



UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

Estudio del Profile

Grupo DA6:

José Carlos Gualo Cejudo

David Rivera Concepción

Álvaro López de Antón Bueno Parrado

Asignatura: Diseño de Algoritmos

Grupo de Titulación (20/21): 3º Computación

Titulación: Grado en Ingeniería Informática

Fecha: 30/04/2021

Contenido

Introducción3

Vista previa3

Instancias.....3

Grafo de llamadas.....4

Árbol de llamada.....5

Telemetría de carga de la CPU.....5

Hot Spots6

Telemetría de memoria.....7

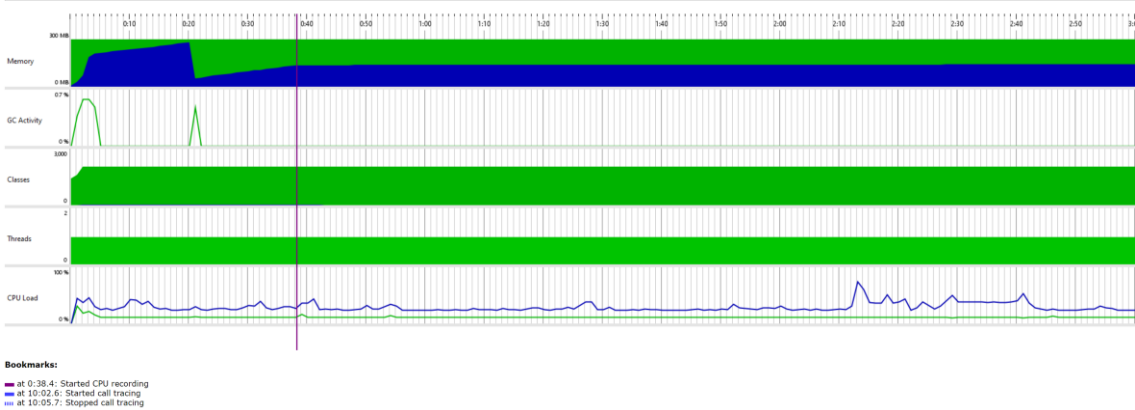
Outliers – Funciones más usadas.....7

Introducción

En este documento se va a detallar la documentación obtenida al ejecutar la herramienta de Profiling JProfile sobre nuestro código, y se tratará de entender lo que nos dice la herramienta sobre nuestro código para sacar conclusiones sobre su eficiencia.

Ahora vamos a ver los diferentes documentos que nos ha generado la herramienta, y a sacar conclusiones sobre ellos.

Vista previa



Aquí tenemos una visión general de lo que nos va a medir la herramienta. Es importante mencionar que temas las clases y los diferentes hilos de ejecución no los vamos a tratar ya que no tiene sentido hacerlo para lo que hemos hecho.

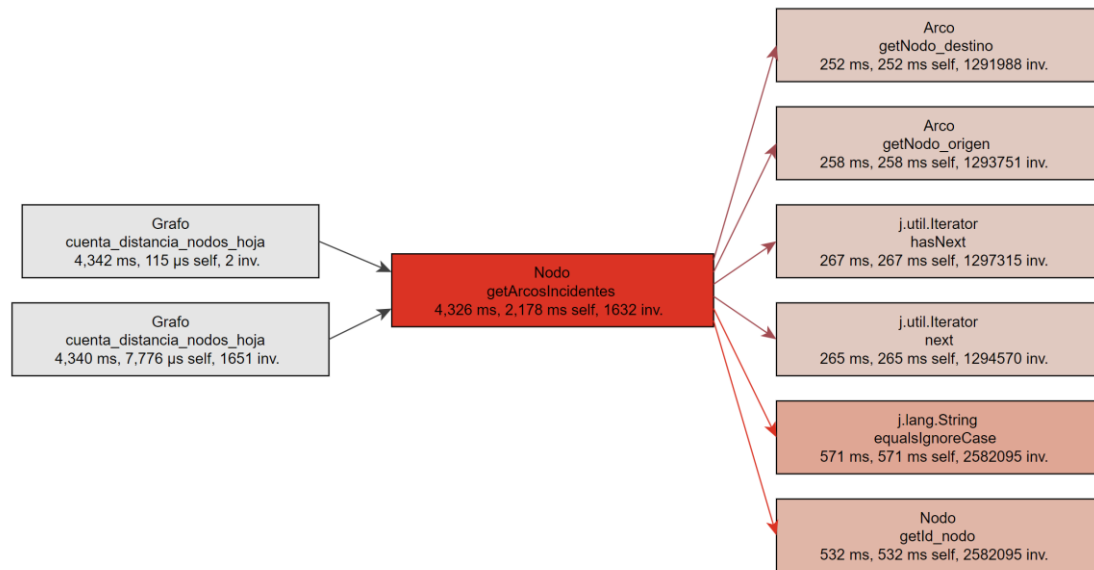
Por otro lado, podemos ver como la mayor consumición de memoria se realiza a la hora de cargar todos los archivos en el programa, y justo un instante después tendrá lugar la etapa de mayor actividad del recolector de basura. Y en cuanto a la carga del procesador, podemos decir que se mantiene constante casi todo el programa, excepto en las zonas en las que se requiere una mayor capacidad computacional.

Instancias

Name	Instance Count	Size
java.util.ArrayList	1,219,032	29,256 kB
java.lang.Object[]	842,193	65,791 kB
byte[]	113,103	3,848 kB
java.lang.String	111,411	2,673 kB
Arco	75,434	2,413 kB
Nodo	28,640	916 kB
java.lang.Object	11,432	182 kB
java.util.ArrayList\$Itr	7,254	232 kB
java.lang.management.MemoryUsage	5,616	269 kB
int[]	2,832	35,264 kB
java.util.concurrent.ConcurrentHashMap\$Node	2,405	76,960 bytes
java.lang.Class	2,325	290 kB
java.util.HashMap\$Node	2,116	67,712 bytes
long[]	712	125 kB
java.lang.Class[]	628	18,152 bytes

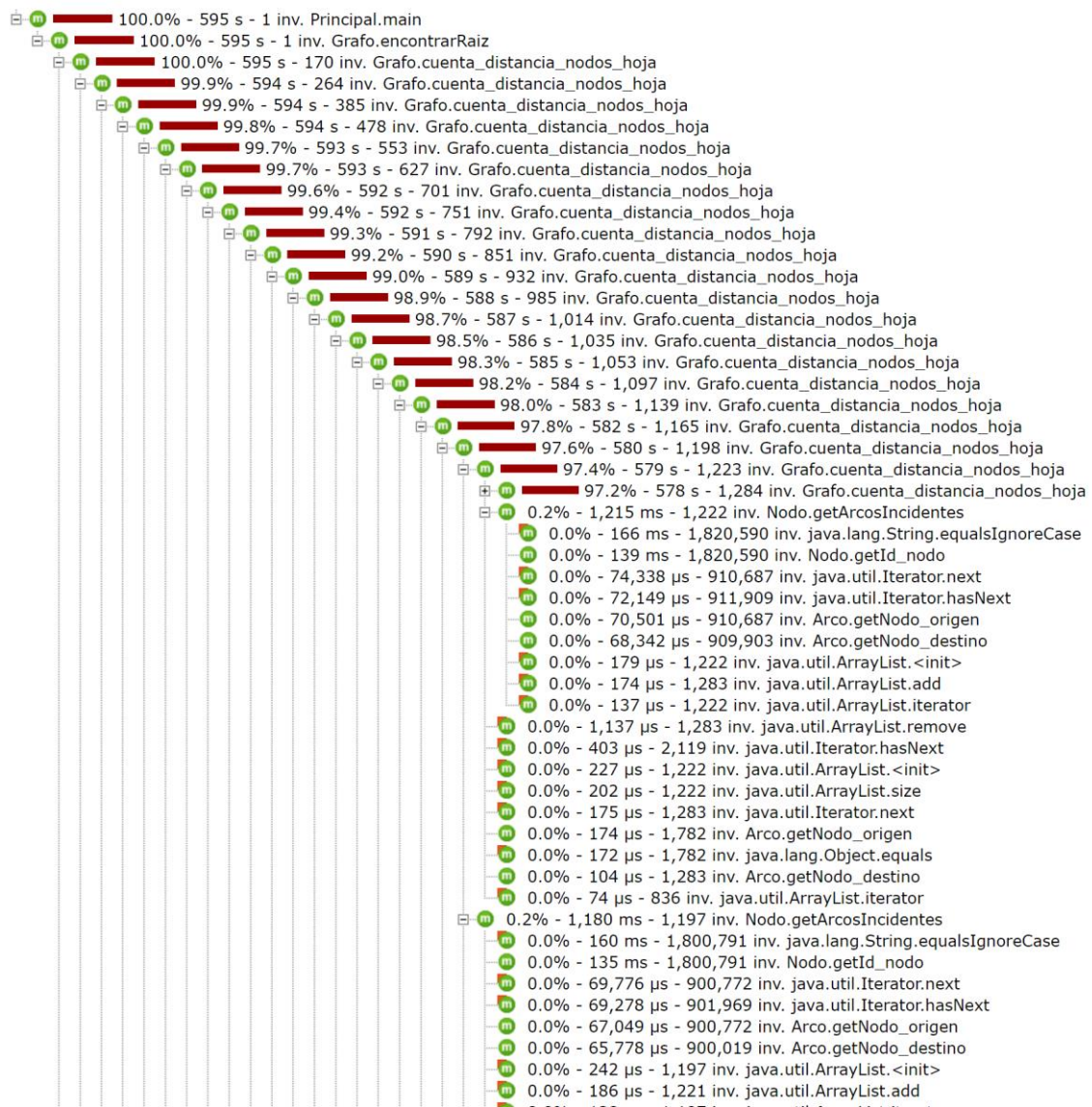
En esta imagen se muestra una contabilización de los tipos de datos que mas instancias tienen creadas a lo largo del programa, y cuanto ocupan en memoria. En este caso, tenemos que el tipo de datos que más veces se crea es el ArrayList, en segundo lugar, objetos propios del programa, y en tercer lugar, con muchas menos instancias ya, listas de enteros sin signo.

Grafo de llamadas



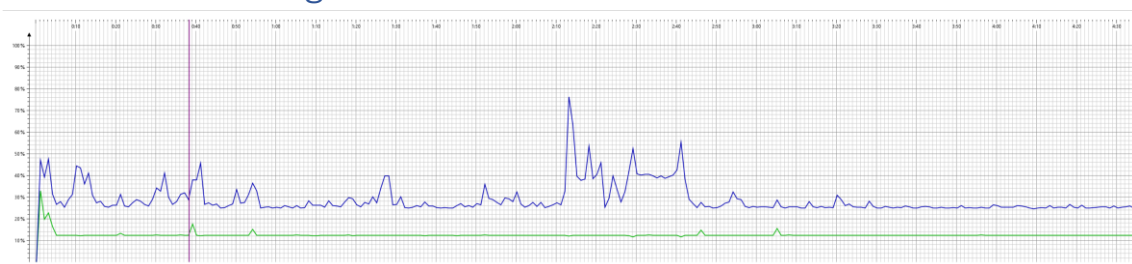
En esta imagen podemos ver el grafo de llamadas que se realiza alrededor de unos de los métodos críticos del programa como es el caso del método `getArcosIncidentes`. En este grafo se puede ver que métodos son los que llaman a esa función, y que otros métodos son llamados por esta función. Se ha hecho sobre este método ya que es uno de los que más tiempo y operaciones consume.

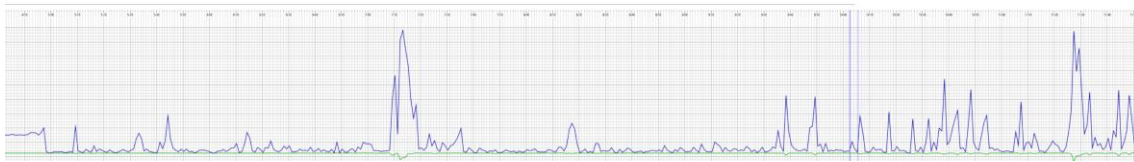
Árbol de llamada



Este es el árbol de llamada, en el que vemos como nuestro programa se enfrenta al problema de la recursión a la hora de recorrer los árboles, concretamente en el método de `cuenta_distancia_nodos_hoja`. Como vemos, este es de los métodos que más castigan a la eficiencia del programa debido a su alta complejidad y al gran número de veces que puede llegar a ejecutarse. Por otro lado, también se puede ver como cuando llega a un caso base, las recursiones se van recogiendo, volviendo al flujo de ejecución original.

Telemetría de carga de la CPU





Aquí tenemos el diagrama que representa la carga de la CPU en un tramo de tiempo de la ejecución. El eje de las Y representa la carga de la CPU. Se puede apreciar la carga de la CPU media está entre el 30% y el 50%, pero se producen picos en algunos instantes de tiempo en los que se necesita mayor capacidad computacional. Estos picos representan los momentos en los que se están procesando los nodos raíces de las ciudades más grandes y complejas, por lo que al programa le cuesta más calcularlos. Ejemplos: Ciudad Real, Alcázar, Puertollano, Valdepeñas, etc.

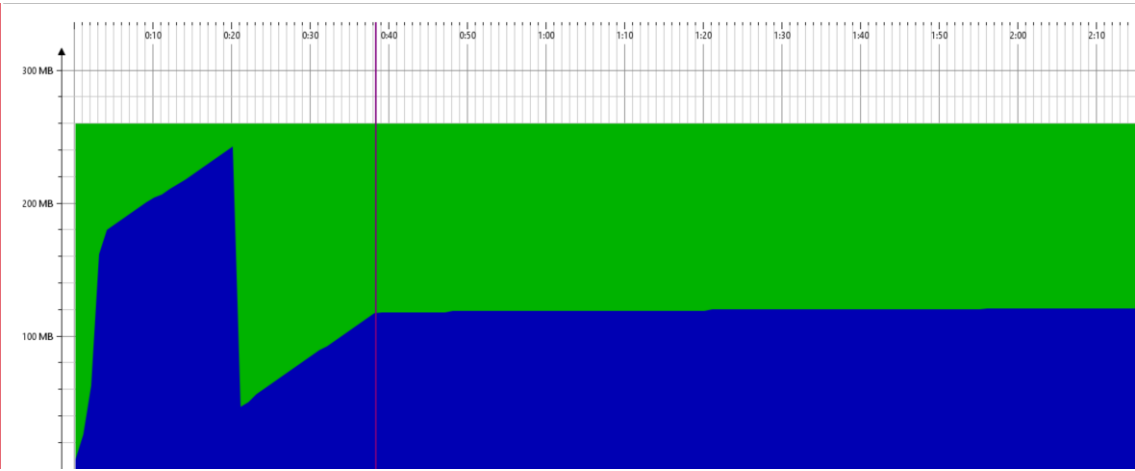
Hot Spots

	Hot Spot	Self Time	Average Time	Invocations
▲	Nodo.getArcosIncidentes	301 s (50 %)	1,218 µs	247,035
■	50.1% - 300 s - 246,865 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	50.1% - 300 s - 246,600 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	50.0% - 300 s - 246,213 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	50.0% - 300 s - 245,734 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	50.0% - 300 s - 245,181 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	49.9% - 299 s - 244,554 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	49.8% - 299 s - 243,853 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	49.8% - 299 s - 243,102 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	49.7% - 298 s - 242,310 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	49.6% - 298 s - 241,459 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	49.6% - 297 s - 240,526 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	49.5% - 297 s - 239,540 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	49.4% - 296 s - 238,525 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	49.3% - 295 s - 237,488 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	49.2% - 295 s - 236,434 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	49.1% - 294 s - 235,335 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	49.0% - 294 s - 234,194 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	48.9% - 293 s - 233,027 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	48.8% - 293 s - 231,828 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	48.7% - 292 s - 230,603 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	0.1% - 626 ms - 1,225 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, int, double, java.util.ArrayList)			
■	0.1% - 613 ms - 1,199 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, int, double, java.util.ArrayList)			
■	0.1% - 589 ms - 1,141 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, int, double, java.util.ArrayList)			
■	0.1% - 579 ms - 1,099 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, int, double, java.util.ArrayList)			
■	0.1% - 561 ms - 1,054 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, int, double, java.util.ArrayList)			
■	0.1% - 560 ms - 1,037 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, int, double, java.util.ArrayList)			
■	0.1% - 546 ms - 1,015 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, int, double, java.util.ArrayList)			
■	0.1% - 539 ms - 986 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, int, double, java.util.ArrayList)			
■	0.1% - 492 ms - 933 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, int, double, java.util.ArrayList)			
■	0.1% - 465 ms - 851 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, int, double, java.util.ArrayList)			
■	0.1% - 429 ms - 792 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, int, double, java.util.ArrayList)			
■	0.1% - 381 ms - 751 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, int, double, java.util.ArrayList)			
■	0.1% - 349 ms - 701 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, int, double, java.util.ArrayList)			
■	0.1% - 314 ms - 627 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, int, double, java.util.ArrayList)			
■	0.0% - 277 ms - 553 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, int, double, java.util.ArrayList)			
■	0.0% - 247 ms - 479 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, int, double, java.util.ArrayList)			
■	0.0% - 196 ms - 387 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, int, double, java.util.ArrayList)			
■	0.0% - 154 ms - 265 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, int, double, java.util.ArrayList)			
■	0.0% - 132 ms - 170 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, int, double, java.util.ArrayList)			
▲	java.lang.String.equalsIgnoreCase	77,951 ms (12 %)	0 µs	358,038,732
■	13.0% - 77,951 ms - 358,038,732 hot spot inv. Nodo.getArcosIncidentes			
■	13.0% - 77,912 ms - 357,546,478 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	13.0% - 77,871 ms - 356,989,526 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	13.0% - 77,819 ms - 356,273,536 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	12.9% - 77,755 ms - 355,400,269 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	12.9% - 77,681 ms - 354,410,893 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	12.9% - 77,599 ms - 353,314,227 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			
■	12.9% - 77,506 ms - 352,086,381 hot spot inv. Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)			

En este apartado podemos observar los métodos que dan alertas por ser puntos calientes, es decir, métodos que se llaman en gran cantidad de ocasiones y que afectan al tiempo de ejecución del programa. En esta caso podemos ver que salta un punto caliente en el método `getArcosIncidentes`, el cual lee la lista de arcos una gran cantidad de veces, y como hemos visto antes es llamado por las funciones de `get_distancia_nodos_hoja`.

También se puede apreciar que saltan puntos calientes de la ejecución en otros métodos como son `equalsIgnoreCase` o `getId_nodo`, ya que son frecuentes para la comprobación del id de los nodos en el flujo de ejecución del programa.

Telemetría de memoria



En esta imagen podemos ver en color verde la memoria libre, y en azul la memoria que esta siendo utilizada en el momento dado. La representación del pico que hay se produce a la hora de cargar todos los grafos en memoria. Una vez se han cargado todos, como hemos, el recolector de basura elimina datos no necesarios, y luego se estabiliza instantes después, aunque crece lentamente por la continuación de la ejecución del programa y la necesidad de usar distintas variables, pero no es nada preocupante.

Outliers – Funciones más usadas

Method	Total Time	Inv.	Avg. Time	Max. Time	Outlier Coeff.
java.lang.String.equalsIgnoreCase(java.lang.String)	72,060 ms	328,428,918	0 µs	15,794 µs	15794
Arco.getNodo_destino()	33,082 ms	164,371,691	0 µs	14,766 µs	14766
java.util.Iterator.hasNext()	34,646 ms	165,117,543	0 µs	11,900 µs	11900
Grafo.cuenta_distancia_nodos_hoja(Nodo, Arco, int, int, double, java.util.ArrayList)	62,672 s	226,457	276 ms	6,897 ms	23.92
Nodo.getArcosIncidentes(java.util.ArrayList)	553 s	226,593	2,443 µs	58,372 µs	22.89
Grafo.cuenta_distancia_nodos_hoja(Nodo, int, double, java.util.ArrayList)	555 s	157	3,537 ms	6,899 ms	0.95

En esta imagen se puede ver una clasificación de las funciones que más son llamadas, y algunas características acerca de ellas. Como ya hemos mencionado antes, aunque los métodos que son más usados sean los primeros que aparecen, si observamos bien las estadísticas, nos podemos dar cuenta de que en realidad los métodos que más tiempo consumen son los que hemos mencionado anteriormente: cuenta_distancia_nodos_hoja, y getArcosIncidentes.