UNIVERSIDAD DE GRANADA

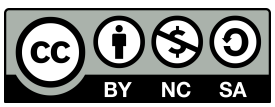
Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

Práctica 3: Algoritmos Greedy

Doble Grado Ingeniería Informática y Matemáticas

Autores:
Jose Alberto Hoces Castro
Javier Gómez López
Moya Martín Castaño

Mayo 2022



Índice

1. Introducción **3**

2. Desarrollo **3**
2.1. Ejercicio 1. Contenedores 3
2.1.1. Primer ejercicio 3
2.1.2. Segundo ejercicio 5
2.2. Ejercicio 2. El problema del viajante de comercio 6
2.2.1. Heurística del vecino más cercano 7
2.2.2. Heurística por inserción 11
2.2.3. Heurística propia: perturbaciones 14
2.2.4. Comparación de heurísticas 19

3. Conclusiones **20**

1. Introducción

El objetivo de esta práctica es aprender a implementar y utilizar algoritmos *"greedy"* o voraces para resolver problemas de manera rápida aunque no por ello menos óptima. Para ello, se plantean los siguientes dos problemas:

- **Ejercicio 1** (Contenedores): Se quiere rellenar un buque mercante con una cierta capacidad de peso con contenedores, cada uno de los cuales tiene su propio peso.
- **Ejercicio 2** (TSP): El problema del viajero. Se quiere recorrer una serie de ciudades, pasando por ellas solo una vez y volviendo al punto de partida. Se quiere encontrar la ruta más óptima.

2. Desarrollo

Para el primer ejercicio, nos centraremos en identificar el problema de los contenedores como un problema Greedy identificando sus características. También, justificaremos los optimalidad o la no optimalidad de los algoritmos Greedy desarrollados.

Para el análisis de los algoritmos del viajante de comercio que desarrollaremos, hemos realizado los siguientes pasos:

1. Un **análisis teórico** donde se comentará el código desarrollado para la resolución de los problemas propuestos.
2. Un **análisis empírico** donde hemos ejecutado los algoritmos en nuestros ordenadores bajo las mismas normas y condiciones. Hemos compilado usando la optimización `-Og`. Además, hemos usado como *datasets* de pruebas los datos proporcionados por la profesora en el caso del TSP, y valores aleatorios de los pesos de los contenedores para el primer problema. Por otro lado, para automatizar el proceso, hemos creado unos *scripts* de generació de datos de prueba y de ejecución de nuestros programas. Hemos ejecutado cada algoritmo 15 veces en cada uno de los tamaños probados, y hemos hecho la media de ellos para reducir perturbaciones que puedan alterar el resultado.
3. Un **análisis híbrido** donde hemos tomado los datos de cada uno de los alumnos del grupo y hemos hallado la K (constante oculta). Para ello hemos usado gnuplot.

2.1. Ejercicio 1. Contenedores

El enunciado del problema es el siguiente: *Se tiene un buque mercante cuya capacidad de carga es de* K *toneladas y un conjunto de contenedores* $c_1, \ldots, c_n$ *cuyos pesos respectivos son* $p_1, \ldots, p_n$ *(expresados también en toneladas). Teniendo en cuenta que la capacidad del buque es menor que la suma total de los pesos de los contenedores:*

- *Diseñe un algoritmo que maximice el número de contenedores cargados, y demuestre su optimalidad.*
- *Diseñe un algoritmo que intente maximizar el número de toneladas cargadas.*

2.1.1. Primer ejercicio

Nuestro objetivo es que podamos cargar el máximo número de contenedores en un buque mercante de K toneladas. Para ello, el algoritmo que nosotros proponemos es tomar los contenedores, ordenarlos de menor a mayor peso, y comenzar añadiendo los de menor peso. De esta forma, podemos cargar más contenedores, ya que si empezásemos por los de pesos intermedios o mayores, acabaríamos cargando menos.

Veamos las 6 características de nuestro problema Greedy:

- **Un conjunto de candidatos**: En este caso, los contenedores a cargar.
- **Una lista de candidatos ya usados**: Los contenedores que ya han sido cargados.
- **Un criterio que dice cuándo un conjunto de candidatos forma una solución**: El criterio es que la suma de los pesos de un conjunto de contenedores no sea superior a las K toneladas del buque.
- **Un criterio que dice cuándo un conjunto de candidatos es factible (podrá llegar a ser una solución)**: el conjunto de contenedores que se evalúe no debe superar en peso las K toneladas del buque.

- **Una función de selección que indica en cualquier instante cuál es el candidato más prometedor de los no usados todavía**: El contenedor de menor peso de los que aún no están cargados, de ahí que los ordenemos de menor a mayor peso.
- **La función objetivo que intentamos optimizar**: El número de contenedores a cargar, es lo que queremos maximizar.

A continuación pasamos a mostrar el código de nuestro algoritmo. Para representar el conjunto de contenedores, hemos considerado un vector de enteros, siendo cada entero el peso de cada contenedor. Para asegurarnos de que la suma de los pesos de todos los contenedores supera a las K toneladas del buque, el vector es de dimensión K con enteros aleatorios desde 0 hasta K (en nuestro código lo representamos por el parámetro n, introducido por el usuario). Nuestra función devuelve el número de contenedores que se han podido cargar:

```
int contenedoresGreedy1(int *T, int n){

        int used = 0;
        int result = 0;
        vector<int> myvector(T,T+n);
        sort(myvector.begin(),myvector.end());

        for(int i = 0; (i < n) && (used <= n); i++){
                used += T[i];
                result++;
        }

        return result;
}
```

- **Estudio de la optimalidad**

  Vamos a probar la optimalidad de nuestro algoritmo mediante una demostración matemática. Sea $T = \{c_1, ..., c_n\}$ y llamemos $S = \{c_1, ..., c_m\}$ a la solución de nuestro algoritmo Greedy. Vamos a fijar notación para lo que sigue:

  - $|S| \rightarrow$ Cardinal del conjunto de contenedores S
  - $c_i \rightarrow$ Contenedor i
  - $p_i \rightarrow$ Peso del contenedor i

  Consideramos que $|S| = m$. Por cómo hemos diseñado el algoritmo, sabemos que los pesos de los contenedores de la solución cumplen que:

$$\sum_{c_i \in S} p_i = \sum_{i=1}^{m} p_i \leq K \text{ y } \sum_{i=1}^{m+1} p_i > K$$

  También, dado un conjunto Q cualquiera no vacío que no está contenido en S, deducimos de lo anterior que:

$$\sum_{c_i \in S} p_i + \sum_{c_i \in Q} p_i > K$$

  Para ver que nuestra solución es óptima, vamos a ver que dado cualquier subconjunto U de T con un cardinal mayor que S, es decir, con más contenedores que S ($|U| = m' > m = |S|$), nunca podrá ser solución del problema. Para ver que no es solución del problema, hay que probar que $\sum_{c_i \in U} p_i > K$.

  Comenzaremos descomponiendo el conjunto de contenedores U de una forma que nos interesa. Sabemos que $U = (S \cap U) \cup (U \setminus S)$. Por ello, podemos escribir la siguiente igualdad:

$$\sum_{c_i \in U} p_i = \sum_{c_i \in S \cap U} p_i + \sum_{c_i \in U \setminus S} p_i$$

  Debido a esto, $|U| = m' = |S \cap U| + |U \setminus S|$. También se tiene que $(S \cap U) \cup (S \setminus U)$, luego $|S| = m = |S \cap U| + |S \setminus U|$. Teniendo en cuenta las dos igualdades en términos de cardinales que acabamos de ver, se tiene que $|U \setminus S| = m' - |S \cap U| > m - |S \cap U| = |S \setminus U|$. Esto nos demuestra que en $|U \setminus S|$ hay más elementos que en $|S \setminus U|$.

A continuación, consideramos un subconjunto R de $U \setminus S$ cuyo cardinal coincida con el de $S \setminus U$ (esto lo podemos hacer porque hemos demostrado antes que $|U \setminus S| > |S \setminus U|$), de forma que $U \setminus S \setminus R \neq \emptyset$. Esto nos permite seguir desarrollando la igualdad anterior:

$$\sum_{c_i \in S \cap U} p_i + \sum_{c_i \in U \setminus S} p_i = \sum_{c_i \in S \cap U} p_i + \sum_{c_i \in R} p_i + \sum_{c_i \in U \setminus S \setminus R} p_i$$

Ahora, hemos de tener en cuenta que todos los contenedores de $U \setminus S$ son de mayor peso que los de $S \setminus U$, y como R es subconjunto de $U \setminus S$, se cumple concretamente para los contenedores de R. Esto nos permite afirmar que la siguiente desigualdad es cierta (recordar que esto también se debe a que R y $S \setminus U$ tienen el mismo cardinal):

$$\sum_{c_i \in S \cap U} p_i + \sum_{c_i \in R} p_i + \sum_{c_i \in U \setminus S \setminus R} p_i \geq \sum_{c_i \in S \cap U} p_i + \sum_{c_i \in S \setminus U} p_i + \sum_{c_i \in U \setminus S \setminus R} p_i$$

Y el miembro de la derecha lo podemos reescribir como:

$$\sum_{c_i \in S \cap U} p_i + \sum_{c_i \in S \setminus U} p_i + \sum_{c_i \in U \setminus S \setminus R} p_i = \sum_{c_i \in S} p_i + \sum_{c_i \in U \setminus S \setminus R} p_i$$

Y por ser S la solución de nuestro algoritmo Greedy, sabemos que $\sum_{c_i \in S} p_i + \sum_{c_i \in U \setminus S \setminus R} p_i > K$, y si seguimos la secuencia de igualdades y la desigualdad final que hemos ido exponiendo, uniendo los extremos, llegamos a que $\sum_{c_i \in U} p_i > K$, que era lo que queríamos demostrar. Por tanto, S es la solución óptima del problema.

2.1.2. Segundo ejercicio

En este segundo ejercicio, lo que queremos es maximizar las toneladas cargadas en el buque sin sobrepasar su capacidad total. Para ello, seguimos el pensamiento inverso al planteado en el anterior ejercicio. Como lo que nos interesa es cargar el máximo de toneladas posibles, empezaremos cargando aquellos contenedores cuyo peso sea el más grande. Para ello, los ordenamos de mayor a menor peso, justo al contrario que antes. Identificamos las 6 características de un problema Greedy:

- **Un conjunto de candidatos**: En este caso, los contenedores a cargar.
- **Una lista de candidatos ya usados**: Los contenedores que ya han sido cargados.
- **Un criterio que dice cuándo un conjunto de candidatos forma una solución**: El criterio es que la suma de los pesos de un conjunto de contenedores no sea superior a las K toneladas del buque.
- **Un criterio que dice cuándo un conjunto de candidatos es factible (podrá llegar a ser una solución)**: el conjunto de contenedores que se evalúe no debe superar en peso las K toneladas del buque.
- **Una función de selección que indica en cualquier instante cuál es el candidato más prometedor de los no usados todavía**: El contenedor de mayor peso de los que aún no están cargados, de ahí que los ordenemos de mayor a menor peso.
- **La función objetivo que intentamos optimizar**: El número de toneladas a cargar, es lo que queremos maximizar.

En el código del algoritmo, al igual que antes, simulamos los contenedores con sus respectivos pesos con un vector de enteros, el cual ordenamos de mayor a menor usando el sort de la STL. Se van sumando los pesos hasta que se sobrepase el tope de toneladas del buque. Nuestra función devuelve el número de toneladas que se han podido cargar en total:

```
int contenedoresGreedy2(int *T, int n){

        int used = 0;
        vector<int> myvector(T,T+n);
        sort(myvector.begin(),myvector.end(), greater<int>());

        for(int i = 0; (i < n) && (used <= n); i++){
                used += T[i];
        }

        return used;
}
```

- **Estudio de la optimalidad**

  Sin embargo, en este caso nuestro algoritmo no nos da la solución óptima. Veámoslo con un contra-ejemplo. Con n = 10, imaginemos que tenemos el vector [5, 4, 6, 1, 1, 2, 7, 9, 8, 3]. Nuestro algoritmo lo ordenaría de mayor a menor, obteniendo el vector [9, 8, 7, 6, 5, 4, 3, 2, 1, 1]. Tras esto, se incluiría el contenedor de peso 9, pero el siguiente ya no sería posible cargarlo ya que $9 + 8 = 17 > 10$. La solución óptima en este caso sería tomar un contenedor de peso 1, otro de 2, otro de 3 y otro de 4, aprovechando así las 10 toneladas en su totalidad.

2.2. Ejercicio 2. El problema del viajante de comercio

El enunciado del problema es el siguiente: *dado un conjunto de ciudades y una matriz con las distancias entre todas ellas, un viajante debe recorrer todas las ciudades exactamente una vez, regresando al punto de partida, de forma tal que la distancia recorrida sea mínima.*

Además, se nos pide enfocarlo usando dos heurísticas distintas:

- **Vecino más cercano**: dada una ciudad inicial v_0, se agrega como ciudad siguiente aquella v_i (no incluida en el circuito) que se encuentre más cercana a v_0. El procedimiento se repite hasta que todas las ciudades se hayan visitado.
- **Inserción**: la idea es comenzar con un recorrido parcial, que incluya algunas de las ciudades, y luego extender este recorrido insertando las ciudades restantes mediante algún criterio de tipo greedy.

Además, se debe proponer por parte del equipo otra heurística propia para resolver el problema.

Elementos comunes

A lo largo de la solución del problema usaremos la siguiente notación para todas las heurísticas a desarrollar:

- n es el **número de ciudades**.
- D es la **matriz de distancias**.
- r es el **vector de recorrido**, que contiene unra ruta que pasa por todas las ciudades, es decir, n elementos no repetidos.
- W_r es el **coste** de un recorrido, es decir, la distancia de un recorrido r.

Adicionalmente, se han implementado una serie de funciones comunes a todas las heurísticas trabajadas en esta práctica:

```
#Creating a struct point
@dataclass
class Point:
    x: float
    y:float

def parse_input(input):
    points = []

    with open(input, "r") as archive:
        archive.readline()

        for line in archive.readlines():
            line = "␣".join(line.split())
            line = line.strip().split('␣')

            p = Point(float(line[1]), float(line[2]))

            points.append(p)
    return points

def distance(p1,p2): #O(1)
    if p1 != p2: #O(1)
        result = sqrt(pow((p1.x - p2.x),2) + pow((p1.y - p2.y),2)) #O(1)
        result = round(result.real) #O(1)
    else:
        result = 0 #O(1)
    return result
```

```
def gen_distance_matrix(points, num_cities): #O(n^2)
    matrix = [[0 for x in range(num_cities)] for x in range(num_cities)] #O(n^2)

    for i in range(num_cities): #O(n)                                    |
        for j in range(num_cities): #O(n)                                | O(n^2)
            matrix[i][j] = distance(points[i], points[j]) #O(1)          |

    return matrix


def get_road_distance(road): #O(n)
    road_distance = 0 #O(1)

    for i in range(len(road)-1): #O(n)
        road_distance += distance(road[i], road[i+1]) #O(1)

    #First and last point
    road_distance += distance(road[0], road[len(road)-1]) #O(1)
    return road_distance

def file_reordered(points):
    for point in points:
        print(str(point.x) + " " + str(point.y))

def get_swap(road, pos1, pos2): #O(1)
    changed = road

    tmp = road[pos1]
    road[pos1] = road[pos2]
    road[pos2] = tmp

    return changed
```

Hemos definido una "estructura" `Point` para representar los puntos, una función `parse_input` para parsear el input según el formato dado en los archivos de clase, `distance` que mide la distancia entre dos puntos, `gen_distance_matrix` que genera la matriz de distancias dada una lista de puntos y `get_road_distance` que da la longitud de un vector de recorrido.

Cabe destacar que los datos que se van a mostrar son de los *datasets* dados por la profesora, pero se han trabajados con conjuntos de datos más grandes. En estas experiencias con datos más grandes hemos observado la importancia de jugar con más de un tipo distinto de enfoque *greedy* pues en algunos casos el tiempo de ejecución puede ser demasiado alto.

2.2.1. Heurística del vecino más cercano

Este algortimo *greedy* es muy simple:

1. Partimos de un nodo cualquiera (en nuestro caso siempre usaremos el primer elemento de nuestro *input*).
2. Encontramos el nodo más cercano a este nodo, y los añadimos al recorrido.
3. Repetimos el proceso hasta cubrir todos los nodos.

Hacemos uso del siguiente código:

```
def  get_min_row_element(matrix, position): #O(n)
    min_pos = 0 #O(1)
    starting_pos = 0 #O(1)

    while matrix[position][starting_pos] == 0 and starting_pos < len(matrix)-1: #O(n)
        starting_pos += 1

    #Security check
    if starting_pos == len(matrix)-1: #O(1)
        return -1

    min_pos = starting_pos #O(1)

    min_val = math.inf #O(1)

    for j in range(starting_pos, len(matrix[position])): #O(n)
        if matrix[position][j] < min_val and position != j and matrix[position][j] != -1: #O(1)
            min_pos = j
            min_val = matrix[position][j]
```

```
    return min_pos

def clean_position(matrix, pos): #O(n)
    for col in range(len(matrix)): #O(n)
        matrix[pos][col] = -1

    for row in range(len(matrix)): #O(n)
        matrix[row][pos] = -1

def get_best_solution_nna(points): #O(n^2)
    road = []
    order = []

    distance_matrix = gen_distance_matrix(points, len(points)) #O(n^2)

    #We start always at first point
    last_point = 0 #O(1)
    road.append(points[last_point]) #O(1)
    order.append(0) #O(1)

    while len(road) < len(points):              #                                    |
        best_position = get_min_row_element(distance_matrix, last_point) #O(n)  |
                                                                            #   |
        road.append(points[best_position]) #O(1)                            #   |
        order.append(best_position) #O(1)                                   #   |  O(n^2)
                                                                            #   |
        clean_position(distance_matrix, last_point) #O(n)                       |
                                                                            #   |
        last_point = best_position #O(1)                                        |

    road_distance = get_road_distance(road) #O(1)

    return road, road_distance, order


file = sys.argv[1]

points = parse_input(file)

recorrido, distancia, orden = get_best_solution_nna(points)

str_orden = [str(n) for n in orden]

print("El mejor orden a seguir: [" + ','.join(str_orden) + "]")
print("Su distancia es: " + str(distancia))
```

Análisis teórico

En este programa, básicamente tenemos lo que hacemos es partir de el primer punto pasado en el *input* y tras ello vamos buscando los nodos más cercanos sin repetir los ya añadidos. Primero parseamos el input, generamos la matriz de distancia, y la vamos transformando para descartar los nodos ya considerados. Si observamos los comentarios que tenemos en el código, vemos que la función que resuelve nuestro problema es $\mathcal{O}(n^2$, es decir:

$$T(n) \in \mathcal{O}(n^2)$$

Esto se debe a que las funciones `get_min_row_element` y `clean_position`, que son $\mathcal{O}(n)$ se encuentran dentro de un bucle `while` que es $\mathcal{O}(n)$.

Análisis empírico

A continuación, mostramos los resultados obtenidos tras probar nuestro programa con los *datasets* proporcionados en la asignatura:

- `ulysses16.tsp`: El mejor orden para este *dataset* (teniendo en cuenta el orden del fichero original) es:

$$[0, 7, 15, 12, 11, 13, 6, 5, 14, 4, 8, 9, 3, 1, 2, 10]$$

  y la distancia a recorrer es: 103

  Si representamos los puntos con el recorrido generado obtenemos el siguiente gráfico:

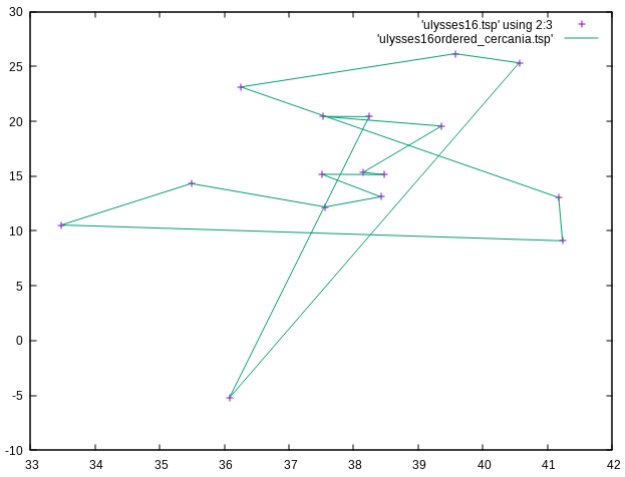


Figura 1: Gráfica de Ulysses mediante el Vecino más cercano

- `bayg29.tsp`: El mejor orden para este *dataset* (teniendo en cuenta el orden del fichero original) es:

$$[0, 27, 5, 11, 8, 4, 20, 1, 19, 9, 3, 14, 17, 13, 21, 16, 10, 18, 24, 6, 22, 26, 7, 23, 15, 12, 28, 25, 2]$$

  y la distancia a recorrer es: 10209

  Si representamos los puntos con el recorrido generado obtenemos el siguiente gráfico:

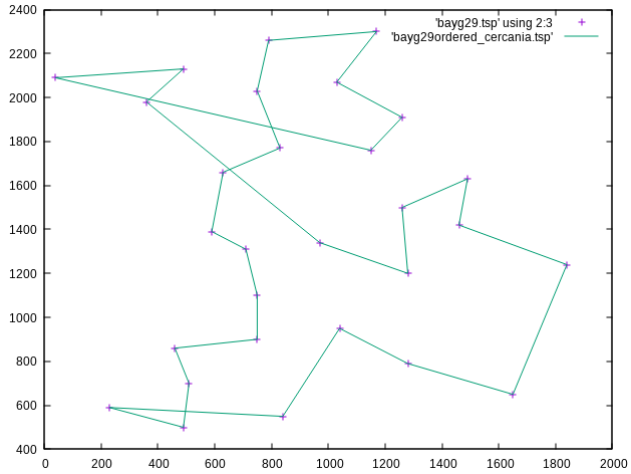


Figura 2: Gráfica de Bayg mediante el Vecino más cercano

- `eil76.tsp`: El mejor orden para este *dataset* (teniendo en cuenta el orden del fichero original) es:

$$[0, 72, 32, 62, 15, 2, 43, 31, 8, 38, 71, 57, 9, 37, 64, 10, 65, 52, 13, 18, 34, 6, 7, 45, 33, 51, 26, 44, 28,$$
$$47, 46, 20, 73, 27, 61, 1, 29, 3, 74, 75, 66, 25, 11, 39, 15, 50, 5, 67, 4, 36, 19, 69, 59, 70, 35, 68, 60, 2$$
$$1, 41, 40, 42, 22, 55, 48, 23, 17, 49, 24, 54, 30, 58, 53, 12, 56, 14, 63]$$

  y la distancia a recorrer es: 642

  Si representamos los puntos con el recorrido generado obtenemos el siguiente gráfico:

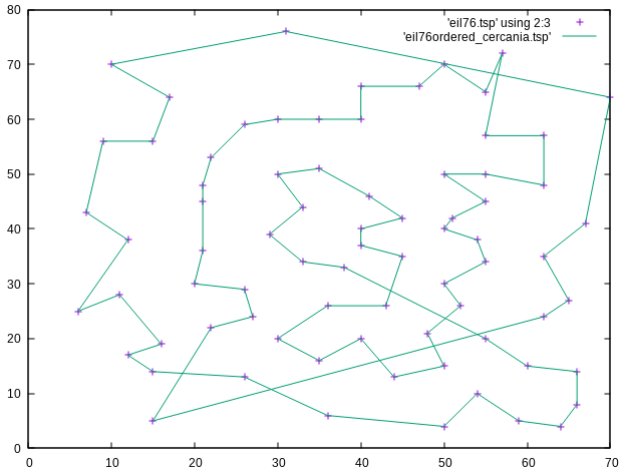


Figura 3: Gráfica de Eil mediante el Vecino más cercano

Además, hemos obtenido más datos de problemas TSP de la página `TSPLIB` y hemos realizado un estudio con más números de ciudades y hemos medido el tiempo de ejecución de nuestro algoritmo, obteniendo los siguientes datos:

Heurística del Vecino más cercano	
Ciudades (n)	**Tiempo (s)**
358	0.134928381001373
537	0.301542887000323
716	0.571918544999789
895	0.886006714001269
1074	1.24296611099999
1253	1.64954555899931
1432	2.64052301800075
1611	2.85592838900084
1790	3.59707787300067
1969	4.38948754500052
2148	5.39098084799844
2327	6.98254896399885
2506	7.10732670200014
2685	8.94702127299934
2874	9.27537115900122
3053	9.69826381100029
3232	11.1923151139999
3411	12.4046790310003
3590	12.853421451
3769	14.3906136509995
3948	15.625681644
4127	17.1084850459993
4306	18.5073586279996
4461	19.8675596370013

Cuadro 1: Experiencia empírica de el vecino más cercano

Análisis híbrido

Después, hemos representado con `gnuplot` los datos de tiempos obtenidos en el apartado anterior y les hemos realizado un ajuste cuadrático (dato obtenido en el análisis teórico), obteniendo la gráfica que se muestra a continuación. El análisis hbirido nos permitirá comprobar que nuestro análisis teórico era correcto.

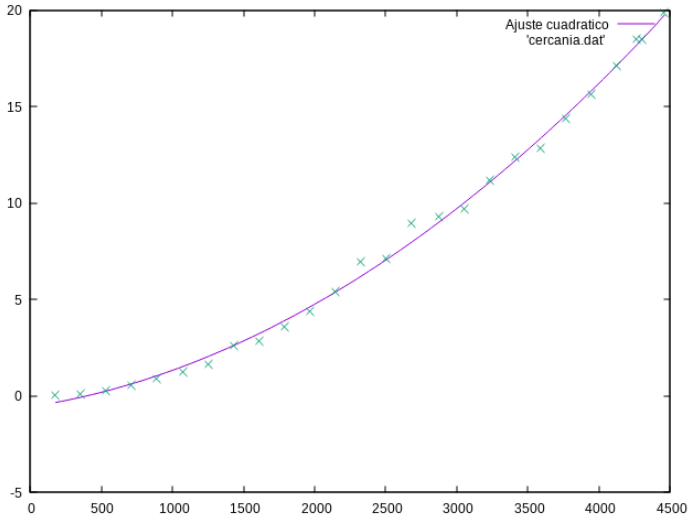


Figura 4: Gráfica con los tiempos de ejecución del vecino más cercano

Y las constantes ocultas son: $T(n) = 7{,}6769 \cdot 10^{-7}n^2 + 0{,}00112191n - 0{,}56077$

Para terminar nuesto análisis, terminaremos de confirmar que el ajuste cuadrático es el óptimo viendo el coeficiente de determinación que nos ha proporcionado gnuplot:

Coef.determination = 0.869828

Vemos que es un muy cercano a 1, por tanto es correcto nuestro análisis. Hay que tener en cuenta que los datos de los que partimos no son aleatorios y por tanto puede generar una pequeña perturbación en los tiempos.

2.2.2. Heurística por inserción

Tenemos de nuevo el mismo problema que antes del viajante del comercio pero con una heurística distinta. Veamos las 6 características de nuestro problema Greedy:

- **Un conjunto de candidatos**: En este caso, las ciudades por las que pasar.
- **Una lista de candidatos ya usados**: Los ciudades por las que ya se ha pasado. Cabe resaltar que se debe comenzar por un recorrido inicial de 3 ciudades previamente. Para escoger estas ciudades simplemente lo haremos escogiendo la ciudad más al oeste, la más al este y la más al norte.
- **Un criterio que dice cuándo un conjunto de candidatos forma una solución**: El criterio es que el recorrido que se haga forme un circuito pasando por todas las ciudades una sola vez.
- **Un criterio que dice cuándo un conjunto de candidatos es factible (podrá llegar a ser una solución)**: En caso de que no se repita ningún nodo (ciudad), el conjunto de candidatos es factible.
- **Una función de selección que indica en cualquier instante cuál es el candidato más prometedor de los no usados todavía**: Utilizaremos un criterio que denominaremos inserción mas económica: de entre todas las ciudades no visitadas, elegimos aquella que provoque el menor incremento en la longitud total del circuito. En otras palabras, cada ciudad debemos insertarla en cada una de las soluciones posibles y quedarnos con la ciudad (y posición) que nos permita obtener un circuito de menor longitud. Seleccionaremos aquella ciudad que nos proporcione el mínimo de los mínimos calculados para cada una de las ciudades
- **La función objetivo que intentamos optimizar**: El coste del recorrido total del circuito.

Análisis teórico

Visualizaremos el pseudocódigo del programa para hablar de la eficiencia:

```
ALGORITMO INSERCION TSP (G=(V,R))
C = V [C = Lista de Candidatos]
S = Vacio [S = Conjunto Solución]
Crear a partir de R recorridoInicial
S << {este, oeste, norte}
C = C\{este, oeste, norte}
```

```
Fin de Crear
Repetir hasta que cardinal(C)= Vacio
        Crear vector pCandidatos
        Para cada s en S
                Para cada c en C
                        Calcular distancia(s,c)
                        Seleccionar punto cercano de C al punto "s"
                Fin-Para
        Almacenar (c,distancia) >> puntosCandidatos
        Fin de Para
        Seleccionar p en puntosCandidatos que produzca menor incremento
        Insertar p en T
        C = C\{p}
Fin de Repetir
```

Como se puede apreciar en el pseudocódigo, el programa recorre tres bucles distintos donde llamando n al número de ciudades, sabemos que a pesar de reduciendo el tamaño con el que se recorren los bucles, por lo visto en teoría la función acabará teniendo eficiencia $\mathcal{O}(n^3)$. Como aclaración la función distancia utilizada dentro del tercer bucle es $\mathcal{O}(1)$ y por eso no sube el orden de eficiencia.

Análisis empírico

Mostraremos ahora los resultados obtenidos tras probar nuestro programa con los *datasets* proporcionados en la asignatura:

- `ulysses16.tsp`: El mejor orden para este *dataset* (teniendo en cuenta el orden del fichero original) es:

$$[5, 6, 7, 10, 12, 14, 15, 13, 16, 1, 4, 8, 9, 11, 2, 3]$$

  y la distancia a recorrer es: 99

  Si representamos los puntos con el recorrido generado obtenemos el siguiente gráfico:

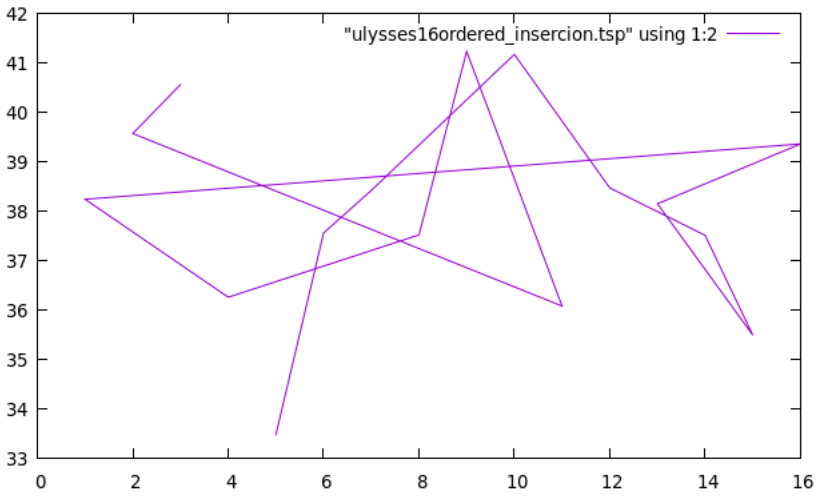


Figura 5: Gráfica de Ulysses mediante insercion

- `bayg29.tsp`: El mejor orden para este *dataset* (teniendo en cuenta el orden del fichero original) es:

  $$[3, 23, 12, 6, 5, 26, 29, 21, 2, 20, 10, 13, 4, 15, 19, 25, 7, 18, 14, 22, 11, 17, 9, 28, 1, 24, 27, 16, 8]$$

  y su distancia es 10048.

  Si representamos los puntos con el recorrido generado obtenemos el siguiente gráfico:

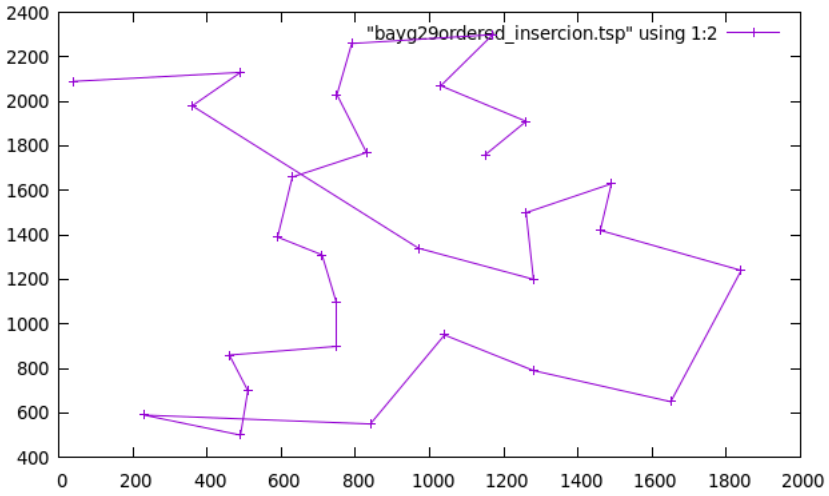


Figura 6: Gráfica de Bayg mediante insercion

- `eil76.tsp`: El mejor orden para este *dataset* (teniendo en cuenta el orden del fichero original) es:

  $$[56, 23, 63, 33, 73, 62, 22, 28, 61, 74, 30, 48, 5, 15, 57, 37, 20, 70, 60, 71, 47, 36, 69, 21, 29, 45, 27, 13, 54, 52, 34, 67, 26, 76, 75, 4, 6$$

  y su distancia es 611.

  Si representamos los puntos con el recorrido generado obtenemos el siguiente gráfico:

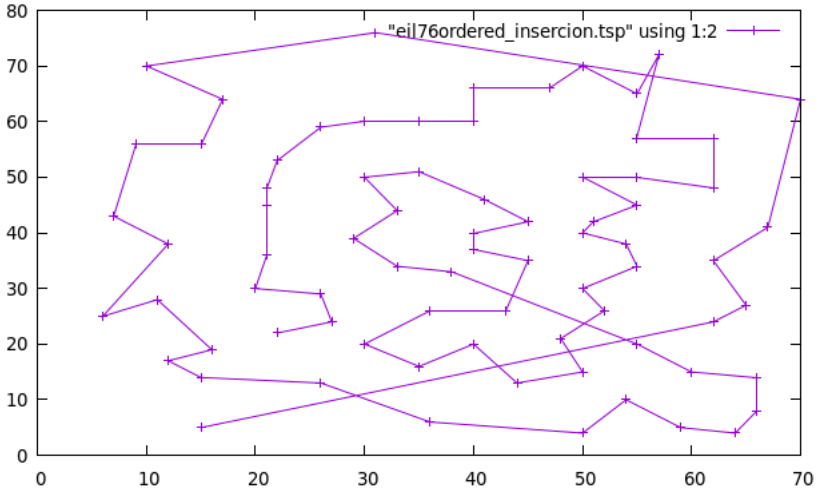


Figura 7: Gráfica de Eil mediante insercion

Además, hemos obtenido más datos de problemas TSP de la página `TSPLIB` y hemos realizado un estudio con más números de ciudades y hemos medido el tiempo de ejecución de nuestro algoritmo, obteniendo los siguientes datos:

Análisis híbrido

Después, hemos representado con `gnuplot` los datos de tiempos obtenidos en el apartado anterior y les hemos realizado un ajuste cuadrático (dato obtenido en el análisis teórico), obteniendo la gráfica que se muestra a continuación. El análisis hbirido nos permitirá comprobar que nuestro análisis teórico era correcto.

Heurística de inserción	
Ciudades (n)	**Tiempo (s)**
358	0.276428
537	0.925309
716	2.23608
895	4.21995
1074	7.40272
1253	0.0587605
1432	0.275992
1611	0.943963
1790	2.39547
1969	4.42937
2148	7.34342
2327	11.8085
2506	17.2049
2685	24.4085
2874	33.8853
3053	44.5835
3232	57.4245
3411	73.2457
3590	92.1468
3769	113.815
3948	139.366
4127	165.838
4306	200.112
4461	235.333

Cuadro 2: Experiencia empírica de inserción

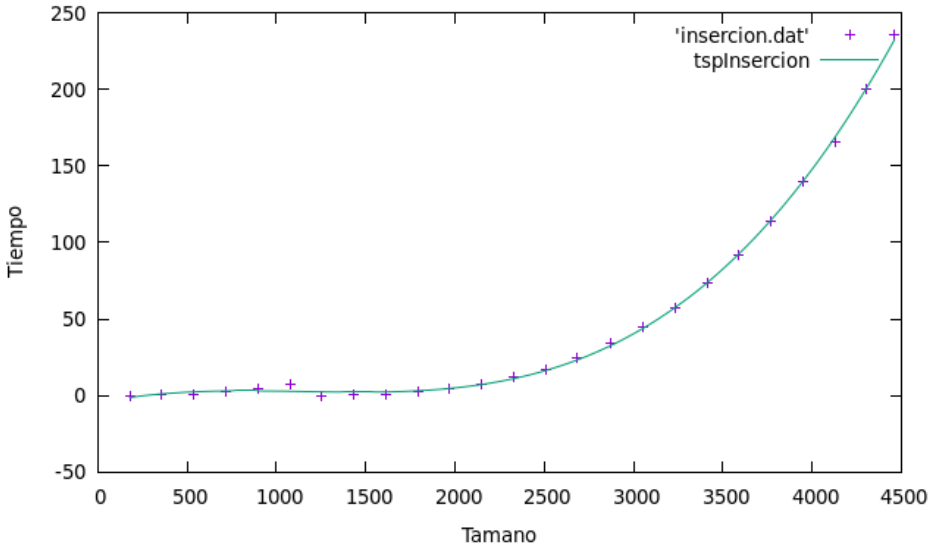


Figura 8: Gráfica con los tiempos de ejecución de insercion

Y las constantes ocultas son: $T(n) = 6{,}46703 \cdot 10^{-9}n^3 - 2{,}21761 \cdot 10^{-5}n^2 + 0{,}02336n - 4{,}75186$

Para terminar nuesto análisis, terminaremos de confirmar que el ajuste cúbico es el óptimo viendo el coeficiente de determinación que nos ha proporcionado gnuplot:

Coef.determination = 0.85647

Vemos que es un muy cercano a 1, por tanto es correcto nuestro análisis. Hay que tener en cuenta que los datos de los que partimos no son aleatorios y por tanto puede generar una pequeña perturbación en los tiempos.

2.2.3. Heurística propia: perturbaciones

Este enfoque, de nuevo *greedy*, realiza las perturbaciones indicadas por un parámetro sobre un recorrido dado para intentar mejorarlo. Tenemos el siguiente código:

```
def get_worst_node(road): #O(n)
    worst_distance = distance(road[0], road[1]) #O(1)
    worst_pos = 0 #O(1)

    for i in range(len(road)-1): #O(n)
        if distance(road[i],road[i+1]) > worst_distance: #O(1)
            worst_distance = distance(road[i],road[i+1]) #O(1)
            worst_pos = i

    return worst_pos

#Perturbate the road from a pos
```

```
def perturbate(road, orden, pos): #O(n^2)

    best_perturbation = pos #O(1)
    base_distance = get_road_distance(road) #O(n)

    #Calcualte all perturbations
    for i in range(len(road)): #O(n)                                    |
        current_perb = copy.deepcopy(road) #O(n)                        |
        current_perb = get_swap(current_perb, pos, i) #O(1)             | #O(n^2)
        swap_distance = get_road_distance(current_perb) #O(n)           |

        if swap_distance < base_distance: #O(1)                         |
            best_perturbation = i #O(1)                                 |
            base_distance = swap_distance #O(1)                         |

    #Make the change with the best perturbation
    road = get_swap(road, pos, best_perturbation) #O(1)
    orden = get_swap(orden, pos, best_perturbation) #O(1)



def get_best_solution_perturbations(points, orden, perturbations):
    base_road = points #O(1)

    for i in range(perturbations): #O(perturbations)        |
        pos = get_worst_node(points) #O(n)                  |
                                                    #       | O(n^2*perturbations)
        perturbate(base_road,orden, pos) #O(n^2)            |

    return base_road, get_road_distance(base_road), orden

file = sys.argv[1] #"./Problem2/Datasets/ulysses16.tsp"
perturbations =  int(sys.argv[2])

points = parse_input(file)

start = time.perf_counter()

base_road, distance_nna, orden = get_best_solution_nna(points)

final_road, final_distance, final_orden = get_best_solution_perturbations(base_road, orden, perturbations)

end = time.perf_counter()


correct = len(set(final_orden)) == len(final_orden) #Checks if there is any duplicate

'''if correct:
    print("Correct order!")
else:
    print("Incorrect order")'''

#print(final_orden)
#print("Su distancia es: " + str(final_distance))

# file_reordered(final_road)

time_total = end-start

print(str(len(points)) + " " + str(time_total))
```

Análisis teórico

Primero lo que hacemos es obtener una solución mediante la heurística del vecino más cercano. Tras esto, tenemos la función `get_worst_node` que es $\mathcal{O}(n)$, que dado un recorrido, haya el nodo que más dista de su nodo siguiente. Por ello es necesario usar ya una solución previa (mediante el vecino más cercano) para calcular este "nodo peor". Por último, tenemos la función `perturbate` que es $\mathcal{O}(n^2)$, que prueba todas las posibles permutaciones que podemos hacer desde el respectivo "peor nodo" y se queda con la mejor de ellas. Por todo ello, apoyándonos en los comentarios realizados en el código mostrado, podemos concluir que:

$$T(n) \in \mathcal{O}(n^2 \cdot perturbaciones) \Rightarrow T(n) \in \mathcal{O}(n^2)$$

Análisis empírico

A continuación, mostramos los resultados obtenidos tras probar nuestro programa con los *datasets* proporcionados en la asignatura:

- `ulysses16.tsp`: Aplicando 10 perturbaciones, obtenemos que el mejor orden (teniendo en cuenta el orden del fichero original):

  $$[0, 7, 15, 12, 11, 13, 6, 5, 14, 4, 8, 9, 2, 1, 3, 10]$$

  y su distancia es 101.

  Si representamos los puntos con el recorrido generado para 10 perturbaciones obtenemos el siguiente gráfico:

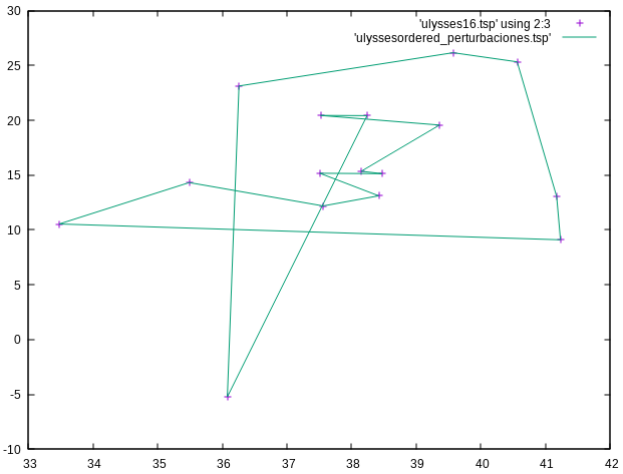


Figura 9: Gráfica de Ulysses mediante perturbaciones

- `bayg29.tsp`: Aplicando 10 perturbaciones, obtenemos que el mejor orden (teniendo en cuenta el orden del fichero original):

$$[0, 27, 5, 11, 8, 4, 20, 1, 19, 9, 3, 14, 17, 13, 21, 16, 10, 18, 24, 6, 22, 26, 7, 23, 15, 12, 28, 25, 2]$$

  y su distancia es 10209.

  Si representamos los puntos con el recorrido generado para 10 perturbaciones obtenemos el siguiente gráfico:

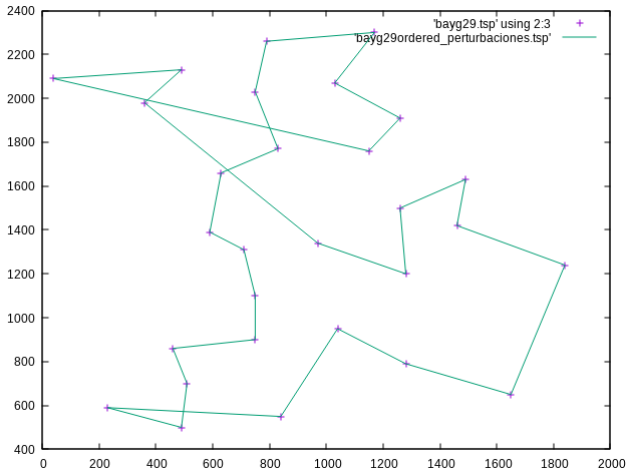


Figura 10: Gráfica de Bayg mediante perturbaciones

- `eil76.tsp`: Aplicando 10 perturbaciones, obtenemos que el mejor orden (teniendo en cuenta el orden del fichero original):

$$[0, 72, 32, 62, 15, 2, 43, 31, 8, 38, 71, 57, 9, 37, 64, 10, 65, 52, 13, 18, 34, 6, 7, 45, 33, 51, 26, 44, 28,$$

$$47, 46, 20, 73, 27, 61, 1, 29, 3, 74, 75, 66, 25, 11, 39, 16, 50, 5, 67, 4, 36, 19, 69, 59, 70, 35, 68, 60, 2,$$

$$1, 41, 40, 42, 22, 55, 48, 23, 17, 49, 24, 54, 30, 58, 53, 12, 56, 14, 63]$$

  y su distancia es 642.

  Si representamos los puntos con el recorrido generado para 10 perturbaciones obtenemos el siguiente gráfico:

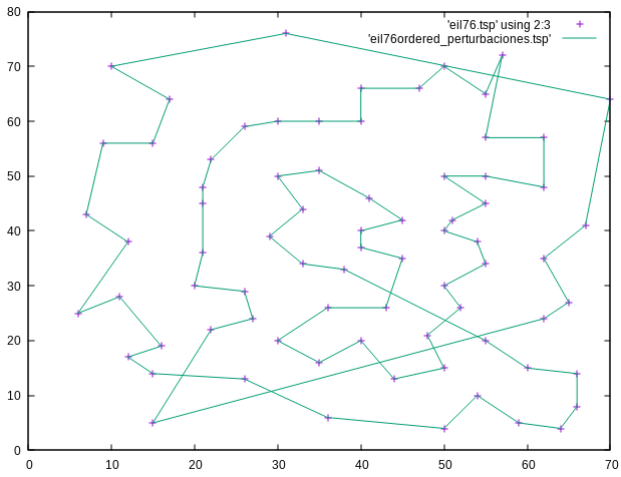


Figura 11: Gráfica de Eil mediante perturbaciones

Además. hemos obtenido más de problemas TSP de la página `TSPLIB` y hemos realizado un estudio con más números de ciudades y hemos medido el tiempo de ejecución de nuestro algoritmo, obteniendo los siguientes datos:

Heurística de perturbaciones	
Ciudades (n)	**Tiempo (s)**
358	7.09205518299859
537	14.8038363970009
716	27.0365697790003
895	42.5021277929991
1074	60.6207957549996
1253	79.9574952149997
1432	106.705971191001
1611	133.111995577001
1790	163.618691645999
1969	198.794982856998
2148	241.148463359001
2327	275.541576210999
2506	323.217905321999
2685	371.290206549998
2874	429.912281959001
3053	485.322751331001
3232	545.820733338998
3411	603.358232573999
3590	678.706273265001
3769	751.554062298001
3948	825.585648235003
4127	890.451960293001
4306	973.566447267
4461	1045.84781561849

Cuadro 3: Experiencia empírica de perturbaciones

Análisis híbrido

Después, hemos representado con `gnuplot` los datos de tiempo obtenidos en el apartado anterior y les hemos realizado un ajuste cuadrático (dato obtenido en el análisis teórico), obteniendo la gráfica que se muestra a continuación. El análisis híbrido nos permitirá comprobar que nuestro análisis teórico era correcto.

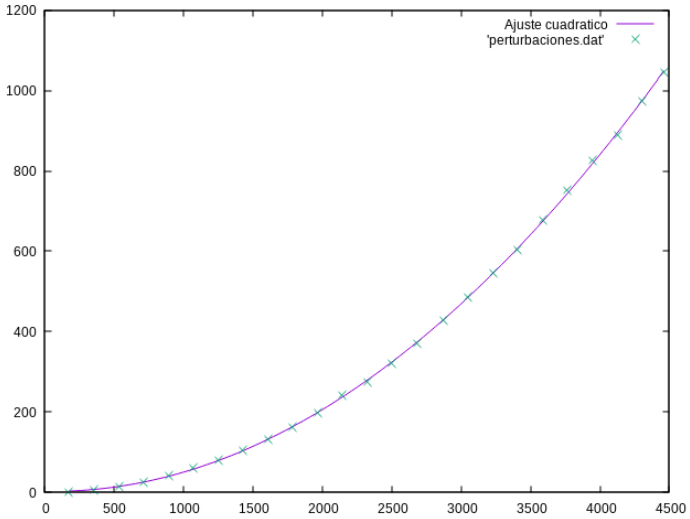


Figura 12: Gráfica con los tiempos de ejecución de perturbaciones

Y las constantes ocultas son $T(n) = 5{,}39602 \cdot 10^{-5} \cdot n^2 - 0{,}00659642n + 3{,}2468$.

Para terminar nuestro análisis, terminaremos de confirmar que el ajuste cuadrático es el óptimo viendo el coeficiente de determinación que nos ha proporcionado gnuplot:

Coef.determination = 0.93507

Vemos que es muy cercano a 1, por tanto nuestro análisis es correcto.

2.2.4. Comparación de heurísticas

Los tres algoritmos que tenemos son: cercanía (el vecino más cercano), inserción y por perturbaciones. Veremos la gráfica comparativa de los tiempos de cada heurística.

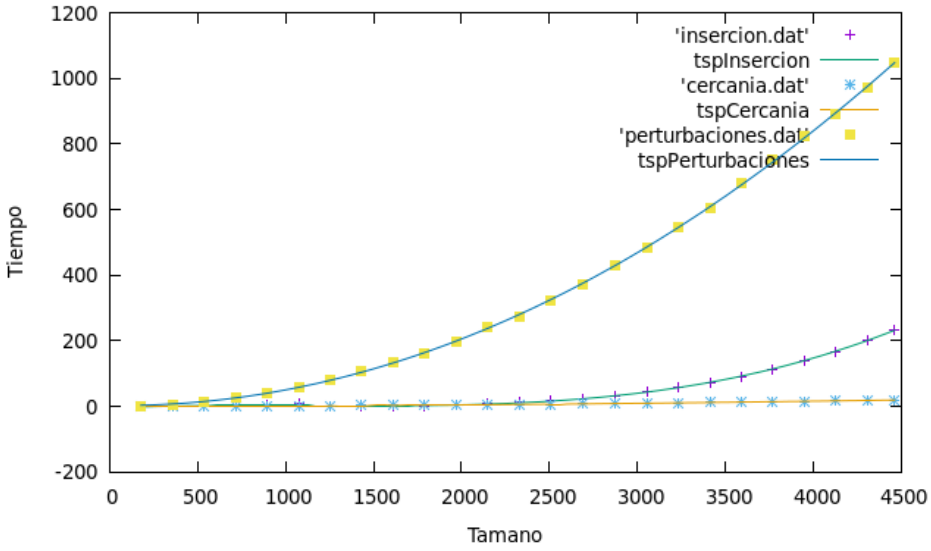


Figura 13: Gráfica comparativa con los tiempos de ejecución de las distintas heurísticas

Heurística de cercania	
Ciudades (n)	**Distancia ()**
358	22268
537	28232
716	38482
895	45794
1074	54590
1253	64034
1432	72686
1611	81277
1790	92277
1969	97931
2148	106301
2327	118281
2506	127627
2685	132398
2874	149824
3053	154575
3232	167010
3411	173936
3590	181619
3769	193185
3948	199550
4127	208015
4306	222743
4461	229963

Heurística de insercion	
Ciudades (n)	**Distancia ()**
358	22081
537	27956
716	37891
895	45087
1074	53629
1253	63147
1432	71017
1611	80478
1790	91078
1969	97106
2148	105102
2327	108140
2506	124987
2685	130156
2874	147103
3053	152685
3232	165432
3411	170914
3590	179974
3769	190034
3948	196996
4127	205108
4306	219841
4461	226683

Heurística de perturbaciones	
Ciudades (n)	**Distancia ()**
358	22247
537	28232
716	38409
895	45789
1074	54287
1253	63952
1432	72662
1611	81077
1790	92277
1969	97913
2148	106280
2327	118281
2506	126984
2685	132239
2874	149530
3053	154265
3232	167010
3411	173936
3590	181411
3769	192804
3948	199525
4127	207947
4306	222743
4461	229960

Cuadro 4: Experiencia empírica de las distintas heurísticas

Vemos como los tiempos de ejecución son mayores para inserción debido a que su eficiencia es cúbica respecto a los demás, donde también se observa que la heurística de perturbaciones tiene un tiempo algo mayor que la de cercanía. Sin embargo este mayor tiempo se puede justificar a la hora de escoger estas heurísticas debido a que el algoritmo que proporciona mejores soluciones es, como se puede comprobar con los costes previamente reflejados, el que utiliza la heurística de inserción y posteriormente al que utiliza la heurística de perturbaciones. Por tanto la conclusión en este caso es que a mayor tiempo de ejecución mejor solución obtenemos y por tanto debemos de tener esto en cuenta a la hora de elegir una heurística u otra.

3. Conclusiones

Con esta práctica, hemos aprendido a crear algoritmos voraces para resolver problemas que, en su versión obvia o de fuerza bruta tienen una complejidad en ocasiones demasiado elevadas.

Además, hemos comprobado como dentro de esta misma técnica podemos atacar los problemas de manera distinta. Debemos de prestar atención a la dificultad de obtener la solución óptima, y **primar el "acercarnos"** a ella mediante algoritmos que van mejorando la solución anterior y **no presentan una eficiencia baja**. De hecho, hemos visto que el algoritmo de inserción para el TSP es el que daba la **solución más óptima** con respecto a los otros dos, **pero es de eficiencia cúbica**, y **para tamaños muy grandes del problema no nos serviría**. Es por esto, que debemos hallar un **término medio entre eficiencia y optimalidad**.

Esto es algo de especial relevancia a la hora de trabajar con cantidades ingentes de datos, en el que dar una solución inicial puede ser muy complicado, pudiendo partir incluso de soluciones arbitrarias, mejorándolas conforme nuestros algoritmos se van ejecutando.