Análisis híbrido: una propuesta práctica

Francisco Palomo Lozano, Inmaculada Medina Bulo
Dpto. de Lenguajes y Sistemas Informáticos. Universidad de Cádiz.
Escuela Superior de Ingeniería de Cádiz. C/ Chile, s/n. 11003 Cádiz.
{francisco.palomo, inmaculada.medina}@uca.es

Resumen

Este artículo defiende la necesidad de introducir el *análisis híbrido* en las prácticas de las asignaturas en las que se imparte análisis y diseño de algoritmos secuenciales, y presenta un entorno de trabajo adecuado para su realización en el laboratorio.

Esta técnica de análisis es una mezcla de las técnicas tradicionales de *análisis teórico y empírico*: en el análisis híbrido se emplea la primera de ellas para establecer un modelo y la segunda para obtener datos experimentales que, por último, permiten ajustar el modelo mediante regresión.

1. Introducción

En las recomendaciones ACM/IEEE-CS del año 1991 [1] se sugiere explícitamente que dentro del área AL de *algoritmos y estructuras de datos* y, en concreto, en las unidades AL4 (análisis de la complejidad), AL5 (clases de complejidad) y AL6 (ordenación y búsqueda), se realice la

> *...medida de los tiempos de algoritmos seleccionados de distintas clases de complejidad...*

y también la

> *...corroboración de la complejidad teórica mediante el empleo de métodos experimentales...*

como parte de las prácticas de laboratorio a desarrollar siguiendo un modelo de *laboratorio cerrado*.

Actualmente, tras la última revisión realizada en el 2001 [2], el área AL ha pasado a ser *algoritmos y complejidad* y en AL1 (análisis básico de algoritmos) aparece explícitamente recomendada la

> *...medida empírica del rendimiento de los algoritmos*.

Como expondremos a continuación, limitar los métodos experimentales a un *análisis empírico* presenta claros inconvenientes.

Por otro lado, pensamos que un enfoque más integrado en el que se emplee un método de *análisis híbrido* ayuda a que el alumno aplique en el laboratorio una parte mayor de los conocimientos sobre *análisis teórico* adquiridos en las clases de teoría.

Nos centraremos en el análisis del tiempo de ejecución, aunque el método descrito es aplicable al análisis de otros recursos computacionales (como el espacio) siempre que exista una manera sencilla de medir su consumo en la práctica.

2. Técnicas de análisis

Siguiendo a [4], un texto de análisis y diseño de algoritmos de amplio uso, las dos técnicas principales de análisis de algoritmos son el *análisis teórico* y el *análisis empírico*. De la mezcla de ambas surge el *análisis híbrido*.

2.1. Análisis teórico

En el análisis teórico se fija una determinada operación patrón y se calcula una función matemática que mide el número de instrucciones de dicho tipo que ejecuta el algoritmo frente a una determinada medida del tamaño de su entrada.

Decimos que una operación es *crítica* cuando su función asociada tiene al menos el mismo orden que la correspondiente a cualquier otra operación del algoritmo y *elemental* si en su implementación su tiempo de ejecución tiene un orden constante.

Cuando el algoritmo se analiza respecto de una operación crítica, el *principio de invariancia* garantiza que el orden de la función que mide el tiempo de ejecución de cualquier implementación del algoritmo (en la que dicha operación sea elemental) coincide con el del algoritmo.

Es decir, el resultado es independiente de la máquina, del lenguaje de programación y del programador. En tanto en cuanto se mantengan las condiciones descritas, los resultados podrán extrapolarse a la implementación: el orden del tiempo abstracto del algoritmo corresponderá con el del tiempo físico de ejecución del programa.

La principal desventaja radica en que un análisis teórico preciso puede ser muy complejo. Por ejemplo, el análisis de la familia de algoritmos de ordenación de Shell no se ha completado, pese a ser un método clásico sobre el que se ha investigado exhaustivamente.

Afortunadamente, ocurre en muchos casos que, aunque el análisis preciso de la función puede ser muy complicado, la obtención de su orden puede realizarse con un esfuerzo bastante menor.

2.2. Análisis empírico

En el análisis empírico se mide el tiempo de ejecución de una implementación para un número suficiente de datos de entrada. La gráfica resultante de representar los tiempos frente al tamaño de la entrada puede emplearse para comprobar la tendencia del algoritmo y compararlo con otros.

Su principal ventaja es que proporciona datos reales de ejecución para una determinada implementación. Las desventajas de este método son evidentes. Desde un punto de vista teórico, y a falta de otra información, no podemos predecir nada acerca del comportamiento del algoritmo fuera de los valores observados.

Desde un punto de vista práctico, se puede perder una gran cantidad de tiempo en implementar un algoritmo muy ineficiente que no nos interesa en absoluto. En muchas ocasiones este tiempo no compensará al que se hubiera empleado en realizar un análisis teórico con el fin de rechazar el algoritmo como impracticable.

Incluso si el algoritmo es ineficiente pero interesante, como en el caso de los que resuelven problemas inherentemente complejos (por ejemplo, problemas NP-completos), este enfoque presenta sus inconvenientes, ya que un análisis empírico requerirá ingentes cantidades de tiempo para recabar los datos experimentales.

2.3. Análisis híbrido

En el análisis híbrido se realiza primero un análisis teórico encaminado a determinar la forma aproximada de la función de tiempo y posteriormente un análisis empírico para recabar datos. Finalmente se completa nuestro conocimiento de la función con los datos extraídos experimentalmente.

Se observa que podemos descomponer el proceso de análisis en tres etapas: modelado, experimentación y regresión.

Modelado

Es necesario establecer un modelo aproximado del comportamiento de la función de tiempo.

Una forma inicial de establecer dicho modelo es emplear un criterio asintótico. La ventaja de este enfoque es que únicamente necesitamos conocer el orden del algoritmo.

Supongamos que el tiempo del algoritmo viene dado por $t(n)$, siendo n el tamaño de la entrada, y que hemos determinado que $t(n) \in \Theta(f(n))$, pese a desconocer la expresión exacta de $t(n)$.

En la gran mayoría de los casos que se nos presentan en la práctica $t(n) \in \Theta(f(n))$ porque

$$\lim \frac{t(n)}{f(n)} = a \in \mathbb{R} \ .$$

En tales casos, podemos tomar $\hat{t}(n) = af(n)$ como aproximación de $t(n)$ para valores de n lo suficientemente grandes, es decir, $\hat{t}(n)$ será un modelo de la función desconocida $t(n)$.

La desventaja es que, al ser un criterio asintótico, la estimación sólo será realmente precisa para valores de n lo suficientemente grandes, es decir, a partir de un cierto umbral. Con frecuencia, este umbral es pequeño y se obtienen buenos resultados incluso para valores pequeños de n.

Si el algoritmo es lo suficientemente sencillo y no sólo se conoce el orden de $t(n)$ sino que también se conoce su expresión exacta, pueden emplearse modelos más detallados, con más parámetros.

Experimentación

El algoritmo debe implementarse con sumo cuidado, de manera que el programa obtenido refleje fielmente la idea subyacente sin introducir costes ocultos adicionales que no hayan sido tenidos en cuenta durante el modelado.

Pero esto no es suficiente. Cualquier hipótesis sobre los datos de entrada que haya sido considerada durante el modelado (por ejemplo, al calcular el orden) debe ser reflejada al seleccionar los datos de entrada para el experimento.

Por ejemplo, al analizar en el caso promedio un algoritmo de ordenación por comparación para vectores, es muy común que se suponga que todos los elementos del vector de entrada son distintos y sus permutaciones equiprobables. A menos que los vectores de entrada para el experimento se seleccionen de acuerdo con dichos criterios, el resultado no será significativo.

La medida de los tiempos también ha de realizarse con cuidado. Es posible que los tiempos que estemos midiendo sean menores que la resolución de nuestro aparato de medida. Por ejemplo, hemos comprobado que, en nuestro sistema, la resolución de la función estándar `clock()` es muy baja: aproximadamente $0{,}01$ s. Si no disponemos de un instrumento más preciso, podemos realizar una medida indirecta, midiendo el tiempo total de un número suficiente de repeticiones del experimento y promediando el resultado. Es muy importante que para cada repetición de un mismo experimento se empleen exactamente los mismos datos de entrada.

Siguiendo con el ejemplo, basta medir el tiempo total de 10 repeticiones de un mismo experimento y dividirlo entre 10 para obtener su tiempo con una precisión de $0{,}001$ s.

Regresión

Una vez que se dispone de los datos experimentales, hay que estimar los parámetros de regresión que aparecen en el modelo. Nótese que, salvo en casos muy sencillos, el modelo no es lineal. Por lo tanto, una regresión lineal simple no es aplicable.

Los métodos de tipo LLS (*linear least squares*) permiten, a partir de una función $\hat{y}(x) = ax + b$ y un conjunto de observaciones (x_i, y_i), obtener los parámetros a y b que proporcionan el mejor ajuste del modelo a las observaciones en el sentido de minimizar el error cuadrático medio [6, 9].

El método se denomina lineal no porque lo sea el modelo respecto de x, sino porque lo es respecto de a y b. De hecho, puede generalizarse para tratar modelos del siguiente tipo:

$$\hat{y}(x) = \sum_{k=1}^{n} a_k f_k(x) \ ,$$

donde las $f_k(x)$ son funciones arbitrarias.

Los métodos de tipo NLLS (*non-linear least squares*) son más generales y permiten trabajar prácticamente con cualquier tipo de modelo. El más utilizado por los diversos paquetes especializados es el de Levenberg-Marquardt [7, 8, 9].

3. El entorno de trabajo

Nuestro entorno de trabajo en un laboratorio equipado para realizar prácticas de las características descritas está constituido íntegramente por herramientas de software libre. Todas pueden encontrarse en cualquiera de las distribuciones habituales de LINUX.

Sobre todo empleamos GNU MAKE, GNU C++ y GNU PLOT, que pertenecen todas al proyecto GNU. Son herramientas potentes, estables, muy probadas y, sobre todo, gratuitas, algo muy importante si queremos que el alumno pueda trabajar en casa sin tener que realizar un costoso desembolso adicional.

MAKE nos ayuda a organizar el trabajo, no sólo de compilación, sino de experimentación. Nuestra experiencia demuestra que emplear un poco de tiempo en explicar esta herramienta, y suministrar a los alumnos unos apuntes no muy extensos sobre ella, ayuda a mejorar el rendimiento de éstos en el laboratorio.

GNU PLOT nos permite representar gráficamente los resultados experimentales, contenidos en ficheros de texto, frente a los modelos, suministrados como funciones, y comparar «a ojo» la bondad del ajuste (orden `plot`). Además, implementa internamente el método de Levenberg-Marquardt con lo que nos permite ajustar los parámetros de regresión de modelos muy complejos (orden `fit`) sin tener que emplear herramientas externas.

Otra ventaja que presenta es que es programable mediante *guiones* que pueden ser ejecutados desde la línea de órdenes o, automáticamente, desde MAKE.

3.1. ¿Por qué C++?

C++ [5, 10] es un lenguaje multiparadigma que permite programar en una variedad de estilos. Podemos diferenciar tres paradigmas básicos dentro del lenguaje que pueden mezclarse entre sí para lograr aún más flexibilidad. Esto hace que el lenguaje se adapte bien a una gran variedad de itinerarios curriculares.

Por un lado, puede emplearse como un lenguaje estructurado. En este sentido, puede considerarse que C++ es un C mejorado con un sistema de tipos más estricto que el de C, espacios de nombres separados, un mecanismo de control de excepciones y sobrecarga de funciones y operadores.

Por otro lado, implementa el paradigma de la programación genérica a través del concepto de *plantilla*. Las plantillas son definiciones paramétricas que permiten un alto grado de abstracción.

Por último, permite programar utilizando orientación a objetos. A diferencia de los lenguajes orientados a objetos puros, C++ posee herencia múltiple y el enlace es estático por omisión.

La posibilidad de sobrecargar el operador de llamada a función dentro de una clase, permite además la creación de *objetos función* que pueden pasarse como parámetro dotando a C++ de características propias de los *lenguajes de orden superior*.

No es necesario para el alumno dominar el lenguaje para sacar partido de él. Mostrarle un subconjunto adecuadamente escogido en relación a sus conocimientos previos puede ser suficiente. En una asignatura donde se enseña análisis y diseño de algoritmos, los alumnos aprecian la eficiencia del lenguaje como un valor añadido. C++ fue diseñado expresamente para tal fin.

Otra razón de peso para elegir C++ es la existencia de una *biblioteca estándar de plantillas* asociada al lenguaje que permite trabajar cómodamente con contenedores y algoritmos genéricos. Se trata de la STL [3, 5]: la única biblioteca que conocemos que forma parte de un estándar internacional y que incluye como parte de su especificación (normativa para todas las implementaciones) requisitos de complejidad. Esto la hace especialmente apropiada para su empleo en un laboratorio de análisis y diseño de algoritmos.

La STL se incorporó al lenguaje tras aprobarse una propuesta de A. A. Stepanov y M. Lee en una reunión del comité ANSI/ISO para la estandarización de C++ celebrada en julio de 1994. El estándar ISO/IEC [5] se aprobó finalmente en 1998.

3.2. ¿Por qué la STL?

La biblioteca estándar de plantillas es una biblioteca de clases contenedoras, iteradores y algoritmos. Proporciona implementaciones muy eficientes de estructuras de datos y algoritmos que se necesitan habitualmente en el desarrollo de programas más complejos y es el resultado de años de investigación desarrollada por sus autores sobre *programación genérica*.

El que la STL sea una biblioteca especialmente diseñada para la programación genérica se refleja en el hecho de que prácticamente todos sus componentes son paramétricos.

Existen otras bibliotecas escritas en C++, algunas muy completas, como LEDA, que podrían utilizarse para nuestros propósitos. Frente a ellas, STL presenta la ventaja de formar parte de cualquier implementación de C++ que cumpla con el estándar. La mayoría de los fabricantes de compiladores tienden a que sus productos se encuentren en dicho estado, con lo que incluyen implementaciones de calidad de la STL integradas en la distribución de sus compiladores. Con esto se facilita la creación de código transportable entre distintas plataformas.

Pero, sin duda, una de las características más sorprendentes e innovadoras de la STL es que como parte de su especificación incluye la complejidad asintótica temporal. Esto significa que cualquier fabricante de compiladores o bibliotecas que desee cumplir el estándar de C++ está obligado a proporcionar implementaciones de sus algoritmos que cumplan ciertos requisitos de eficiencia.

4. Conceptos clave de la STL

A continuación se exponen de manera general, los aspectos más relevantes que presenta la STL y que facilitan nuestra tarea a la hora de analizar y diseñar algoritmos escritos en C++.

4.1. Especificaciones de complejidad

Todas las operaciones de la STL poseen una especificación de complejidad. Ésta puede venir dada por un orden asintótico o, en los casos más simples, por una función concreta. La mayoría de las veces se especifica la complejidad del peor caso y, en ocasiones, la del caso promedio.

Otras veces, la complejidad temporal se especifica mediante un *análisis amortizado*. Esto es útil cuando el tiempo de una operación puede sufrir grandes variaciones a lo largo de una secuencia de operaciones. En este caso un análisis en el peor caso podría ser excesivamente pesimista y un análisis en el promedio, poco significativo, por la gran desviación de los tiempos.

4.2. Contenedores

Los contenedores son objetos que se utilizan para almacenar otros objetos (incluso otro contenedor) proporcionando operaciones para su manipulación.

La STL posee diversas clases contenedoras agrupadas en dos categorías: secuencias (como los vectores) y contenedores asociativos ordenados (como los conjuntos). También posee adaptadores de secuencia (como las pilas) que se comportan como contenedores especializados. Los nombres de las clases y su descripción aparecen en el siguiente cuadro:

`vector`	Vector
`deque`	Cola doble
`list`	Lista
`set`, `multiset`	Conjunto y multiconjunto
`map`, `multimap`	Asociación mono/multivalor
`stack`	Pila
`queue`	Cola simple
`priority_queue`	Cola simple de prioridades

4.3. Iteradores

Los iteradores son una generalización del concepto de puntero y se emplean principalmente para recorrer un contenedor. Muchos algoritmos manejan rangos de iteradores.

El rango $[i, j)$ representa a todos los elementos comprendidos entre los iteradores i y j sin incluir al último. Si `c` es un contenedor, todos sus elementos pueden representarse mediante el rango $[$`c.begin()`, `c.end()`$)$.

Existen distintos tipos de iteradores y cada contenedor proporciona los apropiados para que puedan emplearse algoritmos eficientes sobre ellos.

Los vectores y las colas dobles proporcionan iteradores de acceso directo. Esto significa que se puede acceder a cualquier elemento en tiempo constante, es decir, que el acceso se puede considerar como una operación elemental. Sin embargo, los iteradores de las listas y los contenedores asociativos son únicamente bidireccionales, ya que no es posible realizar acceso directo a un elemento de estos contenedores en tiempo constante.

Así, un algoritmo genérico diseñado eficientemente para emplear acceso directo, como es el caso del algoritmo de ordenación `sort()` de la STL, sería ineficiente si se aplicara a una lista que proporcionara iteradores de «acceso directo» de complejidad $O(n)$. Por lo tanto, las listas no proporcionan tales iteradores y para compensar poseen su propia función `sort()`.

4.4. Algoritmos

Hemos visto que la STL define contenedores que incluyen algoritmos específicos, pero también define algoritmos que son independientes del contenedor, en el sentido de que pueden emplearse sobre cualquiera que cumpla unos determinados requisitos.

La genericidad de los algoritmos independientes del contenedor puede descomponerse en tres factores ortogonales de diseño: son paramétricos, reciben iteradores (en lugar de contenedores) y pueden recibir objetos función.

5. Algunos ejemplos concretos

Vamos a aplicar las técnicas previamente descritas al análisis en el promedio de tres algoritmos de ordenación tal y como debería hacerlo un alumno en el laboratorio.

Las funciones en C++ que se presentan a continuación implementan distintos algoritmos genéricos que ordenan un rango $[i, j)$ de iteradores de acceso directo. En adelante, $n = j - i$ representará el número de elementos del rango.

La función `merge_sort()` implementa el algoritmo de ordenación por fusión. La STL define `inplace_merge()` que funde dos rangos ordenados $[i, k)$ y $[k, j)$ en no más de $n - 1$ comparaciones si hay memoria suficiente. Supondremos que la hay, en caso contrario, `inplace_merge()` puede emplear hasta $O(n \log n)$ comparaciones.

```
template <typename I>
void merge_sort(I i, I j)
{
  if (j - i > 1) {
    I k = i + (j - i) / 2;
    merge_sort(i, k);
    merge_sort(k, j);
    inplace_merge(i, k, j);
  }
}
```

La función `median_quick_sort()` implementa una variante del algoritmo de ordenación rápida, de Hoare. En esta variante se emplea la mediana como pivote. La STL define `nth_element()` que emplea un tiempo promedio lineal en reorganizar los rangos $[i, k)$ y $[k, j)$ de forma que los elementos del primer rango no sean mayores a los del segundo y que en k quede el elemento que ocuparía dicha posición si $[i, j)$ estuviera ordenado.

```
template <typename I>
void median_quick_sort(I i, I j)
{
  if (j - i > 1) {
    I k = i + (j - i) / 2;
    nth_element(i, k, j);
    median_quick_sort(i, k);
    median_quick_sort(k, j);
  }
}
```

La función `heap_sort()` implementa el algoritmo de ordenación por montículo, de Williams. La STL define `make_heap()`, que construye un montículo en no más de $3n$ comparaciones, y también `sort_heap()`, que lo ordena en $n \log_2 n$ comparaciones.

```
template <typename I>
void heap_sort(I i, I j)
{
  make_heap(i, j);
  sort_heap(i, j);
}
```

Primero se elige el modelo. Estamos interesados en un análisis en el promedio. No es difícil deducir, aun sin conocer la expresión exacta de $t(n)$, que los tres algoritmos realizan $\Theta(n \log n)$ comparaciones en tal caso. Así, según el criterio asintótico, podemos tomar $\hat{t}(n) = an \ln n$ como modelo.

En segundo lugar, hemos de realizar los experimentos. Hay que cronometrar el tiempo entre dos puntos de un programa, para lo que creamos una clase que emplea la función `clock()` de la biblioteca para medir el tiempo por diferencia. Aquí `CPS` es `double(CLOCKS_PER_SEC)`, el número de unidades internas de tiempo por segundo.

```
class Cronometro {
  clock_t t0;
public:
  Cronometro() { t0 = clock(); }
  double tiempo()
  { return (clock() - t0) / CPS; }
};
```

Con el siguiente programa obtenemos los datos para un análisis en el promedio. Por cada n se crea un vector de n elementos que se rellena con los valores $1, \ldots, n$. Entonces, se permuta aleatoriamente obteniendo una de las $n!$ permutaciones posibles. Así es como se selecciona la entrada para cada experimento.

```
int main()
{
  for (int n = N0; n <= N; n += I) {
    vector<double> v(n);
    generate(v.begin(), v.end(), G());
    random_shuffle(v.begin(), v.end());
    vector<double> t = v;
    long int r = 0;
    Cronometro c;
    do {
      ORDENAR(v.begin(), v.end());
      v = t;
      ++r;
    } while (c.tiempo() < 1.0);
    cout << n << '\t'
         << c.tiempo() / r << endl;
  }
}
```

La medida del tiempo es adaptativa. Cada experimento se repite durante, al menos, 1 s. Esto asegura que si el experimento dura menos de $0{,}01$ s se repita al menos 100 veces, permitiendo obtener una precisión de $0{,}1$ ms.

En el programa, `ORDENAR` representa a cualquier algoritmo que ordene un rango. Por otro lado, `N0` es el tamaño inicial, `N` el final e `I` el incremento. Para el experimento escogimos los valores `500`, `50000` y `1000`, respectivamente.

Como se observa, `generate()` requiere un objeto función para poder generar la secuencia $1, \ldots, n$. Para ello creamos la clase `G`, en la que se sobrecarga el operador de llamada a función.

```
class G {
  double i;
public:
  G(): i(1.0) {}
  double operator ()() { return i++; }
};
```

Por último, un sencillo guión para GNU PLOT permite realizar la regresión y obtener una gráfica del resultado frente a las observaciones. Por ejemplo, para `heap_sort()` hacemos:

```
set dummy n
t(n) = a * n * log(n)
fit t(n) "heap_sort.dat" via a
plot "heap_sort.dat", t(n)
```

En este caso `fit` calcula que $a = 83{,}0 \cdot 10^{-9}$, con lo que podemos estimar que el programa tarda un tiempo de $83{,}0\, n \ln n$ ns $\approx 57{,}5\, n \log_2 n$ ns. Análogamente, se obtiene $a = 133{,}1 \cdot 10^{-9}$ para `median_quick_sort()` y $a = 192{,}3 \cdot 10^{-9}$ en el caso de `merge_sort()`.

En los tres casos el sistema nos informa de que la desviación típica de los errores es inferior a $0{,}002$ y de que el error asintótico para el parámetro estimado es inferior al $1\,\%$. Esto da una idea de la bondad del modelo. En la figura 1 se presenta una gráfica comparativa que muestra los datos experimentales y su regresión.

6. Evaluación de la propuesta

Actualmente utilizamos exclusivamente análisis empírico en nuestras prácticas. Parece que ésta es la situación general en la Universidad española, donde no tenemos conocimiento de experiencias docentes de implantación del método aquí propuesto.

La principal desventaja de este enfoque es que no resulta fácil para los alumnos contrastar los datos experimentales obtenidos en el laboratorio con los resultados teóricos presentados en clase. Se produce así un salto entre ambos niveles que influye negativamente en la formación del alumno.

Si bien suele ser fácil para ellos comprobar cualitativamente que dos algoritmos tienen complejidades diferentes, no lo es comparar dos algoritmos de similar complejidad ni comprobar el impacto de ciertas mejoras como la elección del umbral óptimo en algoritmos de «divide y vencerás».

No obstante, realizamos una experiencia piloto con la ocasión de un curso de verano y las encuestas establecieron un grado elevado de satisfacción por parte de los alumnos. Entre las ventajas apreciadas al utilizar análisis híbrido cabe destacar que:

- Permite plantear prácticas más realistas a los alumnos.
- Facilita la comparación de algoritmos de similar complejidad.
- Permite estimar la constante oculta en la notación asintótica.
- Permite realizar predicciones cuantitativas sobre la evolución temporal.

En concreto, hemos observado que el alumno queda muy satisfecho cuando se le suministra un algoritmo desconocido y es capaz por sus propios medios de establecer un modelo cuantitativo de su complejidad temporal en el laboratorio.

7. Conclusiones

Hemos diseñado un entorno de trabajo adecuado a la realización de prácticas de análisis híbrido en el laboratorio. Este entorno está construido a partir de herramientas potentes y estables de software libre, lo que reduce los costes de implantación y facilita su uso al alumno.

Se ha ilustrado paso a paso el desarrollo con dichas herramientas de una práctica de laboratorio de estas características en la que se comparan diversos algoritmos de ordenación y se han expuesto las ventajas que aporta la técnica de análisis híbrido frente a un análisis meramente empírico.

Pretendemos implantar esta propuesta en el próximo curso, en las asignaturas *análisis y diseño de algoritmos I y II* de los nuevos planes de estudio de Informática en nuestra universidad. Estas asignaturas se impartirán en segundo curso, durante el primer y segundo cuatrimestres, respectivamente.

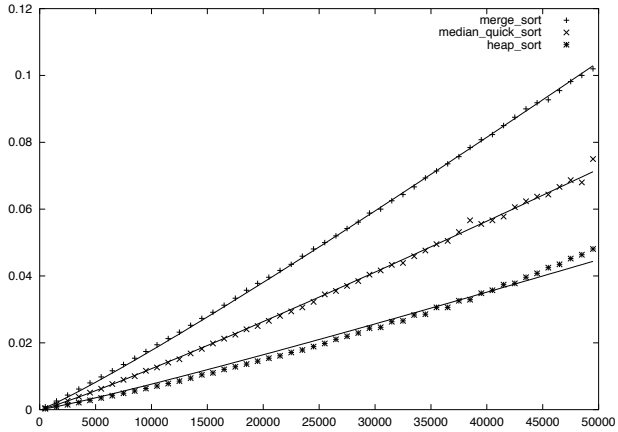

Figura 1: Resultados del análisis híbrido de los tres algoritmos

Referencias

[1] Joint Task Force on Computing Curricula. IEEE/ACM. *Computing Curricula 1991*. ACM Press and IEEE Computer Society Press (1991)

[2] Joint Task Force on Computing Curricula. IEEE/ACM. *Computing Curricula 2001*. ACM Press and IEEE Computer Society Press (2001)

[3] Austern, Matthew H. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley (1998)

[4] Brassard, Gilles & Bratley, Paul. *Fundamentos de Algoritmia*. Prentice-Hall (1997)

[5] ISO/IEC 14882:1998. *Programming Languages – C++*. (1998)

[6] Jain, Raj. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*. Wiley (1991)

[7] Levenberg, Kenneth. *A Method for the Solution of Certain Non-linear Problems in Least-Squares*. Quarterly of Applied Mathematics **2**(2) (1944)

[8] Marquardt, Donald W. *An Algorithm for the Least-Squares Estimation of Nonlinear Parameters*. SIAM Journal of Applied Mathematics **11**(2) (1963)

[9] Press, William H.; Teukolsky, Saul A.; Vetterling, William T. & Flannery, Brian P. *Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge University Press. 2ª ed. (2002)

[10] Stroustrup, Bjarne. *The C++ Programming Language. Special Edition*. Addison-Wesley (2000)