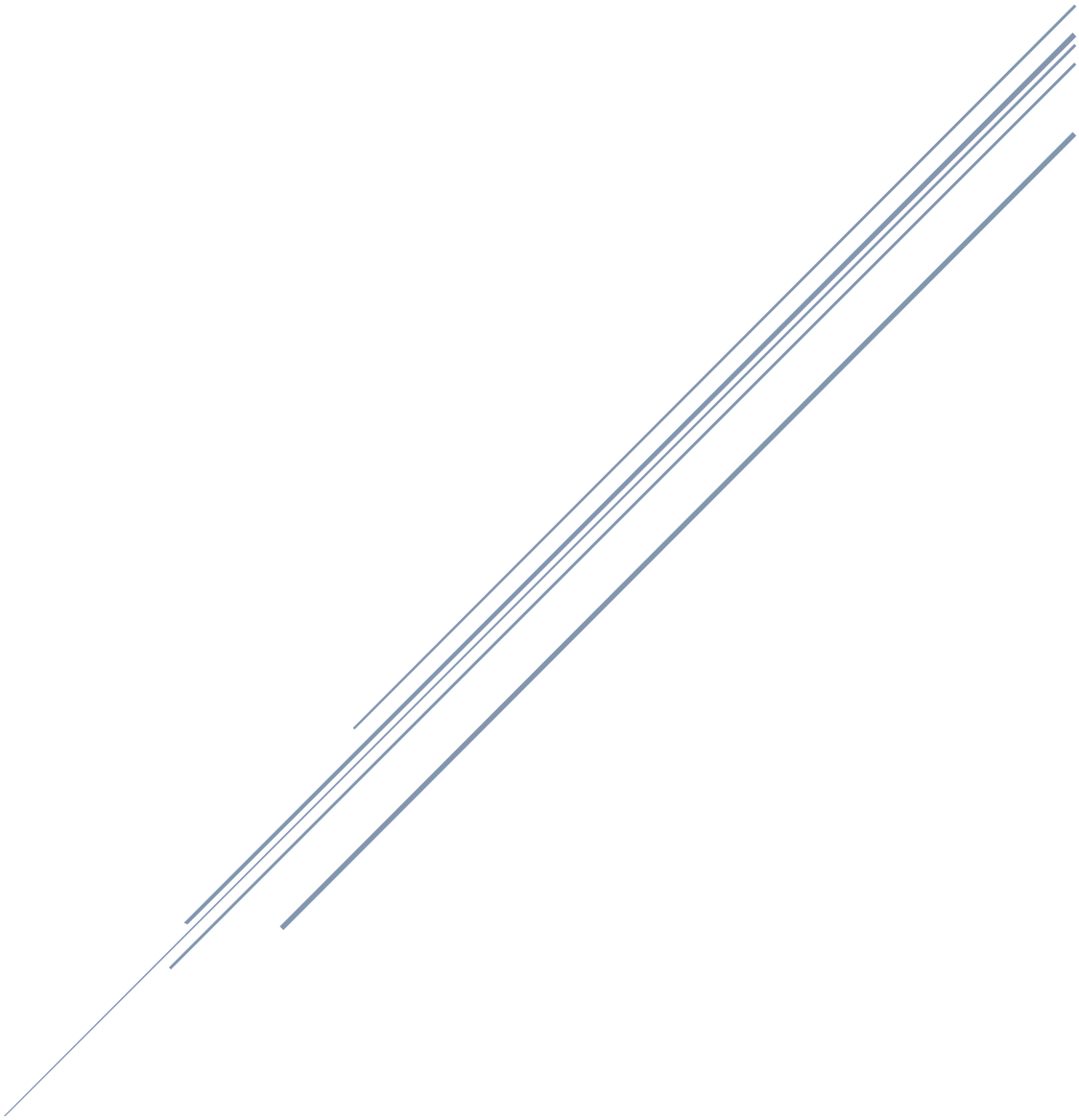
TEORÍA AVANZADA DE LA COMPUTACIÓN

PRÁCTICA 3, 4º curso

Universidad Carlos III Madrid, Ingeniería Informática, 2022



grupo 80 - Colmenarejo - IVAN AGUADO PERULERO – 100405871
grupo 80 - Colmenarejo – JAVIER CRUZ DEL VALLE – 100383156
grupo 80 - Colmenarejo - JORGE SERRANO PÉREZ – 100405987

Contenido

1.- INTRODUCCIÓN 2
2.- DESCRIPCIÓN DEL CÓDIGO 3
3.- BACKTRACKING 4
3.1.- Estudio empírico 4
3.2.- Estudio analítico 5
4.- HEURÍSTICA Y PODA 8
4.1.- Estudio empírico 8
4.2.- Estudio analítico 9
5.- ALGORITMO GREEDY 11
5.1.- Estudio empírico 11
5.2.- Estudio analítico 12
6.- OPERADOR 2-OPT 14
6.1.- Estudio empírico 14
6.2.- Estudio analítico 15
7.- CONCLUSIONES 18
8.- BIBLIOGRAFÍA 19
9.- ANEXO 20

1.- INTRODUCCIÓN

El objetivo de esta práctica es estudiar el Problema del Viajante o *Travelling Salesman Problem* (TSP) mediante distintos algoritmos realizando un análisis de coste y complejidad computacionales, tanto de manera empírica como analítica. Estos algoritmos serán el de búsqueda basada en *Backtracking* (o equivalente iterativa), al que más adelante se añadirá una heurística sencilla o una poda, y el de búsqueda local, que utilizará un algoritmo *Greedy* para más tarde aplicar el operador *2-opt*.

El Problema del Viajante se trata de uno de los 23 problemas originales pertenecientes a la clase *NP-Complete* y consiste, en la versión euclídea, en encontrar el camino más corto (ciclo hamiltoniano) que recorra las n ciudades (nodos) del problema conectados por aristas con un peso correspondiente a la distancia entre las mismas y que tenga un coste total mínimo.

Para realizar el análisis, deberemos implementar primero los algoritmos mencionados y generar nuestros casos de prueba de forma aleatoria, utilizando un lenguaje de programación como *Python* o *C*.

En cuanto al presente documento, en primer lugar encontramos una introducción a los algoritmos y a la práctica. A continuación, distintas secciones que contarán con su correspondiente estudio analítico y empírico. Estas se corresponden con los algoritmos mencionados anteriormente. Finalmente, se detallarán una serie de conclusiones extraídas de la realización del análisis.

2.- DESCRIPCIÓN DEL CÓDIGO

En esta sección se detallarán partes del código que consideramos relevantes que no pertenecen a un algoritmo concreto, sino que son de aspectos más generales.

Para disponer de conjuntos de distribuciones de ciudades y generar nuestros propios **casos de prueba,** hemos creado un método que recibe el número de ciudades n como parámetro y genera ese número de ciudades utilizando la función *random* en Python para asignar las coordenadas x e y.

Por otro lado, para **dibujar el problema y verificar las soluciones**, hemos diseñado un método para imprimir el gráfico con las ciudades distribuidas en el plano y el recorrido generado que las conecta. Para ello, hemos hecho uso de la librería *matplotlib*, que nos permite dibujar las ciudades en sus respectivas coordenadas y, en línea roja, el ciclo hamiltoniano que las une. Además, se especifica el camino óptimo como subtítulo del gráfico. Aquí se puede ver un ejemplo:

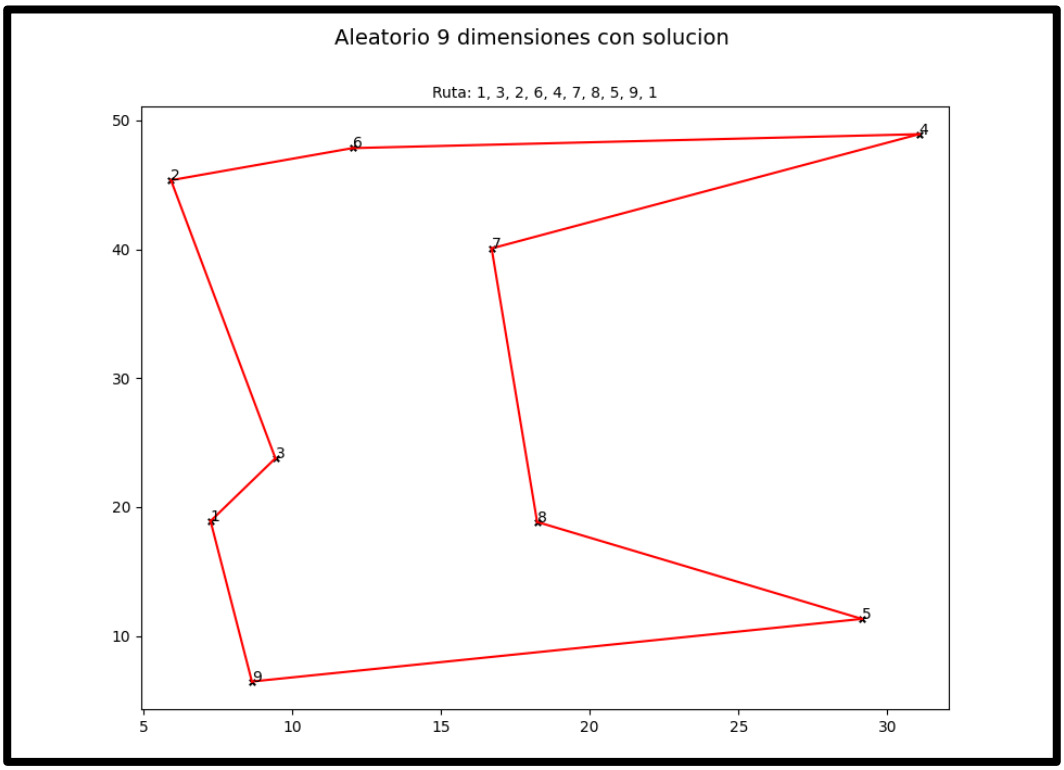


Ilustración 1: Ejemplo solución para 9 dimensiones

Por último, otra función importante es la de calcular la distancia entre dos ciudades, en la que se utiliza la fórmula de la distancia euclidiana:

$$dE(P1, P2) = \sqrt{(x_1 + x_2)^2 + (y_1 + y_2)^2}$$

siendo $P1 = (x_1,\ y_1)$ y $P2 = (x_2,\ y_2)$ con sus coordenadas cartesianas respectivamente.

3.- BACKTRACKING

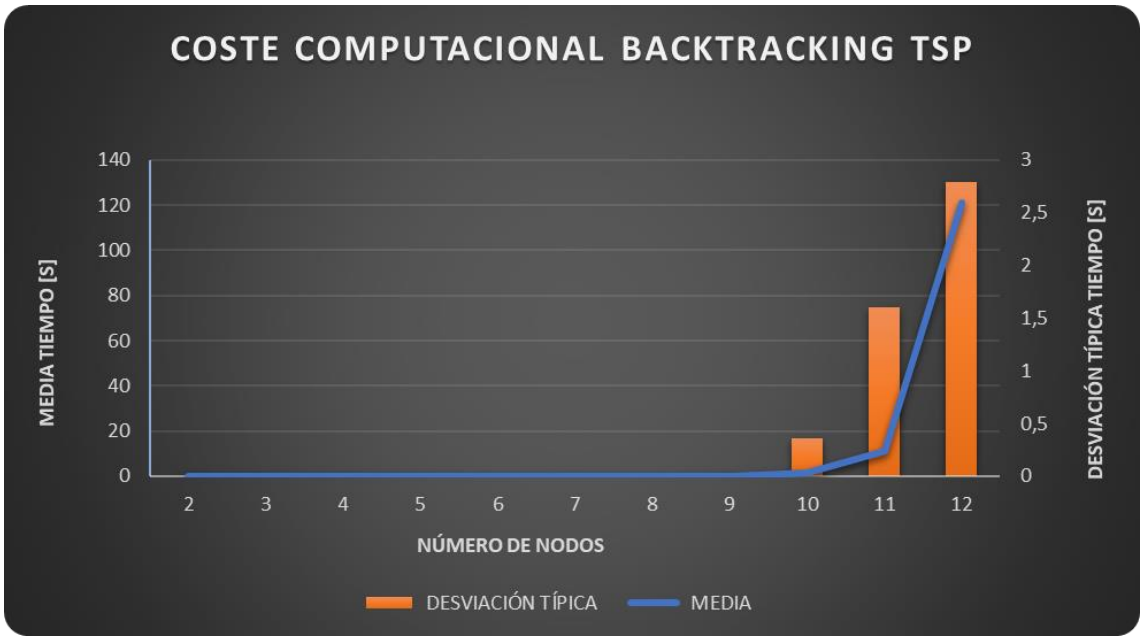
En esta tercera sección, el objetivo es estudiar el algoritmo de búsqueda con *Backtracking* tanto empírica como analíticamente. Se calculará también la complejidad computacional y las diferencias finitas.

En este caso, hemos repetido el experimento **10 veces** para cada número de nodos para evaluar cómo afecta la creación del mapa aleatorio en el tiempo de computación.

El algoritmo implementado se trata de, desde una ciudad considerada como punto de partida y de final, moverse a los nodos adyacentes mediante **búsqueda en profundidad**, calcular el respectivo coste e ir almacenándolo. Una vez calculado el de todas las distintas posibilidades, se devuelve el menor de todos. Utiliza distintos *arrays* para saber qué nodos están visitados y cuáles no, así como las distancias y el recorrido o solución final de cada uno.

3.1.- Estudio empírico

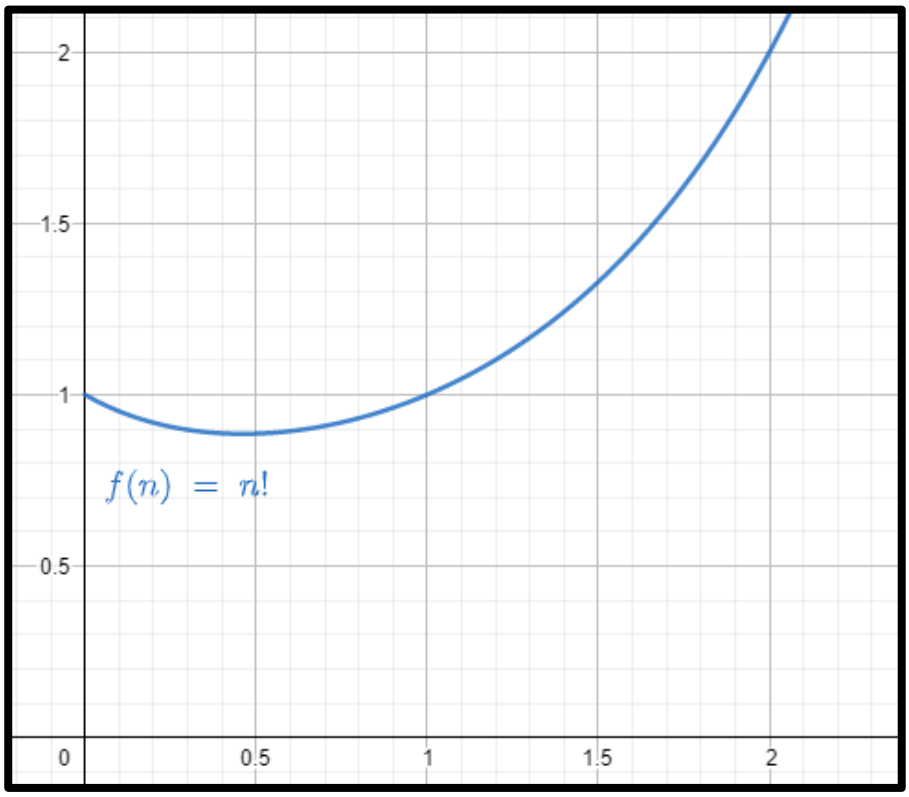
En primer lugar, realizaremos una serie de pruebas empíricas determinando el tiempo medio y la desviación típica para un total de once experimentos. Estos van desde dos nodos o ciudades (ya que es el mínimo posible para establecer un ciclo hamiltoniano) hasta 12. No hemos podido aumentar más este valor porque los medios que tenemos disponibles no tienen suficiente potencia de cálculo como para determinar tiempos tan altos. Sin embargo, esto ya nos ha dado una ligera idea de cómo podría resultar la complejidad resultante.



Gráfica 1: Coste computacional Backtracking TSP

Como se puede apreciar en la gráfica, es al final cuando el tiempo obtenido empieza a aumentar. Concretamente, a partir del décimo valor se aprecia el inicio de la curva, y en él se ha obtenido una media de 1,2 segundos, **incrementándose exponencialmente** hasta alcanzar un valor aproximado de 121,31 con un total de 12 nodos.

Por lo que se puede ver, extraemos como conclusión que quizá se trate de una **complejidad factorial** debido a su similitud con la gráfica de la misma.



Gráfica 2: Función factorial

3.2.- Estudio analítico

Tras los resultados obtenidos, como hemos mencionado, es de sospechar la complejidad factorial del algoritmo. Esto tendría sentido, pues para el primer nodo hay n posibles conexiones, para el segundo $n-1$, para el tercero $n-2$ y así sucesivamente, siendo "n" el número de nodos (ciudades del TSP).

Si nos fijamos en el código desarrollado, observamos que se trata de un **algoritmo recursivo**, ya que se llama a sí mismo aumentando el número de nodos visitados:

```
def tsp_backtracking(self, graph, v, currPos, n, count, cost, answer, path, all_paths):
    if count == n and graph[currPos][0]:
        answer.append(cost + graph[currPos][0])
        all_paths.append(path)
        return
    for i in range(self.dimension):
        if v[i] is False and graph[currPos][i]:
            v[i] = True
            self.tsp_backtracking(graph, v, i, n, count + 1, cost + graph[currPos][i],answer, path + "->" + str(i + 1), all_paths)
            v[i] = False
```

Ilustración 2: Primer parte del código referente a Backtracking

Dicho esto, para los nodos obtendremos una complejidad temporal de:

$$n * (n-1) * (n-2) \ldots = O(n!)$$

Para respaldar y confirmar esta complejidad se realizará un estudio con las diferencias finitas y cota asintótica, como se ha hecho en las prácticas anteriores.

Diferencias finitas y Cálculo de $T(n)$:

N	5	6	7	8	9	10	11	12
TIEMPO	**0**	**0,0016**	**0,0051**	**0,0228**	**0,1352**	**1,2009**	**10,9800**	**121,3082**
DIF 1		0,0016	0,0035	0,0177	0,1124	1,0657	9,7791	110,3282
DIF 2			0,0019	0,0142	0,0947	0,9533	8,7134	100,5491
DIF 3				0,0123	0,0805	0,8586	7,7601	91,8357
DIF 4					0,0682	0,7781	6,9015	84,0756
DIF 5						0,7099	6,1234	77,1741
DIF 6							5,4135	71,0507
DIF 7								65,6372

Tabla 1: Cálculo de diferencias finitas

Como se puede observar, para este caso las diferencias finitas no consiguen anularse. Tras esto, y con los resultados obtenidos en el análisis empírico y al principio de este apartado, en el que se intuye una **complejidad factorial**, analizamos el código de la función desarrollada para conseguir un $T(n)$.

```
def tsp_backtracking(self, graph, v, currPos, n, count, cost, answer, path, all_paths):
    if count == n and graph[currPos][0]: # 3 + 5 = 8
        answer.append(cost + graph[currPos][0]) # 3
        all_paths.append(path)  # 1
        return # 1
    for i in range(self.dimension): # 1 + 1 + n(1 + 1 + 8 + n!) = n*n! + 10n + 2
        if v[i] is False and graph[currPos][i]: # 4 + 4 + n! = n! + 8
            v[i] = True # 2
            self.tsp_backtracking(graph, v, i, n, count + 1, cost + graph[currPos][i], # n!
                                  answer, path + "->" + str(i + 1), all_paths)
            v[i] = False # 2
#T(n) = n * n! + 10n + 10  =====> O (n * n!)
```

Ilustración 3: Segunda parte del código referente a Backtracking

Obtenemos un $T(n) = n * n! + 10n + 10$

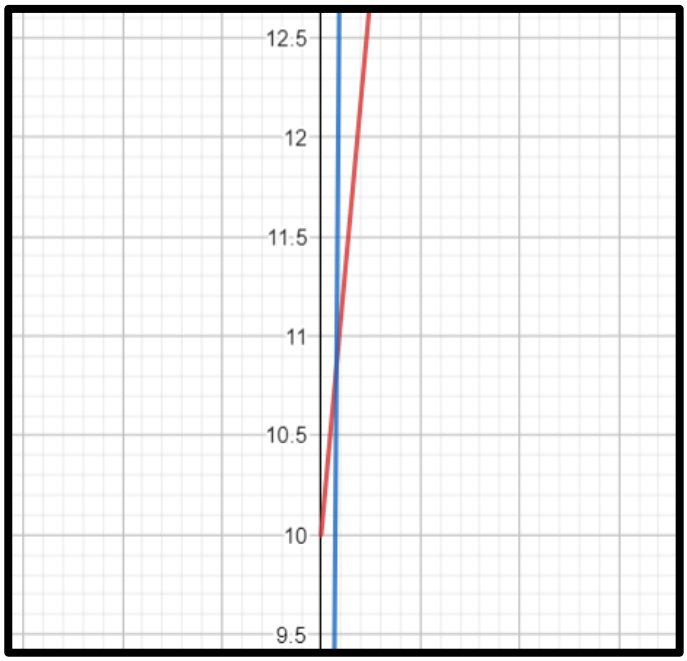
Cota superior asintótica

Para determinar la cota asintótica en este caso se ha utilizado un $n_0 = 4$ para la mejor visualización de la gráfica.

- $g(n) = n * n!$
- $f(n) = T(n) = n * n! + 10n + 10$

- $n = n_0 = 4$
- $c * 4 = 4 * 4! + 40 + 10$
  $c = 146$

Por lo que $c(n) = 146 * n * n!$



Gráfica 3: Cota superior asintótica para el algoritmo de Backtracking

$f(n) = n * n! + 10n + 10$ $\qquad$ $c(n) = 146 * n * n!$

Como se puede ver en la gráfica, la función $c(n)$ es cota superior asintótica de $f(n)$ a partir del punto $x = 0.077, y = 10.85$, cuando n tiende a infinito

Tras el estudio realizado se puede concluir que la complejidad del algoritmo analizado es factorial.

4.- HEURÍSTICA Y PODA

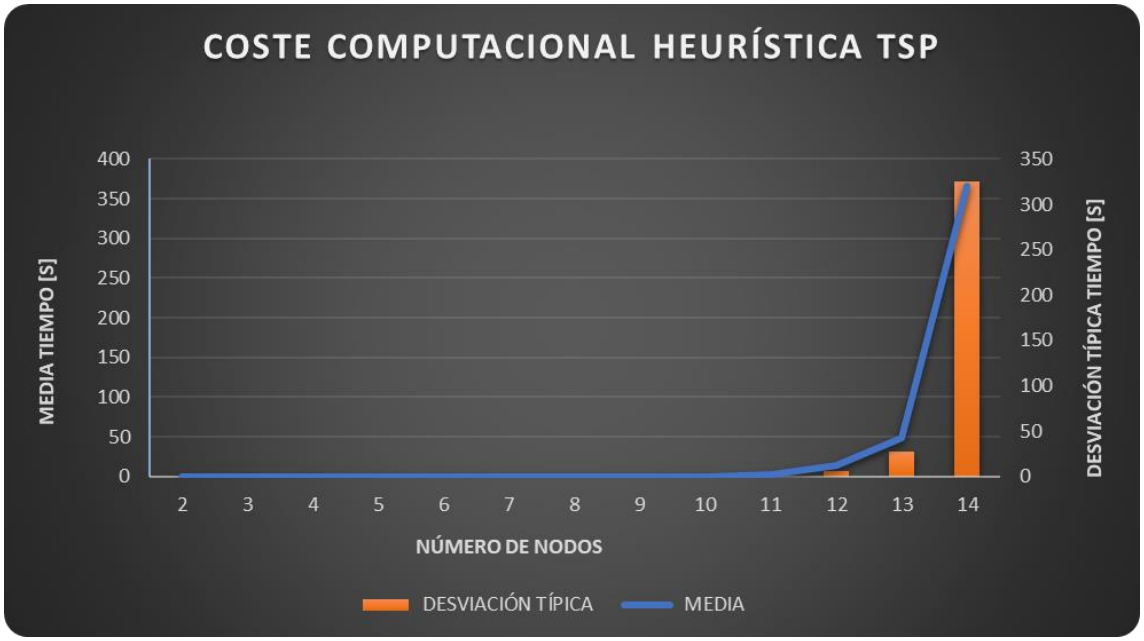
En esta segunda sección, el objetivo es estudiar el algoritmo de búsqueda con *Backtracking* tanto empírica como analíticamente, pero aplicando el algoritmo de ramificación y poda con una heurística simple. Se calculará también la complejidad computacional y las diferencias finitas.

El algoritmo implementado es una variante del *Backtracking* mejorada sustancialmente. Este se interpreta como un árbol de soluciones, donde cada una de las ramas nos conduce a una posible solución posterior a la actual. La mejora que se consigue gracias a que esta técnica permite **detectar ramificaciones en las que las soluciones ya no están siendo óptimas** y de esta manera podar el árbol y no malgastar recursos en casos que se alejan de la solución buscada.

Al algoritmo previamente descrito se le ha añadido una **heurística** simple y admisible para mejorar aún más los tiempos, que se basa en el cálculo del **número de ciudades aún no visitadas**. Este valor se tiene en cuenta a la hora de dar solución al problema junto con la poda y consigue optimizar más los cálculos.

4.1.- Estudio empírico

Para comenzar, realizaremos un estudio empírico como en el apartado anterior. En este caso, se han podido **aumentar el número de experimentos (13)** aumentando el número de ciudades, ya que al utilizar heurísticas y podas, en cierta medida se reducen los tiempos y el ordenador no se congela a la hora de obtener la duración de cada uno. Aun así no hemos ejecutado pruebas para muchas ciudades por el tiempo que esto conllevaba, más aún teniendo que realizar varios experimentos para obtener la media. Por ejemplo, para uno de 15 nodos el tiempo sobrepasaba la hora y media.



Gráfica 4: Coste computacional heurística TSP

A pesar de que los tiempos son más cortos, **parece que la complejidad sigue manteniéndose factorial**. Observando la gráfica, los tiempos aumentan exponencialmente a partir de los 11 nodos. Obtenemos una media de $0{,}4233$ segundos con 10 ciudades, en 11 asciende a $2{,}628$, y llega hasta $365{,}2566$ con 14 nodos. Sin embargo, podemos ver que en este caso la desviación típica se mantiene bastante semejante a la media respecto al caso anterior, lo que nos indica que los datos están más agrupados respecto al mismo.

Por lo tanto, en una primera instancia, podríamos concluir que, según el estudio empírico, podría tratarse otra vez de una **complejidad factorial**, aunque podamos realizar experimentos más complejos y con mejores tiempos que los obtenidos con el *Backtracking* a secas.

4.2.- Estudio analítico

A continuación, realizaremos un estudio analítico para determinar la complejidad teórica del algoritmo utilizando heurística y poda y ver si coincide con la obtenida empíricamente.

Como se ha comentado en la parte empírica del análisis, podemos intuir que la complejidad de este algoritmo es factorial. Como en el caso anterior, se trata de un algoritmo recursivo, ya que se llama a sí mismo. Además, al tratarse de una mejora del algoritmo de Backtracking, para el primer nodo hay n posibles ciudades a las que viajar, para el siguiente hay n - 1, y así sucesivamente, por lo que el funcionamiento en general es similar.

Dicho todo esto y con conocimiento de los resultados obtenidos en el estudio analítico previo, consideramos que lo más oportuno para este apartado es realizar un **análisis más teórico y literario**, es decir, uno en el que nos basemos en los fundamentos y conceptos de la ramificación y poda para este problema.

Primeramente, como se menciona en el artículo *"El problema del viajante de comercio: análisis teórico y estrategias de resolución"* [1], la más destacada diferencia entre el algoritmo de Backtracking y el de ramificación y poda es la manera de recorrer el **espacio de soluciones**. Mientras que el primero gestiona el recorrido mediante una **pila** (cuando se llega a un nodo descartado/muerto, se elimina de la pila para deshacer la acción tomada y volver al anterior), el segundo se basa en una **cola con prioridad**. El algoritmo se encargará de "podar" las ramas que no están siendo óptimas para no malgastar recursos y obtener una solución de manera más rápida.

Si analizamos el código implementado, línea por línea, parece tener una complejidad $O(n^2)$ al contar con dos bucles anidados. Sin embargo, como se destaca en *"Travelling Salesman Problem using Brach and Bound approach"* [2], esto es más complejo de analizar. Estamos creando todas las posibles extensiones de nodos de un árbol, es decir, una permutación. Suponiendo n ciudades totales, necesitamos generar todas las permutaciones de las $n-1$ ciudades, excluyendo la raíz. Por lo tanto la complejidad para generar la permutación es:

$$O((n-1)!) \;=\; O(2^{n-1})$$

Por lo tanto la complejidad final del algoritmo sería:

$$O(n^2 * 2^n)$$

Pero, como se ha mencionado anteriormente, este algoritmo se basa en la poda de ramificaciones en las que ya no se están obteniendo soluciones óptimas. Por lo tanto, como bien dice el artículo *"Travelling Salesman Problem using Branch and Bound"* [3], la **complejidad** del peor caso de ramificación y poda es la misma que la de resolverlo por **fuerza bruta**, pues habrá casos en los que no podamos podar ningún nodo (El algoritmo de fuerza bruta se basa en buscar todas las permutaciones posibles ($n!$) y luego elegir la de menor coste). Esto explica la **tendencia factorial** que podemos observar en el gráfico presentado, a pesar de haber conseguido mejores tiempos.

En definitiva, tras todo lo mencionado anteriormente y sabiendo que la complejidad también depende de la función delimitadora elegida, donde se tiene en cuenta la **heurística**, se puede concluir que, al ser esta simple, la complejidad final del algoritmo es:

$$O(n!).$$

5.- ALGORITMO GREEDY

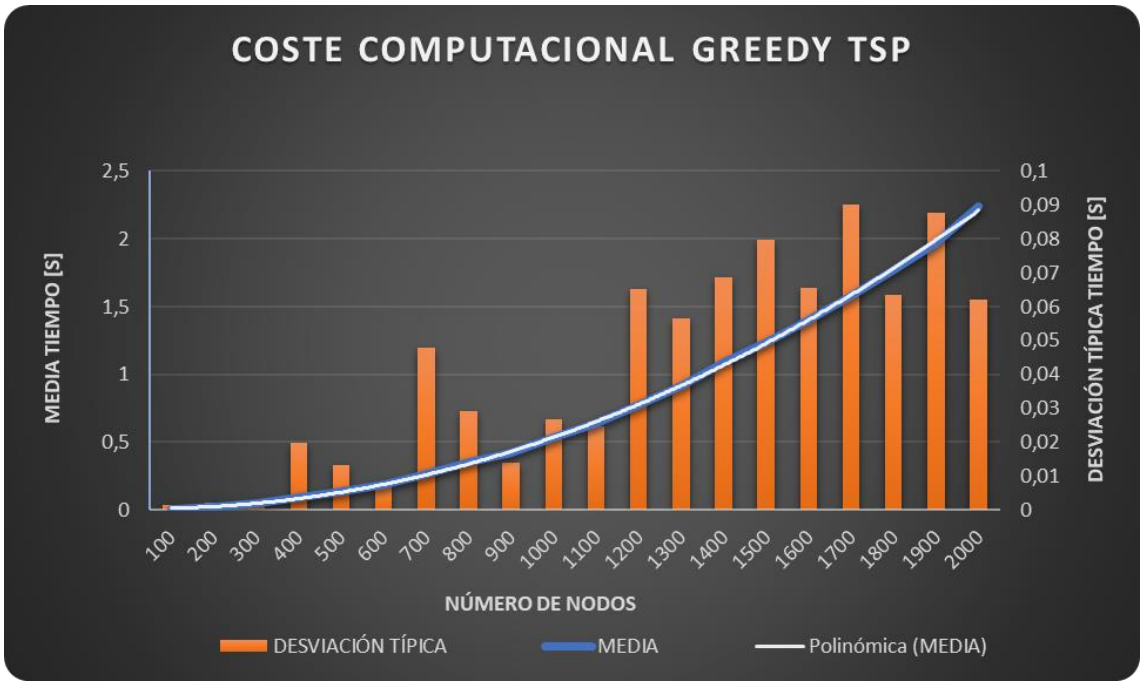
En el siguiente apartado analizaremos el código que implementa el algoritmo *Greedy* para pasar a realizar un estudio empírico y analítico sobre el mismo en donde obtengamos resultados semejantes, permitiéndonos calcular a su vez la complejidad computacional y las diferencias finitas.

Nuevamente, se repetirá el experimento **20 veces** para cada número de nodos evaluando cómo afecta la creación del mapa aleatorio en el tiempo de computación. El algoritmo implementado se basa en iterar entre las distintas ciudades más cercanas que no han sido visitadas previamente, desarrollando una estrategia de búsqueda que consiste en elegir la **opción óptima en cada paso local** con el objetivo de llegar a la solución óptima general.

A diferencia de en casos anteriores, no se utilizan *arrays* para saber qué nodos están visitados y cuáles no, así como las distancias y el recorrido o solución final, si no que guarda el cómputo global y cada vez que encuentra uno nuevo mejor al que figura en ese momento, lo descarta y pasa a colocar este último.

5.1.- Estudio empírico

Respecto al estudio empírico realizaremos un total de 20 pruebas aumentando el número de nodos cada 100 partiendo desde **100 nodos iniciales** hasta llegar a un máximo de **2000 nodos**. Se han escogido estas cifras debido a que son las mejores para observar la tendencia que presenta la complejidad. Esto se debe a que, con dichos valores y sus diferencias entre ellos, ya **somos capaces de elaborar una gráfica donde se aprecie correctamente su comportamiento**.



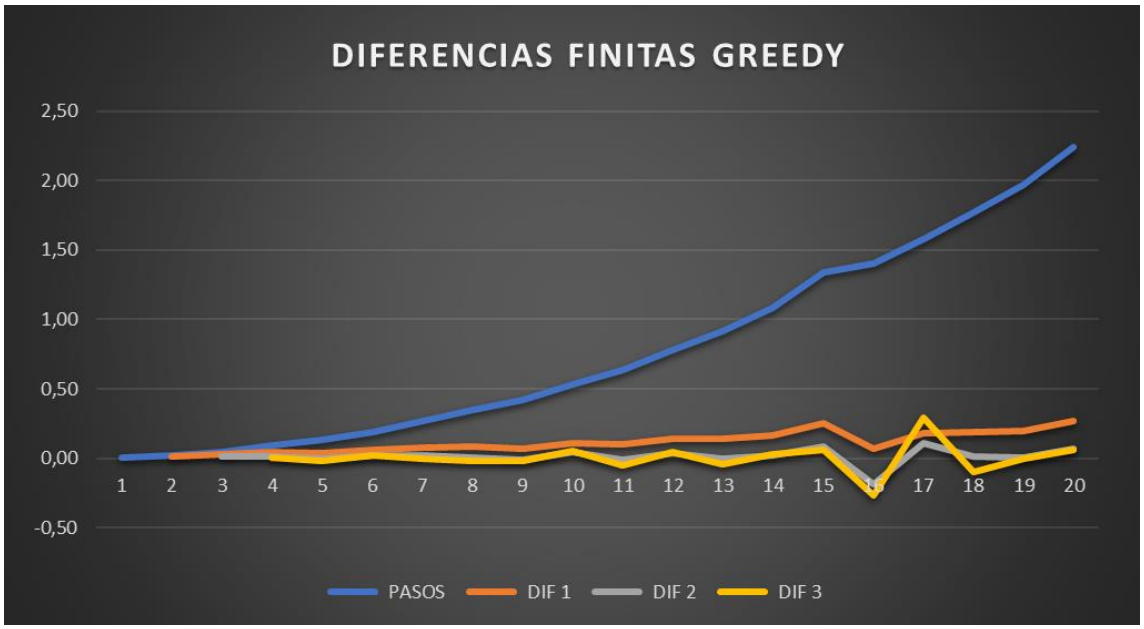
Gráfica 5: Coste computacional Greedy TSP

Observamos una tendencia en los valores medios que se aproxima a una función de complejidad polinómica. Debido a esto, hemos trazado una línea de tendencia respecto a dicha media que se trata de una función polinómica de segundo grado y que encaja casi a la perfección con la media. Obtenemos un resultado empírico que claramente nos indica y guía hacia los valores teóricos que deben darnos en el estudio analítico.

5.2.- Estudio analítico

A continuación, realizaremos en primer lugar el cálculo de las diferencias finitas para contrastar los resultados con los obtenidos anteriormente.

Debido a que se trata de una tabla excesivamente grande, dejaremos en el final del documento los cálculos anexados para que se pueda comprobar que se han realizado correctamente. La gráfica resultante de los mismos es la siguiente.



Gráfica 6: Cálculo de diferencias finitas Greedy

En esta gráfica comprobamos que llegamos a **valores nulos en el tercer paso** del cálculo de las diferencias finitas. A simple vista puede parecer que no es así, y esto se debe a que, como manejamos una precisión muy elevada (del orden de hasta 9 exponentes), nunca obtendremos una línea recta en el valor nulo, si no que todo lo contrario, los decimales generarán en cada paso un mayor **ruido** perdiendo por completo la referencia. Este suceso lo podemos observar en la gráfica anexada al final del documento, que nos muestra cómo a partir de la diferencia finita de nivel tres el resto son réplicas a mayor escala. Para salir de duda, calcularemos el promedio de los pasos y de cada una de las tres diferencias obteniendo los valores 0.79, 0.12, 0.01 y 0. Si utilizamos más decimales observaremos que el 0 realmente es 0.003 y que sus diferencias finitas siguientes van en aumento (a consecuencia del ruido explicado anteriormente), por lo que ahora sí que podemos determinar con rotundidad que se trata del paso final y de una complejidad polinómica de segundo orden.

El cálculo es:

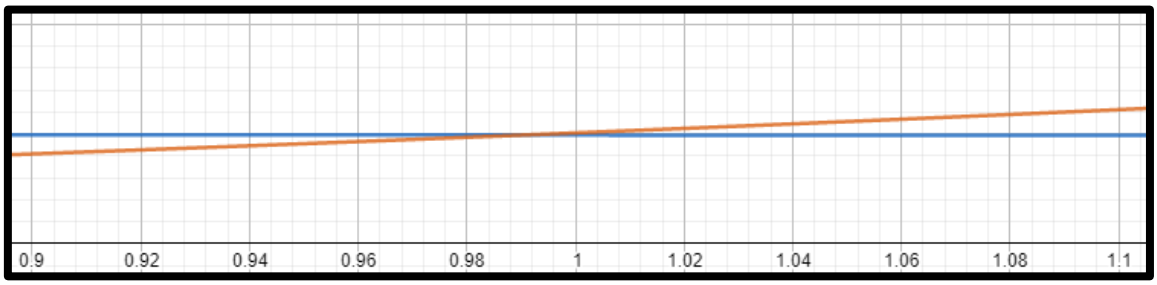
- $T(n) = an^2 + bn + c$
  $a * 100^2 + b * 100 + c = 0.01$
  $a * 200^2 + b * 200 + c = 0.02$
  $a * 300^2 + b * 300 + c = 0.05$
- $T(n) = 0.000001\, n^2 - 0.0002\, n + 0.02$

Se trata de una complejidad cuadrática $O(n^2)$.

A continuación, calcularemos la cota superior asintótica para $n_0 = 1$, ya que será la que nos permita visualizar mejor los resultados.

- $g(n) = n^2$
- $f(n) = 0.000001\, n^2 - 0.0002\, n + 0.02$
- $n_0 = 1$
- $c * 1^2 = 0.000001 * 1^2 - 0.0002 * 1 + 0.02$
  $c = 0.020201$

Por lo que $c(n) = 0.020201\, n^2$



Gráfica 7: Cota superior asintótica para el algoritmo de Greedy

$f(n) = 0.000001\, n^2 - 0.0002\, n + 0.02$ $\qquad$ $c(n) = 0.020201\, n^2$

Como podemos ver la función $c(n)$ es cota superior asintótica de $f(n)$ a partir del punto $x = 0.99, y = 0.02$, cuando n tiende a infinito.

Analizando el código brevemente, ya que este no es muy complejo y no consideramos que merezca la pena darle más importancia que lo explicado anteriormente, observamos que la complejidad reside fundamentalmente en un bucle anidado del cual extraemos nuevamente una complejidad polinómica de segundo orden.

Con todos los resultados obtenidos podemos concluir con que el algoritmo *Greedy* tiene una complejidad polinómica, en concreto, cuadrática/de segundo orden por lo que:

$$O(n^2)$$

6.- OPERADOR 2-OPT

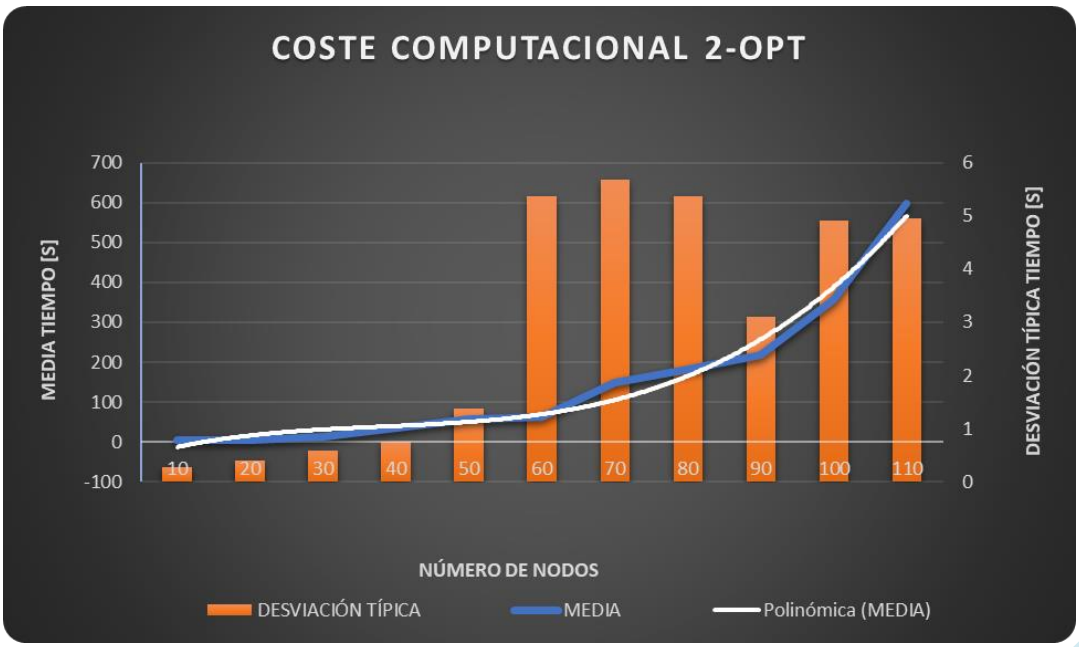
En esta sexta sección, el objetivo es estudiar el algoritmo de búsqueda con el operador *2-Opt* tanto empírica como analíticamente. Se calculará también la complejidad computacional y las diferencias finitas.

El algoritmo implementado se basa en una **búsqueda local completa** que compara todas las combinaciones válidas posibles mediante el **mecanismo de intercambio**. Esta técnica se puede aplicar al problema del viajante de comercio, así como a muchos problemas relacionados. En nuestro caso, la idea principal que existe detrás es la de tomar una ruta que se cruce sobre sí misma y volver a ordenarla para que no lo haga.

Se realizará el experimento con 11 pruebas posibles y 10 veces por cada caso, lo que supone una cifra de 110 experimentos totales. En este caso se hará uso de un bucle que devuelve la solución óptima combinado con dos bucles internos anidados que recorren las rutas comentadas. Una vez termina de comprobarlas se calcula si es se trata de la mejor solución y, en caso afirmativo, expulsa del primer bucle comentado, guardando la solución y devolviendo el tiempo computacional.

6.1.- Estudio empírico

Respecto al estudio empírico realizaremos **iremos aumentando el número de nodos cada 10** partiendo desde **10 nodos iniciales** hasta llegar a un máximo de **110 nodos**. Se han escogido estas cifras debido a que los tiempos computacionales son bastante elevados teniendo en cuenta el número de experimentos totales que se realizan, por lo que se ha tratado de seleccionar aquellas **pruebas más sencillas que permitan observar la complejidad lo antes posible**.

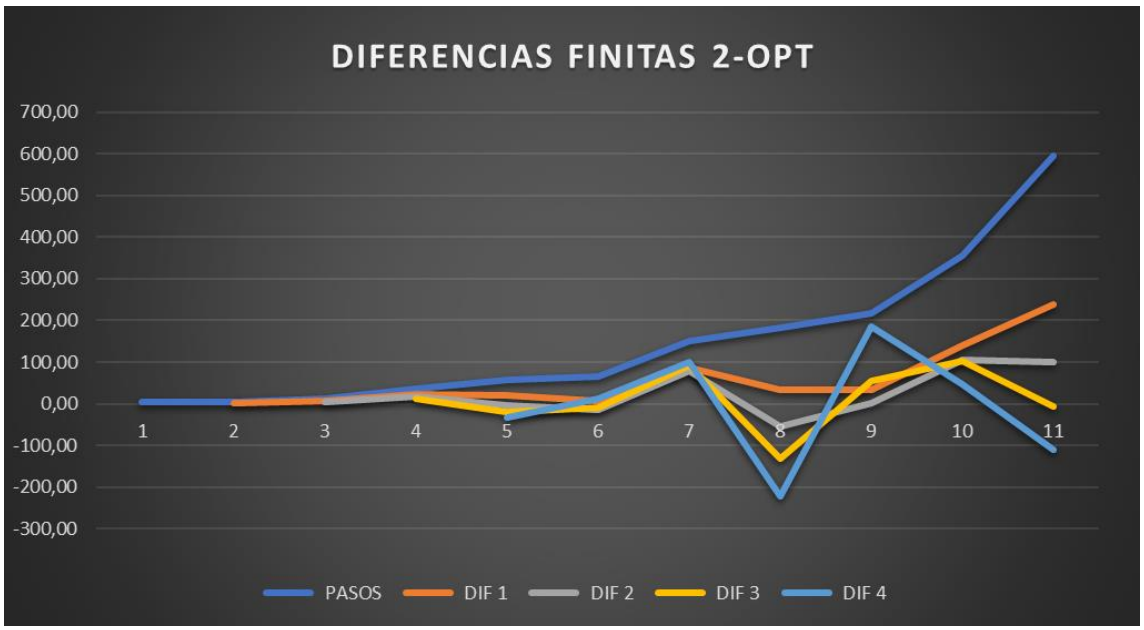


Gráfica 8: Coste computacional 2-OPT

Observamos una tendencia en los valores medios que se aproxima a una función de complejidad polinómica, en especial, de complejidad cúbica. Debido a esto, hemos trazado una línea de tendencia respecto a dicha media, que se trata de una **función polinómica de tercer grado** que parece encajar con la media obtenida. Esto nos da una pista de los valores que nos podrán llegar a salir en la parte analítica.

6.2.- Estudio analítico

A continuación, realizaremos el cálculo de las diferencias finitas para contrastar los resultados que se han obtenido. Debido a que se trata de una tabla excesivamente grande, expondremos los resultados en una gráfica y dejaremos como ejemplo de que su cálculo es correcto la tabla anexada previamente. La gráfica resultante es la siguiente.



Gráfica 9: Cálculo de diferencias finitas 2-OPT

En esta gráfica comprobamos que llegamos a valores nulos en el cuarto paso del cálculo de las diferencias finitas. Como ya ha ocurrido anteriormente, se repite el efecto de que a simple vista puede parecer que no es así. Como ya comentamos, esto se debe a que manejamos una **precisión muy elevada**, por lo que nunca obtendremos una línea recta en el valor nulo, si no que todo lo contrario, los decimales generarán en cada paso un **mayor ruido** perdiendo por completo la referencia.

Este suceso lo podemos observar también en la gráfica anexada previamente, donde, a medida que aumentamos el cálculo de diferencias finitas, el ruido se dispara. En este caso se produce el mismo efecto, pero a mayor escala ya que estamos en un orden de complejidad mayor. Para poder salir de dudas, calcularemos nuevamente el promedio de los pasos y de cada una de las cuatro diferencias obteniendo los valores 152.69, 59.29, 26.43, 11.75 y -2.5.

A partir de este último valor, sus diferencias finitas siguientes van en aumento negativo, por lo que ahora sí que podemos determinar con rotundidad que se trata del paso final y de una complejidad polinómica de tercer orden.

El cálculo de su función es:

- $T(n) = an^3 + bn^2 + cn + d$
  $a * 10^3 + b * 10^2 + c * 10 + d = 3.41$
  $a * 20^3 + b * 20^2 + c * 20 + d = 4.97$
  $a * 30^3 + b * 30^2 + c * 30 + d = 12.02$
  $a * 40^3 + b * 40^2 + c * 40 + d = 36.21$

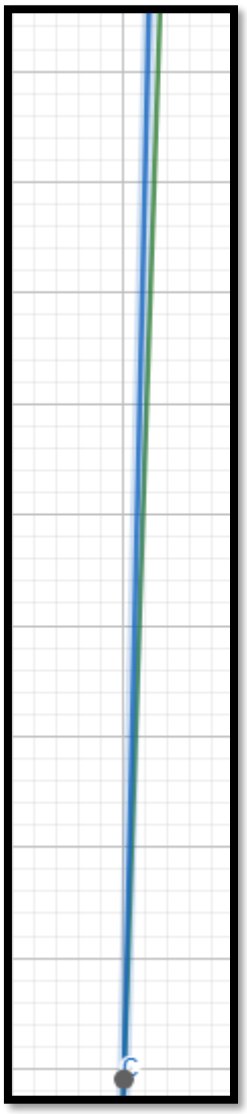
- $T(n) = 0.00194\, n^3 + 1.46833\, n^2 - 0.08905\, n - 4.31$

Se trata de una complejidad cúbica $O(n^3)$.

A continuación, calcularemos la cota superior asintótica para $n_0 = 10$, ya que será la que nos permita visualizar mejor los resultados.

- $g(n) = n^3$
- $f(n) = 0.00194\, n^3 + 1.46833\, n^2 - 0.08905\, n - 4.31$
- $n_0 = 1$
- $c * 10^3 = 0.00194 * 10^3 + 1.46833 * 10^2 - 0.08905 * 1 - 4.31$
  $c = 0.14357$

Por lo que $c(n) = 0.14357\, n^3$



Gráfica 10: Cota superior asintótica para el algoritmo de 2-OPT

$$f(n) = 0.00194n^3 + 1.46833n^2 - 0.08905n - 4.31 \qquad c(n) = 0.14357n^3$$

Como podemos ver la función $c(n)$ es cota superior asintótica de $f(n)$ a partir del punto $x = 10, y = 143$, cuando n tiende a infinito.

Analizando el código brevemente, ya que este no es muy complejo y no consideramos que merezca la pena darle más importancia que lo explicado anteriormente, observamos que la complejidad reside fundamentalmente en un triple bucle anidado del cual extraemos nuevamente una complejidad polinómica de tercer orden.

Con todos los resultados obtenidos podemos concluir con que el algoritmo *2-OPT* tiene una complejidad polinómica, en concreto, cúbica/de tercer orden por lo que:

$$O(n^3)$$

7.- CONCLUSIONES

En primer lugar, consideramos que la práctica ha servido para aunar los conocimientos adquiridos en las clases teóricas y en los dos trabajos anteriores. En este caso se nos ha brindado una mayor libertad para realizar los estudios, lo que ha puesto a prueba nuestra capacidad de decisión entre diferentes métodos según el problema encontrado. Cabe mencionar el seguimiento semanal de los hitos a través de las revisiones con el docente, lo cual queremos destacar, ya que ha sido de gran ayuda y es algo que valoramos y agradecemos enormemente.

Respecto a los problemas y dificultades que hemos tenido en la realización de esta práctica, estos han disminuido consideradamente respecto a las anteriores, pues nuestros conocimientos sobre el tema eran mayores. Por este motivo, los únicos problemas que hemos tenido son los de conseguir un código que se adaptase a lo pedido, del que poder realizar los estudios, y el tratar con complejidades factoriales como la de *Backtracking*.

En definitiva, un trabajo de cohesión para poner en práctica lo aprendido durante la asignatura, resuelta con aparente éxito si comparamos lo obtenido en los análisis empíricos y analíticos de cada una de las partes.

8.- BIBLIOGRAFÍA

[1] «El problema del viajante UC3M,» 2008. [En línea]. Available:

https://www.it.uc3m.es/jvillena/irc/practicas/07-08/ProblemaDelViajante.pdf

[2] «TSP Brand and Bound Opengenus,» 2020. [En línea]. Available:

https://iq.opengenus.org/travelling-salesman-branch-bound/

[3] «TSP Brand and Bound GeeksforGeeks,» 2022. [En línea]. Available:

https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/

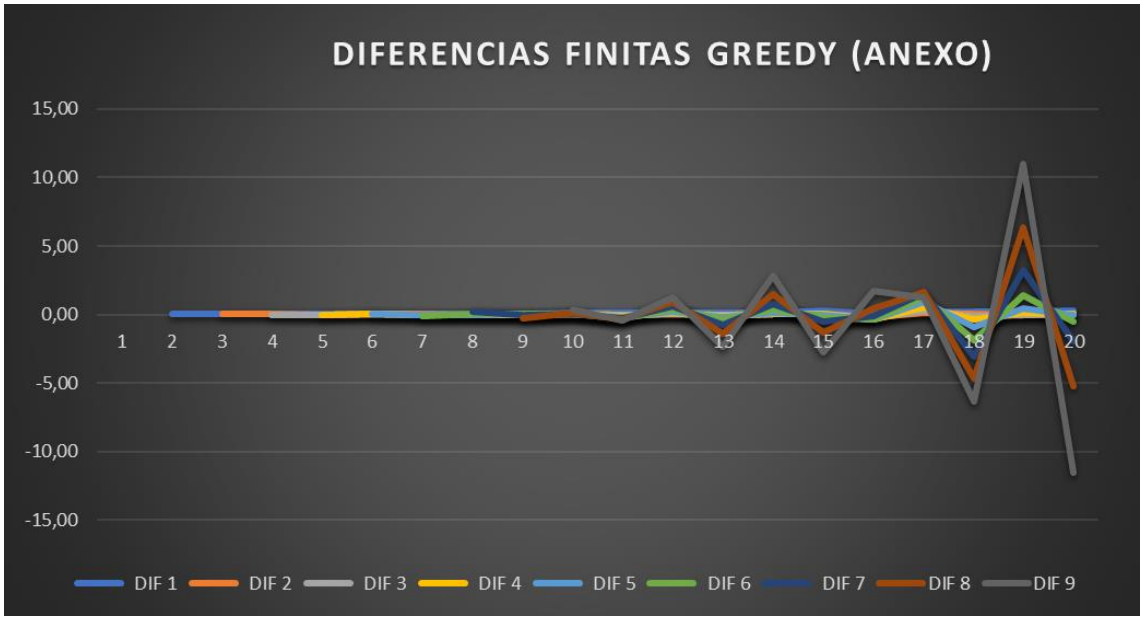
9.- ANEXO

Cálculo de diferencias finitas *Greedy*

N	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
TEMP	0,01	0,02	0,05	0,09	0,13	0,19	0,27	0,35	0,42	0,53	0,64	0,78	0,92	1,08	1,34	1,40	1,58	1,77	1,97	2,24
DIF 1		0,02	0,03	0,04	0,04	0,06	0,08	0,08	0,07	0,11	0,10	0,14	0,14	0,17	0,25	0,07	0,18	0,19	0,20	0,27
DIF 2			0,01	0,02	0,00	0,02	0,02	0,00	-0,01	0,04	-0,01	0,04	0,00	0,03	0,09	-0,18	0,11	0,01	0,01	0,07
DIF 3				0,01	-0,02	0,02	0,00	-0,02	-0,02	0,06	-0,05	0,05	-0,04	0,03	0,06	-0,27	0,29	-0,10	0,00	0,06

Tabla 2: Cálculo de diferencias finitas Greedy

Gráfica completa del cálculo de diferencias finitas Greedy



Gráfica 11: Gráfica completa del cálculo aumentado de diferencias finitas