EJERCICIO PRÁCTICO

PROBLEMA DEL VIAJANTE

10/05/2021 · UC3M · GRUPO 50

Hugo Romero	Irene Fernández	Maribel Medrano
100 386 322	100 386 294	100 386 149

CONTENIDO DEL DOCUMENTO

INTRODUCCIÓN

A lo largo de este documento se abordará la versión simétrica del problema del viajante. Las primeras matemáticas desarrolladas en torno al problema, en versiones específicas del mismo, datan del siglo XIX, por los matemáticos Sir William Rowan Hamilton, irlandés, y Thomas Penyngton Kirkman, inglés. El primero fue el desarrollador del juego Icosian, lanzado en 1857, y cuyo objetivo es encontrar un ciclo hamiltoniano por las aristas de un dodecaedro.

Los primeros estudios en profundidad de la versión general del problema del viajante se le asignan a Karl Menger, un matemático vienés hijo del famoso economista Carl Menger. El cual le dio por primera vez un nombre, aunque no sería el definitivo en su artículo de 1930, donde se puede leer:

"Lo llamamos el problema del mensajero (porque en la práctica el problema debe ser resuelto por todos los carteros, y también por muchos viajeros): encontrar la ruta mínima uniendo todos de un conjunto finito de puntos, cuyas distancias por parejas son conocidas".

Finalmente, un libro alemán de 1932 acoge por primera vez el término "traveling salesman" que vendría a ser "vendedor ambulante". Este libro, escrito precisamente por un vendedor ambulante experimentado, tiene por título "El problema del vendedor, como debería ser y qué debería hacer para ser exitoso en su negocio". Pese a que el objeto del libro no es el problema de ruta entre ciudades, en el último capítulo menciona "Con una buena selección y planificación de la ruta, uno puede normalmente ganar tanto tiempo que debemos hacer sugerencias…El aspecto más importante es cubrir tantos sitios como podamos sin visitar ninguno más de una vez.". Con el tiempo el problema pasó a ser conocido como el Traveling Salesman Problem, al que en este documento se hará referencia en su versión abreviada, TSP.

BÚSQUEDA LOCAL

La búsqueda local es una familia de métodos no completo de resolución de problemas de optimización. De forma general, la estructura de resolución se basa en partir de una solución inicial y tratar de mejorarla realizando modificaciones locales basadas en la vecindad de los estados.

Muchos algoritmos solo avanzarán al estado vecino si es mejor que el actual, esto genera la problemática de que el algoritmo quede estancado en un mínimo local, otros algoritmos, aunque con un mayor coste computacional, pueden permitir saltos a estados peores para tratar de salvar esta problemática, este es el caso del algoritmo de recocido simulado.

Un factor de gran relevancia a la hora de encontrar buenas soluciones es la solución de partida, dado que la probabilidad de que la solución arrojada por búsqueda local sea subóptima es tan elevada, tener un buen punto de partida es vital para el buen desempeño de la solución.

En este trabajo se plantean tres puntos de partida para comparar las diferencias finales tras aplicarles el mismo algoritmo de búsqueda local posterior.

SOLUCIÓNES INICIALES

Se plantean tres soluciones iniciales, solución aleatoria, solución greedy y solución r. Posteriormente se hará una comparación de la actuación de las tres.

SOLUCIÓN ALEATORIA

La solución aleatoria utiliza el método shuffle de la librería random para generar una solución de este tipo. Este método está basado en el algoritmo Fisher-Yates de generación de permutaciones aleatorias.

```
para cada i desde n-1 a 1:
      j <- número aleatorio tal que 0≤j≤i
      intercambiar a[j] y a[i]
```

La idea del algoritmo original era el concepto de ir sacando papeles de un sombrero de forma aleatoria, no obstante, la implementación de este concepto podía dispararse a la complejidad cuadrática con facilidad. La mejora moderna, introducida por Richard Durstenfeld tiene el mismo funcionamiento, pero la lista que simboliza el sombrero va dejando al final los números sacados, y al principio los que quedan dentro. El siguiente pseudocódigo representa el algoritmo, la lista es *a* y tiene tamaño *n*.

El bucle se ejecutará n-1 veces, los procedimientos interiores tienen complejidad constante O(1). Por tanto, la complejidad final del algoritmo en O(N). El incremento temporal que provenga de este algoritmo será despreciable para los TSP abordables por cualquier algoritmo completo.

En la implementación propuesta, solo se mezclan los elementos del segundo en adelante, para mantener el primer nodo como inicial, de todas formas, la complejidad se mantiene en O(N).

SOLUCIÓN GREEDY

La solución Greedy hace uso de las ventajas de la estructura de datos set, para el caso del borrado de un elemento dado en tiempo constante. Este método se basa en la construcción de un recorrido al cual se van añadiendo nodos de forma sucesiva. En cada caso se seleccionará el siguiente nodo (ciudad) más cercano al último nodo añadido que todavía no se ha visitado.

```
Set_por_poner <- set(a[1:])
Solucion <- lista vacía

Mientras set_por_poner tenga más de un elemento:
    actual = último elemento de solución
    mejor <- el elemento de set_por poner cuya arista con actual sea menor
                        (complejidad O(len(set_por_poner))
    eliminar mejor de set_por_poner
    insertar mejor al final de solución
insertar en mejor el elemento
```

SOLUCIÓN R-SOLVE

```
def r_solve(self):
        x <- [coord[0] para cada coord perteneciente a self.problema.values()]
        y <- [coord[1] para cada coord perteneciente a self.problema.values()]
        center <- [media de x, media de y]
        self.solution.sort(key=lambda point: angle(self.problema[point], center))
```

Para obtener la complejidad de esta función se han calculado las complejidades de realizar cada uno de los cálculos necesarios.

En primer lugar, tanto para obtener x como para obtener y, la complejidad resultante es lineal, es decir de $O(N)$, siendo N la dimensión del problema.

Por otro lado, el cálculo de la media de x e y supone una complejidad lineal, obteniendo una complejidad total del cálculo de "center" de $O(N)$.

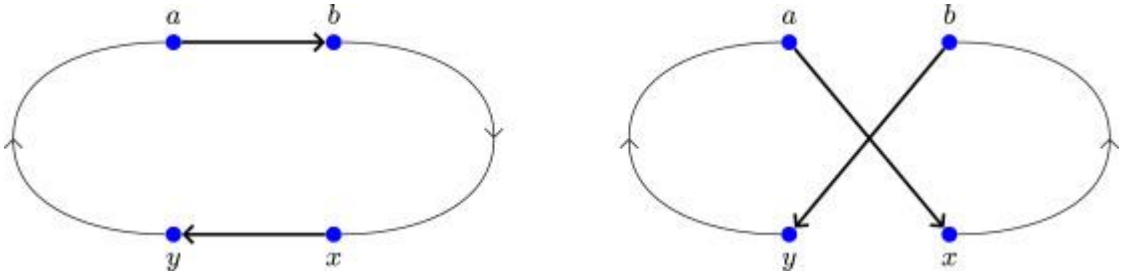
Por último, se ordena la solución, con el algoritmo de ordenación inherente a Python. Para la función sort, Python utiliza el algoritmo Timsort un algoritmo de ordenación derivado de merge sort e insertion sort y cuya complejidad tanto en el caso medio como en el peor caso es de $O(N \log(N))$.

La complejidad total de la función r_solve viene determinada por la mayor complejidad de las obtenidas previamente, es decir, es de $O(N \log(N))$.

MEJORAS LOCALES

2-OPT

El algoritmo 2-opt consiste en la aplicación de un mecanismo de cambio. El mecanismo de cambio consiste en la reestructuración del grafo de la siguiente manera:



Si el cambio produce mejora en el grafo, se mantendrá, si no se seguirán buscando cambios hasta que se produzca mejora. En el momento en el que se buscan todos los cambios posibles y ninguno produce mejora, el algoritmo termina. Se puede entender mejor con el pseudocódigo:

```
repita hasta que no se haga ninguna mejora:
    mejor_distancia = calcularTotalDistancia (ruta_existente)
    empezar de nuevo:
    para cada i desde 1 a self.dimension - 2:
         para cada j desde i+2 a self.dimension:
            nueva_ruta = 2optSwap (ruta_existente, i, k)
            new_distance = calcularTotalDistance (nueva_ruta)
            if (nueva_distancia <mejor_distancia):
                ruta_existente = nueva_ruta
                mejor_distancia = nueva_distancia
                Ir a empezar de nuevo
```

El peor caso es en el que no se mejora la distancia hasta el final, es decir, cuando se recorren los dos bucles completamente.

El cálculo inicial de la mejor distancia es lineal.

Siendo n la dimensión del problema, la complejidad de recorrer los dos bucles anidados es igual a:

$$\sum_{i=1}^{n-3} i = \frac{(n-3)(n-3+1)}{2} = \frac{n^2-5n+6}{2} \rightarrow O(N^2)$$

La complejidad de calcular la nueva distancia es de $O(N)$.

En cuanto a la complejidad de la comparación entre la nueva distancia y la mejor distancia anterior, es constante, así como la de las operaciones contenidas dentro de la condición.

Teniendo en cuenta los cálculos realizados previamente, se obtiene que la complejidad total del algoritmo 2-OPT es de $O(N^3)$.

Esta complejidad puede ser reducida a $O(N^2)$ si en lugar de calcular la nueva distancia completa tras el cambio se analiza únicamente las dos aristas correspondientes a la reestructuración del grafo por parte del opt-2. No obstante para los efectos de este trabajo no era necesaria la implementación de esta mejora dado que la complejidad de los algoritmos completos siempre será mayor independientemente del grado del exponente de la complejidad de búsqueda local. Por ello se deja esta forma de implementación del algoritmo.

COMPARATIVA ALGORITMOS DE BÚSQUEDA LOCAL

Una vez presentados los algoritmos de soluciones iniciales y el algoritmo 2-opt de mejora local, se comienza el estudio de los algoritmos de búsqueda local, tanto su complejidad como su efectividad.

Bajo los efectos de este trabajo, un algoritmo de búsqueda local se compone de la combinación de un algoritmo de solución inicial con un algoritmo de mejora local. Por tanto, se deduce que se estudiarán tres algoritmos la solución aleatoria+2-opt, la solución greedy+2-opt y la solución r-solve+2-opt.

COMPLEJIDAD

Dado que la complejidad de los tres métodos de solución inicial está contenida en la complejidad del algoritmo 2-opt, los tres algoritmos de búsqueda local tienen el orden de complejidad del 2-opt.

Por tanto, los tres algoritmos de búsqueda local tienen complejidad $O(N^3)$.

DESEMPEÑO

Teniendo tres algoritmos distintos pero cuya complejidad es la misma, tiene sentido realizar una comparación de cuál es el desempeño de cada uno de los algoritmos. Para ello, se verá cuanto es capaz de acercarse a la solución ideal cada uno de los algoritmos. Dado que cuantas más ciudades haya en el escenario más complejo es el problema y más difícil será obtener una solución buena por un algoritmo no completo, se representa la bondad del algoritmo para cada tamaño de escenario.

En primer lugar, la tasa de acierto, es decir, cuantas veces encuentra la mejor solución en cada dimensión. La siguiente tabla muestra el porcentaje de veces que el algoritmo logró encontrar la ruta óptima para cada dimensión.

dim. / alg.	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
random + 2-opt	63.4	31.0	25.0	32.5	12.5	5.0	20.0	7.5	0.0	2.5	2.5	7.5	10.0	7.5	0.0	0.0	0.0

greedy + 2-opt	68.8	47.6	37.5	32.5	20.0	30.0	20.0	12.5	10.0	10.0	10.0	10.0	7.5	12.5	8.3	10.0	5.0
r-solve + 2-opt	78.5	76.2	67.5	72.5	62.5	60.0	50.0	55.0	50.0	30.0	37.5	25.0	35.0	27.5	29.2	25.0	20.0

Esta tabla empieza a indicar que el mejor algoritmo será el que tiene por solución inicial el r-solve. No obstante, es posible que pese a acertar más, en los casos en los que no acierta arroje soluciones muy alejadas a la correcta. Se dan muchos casos en la vida real en los que se valora más una solución cercana a la buena y obtenida de forma rápida que la buena con mucho retraso.

Si además del porcentaje de acierto, se quiere estudiar cómo de buenas son las soluciones que plantea cada algoritmo, se debe plantear otra forma de representar los datos. La siguiente tabla es similar a la anterior, pero muestra las ratios de cada algoritmo respecto a la solución óptima. La ratio se calcula como el cociente de la solución planteada entre la solución óptima.

Es decir, para los escenarios de dimensión *i* y el algoritmo random con 2-opt, la ratio será:

$$\frac{\frac{\sum_{e \epsilon Escenarios\_dimension\_i} solucion\_greedy\_2opt_e}{|Escenarios_{dimension_i}|}}{\frac{\sum_{e \epsilon Escenarios\_dimension\_i} solucion\_optima_e}{|Escenarios_{dimension_i}|}}$$

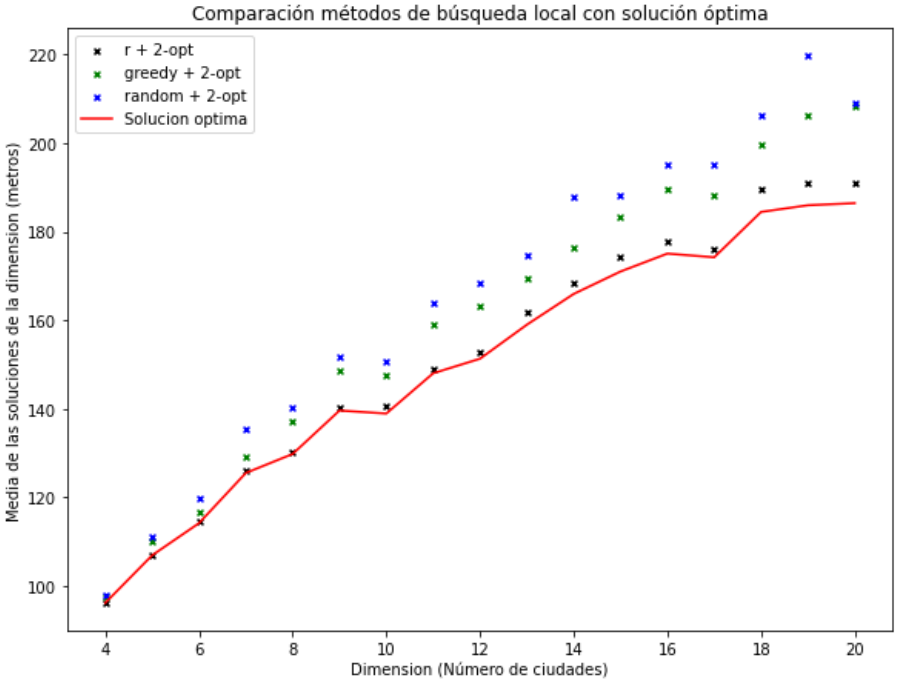
De forma más simplificada:

$$\frac{media(soluciones\_greedy\_2opt_i)}{media(soluciones\_optimas_i)}$$

Esta ratio representa cómo de lejos se queda el algoritmo de la solución óptima, idealmente la ratio será 1. En caso de no serlo, cuanto mayor sea, peor será el desempeño del algoritmo.

dim. / alg.	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
random + 2-opt	1.016	1.040	1.048	1.078	1.081	1.086	1.085	1.108	1.113	1.099	1.132	1.100	1.114	1.119	1.118	1.181	1.121
greedy + 2-opt	1.014	1.028	1.022	1.028	1.057	1.065	1.063	1.074	1.078	1.066	1.063	1.072	1.084	1.080	1.083	1.110	1.117
r-solve + 2-opt	1.000	1.001	1.004	1.003	1.002	1.004	1.013	1.006	1.010	1.017	1.015	1.018	1.016	1.011	1.029	1.026	1.024

Para una visualización gráfica se representa la media de distancias de ruta de cada uno de los algoritmos para cada dimensión de escenario. La línea roja representa la media de las soluciones óptimas, es decir, el límite inferior para cualquier algoritmo, cuánto más cercano esté un punto a la línea mejor será el desempeño del algoritmo.



Tanto por la tabla de las ratios como por las gráficas podemos concluir que el algoritmo que ofrece mejor desempeño es el que toma el r-solve como punto de partido, seguido del greedy y, por último, como era de esperar, el que parte de una solución aleatoria.

BÚSQUEDA COMPLETA

SOLUCIÓN BACKTRACKING

Esta solución se basa en encontrar el ciclo hamiltoniano mínimo, es decir, el mínimo camino que visita todos los vértices del grafo una sola vez y el vértice de partica es también el último.

```
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
```

Dado que los resultados obtenidos con esta metodología superan enormemente en tiempo a los de la solución de Branch and Bound, que ha sido la que más se ha estudiado en este trabajo, el análisis de la solución de backtracking se mantendrá en un nivel teórico analítico.

El algoritmo consiste en encontrar la ruta mínima empezando por cada uno de los vecinos, y devuelve la mínima entre ellas. Cuando estamos en un escenario de n ciudades tenemos n posibilidades de comienzo y posteriormente se hará una llamada para cada subgrafo. En un grafo completo de n vértices se puede demostrar que el número de subgrafos viene dado por 2^n. Además, cada llamada, de nuevo

recorrerá el bucle para cada vecino, lo que es de nuevo n en el peor caso. Juntando las tres etapas se obtiene que la complejidad es $O(n * 2^n * n) = O(n^2 * 2^n)$.

SOLUCIÓN BRANCH AND BOUND

Esta solución consiste en encontrar para cada nodo actual de un árbol la mejor solución posible si continuamos por ese nodo y, si esa solución es peor que la calculada hasta ese momento se ignora el subárbol que se obtiene de ese nodo.

El cálculo del coste para cada nodo incluye tanto el coste para alcanzar ese nodo como el coste de llegar a la solución a través de ese nodo.

La complejidad de esta solución vendrá determinada por la heurística, es decir, por el cálculo de la mejor solución posible a través de un nodo. Ya que, para esta solución, el peor caso es la resolución del problema mediante fuerza bruta, esto ocurrirá cuando no sea posible podar ningún nodo.

Es decir, la complejidad para el peor caso es de $O(N!)$.

El uso de este algoritmo es el que ha proporcionado los casos que han servido de base para los estudios de este mismo trabajo, por ejemplo, para los algoritmos de búsqueda local. Para cada dimensión estudiada se han generado escenarios de forma aleatoria y se ha almacenado en el formato de archivos .tsp. Posteriormente se ha obtenido la solución óptima con el algoritmo de Branch and Bound y se ha almacenado de forma que se pueda relacionar con el escenario correspondiente en el formato de archivos .opt.tour. Tras las diferentes iteraciones de mejoras del algoritmo y pruebas se han obtenido el siguiente número de problemas con la solución óptima para cada dimensión:

Dimensión	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Archivos de escenario generados	93	42	40	40	40	40	40	40	40	40	40	40	40	40	24	20	20

No obstante, para el algoritmo en su forma final se han generado tan solo 20 casos por generación, en este caso, además de servir para extender la base de datos de problemas resueltos, se han almacenado los tiempos de ejecución de cada caso para poder realizar el análisis correspondiente.

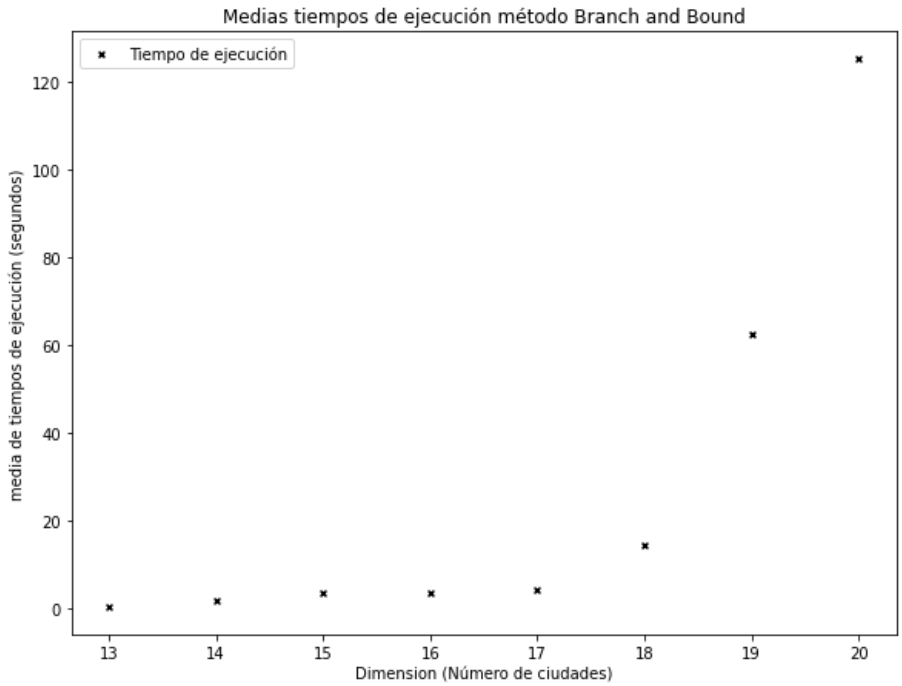
Para el análisis solo se mantendrán aquellas dimensiones cuyo tiempo medio de ejecución sea relevante, se ha elegido como criba que el tiempo sea mayor a una décima de segundo para no reducir en exceso el número de dimensiones mantenidas. Cabe destacar que como se está usando el decorador jit de numba, el primer caso tiene una penalización debido al tiempo de compilación, la penalización es sufrida en la generación del primer escenario de dimensión 4, y es de 1’25 segundos. No obstante, no se mete la dimensión 4 en el análisis porque el tiempo no se corresponde con el tiempo de ejecución del algoritmo. Se mantienen por tanto las dimensiones entre 13 y 20, ambos incluidos. Una breve descripción de los datos arroja los siguientes resultados:

Dimensión	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Número de casos	20	20	20	20	20	20	20	20
Media de tiempos	0.328s	1.720s	3.370s	3.630s	4.290s	14.3s	62.3s	125s
Desviación típica de los tiempos	0.427s	3.49s	3.72s	3.6s	5.27s	17.1s	85.8s	163s

Tiempo mínimo	0.032s	0.076s	0.243s	0.121s	0.185s	1.2s	2.32s	15.4s
Tiempo máximo	1.75s	12.3s	13s	13.5s	17.7s	58.9s	287s	452s

Se puede observar que la desviación de la típica es muy grande, incluso superior a la media, lo que se va a traducir en una mayor dificultad de obtención de buenos resultados a la hora de analizar la complejidad del algoritmo.

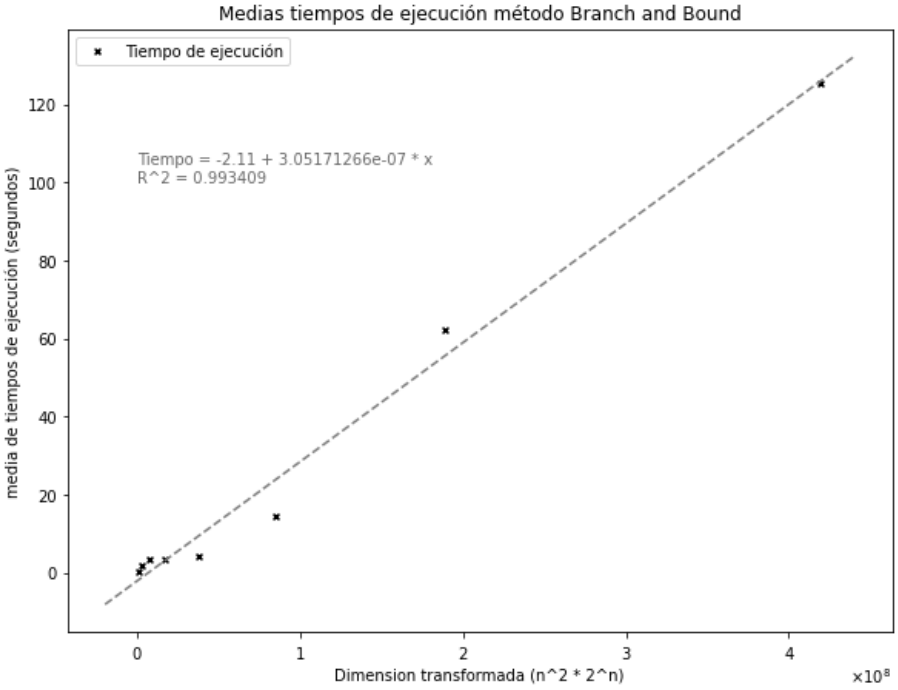
No obstante, a continuación, se representan las medias de los tiempos de ejecución para cada dimensión:



En el estudio analítico se ha determinado la complejidad del algoritmo para el peor caso como $O(N!)$. Sin embargo, la obtención de peor caso no es trivial y por tanto no se puede contrastar con los resultados empíricos dado que en la práctica difícilmente se obtendrá un caso en el que no se produzca poda en el algoritmo. Por ello, en el plano empírico, se realizará el estudio para el caso medio del algoritmo para ver a qué complejidad podemos aproximar el caso medio del algoritmo. Esto no desciende directamente del estudio analítico, pues cuantos más nodos, más posible es que se produzca poda por tanto se realizan transformaciones a los ejes hasta obtener una gráfica lo más lineal posible.

De nuevo se hace hincapié en que el hecho de que las desviaciones de cada muestra sean tan grandes hace que no se pueda concluir con certeza los resultados que se obtengan. De hecho, para entender el error que puede que se esté introduciendo, se puede generar un intervalo de confianza para la media poblacional. En el caso de 20 ciudades se obtiene que, para poder afirmar que la media se encuentra en el intervalo especificado con una confianza del 95%, el intervalo que hay que establecer es el (53’56, 196’44). El error por tanto está asegurado; no obstante, se continua con el estudio asumiendo que la media total (poblacional) coincide con la media que se ha obtenido (muestral).

El culmen del proceso de transformaciones se obtiene cuando hacemos la transformación que arrojaría una complejidad, que curiosamente se corresponde con la complejidad estudiada en el algoritmo dinámico:



Como se ve, el R^2 es muy cercano a 1. Así que daremos este resultado como el resultado del estudio empírico realizado, la transformación realizada al eje x es $f(x) = x^2 * 2^x$. Por ello decimos que, en el caso medio, la complejidad del algoritmo propuesto de ramificación y poda tiene una complejidad de $O(n^2 * 2^n)$. Pese a tener la misma complejidad que el propuesto en backtracking, los tiempos obtenidos son mucho mejores, lo que lleva a pensar que, pese a que la complejidad es la misma, la constante que acompaña a la complejidad en la práctica es mucho menor para el caso de ramificación y poda.

IMPLEMENTACIÓN

El algoritmo implementado es un algoritmo de búsqueda en profundidad y poda con el uso de una heurística. Cuando el coste del nodo actual más la heurística aplicada al mejor nodo actual es superior a la mejor solución obtenida hasta el momento se detiene toda la ramificación del nodo actual.

La implementación se ha realizado en Python, pero, para lograr velocidades similares a lenguajes como C, se ha hecho uso de la librería numba. Esta librería permite la compilación de partes de código Python a código nativo. Existen dos usos principales usos de la librería numba, el decorador *@jit* y el decorador *@njit*. Cuando un código no está optimizado para su uso con numba, el decorador *@jit* hará la compilación y advertirá de que el código debería modificarse para obtener todo el potencial de la librería. Por su parte *@njit* arrojará error ante la misma situación. Por ello se ha utilizado la versión *@njit*, para buscar la máxima mejora en el código.

Otra forma de mejorar el rendimiento del programa ha sido la búsqueda de heurísticas que, sin sobreestimar costes, apuren lo máximo posible. La primera implementación de heurística consistía en la suma de las aristas mínimas de cada una de las ciudades aún no visitadas. Finalmente, y dado que cada una de esas ciudades no visitadas tendrá que hacer uso de dos de sus aristas, se puede recoger este hecho como que la arista usada por la ciudad tiene un coste que es la media de sus dos menores.

```
Heuristica(estado):
       Coste_estimado = 0
       para cada ciudad c no vistada:
              Coste_estimado <- Coste_estimado + (aristaMinima(c) + aristaSegundaMinima(c))/2
       Devolver Coste_estimado
```

Por tanto, la heurística utilizada ha sido:

En el siguiente punto se recoge el proceso de implementación de dicha heurística.

OPTIMIZACIONES HEURÍSTICA MIN2

Dada la heurística de la media de las dos aristas mínimas se han buscado formas de optimizar el propio cálculo de la heurística, pues es un cálculo que se realiza en cada llamada a la función. Recordemos que la heurística consiste en sumar al coste total, para cada ciudad no visitada, la media de las dos menores aristas que conectan con esa ciudad.

A la hora de realizar este estudio, no se trabajará con la función 100% adaptada a la forma del TSP, sino que se hará una versión que represente la funcionalidad de forma más básica, pero con mismo núcleo.

Por ello, se eliminan las verificaciones de nodos visitados y se dejan funciones que dada una matriz de grafo TSP, sumen a un acumulador la media de los dos menores valores estrictamente positivos de cada fila. Con este cambio se pretende aumentar el tamaño de entrada para obtener tiempos significativos, entendiendo por tamaño de entrada el número de filas de la matriz (también de columnas).

Tras muchas implementaciones que logran hacer el mismo objetivo, se han obtenido tres funciones “finalistas” candidatas a ser la definitiva.

Primera forma: min2_1

La primera implementación está basada en realizar el recorrido de los arrays e ir comparando los elementos con los valores mínimo y segundo mínimo actuales para mantener los más pequeños.

```
def min2_1(arr: np.array):
    bound = 0
    for i in range(arr.shape[0]):
        minim = np.inf
        second_minim = np.inf
        for j in range(arr.shape[0]):
        if arr[i][j] > 0 and arr[i][j] < minim:
            second_minim = minim
            minim = arr[i][j]
        elif arr[i][j] > 0 and arr[i][j] < second_minim:
            second_minim = arr[i][j]
        bound = bound + (minim+second_minim)/2
    return bound
```

Segunda forma: min2_2

La segunda implementación se basa en el método partition de la librería numpy. Dado que los métodos primitivos de esta librería suelen estar muy bien optimizados puede ser una buena candidata.

```
def min2_2(arr: np.array):
    bound = 0
    for i in range(arr.shape[0]):
        A, B = np.partition(arr[i], 2)[1:3]
        bound = bound + (A+B)/2
    return bound
```

Tercera forma: min2_3

La tercera forma de implementar la función se espera que sea peor dado que requiere de una ordenación. No obstante, es mencionada en foros de internet cuando se busca la obtención de elementos mínimos de un array, por lo que se incorpora a la comparativa para refutar definitivamente este método.

```
def min2_3(arr: np.array):
    bound = 0
    for i in range(arr.shape[0]):
        bound = bound + np.mean(np.sort(arr[i])[1:3])
    return bound
```

Comparativa

Esta comparativa intermedia del coste computacional es tangencial, pero externa, al propósito de este trabajo, por ello basaremos la decisión en un estudio meramente empírico.

El estudio se ha hecho para una generación de grafo de 1000 nodos, se repite 50 veces el cálculo para cada candidata y se realiza la media de los tiempos. El resultado es el siguiente:

Función	min2_1	**min2_2**	**min2_3**
Tiempo medio	1833,8ms	11,47ms	57,37ms

La mejor forma de implementar la heurística sería con la segunda forma de algoritmo. Sin embargo, con muy poco esfuerzo extra, podemos probar qué ocurre si utilizamos la librería numba y añadimos el decorador de jit a la cabecera de cada una de las funciones. En este caso se obtienen los siguientes resultados:

Función	min2_1_jit	**min2_2_jit**	**min2_3_jit**
Tiempo medio	1,80ms	13,73ms	55,29ms

La mejora obtenida sobre la primera función ha sido dramática, tanto que sobrepasa a las otras dos funciones en cualquiera de sus versiones. Recordemos que el uso de jit produce una penalización sobre la primera iteración debido al tiempo de compilación, por ello para los cálculos de esta versión se ha desestimado el primer resultado y se ha hecho la media de los 49 restantes. Esta forma de proceder

será más fiel a la realidad, dado que a la hora de la verdad el cálculo se va a hacer tantas veces que el tiempo extra del primer cálculo no impactará de forma perceptible sobre el total de tiempo gastado en la función.

Finalmente, la implementación usada será la de la función min2_1, en su versión de jit.

CONCLUSIONES

En esta práctica se han presentado distintas soluciones para encontrar un ciclo hamiltoniano que recorra todas las ciudades, utilizando tanto soluciones de búsqueda local con la implementación de mejoras locales, como soluciones de búsqueda completa implementadas con optimizaciones heurísticas que permiten disminuir el tiempo de ejecución.

Además, se ha hecho un estudio de cada una de las implementaciones tanto analítico como empírico utilizando tablas y gráficas para una mejor visualización, lo que ha permitido conocer las diferencias entre las distintas implementaciones.

Todo esto se ha realizado poniendo en práctica las ideas impartidas en la asignatura, así como las competencias adquiridas a lo largo de los cursos, lo que nos ha permitido utilizar recursos matemáticos, de programación y de búsqueda de información para la correcta realización de la práctica.

BIBLIOGRAFÍA

Wikipedia contributors. (2021, April 19). Icosian game. In *Wikipedia, The Free Encyclopedia*. Retrieved 10:19, May 17, 2021,
from https://en.wikipedia.org/w/index.php?title=Icosian_game&oldid=1018803255

Wikipedia contributors. (2020, December 31). Timsort. In *Wikipedia, The Free Encyclopedia*. Retrieved 10:11, May 17, 2021,
from https://en.wikipedia.org/w/index.php?title=Timsort&oldid=997404113

LAPORTE, G. (2006) *A Short History of the Traveling Salesman Problem.* Available on: http://neumann.hec.ca/chairedistributique/common/laporte-short.pdf

Michalewicz Z. (1996) The Traveling Salesman Problem. In: Genetic Algorithms + Data Structures = Evolution Programs. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-03315-9_11

Sebastian Oberhoff (https://cs.stackexchange.com/users/63772/sebastian-oberhoff), Analysis of time complexity of travelling salesman problem, URL (version: 2018-04-03):
https://cs.stackexchange.com/q/90152