Heurística Constructiva Golosa "Merge más cercanos"

La heurística 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
```

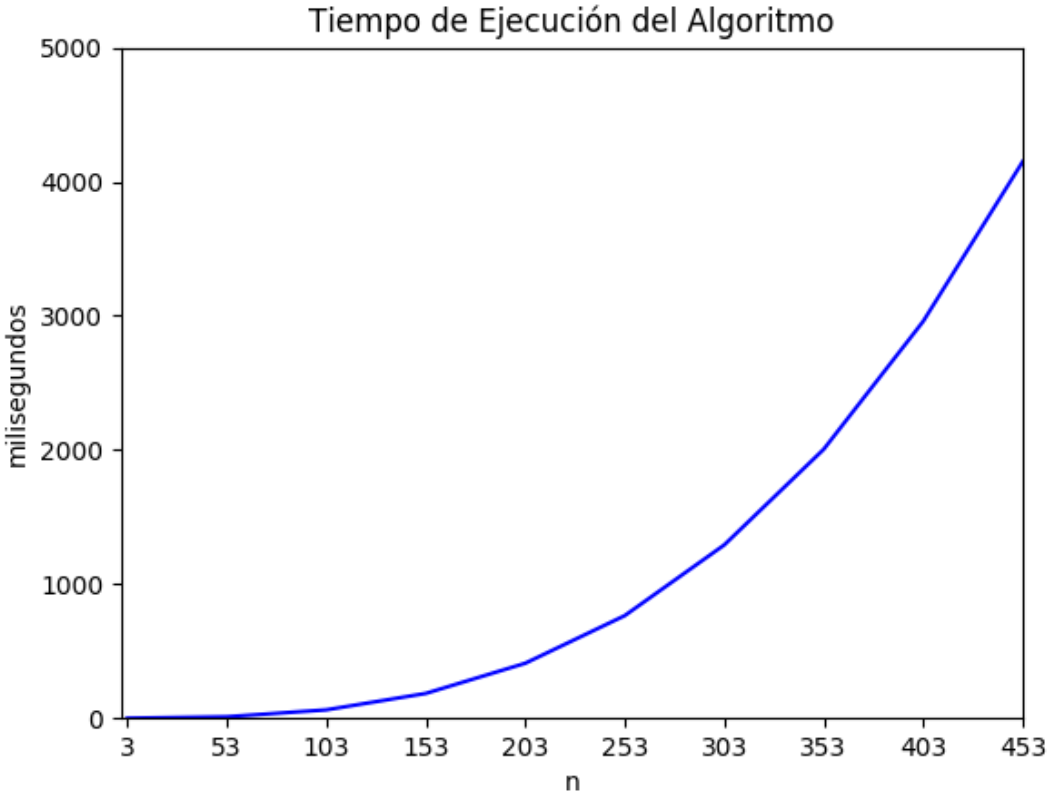
Como podemos ver, el algoritmo es muy simple. Una aclaración importante es que la union entre las rutas une el último elemento no depósito de la rutaA con el primer elemento no depósito de la rutaB y luego descarta estos depósitos de manera que la ruta obtenida sea una ruta válida. Tampoco es menor enfatizar 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.

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)

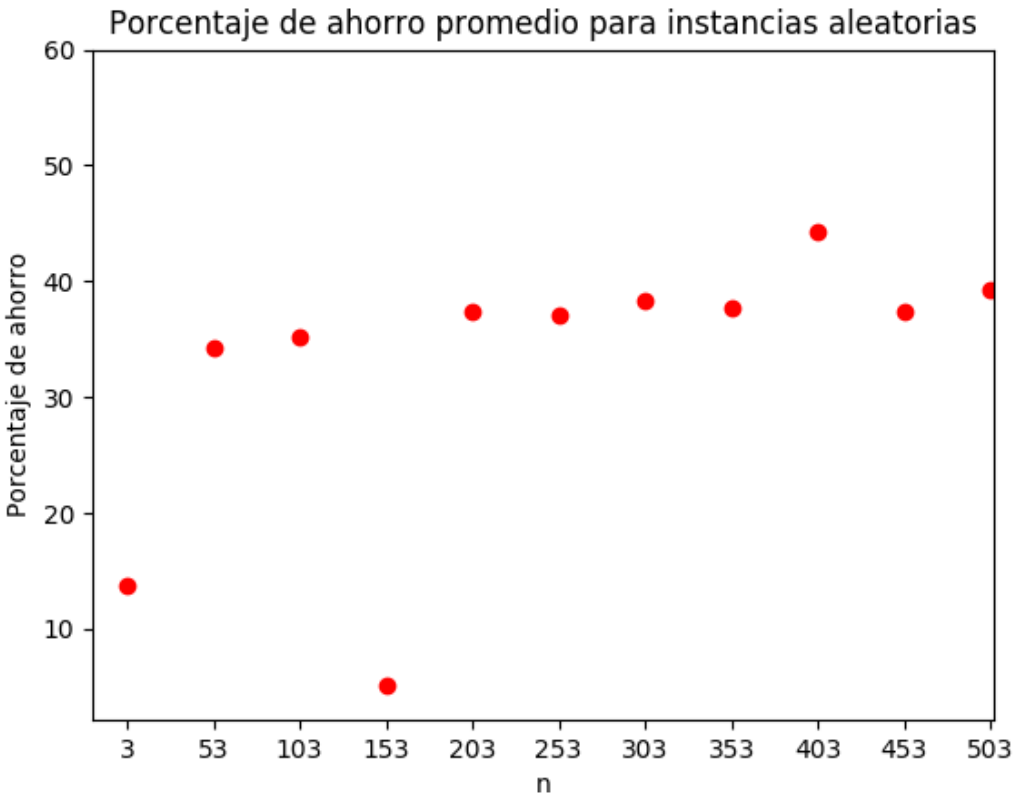
Comprobemoslo realizando las mediciones directamente. Generamos 400 instancias de grafo de para cada tamaño desde n = 3 hasta n = 453, con saltos de a 50 y promediamos el tiempo obtenido. Los resultados se pueden ver a continuación:

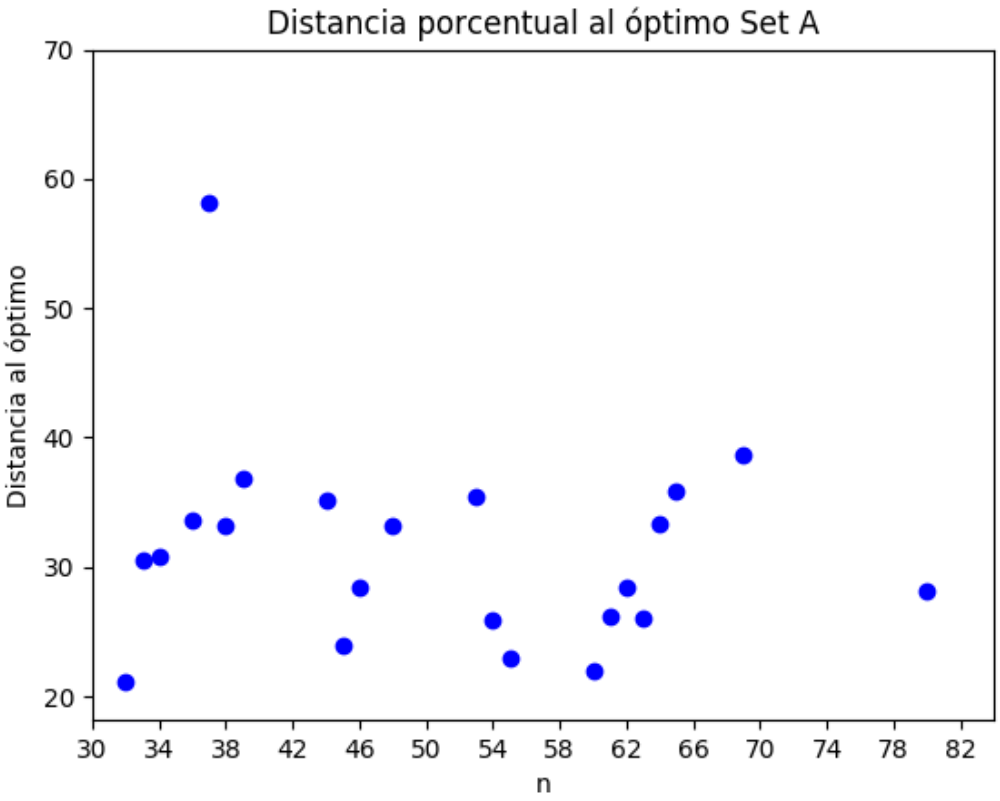


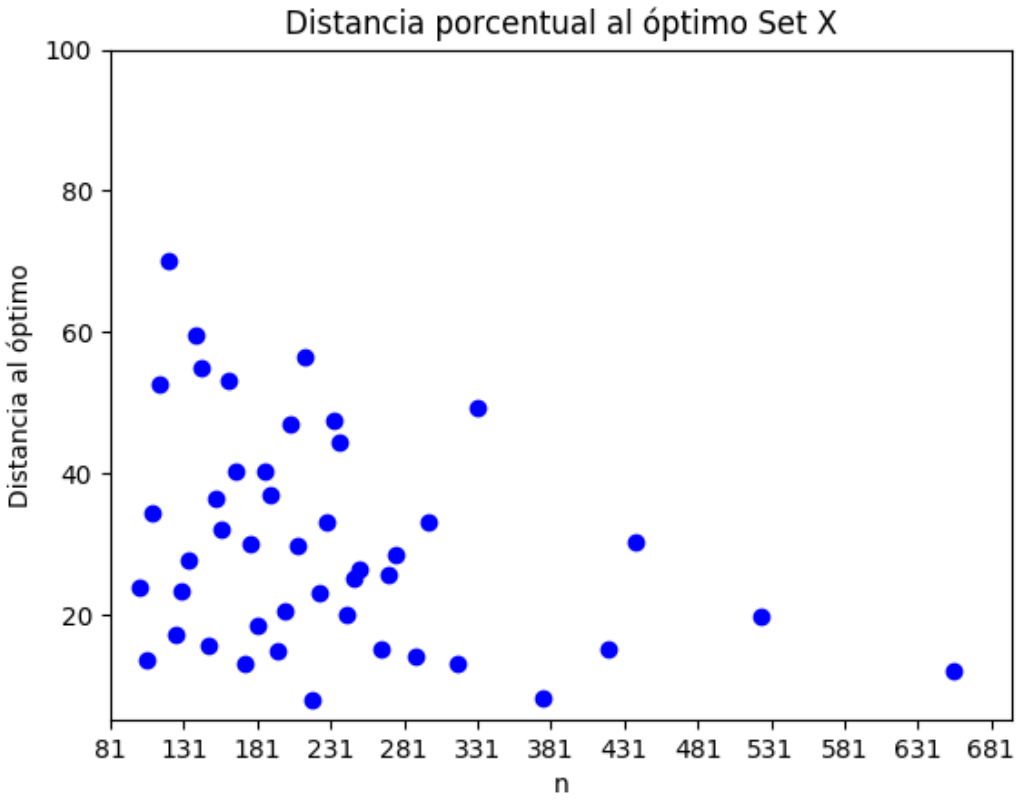
Rendimiento para diferentes sets de instancias

`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

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.

Simulated Annealing

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').

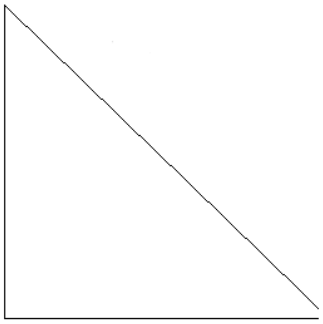
Siendo `shift` y `exchange` los siguientes procedimientos:

```
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
```

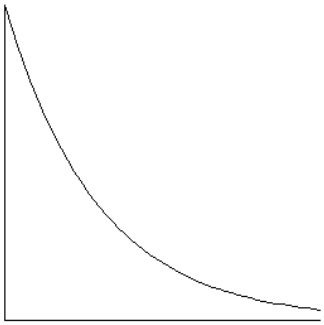
```
1  exchange(vector<Ruta> rutas, Ruta ruta1, Ruta ruta2,c1,c2, Grafo G)
2    if(Σ(demandas(ruta1)) - rutas[ruta1][c1].demanda + rutas[ruta2]
[c2].demanda <= G.capacidad_total &&
3    Σ(demandas(ruta2)) - rutas[ruta2][c2].demanda + rutas[ruta1][c1] <
G.capacidad_total)
4          temp = rutas[ruta1][c1]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

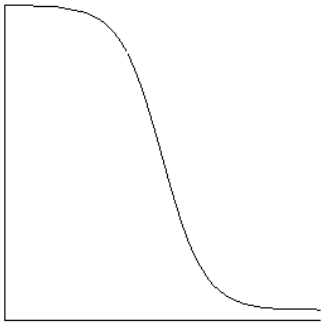
Como **Cooling Schedule** tendremos varias opciones con diferente efectividad. Más adelante realizaremos experimentos comparándolos entre sí. Las variantes que consideraremos serán *CS0*: Ti = Ts - i*(Ts - Tf)/ NIt, *CS1*: Ti = Ts * Tf/ Ts^(i/NIt) y *CS2*: Ti = ((Ts-Tf)/(1+e^(0.3*(i - NIt/2)))) + Tf 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)

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

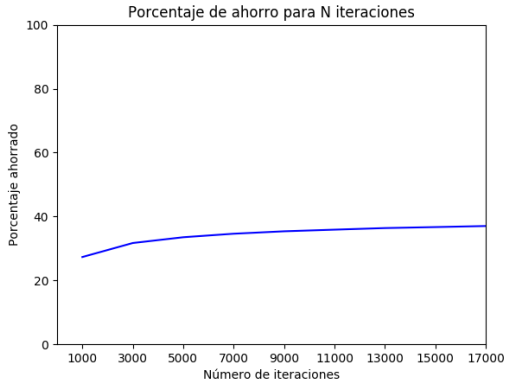
Búsqueda de parámetros óptimos

Caso aleatorio

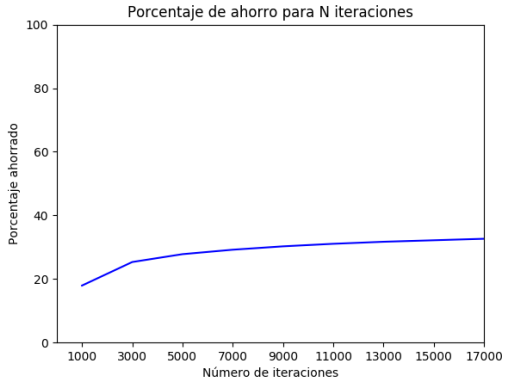
Busquemos el valor óptimo de los parámetros de nuestro algoritmo de Simulated Annealing para el casos aleatorio. Es decir, queremos encontrar el valor de los parámetros que mejores resultados obtiene en el promedio de muchos casos aleatorios, lo cual es útil para tener una noción general de qué valores 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, $n = 103$



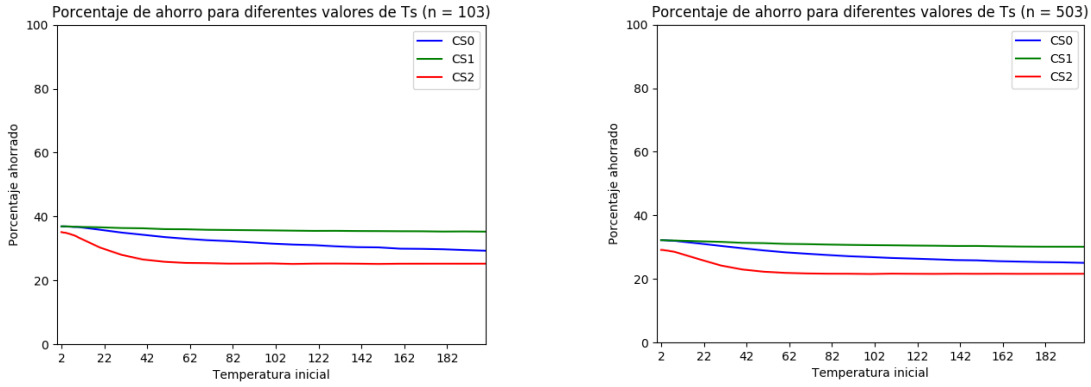
Ahorro porcentual de la solución canónica en función de NIt, $n = 503$caption)

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"

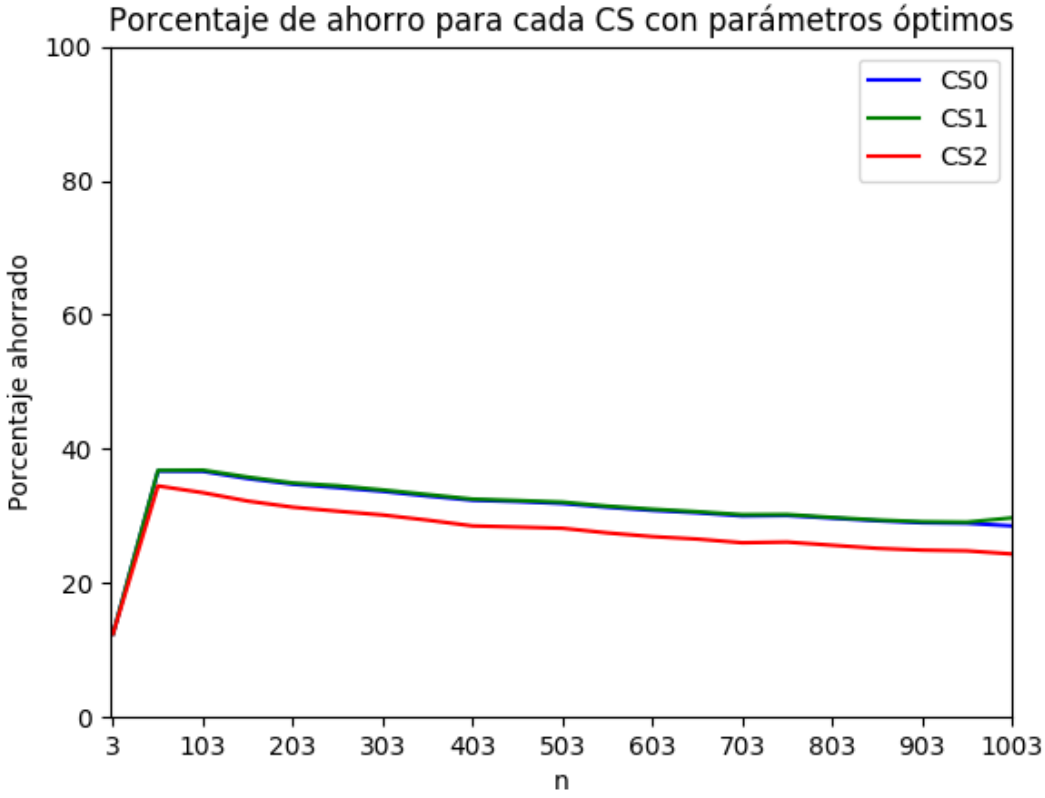
Teniendo en cuenta las caracteristicas de las instancias que mediremos y que no disponemos de tiempo ilimitado para realizar las mediciones, decidimos que *Nit* = 10.000 es suficientemente grande para ser representativo del resultado del algoritmo y suficientemente pequeño para poder realizar el resto 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)`

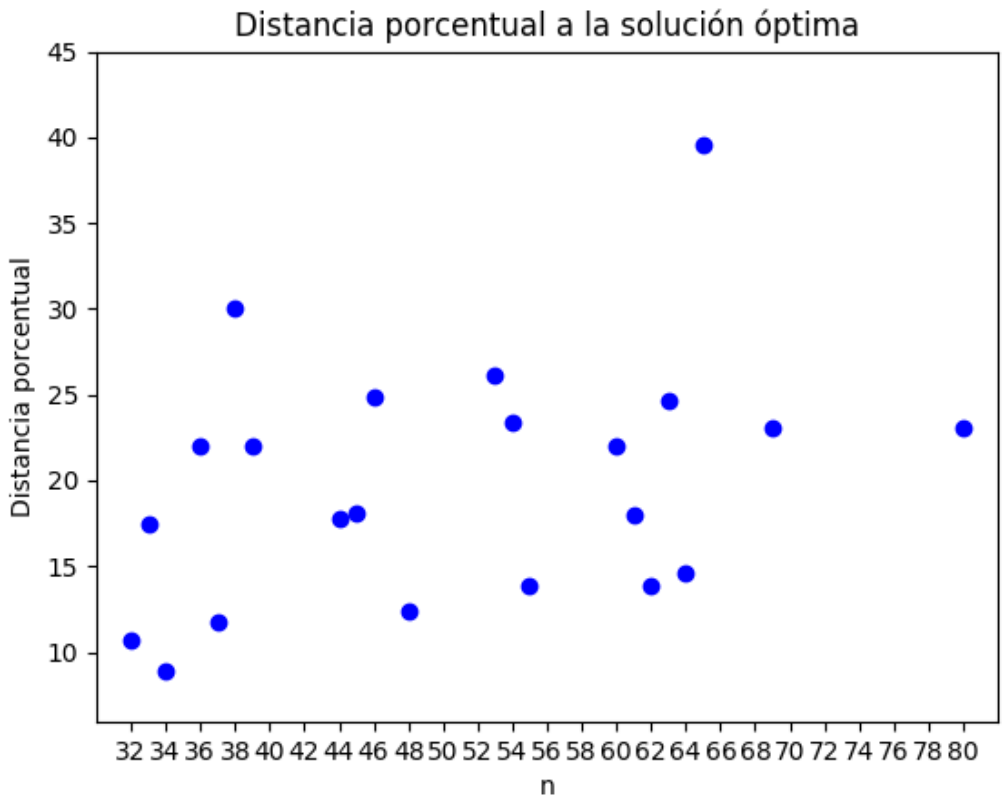
Parámetros óptimos para Set A

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:

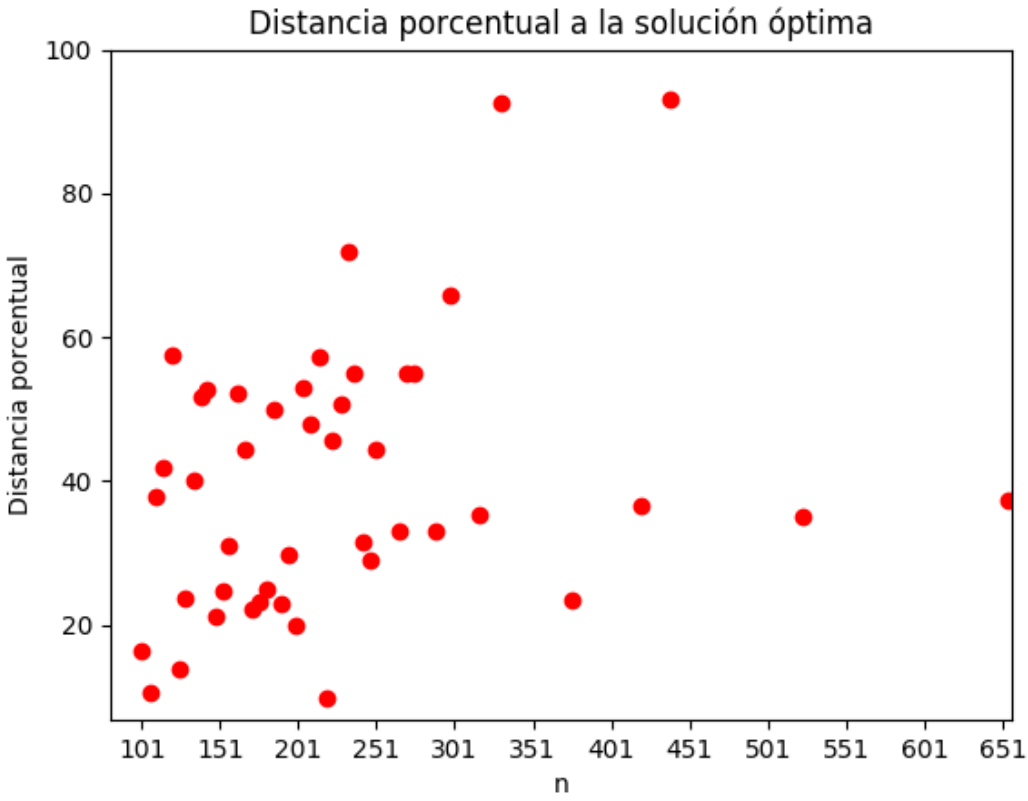


Parámetros óptimos para Set X

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:



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.