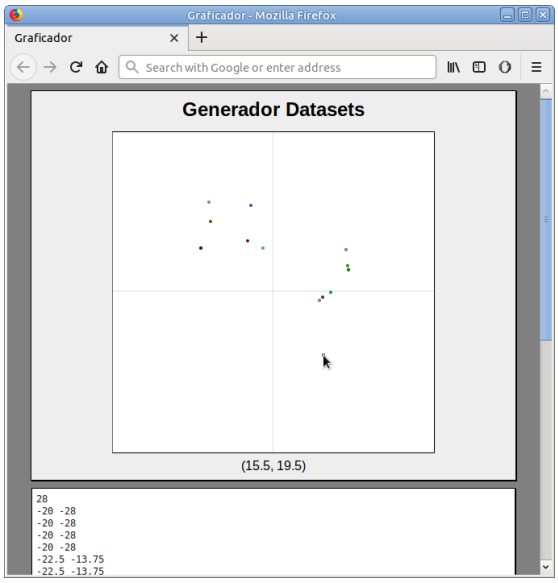
2. Herramientas y equipo

2.1. Equipo utilizado

Las mediciones presentadas en los experimentos que se presentarán en las próximas secciones fueron llevadas a cabo en un equipo que consta de las siguientes prestaciones: procesador AMD FX-6300 a 3.6GHz, 8.14 GB de memoria RAM y sistema operativo OpenSUSE Leap 15.0.

2.2. Generador de clusters

Los datasets que utilizaremos para testear y analizar los heurísticas basdas en clustering (sobre las cuales hablaremos en la **sección 3.3**) fueron generadas utilizando un generador propio de datasets, diseñado para poder diseñar instancias a mano. El mismo se encuentra en el directorio `Generador_Datasets_Cluster`. Para ejecutarlo se debe abrir con un navegador web el archivo `generador.html`. Para usarlo no hace falta más que hacer click en el lugar donde se quiera colocar un punto. La coordenada del punto será añadida al área de texto debajo del gráfico, que luego podrá ser copiada a un archivo de texto, parseada, reestructurada conforme al estándar especificado por la TSPLib y provista al ejecutable del trabajo práctico por `stdin`.



Generador de datasets propio

2.3. Generador de instancias

Para poder evaluar nuestros algoritmos más allá de los casos de test provistos por la cátedra, en lo que llamamos *instancias desconocidas*, es decir, instancias de las cuales no se sabe nada (y por tanto, confiamos en que representan un caso genérico en el que se desee resolver un problema aplicando lógica de CVRP) hemos diseñado un generador de instancias aleatorias. Dados tres valores n, m y c, el generador produce una serie de instancias de $[3, n]$ puntos, m por cada n, con capacidad c para los camiones. En nuestros experimentos hemos decidido medir utilizando $n = 200$, $m = 1000$ y c aleatorio entre 70 y 150. Los puntos de las instancias aleatorias, como su nombre indica, se encuentran distribuidos de forma aleatoria sobre un espacio bidimencional y se adaptan al estándar TSPLib, con lo cual pueden ser alimentadas a nuestro trabajo práctico directamente por `stdin`.

3. Soluciones Aproximadas

3.a. Introducción

Nuestro objetivo en esta sección será medir la eficacia de las heurísticas que implementamos. Para esto compararemos los resultados de aplicar cada una de ellas sobre los casos de test ofrecidos por la cátedra (de los cuales una solución óptima o, cuando menos, una cota inferior es conocida) y sobre casos aleatorios propios, por los motivos explicitados en la **sección 1.4**. Los resultados obtenidos sobre casos de óptimo conocido serán dados en porcentaje de diferencia con el óptimo, es decir, qué tanto más grandes que el óptimo son. Un resultado que sea 50% más grande que el óptimo será el óptimo y medio, mientras que uno que sea 200% será el doble.

Los resultados obtenidos sobre casos aleatorios, por el contrario, al no disponer de una cota inferior se miden en porcentaje de distancia con el caso canónico, es decir, qué porcentaje del caso canónico representan. Esto puede darse así ya que la solución nunca será peor que el caso canónico y, por tanto, nunca se superará el 100%. Esto ya ha sido visto en detalle en la **sección 1.5**.

Las mediciones no han sido realizadas un número estándar de veces, sino que su repetición ha sido adaptada por algoritmo. Si bien la base elegida ha sido la mencionada en la **sección 2.3**, es decir, mil mediciones por conjunto de instancias, este valor ha variado ulteriormente según la heurística tratada. Se ha intentado decidir por un valor que minimice la presencia de ruido en nuestros resultados, pero se ha tenido en consideración el tiempo requerido para realizar dichos experimentos.

3.b. tsplib.h

Para cargar los archivos que contienen los casos de test sobre los que hemos decidido experimentar, hemos desarrollado el archivo `tsplib.h`, ubicado en la carpeta `Codigo/tsplib` del repositorio de este informe. El mismo provee de funciones de carga de archivos en el formato especificado por el documento *TSPLIB 95*, por Gerhard Reinelt, y una interfaz estática que permite acceder a la información contenida en estos de forma sencilla. Si se desea conocer más al respecto, el lector está invitado a echarle un vistazo al código.

3.c. S_CVRP

Para el modelado del problema decidimos crear una clase la cual nos va a servir como interfaz para resolverlo. La misma provee toda la información necesaria para la implementación de nuestros algoritmos. S_CVRP hace uso de tsplib.h para recibir todos los datos de una instancia en particular para luego proceder a

- Reconocer el depósito y guardarlo en una variable.
- Guardar la capacidad de los camiones en otra variable.

- Mover todos los puntos para que el depósito quede centrado en el origen (facilita la implementación).
- Guardar cada nodo en un vector de nodos asignandole su id, coordenadas y demanda correspondiente.
- Generar la matriz de adyacencia en base a las distancias entre puntos.

Nota: Además de S_CVRP creamos un struct S_Nodo para representar los vertices del grafo. El mismo contiene el id del nodo, su demanda y sus coordenadas en el plano.

Para acceder a estos datos, proveemos una serie de métodos básicos para la resolución del problema en cuestión. Los mismos se pueden ver en el archivo estructuras.h.

3.d. p_solucion

```
TODO: Explicar como está compuesta una solución al problema y como la vamos a representar (conjunto de conjuntos de clientes).
```

3.1. Heurística basada en *Savings*: *Heurística del Próximo Mínimo*

3.1.1. El algoritmo

La heurística del próximo mínimo surge del precepto de que, a partir la unión de caminos cortos, podemos formar caminos cortos. Si bien este argumento es meramente hipotético, hemos tomado del paper *Scheduling of Vehicles from a Central Depot to a Number of Delivery Points* , de G. Clarke y J. W. Wright (*savings*) la idea de formar caminos minimales a partir de la unión de caminos preexistentes como la base para diseñar esta heurística. Esta consiste en tres pasos: *armado de caminos*, *corte de caminos* y *unión de caminos*.

```
1  nodo: cliente o depósito
2  heuristicaDelProximoMinimo:
3      rutas ← armarCaminos(nodos)
4      rutas ← cortarCaminos(rutas)
5      rutas ← unirCaminos(rutas)
```

El paso de *armado de caminos* consiste en, desde un punto *A* -que inicialmente es el depósito-, evaluar cuál es el próximo nodo al que podramos llegar en la menor distancia, sin tener en cuenta (aún) las demandas de dicho nodo ni la capacidad de mi vehículo. Se elige este nodo como destino y se mueve el vehículo a esta locación. Luego el proceso se repite nuevamente y se elige un nuevo objetivo. Si nuestro destino fuera a ser el depósito, se considera como que este camión ha vuelto a destino y el proceso se repite nuevamente con otro vehículo. Por precepto del problema, una vez que se ha visitado un nodo no se lo considera dentro de los posibles objetivos, a menos que dicho nodo sea el depósito, el cual se puede visitar múltiples veces.

Por la forma en la que la asignación de camiones funciona, al haber recorrido todos los nodos, encontrándonos parados en el último nodo *B* (cosa que siempre ocurre ya que, en el caso de nunca haber caminos más chicos que ir desde el depósito hasta un cliente y volver -es decir, de no poder combinar caminos- caemos en el caso canónico), siempre existirá todavía como posibilidad el camino de vuelta al depósito. De esta forma, el algoritmo de armado de caminos terminará cuando nos encontremos en el depósito y no haya más nodos por visitar. Hecho esto, tendremos tantas listas de destinos como camiones hayamos usado, y estas listas serán la unión de posibles rutas del caso canónico.

```
1  armarCaminos(nodos):
2      rutas ← ∅
3      rutaActual ← ∅
4      nodoActual ← obtenerDepósito(nodos)
5      mientras quedanClientesSinVisitar(nodos):
6          objetivo ← elegiNodoMásCercanoA(nodoActual)
7          rutaActual ← rutaActual ∪ {objetivo}
8          nodoActual ← objetivo
9          si esDepósito(nodoActual):
10             rutas ← rutas ∪ {rutaActual}
11             rutaActual ← ∅
12     devolver rutas
```

No se ha tenido en cuenta hasta este punto, sin embargo, la demanda de los clientes en contraste a la capacidad de nuestros camiones. Por tanto, el lector agudo habrá observado que, para un camión cualquiera, es posible que su ruta a recorrer supere su capacidad. Es por esto que se entra a la segunda parte del algoritmo: *el corte de caminos*.

El corte de caminos consiste en analizar cada camino individualmente, revisando que la demanda de todos los nodos examinados hasta el momento sumen una demanda menor o igual a la capacidad de nuestros vehículos. De hacerlo, examinamos el siguiente nodo y así sucesivamente, agregando su demanda a la suma. De no hacerlo (y teniendo en cuenta que la demanda de cualquier cliente es menor o igual a la capacidad de un caminón), cortamos el camino antes de este nodo, uniendo ambos extremos del corte al depósito, de forma que acabamos con dos caminos que empiezan y terminan en nuestro depósito. Al terminar de cortar todos los caminos devueltos por el paso anterior, acabaremos con un conjunto de caminos -un camino por camión, con lo cual la necesidad de camiones habrá muy probablemente aumentado con este paso- tales que la demanda de cada uno de sus clientes será suplible por un solo camión.

```
1  cortarCaminos(rutas):
2      paraCada rutaActual en rutas:
3          costeTotal ← 0
4          paraCada nodo en rutaActual:
5              si costeTotal + demanda(nodo) < capacidadCamiones:
6                  costeTotal ← costeTotal + demanda(nodo)
7              sino:
8                  ruta1 ← rutaActual desde inicio hasta anterior a nodo
9                  ruta2 ← rutaActual desde nodo hasta fin
10                 rutas ← rutas - {rutaActual}
11                 unirADepósito(ruta1)
12                 unirADepósito(ruta2)
13                 rutas ← rutas ∪ {ruta1} ∪ {ruta2}
14                 break
15     devolver rutas
```

Una vez que la subrutina presentada sobre este párrafo termina de ejecutar, en `rutas` tenemos un conjunto construídas siguiendo las instrucciones de la primera parte del algoritmo, pero cortadas de forma tal que cada ruta presenta una demanda menor a la de un camión. La heurística podría cortarse acá: en las primeras etapas de experimentación dio resultados positivos de mejora frente al caso canónico. Sin embargo, esta etapa de la heurística presenta un problema. Supóngase que en total contamos con cuatro nodos, para los cuales `armarCaminos` ha devuelto dos rutas: {*depósito*, *nodo1*, *nodo2*, *depósito*}, {*depósito*, *nodo3*, *nodo4*, *depósito*}. Esto puede haber ocurrido porque la distancia entre el depósito y el *nodo1* era menor a la del depósito con el resto de los nodos, la distancia del *nodo1* al *nodo2* era menor que la distancia entre *nodo1* y el resto de los nodos, incluído el depósito, pero la distancia entre el *nodo2* y el resto de los clientes era mayor a la del nodo con el depósito. Si suponemos que cada camino tiene una demanda mayor a la suplible con un camión, es fácil ver que la separación de los caminos acabaría nuevamente en el caso canónico. Sin embargo -y aunque esto es trivial para casos chicos como el presentado, es muy significativo en casos mayores o de clientes muy alejados al depósito- supongamos que la demanda del *nodo2* sumada a la del *nodo3* es menor a la de un camión. Podríamos analizar entonces qué pasaría si, aunque no estuvieran inicialmente cerca, uniéramos el camino *{depósito, nodo2, depósito}* con el camino *{depósito, nodo3, depósito}*. Si la distancia a recorrer de la unión de ambos fuera menor a la de los dos por separado (cosa que por desigualdad triangular es cierta), podríamos unirlos formando un nuevo camino que reduzca la distancia total a recorrer.

El último paso de la heurística consiste en hacer precisamente esto. Revisamos, para cada camino existente (a lo sumo n caminos, que será el caso canónico), si existe otro camino (*candidato*) con el que se lo pueda unir de forma que la demanda de ambos sumada no supere a la de un vehículo y de forma tal que la distancia a recorrer de su unión sea menor a la de la suma se cada uno por separado. De encontrar caminos que sean candidatos a unirse, nos quedamos con todos los pares que más minimicen la distancia tales que dados dos pares p_1 y p_2, $\forall\, i \in p_1,\ i \notin p_2$ (es decir, pares disjuntos uno a uno). Este proceso requiere, cuando mucho, n^2 repeticiones.

Una vez que hemos unido los mejores caminos candidatos es menester volver a analizar si no es posible volver a unir rutas, dado que las rutas han cambiado. Esto implica volver a repetir el proceso del párrafo anterior, con lo cual este paso acaba teniendo una complejidad de $O(n^3)$.

```
1  unirCaminos(rutas):
2      posiblesUniones ← ∅
3      hacer:
4          paraCada ruta en rutas:
5              candidatos ← obtenerCandidatosDeRutasAUnirCon(ruta)
6              paraCada candidato en candidatos:
7                  posiblesUniones ← posiblesUniones ∪ {<ruta, candidato>}
8          //ordeno de mayor a menor las distancias ahorrables por los pares
9          ordenarPorDistanciaAhorrada(posiblesUniones)
10         //Filtro, del peor al mejor, los pares no disjuntos.
11         //Es decir, si existe un par que tiene un elemento que tenga
12         //otro par, elimino aquel cuya unión ahorre menos distancia.
13         filtrarDeAtrasHaciaAdelanteParesNoDisjuntos(posiblesUniones)
14         unirPares(posiblesUniones)
15     repetir si se unieron caminos
16     devolver rutas
```

Hecho esto, tenemos en rutas una serie de caminos que cumplen con la capacidad de los camiones y cuya distancia es igual o menor a la del caso canónico. Cada ruta es disjunta con cualquier otra ruta (a excepción de los depósitos) y la cantidad de camiones a utilizar es la misma que la cantidad de rutas a recorrer.

3.1.2. Ejecutable

Esta heurística se encuentra en directorio `Codigo/proxMinimo`, es el archivo `proxMinimo`. Para ejecutarla basta con escribir el comando `./proxMinimo <set>` con algún archivo de set de TSPLib. Su código fuente es `proxMinimo.cpp`.

3.1.3. Análisis de complejidad temporal

Partiendo de las secciones del algoritmo explicitadas en la **sección 3.1.1**, analicemos la complejidad de la heurística.

Se tiene que la complejidad total del algoritmo será la suma de las complejidades de cada una de sus subsecciones: `armarCaminos`, `cortarCaminos` y `unirCaminos`. En la sección mencionada ya vimos que la `unirCaminos` pertenece a la clase $O(n^3)$, dado que repite n veces una serie de operaciones

cuadráticas (buscar candidatos, ordenar los candidatos) y lineales (filtrar candidatos, unir caminos).

La complejidad de `armarCaminos` es cuadrática, ya que para cada nodo tengo que revisar cada otro nodo comparando su distancia para encontrar a qué punto moverme. El resto de las operaciones que se realizan de forma cíclica es constante, por lo que no hacen al coste del algoritmo. Obtener los depósitos a partir de una serie de nodos es una operación lineal, por lo que `armarCaminos` pertenece a la clase $O(n^2)$.

La complejidad de `cortarCaminos` depende de la cantidad de rutas armadas por el paso anterior. Como todas las operaciones son constantes pero se repiten por cada nodo en cada ruta, se podría afirmar que la complejidad de este paso es también cuadrática: como mucho tengo siempre n nodos y, de caer en el caso canónico, tendré como mucho n caminos. Sin embargo, es importante observar que conforme la cantidad de caminos crece, la cantidad de nodos a recorrer por camino decrece. Si bien es válido, entonces, afirmar que `cortarCaminos` pertenece a la clase $O(n^2)$, es posible que su coste final sea menor. No es relevante esto para saber el coste final del algoritmo y, por tanto, no nos detendremos a analizarlo.

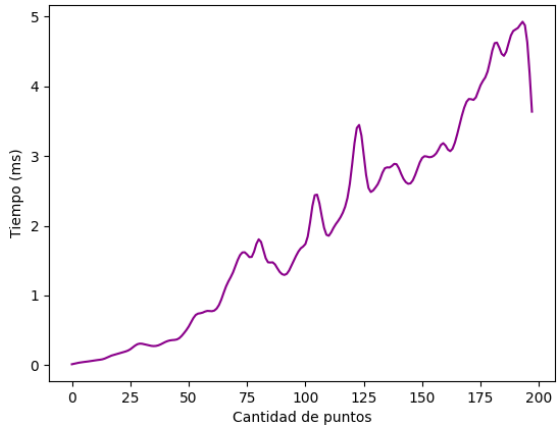
Dado que de los tres pasos, `unirCaminos` es el más costoso, la complejidad de esta heurística acaba siendo $O(n^3)$.

3.1.4. Rendimiento para instancias predefinidas

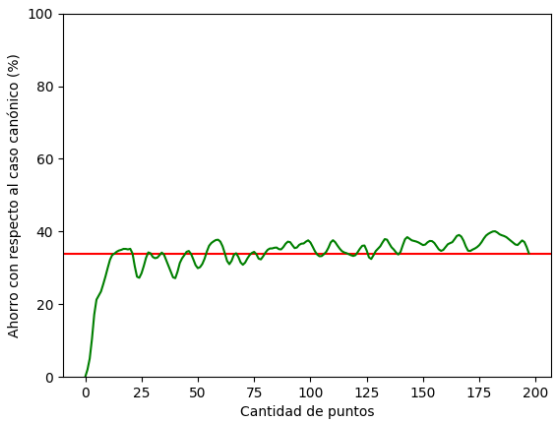
`TODO: Hacer`

3.1.5. Rendimiento para instancias aleatorias

Para estabilizar la medición, la heurística fue ejecutada sobre ciento noventa y ocho mil instancias generadas aleatoriamente por el generador mencionado en la **sección 2.3**, *1000* para cada cantidad de nodos entre *3* y *200*. Se midió el tiempo de ejecución de la heurística para cada caso evaluado y se plasmó el resultado en el gráfico presentado a la derecha de este párrafo. La medición total, en el equipo mencionado en la **sección 2.1** llevó aproximadamente una hora y cuarenta minutos.



Tiempo de ejecución de la heurística del próximo mínimo en relación a la cantidad de puntos del grafo.



Ahorro porcentual de la heurística del próximo mínimo con respecto al coste de la solución canónica.

Si bien las mediciones fueron dispares -por la naturaleza del algoritmo es natural que existan instancias que sean evaluadas mucho más rápidamente que otras (como por ejemplo cuando no es necesario cortar caminos o existen muy pocos caminos a evaluar en el paso de unir caminos)- en general puede observarse una tendencia creciente levemente aparabolada. Estimamos que, de realizar más mediciones y evaluar hasta casos de mayor cantidad de nodos, esta parábola será visible más fácilmente.

En lo que respecta a la calidad de las soluciones, para el caso aleatorio el ahorro porcentual representó, en promedio, un 37% de ahorro con respecto a la solución canónica. En los mejores casos este ahorro fue del 41%, mientras que en los casos más patológicos -presentados principalmente cuando la instancia contaba con poca cantidad de nodos- el ahorro fue nulo.

3.1.6. Caso patológico

Está claro por la sección **3.1.1** que, en el peor caso, esta heurística devolverá el caso canónico. Si no se tiene en cuenta las demandas de los clientes, esto podría suceder en casos donde se tenga dos nodos enfrentados en lados opuestos del depósito únicamente, ya que de haber por lo menos un nodo más por desigualdad triangular será más eficiente unir dos de ellos. En el caso con dos nodos mencionado, unirlos o presentar la solución canónica implica el mismo impacto en el trayecto a recorrer.

Cuando la capacidad de los camiones comienza a entrar en juego, sin embargo, se presentan una serie más de casos que pueden acabar en la solución canónica, y es cuando no es posible unir caminos de nodos de forma que su demanda sea menor a la capacidad de nuestro camión. Esto significa que todo par de nodos i, j cumple que $i + j > capacidad$, entonces nos encontraremos en un peor caso. Esto ocurre cuando para cada nodo i vale que $demanda(i) > capacidad / 2$.

3.1.7. Conclusiones

La *heurística del próximo mínimo* ejecuta en tiempo que escala de forma razonable al tamaño de su entrada y resulta en buenas soluciones si los puntos a visitar se encuentran cerca entre sí y sus demandas son bajas (llegando a costar el 59% del coste ofrecido por el caso canónico) y en soluciones aceptables para el promedio de los casos (aproximadamente el 63% de la solución canónica). Es sencillo de comprender y fácil de implementar, por lo que lo convierte en un acercamiento inicial tentador a la búsqueda de soluciones combinatorias para el CVRP.

3.2.Heurística constructiva golosa: *Merge Más Cercanos*

3.2.1. El algoritmo

La heurística constructiva golosa que hemos llamado Merge más cercanos consiste en unir rutas en función de la cercanía de sus puntos. Se parte desde la solución canónica al problema para obtener rutas iniciales y luego se van realizando uniones entre las rutas que contengan los dos puntos más cercanos que no pertenecen a la misma ruta.

Veamos el pseudocódigo de "Merge más cercanos":

```
1  MergeMasCercanos(G grafo):
2  rutas ← solucionCanonica(G)
3  pares ← paresDePuntosPorDistancia(G)
4  while (pares != ∅):
5      (a,b) ← pares.primero()
6      rutaA ← rutaALaQuePertenece(a)
7      rutaB ← rutaALaQuePertenece(b)
8      if(Σ(demandas(rutaA)) + Σ(demandas(rutaB)) <= capacidad)
9          rutas[rutaA] ← rutaA U rutaB
10     rutas ← rutas - rutaB
11     pares.desencolar()
12 return rutas
```

Como podemos ver, el algoritmo es muy simple. Una aclaración importante es que la unión entre las rutas une el último elemento no depósito de `rutaA` con el primer elemento no depósito de `rutaB` y luego descarta estos depósitos de manera que la ruta obtenida sea una ruta válida. Tampoco es menor enfatizar que no importa qué nodos conformen el par más cercano, la unión será siempre entre el último nodo de la ruta del primer elemento y el primer nodo de la ruta del segundo elemento.

3.2.2. Análisis de complejidad temporal

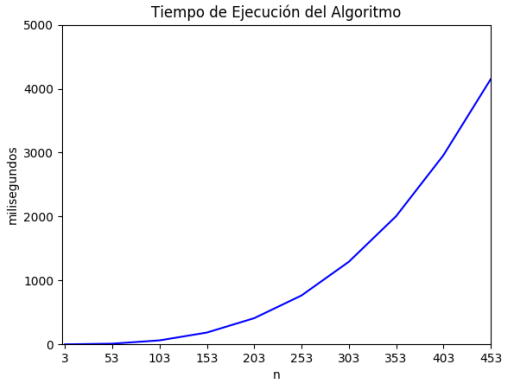
Descompongamos el pseudocódigo en pasos:

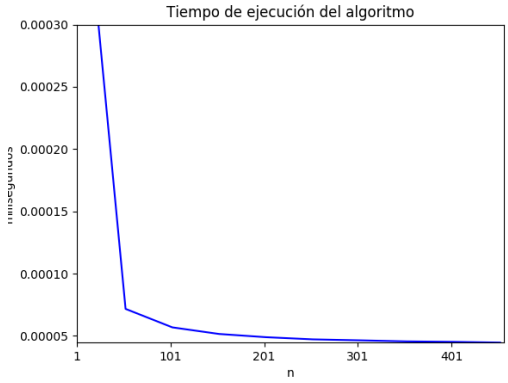
- La solución Canónica se puede obtener fácilmente en $O(n)$.
- Dado que el grafo de entrada está implementado como una Matriz de Distancias, la manera menos costosa de obtener todos los pares de puntos ordenados por distancia es recorrer toda la matriz formando los pares y colocandolos en un vector, y luego ordenar el vector con algún algoritmo eficiente. Esto nos cuesta $O(n^2)$ para recorrer la matriz y $O(n^2 * log(n^2))$ para ordenarla, por lo tanto todo el procedimiento es $O(n^2 * log(n^2)) = O(n^2 * log(n))$.
- Como el `while` se utiliza sobre una estructura que contiene n^2 elementos en la que para cada uno llamamos a `rutaALaQuePertenece`, y dado que nuestra implementación actual de esta función es una búsqueda lineal, el procedimiento es $O(|rutas|)$. $O(|rutas|)$ puede ser $O(n)$ en el caso en el que no es posible realizar ningún merge, por lo que el while sin tener en cuenta el merge es $O(n^3)$ en el peor caso. Los merge no cambian esta complejidad asíntotica dado que la

complejidad de la unión entre `rutaA` y `rutaB` es $O(|rutaA|)$, que pertenece a $O(n)$.

- Podemos concluir entonces que el algoritmo pertenece a $O(n^3)$

Comprobemoslo realizando las mediciones directamente. Generamos 400 instancias de grafo de para cada tamaño desde $n = 3$ hasta $n = 453$, con saltos de a 50 y promediamos el tiempo obtenido. Los resultados se pueden ver a continuación:



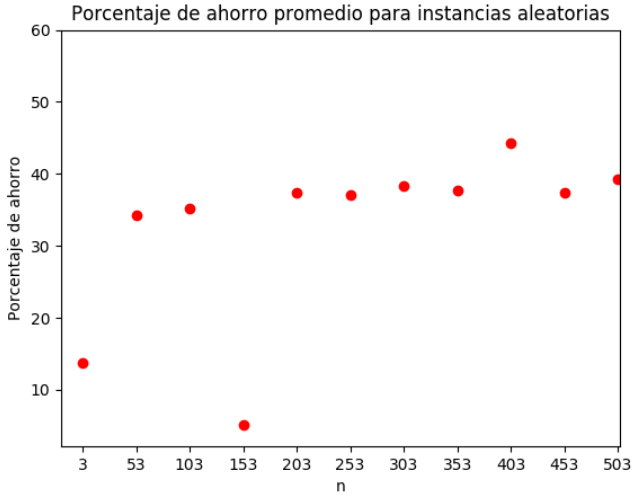


Tiempo de ejecución del algoritmo dividido n^3

3.2.3. Rendimiento para diferentes sets de instancias

TODO: Asi como está lo paso muy por arriba el por qué de como medimos. Lo tengo bien explicado en SA. me parece que deberiamos explicarlo una sola vez

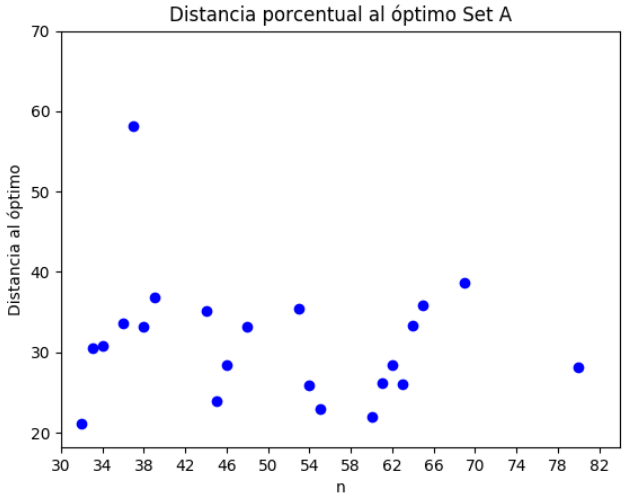
Veamos el rendimiento del algoritmo propuesto para los sets A, X y un set aleatorio de nuestra autoría. El set aleatorio consta de 400 instancias de grafo para cada tamaño desde $n = 3$ hasta $n = 503$, con saltos de a 50. Para las instancias aleatorias expresaremos el rendimiento como porcentaje ahorrado en promedio desde la solución canónica y en los sets en los que se conoce el óptimo lo haremos como porcentaje de la solución óptima.



Como podemos apreciar, para la mayoría de los tamaños probados el porcentaje de ahorro ronda el 35%, y no parece depender fuertemente del número de clientes de la instancia. Sin embargo suponer que su rendimiento sí depende de la distribución de los puntos en el plano ya $n = 153$ resultó en un rendimiento muy pobre. Esto nos da un indicio de la posibilidad de casos patológicos, ya que como calculamos el porcentaje de ahorro de 400 instancias diferentes y resultó ser tan malo, debe haber varias distribuciones posibles de clientes que resulten en malos rendimientos.

Dudamos que el mal rendimiento esté relacionado puntualmente a los grafos de $n = 153$, ya que ninguna parte del algoritmo depende directamente de la cantidad de clientes, si no a la relación posicional entre ellos. Además, si hubiera alguna relación entre el rendimiento y el tamaño probablemente habría evidencia de ello en los tamaños de instancias cercanas anteriores y posteriores (o sea $n = 103$, $n = 203$).

Veamos ahora los rendimientos para el Set A:

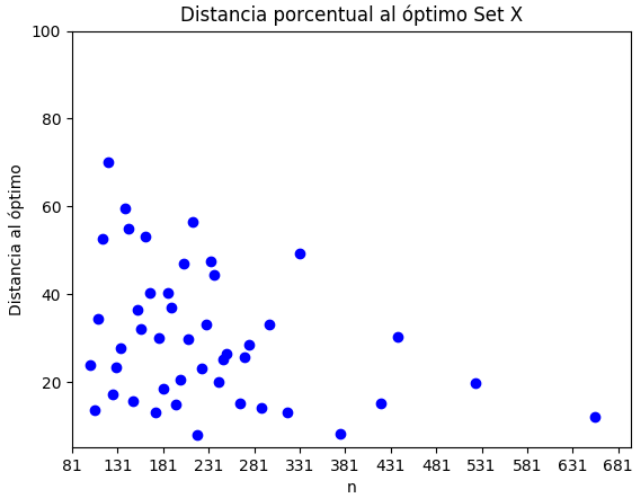


La distancia porcentual al óptimo va entre 25% y 40% para la mayoría de las instancias, lo cual no parece ser particularmente buen rendimiento dado que son instancias de tamaño pequeño. Sin embargo, dado que las instancias de A son generadas aleatoriamente sin ninguna especie de patrón o criterio particular, es entendible que el rendimiento no sea muy bueno.

Es importante destacar el caso de la instancia $n = 36$ cuyo rendimiento fue muy malo, proveyéndonos de más evidencia para nuestra hipótesis de la existencia de casos patológicos.

Por último analizemos los resultados del algoritmo aplicado al Set X:

Con este set de instancias obtuvimos resultados bastante mejores que con el Set A. Hay una gran cantidad de instancias rondando el 15% del óptimo y otra rondando el 30%. Sigue habiendo casos muy poco eficientes, notablemente $n = 120$ que ronda el 70% de distancia porcentual al óptimo.



3.2.4. Caso patológico

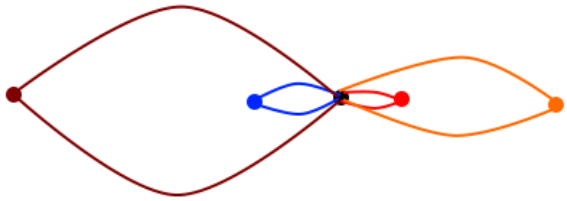
Dada la abundante evidencia de que hay casos para los cuales la heurística tiene un mal rendimiento, veamos si podemos determinar qué características de la distribución de los clientes lo causa. Nuestra hipótesis es que tener casos patológicos se debe al hecho de que si bien estamos utilizando los pares de clientes más cercanos disponibles, la unión entre las rutas se da entre el último elemento de una ruta y el primero de la otra, por lo que si estos dos elementos son lejanos, se generará una ruta con un tramo de gran distancia.

Veamos paso a paso la ejecución del algoritmo en un caso de estas características:

En esta instancia de problema tenemos un depósito identificado con el color negro y cuatro clientes que son el resto de los nodos del grafo. De ahora en más los llamaremos por la inicial del color que los identifica, es decir, serán R para el Rojo, N para el Naranja, A para el Azul y M para el marrón. El valor de sus demandas no es de importancia, lo único que nos importa es que un sólo vehículo pueda cargar con la suma de las cuatro demandas.

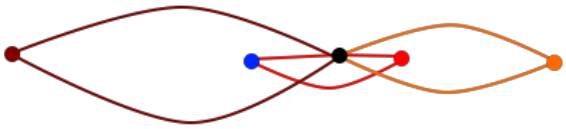
Acorde al primer paso del algoritmo calculamos la solución canónica, formando las siguientes

rutas:

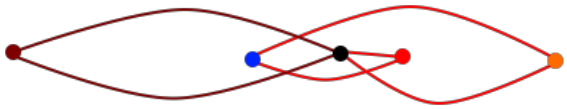


Es importante aclarar que la trayectoria que determinan las curvas utilizadas para mostrar las rutas no son representaciones adecuadas de los movimientos del vehículo, si no que las dibujamos de esta manera para que sean más fácilmente distinguibles. La trayectoria de los camiones será una línea recta.

Ahora ordenamos los pares de nodos por distancia de menor a mayor. Estos serán (R,A),(R,N), (A,N), (A,M),(R,M) y (M,N). Como R y A son el par de menor distancia, no pertenecen a la misma ruta y cumplen que la suma de sus demandas es menor que la capacidad del camión, realizamos un merge entre sus rutas, de la siguiente manera:

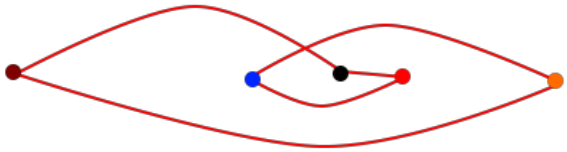


Seguimos iterando por la lista de pares. El próximo par es (R,N). Como no pertenecen a la misma ruta y la suma de las demandas lo permite, mergeamos la ruta que pasa por R y A con la ruta que pasa por N. Por cómo está definido el merge, uniremos el último nodo de la primer ruta con el primero de la segunda ruta, resultando en lo siguiente:



Es una ruta que sale del depósito pasando primero por R, luego por A y finalmente por N antes de regresar al depósito.

Viendo nuestra lista de pares, el siguiente en línea es (A,N), pero ambos ya pertenecen a la misma ruta, por lo que seguimos con (A,M). Realizando el merge, unimos el último elemento de la primera ruta con el primero de la segunda, generando el siguiente grafo:



Luego de esta iteración, no habrá más cambios a las rutas dado que como todos los nodos pertenecen a la misma ruta, no hay más merge posibles.Como se puede observar, generamos una ruta muy ineficiente para un recorrido fácilmente realizable con recorridos más cortos. Por ejemplo, se podrían haber utilizado dos vehículos (uno que pase por A y M y otro por R y N) para dar una solución mejor.

Ejecutando una instancia análoga a este caso con las posiciones cartesianas Deposito = (50,50), R = (60,50), A = (45,50), N = (80,50) y M = (0,50) resultó en un recorrido total igual de extenso que la solución canónica, efectivamente ganando nada de la ejecución del algoritmo excepto la utilización de un único camión en vez de cuatro.

Podemos concluir entonces que los casos patológicos son muy devastadores para el rendimiento de la heurística. Dado que la distribución de puntos necesaria para formar uno de estos casos es de generación relativamente común en una instancia aleatoria no nos sorprende que el rendimiento del algoritmo pueda ser bastante malo para un grupo grande de casos aleatorios.

3.2.5. Conclusiones

La heurística "Merge más cercanos" es una heurística simple de tiempo de ejecución razonable (al menos para grafos no masivamente grandes) que resulta en buenas soluciones si la distribución de los puntos del grafo es la adecuada, en soluciones de mediana calidad (entre 15% y 30% más que el óptimo) para la mayor parte de las instancias con las que la probamos y en muy malas soluciones en sus casos patológicos que son relativamente comunes. Sin embargo no deja de ser una herramienta más que podría resultar útil para un grafo particular o familia de grafos que comparten ciertas características.

3.3. Cluster-First & Route-Second

3.3.1. El algoritmo

Cluster-First & Route-Second es una heurística constructiva la cual vamos a usar para resolver el problema planteado de forma polinómica. Como lo indica su nombre la heurística tiene dos procedimientos bien marcados. Uno es el de clusterizar los nodos del grafo, esto significa agruparlos en base a cierto criterio y el otro es el de resolver los problemas de ruteo propiamente dichos para cada cluster en particular. Pasamos a explicar los detalles de cada procedimiento.

Cluster-First mediante Sweep algorithm:

Nuestro criterio para clusterizar, en este caso, va a estar basado en los ángulos que se forman entre distintos pares de nodos. Esto significa que, vamos a elegir un par de nodos arbitrario (uno de ellos va a ser siempre el deposito) y en base a ese vector que acabamos de generar, calculamos los ángulos que hay entre este y los otros vectores que se forman entre el depósito y los demás nodos del grafo.

Finalizado este procedimiento, pasamos a ordenar los angulos calculados de menor a mayor para luego ir agrupando los nodos que tengan ángulos mas cercanos y que no se pasen de la capacidad máxima de nuestros vehículos. En última instancia, vamos a tener agrupados nuestros nodos por "vecindad" y vamos a poder afirmar que la suma de la demanda de cada uno de los nodos de un cluster en particular, no excede la capacidad de nuestros camiones.

Cluster-First por ejes inconsistentes:

Como segunda opción de clusterización, vamos a usar el algoritmo ya implementado en nuestro previo trabajo de investigación, el cual clusteriza transformando el grafo original en un *AGM* y chequeando que si un eje de nuestro *AGM* es considerablemente mas largo que sus vecinos este mismo se elimina quedando dos partes del grafo claramente separadas. Cabe aclarar que en caso de haber generado clusters que exceden la capacidad máxima de nuestros camiones, estos mismos van a ser reclusterizados hasta que los nuevos clusters cumplan con la restricción, ya que este clusterizador no esta diseñado específicamente para resolver *CVRP*

Route-Second:

La segunda parte de la heurística va a ser común para los dos algoritmos de clusterizado. Como sabemos que ningún cluster generado excede la capacidad máxima de nuestros camiones, vamos a "asignarle" un camión a cada cluster. Dado que nuestro objetivo es hacer la ruta mas corta posible por cada camión, vamos a correr un algoritmo de *TSP* para cada uno de ellos. Como bien sabemos, no se conocen algoritmos polinomiales para resolver un problema de *TSP* de manera óptima por lo que

vamos a usar una heurística nuevamente.

Para la resolución del *TSP*, elegimos utilizar *nearest neighbour* la cual es una heurística que va eligiendo el eje más cercano al nodo en el cual nuestro camión esta posicionado sin repetir nodos ya visitado.

3.3.2. Pseudocódigos

Pasamos a mostrar los pseudocódigos de los algoritmos implementados para este caso.

3.3.2.1. Sweep:

```
1  calcularAngulos(vector(Nodo) v, int puntoComienzo) → vector(Nodo):
2    for a in v:
3      producto ← v_puntoComienzo (x) * v_a (x) - v_puntoComienzo (y) * v_a (y)
4      determinante ← v_puntoComienzo (x) * v_a (y) + v_puntoComienzo (y) * v_a (x)
5      angulo ← arcoTangente(determinante, producto)
6      angulos ← angulo //En grados
7    sort(v, angulos)  //Ordeno mi vector de nodos en base a los ángulos
8    return v
```

Correctitud:

El algoritmo itera por todos los nodos del grafo calculando los ángulos entre el vector (deposito, puntoComienzo) y vector(deposito, a) para terminar ordena de menor a mayor los nodos según los ángulos calculados

Complejidad:

Se itera por un vector de tamaño n haciendo operaciones constantes tomando $O(n)$ en total y ordenando el vector en $O(n \log n)$ tomando un total de $O(n \log n)$. Siendo n la cantidad de nodos del grafo.

```
1  clusterizarNodos(Grafo G, vector(Nodo) v) → vector(Cluster):
2      vector(Cluster) clusters
3      int peso ← 0
4      int i ← 0
5      while i < |v|:
6    Cluster cluster
7          peso ← 0
8    while (peso < capacidad(G) && i < |v|):
9        if(peso + demanda(v_i ) > capacidad(G)):
10     peso ← capacidad(G)
11       else:
12     peso ← demanda(v_i )
13         cluster ← v_i
14     i ← i + 1
15   clusters ← cluster
16     return clusters
```

Correctitud:

En este caso iteramos sobre el vector *v* que está ordenado en base a los ángulos calculados previamente, tenemos una variable peso que va a llevar registro de la suma de las demandas de los nodos que pertenecen a un cluster, una vector donde vamos a guardar los clusters calculados, y una variable *i* la cual usamos para iterar los elementos de *v*. El primer while se encarga de crear un nuevo cluster, setear la variable peso a cero y guardar el cluster calculado en el while interno. Como dijimos anteriormente el while interno va a ir guardando nodos en el cluster creado anteriormente siempre y cuando el nodo agregado no haga que la demanda total del cluster se pase de la capacidad de nuestros camiones. De ser así, salimos del while y agregamos el cluster a nuestro vector de clusters (es importante aclarar que el cluster agregado no contiene al nodo que rompía el invariante de nuestros clusters). Al salir de los dos while podemos ver que en nuestro vector de clusters tenemos agrupados a nuestros nodos en clusters que no exceden la capacidad de nuestros camiones y que están cerca en nuestro grafo (esto se debe a que los nodos estan ordenados en base a sus ángulos).

Complejidad:

Al tener dos whiles anidados, se podría pensar a priori que la complejidad de este algoritmo es $O(n^2)$ pero es importante observar que los dos whiles terminan si ya iteramos todos los elementos del vector *v*. Aclarado esto, vemos que en los dos ciclos hacemos operaciones que toman tiempo constante por lo que podemos afirmar que nuestro algoritmo toma tiempo lineal.

```
1  tsp(Grafo G, vector(Nodo) n) → vector(Nodo):
2      vector(Nodo) solucion
3      deposito ← deposito(G)
4      solucion ← deposito
5      //Encuentro el nodo mas cercano al deposito que no
6      //pertenece a la solucion
7      Nodo nodoAgregar ← nodoMasCercano(deposito, n , solucion)
8      solucion ← nodoAgregar
9      for nodo in n:
10   //Encuentro el nodo mas cercano al nodoAgregar
11       //que no pertenece a la solucion
12         nodoAgregar ← nodoMasCercano(nodoAgregar, n, solucion)
13         solucion ← nodoAgregar
14     //Agrego el deposito al final para representar la vuelta
15     solucion ← deposito
```

Correctitud:

En este caso tenemos un vector donde vamos a guardar los nodos en el orden a recorrer, tenemos una variable *deposito* donde nos guardamos el deposito de nuestro grafo. Como tenemos que empezar nuestros recorridos desde el depos ito, lo agregamos a nuestra solución. El próximo paso a realizar consiste en crear una variable de tipo *Nodo* la cual va a representar el próximo nodo que tenemos que agregar a la solución. Para esto tenemos una función que nos devuelve el nodo mas cercano al que le pasamos como parámetro y que además no está en el vector solución. Esto lo hacemos para no repetir nodos en nuestro camino. El primer nodo a agregar que buscamos es el que está mas cercano al depósito para luego agregarlo a nuestra solución. Luego iteramos sobre el vector de nodos y calculamos el nodo que está mas cerca a nuestro *nodoAgregar* para luego agregarlo a la solución. Para finalizar agregamos el depósito a nuestra solición ya que además de empezar desde allí, tenemos que terminar en el mismo.

Complejidad:

Sabiendo que nuestra función *nodoMasCercano* toma tiempo lineal, podemos ver que hacemos un llamado a la función por cada nodo de nuestro vector *n* por lo que efectivamente *tsp* tarda $O(nc^2)$. Siendo *nc* la cantidad de nodos del cluster (en caso de tener un solo cluster tarda $O(n^2)$).

```
1  resolverCVRP_Sweep(Grafo G, int puntoInicial) ← double:
2      vector(Cluster) caminos
3      costoTotal ← 0.0
4      angulos ← calcularAngulos(nodos(G), puntoInicial)
5      clusters ← clusterizarNodos(G, angulos)
6      for cluster in clusters:
7    caminos ← tsp(G,cluster)
8      for camino in caminos:
9          costoTotal ← costoDeCamino(camino)
```

Correctitud:

La función empieza creando un vector de caminos vacío, e inicializa la variable que va a representar el costo total de resolver el problema. Luego ordena los nodos del grafo en base a sus ángulos para despues clusterizarlos con esta información. Terminado este proceso se aplica *tsp* a cada cluster, lo cual nos deja con un vector de caminos a recorrer para cada cluster. Por último se calcula el costo total de cada camino y se lo suma a la variable *costoTotal*, calculando efectivamente el costo de cada camión utilizado.

Complejidad:

Sabemos que *calcularAngulos* toma $O(n \log n)$ para cualquier caso y *clusterizarNodos* toma tiempo lineal para cualquier caso. También sabemos que *tsp* tarda $O(nc^2)$ por lo que al hacer esto para cada cluster de nuestro grafo, esto tarda $O(|clusters| * max(nc)^2)$ (donde *max(nc)* es el cluster con mas cantidad de nodos del grafo). En el caso de que tengamos n clusters, la función tomaría tiempo lineal ya que tenemos $O(n * 1^2)$ y en caso de que tengamos un solo cluster tomaría $O(1 * n^2)$. Lo por que en peor caso nuestra función toma $O(n^2)$.

3.3.2.2. Ejes Inconsistentes:

Aclaración: Dado que el segundo método de clusterización, como mencionamos anteriormente, ya fue tratado en otro trabajo de investigación realizado por nosotros solamente vamos a mostrar los pseudocódigos que no forman parte del mismo. La misma aclaración cabe para la sección de complejidad algorítmica. No vamos a deducir las cotas de complejidad de las funciones que ya fueron deducidas previamente.

```
1  partirCluster(Grafo G, Cluster c) ← vector(Cluster):
2      vector(cluster) particiones
3      if(pesoDeCluster(cluster) > capacidad(G)):
4    c1 ← cluster(0, |cluster|/2)
5    c2 ← cluster((|cluster| / 2) + 1, |cluster|)
6    p1 ← partirCluster(G,c1)
7    p2 ← partirCluster(G,c2)
8    particiones ← append(particiones, p1)
9    particiones ← append(particiones, p2)
10     else:
11   particiones ← cluster
12     return particiones
```

Correctitud:

En este caso, luego de clusterizar los nodos mediante el *clusterizador de ejes inconsistentes*, partimos los clusters generados para que los mismos cumplan con el requerimiento de las cargas de los camiones, ya que el clusterizador no toma en cuenta las demandas de los nodos.

Nuestra función toma un cluster en particular y si el peso del mismo excede la capacidad de los camiones lo partimos a la mitad y repetimos, recursivamente, el mismo proceso para los nuevos clusters. Las particiones *p1* y *p2* generadas, las guardamos en nuestro vector de particiones, sabemos

que las podemos agregar porque nuestra función partir clusters solo devuelve clusters que no excedan el peso de nuestros camiones. Si el cluster no excede el peso directamente lo agregamos al vector de particiones y retornamos. Esto último sería nuestro caso base y es lo que nos permite afirmar que las particiones que devuelve nuestra funciones son clusters validos.

Complejidad:

En este caso tenemos que pesoDeCluster toma *O(n)*, partir el cluster en dos mitades también toma *O(n)* los appends cuestan *O(n)* tambien y la llamada recursiva, por teorema maestro, vemos que cuesta *O(n log n)* (partimos el problema en dos subproblemas de igual tamaño y las otras funciones toman *O(n)*). Por lo que esta función toma *O(n log n)*.

```
1  clusterizadorEjesInconsistentes(Grafo G) ← vector(Cluster):
2      prim(matriz(G), nodos(G))
3      clusters ← detectarYEliminarEjesInconsistentes(G)
4      vector(Cluster) clustersValidos
5      for cluster in clusters:
6    pCluster ← partirCluster(C, cluster)
7    clusterValidos ← append(clusterValidos, pCluster)
8      return clusterValidos
```

Correctitud:

Como dijimos anteriormente no vamos a explicar por qué prim y *detectarYEliminarEjesInconsistentes* son correctas ya que lo hemos hecho en un trabajo previo. Luego de clusterizar el grafo lo único que resta hacer es recorrer los clusters generados y partirlos en clusters validos para luego retornar un vector de clusters validos.

```
1  resolverCVRP_EI(Grafo G) ← double:
2      costoTotal ← 0.0
3      Matriz m ← matriz(G)
4      vector(Cluster) caminos
5      vector(Cluster) clusters ← clusterizadorEjesInconsistentes(G)
6      matriz(G) ← m
7      clusters ← eliminarDeposito(G, clusters)
8      for cluster in clusters:
9    caminos ← tsp(G,cluster)
10     return costoTotal
```

Nota: en esta función copiamos la matriz de distancias ya que prim modifica la misma y nosotros necesitamos utilizar las distancias originales para usar tsp. Correctitud: Luego de clusterizar los nodos mediante el metodo de ejes incosistentes, eliminamos el deposito de nuestra solución (ya que el clusterizador no distingue el depósito de otros nodos) y luego para cada cluster creado, corremos tsp. Luego de correr tsp, en nuestro vector de caminos creado al principio de la función vamos a tener los caminos que tienen que recorrer cada uno de nuestros camiones en los clusters, a estos caminos les calculamos el costo total y asi tenemos efectivamente nuestra solución calculada.

Complejidad:

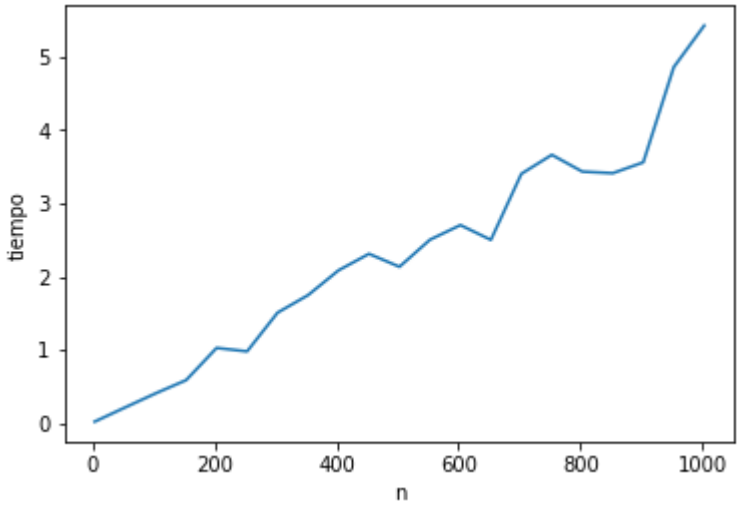
En el caso de que solo tengamos un cluster armardo, vamos a clusterizar en $O(n^2)$ además de copiar la matriz en el mismo tiempo. Eliminar el depósito nos va a tomar tiempo lineal (es recorrer todos los nodos del grafo hasta encontrar el deposito) y luego vamos a aplicar tsp en tiempo cuadrático. Por lo que esta función toma $O(n^2)$ en terminar.

3.3.3. Experimentación:

Perfomance:

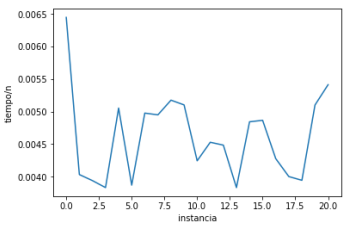
Los dos algoritmos fueron ejecutados con instancias aleatorias de un rango de tamaño entre 3 y 1003 con saltos de tamaño de 50 unidades. Para cada n se corrieron 400 instancias de ese mismo tamaño las cuales fueron posteriormente promediadas.

Sweep:

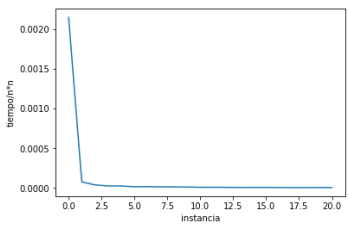


Performance de Sweep

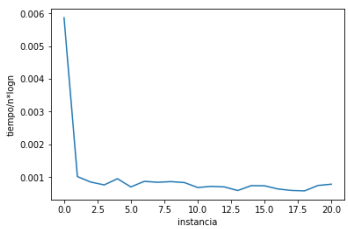
Podemos ver que el algoritmo a simple vista tiene el aspecto de tomar tiempo lineal. Pero al dividir por una función lineal vemos que no resulta en una función constante



Sweep dividido n



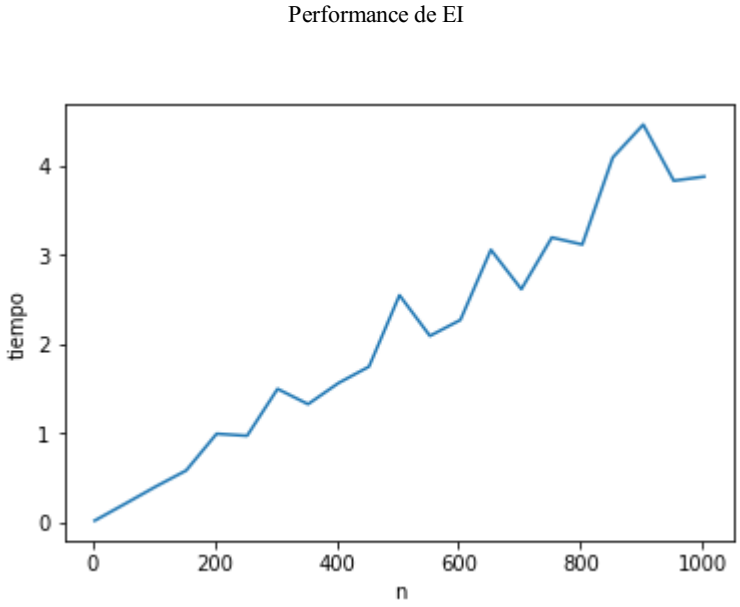
Sweep dividido n^2



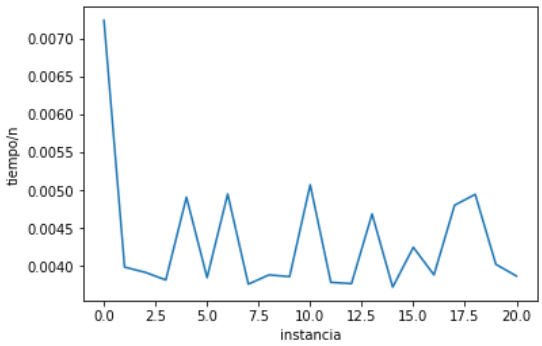
Sweep dividido n log n

así que podemos descartar que pertenezca a *O(n)*. Por otro lado vemos que al dividir el algoritmo por una función cuadrática obtenemos el resultado esperado, pero podemos acotar la complejidad un poco más, ya que el peor caso de nuestro algoritmo se da cuando nos queda todo el grafo en un solo cluster y esto en realidad no pasa usualmente, asi que dividiendo por n log n podemos ver que nuestra función resulta en una constante asi para este set de instancias nuestro algoritmo toma *O(n log n)*.

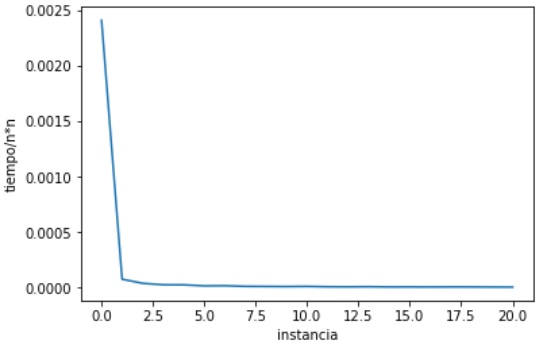
Ejes inconsistentes:



Igual que en el caso anterior podemos ver que el algoritmo a simple vista tiene el aspecto de tomar tiempo lineal pero al dividir por una función lineal vemos que no resulta en una función constante así que podemos descartar esta posibilidad.



EI dividido n

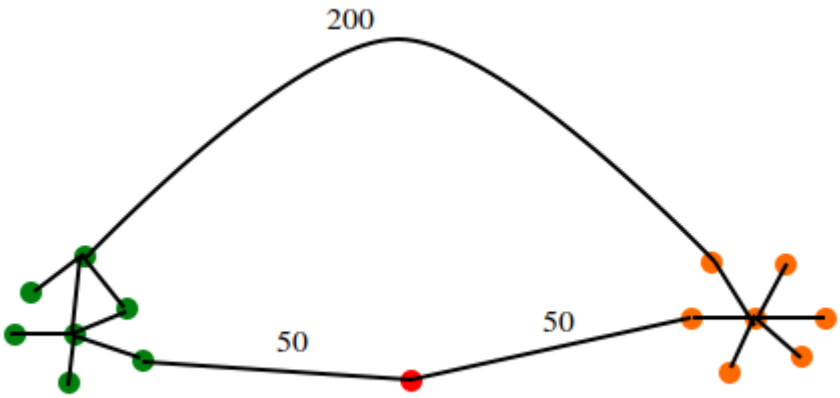


EI dividido n^2

Por otro lado vemos que al dividir el algoritmo por una función cuadrática obtenemos el resultado esperadolo cual confirma nuestra hipotesis de que *EI* es cuadrático.

Caso patológico en algoritmo de sweep:

Es relativamente fácil ver que este algoritmo tiene un problema y es que aunque clusterice en base a proximidad de puntos, el mismo solamente deja definido un cluster cuando agregar un nodo mas a este hace que un camión no pueda realizar el recorrido. Entonces, si nosotros tenemos dos clusters bien definidos, muy separados entre sí, pero la suma de las demandas de los dos no sobrepasa a la capacidad de nuestros camiones, el algoritmo los va a tomar como un único cluster y tendríamos un costo muy alto para ir desde un cluster hasta el otro cuando hubiese sido preferible usar un camión más y hacer un recorrido mayor. Notemos también que nos estamos limitando a hablar de solamente dos cluesters pero esto también podría extenderse a la cantidad de clusters que querramos siempre y cuando el peso de los mismos no exceda la capacidad de nuestros camiones. El punto a mostrar con este caso es que si nuestro objetivo principal al resolver el problema, es encontrar la solución más eficiente posible, *Sweep* prioriza la mínima utilización de camiones por lo que podríamos obtener soluciones mejores ya sea modificando el algoritmo para este caso (por lo que dejaría de ser *Sweep* para ser una suerte de variante) o tendríamos que considerar utilizar otro algoritmo, siempre y cuando la disponibilidad de camiones no sea un problema.



Caso patológico en Sweep

Este grafo es un ejemplo del caso patológico mencionado, vemos que la ruta de más arriba cuesta 200 mientras que volver al depósito e ir hacia el cluster contrario cuesta 100. Si nuestros camiones tuviesen una capacidad de 100 y todos nuestros nodos una demanda de 1, el algoritmo nos diría que solo tenemos que usar un camión para resolver este problema (tendríamos un solo cluster). Esto implica que, como nuestro camión no vuelve al depósito, va a viajar desde el nodo de un cluster hasta el otro cuando volver al depósito y usar otro camión devolvería una solución más eficiente en términos de costos.

Busqueda de mejores parámetros en promedio para un set de instancias partiuclar:

Nota: Es importante aclarar que la busqueda de mejores parámetros para un set de instancias dado fue solamente realizada para el clusterizador que usa el metodo de ejes incosistenses. Por más de que el algoritmo de *sweep* tome un parámetro (el nodo inicial) este mismo depende del tamaño del grafo en sí mismo. Al tener tres sets de instancias de tamaño variable, no tiene sentido encontrar el nodo mas conveniente del cual empezar ya que puede darse el caso de que el mismo (expresado como un natural) este fuera del rango de alguno de los grafos del set. Por ejemplo, podríamos llegar a la

conclusión de que el mejor nodo inicial en promedio en nuestro set de instancias es n pero tenemos grafos pertenecientes al set que son de tamaño estrictamente menor a n. Dicho esto también cabe aclarar que nuestro nodo inicial en estas experimentaciones fue siempre el nodo 1 (siendo 0 el deposito).

Procedemos a mostrar una tabla con los mejores parámtros encontrados para el algoritmo de ejes inconsistentes. La mismas tiene 3 columnas, la primera indica el dataset en el que fue calculado, la segunda el mejor f_t encontrado y la ultima el mejor d.

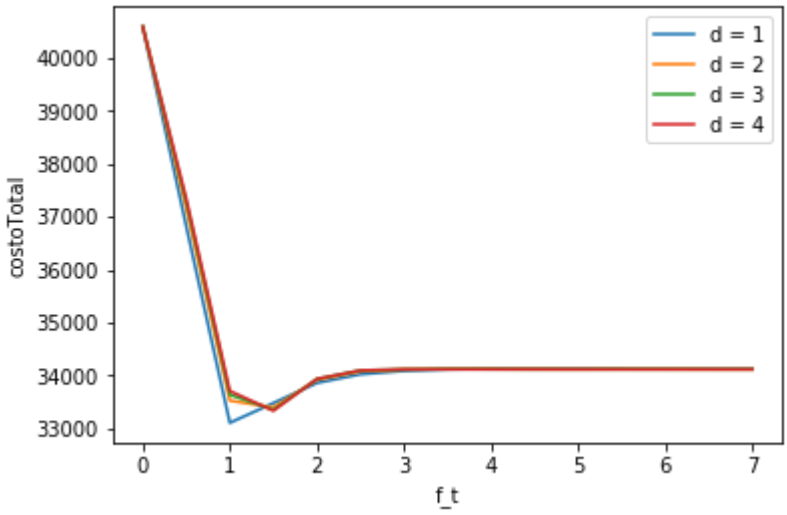
	DataSet	f_t	d
0	random	1.0	1
1	serie A	1.5	2
2	serie X	3.0	2

Caso aleatorio:

En este caso los algoritmos fueron corridos con instancias de tamaño 1 al 100 habiendo 400 instancias de tamaño $1 \leq n \leq 100$.

Ejes inconsistentes:

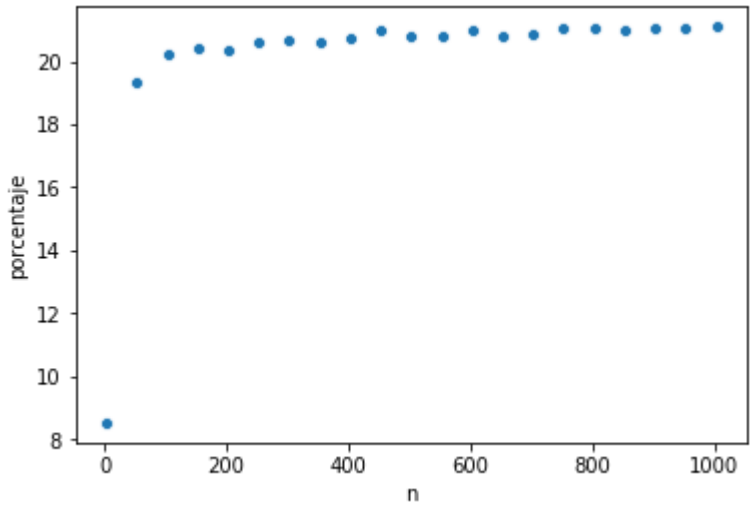
El algoritmo que clusteriza por medio de ejes inconsistentes toma dos parametros (además del grafo) f_t y d. Para encontrar los mejores parámetros y a la vez realizar una experimentación que lleve un tiempo razonable, decidimos tomar 4 valores de d posibles [1,...,4] y variar f_t entre [0,...,7] dando saltos de 0.5. Cada configuración posible de parámetros fue corrida para todas las instancias para luego promediar el costo total de todas. En el siguiente gráfico podemos ver los resultados obtenidos para cada d.



Mejores parámetros para caso aleatorio

Como podemos observar los *d* no varian demasiados unos de los otros pero hay una configuración que a simple vista minimiza los costos siendo la misma $d = 1$ y $f\_t = 1$. Vamos a utilizar estos para medir la perfomance de nuestro algoritmo en términos de costos.

Para esta parte de la experimentación vamos a representar los costos como un porcentaje. Este mismo es el porcentaje ahorrado en base al costo de la solución canónica. Esto quiere decir que si nuestro algoritmo devolvió un 43.4, tenemos una solución que es un 43,4% mejor que la solución canónica. Decidimos medir de esta manera ya que al ser un dataset aleatorio, no sabemos cuál es el ópitmo de cada grafo.

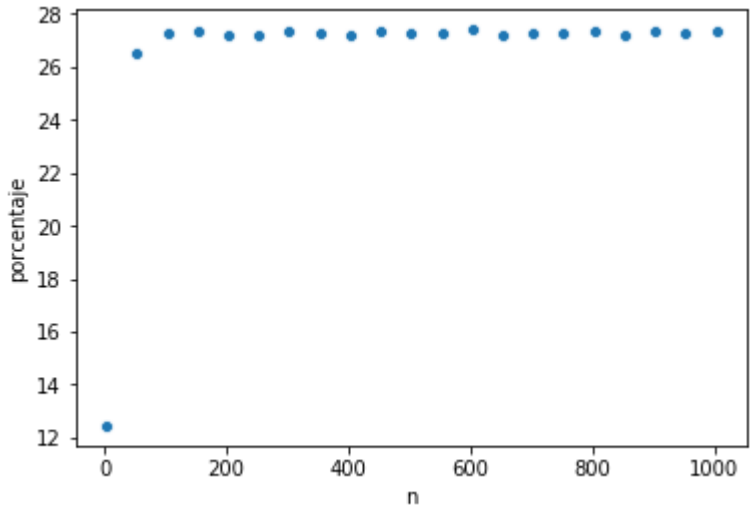


Ahorro de la canónica con mejores parámetros para caso aleatorio con EI

Podemos observar que las soluciones dadas en el siguiente gráfico están en su gran mayoría a una distancia del 20% de la solución canónica promedio de nuestro set de instancias.

Sweep:

En este caso también usamos el mismo criterio de medición que en el algoritmo de clusterización anterior, pero como ya aclaramos previamente no buscamos el mejor parámetro para este caso porque en nuestros sets de instancias carece de sentido.



Ahorro de la canónica para caso aleatorio

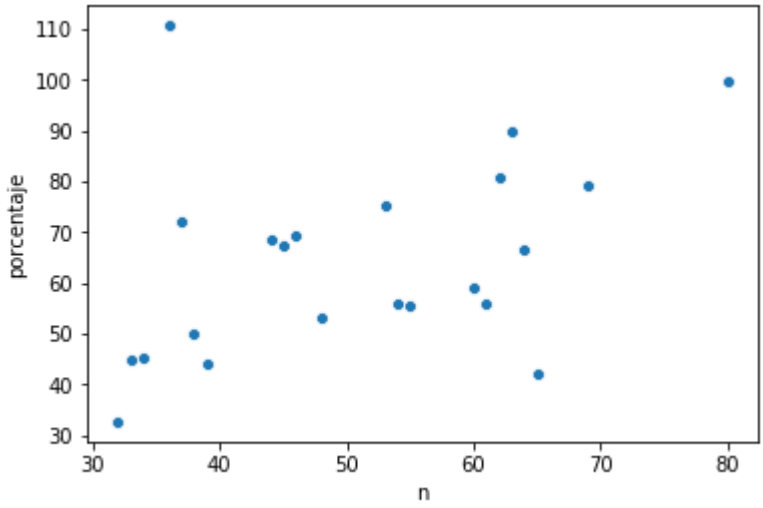
Podemos observar que sweep da soluciones un poco mejores que *EI* llegando casi a un ahorro del **30%** en la mayoría de los casos.

Nota: Tanto para el caso A como para el caso X se uso el mismo criterio para la busqueda de mejores parámetros que en el caso aleatorio y los resultados fueron los siguientes:

Caso A:

Ejes inconsistentes:

Usando los parámetros ya mostrados en la tabla previa medimos los costos promedio pero esta vez lo hacemos midiendo la distancia porcentual al óptimo (dado que en este set son conocidos). Esto quiere decir que si nuestro algoritmo devuelve 30,5 nuestra solución es un 30,5% peor que la solución óptima.

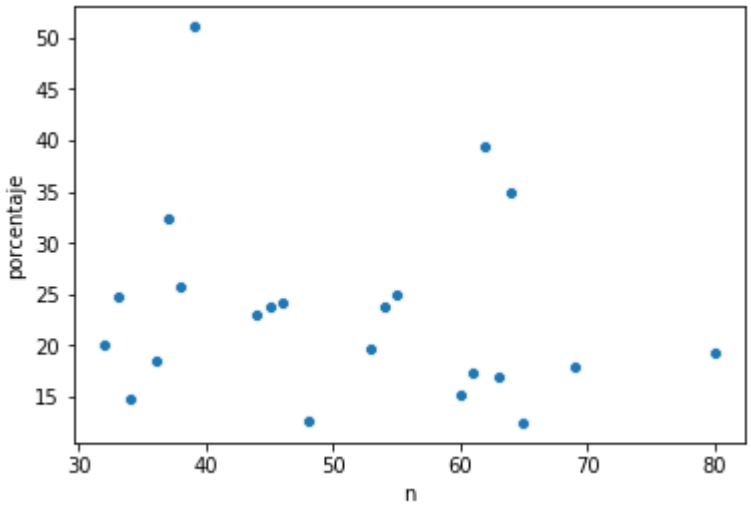


Distancia al óptimo con mejores parámetros caso A para EI

Podemos observar que, en la mayoría de los casos, nuestro algoritmo es entre un **50%** a un **70%** peor que la solución óptima promedio encontrada para este dataset.

Sweep:

En el caso de sweep la mayoría de las soluciones están a una distancia de entre un **15%** a un **25%** de la solución óptima promedio encontrada para el dataset correspondiente.

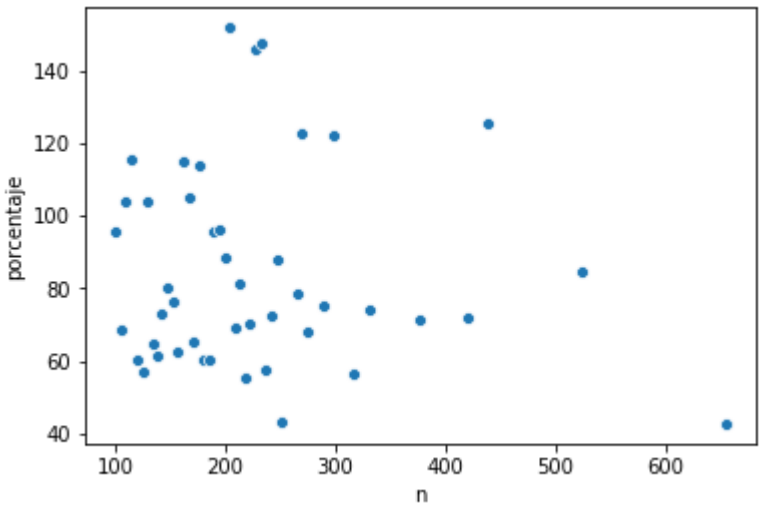


Distancia al óptimo con mejores parámetros caso A para Sweep

Caso X:

Nota: los criterios usados para la medición de este caso son idénticos a los del *caso A* con la salvedad de que se usan los parámetros correspondientes previamente calculados.

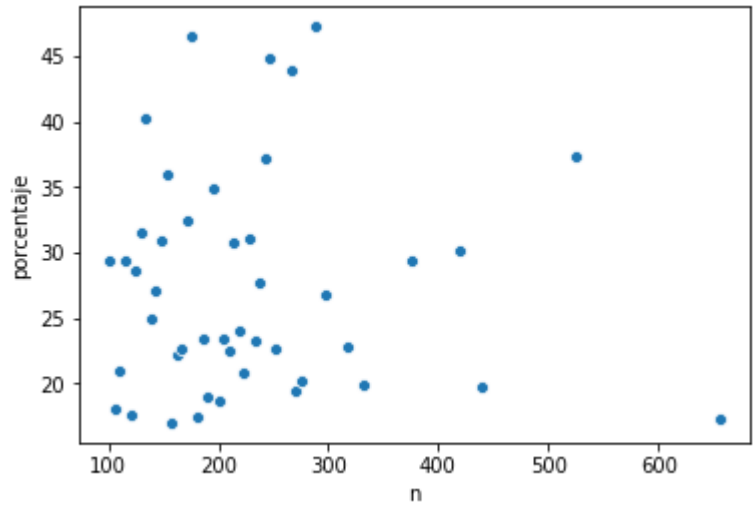
Ejes inconsistentes:



Distancia al óptimo con mejores parámetros caso X para EI

Para este dataset, el cuál tiene un tamaño promedio considerablemente mas grade que el *A*, obtenemos soluciones bastante más pobres. Las mismas estan en un rango de entre el **60%** al **100%** con algunas soluciones distando más del doble con la solución óptima.

Sweep:



Distancia al óptimo con mejores parámetros caso X para Sweep

Teniendo en cuenta que el dataset *X*, como dijimos anteriormente, tiene instancias de tamaño mayor al *A*, vemos que las soluciones en este caso son entre un **20%** a un **30%** peores que la óptima promediada.

3.3.4. Conclusiones:

Ejes inconsistentes:

Teniendo en cuenta todo el análisis previo podemos ver que en *EI* tanto para la serie A como para la *serie X* usamos el mismo *d*. Esto se debe a que aunque los datasets difieran en el tamaño de sus instancias por casi mil unidades, los tamaños de estos grafos no son tan significativos para el parámetro *d*, tener en cuenta nodos que estan a dos ejes de distancia estamos cubriendo bien nuestros grafos. Aumentar el *d* en este caso solo aumentaría lo que tarda el programa y no la calidad de su solución. Por otro lado el *f_t* aumenta para la *serie X*, esto se debe a que la cantidad de puntos aumenta por lo que nuestro criterio para determinar si un eje es inconsistente debe ser mas riguroso.

Al tener mas puntos en el plano queremos agrupar los que esten a una distancia mas corta ya que queremos que nuestros clusters queden bien definidos, achicar el *f_t* en este caso nos agruparía puntos que a simple vista no serían clusterizables. Esto no sucede en el *caso A* porque al ser menor la cantidad de puntos, es mas vaga la definición a simple vista de los clusters, por eso el *f_t* es menor.

Por último podemos observar que en el caso aletorio el *d* se achica en una unidad y el *f_t* baja. Concluímos que esto es debido a que los clusters en este set no siguen ninguna distribución en particular (respetando la aleatoreidad) por lo que nuestro criterio de clusterización es lo mas laxo posible.

En cuanto a costos, el algoritmo no tiene una performance deseable para ninguna de los 3 conjuntos. Más teniendo en cuenta tenemos otro algoritmo que realiza el mismo procedimiento *(cluster-first route-second)* que este y se comporta de manera mucho mas eficiente tanto en tiempo como en calidad de soluciones.

Sweep:

Podemos ver que *Sweep* tiene una performance relativamente buena tanto en tiempo como en costos, el algoritmo corre considerablemente más rápido que nuestra otra opción de *CF-RS*, y da soluciones mucho mas cercanas al óptimo y alejadas del peor caso.

Como mostramos anteriormente tenemos un caso donde el algoritmo funciona de manera ineficiente (en términos de costos) ya que priorizamos clusterizar nodos que no se pasen de la demanda sin tener en cuenta sus distancias. Si es posible analizar la instancia previamente y nos damos cuenta que estamos en este caso sería preferible usar el algoritmo de ejes inconsistentes u otro que clusterice en base a distancias en vez de a demanda (siempre que el uso de más camiones no sea un problema). En cualquier otro caso, si el problema se va a resolver con una heurística de *CF-RS*, en base a el análisis hecho, es preferible usar este algoritmo al otro.

3.4. Simulated Annealing

3.4.1. El algoritmo

Simulated Annealing es una metaheurística que se utiliza para encontrar soluciones aproximadas a problemas que no poseen un algoritmo de resolución polinomial. Difieren de las heurísticas en que se basan en buscar la mejor solución de un conjunto de muchísimas soluciones posibles. Esta busqueda se puede realizar de muchas maneras, pero simulated Annealing utiliza una **búsqueda local**. A partir de una solución S definimos un conjunto de soluciones relacionadas llamadas el vecindario de S. Revisamos estas soluciones y con algún criterio decidimos tomar una solución vecina S' y calculamos su vecindario. Esto se repite hasta que decidamos dejar de buscar con otro criterio.

El párrafo anterior parece ser un poco vago, ya que dejamos sin especificar muchas partes importantes de una búsqueda local como qué vecindario se utiliza, el criterio con el cual elegimos una solución y con cuál dejamos de buscar. Esto es intencional ya que hay una enorme cantidad de variantes posibles y para no perder generalidad debemos analizar cada caso por separado.

En el caso particular de Simulated Annealing buscamos una especie de híbrido entre explorar el conjunto de soluciones de manera de evitar los máximos locales (es decir, la mejor solución de un determinado conjunto de vecindarios que muy probablemente no sea la mejor solución global), pero aún así tender a seleccionar soluciones cada vez mejores para terminar encontrando una buena solución.

Para dar esta variabilidad seleccionaremos la próxima solución en base a un parámetro llamado Temperatura que comienza en un valor inicial (Ts) y decrece a medida que avanza la ejecución del programa. Este valor se utiliza como parámetro de una función de probabilidad P que decide si seleccionaremos una solución dada o no. Como regla general, se tiende a aceptar las soluciones mejores que la actual independientemente de la temperatura pero podemos aceptar soluciones peores si lo dicta nuestra función P. Como regla general, a menor temperatura es menor la probabilidad de selección de una solución inferior a la actual. También contamos con una función de energía E que nos permite comparar soluciones y en general depende del valor de la solución.

Cómo disminuye la temperatura a lo largo de la ejecución se denomina función de enfriamiento o **Cooling Schedule** y es fundamental para el desempeño del algoritmo. Hay muchas opciones posibles que varían en efectividad dependiendo del tipo de problema y de las instancias particulares.

Veamos el pseudocódigo de un esquema de Simulated Annealing:

```
1  Entrada:
2  S0: solución inicial, E: función energía, N: vecindario
3  Ts: Temperatura inicial, Tf: Temperatura final, CS: función que enfría
4  la temperatura, P: función de probabilidad de aceptación de una transición.
5  S* ← S0
6  S ← S*
7  T ← Ts
8  while(T > Tf)
9      Tomar S'∈ N(S)
10     if P(S, S', T) >= random(0,1):
11         S ← S'
12     if E(S) <= E(S*) : S* ← S
13     enfriar(T)
14 return S*
```

En nuestra implementación la función de vecindario que utilizaremos será **1-interchange**. El vecindario utilizando esta función está definido de la siguiente manera:

Sean S y S' soluciones. S' ∈ N(S) <=> ∃ i, j con i != j , i y j <= |rutas(S)|, ∃ c ∈ rutas[i], c != deposito(S) tal que shift(rutas(S)[i], rutas(S)[j], c) == rutas(S') ∨ ∃ c1 ∈ rutas[i] ∧ c2 ∈ rutas[j], c1,c2 != deposito(S) exchange(rutas(S)[i], rutas(S)[j], c1, c2) == rutas(S').

Siendo `Shift` y `Exchange` los siguientes procedimientos:

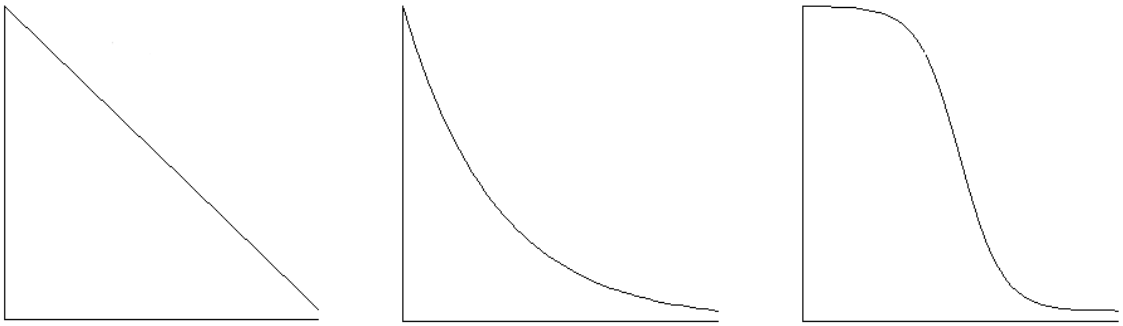
```
1  Shift(vector<Ruta> rutas ,Ruta ruta1, Ruta ruta2, cliente c, Grafo G)
2      if (rutas[ruta1][c].demanda + Σ(demandas(ruta2)) <= G.capacidad_total):
3        w* ← w ∈ rutas[ruta2] / minArg(G.distanciaEntre(rutas[ruta1][c],
rutas[ruta2][w]) + G.distanciaEntre(rutas[ruta2][w+1],rutas[ruta1][c]))
4        rutas[ruta2] ← rutas[ruta2][1..w] ∪ rutas[ruta1][c] ∪ rutas[ruta2]
[w+1..|rutas[ruta2|-1]
5        rutas[ruta1] ← rutas[ruta1] - c
6      return rutas
```

```
1  Exchange(vector<Ruta> rutas, Ruta ruta1, Ruta ruta2,c1,c2, Grafo G)
2    if(Σ(demandas(ruta1)) - rutas[ruta1][c1].demanda + rutas[ruta2][c2].demanda
<= G.capacidad_total &&
3    Σ(demandas(ruta2)) - rutas[ruta2][c2].demanda + rutas[ruta1][c1] <
G.capacidad_total):
4          temp ← rutas[ruta1][c1]
5          rutas[ruta1][c1] ← rutas[ruta2][c2]
6          rutas[ruta2][c2] ← temp
7      return rutas
```

Como podemos ver, `Shift` es un procedimiento tal que dadas dos rutas de la solución, remueve un cliente de una ruta y lo inserta en otra (siempre que la demanda del cliente insertado no exceda la capacidad del camión que recorre esa ruta) y `Exchange` un swap entre dos clientes de dos rutas (siempre que las capacidades de los camiones de ambas rutas no sea excedido). En la operación `Shift` la inserción siempre se realizará de manera que la suma del costo de las aristas agregadas menos la arista removida sea mínimo.

Como **Cooling Schedule** tendremos varias opciones con diferente efectividad. Más adelante realizaremos experimentos comparándolos entre sí. Las variantes que consideraremos serán *CS0*: Ti =

Ts - i*(Ts - Tf)/ NIt, *CS1*: Ti = Ts * Tf/ Ts^(i/NIt) y *CS2*: Ti = ((Ts-Tf)/(1+e^(0.3*(i - NIt/2)))) + Tf siendo *Ti* la temperatura en la i-ésima iteración, *Ts* la temperatura inicial, *Tf* la temperatura final, *i* el número de iteración y *NIt* el total de iteraciones a realizar. Veamos a continuación gráficos de estas funciones para poder apreciarlas más tangiblemente:



CS0 CS1 CS2

Nuestra función de probabilidad será P = e^-Δ/Ti siendo Δ = E(S') - E(S) y E la función de energía de una solución que en nuestro caso corresponde al valor de la solución, es decir, la suma total de la distancia recorrida por cada camión. Notemos que a medida que Ti disminuye el valor de P también lo hace, cumpliendo la regla general "a menos temperatura, menos aceptación" mencionada previamente. En *P* también tenemos en cuenta el valor de delta; si es negativo quiere decir que *S′* es menor que *S* y por lo tanto *P* dará un número mayor que uno de manera que aceptaremos *S′* como solución. En cambio si delta es positivo, su aceptación dependerá de su magnitud y de la temperatura, efectivamente cumpliendo que si *S′* es considerablemente peor que *S* sólo se acepte bajo altas temperaturas.

El único parámetro que nos queda por elegir es la heurística inicial (S0) a ser utilizada. Para S0 se puede elegir cualquier heurística, pero decidimos utilizar la solución a la que llamaremos canónica, en la que la ruta de cada vehículo consiste únicamente en ir a un único cliente y volver al depósito. Si bien esta heurística nos provee con una primera solución muy ineficiente, nos permite observar más fácilmente las ganancias de rendimiento de Simulated Annealing y en muchos casos ha resultado en soluciones mejores que otras heurísticas iniciales.

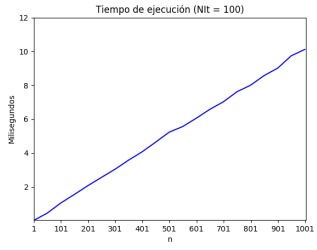
Es importante aclarar que realizamos una leve modificación al esquema de Simulated Annealing propuesto. En vez de determinar la finalización de la búsqueda cuando la temperatura llegue a un determinado valor, hemos agregado un nuevo parámetro *NIt* que nos indica el número de iteraciones a realizar. Este parámetro es útil ya que nos permite modificar la curva descrita por el Cooling Schedule y nos provee una cota superior de iteraciones para hacer análisis de tiempo de ejecución más fácilmente.

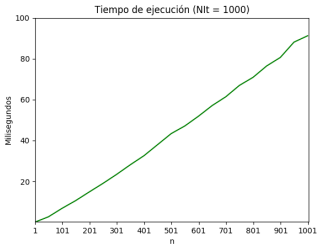
3.4.2. Analisis de complejidad

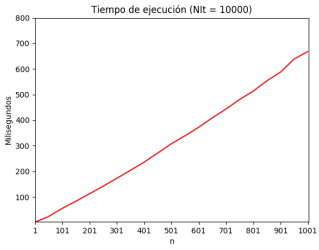
Dado que todas las operaciones se realizan en tiempo constante excepto el cálculo de la solución canónica que es O(n) y `Shift` que en su peor caso inserta en una ruta con todos los elementos menos uno, una cota simple del peor caso sería *n * NIt* donde n es la cantidad de nodos del grafo y *NIt* es la cantidad de iteraciones. Es una cota brusca porque la probabilidad de inserciones en rutas grandes

disminuye a medida que aumenta la cantidad de clientes (porque mientras más clientes tiene una ruta, menos probable es que no se exceda la capacidad del vehículo), pero dar una cota más ajustada es complejo porque depende de la instancia particular.

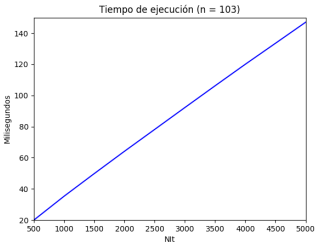
Para corroborar esta cota, veamos que Simulated Annealing es lineal en función de *n* y de *NIt*:

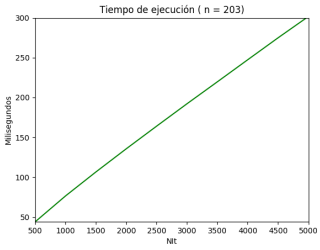


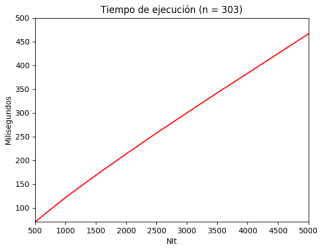




Fijando *NIt* en un valor constante, vemos como el algoritmo es lineal en función de *n*.







Fijando *n* en un valor constante, vemos como el algoritmo es lineal en función de *NIt*.

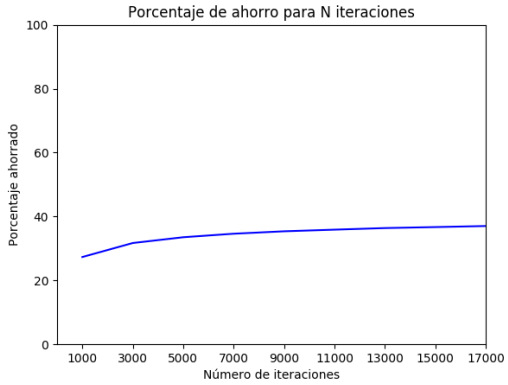
3.4.3. Búsqueda de parámetros óptimos: caso aleatorio

Busquemos el valor óptimo de los parámetros de nuestro algoritmo de Simulated Annealing para el casos aleatorio. Es decir, queremos encontrar el valor de los parámetros que mejores resultados obtiene en el promedio de muchos casos aleatorios, lo cual es útil para tener una noción general de qué valores tienden a obtener mejores resultados.

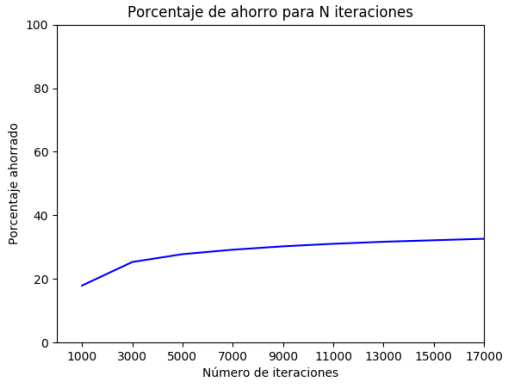
Para esto realizaremos una serie de experimentos, comenzando por un análisis de cómo el número de iteraciones realizadas (*Nit*) afecta el resultado. Sabemos que mientras más iteraciones realizemos mejor será el resultado ya que a fin de cuentas estamos realizando una búsqueda local, pero la pregunta que intentaremos responder es ¿Cuántas iteraciones tiene sentido medir? ¿Hay un punto a partir del cual la mejora obtenida de realizar más iteraciones no justifica el costo en tiempo de ejecución?

El experimento consistirá en medir el ahorro porcentual con respecto a la solución canónica promedio obtenido luego de *Nit* iteraciones para 400 instancias aleatorias del problema), con un varios n fijos. Decidimos comparar con la solución canónica para poder comparar los resultados de instancias de tamaños diferentes más claramente. La diferencia en efectividad observada utilizando las tres variantes de Cooling schedule dieron resultados muy similares, por lo que decidimos utilizar un único

Cooling Schedule para la medición por practicidad. A continuación mostramos los resultados obtenidos:



Ahorro porcentual de la solución canónica en función de NIt, $n = 103$

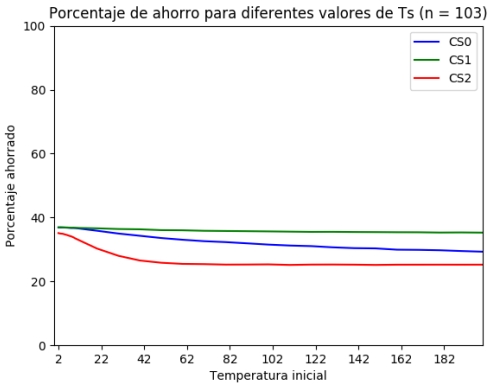


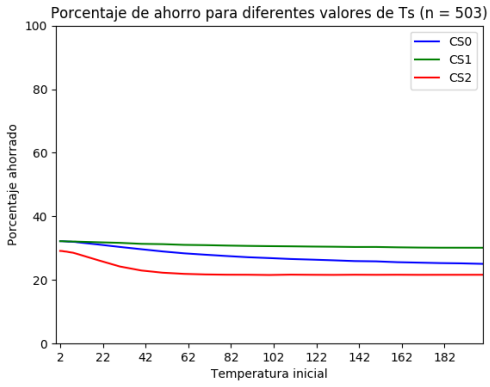
Ahorro porcentual de la solución canónica en función de NIt, $n = 503$

Como podemos apreciar en los gráficos nuestras predicciones fueron correctas. A medida que aumenta *Nit* el porcentaje de ahorro también aumenta. Notemos tambien que en ambos el pocentaje ahorrado aumenta rápidamente hasta las tres mil iteraciones y luego crece más lentamente. Esto se cumple en todos los grafos, pero a medida que aumenta el n el porcentaje ahorrado crece más rápidamente.

<<<<<<< HEAD Teniendo en cuenta las caracteristicas de las instancias que mediremos y que no disponemos de tiempo ilimitado para realizar las mediciones, decidimos que *Nit* = 10.000 es suficientemente grande para ser representativo del resultado del algoritmo y suficientemente pequeño para poder realizar el resto de las mediciones en un tiempo razonable. ======= Teniendo en cuenta que no disponemos de tiempo ilimitado para realizar las mediciones, decidimos que *Nit* = 10.000 es suficientemente grande para ser representativo del resultado del algoritmo y suficientemente pequeño para poder realizar el resto de las mediciones en un tiempo razonable. >>>>>>> 419812510afcca07d1b4e26995dcd1d9e254468b

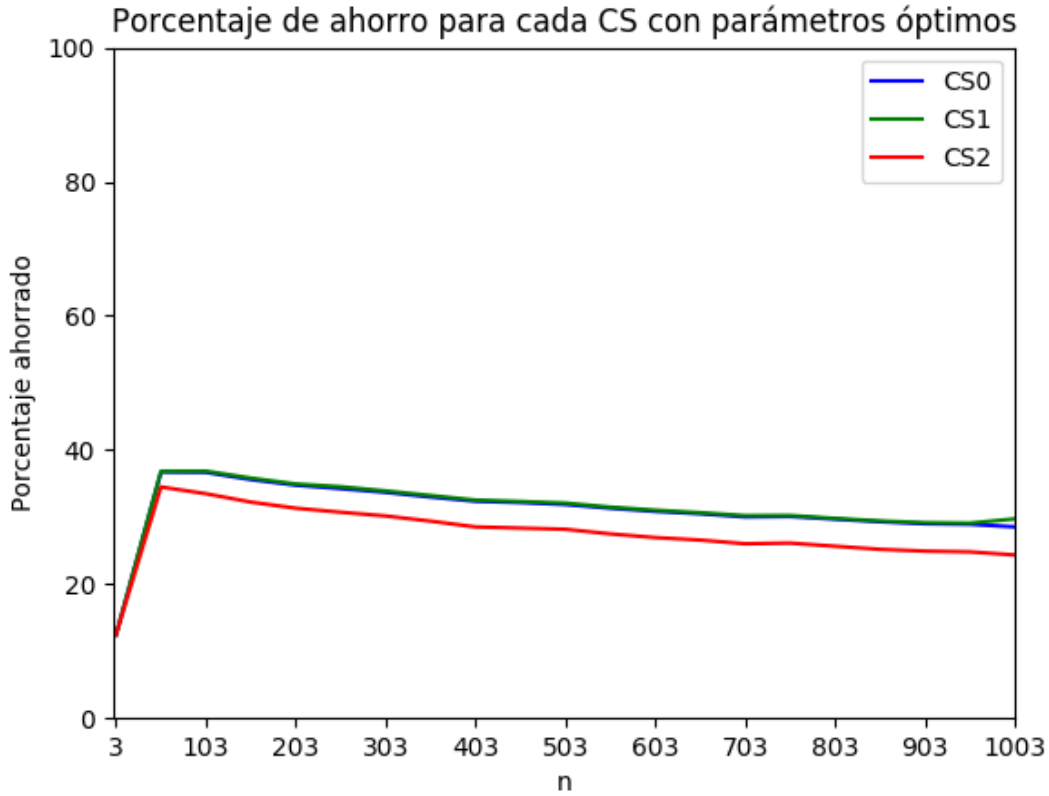
A continuación queremos averiguar el valor óptimo para la temperatura inicial Ts para cada Cooling Schedule. El experimento que realizaremos para encontrarlo será un similar al anterior; promediaremos el porcentaje de ahorro de 400 instancias para cada valor de temperatura inicial desde 2 hasta 202 (haciendo saltos de a 10), para cada Cooling Schedule. El tamaño de las instancias será de un n fijo (usaremos varios tamaños diferentes) y *Nit* = 10000 acorde a los resultados previos. Presentamos a continuación los resultados:





Como podemos observar, la temperatura inicial Ts óptima para las tres alternativas de Cooling Schedule propuestas es Ts = 2 para todos los tamaños de grafo evaluados. Como parece respetarse esto independientemente del tamaño del grafo, podemos afirmar con bastante seguridad que Ts = 2 es óptimo.

Habiendo obtenido los valores óptimos para *Ts* y *NIt*, realizemos la comparación entre el porcentaje de ahorro de las tres funciones Cooling Schedule en funcion del tamaño de la instancia. Realizando saltos de a 50, para cada n desde 3 hasta 1003 calculamos el ahorro promedio de 400 instancias de tamaño *n*, con *Ts* = 2 y *NIt* = 10.000. Estos son los resultados obtenidos:



Se puede ver que *CS0* y *CS1* se comportan de manera identica excepto por los mayores n en los

que *CS1* resultó un poco mejor. *CS2* resultó consistentemente en un peor ahorro, por lo que no sería nuestra elección de Cooling Schedule para una instancia aleatoria de la que no tenemos información.

Es importante destacar que a medida que aumenta n, todos las variantes de Cooling Schedule bajan en efectividad. Esto puede deberse al número de iteraciones, que como vimos anteriormente, debería ser mayor para dar mejores resultados en instancias de mayor tamaño.

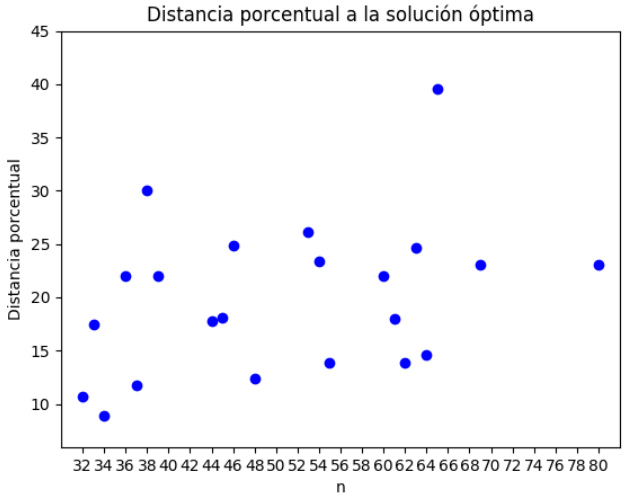
3.4.4. Búsqueda de parámetros óptimos: Set A

TODO: Si antes se aclara cómo medimos, esta aclaracion esta al pedo, puedo referenciar a lo otro.

Busquemos ahora los parámetros óptimos para el set de instancias "Set A" propuesto por Augerat en 1995. Consta de aproximadamente 20 instancias pequeñas que poseen entre 30 y 80 puntos distribuidos aleatoriamente, muy similar la generadas por nuestro generador. Al ser instancias en las que se conoce el óptimo, en vez de medir los resultados como ahorro porcentual de la solución canónica los mediremos como distancia porcentual de la solución óptima; es decir, cuánto porciento mayor es nuestra solución comparada a la solución óptima. Esta manera de medir los resultados nos resulta más significativa que el ahorro porcentual de la solución canónica , que si bien es una medida aceptable para los casos aleatorios porque no tenemos información del óptimo, no es una medida de qué tan buena es la solución con respecto al mejor resultado posible.

La búsqueda de los parámetros óptimos se realizó calculando la distancia porcentual al óptimo para cada instancia del set, para cada variante de Cooling Schedule, para *NIt* desde 1000 hasta 15000 con saltos de a 500, y con cada *Ts* entero desde 2 hasta 100. Para cada configuración se calculó la suma del porcentaje ahorrado para cada instancia y se escogió la configuración que maximize esa suma. Elegimos la suma del porcentaje total ahorrado como medida de optimalidad por encima de la minimizacion de la suma directa de los resultados ya que esta última favorece configuraciones buenas para instancias de mayor tamaño. Con nuestra medida de optimalidad estamos valuando todas las instancias con el mismo nivel de importancia.

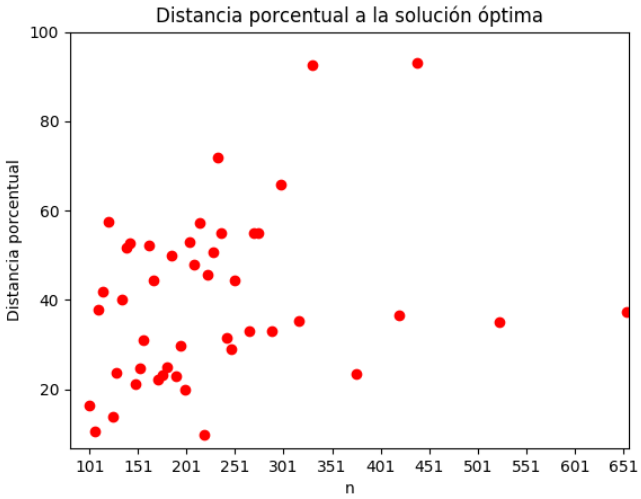
La configuración óptima resultante fue como Cooling Schedule CS1, como temperatura inicial Ts = 8 y número de iteraciones NIt = 15000. Veamos a continuación la distancia porcentual al óptimo para cada instancia del set con la configuración óptima:



3.4.5. Búsqueda de parámetros óptimos: Set X

Busquemos los parámetros óptimos para el set de instancias "Set X" propuesto por Uchoa et al. en 2014. Lo haremos de manera análoga a como lo hicimos para el Set A previamente.

Los resultados obtenidos fueron como Cooling Schedule *CS1*, *Ts* = 20 y *NIt* = 15000. Presentamos a continuación la distancia porcentual al óptimo para cada instancia del set con ésta configuración:



3.4.5. Conclusiones de la experimentación

De la búsqueda de parámetros óptimos para diferentes sets de instancias podemos extraer una serie de conclusiones acerca de como varía el comportamiento del algoritmo en función de los parámetros de entrada. En primer lugar es importante destacar el rol del número de iteraciones en el rendimiento del algoritmo. Si bien lo mencionamos previamente y es bastante sencillo entender su

relevancia, no podemos dejarlo fuera de las conclusiones. No creemos que sea casualidad que los mejores resultados para todos los sets de instancias utilizados se obtuvieron para el mayor número de iteraciones probado.

Sin embargo es importante destacar que si bien a mayor número de iteraciones se tiende a obtener mejores resultados, esto es una tendencia y no se cumple para todas las instancias. Buscando los parámetros óptimos para cada instancia encontramos muchos ejemplos en lo que no se cumplía. Esto no inválida la tendencia pero es importante tenerlo en cuenta a la hora de utilizar el algoritmo para la resolución de instancias. Si se utiliza una única instancia maximizar el número de iteraciones podría no resultar en el mejor rendimiento, pero si se quiere obtener el mejor rendimiento para una serie probablemente sí resulte en él.

Otra conclusion es que el parámetro Ts afecta de manera distinta a la solución obtenida en función del Cooling Schedule que se utilize. Como podemos ver en el gráfico "Porcentaje de ahorro para diferentes valores de Ts", la variación de Ts afecta a los tres Cooling Schedule de manera muy distinta. CS0 decrece en efectividad linealmente en función de Ts, CS2 decrece rápidamente para algunos valores y luego aparenta mantenerse constante y CS1 no parece verse afectado en lo absoluto.

Es importante mencionar la baja efectividad del algoritmo para grafos de gran tamaño. Si bien los grafos de mayor tamaño necesitan un mayor número de iteraciones para dar soluciones de igual calidad que grafos menores (Como muestra la serie de gráficos "Porcentaje de ahorro para N iteraciones"), tenemos otra hipótesis acerca de este fenómenos. Esta es que el algoritmo en grafos de mayor tamaño es más propenso a diverger a malas soluciones dado que su vecindario es mayor y es menos probable dar pasos en la dirección del óptimo o de un vecindario de buenas soluciones. La raíz de este problema es que el algoritmo no tienen ninguna forma de volver hacía soluciones previas; una vez que da un paso, aunque sea un paso que empeore nuestra solución, no hay vuelta atrás.

Una alternativa para solucionar este inconveniente es implementar Resets o reinicios. La idea general es tener un criterio de reinicio que nos indica cuándo es momento de volver a una solución previa, ya que la actual es mucho peor que ésta. Generalmente la solución a la que se vuelve es la mejor solución que hemos encontrado en toda la ejecución del algoritmo hasta este punto. Esto es meramente una hipótesis y debería experimentarse y confirmarse debidamente para poder hacer afirmaciones al respecto.

4. Conclusión

TODO: Llenar esto. Comparar todos los casos.