Cluster-First Route-Second

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.

Pseudocódigos

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

```
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
```

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) ← ???:
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)
```

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) ← ???:
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.

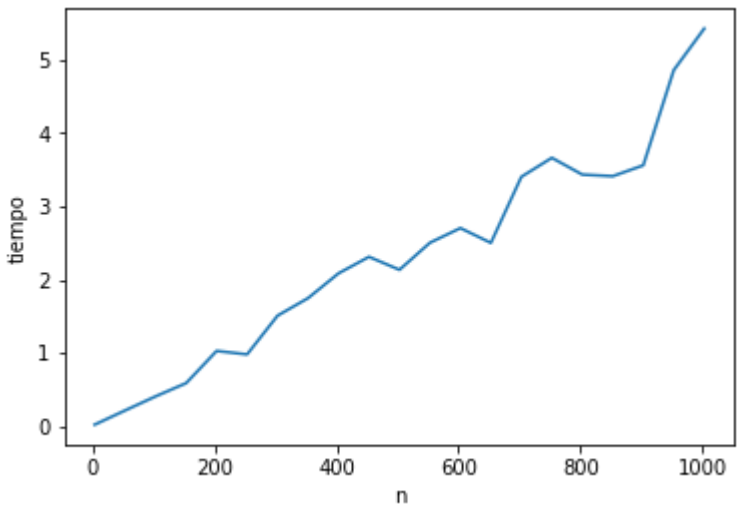
Complejidad:

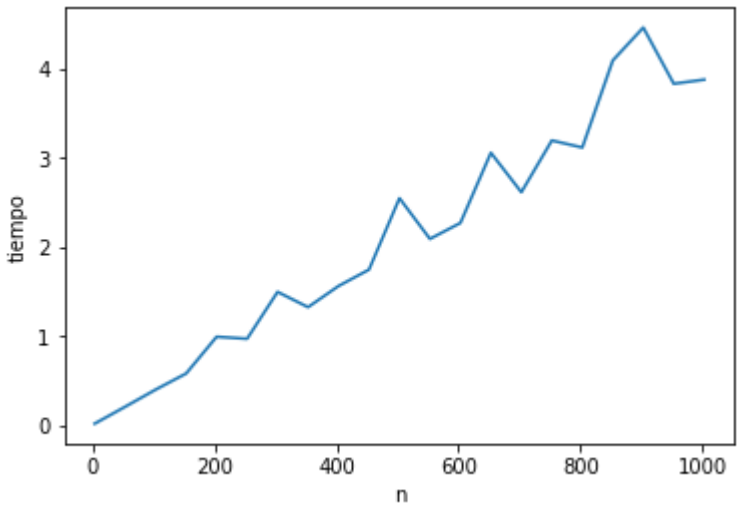
Experimentación

Perfomance:

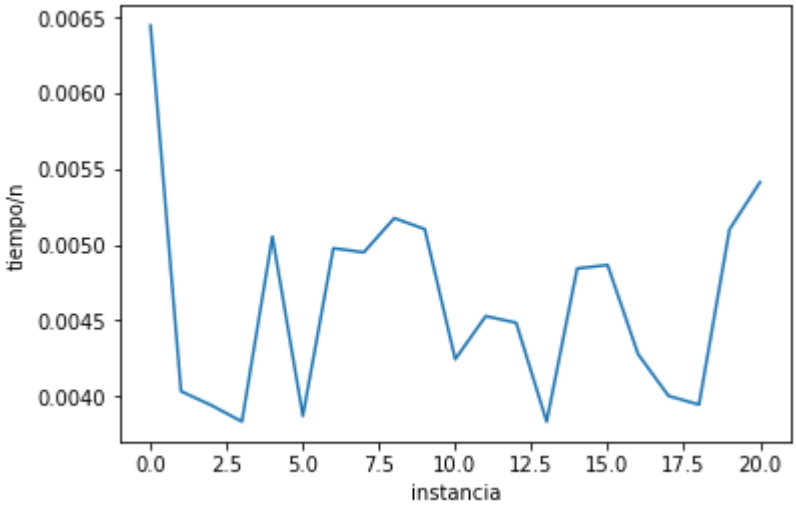
Los dos algoritmos fueron ejecutados con instancias aleatorias de un rango de tamaño entre 3 y 1003 con saltos de tamaño de 50 unidades. Para cada n se corrieron 400 instancias de ese mismo tamaño las cuales fueron posteriormente promediadas.

Sweep:

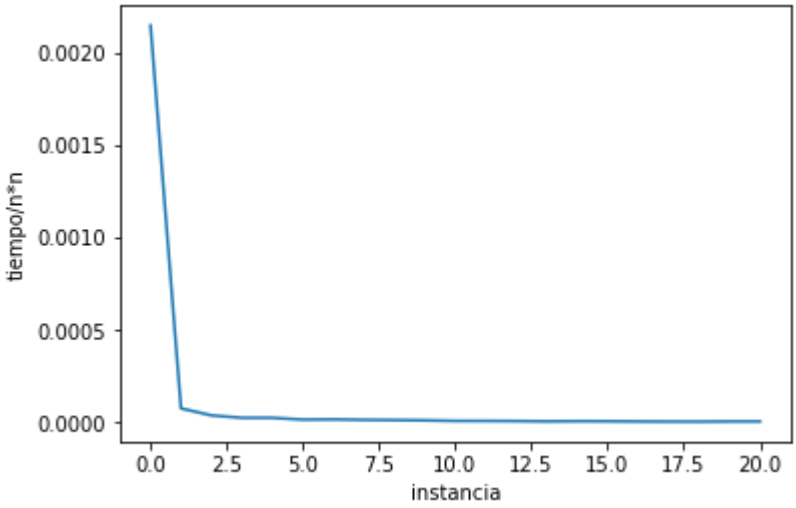




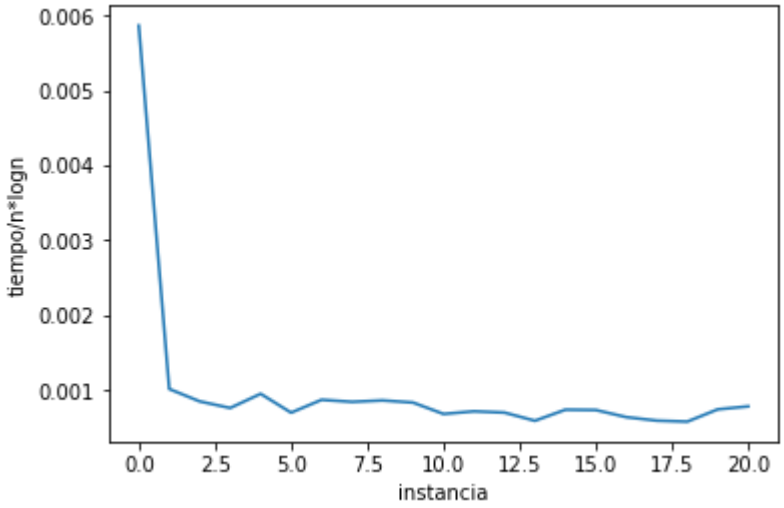
Podemos ver que el algoritmo a simple vista tiene el aspecto de tomar tiempo lineal. Esto sería un problema ya que estaría falsando fuertemente nuestra hipotesis inicial de que pertenece a O(n^2). Ahora al dividir por una función lineal vemos que no resulta en una función constante



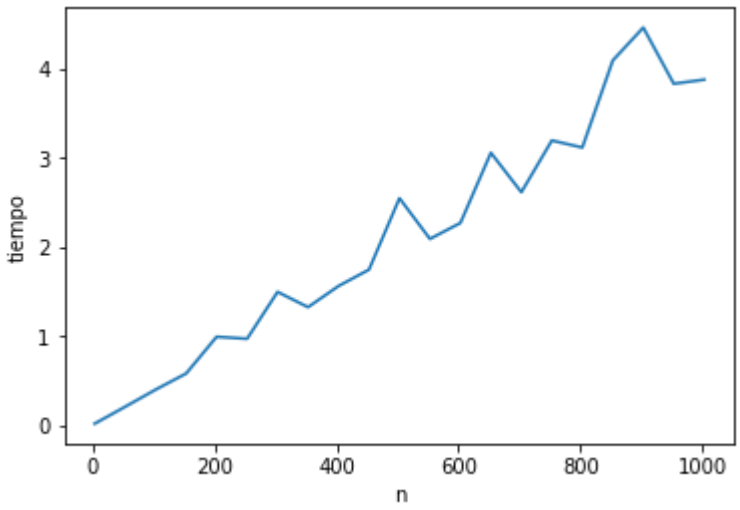
así que podemos descartar esta posibilidad. Por otro lado vemos que al dividir el algoritmo por una función cuadrática obtenemos el resultado esperado,



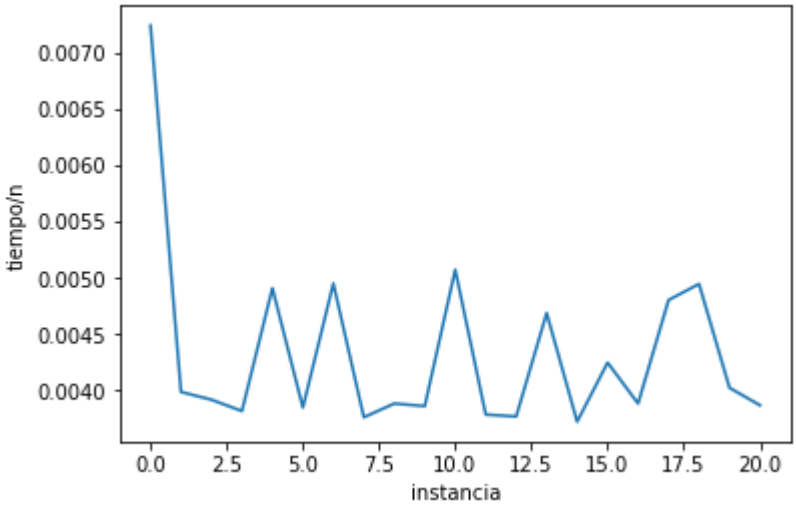
pero podemos acotar la complejidad un poco más, ya que el peor caso de nuestro algoritmo se da cuando nos queda todo el grafo en un solo cluster y esto en realidad no pasa usualmente, asi que dividiendo por n log n podemos ver que nuestra función resulta en una constante asi que en el caso promedio nuestro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



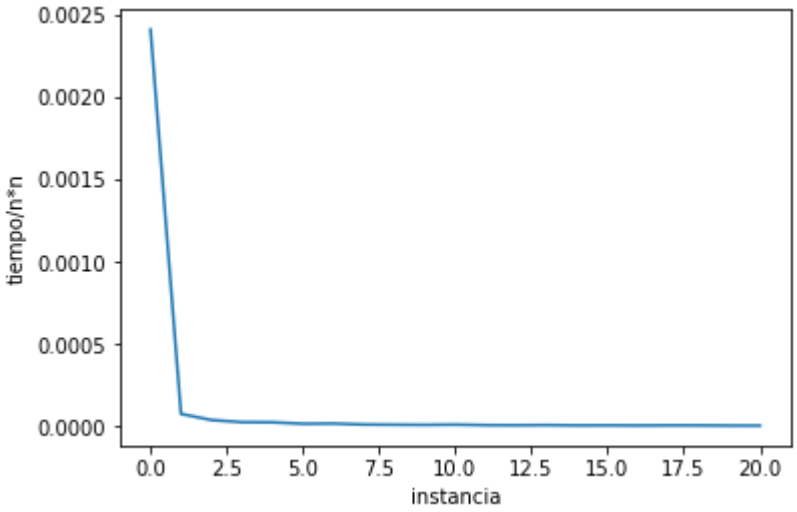
Ejes inconsistentes:



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.

//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

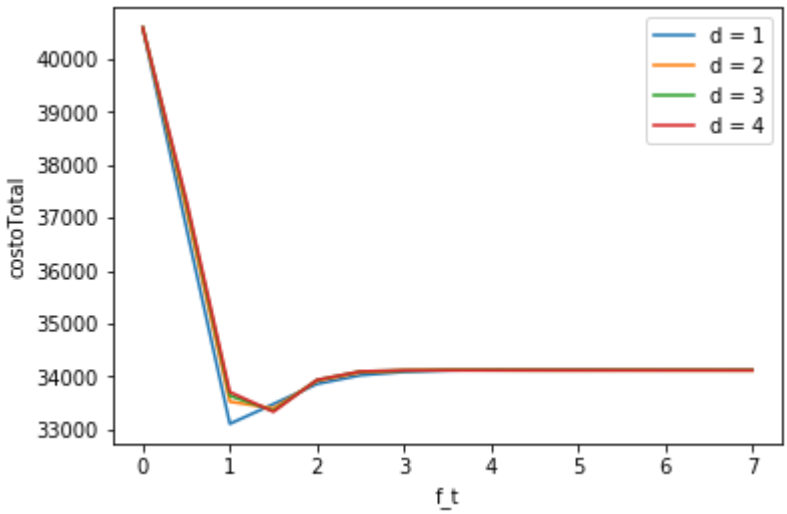
inconsistentes. La mismas tiene 3 columnas, la primera indica el dataset en el que fue calculado, la segunda el mejor f_t encontrado y la ultima el mejor d.

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

Caso aleatorio

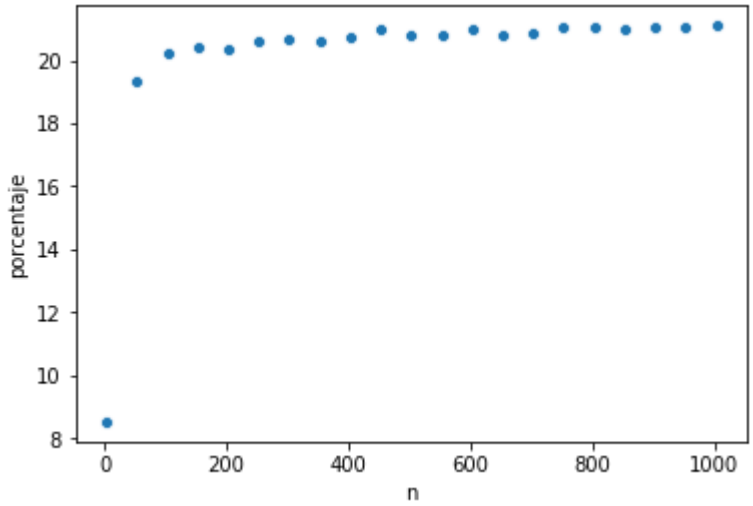
En este caso los algoritmos fueron corridos con instancias de tamaño 1 al 100 habiendo 400 instancias de tamaño 1 <= n <= 100.

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.



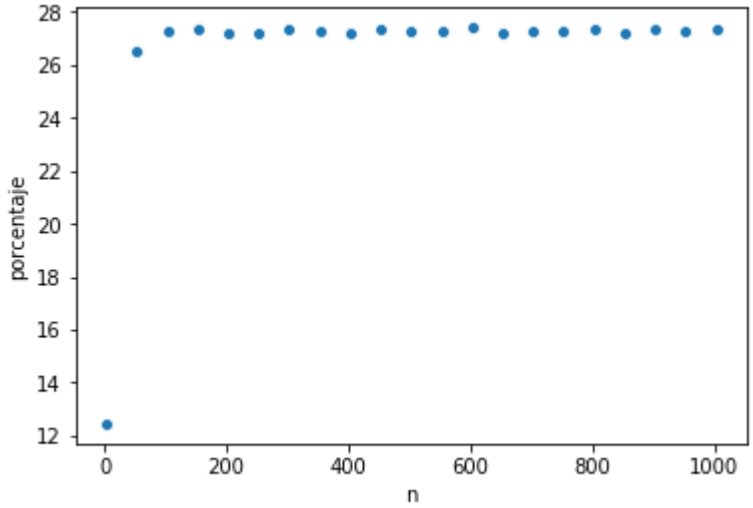
Como podemos observar los d no varian demasiados unos de los otros pero hay una configuración que a simple vista minimiza los costos siendo la misma d = 1 y f_t = 1. Vamos a utilizar estos para medir la perfomance de nuestro algoritmo en términos de costos.

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.



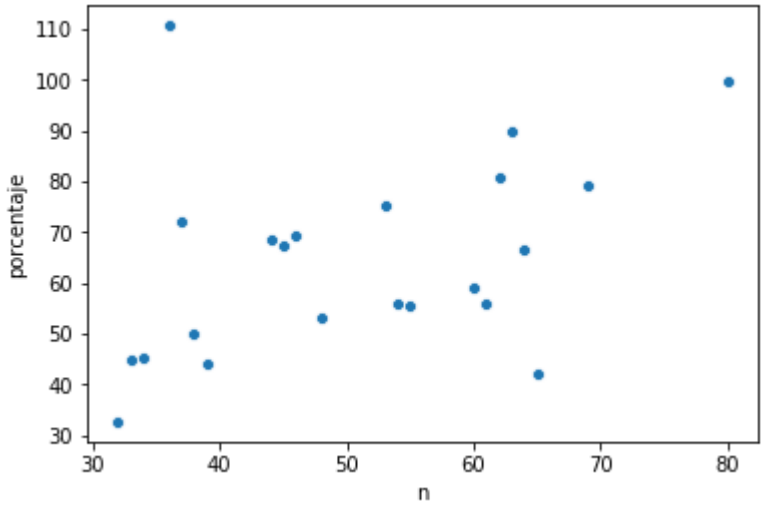
Podemos observar que sweep da soluciones un poco mejores que EI llegando casi a un ahorro del 30% en la mayoría de los casos.

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:

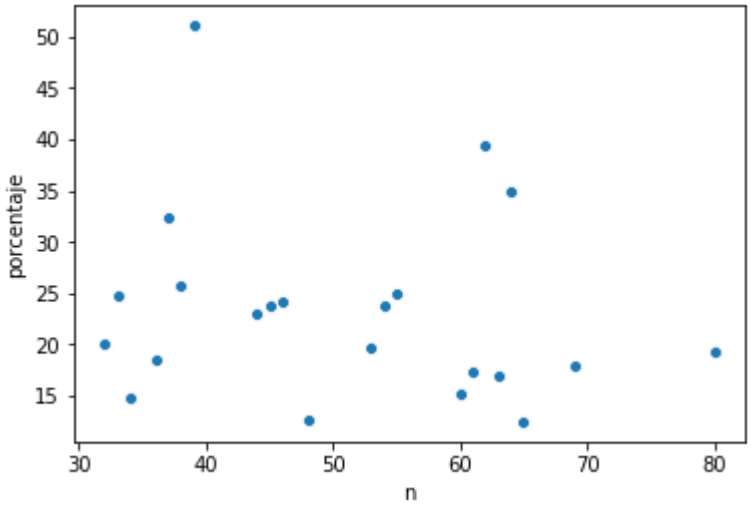
Ejes inconsistentes: Usando los parámetros ya mostrados en la tabla previa medimos los costos promedio pero esta vez lo hacemos midiendo la distancia porcentual al óptimo (dado que en este set

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.

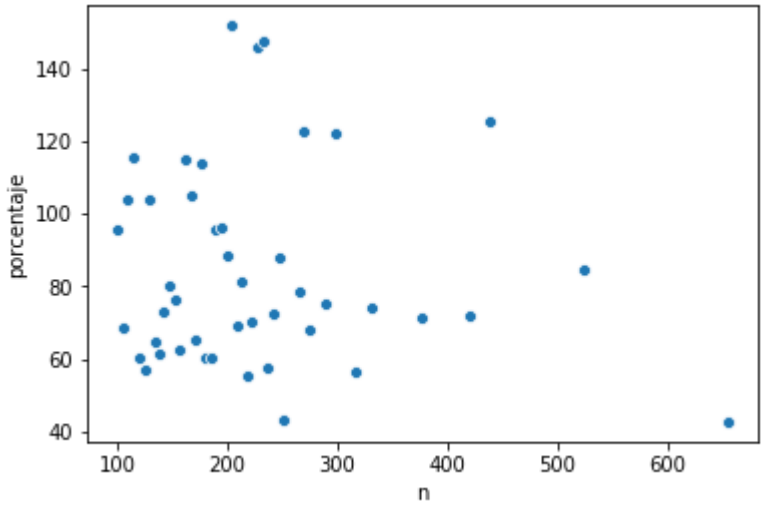
Sweep: En el caso de sweep la mayoría de las soluciones están a una distancia de entre un 15% a un 25% de la solución óptima promedio encontrada para el dataset correspondiente.



Caso X:

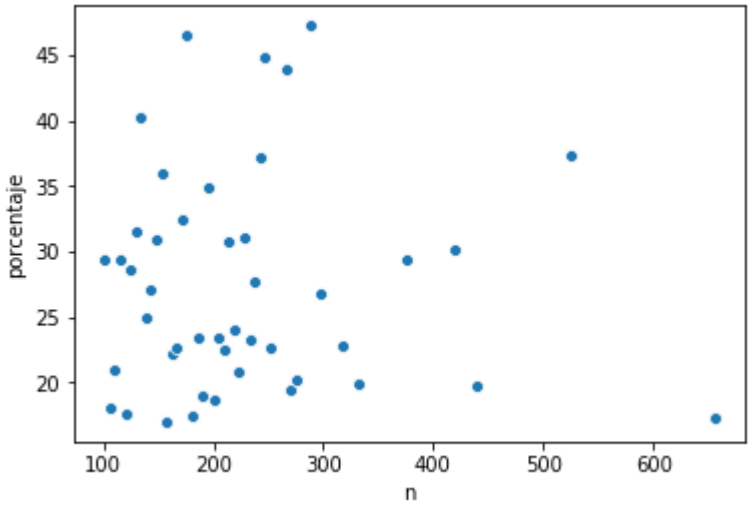
Nota: los criterios usados para la medición de este caso son idénticos a los del caso A con la salvedad de que se usan los parámetros correspondientes previamente calculados.

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).