1. Introducción

1.1. CVRP

TODO: Explicar los problemas. Dar ejemplos y soluciones.

1.2. CVRP en la vida cotidiana

TODO: Hablar de casos donde se pueda usar CVRP para modelar casos de la vida cotidiana

1.3. Problemas a resolver

TODO: Hablar de los casos de test y de nuestros casos random

1.4. Objetivos

TODO: Solucionar los casos de la cátedra y solucionar casos random

2. Herramientas y equipo

2.1. Equipo utilizado

TODO: Equipo utilizado en las mediciones de cada punto

2.2. Generador de instancias

TODO: Explicar cómo generamos instancias

2.3. Generador de clusters

TODO: Copypastear el generador de clusters del TP2

3. Soluciones Aproximadas

3.1. Introducción

TODO: Explicar qué hacemos con estas soluciones y eso

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

TODO: Explicar esta heurística

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 != vacio):
5      (a,b) = pares.primero()
6      rutaA = rutaALaQuePertenece(a)
7      rutaB = rutaALaQuePertenece(b)
8      if(sumatoriaDemandas(rutaA) + sumatoriaDemandas(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 union entre las rutas une el último elemento no depósito de la rutaA con el primer elemento no depósito de la rutaB y luego descarta estos depósitos de manera que la ruta obtenida sea una ruta válida. Tampoco es menor enfatizar que no importa que nodos conformen el par más cercano entre sí; la unión será siempre entre el último nodo de la ruta del primer elemento y el primer elemento de la ruta del segundo elemento.

3.2.2. Análisis de complejidad temporal

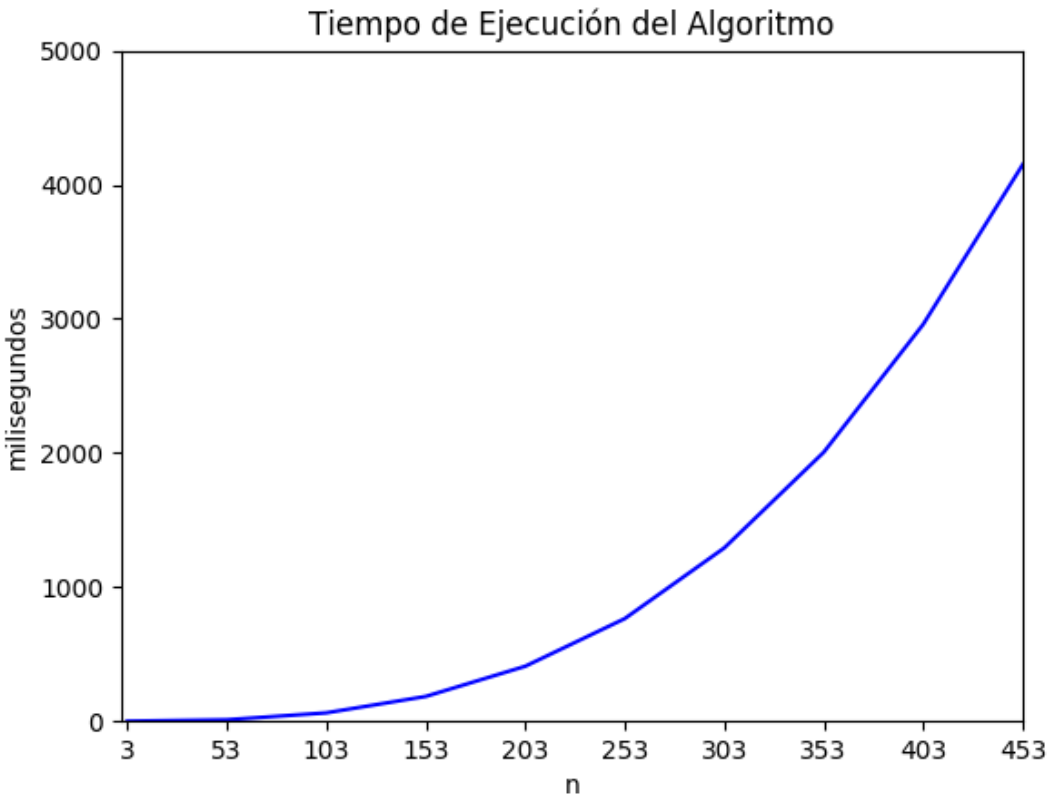
Descompongamos el pseudocódigo en pasos:

- 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)).
- 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)

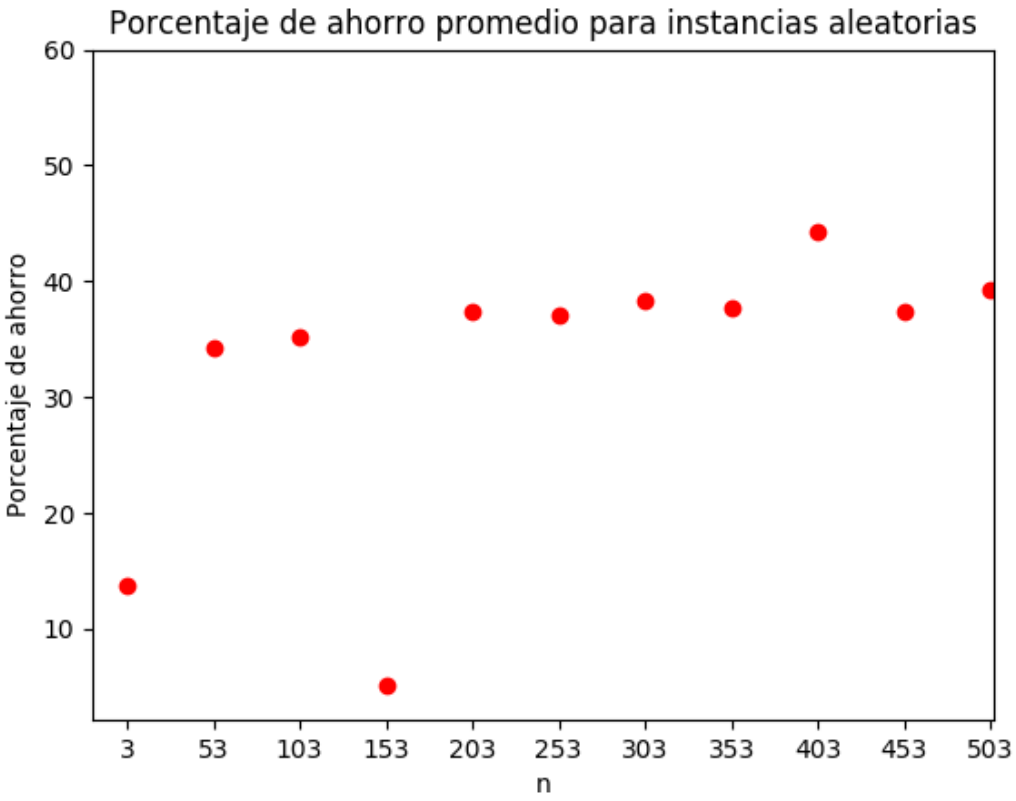
Comprobemoslo realizando las mediciones directamente. Generamos 400 instancias de grafo de para cada tamaño desde n = 3 hasta n = 453, con saltos de a 50 y promediamos el tiempo obtenido. Los resultados se pueden ver a continuación:

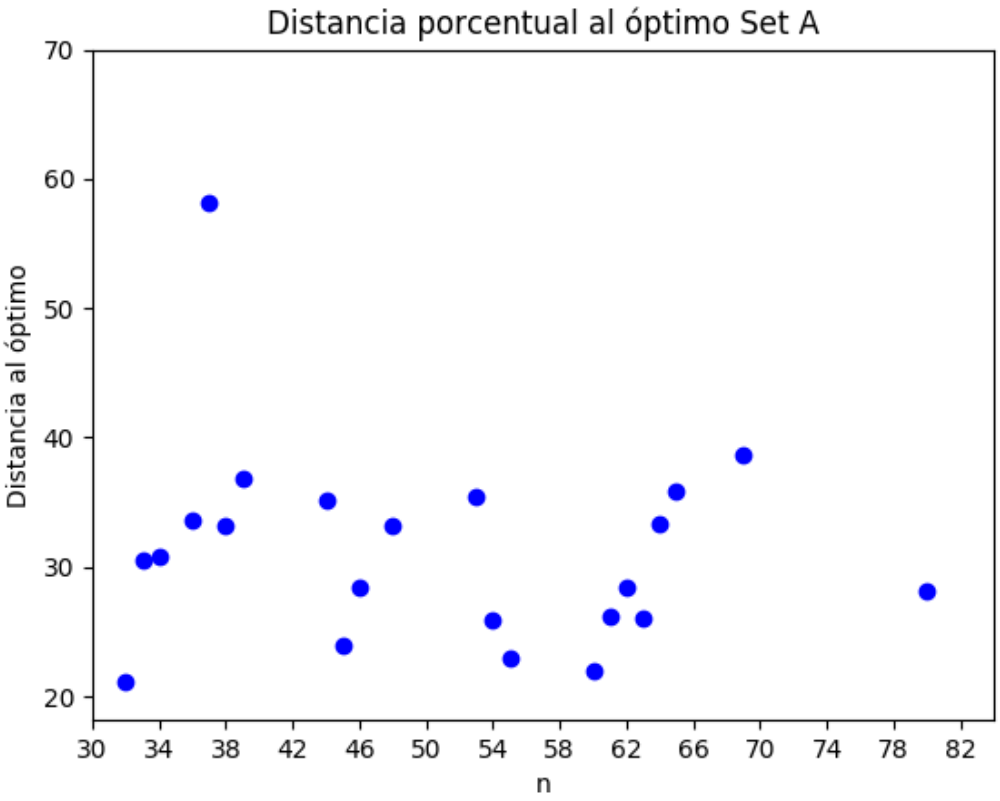


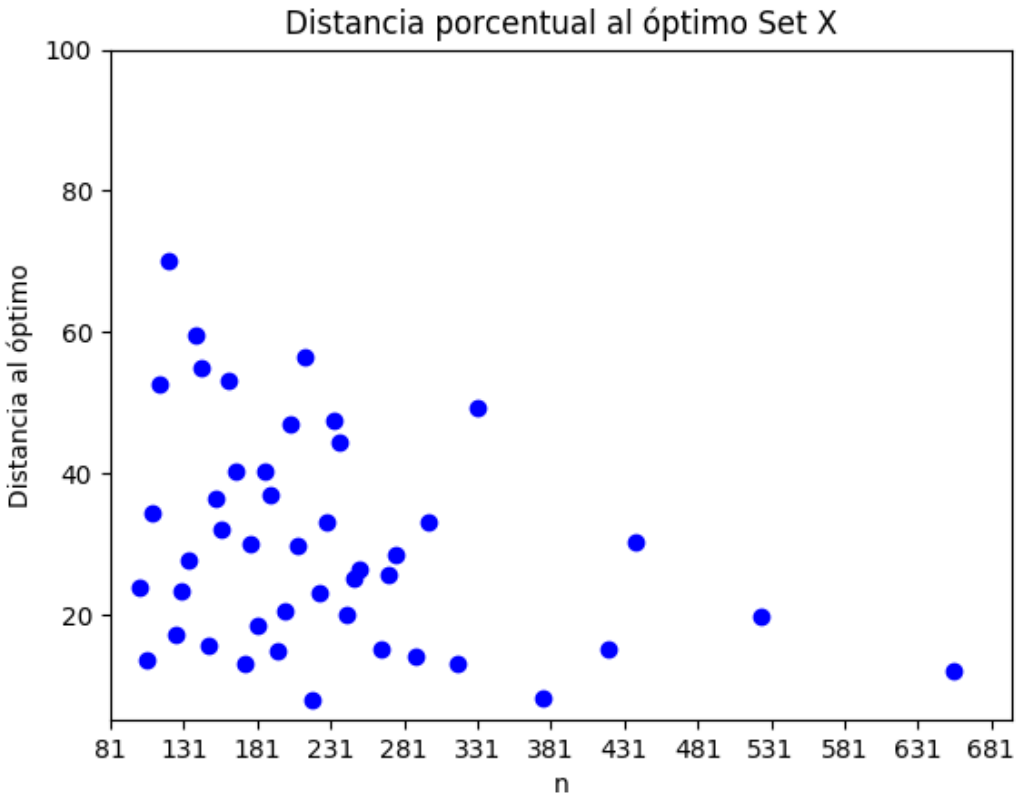
3.2.3. Rendimiento para diferentes sets de instancias

TODO: hablar de los resultados

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.


Porcentaje de ahorro promedio para instancias aleatorias
Porcentaje de ahorro
60
50
40
30
20
10
3
53
103
153
203
253
303
353
403
453
503
n


Distancia porcentual al óptimo Set A
Distancia al óptimo
n
70
60
50
40
30
20
30
34
38
42
46
50
54
58
62
66
70
74
78
82



TODO: Explicación detallada y ejemplo del caso patológico

3.2.4. Caso patológico

Detengámonos a analizar un caso de Grafo para el cual la heurística propuesta resulta en malas soluciones. La posibilidad de tener casos patológicos se debe al hecho de que si bien estamos utilizando los pares de clientes más cercanos, 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 una arista muy larga.

3.3. Cluster-First & Route-Second

TODO: Poner bien los títulos. Usar solo títulos de tres asteriscos. Dos asteriscos es texto en negrita.

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 [1], el cual clusteriza transformando el grafo original en un AGM y chequeando que si un eje de nuestro AGM es "mucho 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.

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)

```
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 deposito, 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(n^2) en el caso de que tuviesemos un solo cluster.

```
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 cuadrático para cualquier caso. Si el grafo quedara clusterizado en un solo conjunto, estariamos resolviendo tsp en tiempo cuadrático para luego calcular el costo total del problema en tiempo lineal por lo que esta función en peor caso tarda O(n^2). De no pasar esto, y tener clusters separados, el proceso de ordenar todos los nodos en O(n log n) ganaría y esto sería lo que tardaría nuestro algoritmo.

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.

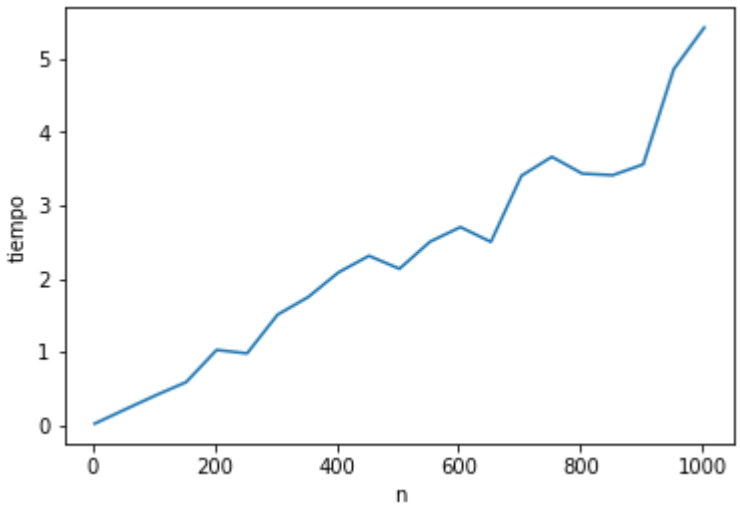
Complejidad: En el caso de que solo tengamos un cluster armardo, vamos a clusterizar en O(n^2) además de copiar la matriz en el mismo tiempo. Eliminar el depósito nos va a tomar tiempo lineal (es recorrer todos los nodos del grafo hasta encontrar el deposito) y luego vamos a aplicar tsp en tiempo cuadrático.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

3.3.3. Experimentación

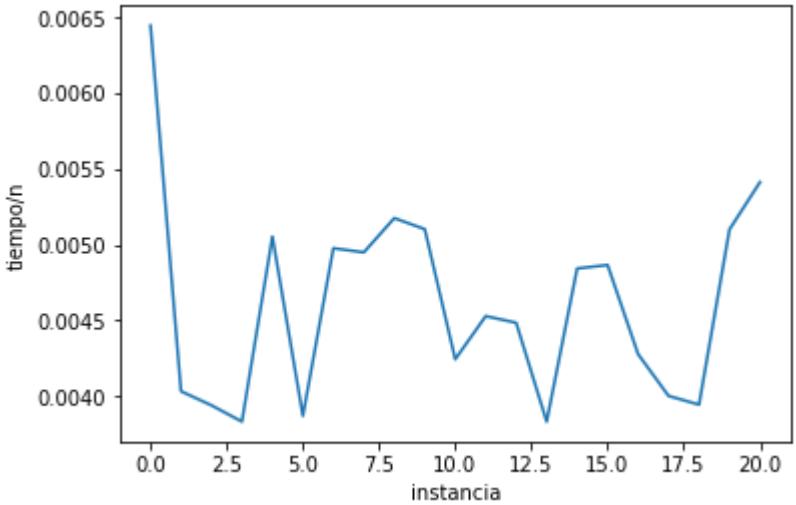
Perfomance:

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.

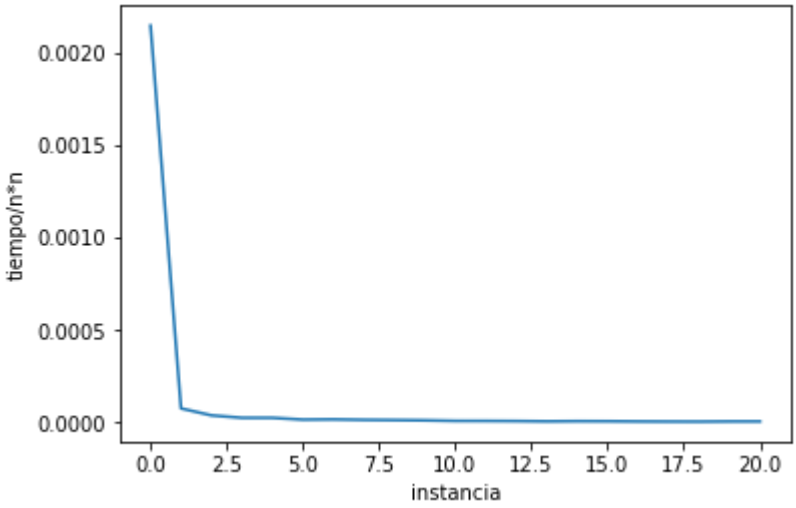
Sweep:



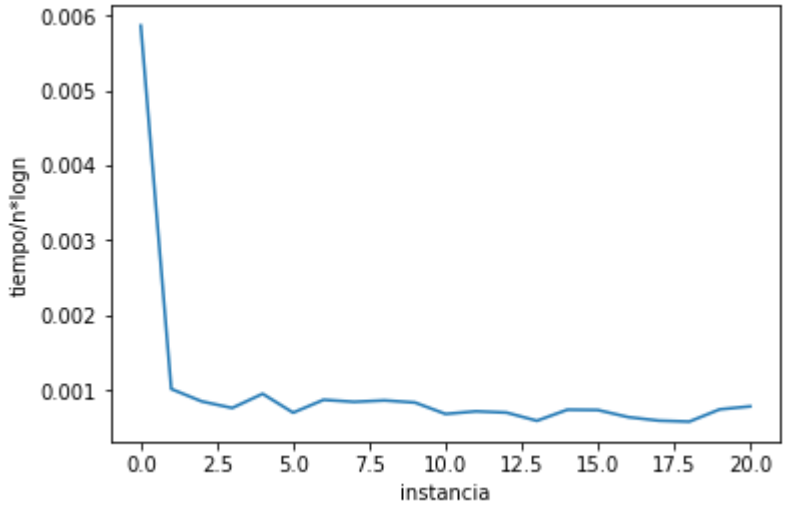
Podemos ver que el algoritmo a simple vista tiene el aspecto de tomar tiempo lineal. Esto sería un problema ya que estaría falsando fuertemente nuestra hipotesis inicial de que pertenece a O(n^2). Ahora al dividir por una función lineal vemos que no resulta en una función constante



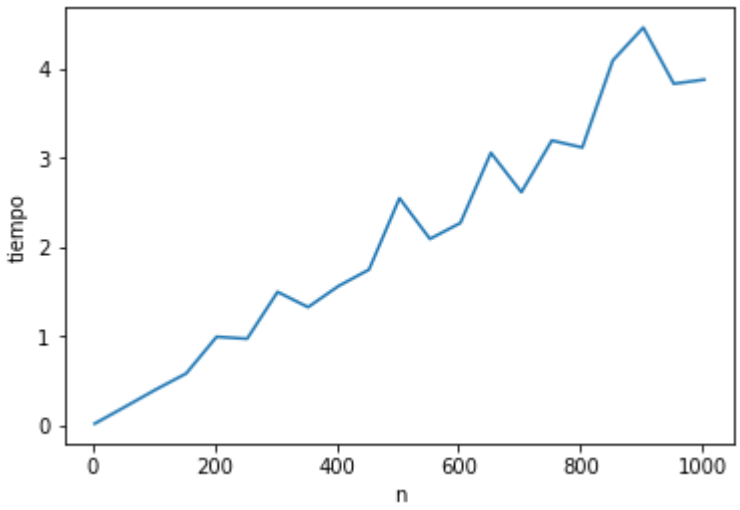
así que podemos descartar esta posibilidad. Por otro lado vemos que al dividir el algoritmo por una función cuadrática obtenemos el resultado esperado,



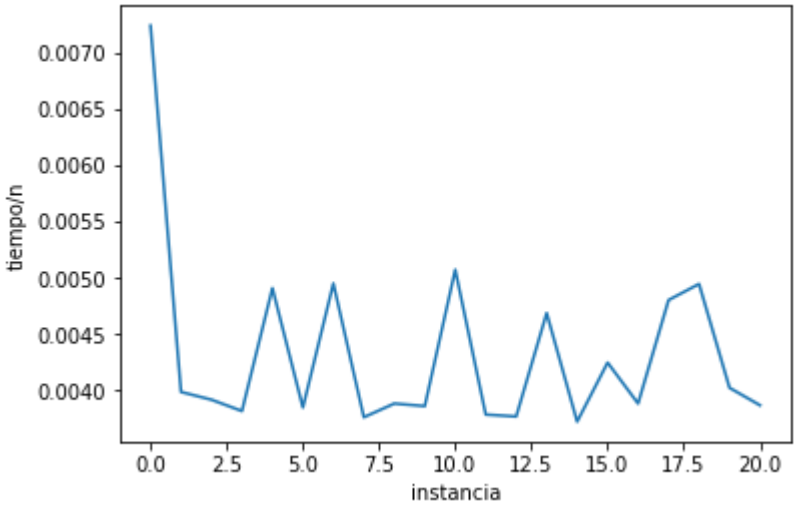
pero podemos acotar la complejidad un poco más, ya que el peor caso de nuestro algoritmo se da cuando nos queda todo el grafo en un solo cluster y esto en realidad no pasa usualmente, asi que dividiendo por n log n podemos ver que nuestra función resulta en una constante asi que en el caso promedio nuestro algoritmo toma O (n log n).



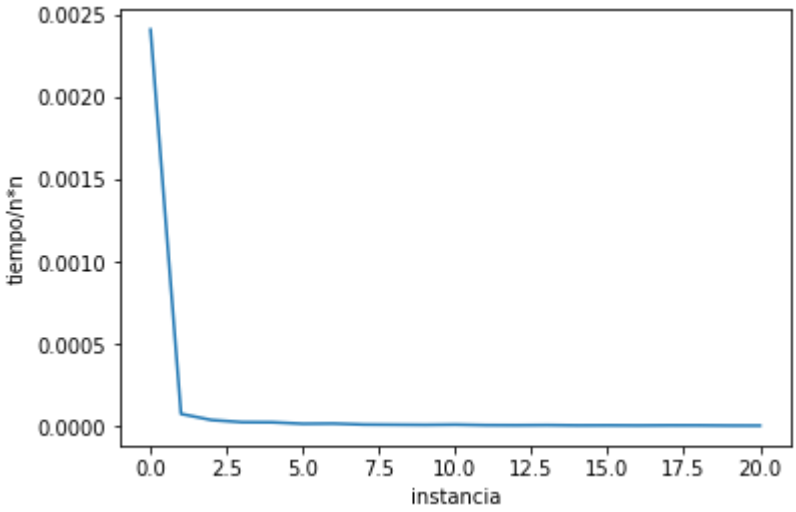
Ejes inconsistentes:



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. 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 (siempre y cuando el agregado de camiones no sea un problema). Podríamos tener en cuenta esta situación en la implementación para evitar que pase, aunque esto queda a criterio del programador que realice esta tarea.

```
TODO: insertar imagen
```

Este grafo es un ejemplo del caso patológico mencionado. Si por ejemplo, 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. 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).

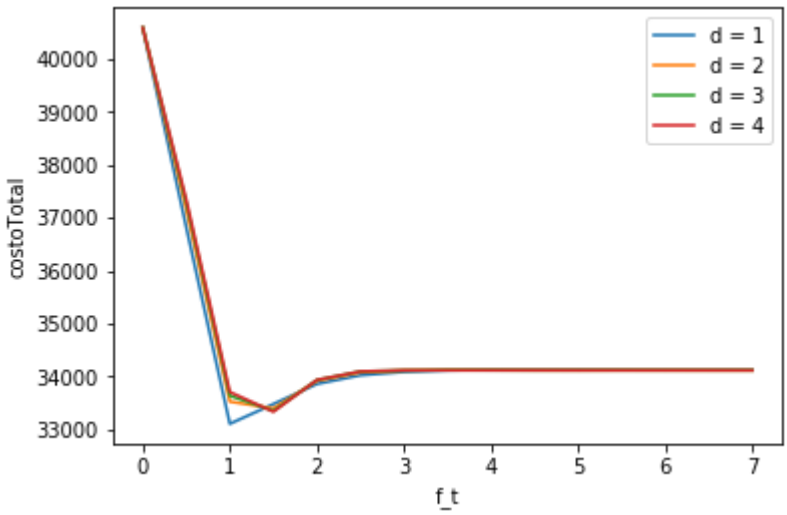
Procedemos a mostrar una tabla con los mejores parámtros encontrados para el algoritmo de ejes inconsistentes. La mismas tiene 3 columnas, la primera indica el dataset en el que fue calculado, la segunda el mejor f_t encontrado y la ultima el mejor d.

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

Caso aleatorio

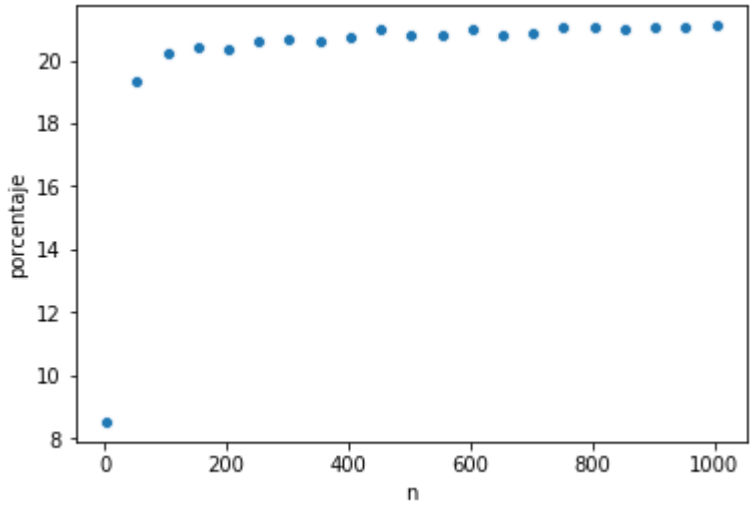
En este caso los algoritmos fueron corridos con instancias de tamaño 1 al 100 habiendo 400 instancias de tamaño 1 <= n <= 100.

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.



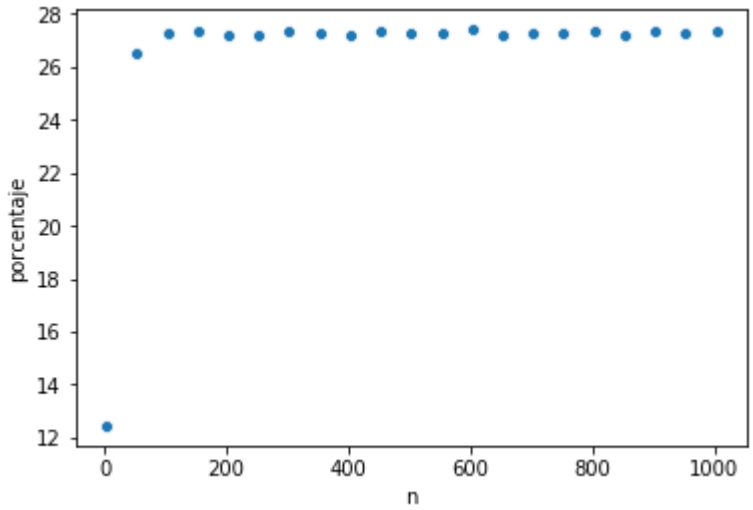
Como podemos observar los d no varian demasiados unos de los otros pero hay una configuración que a simple vista minimiza los costos siendo la misma d = 1 y f_t = 1. Vamos a utilizar estos para medir la perfomance de nuestro algoritmo en términos de costos.

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.



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.

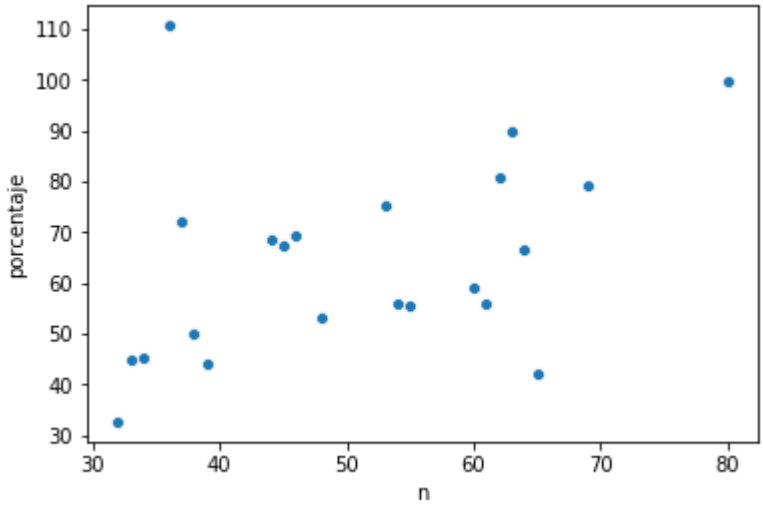


Podemos observar que sweep da soluciones un poco mejores que EI llegando casi a un ahorro del 30% en la mayoría de los casos.

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:

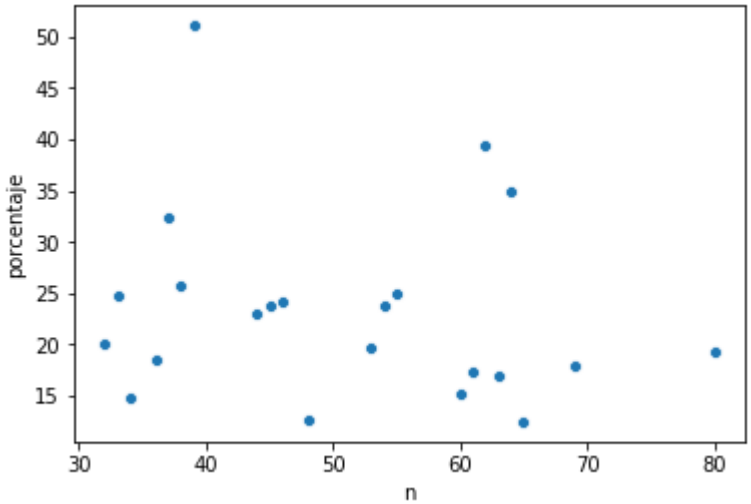
Caso A:

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.



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.

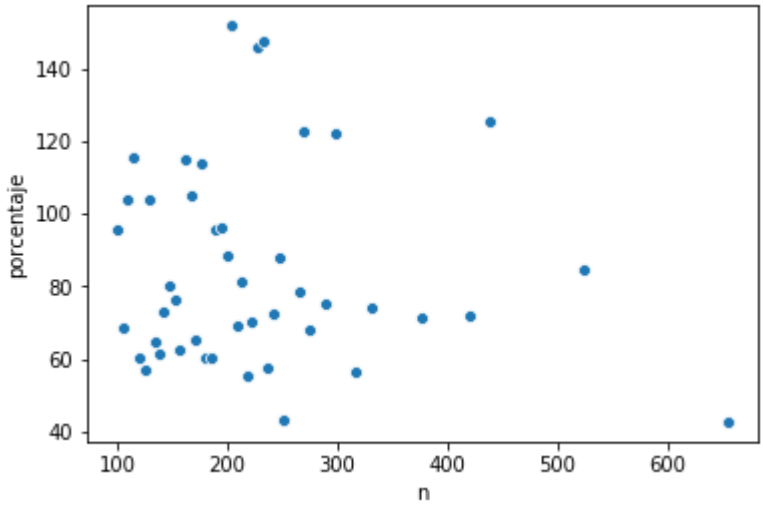
Sweep: En el caso de sweep la mayoría de las soluciones están a una distancia de entre un 15% a un 25% de la solución óptima promedio encontrada para el dataset correspondiente.



Caso X:

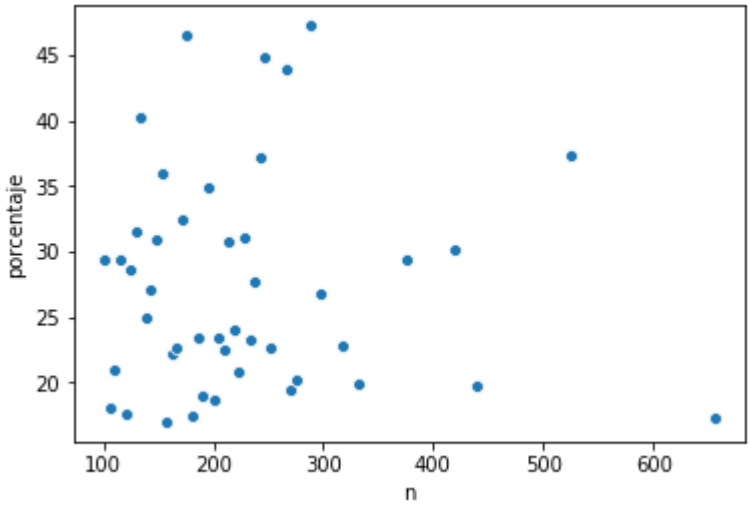
Nota: los criterios usados para la medición de este caso son idénticos a los del caso A con la salvedad de que se usan los parámetros correspondientes previamente calculados.

Ejes inconsistentes:



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

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.

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 mas 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 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 a costas de usar más camiones nos de una solución mejor (siempre que el uso de muchos 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 **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:

```
TODO: poner la imagen del pseudocodigo de las diapos
```

```
TODO: Esto asume que ya explicamos como está compuesta una solucion. O sea, que es un
vector de vectores de clientes.
```

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

TODO: se puede escribir esto de manera mas linda (a nivel formato)

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

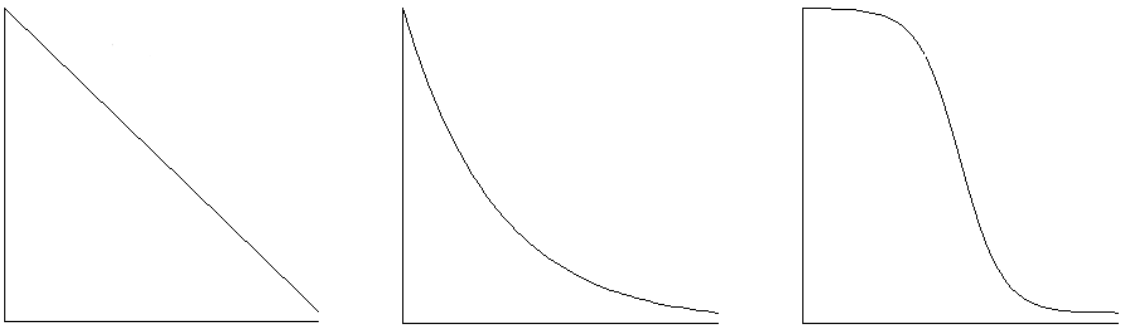
Siendo `shift` y `exchange` los siguientes procedimientos:

```
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
```

siendo shift 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) siendo 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.

Demos un pseudocódigo completo para nuestra implementación de Simulated Annealing:

```
TODO: Dar pseudocódigo completo (obviamente formal, pero usando P, CS y eso con sus valores reales)
```

3.4.2. Analisis de complejidad

```
TODO: explicarlo mejor con el pseudocódigo. No hay cota simple mas ajustada no?
```

Dado que todas las operaciones se realizan en tiempo constante excepto 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 siga teniendo espacio), pero dar una cota más ajustada es complejo porque depende de la instancia particular.

```
TODO: Mostrar los graficos que muestran que el algoritmo es lineal en funcion del nro de iteraciones
```

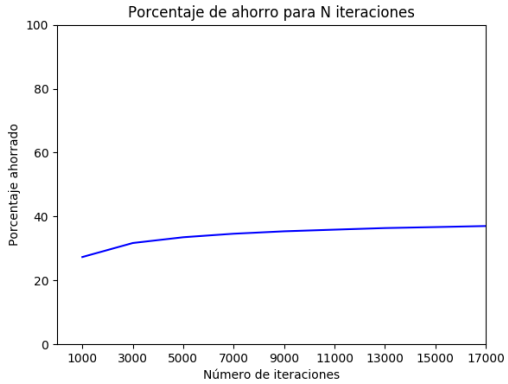
3.4.3. Búsqueda de parámetros óptimos: caso aleatorio

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.

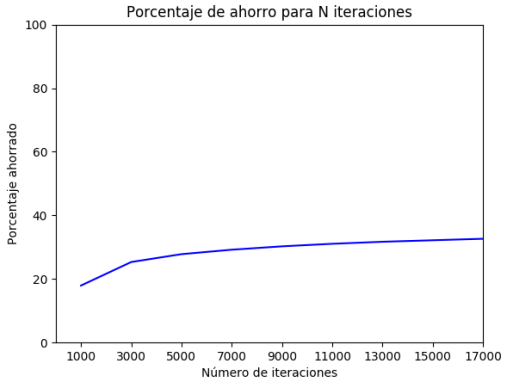
Para esto realizaremos una serie de experimentos, comenzando por un análisis de cómo el número de iteraciones realizadas (*Nit*) afecta el resultado. Sabemos que mientras más iteraciones realizemos mejor será el resultado ya que a fin de cuentas estamos realizando una búsqueda local, pero la pregunta 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?

```
TODO: aclarar como fueron generadas las instancias, supongo que lo voy a aclarar una sola vez!
```

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 diferentes n 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, $n = 103$



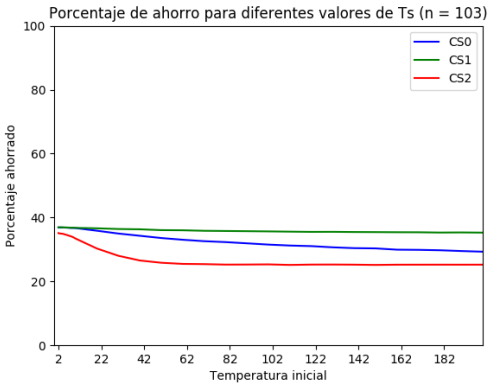
Ahorro porcentual de la solución canónica en función de NIt, $n = 503$caption)

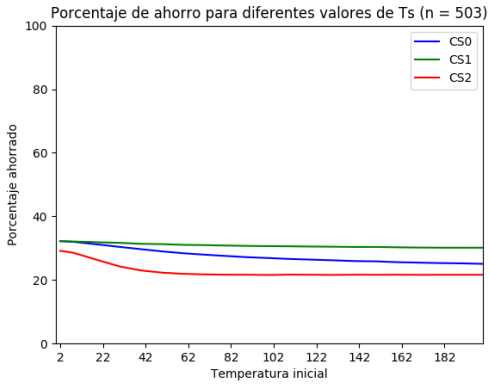
Como podemos apreciar en los gráficos nuestras predicciones fueron correctas. A medida que aumenta *Nit* el porcentaje de ahorro también aumenta. Notemos tambien que en ambos el pocentaje ahorrado aumenta rápidamente hasta las tres mil iteraciones y luego crece más lentamente. Esto se cumple en todos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

TODO: aclarar como son "las instancias"

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.

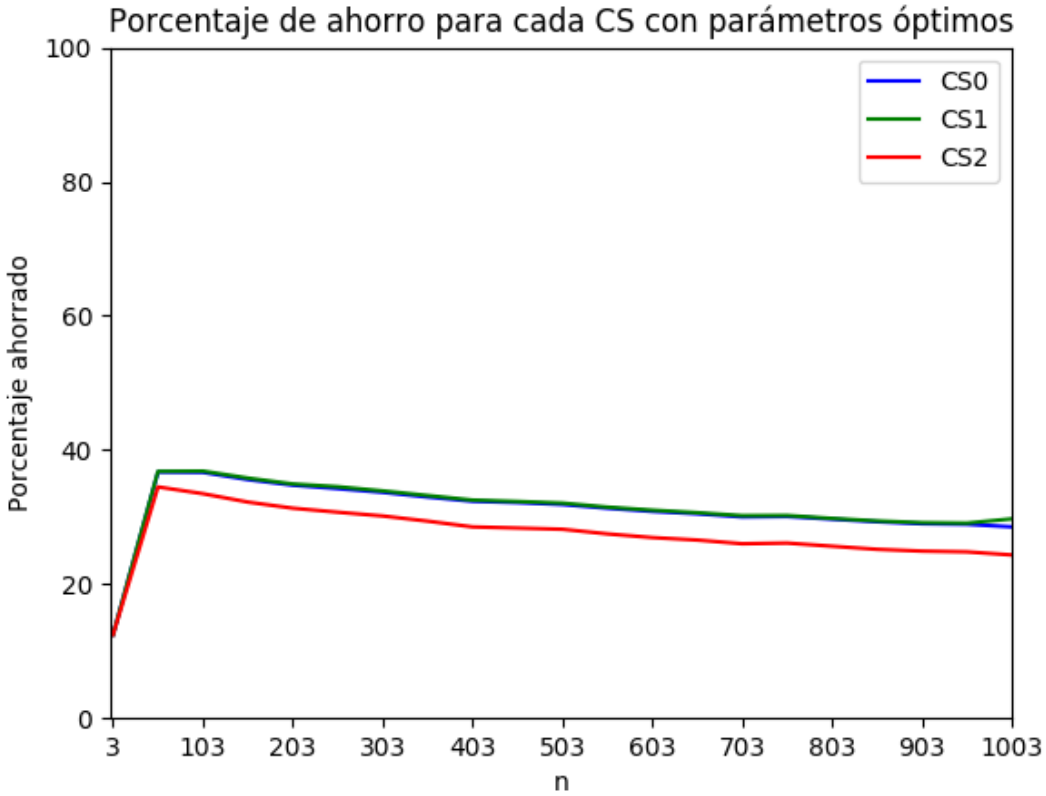
A continuación queremos averiguar el valor óptimo para la temperatura inicial Ts para cada Cooling Schedule. El experimento que realizaremos para encontrarlo será un similar al anterior; promediaremos el porcentaje de ahorro de 400 instancias para cada valor de temperatura inicial desde 2 hasta 202 (haciendo 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.

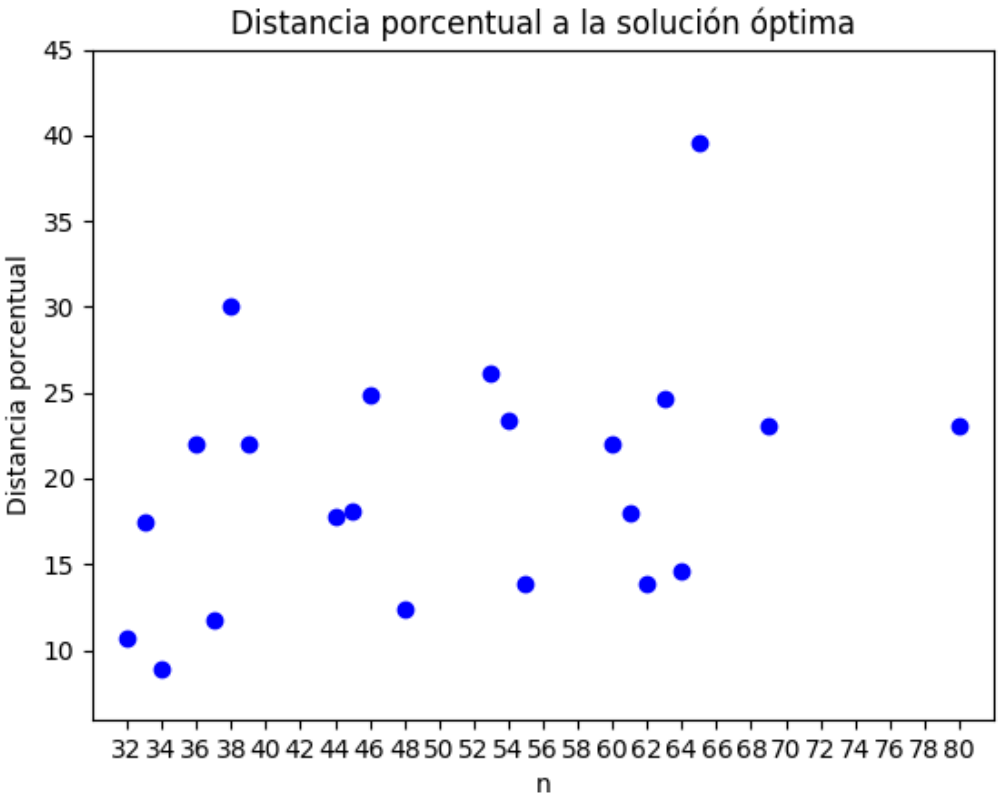
```
TODO: Explicar como son las instancias del set A (a menos que se haga antes en el informe)
```

3.4.4. Búsqueda de parámetros óptimos: Set A

Busquemos ahora los parámetros óptimos para el set de instancias "Set A" propuesto por Augerat en 1995. Son instancias en las que se conoce el óptimo, por lo que 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:

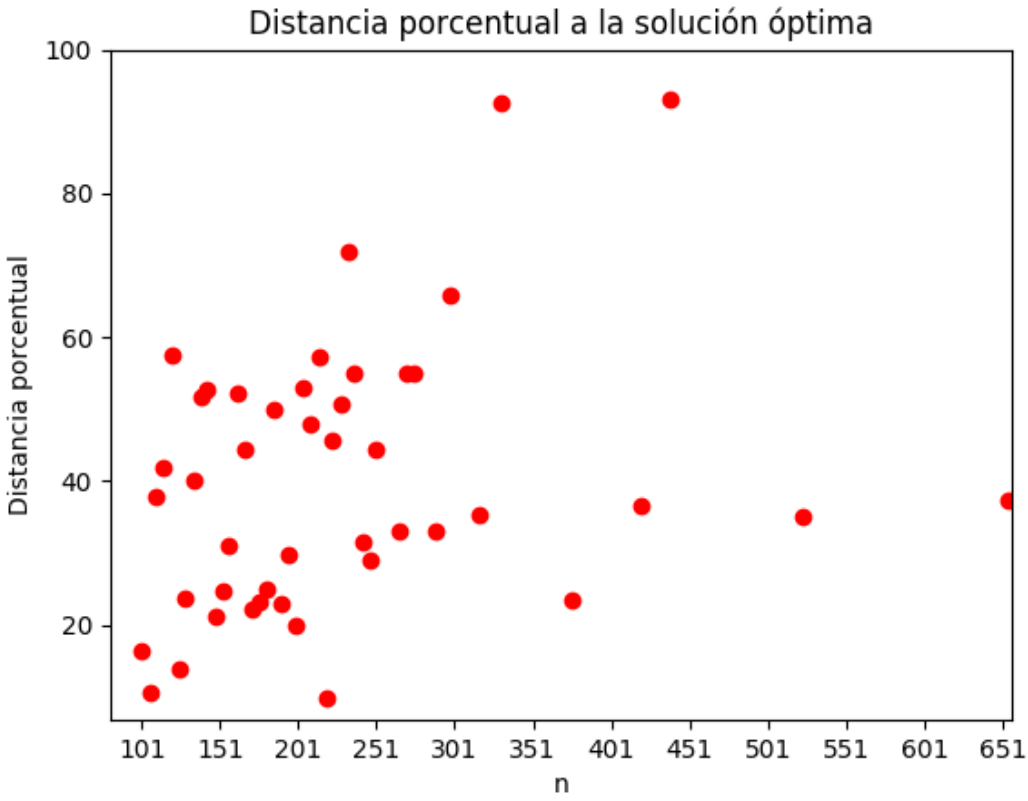


3.4.5. Búsqueda de parámetros óptimos: Set X

TODO: Explicar como son las instancias del set X (a menos que se haga antes en el informe)

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. Comparativa general para un caso desconocido

TODO: Llenar esto

5. Conclusión

TODO: Llenar esto