DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP2: Travelling Salesman Problem

Grupo 21

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Rodriguez Celma, Guido	374/19	guido.rodriguez@outlook.com.ar
Loria, Damian Ezequiel	111/16	c03iif@hotmail.com
Itzcovitz, Ryan	169/19	ryanitzcovitz@gmail.com
Rodriguez, Miguel	57/19	mmiguerodriguez@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
https://exactas.uba.ar

Índice

1. Introducción **2**

2. Desarrollo **2**
2.1. Heurística: Vecino mas cercano 2
2.2. Heurística: Arista más corta 3
2.3. Heurística: Árbol generador mínimo 4
2.4. Metaheurística: Búsqueda tabú basado en últimas soluciones 4
2.5. Metaheurística: Búsqueda tabú basado en estructuras 5
2.6. Vecinos para búsqueda tabú: 2-opt 6

3. Experimentación **7**
3.1. Complejidad de las heurísticas 7
3.2. Familias de instancias 8
3.3. Parámetros de búsqueda tabú 9
3.4. Comparación de rendimiento 10

4. Conclusiones **13**

1. Introducción

El Problema del Viajante de Comercio (TSP, por sus siglas en inglés) en el que dado una lista de ciudades y las distancias entre cada una de ellas, debemos responder ¿cuál es la ruta más corta posible que visita cada ciudad exactamente una vez y al finalizar regresa a la ciudad origen? es un problema de optimización combinatoria muy estudiado en la investigación operativa y ciencias de la computación y es utilizado como prueba para muchos métodos de optimización. Lo interesante de este problema es la cantidad de situaciones reales que se pueden resolver a partir de esta idea, como por ejemplo, la logística de entrega de paquetes, la fabricación de circuitos electrónicos, secuenciación de ADN, etc.

Este problemas se puede definir formalmente a partir de estructuras ya vistas en la materia. Sea $G = (V, E)$ un grafo completo, donde cada arista $(i, j) \in E$ tiene asociado un 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. En nuestro caso, vamos a pedir que el primer vértice del ciclo sea el 1.

Dado que TSP pertenece a la clase de problemas $\mathcal{NP}$-hard, no buscaremos dar la solución óptima para cada instancia, sino que implementaremos distintas heurísticas para minimizar el peso del camino obtenido y las compararemos a partir de distintas métricas.

2. Desarrollo

En esta sección vamos a explicar los algoritmos implementados y las complejidades teóricas de cada uno de ellos. Dado un grafo $G = (V, E)$ definimos a $N = \#(V)$ como la cantidad de vértices del grafo y $M = \#(E)$ la cantidad de aristas del grafo. Dado que G es un grafo completo $M = \frac{(N-1)\times(N-2)}{2}$.

2.1. Heurística: Vecino mas cercano

La primer heurística que vamos a desarrollar es la mas intuitiva de todas, es un procedimiento goloso que va a realizar exactamente lo que dice su nombre, en cada paso busca cuál es el vértice no visitado más cercano al que podemos ir desde el último visitado.

El algoritmo desarrollado consta de dos ciclos para encontrar el circuito hamiltoniano de menor peso posible. Comenzamos con una lista H la cual contiene únicamente al vértice inicial y para cada iteración, buscamos del último nodo agregado en la lista H cuál de todos sus vecinos tiene el menor peso y no forma ningún circuito al agregarlo a la lista. Para ver si una arista (v, w), con $v \in H$, pertenece a un circuito simplemente retornamos el valor de una lista booleana donde la i-ésima posición dice si el vértice i fue agregado o no obteniendo ese resultado en $\mathcal{O}(1)$. Luego de haber encontrado el camino que conecta a los N nodos sin repetir, sumamos el peso de la arista que nos falta para cerrar el ciclo.

Complejidad: La complejidad del algoritmo está relacionada directamente con el primer ciclo ya que el segundo consta de recorrer entre todos los vecinos del último encontrado, el peso de la arista que lo conecta con el vértice inicial y eso es $\mathcal{O}(N)$ en el peor caso ya

que cada nodo está conectado con todos por ser un grafo completo. Ahora solo resta ver la complejidad del primer ciclo y sacar las conclusiones finales.

El ciclo consta de N iteraciones, una por cada vértice encontrado. En cada iteración vamos a recorrer los vecinos del último visitado v y encontrar el w que no esté en el circuito y (v, w) sea la arista de menor peso. Calcular el peso de la arista y ver si no esta en el circuito lo hacemos en $\mathcal{O}(1)$, mientras que recorrer todos los vecino se hace en $\mathcal{O}(N)$. Por lo que la complejidad del algoritmo es $\mathcal{O}(N^2)$ ya que para cada nodo vamos a recorrer todos sus vecinos buscando el de menor peso.

2.2. Heurística: Arista más corta

Nuestro algoritmo puede dividirse en dos secciones, una primera en que se hallan las aristas que forman el circuito hamiltoniano y otra en que se construye el circuito propiamente dicho.

Para hallar las aristas que forman el circuito realizamos un procedimiento goloso, basado en el algoritmo de Kruskal. Iterativamente iremos agregando la arista más corta del grafo que no forme ciclos en nuestro circuito en construcción, en total realizaremos $N - 1$ iteraciones, pues por definición el circuito pasa una única vez por cada vértice.

En cada iteración recorremos cada vértice de G y sus vértices adyacentes. Para cada uno se calcula su grado en el circuito actual y se verifica que agregar la arista que los une no forme un ciclo en el mismo. Si el grado de ambos es menor o igual a 1, su arista no forma ciclos y el peso de la misma es el menor encontrado en esta iteración, se la marca como la arista de menor peso. Al final de la iteración se agrega la arista marcada al vector de aristas, y se suma su peso al total del circuito.

Al terminar las $N - 1$ iteraciones agregamos la arista que cierra el circuito. Notar que obtuvimos un vector de todas las aristas que pertenecen al circuito hamiltoniano, sin ningún tipo de orden entre ellas. En el resto del algoritmo nos dedicamos a darle el orden correspondiente al circuito.

Primero encontramos la arista inicial, en este caso buscamos la que contenga al vértice 1. Luego recorremos el vector de aristas a partir de la inicial, agregándolas en orden al circuito resultado. Finalmente, retornamos el circuito hamiltoniano obtenido junto a su peso total.

Complejidad: El proceso de encontrar la menor arista se repite $\mathcal{O}(N)$ veces. En cada iteración, se recorren todos los vértices de G y para cada uno, todos sus vecinos, esto tiene costo $\mathcal{O}(N^2)$. Calcular el grado de cada vértice y sus vecinos en el circuito cuesta $\mathcal{O}(N)$ en peor caso y verificar que una arista no forma ciclos en el camino lleva $\mathcal{O}(N^2)$, pues el camino tiene a lo sumo N aristas y para cada una se debe recorrerlo en su totalidad.

Luego, encontrar todas las aristas que forman el circuito hamiltoniano nos cuesta $\mathcal{O}(N \times (N \times (N^2 \times (N + N^2))) \in \mathcal{O}(N^5)$. El formateo del circuito requiere que recorramos el mismo en su totalidad por cada arista que lo conforma (en peor caso), esto tiene costo $\mathcal{O}(N^2)$. Por lo tanto, la complejidad total de nuestro algoritmo es de $\mathcal{O}(N^5)$.

2.3. Heurística: Árbol generador mínimo

Otra de las heurísticas implementadas, fue la basada en árbol generador mínimo. La idea de esta heurística se basa en utilizar el AGM del grafo G para armar el circuito hamiltoniano.

Primero, lo que hacemos es obtener el AGM T de G, una vez obtenido esto, recorremos el árbol T con DFS y nos guardamos el recorrido. El recorrido de DFS se construye salteando los vértices ya visitados, por lo que al final vamos a terminar teniendo un circuito hamiltoniano.

Notemos que el peso del AGM T es una cota inferior de la solución de TSP, esto se debe a que la solución de TSP incluye un árbol, y en el mejor caso, puede ocurrir que este árbol sea el generado por AGM, sumando el peso de la arista que conecta el final del árbol con el inicio.

Complejidad: Nuestro algoritmo calcula el AGM del grafo de entrada. Hasta no tener a todo el conjunto de aristas que conforman el AGM, vamos a estar agregando las aristas con menor peso que no formen un circuito con las ya agregadas. El costo de ver todas las aristas del grafo hasta llegar a armar el AGM sin formar circuitos es $\mathcal{O}(N^2 \times M)$.

Una vez obtenido el AGM de G, recorremos este árbol con DFS. Hasta no tener el camino, vamos a agregar al camino actual un vértice tal que exista una arista entre él y el último agregado al camino y que no se encuentre actualmente en el camino. Esto lo hacemos hasta que nos quedamos sin vecinos por visitar, por lo que cerramos el circuito hamiltoniano. Luego, el costo final es $\mathcal{O}(N)$, que termina siendo acotado por el costo de calcular el AGM de G. Finalmente, nos resta calcular el peso total del camino que es costo $\mathcal{O}(M)$ ya que por la forma en la que armamos el árbol DFS, en peor caso vamos a terminar recorriendo todas las aristas del grafo G.

El costo final de la heurística de AGM es $\mathcal{O}(N^2 \times M) \in \mathcal{O}(N^4)$.

2.4. Metaheurística: Búsqueda tabú basado en últimas soluciones

Búsqueda tabú es una metaheurística que guía una heurística de búsqueda local, para explorar el espacio de soluciones y evitar quedar atascados en un óptimo local. Funciona de manera similar a la búsqueda local ya que iterativamente se mueve de una solución a otra hasta que se cumpla algún criterio de terminación.

Este tipo de búsqueda tiene asociado a su vez una memoria, conocida como "lista tabú" que es utilizada para almacenar las $|T|$ ultimas soluciones visitadas. Esto nos permite marcar a estas soluciones como "tabú" y evitar volver a considerarlas en el corto plazo.

Para nuestra implementación, utilizamos 3 parámetros al ejecutar el algoritmo de búsqueda tabú en el problema de TSP:

- Tamaño de la memoria: Cantidad de soluciones previamente observadas que vamos a guardar e ir sobrescribiendo en cada iteración.
- Cantidad de vecinos: Cantidad de soluciones que vamos a explorar en la vecindad de la solución actual.
- Cantidad de iteraciones: Todas las veces que vamos a iterar buscando una solución óptima hasta frenar la ejecución.

Al iniciar la ejecución del algoritmo, utilizamos primero la solución de alguna de las heurísticas anteriormente explicadas, en nuestro caso vamos a usar la de AGM.

Luego, a partir de los parámetros explicados anteriormente, iteramos `cant_iteraciones` veces en búsqueda de una mejor solución. En cada paso, buscamos dentro de la vecindad de la solución actual, a alguno de entre los `cant_vecinos` mejores que no se encuentre en la memoria, de manera aleatoria. Luego, si este vecino encontrado tiene menor peso que la mejor solución encontrada hasta el momento, modificamos la solución óptima.

Complejidad: Encontrar una solución válida con el algoritmo de Árbol Generador Mínimo tiene costo $\mathcal{O}(N^4)$. Luego el ciclo principal de esta metaheurística se realiza $\mathcal{O}(cant\_iteraciones)$ veces. En el mismo se construye una vecindad, mediante el proceso de rotación de aristas implementado como `2-opt`, el cual tiene un costo del orden $\mathcal{O}(N^3)$. Seguido de eso se elige uno de los `cant_vecinos` mejores vecinos de la misma que no estén en memoria, esto tiene costo $\mathcal{O}(cant\_vecinos \times tam\_memoria \times N)$. El resto del algoritmo consiste en comparaciones y asignaciones de enteros, con costo constante. La complejidad total del algoritmo es de $\mathcal{O}(N^4 + cant\_iteraciones \times (N^3 + cant\_vecinos \times tam\_memoria \times N))$.

2.5. Metaheurística: Búsqueda tabú basado en estructuras

Otra forma de realizar la búsqueda tabú es basarse en estructuras, a diferencia de la otra versión que guarda la solución completa. En la lista tabú que utilizaremos como memoria van a estar los dos pares de vértices que fueron alterados (swaps), es decir, las 2 aristas que modificamos. Este intercambio de aristas va a estar definido por el algoritmo de búsqueda local `2-opt`, por lo que en las siguientes iteraciones vamos a evitar repetir los $|T|$ swaps que se encuentren en la memoria.

Esta variante tiene su lado positivo como negativo. La positivo es que al guardar únicamente el intercambio de aristas, cuando tengamos que guardar en la memoria si este intercambio ya fue realizado, la complejidad espacial se va a ver reducida ya que guardaremos mucha menos información, lo mismo sucede con la complejidad temporal ya que no hay que revisar todo el ciclo nodo por nodo para calcular el peso de este nuevo ciclo ya que calcular el peso de la nueva configuración se puede hacer restando el peso de las aristas que estamos sacando y sumarle el peso de las que estamos agregando en el swap. Lo negativo es que al tener únicamente el intercambio de aristas en la memoria, no vamos a saber nada sobre el ciclo en general por lo que si en algún momento queremos realizar un intercambio que esta actualmente en la memoria vamos a descartarlo aunque los ciclos obtenidos hubiesen sido completamente distintos.

Complejidad: Al tener el mismo algoritmo que búsqueda tabú basado el soluciones, la complejidad se calcula de manera similar. Lo único que cambia es el cálculo de la vecindad, ya que solamente guarda las 2 aristas que se van a intercambiar. Esta parte de calcular el mejor vecino es mucho mas rápido ya que solamente iteramos por toda la vecindad calculando el peso del ciclo como detallamos anteriormente. Esta operación de calcular el peso del ciclo se realiza en tiempo constante, obteniendo una función de mejor vecino en un tiempo relacionado al tamaño de la vecindad y la memoria. Como en el otro algoritmo, vamos a realizar una cantidad de iteraciones de búsqueda del mejor vecino definida por parámetros. La complejidad

total del algoritmo es de $\mathcal{O}(N^4 + cant\_iteraciones \times (N^2 + cant\_vecinos \times tam\_memoria))$. Si bien la complejidad es $\mathcal{O}(N^4)$, en la práctica va a depender de los parámetros utilizados en la ejecución como: cantidad de iteraciones, tamaño de memoria y tamaño de la vecindad.

2.6. Vecinos para búsqueda tabú: 2-opt

El algoritmo de búsqueda tabú se basa en calcular la vecindad de un ciclo utilizando una técnica de búsqueda local para poder seguir explorando otros ciclos que puedan mejorar las soluciones obtenidas. Una forma de obtener una vecindad en base a una solución s es con el algoritmo de búsqueda local `2-opt`. Este procedimiento funciona de la siguiente manera:

Dado una lista de vértices s de tamaño n vamos a recorrer cada vértice desde el segundo ($i = 1$), por cada vértice i veremos desde el $i + 1$-ésimo en adelante (vamos a llamarlo j al subíndice que arranca desde $i + 1$ hasta n) hasta el final y realizar la siguiente operación. Vamos a dividir la lista en 3 partes. Desde el índice 0 al $i - 1$ (inicio), del i al j (medio) y del $j + 1$ hasta n (fin). Luego el nuevo camino obtenido para i, j sera el concatenar inicio, el medio dando vuelta la lista y el fin.

Lo que conseguimos con esto es una vecindad del ciclo s donde cada vecino se forma tomando dos aristas $(i - 1, i)$ y $(j, j + 1)$ intercambiando sus extremos, obteniendo 2 aristas nuevas $(i - 1, j)$ e $(i, j + 1)$ como se puede observar en la siguiente imagen.

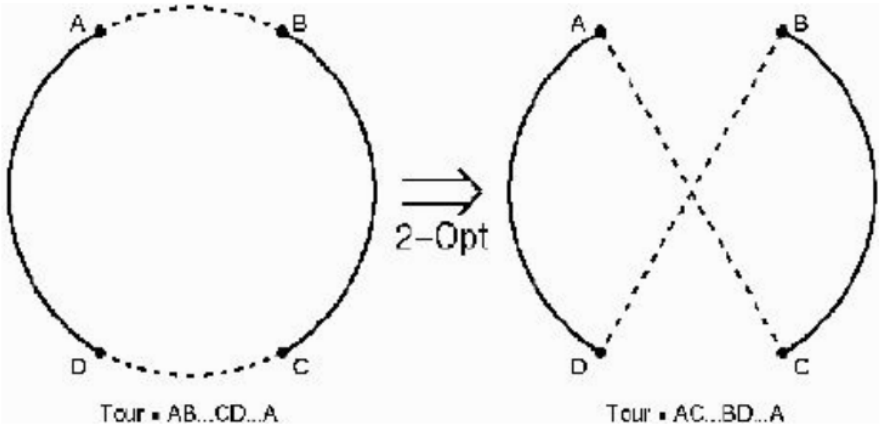

Figura 1: Búsqueda local: 2opt

3. Experimentación

En esta sección vamos a hacer distintos experimentos modificando parámetros de entrada y utilizando distintas configuraciones para los algoritmos. La idea es comparar métricas como cercanía a la solución óptima o tiempo de ejecución y comparar las heurísticas y metaheuristicas implementadas.

3.1. Complejidad de las heurísticas

La idea de este experimento es verificar que las complejidades calculadas anteriormente para las heurísticas "Vecino mas cercano", "Arista mas corta" y "Árbol generador minimo" son correctas. En los siguientes gráficos, podemos ver que esto ocurre.

Los gráficos observados fueron generados a partir de ejecutar las distintas heurísticas incrementando la cantidad de nodos del grafo de forma lineal. Cada algoritmo hizo su ejecución 10 veces por cada grafo y obtuvimos la media de los tiempos para prevenir outliers.

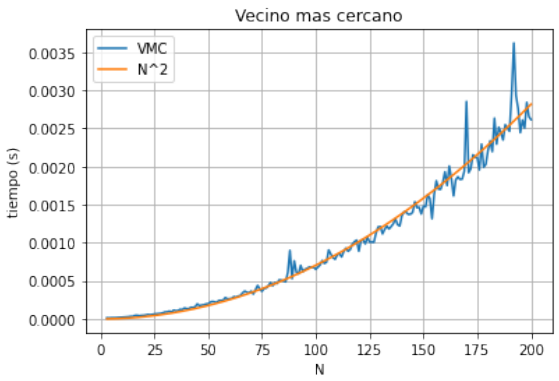

Figura 2: Complejidad heurística VMC

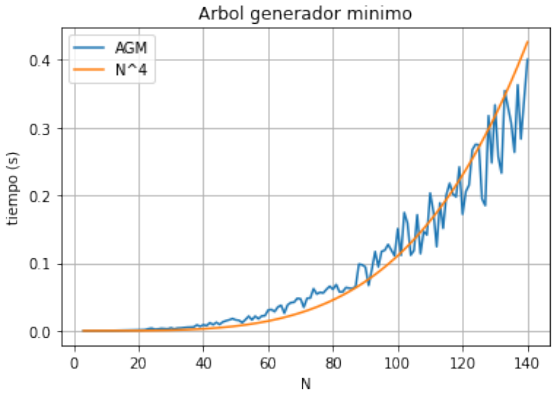

Figura 3: Complejidad heurística AGM

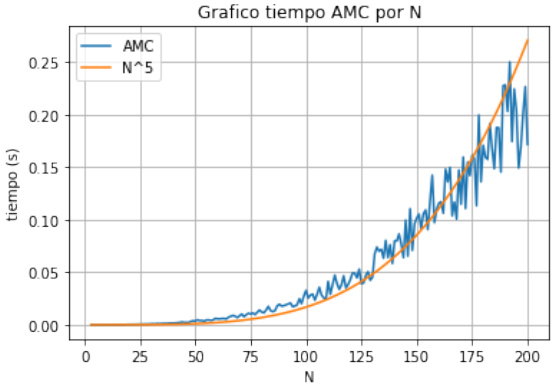

Figura 4: Complejidad heurística AMC

Como podemos ver en las figuras, todas las complejidades de los algoritmos con heurísticas golosas y árbol generador mínimo se corresponden con los vistos en la sección 2.

3.2. Familias de instancias

Por lo visto anteriormente, los algoritmos *Arista mas corta*, *Vecino mas cercano* y *Arbol generador mínimo* son heurísticas golosas en las que en cada paso van descartándose soluciones factibles que no sean compatibles con la decisión golosa tomada. Es decir, las aristas elegidas en cada paso no se descartan del circuito para que se exploren alternativas distintas. Esto significa que el método no saldrá de óptimos locales, por lo tanto no retornará una solución óptima para cualquier instancia en que el óptimo local encontrado no lleve a un óptimo global.

El costo de las soluciones obtenidas por estas heurísticas no tienen cota superior respecto al de la solución óptima. Es decir, pueden fabricarse casos en que el costo del circuito obtenido sea tan grande como se quiera, y el del óptimo tan chico como se quiera. Para estos experimentos, vamos a generar instancias para las cuales sabemos su peso óptimo y fabricaremos un camino no óptimo por el cual el algoritmo goloso va a generar su solución.

Arista mas corta: Lo que hacemos en este caso es un grafo en el cual, el algoritmo deba tomar alguna arista de peso extremadamente grande para poder continuar con su ejecución. Para el grafo en cuestión, el algoritmo agrega las aristas `(0, 1)` y `(1, 2)` al ciclo que esta armando, por lo que luego nunca va a poder agregar la arista `(1, 2)` que forma parte del circuito óptimo ya que si no generaría un ciclo. Luego lo que ocurre es que se agregan las aristas `(3, 4)` y `(2, 3)` por lo que no queda opción que utilizar la arista `(1, 4)` ya que el resto generarían ciclos que no cierran el circuito hamiltoniano. En las siguientes figuras podemos ver el grafo con el camino óptimo y el camino generado por el algoritmo en cuestión.

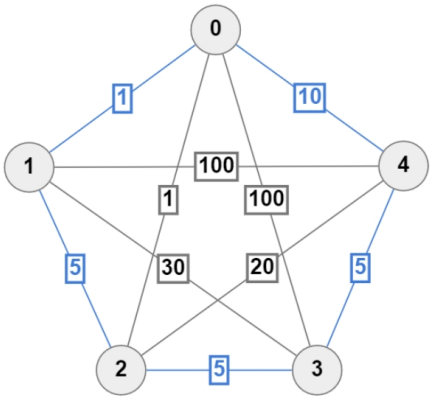

Figura 5: Camino óptimo, peso 26

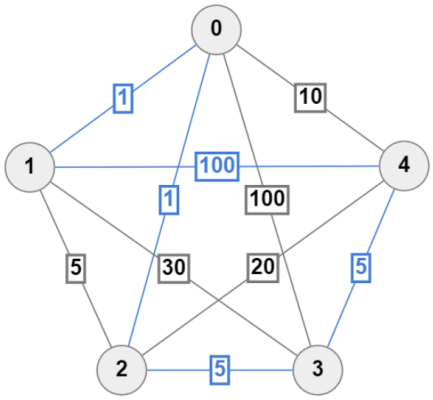

Figura 6: Camino AMC, peso 112

Vecino mas cercano: Este algoritmo empieza desde el primer vértice y sigue el camino a partir de ir agregando al vecino con menor peso sin que se genere un ciclo. En el ejemplo de la figura 8 podemos ver que la primeras dos aristas agregadas al camino son las aristas `(0, 1)` que no forma parte del camino óptimo y la arista `(1, 3)`. Una vez que el algoritmo llega al vértice 3, únicamente puede seguir por la arista `(2, 3)` ya que sino estaría formando un ciclo. Luego, se termina de armar el ciclo hamiltoniano pero con un peso mucho mayor al óptimo ya que podemos ver que el gap de esta solución es del 671 %.

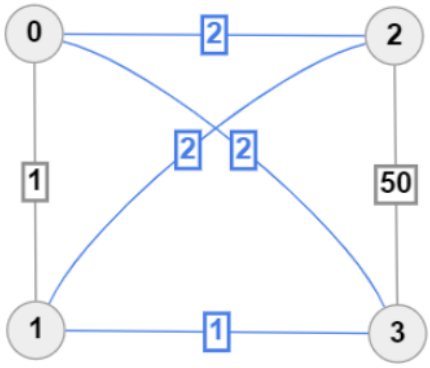

Figura 7: Camino óptimo, peso 7

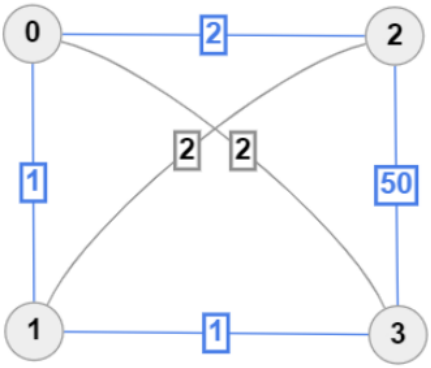

Figura 8: Camino VMC, peso 54

3.3. Parámetros de búsqueda tabú

Realizaremos una experimentación con el objetivo de encontrar valores para los parámetros de entrada tales que optimicen la ejecución de búsqueda tabú (en sus dos versiones), es decir, que produzcan una solución lo más cercana a la óptima en el menor tiempo posible.

Para esto ejecutaremos los algoritmos de busqueda tabú, sobre el conjunto de datos `berlin52`, para el cual conocemos su valor óptimo, variando los valores de los parámetros de entrada de forma sistemática y compararemos los resultados obtenidos.

Identificaremos con `TS` a la implementación con memoria basada en soluciones y con `TSA` a la que tiene memoria basada en estructura.

Resultados:

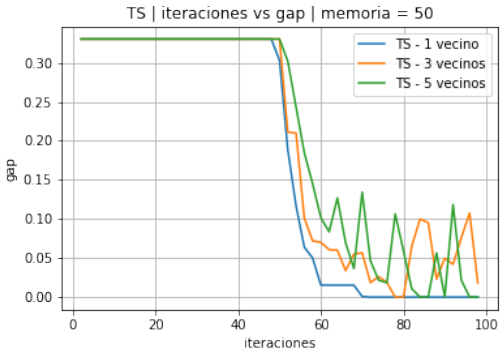

Figura 9: Gap según memoria por soluciones

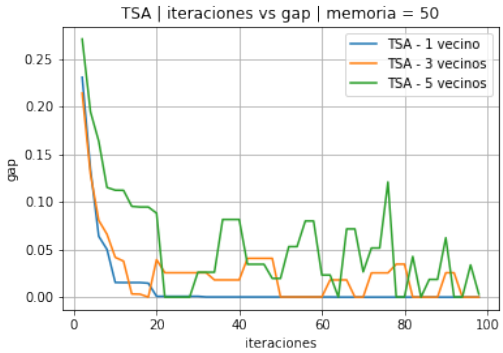

Figura 10: Gap según memoria por estructura

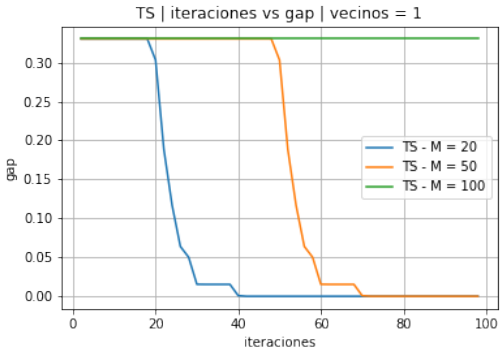

Figura 11: Gap según memoria por soluciones

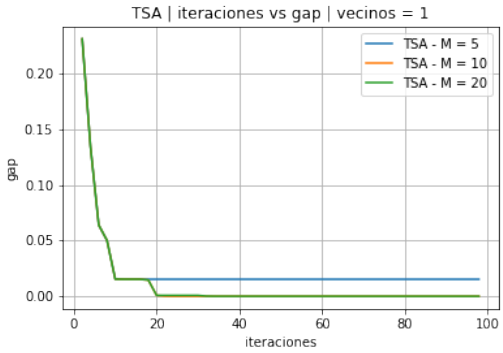

Figura 12: Gap según memoria por estructura

Análisis: Para ambas implementaciones observamos que, para este conjunto de datos explorar alguno de los mejores vecinos encontrados en lugar del mejor no presenta una mejoría en la solución obtenida, siendo peor en algunos casos. Además el tamaño de memoria óptimo para el dataset, considerando que se explora el mejor vecino en cada iteración, es de 20 soluciones en el caso de memoria por soluciones y de 10 pares de aristas en el caso de memoria por estructura.

3.4. Comparación de rendimiento

En esta sección compararemos la efectividad de los distintos métodos implementados en términos de calidad de la solución obtenida y tiempo de ejecución.

En primera instancia compararemos `costo obtenido vs tiempo de ejecución`. Para esto ejecutaremos los 5 métodos sobre 30 instancias generadas aleatoriamente de un grafo completo de 50 vértices.

Luego comparamos `tiempo de ejecución vs tamaño de entrada`. Ejecutaremos los 5 métodos sobre grafos completos generados aleatoriamente de tamaño incremental. En cada iteración tomamos la mediana de los tiempos obtenidos para 5 ejecuciones de los algoritmos.

Finalmente, veremos `costo obtenido vs tamaño de entrada`. En este caso analizaremos las heurísticas `Arista Más Corta` y `Tabú Search` con memoria de estructura. Ejecutaremos cada una sobre 30 instancias de grafos completos de tamaño incremental, generados aleatoriamente.

Los parámetros de entrada usados para Búsqueda Tabú serán los definidos en la sección **3.3**. Estos son:

- `Memoria:` soportará hasta 20 pares de aristas.
- `Vecinos:` se explora únicamente el mejor vecino encontrado.
- `Iteraciones:` se realizarán 40 iteraciones del ciclo principal.

Hipótesis: Creemos que la metaheurística de búsqueda tabú será la que obtendrá los mejores resultados en términos de costo, aunque esperamos que tenga un mayor tiempo de ejecución que las demás heurísticas ya que, en particular, se vale de AGM para encontrar un óptimo local para luego explorar el espacio de soluciones.

Entre las heurísticas golosas, no esperamos ver grandes variaciones tanto en costo como en tiempo de ejecución. Creemos que `AGM` y `AMC` obtendrán soluciones de costo particularmente similar ya que ambas se basan en encontrar aristas con el menor peso posible.

Resultados:

Costo vs Tiempo de ejecución

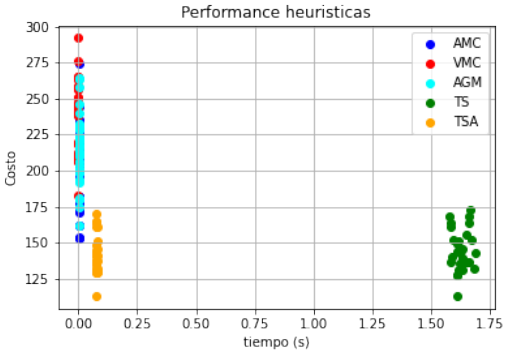

Figura 13

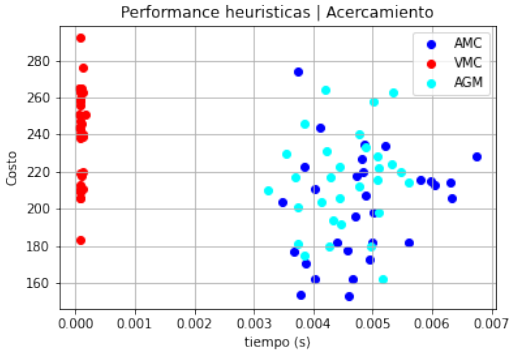

Figura 14

Tiempo de ejecución vs Tamaño de instancias

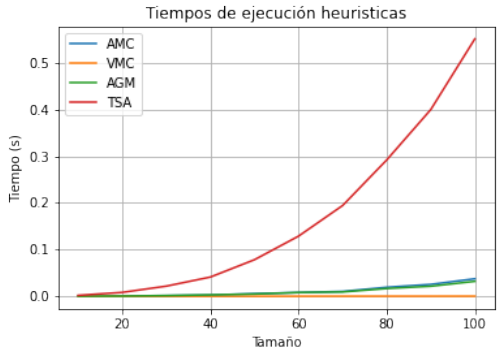

Figura 15

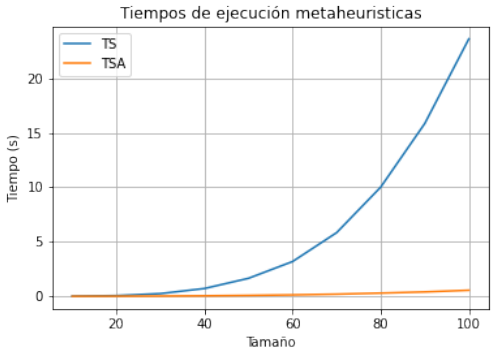

Figura 16

Costo vs Tamaño de instancias

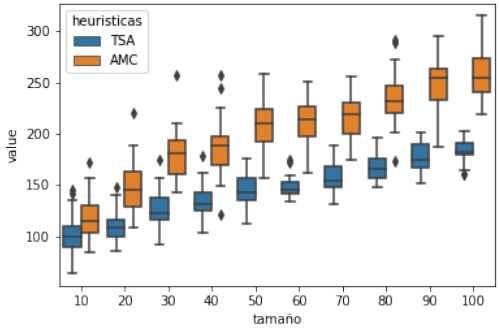

Figura 17

Análisis: Estudiemos los resultados obtenidos en las distintas comparaciones:

- `Costo vs Tiempo de ejecución:` En la figura 13 observamos que la metaheurística de `Búsqueda Tabú`, en ambas versiones, obtuvo soluciones de costo considerablemente menor a las obtenidas por las demás heurísticas. Notamos además que hubo una gran diferencia de tiempo de ejecución entre la implementación con memoria basada en soluciones y la basada en estructura.

  En la figura 14 vemos qué los 3 métodos golosos obtuvieron soluciones de costos similares, aunque `Vecino Más Cercano` tuvo costos algo más altos que las demás. La diferencia en los tiempos de ejecución es insignificante, aunque cabe destacar que `Vecino Más Cercano` fue el algoritmo más rápido.

- `Tiempo de ejecución vs Tamaño de instancias:` En la figura 15 vemos que los tiempos de ejecución de los algoritmos golosos parece crecer muy lentamente y de forma lineal conforme incrementa el tamaño del grafo de entrada, mientras que la metaheuristica `TSA` parece crecer de forma exponencial (con una constante muy baja).

  Comparamos las dos metaheurísticas en la figura 16, vemos que el tiempo de ejecución de `TS` crece de manera mucho más rápida que el de `TSA` a medida que se aumenta el tamaño de las instancias.

- `Costo vs Tamaño de instancias:` En la figura 17 vemos en la distribución de los costos, que aquellos obtenidos por `AMC` son bastante mayores que los de `TSA` y además tiene una mayor dispersión. Es decir que `TSA` obtiene mejores resultados y de manera más consistente.

4. Conclusiones

En el presente trabajo vimos posibles heurísticas que nos acercan a la solución de TSP. Obtuvimos ejemplos de redes en las que se puede llevar por caminos incorrectos a las heurísticas y obtener caminos poco óptimos. A su vez, implementamos algoritmos de búsqueda tabú, que nos permiten salir de óptimos locales e intentar llegar a un óptimo global.

En general, buscar el camino óptimo o un gap muy pequeño puede resultar muy beneficioso ya que, aunque estemos agregando complejidad o iteraciones a la ejecución del algoritmo, una reducción del gap puede resultar en un ahorro significativo de costos para un sistema de distribución, kilometros de cables para una red o cualquier problema que se pueda resolver con esta idea.