

Doble Grado en Ingeniería Informática y Matemáticas

Algorítmica

Práctica 3 - Algoritmos Greedy

David Kessler Martínez

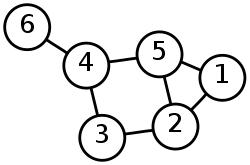
[dkesslerm03@correo.ugr.es](mailto:dkesslerm03@correo.ugr.es)

Santiago López Cerro

[santiloce@correo.ugr.es](mailto:santiloce@correo.ugr.es)

Antonio Javier Rodríguez Romero

[antoniojrr@correo.ugr.es](mailto:antoniojrr@correo.ugr.es)



Granada, curso 2022/2023

Índice:

[**0. Introducción 3**](#_lr9m15zfspxa)

[**0.1. Objetivos 3**](#_lr9m15zfspxa)

[**0.2. Autores y Hardware 3**](#_s5snfqakokyn)

[0.2.1. Autores 3](#_xk1xpcg6ebty)

[0.2.2. Entorno de análisis 3](#_mjk3082qaow3)

[Santiago López Cerro 3](#_t4s09tmx7yko)

[David Kessler Martínez 4](#_nqrwkvdjevuo)

[Antonio Rodríguez Romero 4](#_3hb0b4ygmxff)

[0.2.3. Obtención de tiempos 4](#_ofxykke5mnwh)

**1**[**. P5: Grafos parciales**](#_lr9m15zfspxa) 6

[**1.1. Estudio del problema 6**](#_fknvbqk8n5u8)

[**1.2. Implementación del algoritmo 6**](#_un7eynm7mpk)

[**1.3. Análisis de eficiencia. 8**](#_x18mm0qxwvts)

[1.3.1. Teórico. 8](#_vakpe7k9jmk2)

[1.3.2. Empírico. 8](#_2gjuf4iw894x)

[1.3.3. Híbrido. 9](#_761ggdj2rqt)

[1.3.4. Demostración teórica 10](#_uue94k4c9a2e)

**2**[**. P6: Viajante de Comercio**](#_lr9m15zfspxa) 11

[**2.1. Estudio del problema 11**](#_ez1k0a6nxpkk)

[**2.2. Primera Heurística: Inserción Lejana 12**](#_klg6mnkhdtxg)

[2.2.1. Análisis de eficiencia 13](#_5nwtvuuv56pm)

[2.2.1.1 Análisis teórico 13](#_f0dgdjxr3wi)

[2.2.1.2 Análisis empírico 13](#_lg1xi5d4hd77)

[2.1.2.3 Análisis híbrido 14](#_wlw1ncdqrs15)

[**2.3. Segunda Heurística: Vecino más cercano 15**](#_ptka82t44qyy)

[2.3.1. Análisis de eficiencia 16](#_q5hyarb3p5v6)

[2.3.1.1 Análisis teórico 16](#_c223jkp0q06w)

[2.3.1.2. Análisis empírico 16](#_3tiqmv2jc89f)

[2.3.1.3. Análisis híbrido 17](#_hfdjgcrm9k5j)

[**2.4. Tercera Heurística: Inserción barata 18**](#_eszgy7gp85zw)

[2.4.1. Análisis de eficiencia 18](#_gy3laqhm5d4a)

[2.4.1.1. Estudio teórico 18](#_fjw1xsub2qbl)

[2.4.1.2. Análisis empírico 19](#_s9pdqka6w1id)

[2.4.1.3. Análisis híbrido 20](#_4n7aivvjbs3e)

[**2.5 Comparativa heurísticas 21**](#_q3zic4wel6o9)

0. Introducción

# 0.1. Objetivos

Con esta práctica se pretende comprender y asimilar el funcionamiento de la técnica de diseño de algoritmos voraces. Para ello tendremos que abordar la solución de dos problemas, “grafos parciales con aristas negativas” y “el viajante de comercio (PVC)”, abordando el diseño de heurísticas, algoritmos aproximativos, basados en la filosofía de los métodos voraces.

# 0.2. Autores y Hardware

### 0.2.1. Autores

David Kessler Martínez 35%

Antonio Javier Rodríguez Romero 35%

Santiago López Cerro 30%

Respecto al reparto del trabajo, Santiago se ha encargado del estudio profundo de los problemas ,Antonio se ha dedicado principalmente en el desarrollo del Algoritmo Voraz, realizando así su descripción, su análisis y su justificación, y David fue quien se encargó de demostrar que era el mejor caso posible. Refiriéndonos ahora al caso del Problema del Viajante de Comercio, David desarrolló las diversas heurísticas y fue Santiago quien se encargó de sus análisis, la justificación de su validez y la comparativa final. Respecto al generador de casos del PVC este fue desarrollado por Antonio. Finalmente quisiera recalcar que a pesar de ser este el reparto del trabajo los tres hemos contribuido a las diversas partes, ya sea proponiendo ideas para el código, comentarios para las explicaciones y justificaciones, desarrollo de la memoria, inserción de imágenes, etc.

### 0.2.2. Entorno de análisis

Estos los distinto entornos de cada uno de los compañeros en donde se ha realizado el estudio de los algoritmos:

#### Santiago López Cerro

* Dispositivo: HP Pavilion x360 Convertible 14-dw1xxx
* Procesador: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz
* RAM instalada: 8,00 GB (7,65 GB usable)
* Arquitectura: Sistema operativo de 64 bits, procesador basado en x64
* CPU(s): 4
* Hilo de procesamiento por núcleo: 1
* Socket(s): 1
* Núcleos por socket: 4

Especificaciones de Windows:

* Sistema Operativo: Windows 10 Home

Acerca de mi maquina virtual:

* Memoria de base: 2048 MB
* Procesadores: 4
* Orden de arranque: Disquete, Óptica, Disco Dura

#### 

#### David Kessler Martínez

Dispositivo: Lenovo IdeaPad 5 15ALC05

Procesador: AMD Ryzen 5 5500U with Radeon Graphics

RAM: 16 GB

Arquitectura: x86\_64

Sistema operativo: Ubuntu 22

CPU(s):12

Hilo de procesamiento por núcleo: 2

Socket(s): 1

Núcleos por socket: 6

#### Antonio Rodríguez Romero

Dispositivo: ASUS TUF Dash F15

Procesador: 11th Gen Intel(R) Core(TM) i7-11370H @ 3.30 GHz

RAM: 16GB

Arquitectura: x86\_64

Sistema operativo: Ubunto 22

CPU(s): 8

Hilo de procesamiento por núcleo: 2

Socket(s): 1

Núcleos por socket: 4

En cuanto a la compilación, hemos utilizado el compilador g++ con nivel de optimización 1.

### 

### 0.2.3. Obtención de tiempos

En lo relativo al método de medición de tiempo hemos decidido optar por el uso de la biblioteca ctime, haciendo uso de la función clock() antes y después de la ejecución del método que estudiemos, de manera que encontremos el tiempo de ejecución de este al restar al instante de después de esta el tiempo antes. Además, para mayor estabilidad y mejor representación, el algoritmo se ejecutará un total de 5 veces de forma que el resultado será la media de estos tiempos.

1. Grafos parciales

# Estudio del problema

Se nos pide que, a partir de un grafo conexo y ponderado que introducimos, obtengamos un grafo parcial de este, conexo también, cuya suma de los pesos de sus aristas sea mínima.

Aunque parezca en un primer vistazo simple, tendremos que tener en cuenta diversas cosas.

Comenzamos considerando la forma que tendremos de representar cada grafo dentro del programa, de manera que podamos acceder a cada arista de la forma más simple posible. Para ello, la mejor solución que hemos encontrado es una matriz triangular inferior en la que guardemos en la posición (i,j) el peso de la arista que une el nodo i y el j. Con el objetivo de hacer el tratamiento de estos lo más simple posible, desarrollaremos una clase *Grafo* que se encargue de representarlo.

Por otro lado, utilizaremos un peso muy grande para aquellas aristas que no existan en el grafo, de forma que estas nunca se tomen y pueda reconocerse que no existen. Este valor en la implementación será *PESOMAX*.

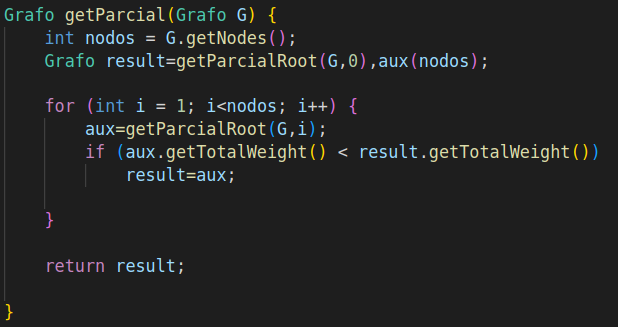
Por último, tenemos que tener en cuenta también que pueden existir aristas negativas, de forma que utilizarlas aunque sus dos nodos ya estén conectados al grafo implicaría un peso menor, por lo que resulta conveniente utilizarlas.

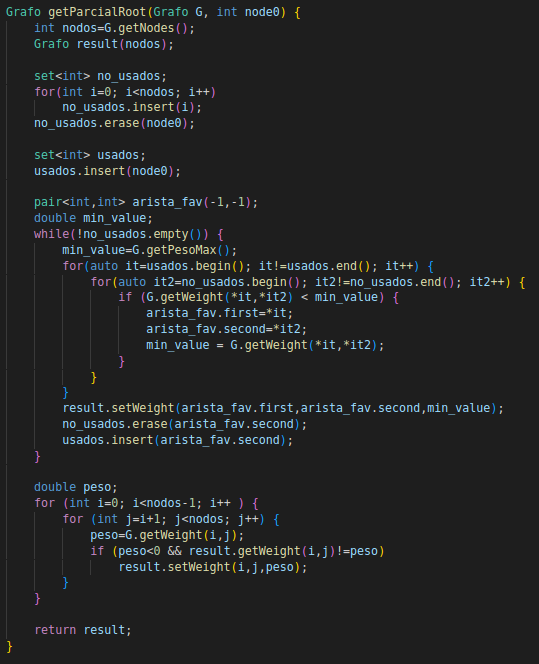
# Implementación del algoritmo

Para facilitar la legibilidad del código se ha creado la clase *Grafo,* de manera que se entienda bien los diferentes accesos que se hacen en cada momento a este objeto.

Los atributos con los que contará serán el número de nodos (*int nodos*) y la matriz que guardará los pesos de las aristas ( *double\*\* M*). Algunos de los métodos más destacables son *setWeight(int i, int j, double peso)*, para establecer los pesos a la hora de la lectura del grafo, *getWeight(int i, int j)* para acceder al peso de una arista en concreto, *getTotalWeight()*, te devuelve la suma de los pesos de todas las aristas del grafo y *getNodes()* para acceder al número de nodos.

El algoritmo voraz implementado será:



Hace uso de distintas llamadas a una función auxiliar *getParcialRoot(Grafo G, int i)*.

Este irá guardando los nodos que no ha incluido en el grafo parcial todavía y los que sí, de forma que una primera parte del algoritmo tome las aristas con menor peso entre un nodo que ya se haya incluido en el grafo parcial y otro que no. Una vez hecho esto, obtendremos un grafo parcial conexo, pero debido a la existencia de aristas con pesos negativos tendremos que incluir otra fase en la cual se compruebe si no hemos tomado alguna de estas, para incluirla también de forma que disminuya el peso.

Se trata de un algoritmo *Greedy* ya que después de la inclusión de cada arista, se comprobará en el momento la “solución óptima”, es decir, la arista restante entre un nodo ya incluido y otro que no lo esté con menor peso. Este proceso se repetirá tomando cada vez un nodo como nodo inicial, y de estas soluciones obtenidas se elegirá la que menos peso consiga.

Cabe destacar que este algoritmo se basa principalmente en el algoritmo de Prim, visto en las clases de teoría, con las modificaciones correspondientes para que tenga en cuenta la existencia de aristas negativas.

# Análisis de eficiencia.

## Teórico.

Si observamos la implementación del algoritmo, la función *getParcial(Grafo G)* nos damos cuenta de que pasa la mayoría del tiempo de ejecución dentro del bucle *for*, donde, a su vez, el tiempo lo ocupará casi enteramente la ejecución de la función auxiliar *getParcialRoot(Grafo G, int i*).

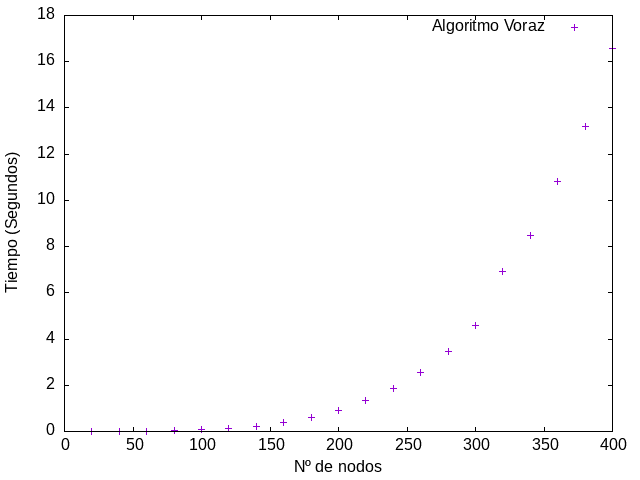
Estudiando este último, nos centraremos en el bucle *while*, y dentro de él, los dos bucles *for* anidados. Nos damos cuenta entonces de que uno de ellos iterará por los nodos no usados y el otro por los usados, de forma que en total llevarán a cabo **n iteraciones** (Tomando n como el número de nodos del grafo de entrada). A su vez, esto se ejecutará hasta que el *set<int> no\_usados* esté vacío. De nuevo, fácilmente deducimos que si por cada repetición del bucle interior se toma un nodo más en el grafo parcial, tardará **n iteraciones** en terminar.

De lo deducido anteriormente, podemos sacar que *getParcialRoot()* será de orden cuadrático y, como nuestra función estudiada llamará **n veces** a la mencionada, se tratará de un algoritmo de orden *O(n³)*.

## Empírico.

Como hemos mencionado en el apartado de toma de datos, los resultados obtenidos se han conseguido a través de la media de 5 tiempos reales, de forma que conseguimos una mejor representación de la eficiencia del algoritmo.

| **NODOS** | **TIEMPO(s)** |
| --- | --- |
| 20 | 0,0002306 |
| 40 | 0,0024838 |
| 60 | 0,0115626 |
| 80 | 0,0296524 |
| 100 | 0,0670820 |
| 120 | 0,1321060 |
| 140 | 0,2320430 |
| 160 | 0,4063730 |
| 180 | 0,6253080 |
| 200 | 0,9183490 |
| 220 | 1,3454300 |
| 240 | 1,8535700 |
| 260 | 2,5431500 |
| 280 | 3,4700000 |
| 300 | 4,5929100 |
| 320 | 6,9435500 |
| 340 | 8,4990000 |
| 360 | 10,8074000 |
| 380 | 13,2126000 |
| 400 | 16,5756000 |



## Híbrido.

Como hemos observado en el análisis teórico, se trata de un algoritmo de orden cúbico, por lo tanto realizaremos el ajuste con una función polinómica de orden 3:

A través del software *gnuplot*, deducimos por regresión mediante mínimos cuadrados que las constantes ocultas serán

**Final set of parameters Asymptotic Standard Error**

**======================= ==========================**

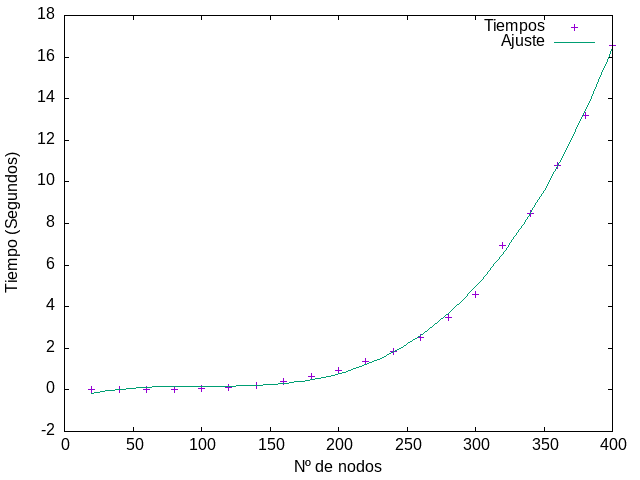
**a3 = 6.10829e-07 +/- 3.433e-08 (5.62%)**

**a2 = -0.000186255 +/- 2.19e-05 (11.76%)**

**a1 = 0.0191105 +/- 0.004009 (20.98%)**

**a0 = -0.483768 +/- 0.1993 (41.2%)**

Representando esta función gráficamente vemos que:



Como no existe gran dispersión entre los puntos y la función, deducimos que se trata de un buen ajuste, representando esta función bastante fielmente una estimación de los tiempos de ejecución en función de los nodos del grafo

## Demostración teórica

En este punto demostraremos de forma teórica que el algoritmo de Prim modificado es solución óptima.

Sea G (V,A) un grafo no dirigido y conexo. Sea U contenido en V un subconjunto de los nodos de de V, y sea e=(u,v) la arista de peso mínimo tal que u ɛ U y v ɛ V \ U. Sabemos que existe un árbol generador minimal T tal que e ɛ T. Supongamos pues que T es un árbol generador minimal que no contiene a e. Luego debe haber como mínimo una arista que una u y v, sea f la de menor peso. Si añadimos e a T’, se crea un ciclo, luego si eliminamos f, obtenemos un nuevo árbol generador mínimo T’ que tendrá una longitud menor o igual a T, ya que hemos eliminado f (una arista de peso mayor o igual a e) y la hemos sustituido por e.

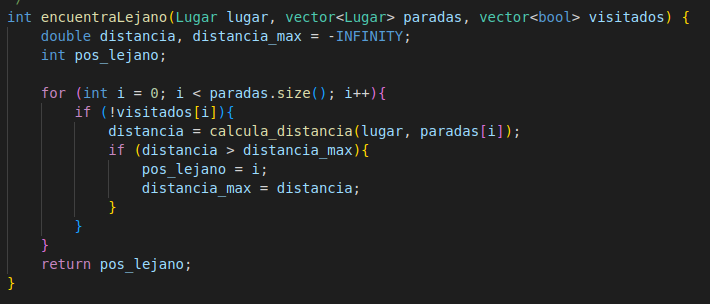
Esto es una contradicción, ya que T es un árbol generador mínimo, luego nuestro árbol T es el óptimo. La única modificación a tener en cuenta a la hora de implementar el algoritmo de Prim para aristas, negativas es que no se crean ciclos negativos, ya que no se alcanzaría una solución óptima, pero esta modificación no afecta a nuestra demostración.

1. El viajante de comercio

# Estudio del problema

Se nos plantea la siguiente situación, una empresa de componentes informáticos ha contratado a un nuevo comercial. Este conoce información relevante sobre una serie de clientes y su objetivo es visitarlos a todos, con el objetivo de hacer buenas ventas. Son diversas las restricciones que se presentan, siendo en primer lugar que no puede volver hasta haberlos visitados a todos, comenzando y finalizando en su empresa, en segundo lugar solo podrá visitar a cada cliente una vez y finalmente lo que se busca es que la distancia recorrida sea la menor posible. Una vez entendido el problema, serán tres los métodos que diseñaremos para la resolución de este.

# Primera Heurística: Inserción Lejana



La idea de esta heurística es seleccionar en cada momento la casa más alejada de todas las recorridas hasta el momento para, una vez obtenida, colocarla en el punto del recorrido óptimo, de forma que se minimice el recorrido hasta este punto.

## Análisis de eficiencia

## 2.2.1.1 Análisis teórico

Vemos como la mayor parte del tiempo de ejecución del programa se encontrará dentro del bucle *while* que, como por cada iteración se añadirá una parada nueva a la ruta, se repetirá **n veces** (Siendo n el número de casas a visitar). Por otro lado, dentro de este encontramos un bucle *for* que se ejecutará tantas veces como visitas se hayan agregado hasta el momento. Dentro de este, se realizan llamadas a la función *calcula\_lejano()*, que consta también de un bucle *for* de **n iteraciones**. Avanzando más en el algoritmo, encontramos dos bucles **for** más, sin embargo, estos no constan de otro bucle en su interior.

Resumiendo y con un sencillo cálculo nos daremos cuenta de que, a causa del primer bucle *for*, se tratará de un algoritmo de orden cúbico (***O(n³)***).

## 2.2.1.2 Análisis empírico

Estudiamos la eficiencia ejecutando el algoritmo pasándole valores desde 50 hasta 700 con saltos de 50, tomando 14 tiempos distintos. Así pues estos son los datos obtenidos:

***Paradas Tiempos***

*50 0.0032802*

*100 0.0189914*

*150 0.0586152*

*200 0.130835*

*250 0.279459*

*300 0.469099*

*350 0.725074*

*400 1.21668*

*450 2.18723*

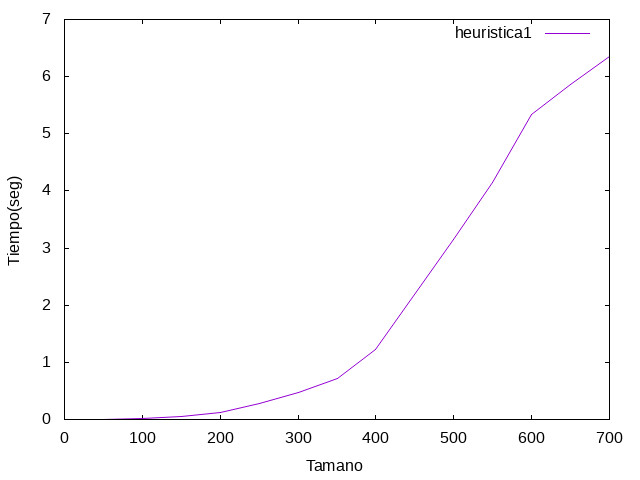
*500 3.15633*

*550 4.15388*

*600 5.34393*

*650 5.86641*

*700 6.34394*

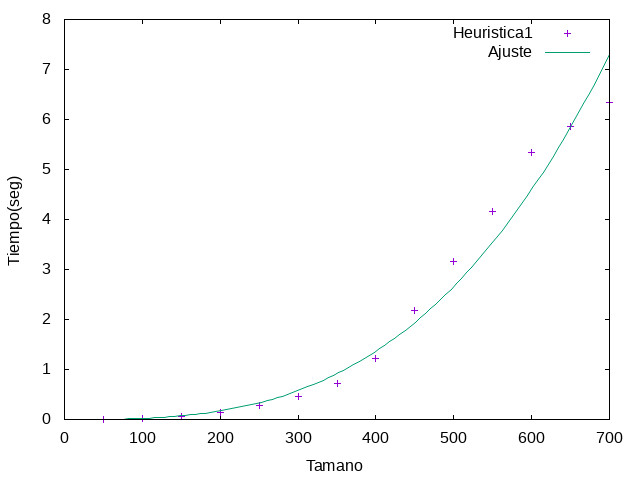


## 2.1.2.3 Análisis híbrido

Como hemos observado en el análisis teórico, se trata de un algoritmo de orden cúbico, por lo tanto realizaremos el ajuste con una función polinómica de orden 3:

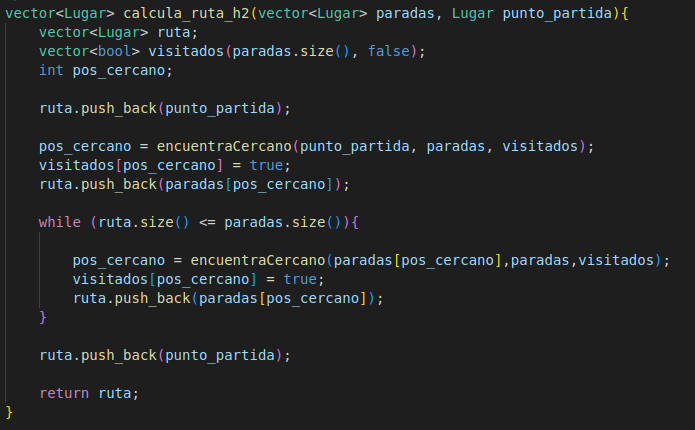
A través del software *gnuplot*, deducimos por regresión mediante mínimos cuadrados que las constantes ocultas serán:

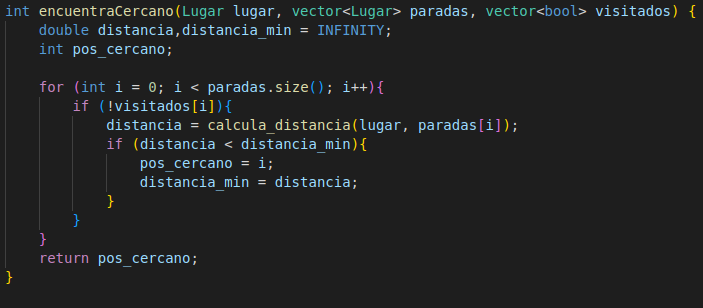
Sabemos que se trata de un buen ajuste porque la varianza toma un valor de -4.2027e-06.



# 

# Segunda Heurística: Vecino más cercano





Para este primer algoritmo determinaremos la ruta óptima mediante la heurística de encontrar la distancia mínima con el último punto visitado. Así pues buscaremos siempre acceder al cliente más cercano a nuestra posición siempre y cuando no lo hayamos visitado. Por tanto se trataría de un greedy ya que en cada paso nuestra óptima sería acceder a la distancia mínima llegando así a una solución general óptima.

**Ejemplo:**

*Supongamos el caso de Andalucía y que la empresa tuviera sede en Granada entonces el algoritmo nos indicaría ir a Almería por ser la provincia más cercana, posteriormente Jaén, así Córdoba, Sevilla, Huelva, Cádiz, Málaga y finalmente de nuevo Granada.*

## 2.3.1. Análisis de eficiencia

## 2.3.1.1 Análisis teórico

Para el estudio del problema consideraré el número de paradas n. Así pues, comenzaremos centrándonos en el bucle while. En este se ejecutarán **n iteraciones** y, por cada iteración, debido a la llamada al método *encuentraCercano()* se ejecutarán **n iteraciones** para poder buscar la próxima parada cercana entre todas las posibles, produciendo así una eficiencia de ***O(n²)***. Vemos así que por este lado nos quedaría una eficiencia cuadrática. Por otro lado, la llamada previa al bucle de *encuentraCercano()* para buscar la distancia más próxima por donde empezar, será O(n).

En conclusión, este algoritmo será de orden cuadrático (***O(n²)***).

## 2.3.1.2. Análisis empírico

Estudiamos la eficiencia ejecutando el algoritmo pasándole valores desde 50 hasta 700 con saltos de 50, tomando 14 tiempos distintos. Así pues estos son los datos obtenidos:

***Paradas Tiempos***

*50 0.0001298*

*100 0.0003902*

*150 0.0008488*

*200 0.0013848*

*250 0.0023368*

*300 0.0030902*

*350 0.0041384*

*400 0.0059134*

*450 0.007506*

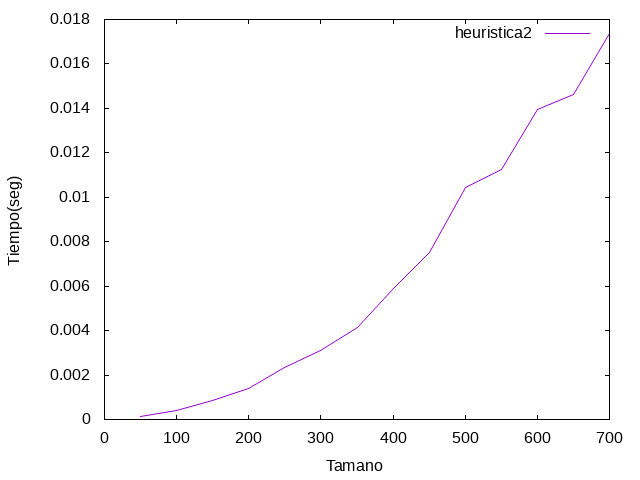
*500 0.0104362*

*550 0.011245*

*600 0.0139508*

*650 0.0146312*

*700 0.0173724*

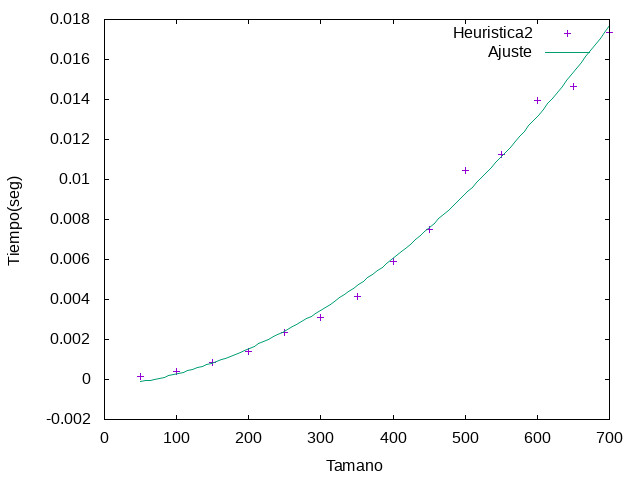


## 2.3.1.3. Análisis híbrido

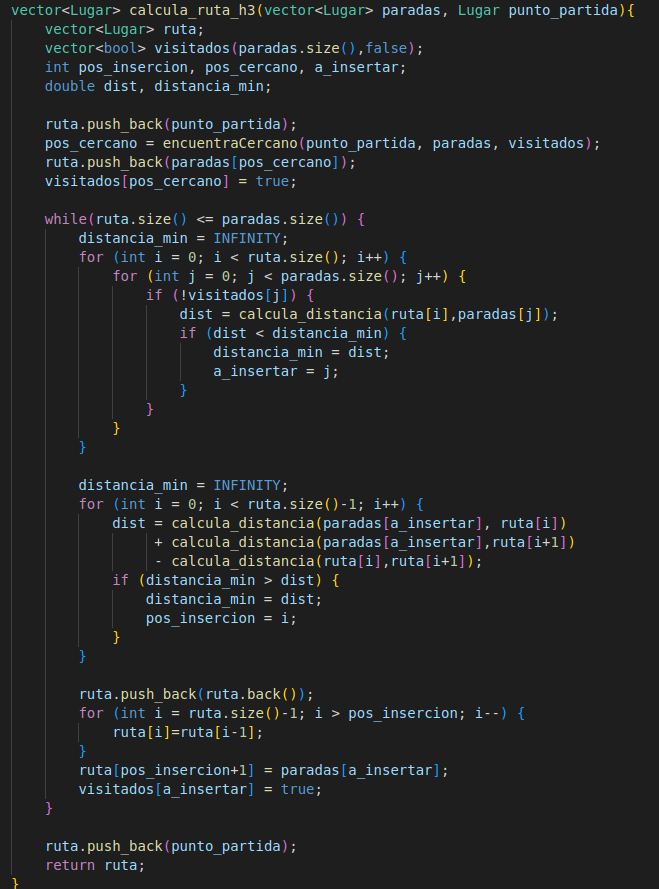
Como hemos observado en el análisis teórico, se trata de un algoritmo de orden cuadrático, por lo tanto realizaremos el ajuste con una función polinómica de orden 2:

A través del software *gnuplot*, deducimos por regresión mediante mínimos cuadrados que las constantes ocultas serán:

Como observamos, aunque algunos puntos se dispersan un poco más de la función de ajuste, encontramos que es una buena aproximación de los tiempos de ejecución que se obtienen en función del número de casas.



# 2.4. Tercera Heurística: Inserción barata



Para nuestra última heurística, utilizaremos el algoritmo de **Inserción barata**. Este se basa en buscar en cada momento la casa más cercana a cualquiera de las que ya se han visitado y, una vez obtenida, se insertará en el punto de la ruta que provoque un menor desplazamiento.

## 2.4.1. Análisis de eficiencia

## 2.4.1.1. Estudio teórico

Observando la implementación del método v*ector<Lugar> calcula\_ruta\_h3(vector<Lugar> paradas, Lugar punto\_partida)* observamos que la mayor tiempo de la ejecución de transcurre dentro del bucle *while*. Considerando el número de paradas como **n**, vemos que el bucle de primeras producirá **n iteraciones.** Dentro de este, encontramos dos bucles *for* anidados, ejecutándose el exterior tantas veces como paradas hayamos añadido a la ruta en el momento y el interior ejecutándose **n veces**. Estos dos bucles serán los que marcarán que este algoritmo sea de orden cúbico (***O(n³)***).

## 2.4.1.2. Análisis empírico

Estudiamos la eficiencia ejecutando el algoritmo pasándole valores desde 50 hasta 700 con saltos de 50, tomando 14 tiempos distintos. Así pues estos son los datos obtenidos:

***Paradas Tiempo***

*50 0.0035568*

*100 0.0275938*

*150 0.0904808*

*200 0.211649*

*250 0.416612*

*300 0.714386*

*350 1.12828*

*400 1.68742*

*450 2.38434*

*500 3.26217*

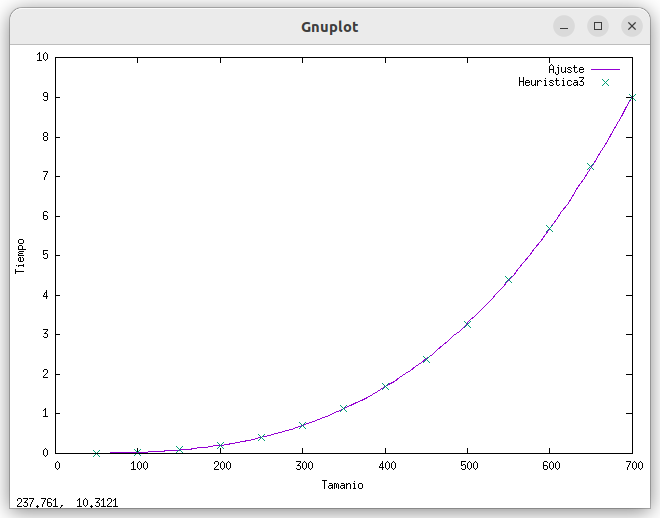
*550 4.38969*

*600 5.69124*

*650 7.25563*

*700 8.99292*

Representación gráfica de los resultados:

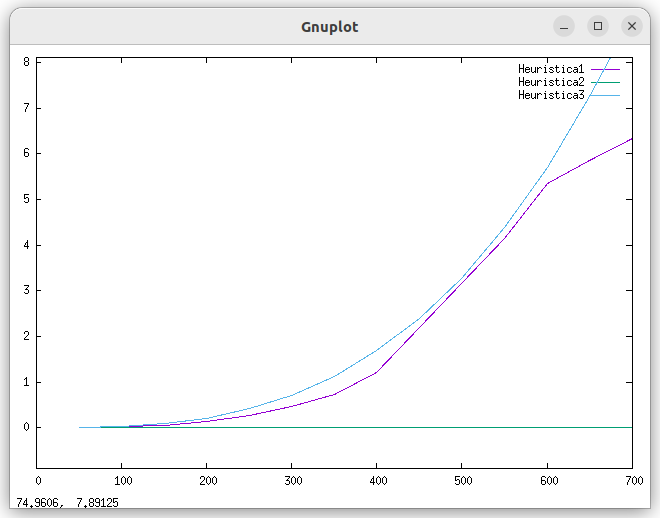


## 2.4.1.3. Análisis híbrido

Como hemos observado en el análisis teórico, se trata de un algoritmo de orden cuadrático, por lo tanto realizaremos el ajuste con una función polinómica de orden 2:

A través del software *gnuplot*, deducimos por regresión mediante mínimos cuadrados que las constantes ocultas serán:

# 2.5 Comparativa heurísticas



Observamos que la heurística 2 es la más rápida, lo que es lógico ya que se trata de un algoritmo cuadrático, mientras que los otros dos son cúbicos. Sin embargo, si tenemos en cuenta los resultados que producirán cada uno de estos, encontramos que los otros dos algoritmos toman rutas más óptimas que este en la mayoría de casos.

Esto se debe a una mayor complejidad en el diseño de estos aún, sin olvidar, el coste que ello provoca en los tiempos de ejecución.