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.

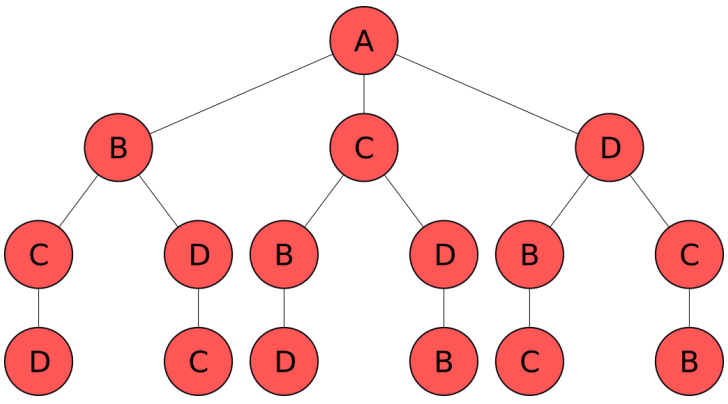

Figura 1: Ejemplo de grafo de todos los caminos posibles con 4 ciudades sin regreso

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.

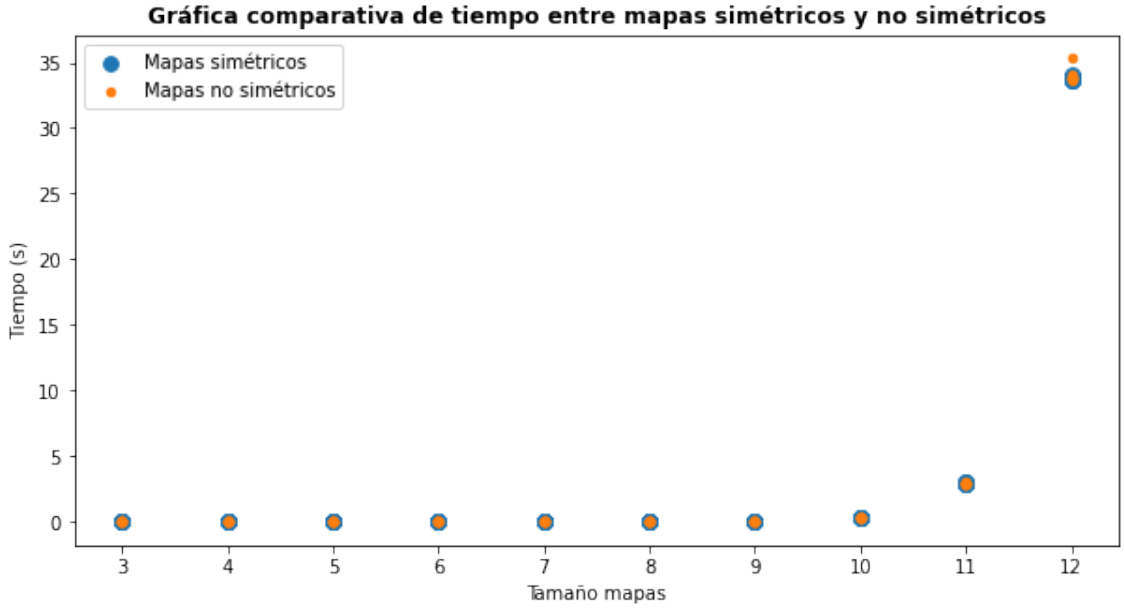

Figura 2: Comparativa de tiempo DFS básico

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.

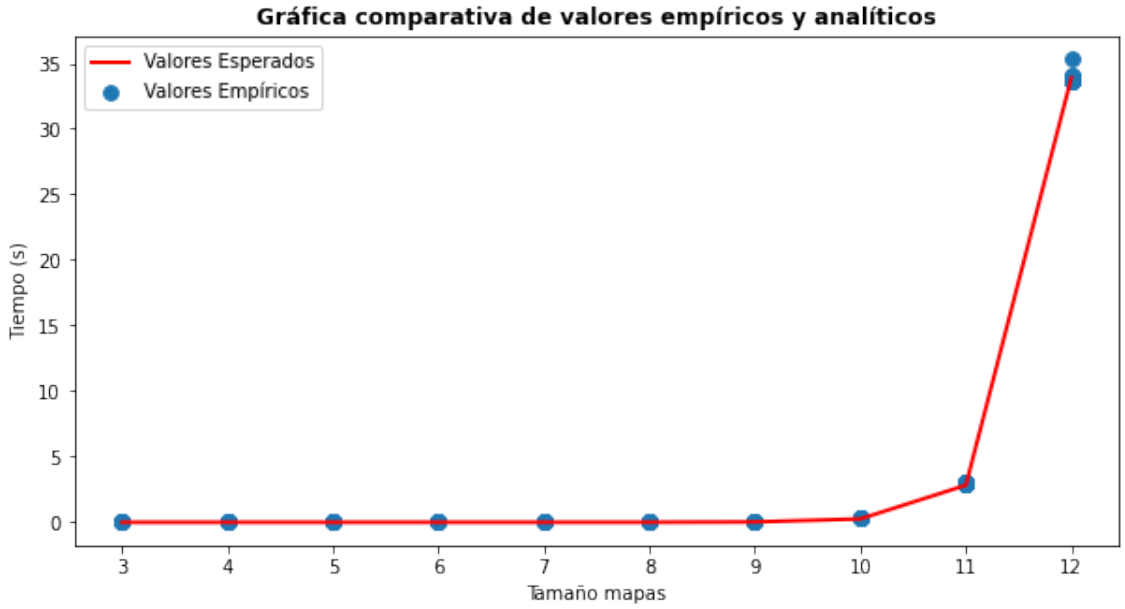

Figura 3: Comparativa de valores empíricos y esperados del DFS básico

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

2.2.2. Estudio Empírico

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 ambos tipos de mapas se está aplicando poda, por lo que es de esperar que los tiempos sean distintos.

Además conforme el tamaño de los mapas va aumentando, la diferencia de tiempo que se tarda en resolver mapas del mismo tamaño también va aumentando, pues cuantas más ciudades haya, más caminos hay, y por tanto más veces se puede aplicar la poda en la resolución del problema.

2.2.3. Estudio Combinado

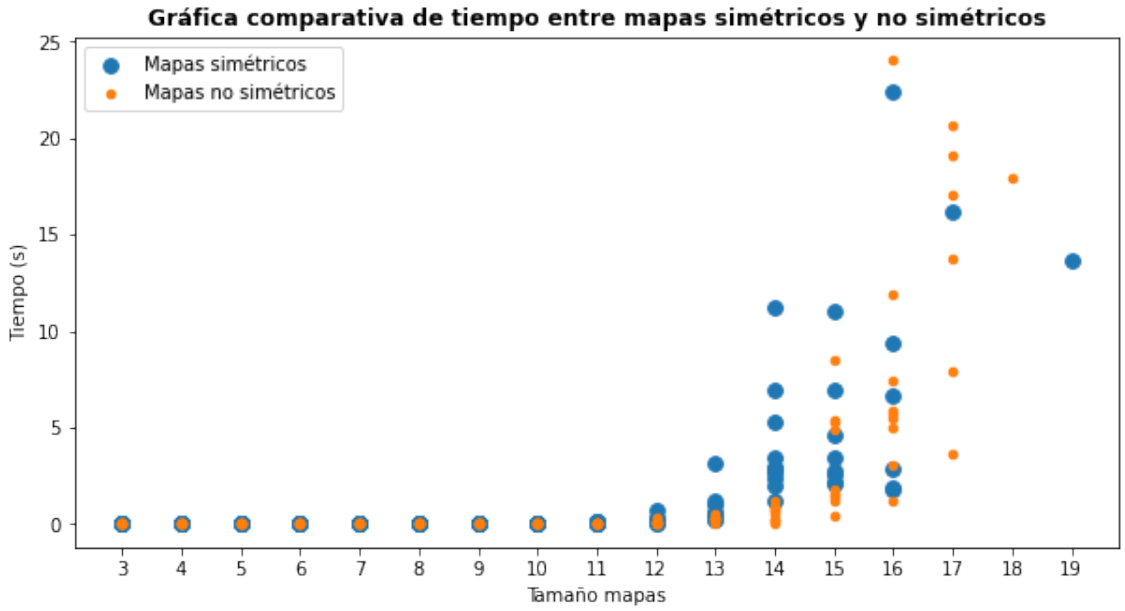

Figura 4: Comparativa de tiempo DFS con poda

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.

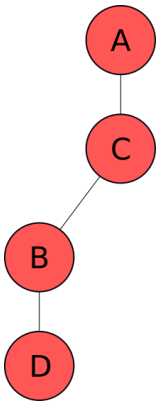

Figura 5: Ejemplo de grafo con un único camino

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 peor caso se da cuando el algoritmo tiene que comprobar todas las posibles combinaciones. Por lo tanto tiene que realizar todas las permutaciones posibles sobre $n-1$ elementos, con una complejidad de:

$$T_{2-opt}(n) = (n-1)!$$

Por tanto, la complejidad total del algoritmo de búsqueda local es:

$$T(n) = (n^2+n) * (n-1)!$$

Y se puede expresar en notación *Big-O* como:

$$O(n) = n!$$

3.1.2. Estudio Empírico

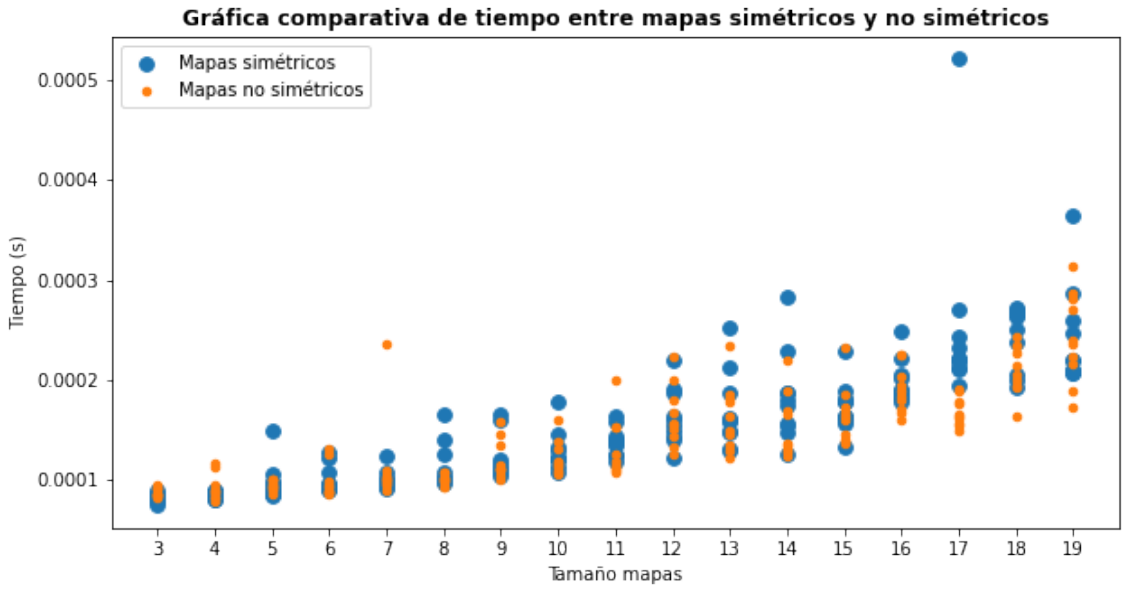

Figura 6: Comparativa de tiempo búsqueda local

3.1.3. Estudio Combinado

4. Conclusión