

Estudio Experimental de un Algoritmo Rápido de Intersección para Secuencias Ordenadas

Ricardo A. Baeza-Yates y Alejandro J. Salinger

Centro de Investigación de la Web

Departamento de Ciencias de la Computación

Universidad de Chile

Blanco Encalada 2120

Santiago 6511224, Chile

{rbaeza,asalinge}@dcc.uchile.cl

Abstract

This work presents an experimental comparison of intersection algorithms for sorted sequences, including the recent algorithm of Baeza-Yates. This algorithm performs on average less comparisons than the total number of elements of both inputs (n and m respectively) when $n = \alpha m$ ($\alpha > 1$). We can find applications of this algorithm on query processing in Web search engines, where large intersections, or differences, must be performed fast. In this work we concentrate in studying the behavior of the algorithm in practice, using for the experiments test data that is close to the actual conditions of its applications. We compare the efficiency of the algorithm with other intersection algorithm and we study different optimizations, showing that the algorithm is more efficient than the alternatives in most cases, especially when one of the sequences is much larger than the other.

Keywords: Set operations, merging, multiple search, Web search engines, inverted indices.

Resumen

Este trabajo presenta una comparación experimental de algoritmos de intersección para dos secuencias ordenadas, incluido el reciente algoritmo de Baeza-Yates. Este algoritmo realiza en promedio menos comparaciones que el número total de elementos de ambas entradas (n y m , respectivamente) cuando $n = \alpha m$ ($\alpha > 1$). Las aplicaciones de este algoritmo se encuentran en el procesamiento rápido de consultas en los motores de búsqueda Web, donde intersecciones o diferencias de conjuntos deben calcularse rápidamente. En este trabajo nos concentramos en estudiar el comportamiento del algoritmo en la práctica, utilizando para los experimentos datos de prueba cercanos a los datos reales de sus aplicaciones. Se comparó la eficiencia del algoritmo con otros algoritmos de intersección y se estudiaron diferentes optimizaciones, mostrando que es más eficiente que las alternativas en la mayoría de los casos, especialmente cuando una secuencia es mucho mayor que la otra.

Palabras claves: Operaciones sobre conjuntos, *merging*, búsqueda múltiple, motores de búsqueda Web, índices invertidos.

1. Introducción

Este trabajo estudia distintos algoritmos para calcular la intersección de secuencias ordenadas. Este problema constituye un caso particular de un problema genérico llamado búsqueda múltiple [3] (ver también [18], problema de investigación 5, página 156), el cual consiste en, dado un multiconjunto D (es decir, puede tener elementos repetidos), de n elementos escogidos de un universo que posee una relación de orden total, buscar en D cada elemento de otro multiconjunto Q , de m elementos escogidos del mismo universo. Los elementos encontrados constituyen, justamente, la intersección de ambos conjuntos.

El problema de intersección de listas ordenadas encuentra su motivación en los motores de búsqueda Web, pues la mayoría de ellos utiliza índices invertidos, en los cuales para cada palabra diferente se tiene una

lista de documentos o posiciones donde ella aparece. Generalmente estas listas están ordenadas por algún criterio, ya sea por posición, un ranking global precalculado, frecuencias de ocurrencias en un documento, u otro. Para calcular el resultado de una consulta, en la mayoría de los casos se requiere intersectar estas listas. En la práctica estas listas total o parcialmente ordenadas pueden tener cientos de millones de elementos, por lo que resulta útil contar con un algoritmo que es rápido y eficiente en promedio.

En el caso en que D y Q son conjuntos (y no multiconjuntos) ordenados, la búsqueda múltiple puede resolverse mezclando (*merging*) ambos conjuntos. Sin embargo, esto no resulta óptimo en todos los casos. Por ejemplo, si m es pequeño (digamos si $m = o(n/\log n)$), es mejor hacer m búsquedas binarias, obteniéndose un algoritmo de tiempo $O(m \log n)$ [2], siendo la métrica de complejidad el número de comparaciones entre cualquier par de elementos. El algoritmo de Baeza-Yates [1] alcanza ambas complejidades dependiendo del valor de m . En promedio realiza menos de $m + n$ comparaciones cuando ambos conjuntos están ordenados, considerando ciertas suposiciones pesimistas.

Este trabajo se enfoca en el estudio experimental de este último algoritmo y de diferentes optimizaciones a éste. Los experimentos consistieron en medir el tiempo de ejecución del algoritmo original y sus optimizaciones con listas de números enteros aleatorios ordenados y compararlos con un algoritmo basado en *merge* y con el algoritmo *Adaptive*, propuesto por Demaine *et al.* [11], el cual parece ser el más utilizado en la práctica. Nuestros resultados muestran que el algoritmo de Baeza-Yates es ligeramente mejor que *Adaptive* y mucho mejor que *Merge* cuando los tamaños de las listas difieren bastante.

En la sección 2 presentamos el trabajo relacionado. La sección 3 presenta la motivación para nuestro problema y algunas consideraciones prácticas. La sección 4 presenta el algoritmo de intersección de Baeza-Yates y algunas optimizaciones propuestas. La sección 5 presenta los resultados experimentales obtenidos. En este artículo supondremos que $n \geq m$ y los logaritmos son en base dos a menos que se indique explícitamente otra base. Una versión más sucinta y revisada de este trabajo será presentada en inglés en [7].

2. Trabajo Relacionado

Para resolver el problema de determinar si cualquier elemento de un conjunto de $n + m$ elementos es igual a otro se requieren a lo menos $\Theta((n + m) \log(n + m))$ comparaciones en el peor caso (ver [14]). Sin embargo, esta cota inferior no se aplica al problema de búsqueda múltiple ni, equivalentemente, al problema de intersección de conjuntos. Inversamente, las cotas inferiores del problema de búsqueda sí se aplican al problema de unicidad de un elemento [13]. Explotando esta idea, Demaine *et al.* definieron un algoritmo adaptativo de intersección múltiple de conjuntos [11, 12]. También definieron la dificultad de una instancia del problema, la cual fue redefinida luego por Barbay y Kenyon [9].

Cuando los conjuntos se encuentran ordenados las cotas inferiores para la intersección constituyen a su vez cotas inferiores para mezclar ambos conjuntos. Sin embargo, lo inverso no es cierto, pues en la intersección no necesitamos encontrar la posición real de cada elemento en la unión de ambos conjuntos, sólo debemos determinar si es que un elemento está en el otro conjunto o no. Existe bastante trabajo sobre el problema de mezcla con un número mínimo de comparaciones en el peor caso. Sin embargo, casi no se ha realizado trabajo sobre el caso promedio, pues éste no hace mucha diferencia. Esto último no es cierto para el problema de búsqueda múltiple, y por lo tanto para la intersección [3].

En el caso de la mezcla, Fernández de la Vega *et al.* [16] analizaron el caso promedio de una versión simplificada de la mezcla binaria de Hwang-Lin [17] encontrando que si $\alpha = n/m$ con $\alpha > 1$ y no potencia de 2, entonces el número esperado de comparaciones es

$$\left(r + \frac{1}{1 - \left(\frac{\alpha}{\alpha+1}\right)^{2^r}} \right) \frac{n}{\alpha}$$

con $r = \lceil \log_2 \alpha \rceil$. Cuando α es potencia de 2, el resultado es más complicado, pero similar. Fernández de la Vega *et al.* [15] también diseñaron un algoritmo probabilístico que mejora el algoritmo de Hwang-Lin en el peor caso para $1,618m \leq n \leq 3m$.

El algoritmo de Baeza-Yates [1] intenta adaptarse a los valores de la entrada. En el mejor caso el algoritmo realiza $\lceil \log(m + 1) \rceil \lceil \log(n + 1) \rceil$ comparaciones, lo cual para $m = O(n)$, es $O(\log^2(n))$. En el peor caso, el número de comparaciones que realiza el algoritmo es

$$W(m, n) = 2(m + 1) \log((n + 1)/(m + 1)) + 2m + O(\log(n)) .$$

Entonces, para m pequeño, el algoritmo es $O(m \log(n))$ y es $O(n)$ si $n = \alpha m$. En este caso, la razón entre este algoritmo y *merging* es $2(1 + \log(\alpha))/(1 + \alpha)$ asintóticamente, siendo 1 cuando $\alpha = 1$. El peor caso es peor que *merging* para $1 < \alpha < 6,3197$, teniendo su máximo cuando $\alpha = 2,1596$, donde es 1,336 veces más lento que *merging*. Por lo tanto el peor caso del algoritmo coincide con la complejidad de ambos enfoques, *merging* y búsqueda binaria múltiple, adaptándose a m . Para el caso promedio, bajo suposiciones pesimistas se demuestra que el número de comparaciones es

$$A(m, n) = (m + 1)(\ln((n + 1)/(m + 1)) + 3 - 1/\ln(2)) + O(\log n) .$$

Para $n = \alpha m$, la razón entre este algoritmo y *merging* es $(\ln(\alpha) + 3 - 1/\ln(2))/(1 + \alpha)$, lo cual es a lo sumo 0,7913 cuando $\alpha = 1,2637$ y 0,7787 cuando $\alpha = 1$. Este algoritmo es detallado en la sección 4.

3. Motivación: Proceso de Consultas en Índices Invertidos

Los índices invertidos se utilizan en la mayoría de los sistemas de recuperación de texto [4]. Lógicamente, éstos son un vocabulario (conjunto de palabras únicas encontradas en el texto) y una lista de referencias por palabra a sus ocurrencias (típicamente un identificador de documento y una lista de posiciones de la palabra en cada documento). En sistemas simples (modelo Booleano), las listas se ordenan por identificador de documento, y no existe *ranking* (es decir, no hay noción de relevancia de un documento). En ese escenario, el algoritmo básico de intersección se aplica directamente para calcular operaciones Booleanas sobre los identificadores de documentos: la unión (OR) equivale a *merge*, la intersección (AND) es la operación bajo estudio (sólo buscamos los elementos comunes), y la diferencia implica eliminar los elementos que están en el otro conjunto, lo que es similar a una intersección. En la práctica, las listas no se ordenan secuencialmente, sino por bloques. Sin embargo, estos bloques son de gran tamaño, y las operaciones pueden realizarse bloque a bloque.

En sistemas complejos se utiliza un *ranking*. Un *ranking* se basa típicamente en estadísticas de palabras (número de ocurrencias de una palabra por documento y la inversa del número de documentos que la incluye). Ambos valores pueden precalcularse y las listas de referencias se ordenan luego en orden decreciente por frecuencia de las palabras en el documento para así tener primero los documentos más relevantes. Las listas son luego procesadas en orden decreciente del inverso del número de documentos en los cuales cada una aparece (es decir, procesamos primero las listas más cortas), para así obtener primero los documentos más relevantes. Sin embargo, en este caso no siempre podemos tener una asignación de identificadores de documentos tal que las listas queden ordenadas según ese orden. Sin embargo, están parcialmente ordenadas por identificador para todos los documentos que tienen igual frecuencia de palabra.

El esquema anterior fue inicialmente utilizado en la Web, pero a medida que la Web fue creciendo, el *ranking* fue deteriorándose pues las estadísticas de palabra no siempre representan el contenido y calidad de una página Web y, además, puede ser engañado repitiendo y agregando palabras (casi) invisibles. En 1998, Page y Brin [10] describieron un motor de búsqueda (el cual fue el punto de partida de Google) que utilizaba enlaces (*links*) para medir la calidad de una página (PageRank). A esto se le llama *ranking* global basado en popularidad, y es independiente de la consulta realizada. Está fuera del alcance de este trabajo explicar Pagerank, pero podemos decir que éste modela un usuario aleatorio de la Web y que el *ranking* de una página es la probabilidad de que el usuario la visite. Esta probabilidad induce un orden total que puede utilizarse como identificador de documento. Por lo tanto, en un motor de búsqueda basado solamente en enlaces podemos utilizar el algoritmo de intersección tal como antes. Sin embargo, actualmente se utilizan esquemas de *ranking* híbridos que combinan evidencia de enlaces y palabras. A pesar de esto, un *ranking* basado en enlaces entrega buenos resultados aproximando bien el *ranking* real (que puede ser corregido mientras se calcula).

Otro tipo importante de consulta es la búsqueda de frases. En este caso utilizamos la posición de la palabra para saber si una palabra sigue o precede a otra. Entonces, como las frases suelen ser cortas, después de encontrar las páginas Web que contienen todas sus palabras, podemos procesar las dos primeras palabras¹ para encontrar pares adyacentes y luego ellas con la tercera y así sucesivamente. Esto es como calcular una intersección particular donde en vez de encontrar elementos repetidos tratamos de encontrar elementos correlativos (i e $i + 1$), y como las posiciones de las palabras están ordenadas, podemos utilizar nuevamente un algoritmo de intersección. Lo mismo es cierto para búsquedas por proximidad. En este caso, podemos

¹En realidad es más eficiente utilizar las dos palabras con listas más cortas, y así sucesivamente hasta llegar a la lista más larga si es que la intersección aún no es vacía.

tener un rango k de posibles posiciones válidas (es decir $i \pm k$) o podemos utilizar un *ranking* diferente dependiendo de la proximidad.

Finalmente, en el contexto de la Web, un algoritmo adaptativo es mucho más rápido en la práctica pues la suposición de una distribución uniforme es pesimista. En la Web, la distribución de las ocurrencias de las palabras es bastante desigual. Lo mismo ocurre con la frecuencia de las consultas. Ambas distribuciones siguen una ley de Zipf generalizada² [4, 6]. Sin embargo, la correlación entre ambas distribuciones es media [8] a baja [5]. Esto implica que el largo promedio de las listas involucradas en la consulta no es tan desbalanceado. Cuando la correlación es 0, si los largos promedio de las listas, n y m , cuando se toma una muestra, satisfacen $n = \Theta(m)$ (uniforme), en vez de $n = m + O(1)$ (ley de Zipf). Sin embargo, en ambos casos el algoritmo bajo estudio contribuye a una mejora.

4. El Algoritmo de Intersección

A continuación describimos el algoritmo de Baeza-Yates. Supongamos que D está ordenado. En este caso, si Q es pequeño, conviene buscar cada elemento de Q en D . Ahora, cuando Q también está ordenado, la intersección puede resolverse con *merging*. Tanto en el peor caso como en el caso promedio, una mezcla directa necesita $m + n - 1$ comparaciones. Sin embargo, para la intersección de conjuntos se puede hacer algo mejor. El siguiente algoritmo, que se basa en una doble búsqueda binaria, constituye una mejora sobre *merge* en el caso promedio bajo suposiciones pesimistas.

El algoritmo funciona de la siguiente manera. Primero buscamos con búsqueda binaria la mediana (elemento del medio) de Q en D . Si se encuentra, agregamos este elemento al resultado. Encontrado o no, hemos dividido el problema en buscar los elementos menores que la mediana de Q a la izquierda de la posición encontrada en D , o en la que debería estar el elemento si es que no se encontró, y los elementos mayores que la mediana hacia la derecha de esa posición. Luego resolvemos recursivamente en ambos pares de segmentos utilizando el mismo algoritmo. Si en cualquier momento, el tamaño del subconjunto de Q que estamos considerando es mayor que el del subconjunto de D , intercambiamos los roles de Q y D . Notar que la intersección de conjuntos es simétrica en este sentido. Si cualquiera de los subconjuntos es vacío, no se hace nada. La figura 1 muestra el algoritmo en pseudo-código.

```

Intersect( $D, Q, \text{min}D, \text{max}D, \text{min}Q, \text{max}Q$ )
1. //si es que  $Q$  o  $D$  es vacío, termina la recursión
2. if  $\text{min}D > \text{max}D$  or  $\text{min}Q > \text{max}Q$ 
3.     return  $\phi$ 
4.  $\text{media}Q \leftarrow Q[\text{posMedia}Q]$ 
5.  $\text{posMedia}D \leftarrow \text{bbinaria}(\text{media}Q, D, \text{min}D, \text{max}D)$ 
6. if  $D[\text{posMedia}D] == \text{media}Q$ 
7.      $\text{Resultado} \leftarrow \text{Resultado} \cup \{\text{media}Q\}$ 
8. if  $|D[\text{min}D..\text{posMedia}D - 1]| > |Q[\text{min}Q..\text{posMedia}Q - 1]|$  //el subconjunto de  $D$  es mayor que el de  $Q$ 
9.      $\text{Resultado} \leftarrow \text{Resultado} \cup \text{Intersect}(D, Q, \text{min}D, \text{posMedia}D - 1, \text{min}Q, \text{posMedia}Q - 1)$ 
10. else //se cambian los papeles de  $D$  y  $Q$ 
11.      $\text{Resultado} \leftarrow \text{Resultado} \cup \text{Intersect}(Q, D, \text{min}Q, \text{posMedia}Q - 1, \text{min}D, \text{posMedia}D - 1)$ 
12. if  $|D[\text{posMedia}D + 1..\text{max}D]| > |Q[\text{posMedia}Q + 1..\text{max}Q]|$  //el subconjunto de  $D$  es mayor que el de  $Q$ 
13.      $\text{Resultado} \leftarrow \text{Resultado} \cup \text{Intersect}(D, Q, \text{posMedia}D, \text{max}D, \text{posMedia}Q + 1, \text{max}Q)$ 
14. else //se cambian los papeles de  $D$  y  $Q$ 
15.      $\text{Resultado} = \text{Resultado} \cup \text{Intersect}(Q, D, \text{posMedia}Q + 1, \text{max}Q, \text{posMedia}D, \text{max}D)$ 
16. return  $\text{Resultado}$ 

```

Figura 1: Algoritmo de doble búsqueda binaria para intersección de dos conjuntos ordenados.

El algoritmo presentado en [1] puede verse como una versión balanceada del algoritmo de Hwang y Ling [17] adaptado a nuestro problema.

Una optimización simple al algoritmo es aplicar el algoritmo original no sobre ambos conjuntos D y Q completos, sino que buscar el subconjunto de uno de los dos donde existe un traslape, y por lo tanto, donde realmente se pueden encontrar elementos que formen parte de la intersección.

²Esta ley indica que el i -ésimo elemento tiene una frecuencia proporcional a $1/i^\alpha$, $\alpha > 0$.

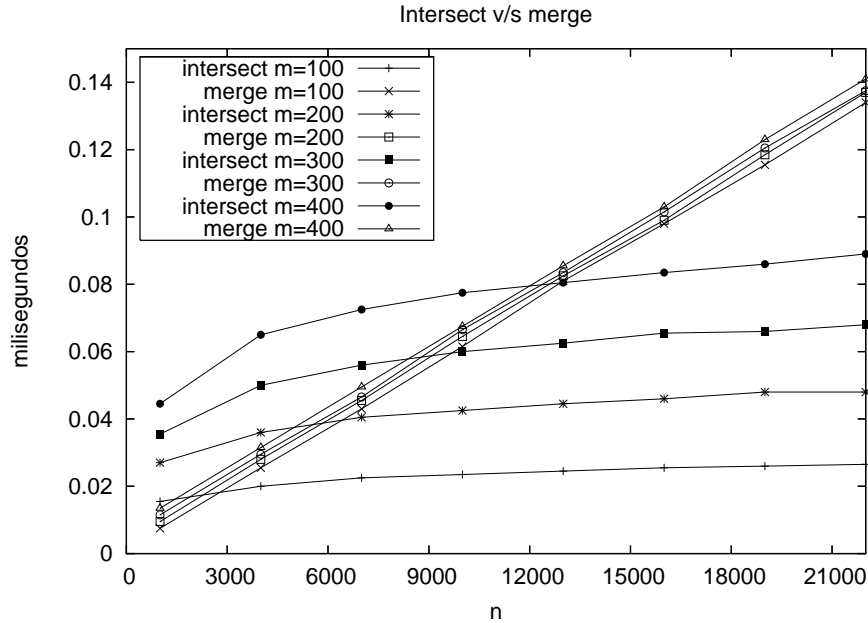


Figura 2: Resultados experimentales para intersect y merge para distintos valores de n y m

Se comienza comparando el menor elemento de Q con el mayor de D , y el mayor de Q con el menor de D . Si es que ambos conjuntos no se traslapan, la intersección es vacía. De lo contrario, se buscan el menor y el mayor elemento de D en Q para encontrar el traslape, lo que toma sólo tiempo $O(\log(m))$. Luego se aplica el algoritmo original con D y el subconjunto encontrado de Q . Esto mejora tanto el caso promedio como el peor caso. El caso en que se busca también el traslape en D también es válido, sin embargo esto toma tiempo $O(\log(n))$ y puede resultar costoso cuando m es mucho menor que n . Esta optimización se menciona en [1], pero no se estudia si es efectiva o no.

5. Resultados Experimentales

Comparamos la eficiencia de nuestro algoritmo, el cual llamamos *Intersect* con una implementación de intersección utilizando *merge* y con una adaptación del algoritmo *Adaptive* [11, 12] para la intersección de dos listas. Esta sección muestra, además, los resultados de los experimentos realizados con las optimizaciones al algoritmo.

Las listas utilizadas contienen números enteros aleatorios uniformemente distribuidos en el rango $[1, 10^9]$. Se varió el largo de una de las listas (n) de 1.000 a 22.000 con un paso de 3.000. Para cada uno de los largos se interseccionaron esas listas con listas de cuatro largos distintos (m), de 100 a 400. Cada punto representa el promedio de la ejecución sobre 20 instancias distintas, y de 1.000 ejecuciones con cada una (para eliminar variaciones debidas al sistema operativo dado los pequeños tiempos resultantes).

Los programas fueron implementados en C utilizando el compilador Gcc 3.3.3 y ejecutados sobre una plataforma Linux con un procesador Intel(R) Xeon(TM) de 3.06GHz con 512 Kb de cache y 2Gb RAM.

La figura 2 muestra una comparación entre nuestro algoritmo y *merge*. Se puede ver que nuestro algoritmo resulta mejor que *merge* a medida que aumenta n y que sus tiempos aumentan considerablemente a medida que aumenta m .

La figura 3 muestra los resultados obtenidos al implementar la optimización del algoritmo *Intersect*. Para esta comparación, también se agregó a *merge* el cálculo del traslape de ambas listas.

Podemos observar que no existen grandes diferencias entre el algoritmo original y el optimizado, más aún, el algoritmo original resultó algo más rápido que el optimizado. La razón por la cual la optimización no se traduce en una mejora considerable puede radicar en la distribución de los datos de prueba. Al ser los números sacados aleatoriamente de una distribución uniforme, en la mayoría de los casos el traslape de

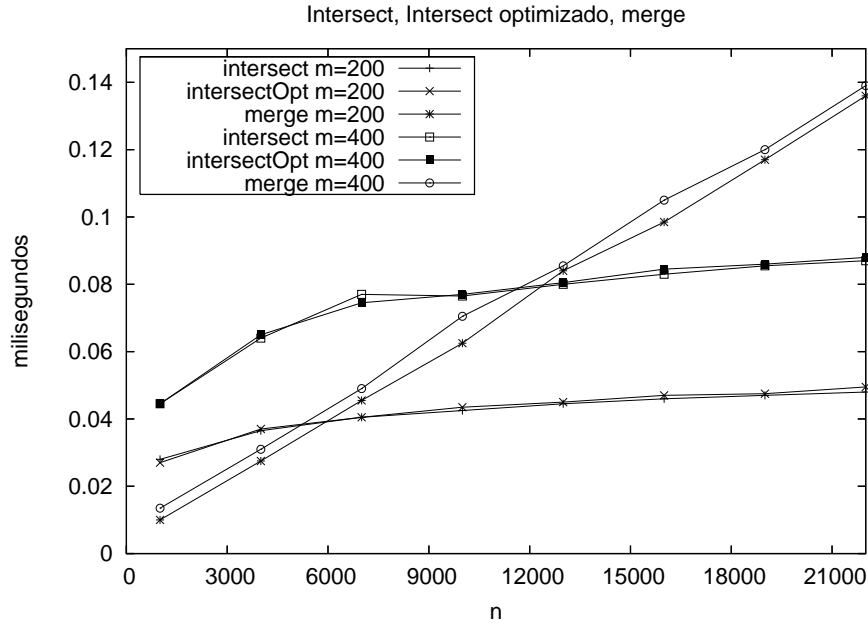


Figura 3: Resultados experimentales para Intersect, Intersect optimizado y merge, para distintos valores de n y para $m = 200$ y $m = 400$

ambos conjuntos abarca gran parte de Q . Entonces, la optimización no produce mejoras y sólo se traduce en que se ocupa algo de tiempo en buscar el traslape.

5.1. Algoritmos Híbridos

A partir de los resultados experimentales podemos ver que existe un tramo de valores de n para los cuales *merge* es mejor que nuestro algoritmo. Surge entonces la idea de combinar ambos algoritmos en un sólo algoritmo híbrido que ejecute cada uno de ellos según resulte conveniente.

Para saber cuál es el punto de corte donde conviene utilizar un algoritmo en vez del otro, para cada valor de n se midió el tiempo de ambos algoritmos con distintos valores de m hasta identificar el valor de m para el cual *merge* resulta más rápido que Intersect. Estos valores de m forman una recta en función de n , la que se observa en la figura 4. Esta recta es aproximada por $m = 0,033n + 8,884$, con una correlación $r^2 = 0,999$.

El algoritmo híbrido consiste en llamar a *merge* cuando $m > 0,033n + 8,884$, y a Intersect de lo contrario. La condición se evalúa en cada momento de la recursión.

Al modificar el algoritmo el punto de corte cambia y lo que nos gustaría sería encontrar el algoritmo híbrido óptimo. Aplicando la misma idea nuevamente, se encontró la recta que define los valores para los cuales *merge* resulta mejor que el algoritmo híbrido. Esta recta se aproxima por $m = 0,028n + 32,5$, con $r^2 = 0,992$. Definimos entonces el algoritmo Híbrido2 que ejecuta *merge* cuando $m > 0,028n + 32,5$ e Intersect en caso contrario. Por último, se combinaron ambos híbridos, creando una tercera versión en la cual la recta de corte entre *merge* e *Intersect* es el promedio entre las rectas de los híbridos 1 y 2. La recta resultante es $m = 0,031n + 20,696$. La figura 4 muestra la recta de corte entre el algoritmo original y *merge* y los resultados obtenidos con los algoritmos híbridos. El algoritmo óptimo en teoría sería el Híbrido. i cuando i tiende a infinito.

Se puede observar que los algoritmos híbridos registran tiempos menores a los del algoritmo original en el tramo en que éste es más lento que *merge*. Sin embargo, en el tramo posterior el algoritmo original resulta más rápido que los híbridos, debido a que en la práctica debemos evaluar el punto de corte en cada paso de la recursión. Entre los algoritmos híbridos, podemos ver que el primero es levemente mejor que el segundo y éste a su vez mejor que el tercero. Una idea para reducir el tiempo en el tramo en que el algoritmo original registra menores tiempos que los híbridos es crear un nuevo algoritmo híbrido que ejecute *merge* cuando resulte conveniente y que luego ejecute el algoritmo original, sin evaluar la relación entre m y n para ejecutar

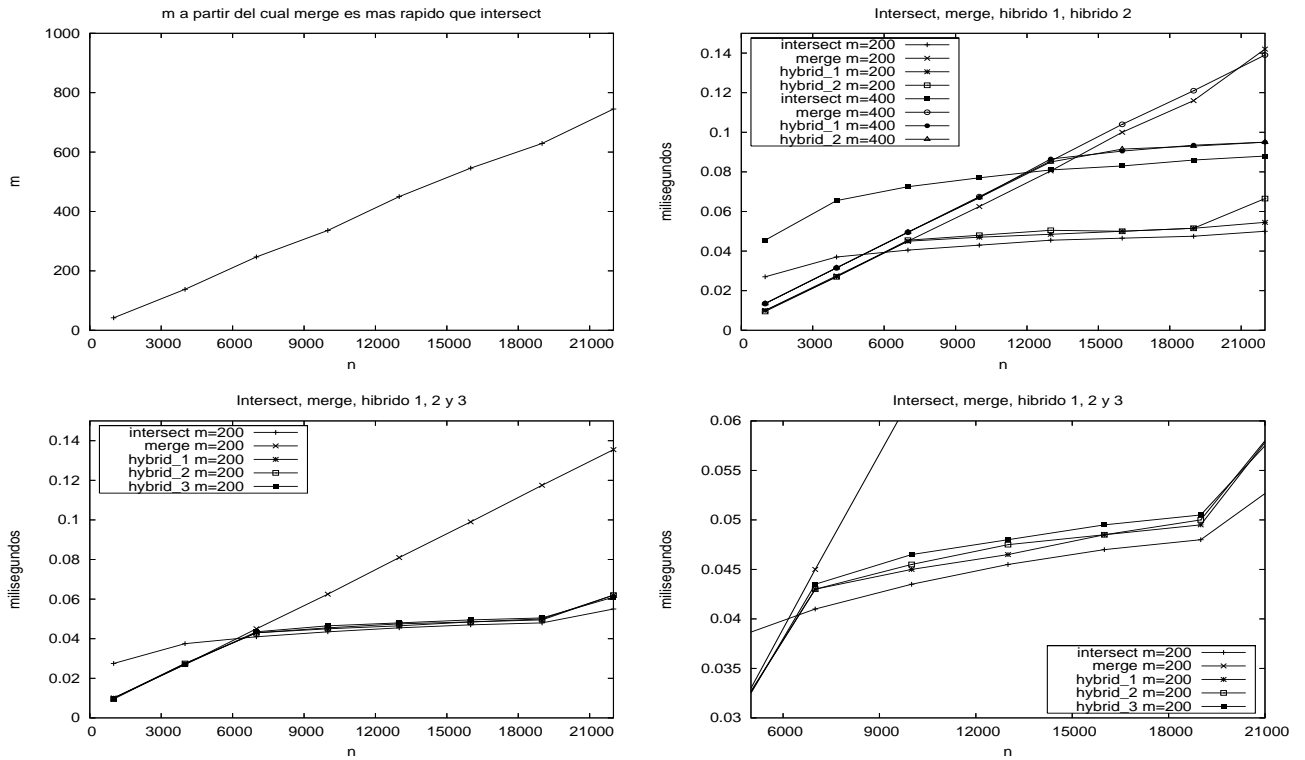


Figura 4: Arriba a la izquierda, valor de m a partir del cual $merge$ es más rápido que $Intersect$. A la derecha, comparación entre el algoritmo original, $merge$ y los híbridos 1 y 2 para $m = 200$ y $m = 400$. Abajo, comparación entre $Intersect$, $merge$ y los tres híbridos para $m = 200$. El gráfico de la derecha es un acercamiento del de la izquierda.

$merge$. Este algoritmo registra el mismo tiempo que $Intersect$ en el tramo donde éste resulta mejor que $merge$, combinando de la mejor manera las ventajas de ambos algoritmos. La figura 5 muestra los resultados obtenidos con este nuevo algoritmo híbrido.

5.2. Algoritmo Adaptive

Comparamos nuestro algoritmo con una versión del algoritmo *Adaptive* [11, 12], el cual funciona de la siguiente manera: se toma uno de los conjuntos, y se elige su primer elemento, el cual llamaremos $elim$. Se busca $elim$ en el otro conjunto, haciendo saltos exponenciales, esto es, en las posiciones $1, 2, 4, \dots, 2^i$. Si es que nos pasamos, es decir, el elemento en la posición 2^i es mayor que $elim$, hacemos una búsqueda binaria entre las posiciones 2^{i-1} y 2^i . Este tipo de búsqueda en un rango no acotado también es $O(\log n)$. Si encontramos $elim$, se agrega este elemento al resultado. Luego, se marca la posición anterior a la posición donde estaba $elim$ (o la posición donde debería haber estado) para acordarnos que desde allí hacia atrás ya se revisó el conjunto. Ahora se elige $elim$ como el menor elemento del conjunto que sea mayor al $elim$ anterior y se intercambian los roles, buscando ahora en el primer conjunto, haciendo saltos desde la posición que indica lo que ya hemos revisado del conjunto. Si en algún momento no existe un elemento mayor que el que se estaba buscando, se termina.

La figura 6 muestra una comparación entre los tiempos de nuestro algoritmo y *Adaptive*. Se puede observar que los tiempos de ambos algoritmos siguen la misma tendencia y que $Intersect$ resulta mejor que *Adaptive*.

5.3. Largos de las Secuencias con Distribución de Zipf

Una de las aplicaciones del algoritmo, como se dijo anteriormente, está en la búsqueda de documentos en la Web, en la que se cumple que el número de documentos en que aparecen las palabras sigue una distribución

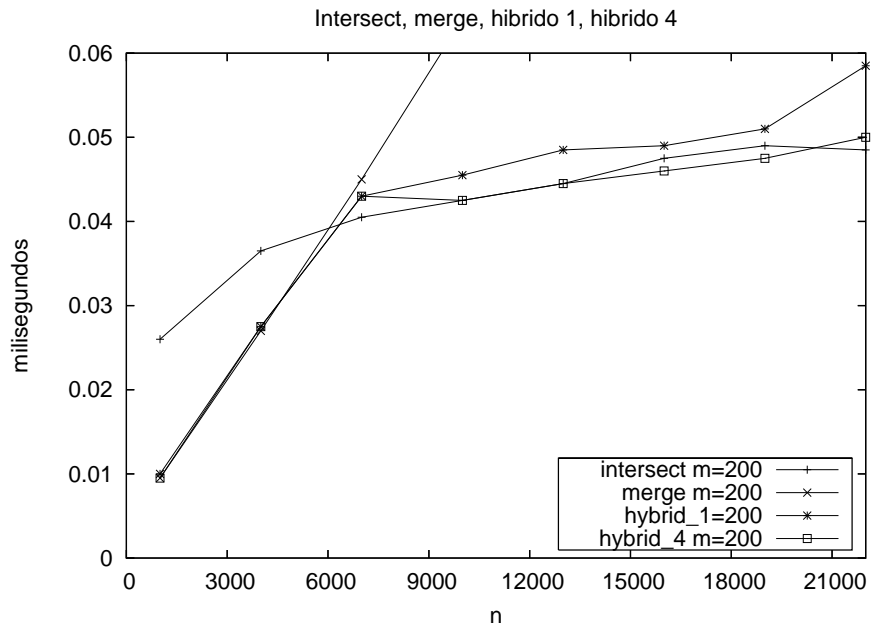


Figura 5: Resultados experimentales para Intersect, merge y los algoritmos híbridos 1 y 4 para distintos valores de n y para $m = 200$

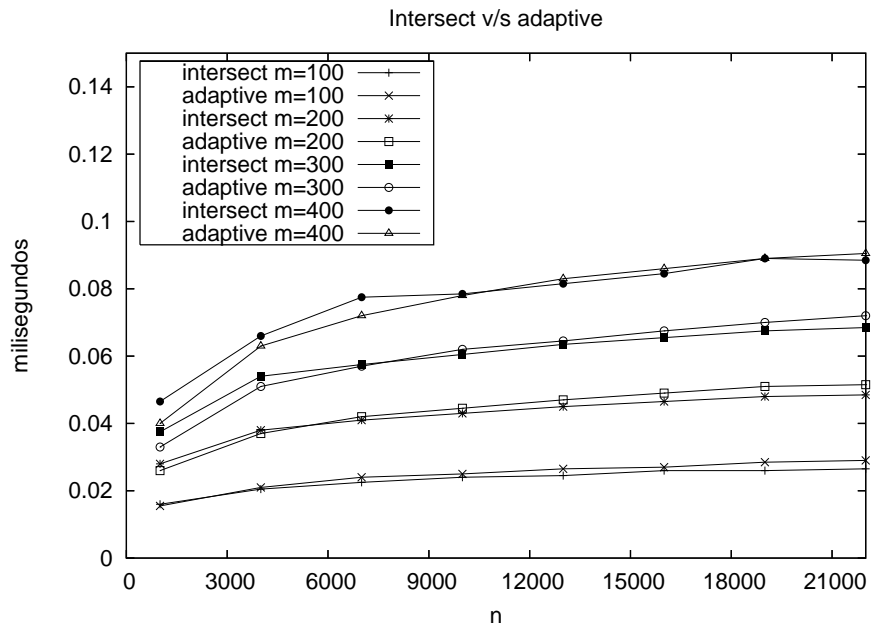


Figura 6: Resultados experimentales para Intersect y Adaptive, para distintos valores de n y m .

de Zipf.

Resulta interesante observar cómo se comporta el algoritmo *intersect* dependiendo de la razón de los largos de las dos secuencias cuando estos largos siguen una distribución de Zipf y la correlación entre ambos conjuntos es cero (caso ideal). Para hacer esta medición, se tomaron dos números aleatorios a y b con distribución uniforme entre 0 y 1.000, con los cuales se calcularon los largos de las listas D y Q como $n = K/a^\alpha$ y $m = K/b^\alpha$, respectivamente, con $K = 10^9$ y $\alpha = 1,8$, asegurando que $n > m$. Se hicieron 1.000 mediciones tomando 80 listas distintas para cada una de ellas, y repitiendo las ejecuciones 1.000 veces para cada lista.

La figura 7 muestra los tiempos obtenidos por ambos algoritmos en función de n/m tanto en escala normal como logarítmica.

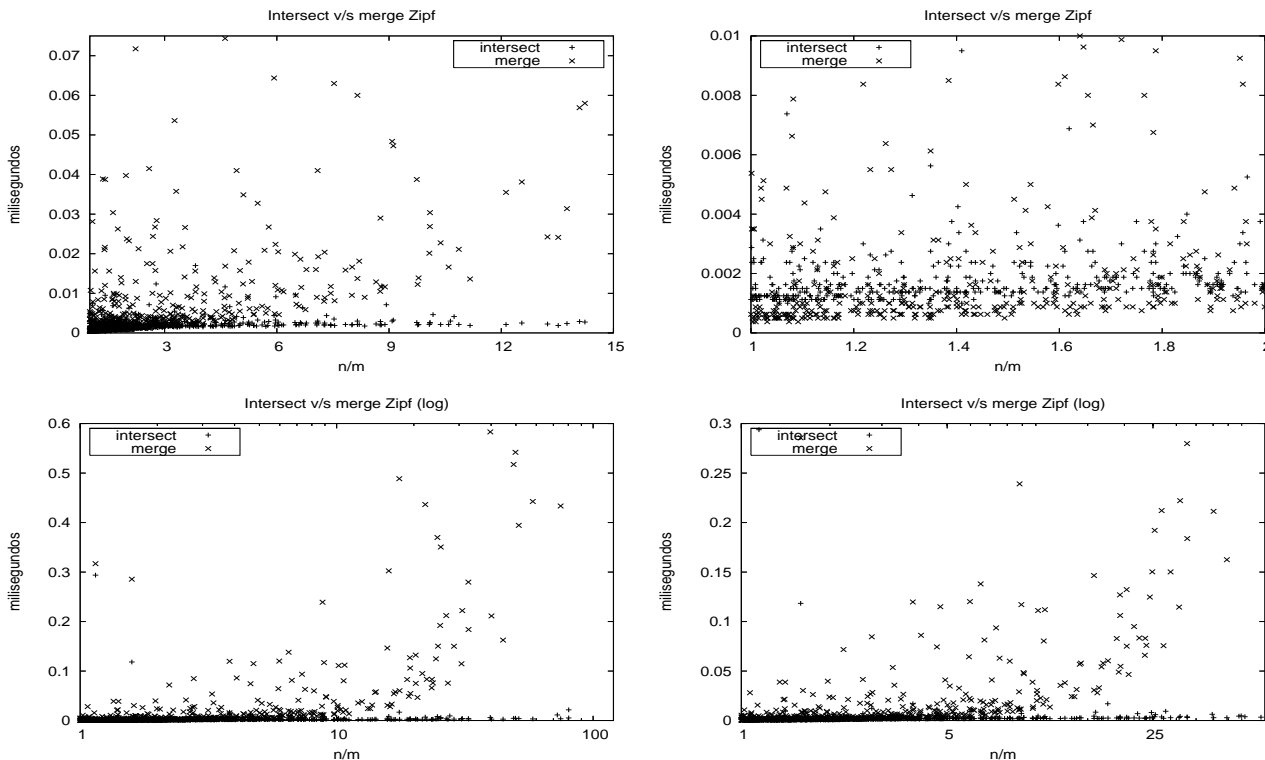


Figura 7: Arriba: tiempos de *Intersect* y *merge* en función de la razón de los largos de las listas cuando éstos siguen una distribución de Zipf. El gráfico de la derecha es un acercamiento del gráfico de la izquierda. Abajo: tiempos de *Intersect* y *merge* en escala logarítmica. El gráfico de la derecha es un acercamiento del de la izquierda.

En estos gráficos podemos observar que los tiempos de *Intersect* son menores que los de *merge* cuando n es bastante mayor que m . A medida que la razón entre n y m se hace menor, ya no es tan claro cuál de los dos algoritmos resulta más eficiente. Cuando $n/m < 2$ en la mayoría de los casos los tiempos de *merge* son los menores.

6. Conclusiones

En este trabajo hemos estudiado experimentalmente un algoritmo simple de intersección de conjuntos ordenados que funciona muy bien en promedio y que no inspecciona todos los elementos involucrados. A partir de los experimentos observamos que nuestro algoritmo original resulta más eficiente que *merge* cuando el tamaño de una de las listas es varias veces el tamaño de la otra. Esta mejora se hace más evidente a medida que n aumenta. A su vez, nuestro algoritmo supera a *Adaptive* [11, 12] para cualquier relación entre los tamaños de las listas. El algoritmo híbrido que combina *merge* y nuestro algoritmo a partir de la información empírica obtenida, aprovecha las ventajas de cada uno de ellos y resultó ser el más eficiente.

En la práctica no necesitamos calcular el resultado completo de la intersección de dos listas, pues la mayoría de la gente sólo mira los primeros resultados de una consulta, usualmente menos de las dos primeras páginas de resultados [6]. Además, calcular el resultado completo puede resultar demasiado costoso si es que una de las palabras de la consulta aparece en varios millones de documentos, como ocurre en la Web. Es por esto que, en general, se utilizan evaluaciones parciales y el resultado se va completando a medida que el usuario lo requiere. Si utilizamos el clásico algoritmo de mezcla obtenemos naturalmente primero las páginas Web más relevantes. Con nuestro algoritmo no es tan simple, pues aunque procesemos primero el lado izquierdo del problema recursivo, no necesariamente obtendremos el resultado en el orden deseado. Una solución simple es procesar el conjunto de menor tamaño de izquierda a derecha realizando búsquedas binarias en el conjunto mayor. Sin embargo, esta solución resulta eficiente sólo para pequeños valores de m , alcanzando una complejidad de $O(m \log n)$ comparaciones. Una variante optimista puede usar una predicción del número de páginas del resultado y utilizar un esquema intermedio adaptativo que divide los conjuntos de menor tamaño en trozos no simétricos con una inclinación hacia el lado izquierdo. Resulta entonces interesante estudiar la mejor manera de calcular resultados parciales de manera eficiente.

Como la correlación entre ambos conjuntos en la práctica es entre 0.2 y 0.6 dependiendo del texto Web usado (distribución de Zipf con α entre 1.6 y 2.0) y las consultas (distribución de Zipf con α menor, por ejemplo 1.4), queremos extender nuestros resultados experimentales a este caso. Sin embargo, ya vimos que en ambos extremos (correlación 0 o 1), el algoritmo en estudio es competitivo.

Referencias

- [1] R. Baeza-Yates. A Fast Set Intersection Algorithm for Sorted Sequences. En *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM 2004)*, Springer LNCS 3109, pp 400-408, Istanbul, Turkey, July 2004.
- [2] R. Baeza-Yates. *Efficient Text Searching*. PhD thesis, Dept. of Computer Science, University of Waterloo, May 1989. Also as Research Report CS-89-17.
- [3] R. Baeza-Yates, P.G. Bradford, J.C. Culberson, y G.J.E. Rawlins. The Complexity of Multiple Searching, unpublished manuscript, 1993.
- [4] R. Baeza-Yates y B. Ribeiro-Neto, *Modern Information Retrieval*, ACM Press/Addison-Wesley, England, 513 pages, 1999.
- [5] R. Baeza-Yates y Felipe Saint-Jean. A Three Level Search Engine Index bases in Query Log Distribution. En *SPIRE 2003*, Springer LNCS, Manaus, Brazil, October 2003.
- [6] Ricardo Baeza-Yates. Query Usage Mining in Search Engines. En *Web Mining: Applications and Techniques*, Anthony Scime, editor. Idea Group, 2004.
- [7] R. Baeza-Yates y A. Salinger. Experimental Analysis of a Fast Intersection Algorithm for Sorted Frequencies. Lecture Notes in CS, Springer, SPIRE 2005, Buenos Aires, November 2005.
- [8] R. Baeza-Yates, Carlos Hurtado, Marcelo Mendoza y Georges Dupret. Modeling user search behavior. En *LA-WEB 2005*, IEEE CS Press, Buenos Aires, Noviembre, 2005.
- [9] Jérémy Barbay y Claire Kenyon. Adaptive Intersection and t -Threshold Problems. En *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 390-399, San Francisco, CA, January 2002.
- [10] S. Brin y L. Page. The anatomy of a large-scale hypertextual Web search engine. En *7th WWW Conference*, Brisbane, Australia, April 1998.
- [11] Erik D. Demaine, Alejandro López-Ortiz, y J. Ian Munro. Adaptive set intersections, unions, and differences. En *Proceedings of the 11th Annual ACM-SIAM Symposium, on Discrete Algorithms*, pages 743-752, San Francisco, California, January 2000.
- [12] Erik D. Demaine, Alejandro López-Ortiz, y J. Ian Munro. Experiments on Adaptive Set Intersections for Text Retrieval Systems. En *Proceedings of the 3rd Workshop on Algorithm Engineering and Experiments*, LNCS, Springer, Washington, DC, January 2001.

- [13] Paul Dietz, Kurt Mehlhorn, Rajeev Raman, y Christian Uhrig. Lower Bounds for Set Intersection Queries. En *Proceedings of the 4th Annual Symposium on Discrete Algorithms*, 194-201, 1993.
- [14] David Dobkin y Richard Lipton. On the Complexity of Computations Under Varying Sets of Primitives, *Journal of Computer and Systems Sciences*, 18, 86-91, 1979.
- [15] W. Fernández de la Vega, S. Kannan, y M. Santha. Two probabilistic results on merging, *SIAM J. on Computing* 22(2), pp. 261-271, 1993.
- [16] W. Fernández de la Vega, A.M. Frieze, y M. Santha. Average case analysis of the merging algorithms of Hwang and Lin. *Algorithmica* 22 (4), pp. 483-489, 1998.
- [17] F.K. Hwang y S. Lin. A Simple algorithm for merging two disjoint linearly ordered lists, *SIAM J. on Computing* 1, pp. 31-39, 1972.
- [18] Gregory J.E. Rawlins. *Compared to What?: An Introduction to the Analysis of Algorithms*, Computer Science Press/W. H. Freeman, 1992.