Universidad
Carlos III de Madrid

Grado en Ingeniería Informática

Curso 2020/2021

Teoría Avanzada de la Computación

Optimización combinatoria

Autores:

Iván Miguélez García	*100383387*
Alba Reinders Sánchez	*100383444*
Alejandro Valverde Mahou	*100383383*

Índice

1. Introducción

En esta práctica se plantea resolver el TSP (*Travelling Salesman Problem*), también conocido como problema del viajante, a través de dos algoritmos distintos. El primer algortimo es DFS, basado en búsqueda en profundidad, mientras que el segundo algoritmo utiliza un algoritmo *greedy*, para aplicar búsqueda local y tratar de mejorar la solución obtenida con el operador 2-opt. Para desarrollar ambos algoritmos, se va a utilizar *C++*, ya que ofrece mayor rendimiento que otros lenguajes de programación.

El objetivo es desarrollar y analizar estos algoritmos para resolver el problema de optimización combinatoria TSP. Éste es un problema que plantea qué camino tomar dadas una serie de ciudades conectadas, empezando y terminando en la misma ciudad, de forma que la solución tenga la menor distancia posible. La solución de este problema consiste en encontrar el ciclo Hamiltoniano que tenga el menor tamaño. Encontrarlo no es una tarea trivial, dado que el número de caminos posibles crece factorialmente al añadir una ciudad más.

El TSP se considera como un problema NP-completo, y es uno de los algoritmos más estudiados, para el cuál existen numerosos métodos de resolución y heurísticas. Para esta práctica se va a realizar un acercamiento por fuerza bruta a través del algoritmo de profundidad, un acercamiento ligeramente más sofisticado con una poda sobre la fuerza bruta y una nueva implementeación con búsqueda local.

La memoria se divide en dos apartados principales, uno para el algoritmo DFS y otro para el algoritmo de búsqueda local. En cada apartado se estudia el coste computacional de cada algoritmo mediante un estudio analítico y empírico del mismo. Además, en el algoritmo de DFS se analizan dos variantes, uno básico y uno al que se le aplica poda.

Para permitir la replicabilidad de los resultados, así como para poder comprobar los resultados obtenidos, se va a hacer uso de un cuaderno de *Python* en la herramiento de *Google Colab*, donde se ejecutarán los distintos algoritmos y se generarán las distintas gráficas comparativas.

2. Algoritmo de búsqueda iterativa

El algoritmo que se plantea es DFS (*Depth First Search*). Este algoritmo consiste en expandir desde el nodo raíz, que representa la posición inicial, todos sus nodos hijos. A continuación, se expanden todos los nodos del primer hijo. Esto se repite por el mismo camino hasta que termina y vuelve al origen. Una vez termina, va iterando hacia atrás y realiza el mismo proceso con todos sus nodos hermanos.

Para la implementación de este algoritmo se ha hecho uso de una pila de nodos expandidos, y cada vez que un nodo generaba nuevos hijos, se añadían a la pila. Cada camino se completa cuando no es capaz de generar más hijos porque ya ha visitado todos los nodos.

Este algoritmo, al aplicarse en el problema del TSP, genera todos los ciclos Hamiltonianos posibles comenzando en un mismo punto y, por tanto, es completo. También se puede asegurar su optimalidad ya que, al generar todos los caminos, y ser todos del mismo tamaño, tan solo hay que almacenar el camino que genera mejores resultados.

Al tener que expandir todos los nodos, el tiempo de computo aumenta considerablemente según el tamaño del problema crece, es decir, cuando se añaden más ciudades el número de nodos a generar crece muy rápido. Esto hace que se busque algún método para reducir el número de nodos a generar. Esta mejora consiste en realizar una poda del árbol de búsqueda cuando el coste acumulado del camino es mayor que el costed del mejor camino ya encontrado.

2.1. Coste Computacional DFS básico

2.1.1. Estudio Analítico

Usando n como el número de ciudades del problema, el número de ciclos Hamiltonianos que se pueden realizar es $(n-1)!$ porque tanto la primera como la última ciudad son la misma y son fijas. En esta implementación no existe un peor caso, ya que es necesario expandir todos los nodos en todos los casos, independientemente de si el mejor camino es el primero o el último.

En el código se puede ver que la función que lleva a cabo la búsqueda en profundidad está formada por un bucle *while* que se ejecuta mientras la pila de nodos no visitados no esté vacía. En su interior hay 2 bucles *for* anidados que se realizan ambos n veces, el primero crea los hijos para cada nodo y el anidado actualiza la lista de nodos visitados y la lista con el orden nuevo. El segundo bucle se encuentra dentro de un *if* cuya condición es verdadera tantas veces como ciudades queden por visitar.

Analizando el funcionamiento del código, se puede ver que el *while*, junto con el primer *for* se realiza tantas veces como nodos sean creados. El número de nodos creados depende del número de ciudades iniciales. Si se entienden los caminos como un árbol, en el primer nivel habría $n-1$ nodos, en el segundo $(n-1)*(n-2)$ nodos, etc. En la figura 1 se ve con más detalle esta explicación para un ejemplo con 4 ciudades.

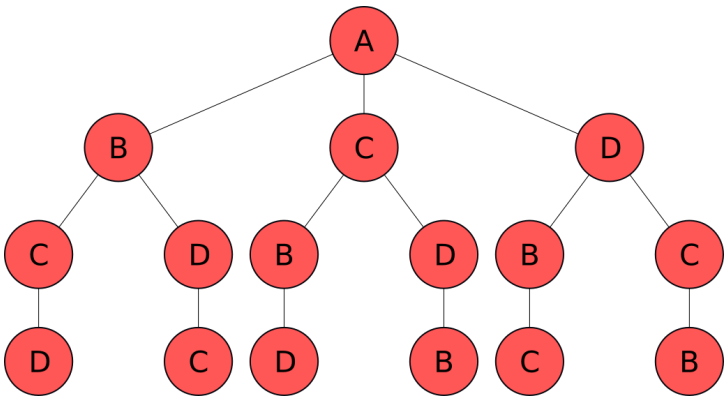

Figura 1: Ejemplo de grafo de todos los caminos posibles con 4 ciudades sin regreso

Por tanto el bucle *while* junto con el primer *for* se ejecuta:

$$\sum_{i=0}^{n-2} \frac{(n-1)!}{i!}$$

veces, dado que no hay que tener en cuenta ni la raíz ni el regreso a la primera ciudad.

Así que la complejidad total de este algoritmo es:

$$T(n) = n * \sum_{i=0}^{n-2} \frac{(n-1)!}{i!}$$

donde el primer n corresponde al segundo *for* y el sumatorio al *while* junto al primer *for* explicado previamente. Hay que multiplicar por n dado que cada vez que se crea un nodo, este segundo *for* tiene que inicializar los valores de ciudades ya visitadas y el orden por el que se visitan. Por tanto, la complejidad de este algoritmo en notación de *Big-O* es:

$$O(n) = n!$$

2.1.2. Estudio Empírico

Para llevar a cabo el estudio empírico, se han utilizado varios ejemplos de grafos generados a partir de un *script* generador de grafos, tanto simétricos como asimétricos. Este *script* está hecho en *Python*, toma como *inputs* el número de ciudades y si el grafo debe ser o no simétrico, y devuelve un fichero con la matriz de costos para representar el grafo y el tamaño de dicha matriz (número de ciudades).

Se han creado diversos archivos, desde 3 hasta 19 ciudades. Para un grafo de n ciudades, se han creado 10 grafos simétricos y 10 grafos asimétricos. Para cada ejemplo, se ha medido el tiempo de ejecución y se ha representado en la figura 2.

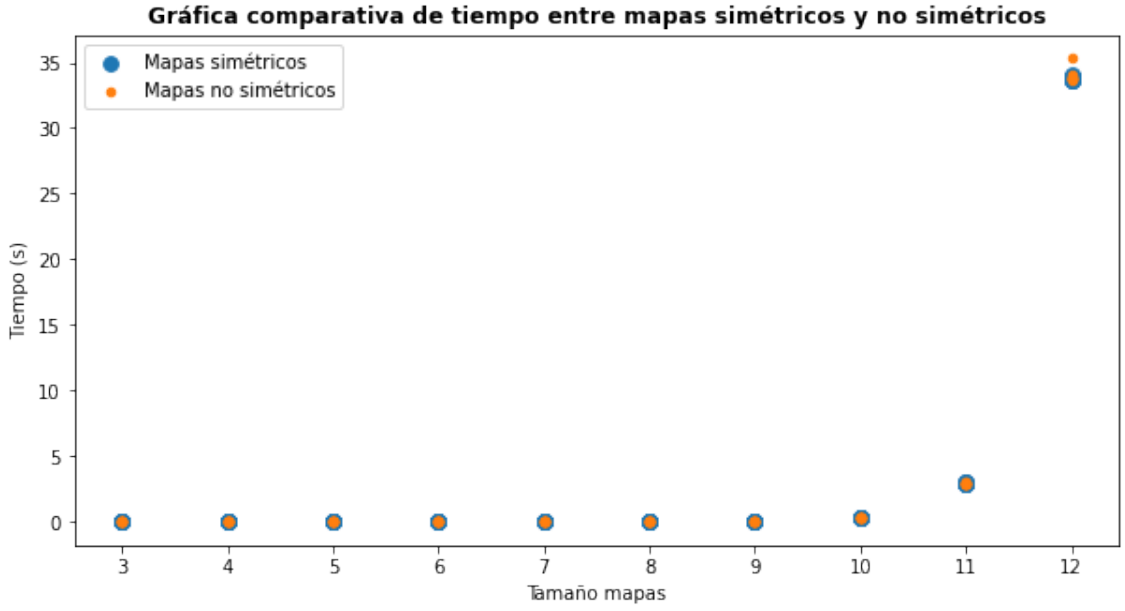

Figura 2: Comparativa de tiempo DFS básico

Los tiempos son prácticamente similares para mapas simétricos como no simétricos, aunque se puede apreciar un crecimiento muy rápido a partir de las 10 ciudades, como era esperable. El problema es que también se traduce en un crecimiento factorial del uso de memoria.

Para más de 12 ciudades el algoritmo no se puede ejecutar porque consume demasiada memoria y no es capaz de terminar la ejecución, por ello no aparecen sus valores en la gráfica.

2.1.3. Estudio Combinado

En este apartado se intentan aplicar la función de complejidad obtenida analíticamente junto a los datos empíricos. Se quiere demostrar si la complejidad que se obtiene al ejecutar el algoritmo coincide con el estudio analítico previo.

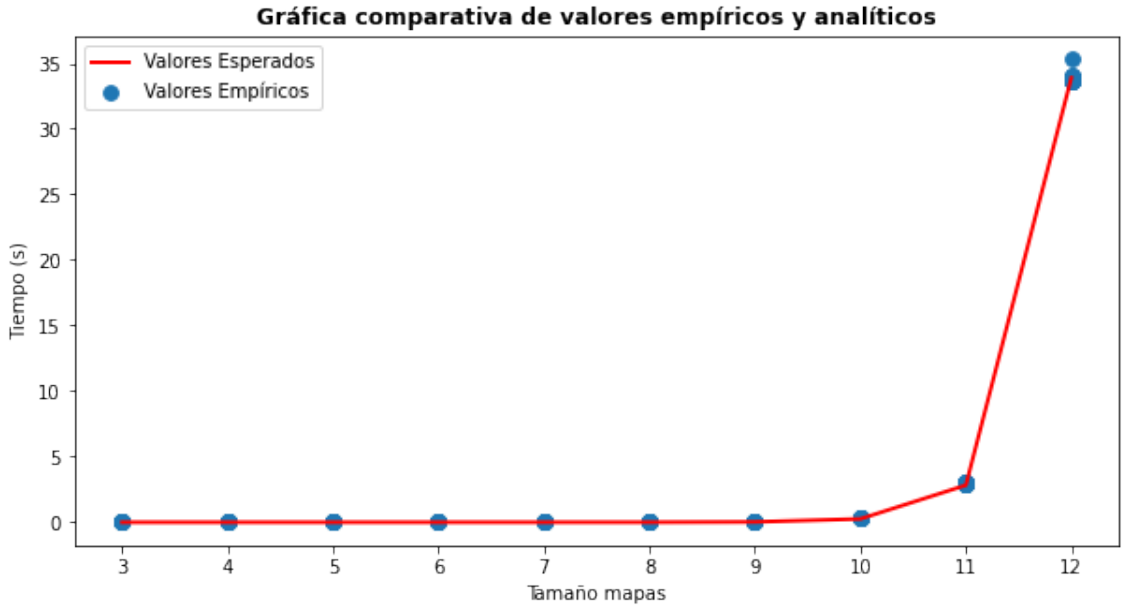

Figura 3: Comparativa de valores empíricos y esperados del DFS básico

En la figura 3 se puede ver que la función se aplica con una precisión muy alta sobre los datos empíricos, confirmando que es posible que esta sea su complejidad. Sin embargo, al tener una cantidad tan reducida de ejemplos, y no poder probar mapas con mayor número de ciudades, no se puede asegurar que la complejidad sea la misma con ejemplos que hacen uso de una n mayor.

2.2. Coste Computacional DFS con poda

2.2.1. Estudio Analítico

Este acercamiento aplica el mismo algoritmo de búsqueda en profundidad iterativa, pero se le realiza una poda al árbol de búsqueda. Con esto se pretende reducir el número de nodos a expandir y por tanto que disminuya el tiempo y consumo de memoria requeridos. Asimismo se espera poder ejecutar mapas de mayor tamaño.

La poda consiste en no expandir nodos cuando el coste del camino acumulado supera el coste del mejor camino encontrado hasta el momento. Es una poda muy sencilla que permite reducir en gran medida el tiempo de computación. Sin embargo, al usar esta mejora aparece una diferenciación entre mejores y peores casos. El mejor caso se da cuando el primer camino es el óptimo y el resto de hijos de la raíz tienen un coste mayor que el camino entero. Por otro lado, el peor caso se da cuando cada camino es mejor que el anterior, de forma que es necesario expandir todos los nodos. Éste caso es idéntico, en complejidad, al algoritmo DFS sin poda.

Respecto a la implementación de este algoritmo, su código es prácticamente igual al anterior a excepción de que a la hora de crear nuevos hijos exite un *if* que hace que solo se creen los hijos si el coste acumulado es menor que el mejor camino encontrado.

El estudio de la complejidad se divide en el estudio del mejor y peor caso por separado porque la complejidad de la mayoría de ejemplos se podrá ajustar usando la media de estos dos casos límite.

En el mejor de los casos será necesario expandir todo el primer nivel del árbol ($n-1$ nodos). Después hay que expandir todos los nodos de uno de ellos ($n-2$ nodos), y así sucesivamente se repite el proceso hasta que solo haya un único nodo. Entonces, la complejidad de este caso es:

$$T_{mejor}(n) = \sum_{i=1}^{n-1} n - i$$

En el peor de los casos la complejidad será la misma que en el problema anterior, pues se tienen que expandir todos los nodos.

$$T_{peor}(n) = n * \sum_{i=0}^{n-2} \frac{(n-1)!}{i!}$$

Por lo tanto, la complejidad media de este algoritmo estará entre la complejidad del peor caso y del mejor. Así que la complejidad del algoritmo DFS con poda en notación *Big-O* sigue siendo:

$$O(n) = n!$$

2.2.2. Estudio Empírico

El estudio empírico se realiza con los mismos mapas simétricos y asimétricos que se crearon para el estudio del algoritmo sin poda. Los resultados obtenidos se muestran en la figura 4, donde esta vez sí aparecen en la gráfica valores para mapas de mayor tamaño, llegando hasta mapas con 19 ciudades.

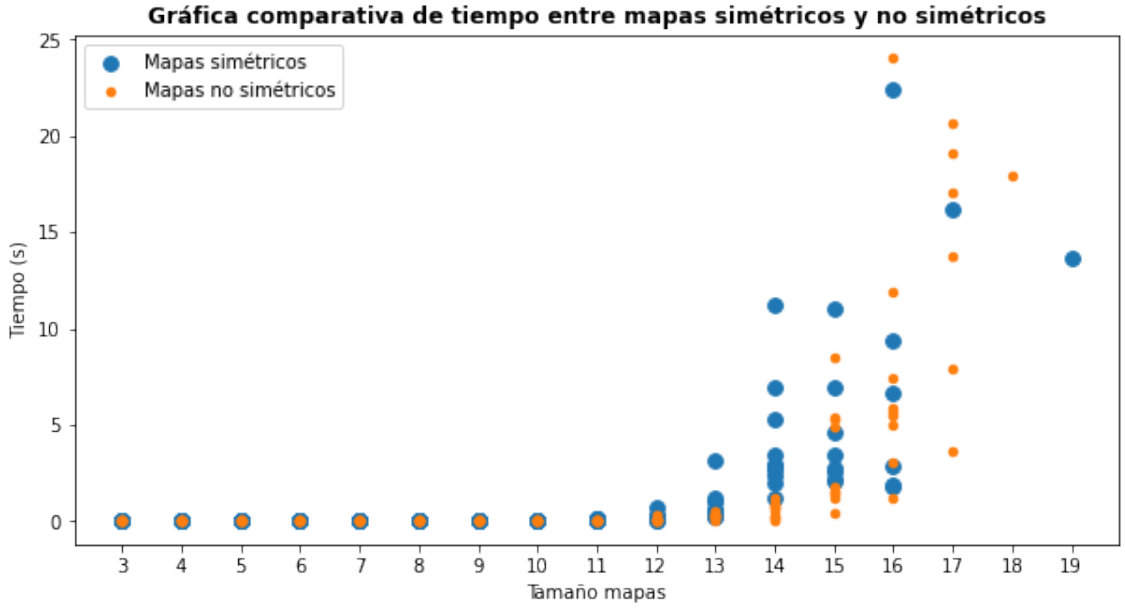

Figura 4: Comparativa de tiempo DFS con poda

Estudiando la gráfica de los resultados obtenidos al resolver el problema del viajante utilizando poda en el algoritmo, se observa que ahora, tanto los mapas simétricos como los mapas asimétricos, presentan diferencias en los tiempos de ejecución. Esto es debido a que en cada mapa se está aplicando una poda diferente y por tanto tienen un coste computacional distinto.

Además, conforme el tamaño de los mapas va aumentando, la diferencia de tiempo que se tarda en resolver mapas del mismo tamaño también va creciendo, pues cuantas más ciudades haya, más caminos hay, y más posibilidades de aplicar poda en la resolución del problema. También se observa que para tamaños grandes no se están ejecutando todos los mapas, algo que ya ocurría en el anterior algoritmo y que se debe a un gran consumo de memoria.

2.2.3. Estudio Combinado

Para el estudio combinado del algoritmo DFS con poda se quiere comprobar si las hipótesis del estudio analítico coinciden con los resultados obtenidos empíricamente. Se analizan por separado el peor y mejor caso con el fin de comprobar esta teoría. El problema surge debido a la enorme diferencia de tamaños entre el peor y el mejor caso. Mientras que el mejor caso tiene un coste polinómico, el peor tiene un coste factorial, que eclipsa por completo al otro caso a la hora de realizar comparaciones.

Graficar estas dos complejidades como límite inferior y superior es complejo dado estas diferencias. Por lo tanto se ha decidido no mostrar las gráficas dado que no aportaban nada al estudio. Se ha podido comprobar que los datos empíricos no siguen ni la función del mejor caso ni la del peor, sino una complejidad intermedia. Esto se debe a que los mapas graficados han sido generados aleatoriamente, y por tanto no corresponden ni a los peores ni a los mejores casos.

3. Algoritmo de búsqueda local

Incialmente se realiza una búsqueda con un algoritmo *(*greedy) o algoritmo voraz, que selecciona en cada paso el camino de menor coste, con la idea de conseguir la solución óptima. Sin embargo, los algoritmos voraces, a pesar de ser algoritmos completos, ya que garantizan encontrar siempre una solución, no garantizan que la solución encontrada sea óptima.

Una vez se tiene un camino usando la búsqueda *greedy*, se aplica sobre éste una búsqueda local con 2-opt, que tratará de mejorar la solución ofrecida inicialmente. Para ello, el algoritmo se basa en cambiar el orden en el que se visitan algunas ciudades en la ruta ofrecida, recalculando el coste de recorrer todas las ciudades. En el caso de que el coste total de recorrer el grafo no mejore, el algoritmo se detiene, ofreciendo una solución a priori mejor que la solución incial. De nuevo, al igual que el algoritmo voraz, la optimización 2-opt no garantiza encontrar la solución óptima al problema.

En este caso, para la implementación del algoritmo no es necesaria una pila de nodos expandidos porque no hace falta explorar todos los caminos. Ya que se expande siempre en cada nodo el de menor coste.

3.1. Coste Computacional

3.1.1. Estudio Analítico

Dado un número de ciudades n, el algoritmo voraz debe elegir una ciudad nueva únicamente cada vez que se desciende un nivel en el árbol. En el último nivel, todas las ciudades deben haber sido visitadas. Como todos los nodos del grafo se deben recorrer, no existe peor caso. En la figura 5 se puede ver un ejemplo en un grafo de este algoritmo con 4 ciudades.

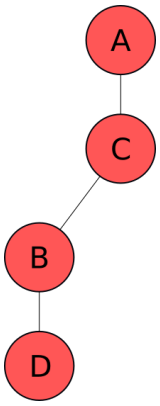

Figura 5: Ejemplo de grafo con un único camino

El algoritmo diseñado consta de un bucle *for* exterior en el que se elige en cada iteración el nodo siguiente a crear, y un bucle *for* interior, que elige el camino de menor coste para viajar a la siguiente ciudad, teniendo en cuenta no visitar ninguna ciudad que ya haya sido visitada. Por tanto, teniendo n ciudades, el bucle exterior se ejecutará $n-1$ veces, ya que la primera ciudad es la ciudad inicial, que se cuenta como visitada, y el bucle interior se ejecuta n veces, puesto que en cada iteración se comprueba si cada ciudad ha sido visitada o no, incluyendo la ciudad inicial. La complejidad del algoritmo voraz queda por tanto como:

$$T_{greedy}(n) = n * (n-1) = n^2 + n$$

La segunda parte de la solución planteada pasa por utilizar el algoritmo 2-opt para mejorar la solución dada por el algoritmo voraz. Este algoritmo elige dos ciudades de la ruta propuesta y cambia el orden de todas la ciudades que estén entre estas dos, con la intención de que, al intercambiar el orden, el coste total de recorrer todas las ciudades disminuya.

Teniendo en cuenta que cada una de las ciudades puede ocupar todas las posiciones en la ruta (excepto la ciudad de partida, que debe aparecer en la primera y última posición siempre), teniendo un total de n ciudades, dado que la primera y última son fijas, tienen que poder combinarse $n-1$ ciudades.

La codificación del algoritmo consiste en ir realizando intercambios en el orden del camino base usando parejas de valores (i y k), donde se realiza un bucle *for* que itera por i dándole un valor inicial de 1, y dentro hay un segundo bucle *for* que itera por k donde se inicializa siempre como $i+1$. Si se recorren ambos bucles realizando permutaciones sin haber encontrado una solución que mejore el estado actual, el algoritmo termina. En caso contrario, vuelve a realizar ambos bucles.

El mejor caso se da cuando el algoritmo *greedy* encuentra la solución óptima o una solución que no se puede mejorar en un ciclo de 2-opt. En este caso la complejidad total del algoritmo sería:

$$T_{mejor}(n) = n^2 + n$$

El peor caso se da cuando el algoritmo tiene que cambiar el camino en todos los ciclos hasta que termina. Según los autores de *The Traveling Salesman Problem: A Case Study in Local Optimization* [1] (página 19), en 1980 Van Leeuwen y Schoone demostraron que como mucho el número máximo de cruces será n^3. Así que en este caso la complejidad es:

$$T_{peor}(n) = (n^2 + n) + n^3$$

Se puede expresar en notación *Big-O* como:

$$O(n) = n^3$$

3.1.2. Estudio Empírico

Al realizar el estudio empírico (de nuevo, con los mismos archivos que se han usado para el resto de los estudios empíricos realizados), se observa que cuantas más ciudades existen en el grafo, más difícil se vuelve la tarea de saber exactamente cuánto tiempo tardará el algoritmo *greedy* con la mejora 2-opt en devolver una solución final.

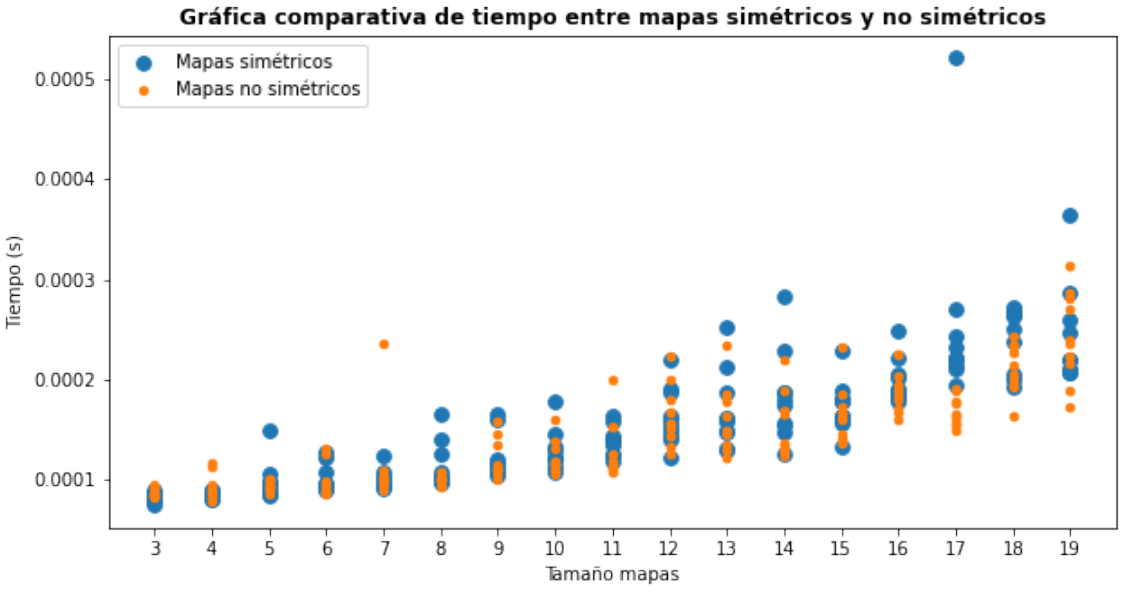

Figura 6: Comparativa de tiempo búsqueda local

El algoritmo ha calculado soluciones para grafos desde 3 hasta 19 ciudades, como se puede ver en la figura 6. Esto se debe principalmente a que, en función de cuál sea la solución proporcionada por el algoritmo greedy, el tiempo necesario para aplicar la optimización 2-opt puede variar. La parte de mejora de la solución con 2-opt es además el motivo por el cual, cuando el número de ciudades es bajo, se tarda menos en encontrar una solución, pues el número de combinaciones posibles para colocar en orden las ciudades es más pequeño que en grafos con más ciudades.

3.1.3. Estudio Combinado

En el estudio combinado del algoritmo de búsqueda local se quiere comparar el estudio analítico con el estudio empírico, con el fin de demostrar que, efectivamente, la complejidad descrita en el estudio analítico se corresponde con la obtenida en el estudio empírico.

Sin embargo, de nuevo sucede lo mismo que en el estudio combinado del algoritmo de búsqueda iterativa con poda. La función de complejidad obtenida no se corresponde con los resultados empíricos. De nuevo, se ha comprobado que los resultados obtenidos siguen una complejidad intermedia entre la complejidad del mejor caso y la complejidad del peor caso, ambos descritos durante el estudio analítico.

Remarcar que las soluciones generadas por este algoritmo no son siempre las óptimas y que al compararse con los resultados obtenidos con el DFS con poda, la diferencia del coste de recorrer un grafo es de un 14 %. Que de acuerdo con el *paper* [1] (página 4) mencionado anteriormente, se encuentra dentro del rango de la mayoría de las heurísticas. Aunque estos resultados pueden depender de un gran número de factores como puede ser la distancia entre las ciudades o el número de ciudades del mapa.

4. Conclusión

Tras la finalización de la práctica, se ha valorado muy positivamente el hecho de haber elegido finalmente *C++* como lenguaje de programación para implementar los algoritmos, pues al ser el problema del viajante un problema de tipo NP-completo dentro de la rama de optimización combinatoria y tener un espacio de soluciones tan amplio, *C++* es una de las mejores opciones para tratar con este tipo de problemas.

Se concluye que se han utilizado dos formas para resolver el problema del viajante. Por un lado, está la solución propuesta utilizando el algoritmo DFS, un algoritmo de búsqueda iterativa, donde se comprueban todas las combinaciones posibles, con el fin de asegurar encontrar la solución óptima para cualquier dominio del problema. Aplicando la poda, el coste computacional del algoritmo se reduce, pues en la mayoría de casos, el número de estados que se deben comprobar es menor.

Por otro lado, se encuentra la solución propuesta por el algoritmo voraz aplicando el algoritmo de optimización 2-opt a la solución obtenida del algoritmo voraz. Los dos algoritmos son algoritmos de búsqueda local. En este caso, no se garantiza que la solución encontrada sea la solución óptima, a cambio de intentar reducir el coste computacional del problema.

En resumen, ambas opciones son válidas, pues aunque las soluciones que ofrecen los algoritmos de búsqueda local no son óptimas, son lo suficientemente buenas y cercanas a la solución óptima como para poder discriminar esa diferencia y aceptarlas.

Referencias

[1] D. Johnson and L. A. McGeoch. The traveling salesman problem: A case study in local optimization. 2008.