DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

TP2 Optimización Combinatoria: TSP

10 de junio de 2021 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Bulacio Granata, Blas	284/19	blasbulacio@gmail.com
Ernst, Erik	111/19	erik_ernst@hotmail.com.ar
Salmun, Daniel	108/19	salmundani@gmail.com
Sujovolsky, Tomás	113/19	tsujovolsky@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2610 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (++54 +11) 4576-3300

http://www.exactas.uba.ar

Índice

1. Introducción

En los problemas de optimización combinatoria, la idea es encontrar la mejor solución dentro de todas las soluciones posibles. En general, estos problemas tienen un conjunto de posibles soluciones muy grande, por lo cual encontrar la solución óptima es muy complejo o costoso. Sin embargo, muchos problemas de la realidad de gran interés pueden ser resueltos con optimización combinatoria, por lo que es un área de la computación con gran empuje en investigación.

En este trabajo, intentaremos resolver el Problema del Viajante de Comercio (TSP, por sus siglas en ingles). Lo que plantea resolver este problema es buscar un camino que contenga ciertas ciudades por las cuales un comerciante iría vendiendo sus productos, de costo mínimo, ya que el comerciante quiere maximizar sus ganancias. Si tomamos a las ciudades como vértices, y los viajes entre ciudades como aristas, podemos definir con las herramientas de la materia el problema modelado como grafo. Visto como grafo, la idea es buscar un camino hamiltoniano de costo mínimo, es decir, un camino que pase exactamente 1 vez por cada vértice.

El problema TSP pertenece a la clase $NP - difícil$, lo que significa que no conocemos un algoritmo que nos permita dar una solución óptima a este problema en tiempo polinomial. Por lo tanto nos vamos a enfocar en realizar algoritmos que nos van a devolver una solución de buena calidad, pero que no necesariamente será la mejor. Estos algoritmos son llamados **heurísticas**.

1.1. Alcances en la vida real

Ejemplos de la realidad que pueden ser modelados con este problema sobran. Muchos son problemas de logística como el camino que tiene que tomar un camión de carga entre sus distintos puntos de entrega, intentando minimizar el tiempo tomado, el combustible usado, o ambas. Esta misma lógica se le puede atribuir a los deliveries de comida con varias entregas, cuyo objetivo sería minimizar el tiempo tomado para que la comida no llegue fría.

El hecho de que haya tantos problemas tan cruciales para nuestras vidas cotidianas sean modelables por TSP hace que sea tan importante encontrar algoritmos que devuelvan una buena solución en el menor tiempo posible. Esto es lo que intentaremos hacer a lo largo de este trabajo.

1.2. Enunciado del problema

[1]El Problema del Viajante de Comercio se puede definir formalmente de la siguiente manera. Sea $G = (V, E)$ un grafo completo donde cada arista $(i, j) \in E$ tiene asociado un

[1]Para la explicación del enunciado formal del problema utilizamos la que se encuentra en el enunciado del TP.

costo c_{ij}. Se define el costo de un camino p como la suma de los costos de sus aristas $c_p = \sum_{(i,j)\in p} c_{ij}$.

El problema consiste en encontrar un circuito hamiltoniano p de costo mínimo. Sin pérdida de generalidad, vamos a pedir que el primer vértice del ciclo sea el 1.[2]

Dado que el TSP pertenece a la categoría de problemas $NP - difícil$, no buscaremos dar la solución óptima para cada instancia sino una de la mejor calidad posible. Sin embargo, existen algoritmos exactos que pueden resolver instancias de este problema con hasta 85900 vértices. También, existe un algoritmo de programación dinámica cuya complejidad pertenece a $\mathcal{O}(2^n n^2)$.

2. Heurísticas constructivas golosas

Las heurísticas constructivas golosas son métodos que construyen una solución factible de una instancia de un problema mediante procesos golosos. Esto significa ir armando la solución tomando las mejores decisiones posibles de forma local y no volver atrás en ellas, lo que resulta en soluciones factibles que se acercan a un buen resultado mucho más que generar una aleatoriamente pero de igual forma no se garantiza que se construya la mejor solución.

La entrada de estos algoritmos será un grafo pesado completo $G = (V, E)$, donde $n = |V|$ y $m = |E| = \frac{n*(n-1)}{2}$. Globalmente, definimos l como la función peso $l : E \to \mathbb{N}_{\geq 0}$, e implementaremos una matríz de adyacencias $M \in \mathbb{N}_{\geq 0}^{n\times n}$ donde $M[i][j] = l((i,j)),\ \forall i, j \in E$.

2.1. Heurística del vecino más cercano

Comenzamos desde el vértice 1, y luego en cada paso elegimos como siguiente vértice del circuito el que, entre los que todavía no fueron visitados, se encuentre más cerca del actual. Este es un procedimiento goloso.

HEURÍSTICAVECINO$(G = (V, E)) \longrightarrow solucion$

1: $v \leftarrow 1$
2: $solucion \leftarrow [v]$
3: **mientras** $|solucion| < n$ **hacer**
4: $\quad w \leftarrow$ **argmín**$\{l(v, w), w \in \{V - solucion\}\}$
5: $\quad solucion + w$
6: $\quad v \leftarrow w$

---

[2]Notar que puede tener muchas soluciones óptimas.

2.1.1. Complejidad

El **mientras** se ejecuta n veces, dado que vamos agregando de a uno a *solución* hasta que dicho conjunto incluya a todos los vértices. En el cuerpo del ciclo, debemos encontrar el vecino más cercano al vértice v, esto lo podemos hacer en $O(n)$ encontrando el $w \in V$ que minimice $M[v][w]$. Por lo tanto, el algoritmo en total cuesta $O(n) * O(n) = O(n^2)$.

2.2. Heurística Inserción

Esta heurística comienza formando un ciclo compuesto por el vértice inicial 1 y otros dos vértices distintos elegidos de forma aleatoria v, w. Iremos eligiendo e insertando vértices nuevos que se encuentran fuera de este ciclo hasta que sea hamiltoniano.

HEURÍSTICAINSERCIÓN$(G = (V, E)) \longrightarrow solucion$

```
1: solucion ← [1, v, w]
2: mientras |solucion| < n hacer
3:     ELEGIR v ∈ {V − solucion}
4:     INSERTAR v en solucion
```

Donde las reglas para elegir e insertar en nuestro caso son las siguientes:

- ***ELEGIR*** **elige el vértice más cercano a un vértice que ya está en el circuito**:
  - Para cada vértice j en *solucion*, nos quedaremos con el vértice k en $V - solucion$ que minimice $M[j][k]$.
- ***INSERTAR*** **inserta v de forma tal que *solucion* tenga el mínimo costo:**
  - Armamos un conjunto que contenga todas las aristas que unen los vértices de *solucion*.
  - Recorreremos cada arista (i, j) del conjunto y preguntamos cómo cambia el peso del camino si intercambiamos la arista (i, j) por la secuencia $(i, v), (v, j)$. Guardamos entre qué vértices la inserción de v genera menor costo.
  - Reconstruimos *solucion* con el vértice v ubicado en el lugar encontrado.

2.2.1. Complejidad

El **mientras** termina luego de $O(n)$ iteraciones, porque vamos incrementando el tamaño de *solucion* de a uno, analicemos cuanto nos cuesta cada una de ellas:

- ***ELEGIR***: Como $|solucion| = O(n)$ y elegir el mínimo de la fila j de la matríz es $O(n)$, elegir v nos cuesta $O(n) * O(n) = O(n^2)$

- ***INSERTAR***: En primer lugar, obtener el conjunto de aristas de *solucion* nos tomará $O(n * log(n))$, dado que por cada vértice i debemos insertar $(i, i+1)$ en un *std::set*. Luego hay buscar entre qué vértices ubicar v, el conjunto de aristas tiene tamaño n y por cada $(i, i+1)$ debemos calcular cuanto cuesta reemplazar esta arista por $(i, v), (v, i+1)$, gracias a nuestra matriz de adyacencias M podemos calcular el costo de la solución luego de intercambiar aristas en $O(1)$, en total este paso cuesta $O(n)$. Por último reconstruir *solucion* para ubicar v en la posición correcta se realiza en $O(n)$, teniendo la inserción un coste total de $O(n * log(n)) + O(n) + O(n) = O(n * log(n))$.

Por lo tanto el cuerpo del ciclo cuesta $O(n^2) + O(n * log(n)) = O(n^2)$, y en total el algoritmo $O(n) * O(n^2) = O(n^3)$.

2.3. Heurística AGM

Consiste en obtener un árbol generador mínimo T del grafo de entrada G, luego asignar un orden a los vértices de T utilizando **DFS** y siguiendo este orden formar un ciclo hamiltoniano. Formalmente, marcaremos todo $v \in G$ de forma tal que v_i sea el *i-ésimo* vértice de un recorrido DFS de T, por último $v_1, v_2, \ldots, v_n, v_1$ es el ciclo hamiltoniano que buscamos. Como fue mencionado previamente, en nuestro caso siempre $v_1 = 1$, es decir siempre ejecutamos el orden DFS desde el vértice 1.

HEURÍSTICAAGM$(G = (V, E)) \longrightarrow solucion$

1: $T \leftarrow Prim(G)$
2: $solucion \leftarrow recorridoDFS(G)$

Analicemos el algoritmo con un poco más de detalle. Llamemos C a la solución obtenida por la heurística y OPT a la solución óptima de TSP del grafo de entrada G. En primer lugar, el peso del árbol generador mínimo T, es cota una inferior de OPT, esto vale porque la solución de TSP incluye a un árbol. Luego, observemos la siguiente definición:

Def: Un grafo pesado completo $G = (V, E)$ es *euclideano* si se cumple la desigualdad triangular:

$$l((i, k)) \leq l((i, j)) + l((j, k)) \text{ para todo } i, j, k \in V.$$

Pensemos en un nuevo grafo $2T$, que se obtiene duplicando las aristas de T. Es fácil de ver que podemos formar un ciclo hamiltoniano C_{2T} comenzando en 1 y pasando una vez por cada arista, y $peso(C_{2T}) = 2 * peso(T)$. Observemos que si el grafo original G es euclidiano, se cumple que

$$peso(T) \leq peso(C) \leq peso(C_{2T}).$$

Esto es así porque para dos vértices consecutivos en el orden dado por DFS v_i, v_{i+1}, el peso de la arista (v_i, v_{i+1}) es menor o igual al de hacer un camino intermedio entre v_i y otro vértice para luego volver a v_{i+1}.

Por otro lado, observemos que como $peso(OPT) \leq peso(C)$

$$peso(T) \leq peso(OPT) \leq peso(C_{2T}) = 2 * peso(T),$$

lo que significa que si el grafo es euclideano, C está a menos de $peso(T)$ del óptimo.

2.3.1. Complejidad

- **Prim:** Hacemos uso de un vector $pesoUnir$, donde $pesoUnir[i]$ guarda el coste mínimo de agregar el vértice i al AGM que estamos construyendo. Se puede encontrar el vértice que minimiza dicho vector y luego actualizar el resto de las estructuras en $O(n)$. Este proceso lo repetimos desde el vértice 1 hasta n, resultando en una complejidad de $O(n^2)$.
- **DFS:** Debemos marcar cada vértice v, esto cuesta $O(n)$. Además, revisamos la vecindad de cada v exactamente una vez, y como $\sum |N(v)| = 2 * m = O(m)$, en total se realizan $O(n + m) = O(n + n^2) = O(n^2)$ operaciones.

Por lo tanto el algoritmo en total cuesta $O(n^2)$.

3. Metaheurística Tabú Search

Una de las metaheurísticas que vimos en clase fue Tabú Search. Vamos a implementarla para conseguir o intentar aproximarnos a la solución óptima. En particular vamos a probar dos versiones distintas: una donde se guardan las últimas k soluciones encontradas y otra donde se guardan las últimas k aristas modificadas.

Memoria basada en soluciones exploradas

Para el primer caso, implementamos el siguiente algoritmo:

```
tabuSearch(G, k, cantIteraciones, porcentajeVecindario) ⟶ solucion
 1: mejorCandidato ← heurísticaVecino(G)
 2: mejorSolucion ← mejorCandidato
 3: costoMejorSol ← calcularCosto(mejorSolucion)
 4: costoMejorCandidato ← costoMejorSol
 5: colaUltimasKSols ←<>
 6: setUltimasKSols ←
 7: iterSol ← setUltimasKSols.insert(mejorCandidato)
 8: colaUltimasKSols.encolar(iterSol)
 9: for i = 0; i < cantIteraciones; i++ hacer
10:     vecindario ← 2-opt(mejorCandidato, porcentajeVecindario)
11:     vecindario ← filtrarSolsRepetidas(vecindario, setUltimasKSols)
12:     si vacio?(vecindario) entonces
13:         iterABorrar ← colaUltimasKSols.desencolar()
14:         setUltimasKSols.borrar(iterABorrar)
15:     mejorCandidato ← vecindario[0]
16:     for candidato ∈ vecindario hacer
17:         costoCandidato ← calcularCosto(candidato)
18:         si costoCandidato < costoMejorCandidato entonces
19:             mejorCandidato ← candidato
20:             costoMejorCandidato ← costoCandidato
21:     si costoMejorCandidato < costoMejorSol entonces
22:         mejorSolucion ← mejorCandidato
23:         costoMejorSol ← costoMejorCandidato
24:     iterSol ← setUltimasKSols.insert(mejorCandidato)
25:     colaUltimasKSols.encolar(iterSol)
26:     si |ultimasKSols| > k entonces
27:         iterABorrar ← colaUltimasKSols.desencolar()
28:         setUltimasKSols.borrar(iterABorrar)
    devolver mejorSolucion
```

Se inicializa el primer candidato con la solución generada por la heurística del vecino más cercano. Se toma esta solución como el mejorCandidato y la mejorSolucion inicial.

Luego, en cada iteración busca una vecindad de soluciones. Por lo que vimos en la teórica uno de los métodos de vecindario más utilizados para el problema de TSP es *2-opt*. Nuestra implementación de *2-opt* es la siguiente:

2-OPT($recorrido$, $porcentaje$)$\longrightarrow$ $vecindario$

```
1: vecindario ← <>
2: for i = 2; i < n; i++ hacer
3:     for j = i + 1; j < n; j++ hacer
4:         nuevoRecorrido ← 2-optSwap(recorrido, i, j)
5:         vecindario.agregar(nuevoRecorrido)
   devolver sample(vecindario, porcentaje)
```

Donde *2-optSwap* consiste en:

1. Tomar desde el primer elemento de *recorrido* a $recorrido_{i-1}$ y agregarlo en el mismo orden al nuevo recorrido
2. Tomar desde $recorrido_i$ a $recorrido_j$ y agregarlo en orden inverso al nuevo recorrido
3. Tomar desde $recorrido_{j+1}$ al final y agregarlo en el mismo orden al nuevo recorrido

Otra función que usamos en el pseudo código es *sample*, que escoge elementos al azar del vecindario hallado hasta quedarnos con un porcentaje del original definido por el parámetro *porcentaje*. Nuestra hipótesis es que agregando más azar al tabú vamos a tener mayores chances de salir de mínimos locales y acercarnos más a la solución óptima deseada.

Notemos que el primer for comienza en 2, ya que caso contrario en *2-optSwap* se podría modificar el orden tal que el primer vértice no se encuentre primero en la solución. Otra cosa es que al tener en cuenta que $i < j$ para que tenga sentido el swap que aplicamos, el for anidado comienza en $i + 1$ por lo que *vecinos* tendrá $\frac{n*(n-1)}{2}$ elementos. Si tomamos en cuenta que hacer el swap y agregarlo a *vecindario* es $O(n)$ y que $\frac{n*(n-1)}{2}$ está acotado superiormente por n^2, entonces *2-opt* tiene una complejidad de $O(n^3)$.

Una vez que tenemos el vecindario, filtramos las que están en las últimas k soluciones. En caso de que el vecindario quede vacío luego de este filtro, se elimina la última solución de la memoria ya que que filtre todas implica que la memoria es muy restrictiva, y se continúa a la siguiente iteración.

Ya tenemos el vecindario filtrado, ahora agarramos el primer elemento y lo guardamos como mejorCandidato. Vamos iterando por el resto del vecindario y vamos sobre escribiendo el mejorCandidato si encontramos uno que tenga un costo menor que el del mejorCandidato elegido.

Una vez elegido el mejorCandidato vemos si es mejor que la mejorSolucion que teníamos comparando el costo de cada uno y quedándonos con el menor.

Finalmente, guardamos el mejorCandidato hallado en esta iteración para no volver a repetirlo en la próximas k iteraciones. Si vemos que la memoria queda con una longitud mayor a k, eliminamos el último elemento de la cola para hacer lugar.

Analicemos su complejidad:

- Como ya mencionamos, la cantidad de vecinos que va a generar está acotada por $O(n^2)$ y la complejidad de *2-opt* es $O(n^3)$

- Luego, hay que filtrar del vecindario los candidatos que se encuentras en las últimas k soluciones. Esto implica que por cada candidato del vecindario (que dijimos que está acotado por n^2) debemos ver si está en las últimas soluciones. Utilizando un set basado en un árbol balanceado, esto lo podemos hacer en $O(n * \log k)$ por lo que el filtro cuesta $O(n^3 * \log k)$
- Por cada candidato del vecindario restante hay que ver si es mejor que el candidato elegido. Ver el costo total de un candidato es $O(n)$ por lo que buscar el mínimo del vecindario costaría $O(n^3)$
- El guardado de soluciones es $O(n \log k)$ ya que está basado en un árbol balanceado

Como todos los ítems anteriores se ejecutan *cantIteraciones* veces en el peor caso, la complejidad final del algoritmo es $O(cantIteraciones * n^3 * \log k)$

Memoria basada en últimas aristas modificadas

Es idéntico al método anterior solo que en vez de ir guardando las últimas k soluciones se guardan las últimas k aristas modificados. En el pseudo código: reemplazaríamos las variables setUltimasKSols y colaUltimasKSols por setUltimasKAristas y colaUltimasKAristas; en vez de filtrar por soluciones repetidas, vamos a filtrar a las que modifican a las últimas k aristas que fueron modificadas recientemente. Finalmente, vamos a comparar las aristas del mejor candidato con las del mejor candidato de la iteración pasada para ver las diferencias en aristas y guardarlas, removiendo las últimas del set y de la cola si nos pasamos de k.

Su análisis de complejidad es muy similar al anterior. El filtro que usamos es $O(\log n * n^3)$ ya que hay que calcular las aristas de la solución encontrada y luego compararlas con las últimas k aristas. Como implementamos ambas como set, el costo de generar las aristas del candidato es $O(n * \log n)$ y luego chequear que las últimas k estén en este set de aristas es $O(k * \log n)$. Como tenemos que hacer esto para todo el vecindario, la complejidad del filtro es $O(\log n * n^3)$.

Por lo tanto, siguiendo la misma idea que el modelo anterior, la complejidad total de este algoritmo es $O(cantIters * logn * n^3)$

4. Casos patológicos de las heurísticas

4.1. Heurística del vecino más cercano

Usando que el algoritmo siempre extenderá la solución por el vecino más cercano, lo obligaremos a pasar por una arista que tendrá un peso tan grande como queramos, mientras que la solución óptima se obtiene no yendo siempre por lo más barato. Dado $G = (V, E)$ completo con $n > 3$ (porque para $n = 3$ el único ciclo que hay es el óptimo), asignaremos los pesos de las aristas de la siguiente forma:

- $l((1,n)) = k$ con $k > 3$.
- $l((i,i+1))$ para todo $1 \leq i \leq n-1$
- El resto de las aristas tendrán peso 2.

De esta forma, forzamos que $solucion = [1, \ldots n, 1]$. Comenzando desde 1, en cada paso $1 \leq i < n$, la heurística elegirá incluir al vértice que lo une con $i+1$, porque para todo $v \neq i+1$ tenemos que $l((i,v)) = 2 > 1 = l((i,i+1)$ o que $v \in solucion$. Cuando lleguemos al paso $i = n$ no queda otra opción que utilizar la arista $(n,1)$ para cerrar el ciclo. El peso de la solución que encontrará será de $VEC = k + n - 1$, porque tomará $n-1$ aristas de peso 1 y una arista de peso k.

Para los grafos de esta familia, la solución óptima de TSP es saltarnos la arista $(n,1)$ de peso k utilizando aristas intermedias de peso 2. En lugar de tomar el camino $[1 \ldots n-2, n-1, n, 1]$ haremos $[1 \ldots n-2, n, n-1, 1]$, así reemplazamos recorrer $n-1$ aristas de peso 1 más la arista de peso k por recorrer $n-2$ aristas de peso 1 (ya no estamos usando la arista $(n-2, n-1)$) y dos aristas de peso 2, de esta forma la solución óptima valdrá $OPT = 4 + n - 2$.

Por lo tanto, como para todo grafo completo de tamaño $n > 3$ y $k > 3$ se cumple que

$$VEC = k + n - 1 > 4 + n - 2 = OPT,$$

podemos elegir un k tan grande como queramos y alejar tanto como deseemos el resultado obtenido del resultado óptimo, dado que siempre $VEC - OPT = k - 3$. En **Figura 1** vemos un ejemplo con $n = 5$ y $k = 100$.

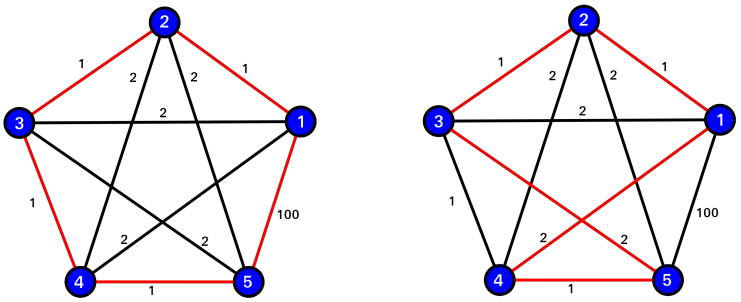

Figura 1: A la izquierda, lo que nos devolvería el algoritmo basado en la heurística del vecino más cercano, el valor del circuito encontrado es de $VEC = k + n - 1 = 100 + 4 = 104$. A la derecha la solución óptima del problema $OPT = 4 + n - 2 = 4 + 3 = 7$. Vemos que se cumple que $VEC - OPT = 97 = k - 3$.

4.2. Heuristica AGM

Podemos utilizar la familia de grafos definida en el punto anterior para la heurística del vecino. Esto funciona porque el árbol generador mínimo de los grafos en esta familia es

el camino de aristas de peso unitario $[1, \ldots, n]$, luego ejecutar DFS desde el vértice $v = 1$ en el árbol generador mínimo nos devolverá el recorrido $[1, \ldots, n]$. Cerraremos dicho ciclo para que sea hamiltoniano, por lo tanto $solucion = [1, \ldots, n, 1]$ y el peso de la solución que nos da esta heurística es $AGM = k + n - 1$.

Nuevamente la solución óptima es saltearse la arista $(n, 1)$ de peso $k > 3$ realizando el recorrido $[1, \ldots, n-2, n, n-1, 1]$, y el peso de la solución óptima es $OPT = 4 + n - 2$.

Se repite que para todo grafo completo de tamaño $n > 3$ y $k > 3$ se cumple $VEC > OPT$ y se mantiene una diferencia $VEC - OPT = k - 3$ que podemos hacer tan grande como queramos.

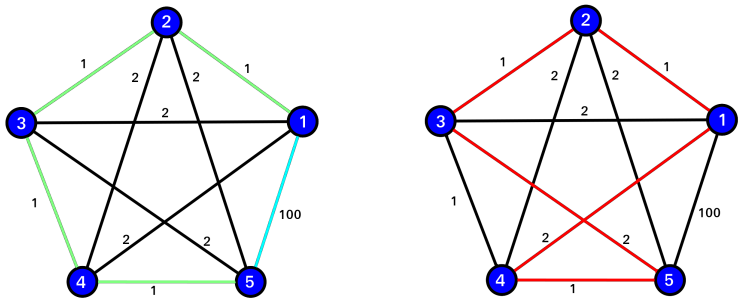

Figura 2: A la izquierda, lo que nos devolvería el algoritmo basado en la heurística del árbol generador mínimo, en verde marcamos el árbol generador mínimo que se genera en el primer paso del algoritmo, y en celeste la arista que cierra el ciclo hamiltoniano. El valor del circuito encontrado es de $AGM = k + n - 1 = 100 + 4 = 104$. A la derecha la solución óptima del problema $OPT = 4 + n - 2 = 4 + 3 = 7$. Vemos que se cumple que $AGM - OPT = 97 = k - 3$.

4.3. Heurística Inserción

En el caso de la heurística de inserción el problema con el que se encuentra es lo estático que es con respecto a las subsoluciones. Aún teniendo una subsolucion óptima, es decir, que se tenga un ciclo hamiltoniano mínimo de un subconjunto de vértices, al elegir un vértice solo se analizan $n - 1$ lugares donde se podría insertar dentro del ciclo. En este paso no se analizan las posibles inserciones con las permutaciones de la subsolución, cuyo orden relativo se mantendrá siempre igual. Las posibles soluciones que no se analizan crecen factorialmente y dependiendo del peso de las aristas en los subgrafos podrían llevar al óptimo.

En cuanto a nuestra implementación, partimos de un ciclo obtenido aleatoriamente por lo que no será posible determinar casos patológicos aunque sí sabemos la pinta de esas instancias. Como el orden relativo se mantiene constante, cuando se decide un orden relativo distinto del que tendrá la solución óptima, ésta última nunca será alcanzada.

5. Experimentación

En esta sección analizaremos resultados de experimentos computacionales con el objetivo de estudiar cómo funcionan en la práctica los algoritmos presentados en este informe.

Construiremos de acuerdo a cada experimento una entrada para el algoritmo, es decir, los parámetros de entrada del TSP: un grafo pesado $G = (V, E)$ completo con una cantidad de vértices n y de aristas $m = \frac{n*(n-1)}{2}$, y una asignación $l : E \to \mathbb{N}_{\geq 0}$ de los pesos de las aristas. Los experimentos fueron realizados con un procesador Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz.

5.1. Algoritmos

- **VEC:** Heurística del vecino más cercano.
- **INS:** Heurística de inserción.
- **AGM:** Heurística del árbol generador mínimo.
- **TABU_A:** Metaheurística tabú search con memoria basada en las aristas.
- **TABU_S:** Metaheurística tabú search con memoria basada en las últimas soluciones exploradas.

5.2. Instancias

Generaremos distintos *datasets*, son conjuntos de instancias que cumplen propiedades que las hacen adecuadas para experimentación.

5.2.1. Generadas

Para distintos valores de n, creamos uno o varios grafos pesados completos $G = (V, E)$ y una asignación de pesos l de la siguiente forma en cada *dataset*.

- **aleatorias-complejidad:** Para todo $e \in E$, $l(e)$ es un número aleatorio en el rango $0...n$. Nos sirven para analizar la complejidad de los algoritmos.
- **aleatorias:** Por cada n generamos 10 grafos K_n, donde los pesos de las aristas se eligen aleatoriamente en el rango $0...n$. No conocemos el resultado óptimo de estas instancias, pero generar muchas para un mismo n nos sirve para analizar la distribución de los resultados obtenidos en gráficos como *boxplots*.
- **malo-agm-vec:** Es el caso donde los algoritmos AGM y VEC dan mal siempre, este dataset se genera con cierto valor $k > 3$ que crece de forma cuadrática respecto a n y va alejando el óptimo del obtenido. Como vimos en el desarrollo del informe para

$1 \leq i < n$ se asigna $l((i, i+1)) = 1$, luego $l((n, 1)) = k$ y para el resto de las aristas $l((u, v)) = 2$. Conocemos que el óptimo de estas instancias es igual a $4 + n - 2$ y que el algoritmo siempre nos devolverá $k + n - 1$, por lo tanto hay un gap de $k - 3$.

5.2.2. TSPlib

Para evaluar la calidad de las soluciones respecto a óptimos conocidos, utilizamos las instancias *ch130*, *burma14*, *berlin52* y *bays29* de la libreria TSPlib. En la página de TSPlib se pueden encontrar las respuestas para cada una de ellas. Juntamos las instancias en el dataset **instancias-tsplib**.

5.3. Análisis de los casos patológicos

Vamos a correr los algoritmos AGM y VEC sobre el dataset **malo-agm-vec**. Recordemos que en base a un valor k, podemos alejar las instancias entre el obtenido y el óptimo tanto como deseemos, para mostrar este efecto fuimos aumentando el k de forma cuadrática respecto de la cantidad de vértices n dada una instancia. Recordemos que bajo estas instancias los algoritmos siempre nos devolvían un camino de largo $k+n-1$ mientras que el óptimo era $4+n-2$, por lo tanto la diferencia entre el obtenido y el óptimo es $k*3$, se mantiene lineal respecto de k.

Heurística Vecino

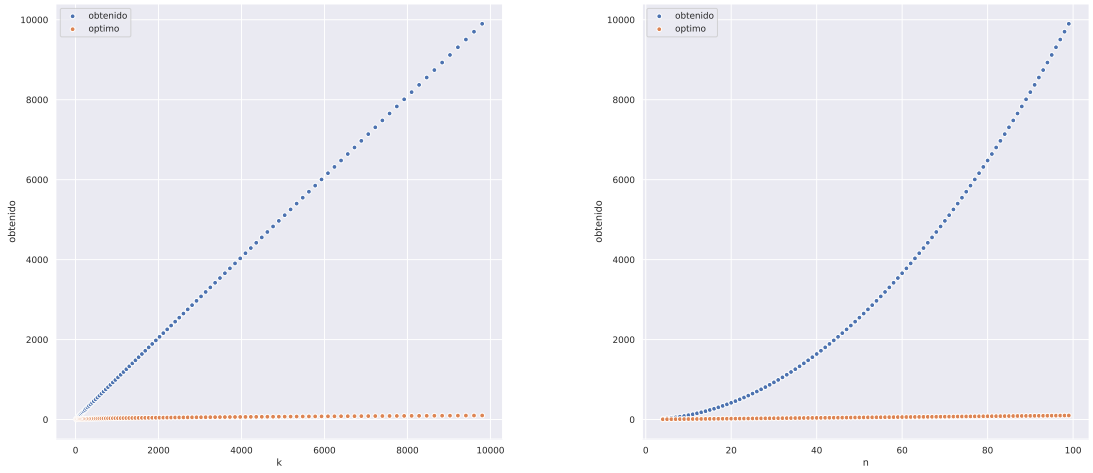

Figura 3: El gráfico de la izquierda muestra cómo el valor obtenido por VEC se aleja del valor óptimo en forma lineal a medida que k aumenta. A la derecha vemos lo mismo en función de n, recordemos que k crece cuadráticamente respecto de n por lo tanto la diferencia entre el obtenido y el óptimo también.

También es importante verificar que nuestro algoritmo efectivamente nos devuelve el peor caso que esperábamos, por lo tanto calculamos la correlación de *valor obtenido vs valor esperado* en cada instancia. En la **Figura 4** se ve una correlación perfecta.

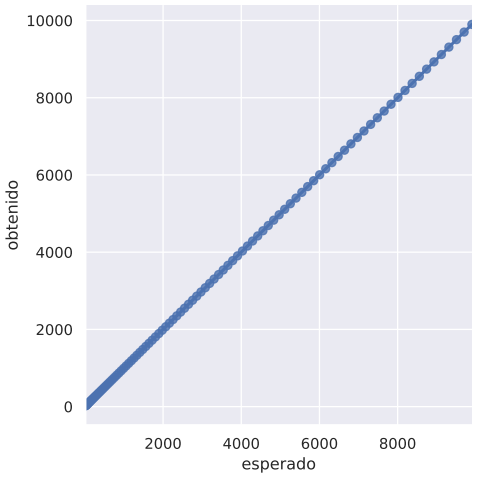

Figura 4: Gráfico de correlación entre el valor obtenido y el valor esperado $k + n - 1$ para las instancias donde VEC da mal. Se obtuvo $r = 1$.

Heuristica AGM

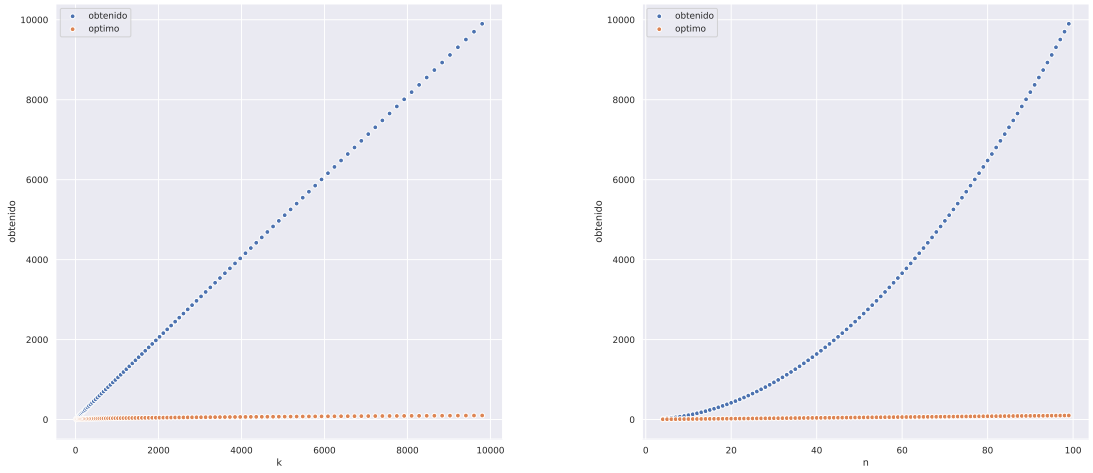

Figura 5: El gráfico de la izquierda muestra cómo el valor obtenido por AGM se aleja del valor óptimo en forma lineal a medida que k aumenta. A la derecha ídem en función de n.

Realizamos el mismo gráfico de correlación que en el ítem anterior y se observan buenos resultados.

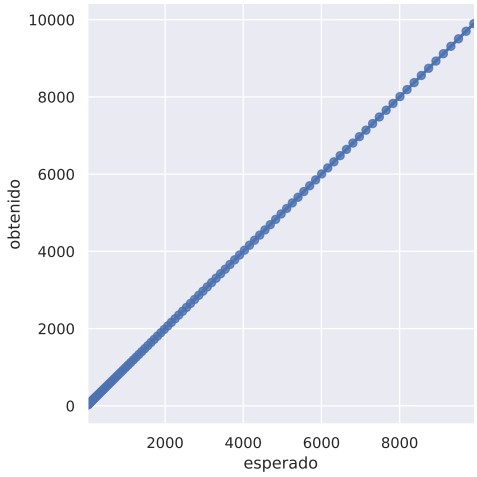

Figura 6: Gráfico de correlación entre el valor obtenido y el valor esperado para las instancias donde AGM da mal. Se obtuvo $r = 1$.

5.4. Análisis de la complejidad de los algoritmos

Para analizar la complejidad de los algoritmos, utilizamos el dataset **instancias-aleatorias** que van desde 3 hasta 100 vértices. La idea es ver como crece el tiempo de computo con respecto al n. Para la metaheurística de tabú, además analizamos su complejidad con respecto a sus otros parámetros de entrada.

Graficaremos los tiempos teóricos encima de los resultados de la experimentación para observar si se presenta un crecimiento similar y también haremos un gráfico de correlación, que consiste en enumerar las instancias y para cada una graficar el tiempo de ejecución real contra el tiempo esperado.

Heurística Vecino

Como hipótesis esperamos encontrar probar que la complejidad del algoritmo VEC es $O(n^2)$ como fue explicado en el desarrollo del informe.

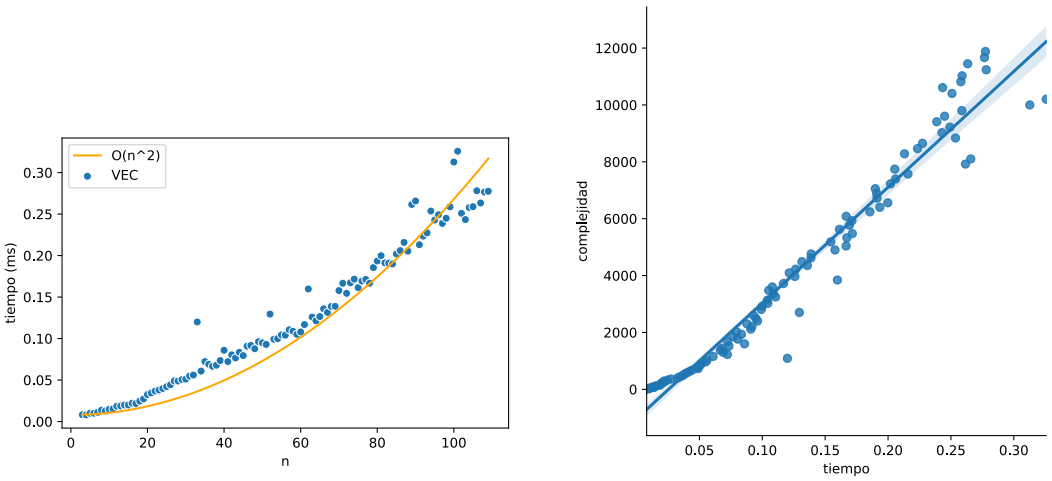

Figura 7: A la izquierda, graficamos los tiempos de ejecución de VEC sobre el dataset **instancias-aleatorias** a la par de la función $f(n) = n^2$. A la derecha el gráfico de correlación de dichos datos, donde se presentó un $r = 0{,}98$.

Por los resultados observados en **Figura 2**, se comprueban nuestras hipótesis.

Heurística Inserción

Como hipótesis esperamos que la complejidad del algoritmo INS es $O(n^3)$.

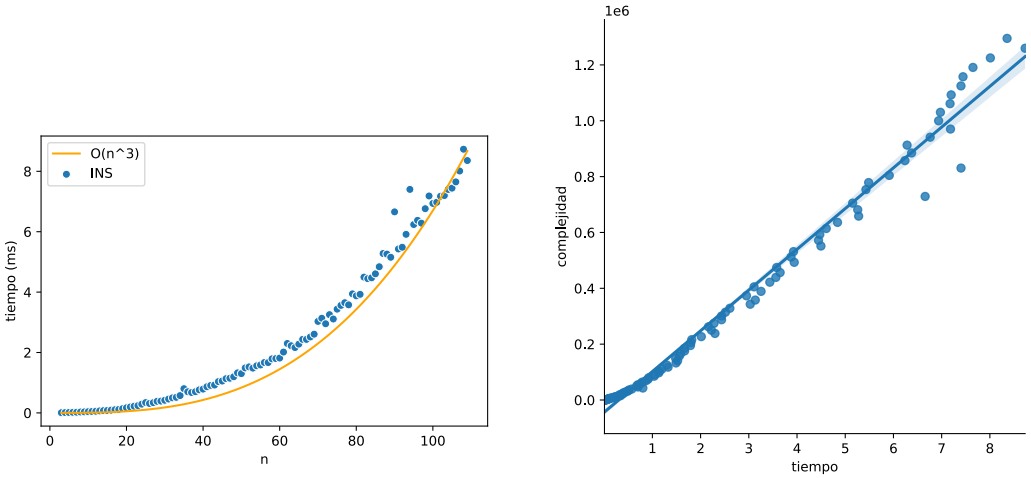

Figura 8: A la izquierda, graficamos los tiempos de ejecución de INS sobre el dataset **instancias-aleatorias** a la par de la función $f(n) = n^3$. A la derecha el gráfico de correlación de dichos datos, donde se presentó un $r = 0{,}99$.

Podemos confirmar con los gráficos de la **Figura 4** que el tiempo de ejecución del algoritmo INS se corresponde con lo que esperábamos.

Heurística AGM

Como hipótesis esperamos que la complejidad del algoritmo AGM sea de $O(n^2)$. Veamos en **Figura 5** los resultados de las experimentaciones.

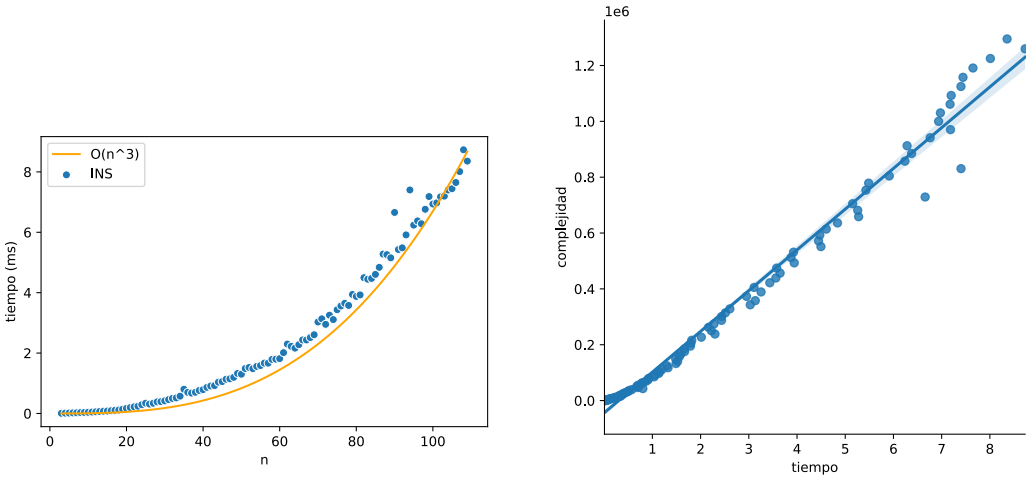

Figura 9: A la izquierda, graficamos los tiempos de ejecución de AGM sobre el dataset **instancias-aleatorias** a la par de la función $f(n) = n^2$. A la derecha el gráfico de correlación de dichos datos, donde se presentó un $r = 0{,}98$.

Por el crecimiento que presentan los tiempos de ejecución y el alto índice de correlación, se puede comprobar que AGM es $O(n^2)$.

Metaheurística Tabú Search

Como la metaheurística de tabú toma 4 parámetros de entrada, analizaremos cada uno fijando los otros 3. Para la complejidad con respecto al n, utilizamos las instancias anteriores. Para los parámetros, utilizamos la instancia de tsplib de ch130, a manera de fijar el n, fijando dos de los tres restantes y poniendo el ultimo a analizar en un rango de 0 a 100.

Memoria de Soluciones

Dado que la complejidad esperada $\mathcal{O}(iteraciones * porcentaje * logk * n^3)$, esperamos ver esa relación entre las variables y las funciones con las que las compararemos.

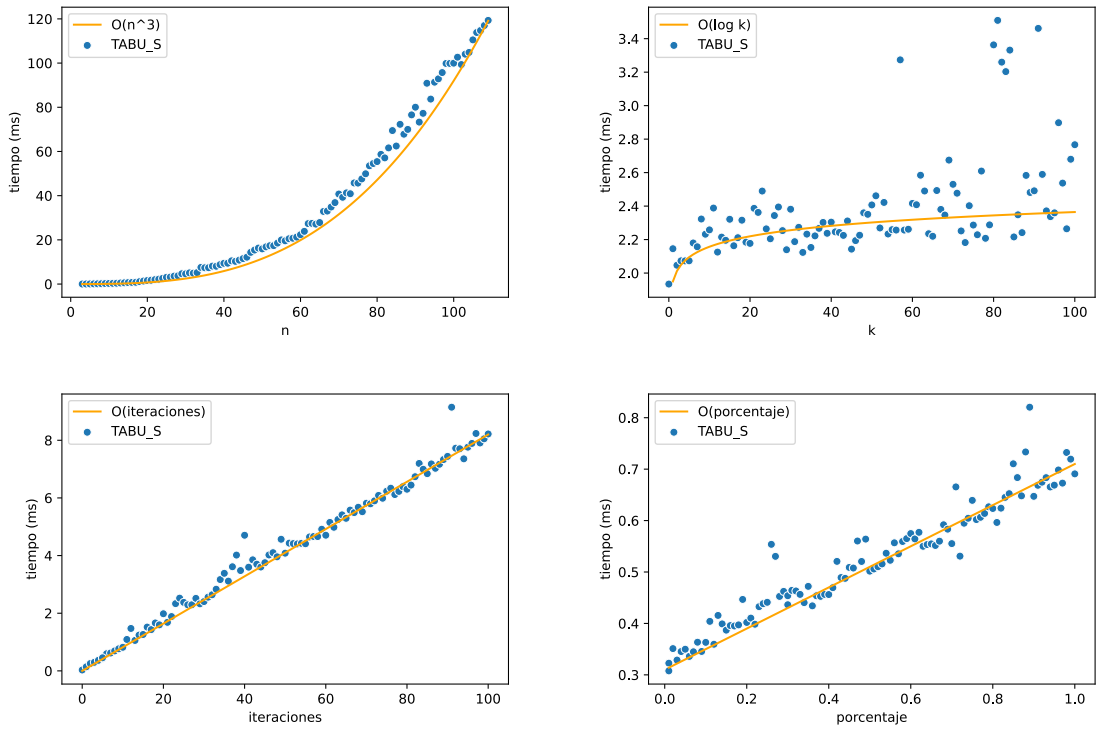

Figura 10: Gráficos de complejidad de Tabú con memoria de soluciones con respecto al tamaño de la entrada y sus parámetros dados.

A excepción del grafico de k en relación al tiempo, los otros confirmaron nuestra hipotesis propuesta con altos coeficientes de Pearson. El comportamiento del k no fuimos capaces de confirmarlo debido a problemas que nos encontramos al calcular su coeficiente de correlación. Si bien el gráfico que incluimos podría describir una función logarítmica, no nos alcanza con lo experimentado para afirmarlo.

Memoria de Aristas

Dado que la complejidad esperada $\mathcal{O}(iteraciones * porcentaje * logn * n^3)$, esperamos ver esa relación entre las variables y las funciones con las que las compararemos.

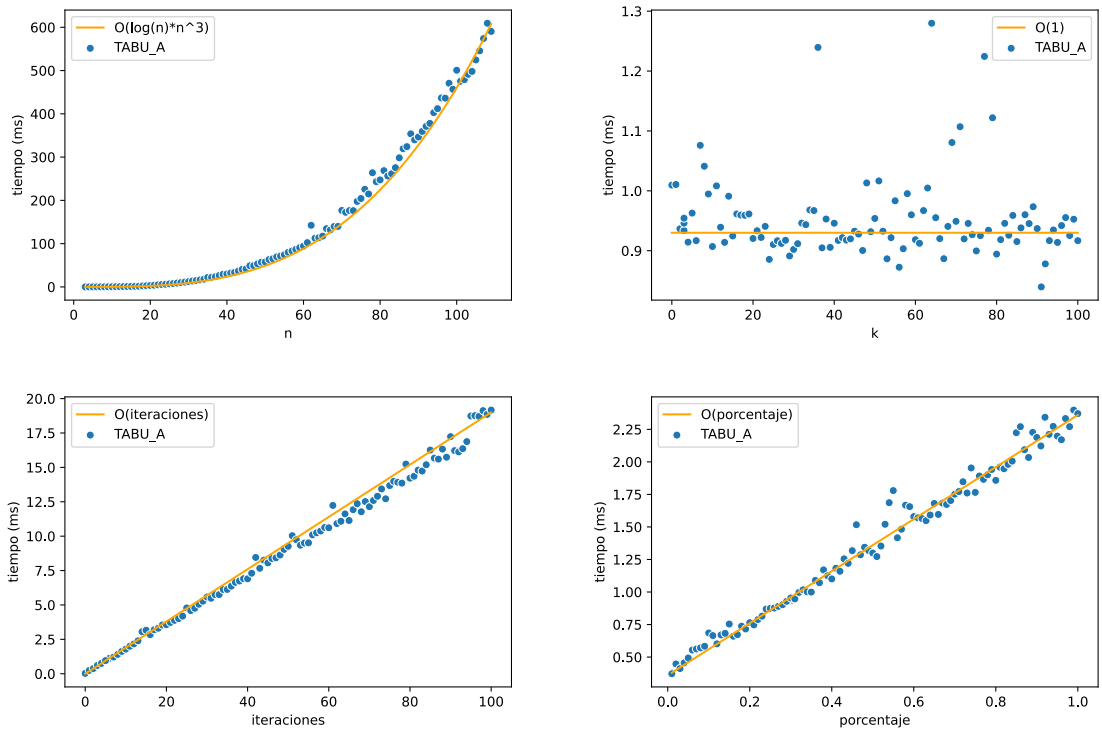

Figura 11: Gráficos de complejidad de Tabú con memoria de aristas con respecto al tamaño de la entrada y sus parámetros dados.

Como se puede apreciar en los gráficos, se cumplieron nuestras hipótesis. Todas la variables excepto el k, dieron un alto coeficiente de Pearson. Con la variable k, nos encontramos devuelta incapaces de calcularle su coeficiente. Se hace muy visible en el gráfico presentado que, a pesar de unos ciertos outliers, no parecería influir en el tiempo de ejecución. Sin embargo, no podemos afirmar esto con certeza debido a la falta de prueba de correlación.

Búsqueda mejores parámetros tabú

Para buscar los mejores parámetros de tabú, decidimos un rango arbitrario de cada parámetro excepto el n, y los corrimos con todas las combinaciones de esos rangos para 3 de las 4 instancias que utilizamos de **tsplib**, ya que estas tienen el óptimo calculado. Las tres que usamos son burma14, ch130 y berlin52. Analizaremos cuál combinación dio en promedio el mejor resultado. Para todas las instancias, utilizaremos el *gap relativo* como punto de comparación.

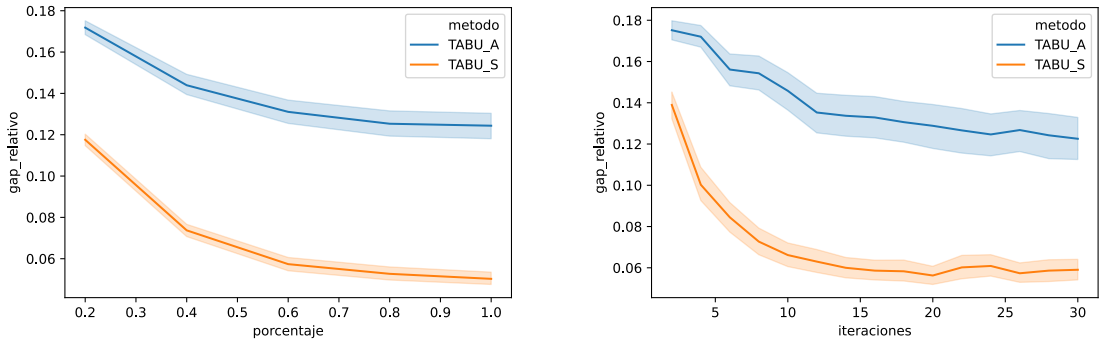

Figura 12: Gráficos de porcentaje e iteraciones con respecto al gap relativo. Representan el comportamiento general en las tres instancias donde se corrió el experimento.

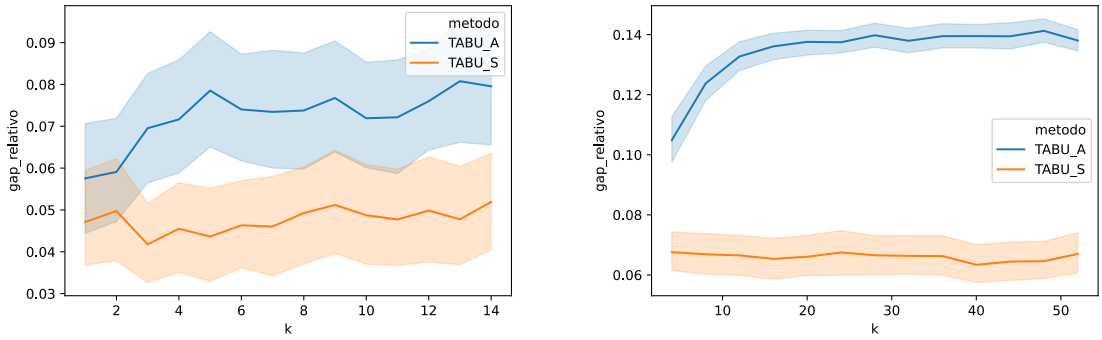

Figura 13: Gráficos de k con respecto al gap relativo. A la izquierda, se encuentra el grafico de las relaciones en burma14. A la derecha, el de berlin52.

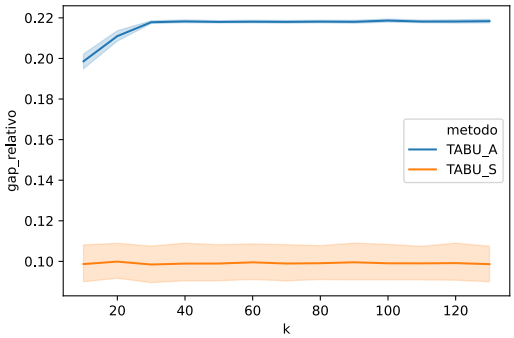

Figura 14: Gráficos de k con respecto al gap relativo, en relación a la instancia de de ch130

Vemos que con un k bajo, un *porcentaje* alto y una *cantIteraciones* alto con diminishing returns a medida que crece son los valores óptimos de tabú. Esto implicaría que la presencia de mínimos locales en *2-opt* es relativamente baja, y que es una buena heurística para encontrar una aproximación óptima independientemente de nuestra búsqueda tabú.

Es por esto, que determinamos como mejores parámetros k bajos, en el rango de 1 a 3. El mejor *porcentaje* es 1. Y, con respecto a las iteraciones, la idea seria dar la mayor cantidad posible sin incrementar mucho el tiempo de ejecución, pero se determinara a la relevancia entre el tiempo de computo y la necesidad de un resultado mas exacto, además de las capacidades de la computadora que lo corra.

5.5. Tiempo vs performance de las heurísticas

Para esta sección utilizaremos los datasets **aleatorias** e **instancias-tsplib**. En las instancias de tsplib, al conocer el resultado óptimo comparamos la calidad de los resultados mediante el $gap\_relativo = \frac{(obtenido - optimo)}{optimo}$. Para las instancias aleatorias observaremos la distribución de los resultados obtenidos, dado que no conocemos el óptimo. De esta manera podremos comparar sus medianas y los outliers mínimos.

En primer lugar, vemos los resultados con las instancias cuya solución óptima es conocida.

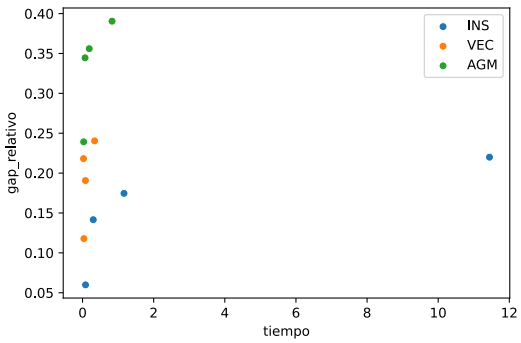

Figura 15: Vemos el gap relativo obtenido de las heurísticas sobre las **instancias-tsplib** respecto al tiempo de ejecución.

La heurística INS muestra buenos resultados con pequeño gap relativo, pero es la que más tiempo demora (esto tiene sentido dado que era $O(n^3)$). AGM tarda menos que INS, pero sus resultados son considerablemente peores que los de VEC, que tarda lo mismo porque las dos son $O(n^2)$. Esto nos lleva a pensar que VEC podría ser la mejor de las tres heurísticas.

Veamos el comportamiento de nuestras heurísticas con las instancias **aleatorias**.

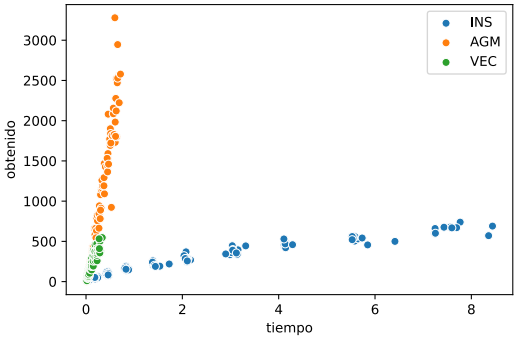

Figura 16: A medida que aumenta el n, los resultados de VEC se mantienen menores a los de las otras heurísticas.

Podemos comprobar que VEC presenta resultados considerablemente mejores, con un gap bajo y un tiempo de ejecución óptimo. A continuación un gráfico de boxplots donde se ve claro cómo los pesos de los caminos obtenidos por VEC presentan una distribución menor.

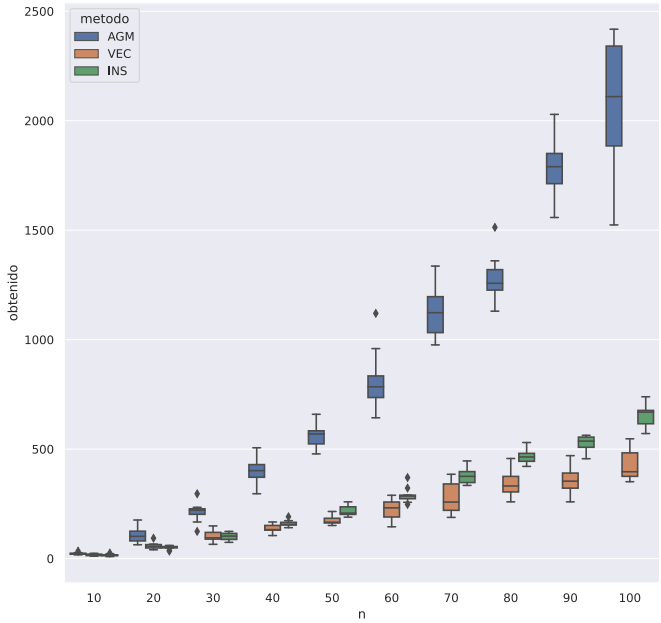

Figura 17: A medida que aumenta el n los resultados de VEC se mantienen menores a los del resto de las heurísticas.

Comparación entre metaheurísticas y heurísticas

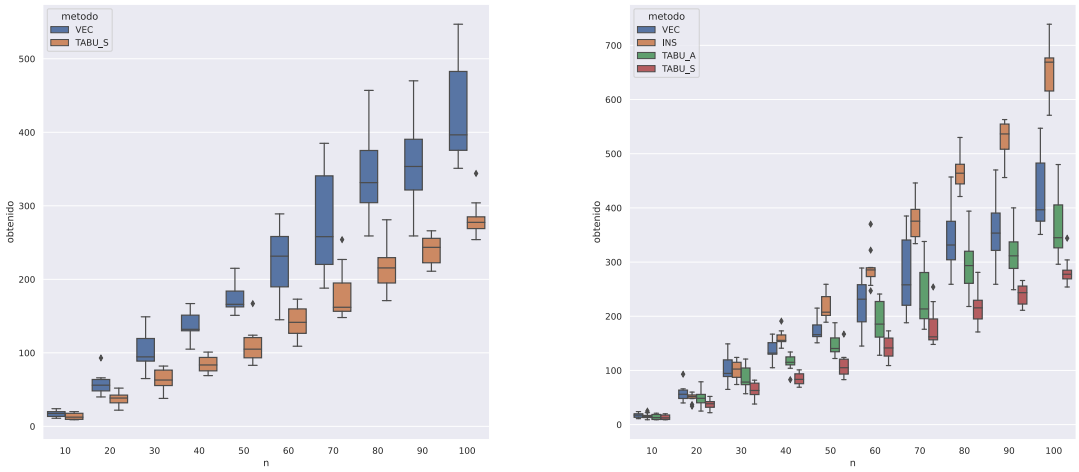

Figura 18: Caption
Como se puede observar, tabú tiene una performance significativamente superior a los otros métodos.

Finalmente, corrimos los métodos sobre otras instancias de **tsplib**, en particular, corrimos a tabú con sus mejores parámetros, que encontramos anteriormente. De esta manera esperamos encontrar una mejor performance del algoritmo tabú search.

6. Conclusión

Como el TSP es $NP - hard$ al encarar el problema tuvimos que, primero, diseñar heurísticas golosas constructivas para acercarnos lo máximo posible a un impacto máximo. En este punto notamos que la heurística secuencial obtuvo en casi todos los casos un impacto mayor y en un tiempo menor. Sin embargo, sabemos que ambos algoritmos pueden dar resultados tan alejados del máximo como se quiera, por lo que siempre la performance dependerá de la instancia.

Luego, al incurrir en el método metaheurístico tabú search obtuvimos resultados aún mejores partiendo de la heurística constructiva secuencial. El costo temporal es significativamente superior, pero dependiendo el contexto puede ser un algoritmo altamente preferible. Al diseñar este algoritmo mantuvimos muchos parámetros de entrada variables que modifican los tiempos de ejecución y la performance. Realizamos un análisis y obtuvimos una combinación de parámetros, pero éstos variaban según las instancias que se seleccionaban para training. Además la cantidad de instancias no era tan grande por lo

que agregó variabilidad. Contraria a nuestra hipótesis, los parámetros óptimos dieron ligeramente menor para k y mayor para *porcentaje* de lo que esperábamos ya que creíamos que iba a ser atrapado en mínimos locales de manera más frecuente de la encontrada usando el método de vecindario seleccionado.