Fast Intersection Algorithms for Sorted Sequences

Ricardo Baeza-Yates[1] and Alejandro Salinger[2]

[1] Yahoo! Research, Barcelona, Spain & Santiago, Chile
[2] Dept. of Computer Science, Univ. of Waterloo, Canada

Abstract. This paper presents and analyzes a simple intersection algorithm for sorted sequences that is fast on average. It is related to the multiple searching problem and to merging. We present the worst and average case analysis, showing that in the former, the complexity nicely adapts to the smallest list size. In the latter case, it performs less comparisons than the total number of elements on both inputs, n and m, when $n = \alpha m$ ($\alpha > 1$), achieving $O(m \log(n/m))$ complexity. The algorithm is motivated by its application to fast query processing in Web search engines, where large intersections, or differences, must be performed fast. In this case we experimentally show that the algorithm is faster than previous solutions.

1 Introduction

Our problem is a particular case of a generic problem called multiple searching [2] (see also [20], research problem 5, page 156). Given an n-element data multiset, D, drawn from an ordered universe, search D for each element of an m-element query multiset, Q, drawn from the same universe. An algorithm solving the problem must report any elements in both multisets. The metric is the number of three-way comparisons ($<, =, >$) between any pair of elements, worst case or average case. Throughout this paper $n \geq m$ and logarithms are base two unless explicitly stated otherwise.

Multiply search is directly related to computing the intersection of two sets. In fact, the elements found is the intersection of both sets. Although in the general case, D and Q are arbitrary, an important case is when D and Q are sets (and not multisets) already ordered. In this case, multiply search can be solved by merging both sets. However, this is not optimal for all possible cases. In fact, if m is small (say if $m = o(n/\lg n)$), it is better to do m binary searches obtaining an $O(m \lg n)$ algorithm. Can we have an adaptive algorithm that matches both complexities depending on the value of m? We present an algorithm which on average performs less than $m+n$ comparisons when both sets are ordered under some pessimistic assumptions. Fast average case algorithms are important for large n and/or m.

This problem is motivated by Web search engines. Most search engines use inverted indexes, where for each different word, we have a list of positions or documents where it appears. In some settings those lists are ordered by position or by a global precomputed ranking, to facilitate set operations between lists (derived from Boolean query operations), which is equivalent to the ordered case. In other settings, the lists of positions are sorted by frequency of occurrence in a document, to facilitate ranking based on the vector model [1,3]. The same happens with word positions in each file (full inversion to allow sentence searching). Therefore, the complexity of this problem is interesting also for practical reasons, as in search engines, partial lists can have hundreds of millions elements for very frequent words.

In Section 2 we present related work. Section 3 presents our intersection algorithm for two sequences as well as its analytical and experimental analysis. We also extend the algorithm to multiple sequences. Section 4 presents several hybrid algorithms tuned through experimental analysis. Section 5 presents the motivation for our problem, Web search engines, and experimental results for this case. We end with some concluding remarks and on-going work. This paper is an extended and revised version of [6,7].

2 Related Work

If an algorithm determines whether any elements of a set of $n + m$ elements are equal, then, by the element uniqueness lower bound in algebraic-decision trees (see [16]), the algorithm requires $\Omega((n + m) \lg(n + m))$ comparisons in the worst case. However, this lower bound does not apply to the search problem because a search algorithm does not need to determine the uniqueness of either D or Q; it need only to determine whether $D \cap Q$ is empty. For example, an algorithm for $m = 1$ must find whether some element of D equals the element in Q, not whether any two elements of D are equal. Conversely, however, lower bounds on the search problem (or, equivalently, the set intersection problem) apply to the element uniqueness problem [15]. In fact, this idea was exploited by Demaine *et al.* to define an *adaptive multiple set* intersection algorithm [13,14]. They also defined the difficulty of a problem instance, which was refined later by Barbay and Kenyon [8].

This adaptive algorithm [13,14] works as follows: we take one of the sets, and we choose its first element, which we call x. We search x in the other set, making exponential jumps, this is, looking at positions $1, 2, 4, \ldots, 2^i$. If we overshoot, that is, the element in the position 2^i is larger than x, we binary search x between positions 2^{i-1} and 2^i. This is an application of what is called *doubling search* or *galloping search*, which mimics binary search for unbounded sequences obtaining the same $O(\log n)$ complexity, a classical result of Bentley and Yao [10]. If we find x, we add it to the result. Then, we remember the position where x was (or the position where it should have been) so we know that from that position backwards we already processed the set. Now we set x as the smallest element of the set that is greater than the former x and we exchange roles, making jumps from the position that signals the processed part of the set. We finish when there is no element greater than the one we are searching.

For the ordered case, lower bounds on set intersection are also lower bounds for merging both sets. However, the converse is not true, as in set intersection we do not need to find the actual position of each element in the union of both sets, just if it is in D or not. Although there has been a lot of work on minimum comparison merging in the worst case, almost no research has been done on the average case because it does not make much of a difference. However, this is not true for multiple search, and hence for set intersection [2].

In the case of merging, Fernandez de la Vega *et al.* [18] analyzed the average case of a simplified version of Hwang-Lin's binary merge [19] finding that if $\alpha = n/m$ with $\alpha > 1$ and not a power of 2, then the expected number of comparisons is

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

with $r = \lfloor \lg_2 \alpha \rfloor$. When α is a power of 2, the result is more complicated, but similar. Simplifying, the average complexity is $O(m \log(n/m))$. Fernandez de la Vega *et al.* [17] also designed a probabilistic algorithm that improved upon Hwang-Lin's algorithm on the worst case for $1.618m \leq n \leq 3m$.

In the case of upper bounds, good algorithms for multiple search can be used to compute the intersection of two sets, obtaining the same time complexity. They can be also used to compute the union of two sets, by subtracting the intersection of both sets to the set obtained by merging both sets. Similarly to compute the difference of two sets.

As the most time-demanding operation on inverted indexes is the merging or intersection of the lists of occurrences, it is important to optimize it. Consider one pair of lists of sizes m and n respectively, that needs to be intersected. If m is much smaller than n, it is better to do m binary searches in the larger list to do the intersection, obtaining an $O(m \lg n)$ algorithm. Hence, if m is $o(n/\lg n))$, this algorithm is better than the linear merging algorithm that has complexity $O(n + m)$. Notice that each binary search can be performed in what was left to the right of the larger list in the previous binary search.

Later, Baeza-Yates' [6] devised a double binary search algorithm that is very fast if the intersection is trivially empty ($O(\log n)$) and requires less than $m + n$ comparisons on average. The exact average complexity is $O(m \log(n/m)))$ and although it is not shown explicitly in the original paper [13], the Adaptive algorithm also has the same average complexity for two sequences. A recent paper does a thorough performance comparison of these and other algorithms showing that the best algorithm depends also in the data distribution [9]. However, two other papers show that for compressed lists the best algorithms can be quite different [21,12].

3 A Simple But Good Average Case Algorithm

Suppose that D is sorted. In this case, obviously, if Q is small, will be faster to search every element of Q in D by using binary search. Can we do better if both sets are sorted? In this case set intersection can be solved by merging. In the

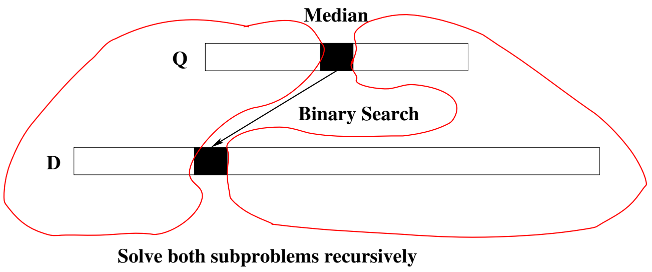

Fig. 1. Divide and conquer step of double binary search

worst or average case, straight merging requires $m+n-1$ comparisons. Can we do better for set intersection? The following simple algorithm improves on average under some pessimistic assumptions. We call it double binary search and can be seen as a balanced version of Hwang and Lin's [19] algorithm adapted to our problem, although in the literature is also called the Baeza-Yates' intersection algorithm (see for example [21]).

3.1 Double Binary Search

We first binary search the median (middle element) of Q in D. If found, we add that element to the result (a technical caveat is described later). Found or not, we have divided the problem in searching the elements smaller than the median of Q to the left of the position found on D, and the elements bigger than the median to the right of that position. We then solve recursively both parts using the same algorithm. If in any case, the size of the subset of Q to be considered is larger than the subset of D, we exchange the roles of Q and D. Note that set intersection is symmetric in this sense. If any of the subsets is empty, we do nothing. Figure 1 shows this divide and conquer approach.

An important detail is that if we want to use this algorithm in a sequential fashion, the output sequence should be sorted. For this, the comparison of the median should be done in between the two recursive calls as in Quicksort. Figure 3.1 shows the algorithm in pseudo-code.

A simple way to improve this algorithm is to start comparing the smallest elements of both sets with the largest elements in both sets. If both sets do not overlap, we use just $O(1)$ time. Otherwise, we search the smallest and largest element of D in Q, to find the overlap, using just $O(\lg m)$ time. Then we apply the previous algorithm just to the subsets that actually overlap. This improves both, the worst and the average case. The dual case is also valid, but then finding the overlap is $O(\lg n)$, which is not good for small m.

In the case of variable size lists, these algorithms can be applied in sequence by following the lists in order, like in the merging algorithm.

```
Intersect($D, Q, minD, maxD, minQ, maxQ$)
1.    //if $Q$ or $D$ are empty, we finish the recursion
2.    if $minD > maxD$  $bfor$ $minQ > maxQ$
3.        return $\emptyset$
4.    $miqQ \leftarrow round((minQ + maxQ)/2)$
5.    $midQval \leftarrow Q[midQ]$
6.    $midD \leftarrow binsearch(midQval, D, minD, maxD)$
7.    if $|D[minD..midD - 1]| > |Q[minQ..midQ - 1]|$ // subset($D$) > subset($Q$)
8.        $Result \leftarrow Result \cup Intersect(D, Q, minD, midD - 1, minQ, midQ - 1)$
9.    else //we exchange the roles of $D$ and $Q$
10.       $Result \leftarrow Result \cup Intersect(Q, D, minQ, midQ - 1, minD, midD - 1)$
11.   if $D[midD] == midQval$
12.       $Result \leftarrow Result \cup \{midQval\}$
13.       $midD \leftarrow posModD - 1$
14.   if $|D[midD + 1..maxD]| > |Q[midQ + 1..maxQ]|$// subset($D$) > subset($Q$)
15.       $Result \leftarrow Result \cup Iintersect(D, Q, midD, maxD, midQ + 1, maxQ)$
16.   else //we exchange the roles of $D$ and $Q$
17.       $Result \leftarrow Result \cup Intersect(Q, D, midQ + 1, maxQ, midD, maxD)$
18.   return $Result$
```

Fig. 2. Double binary search algorithm for intersecting two sorted sequences

3.2 Best and Worst Case Analysis

In the best case, the median element in each iteration always falls outside D (that is, all the elements in Q are smaller or larger than all the elements in D). Hence, the total number of comparisons is $\lceil \lg(m+1) \rceil \lceil \lg(n+1) \rceil$, which for $m = O(n)$ is $O(\lg^2 n)$. This shows that there is room for doing less work. The worst case happens when the median is not found and divides D into two sets of the same size (intuitively seems that the best and worst case are reversed). Hence, if $W(m, n)$ is the cost of the set intersection in the worst case, for m of the form $2^k - 1$, we have

$$W(m,n) = \lceil \lg(n+1) \rceil + W((m-1)/2, \lceil n/2 \rceil) + W((m-1)/2, \lfloor n/2 \rfloor) \ .$$

It is not difficult to show that

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

That is, for small m the algorithm has $O(m \lg n)$ worst case, while for $n = \alpha m$ it is $O(n)$. In this case, the ratio between this algorithm and merging is $2(1 + \lg(\alpha))/(1 + \alpha)$ asymptotically, being 1 when $\alpha = 1$. The worst case is worse than merging for $1 < \alpha < 6.3197$ having its maximum at $\alpha = 2.1596$ where it is 1.336 times slower than merging (this is shown in Figure 5). Hence the worst case of the algorithm matches the complexity of both, the merging and the multiple binary search, approaches, adapting nicely to the size of m. Figure 3 shows these two cases (top).

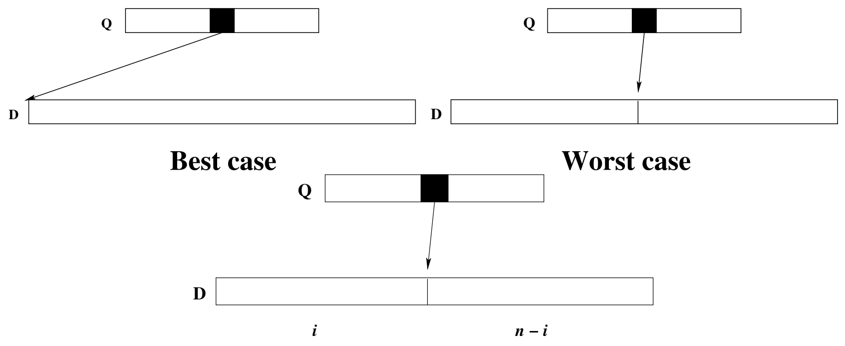

Fig. 3. Best and worst (top) as well as average (bottom) case analysis

3.3 Average Case Analysis

Let us consider now the average case. We use two assumptions: first, that we never find the median of Q and hence we assume that some elements never appear in D; and second, that the median will divide D in sets of size i and $n-i$ with the same probability for all i (this is equivalent to consider every element on D as random, like in the average case analysis of Quicksort). The first assumption is pessimistic, while the second considers that overlaps are uniformly distributed, which is also pessimistic regarding our practical motivation as we do not take in account that word occurrences may and will have locality of reference. The recurrence that we have to solve is

$$A(m,n) = \lceil \lg(n+1) \rceil + A(\lceil (m-1)/2 \rceil, \lceil n/2 \rceil) + A(\lceil (m-1)/2 \rceil, \lfloor n/2 \rfloor) \ ,$$

with $A(m,n) = A(n,m)$ if $m > n$ and $A(m,0) = A(0,n) = 0$, where $A(m,n)$ is the average number of comparisons to intersect two lists of size m and n. The rationale for this formula is shown in Figure 3 (bottom). Figure 4 shows the actual number of comparisons for $n = 128$ and all powers of 2 for $m \leq n$, for all the cases already mentioned.

To analyze the recurrence above, let us consider, without loss of generality, the case for m of the form $2^k - 1$. Then we have

$$A(m,n) = \lceil \lg(n+1) \rceil + \frac{1}{n+1} \sum_{i=0}^{n} (A((m-1)/2, i) + A((m-1)/2, n-i)) \ .$$

We now show that

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

The recurrence equation can be simplified to

$$A(m,n) = \lceil \lg(n+1) \rceil + \frac{2}{n+1} \sum_{i=0}^{n} A((m-1)/2, i) \ .$$

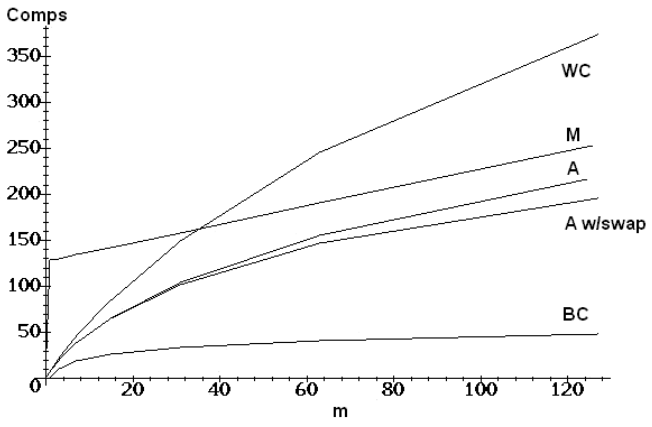

Fig. 4. Number of comparisons in the best, worst and average case (with and without swaps) for $n = 128$, as well as for merging (M)

As the algorithm is adaptive on the size of the lists, we have

$$A(m,n) = \lceil \lg(n+1) \rceil + \frac{2}{n+1} \left[\sum_{i=0}^{(m-1)/2} A\left(i, \frac{m-1}{2}\right) + \sum_{(m+1)/2}^{n} A\left(\frac{m-1}{2}, i\right) \right]$$

by noticing that we switch the sets when $m > n$. However, solving this version of the recurrence is too hard, so we do not include this improvement in the analysis. Nevertheless, this does not affect the main order term. Notice that our analysis allows any value for n.

Making the change of variable $m = 2^k - 1$ and using k as sub-index we get

$$A_k(n) = \lceil \lg(n+1) \rceil + \frac{2}{n+1} \sum_{i=0}^{n} A_{k-1}(i) \ .$$

Eliminating the sum, we obtain

$$(n+1)A_k(n) = nA_k(n-1) + 2A_{k-1}(n) + \lceil \lg(n+1) \rceil + n\delta(n = 2^j) \ ,$$

where $\delta(n = 2^j)$ is 1 if n is a power of 2, or 0 otherwise. Let $T_n(z) = \sum_k A_k(n) z^k$ be the generating function of A in the variable k. Hence

$$T_n(z) = \frac{n}{n+1-2z} T_{n-1}(z) + \frac{\lceil \lg(n+1) \rceil + n\delta(n = 2^j)}{(n+1-2z)(1-z)} \ .$$

Unwinding the recurrence in the sub-index of the generating function, as $T_0(z) = 0$, we get

$$T_n(z) = \frac{n!}{(1-z)\Gamma(n+2-2z)} \sum_{i=1}^{n} \frac{\Gamma(i+1-2z)}{i!} (\lceil \lg(i+1) \rceil + i\delta(i = 2^j)) \ ,$$

where $\Gamma(x)$ is the Gamma function (if x is a positive integer, then $\Gamma(x) = (x-1)!$). Let $\alpha_{i,j}$ be $\lceil \lg(i+1) \rceil + i\delta(i = 2^j)$. Simplifying, we have

$$T_n(z) = \frac{1}{(1-z)(n+1)} \sum_{i=1}^{n} \prod_{j=i+1}^{n+1} \frac{\alpha_{i,j}}{\left(1 - \frac{2z}{j}\right)} .$$

Expanding we have

$$T_n(z) = \frac{\sum_{r \geq 0} z^r}{n+1} \sum_{i=1}^{n} \prod_{j=i+1}^{n+1} \alpha_{i,j} \sum_{\ell \geq 0} \left(\frac{2z}{j}\right)^{\ell} .$$

Now, we have $A(2^k - 1, n) = [z^k]T_n(z)$ where $[z^k]f(z)$ is the coefficient of z^k in $f(z)$. As the coefficient of z^r is 1, for $r \leq k$ we need to compute in the right side the coefficient of z^{k-r}. That is

$$A(2^k - 1, n) = \frac{1}{n+1} \sum_{r=0}^{k} \sum_{i=1}^{n} [z^{k-r}] \prod_{j=i+1}^{n+1} \alpha_{i,j} \sum_{\ell \geq 0} \left(\frac{2z}{j}\right)^{\ell} .$$

Then

$$A(2^k - 1, n) = \frac{1}{n+1} \sum_{r=0}^{k} 2^{k-r} \sum_{i=1}^{n} \prod_{\sum_{j=i+1}^{n+1} \ell_j = k-r} \alpha_{i,j} \left(\frac{1}{j^{\ell_j}}\right)^{\ell} .$$

With the help of the Maple symbolic algebra system, we obtain the main order terms sought.

For $n = \alpha m$, the ratio between this algorithm and merging is $(\ln(\alpha) + 3 - 1/\ln(2))/(1+\alpha)$ which is at most 0.7913 when $\alpha = 1.2637$ and 0.7787 when $\alpha = 1$. This is also shown in figure 5, where we also include the average case analysis of Hwang and Lin's algorithm [18]. Recall that this analysis uses different assumptions, however shows the same behavior, improving over merging when $\alpha \geq 2$.

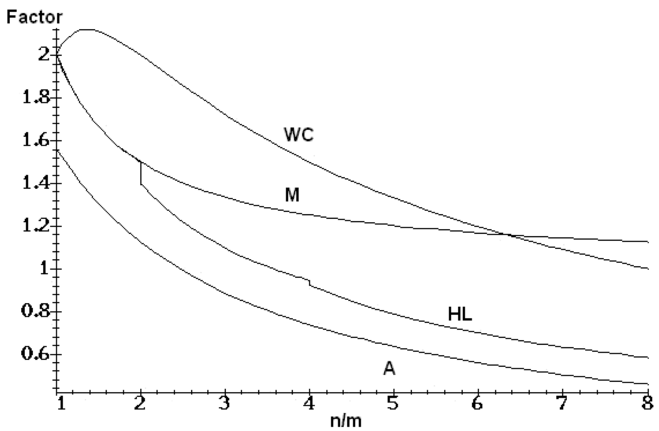

Fig. 5. Constant factor on n depending on the ratio $\alpha = n/m$

3.4 Experimental Analysis

Now we compare the efficiency of the algorithm, which we call *Intersect* in this section, with an intersection algorithm based on merging, and with an adaptation of the *Adaptive* algorithm [13,14] for the intersection of two sequences. In addition, we show the results obtained with the optimizations of the algorithm.

We used sequences of integer random numbers, uniformly distributed in the range $[1, 10^9]$. We varied the length of one of the lists (n) from 1,000 to 22,000 with a step of 3,000. For each of these lengths we intersected those sequences with sequences of four different lengths (m), from 100 to 400. We use twenty random instances per case and ten thousand runs (to eliminate the variations due to the operating system given the small resulting times).

The programs were implemented in C using the Gcc 3.3.3 compiler in a Linux platform running an Intel(R) Xeon(TM) CPU 3.06GHz with 512 Kb cache and 2Gb RAM.

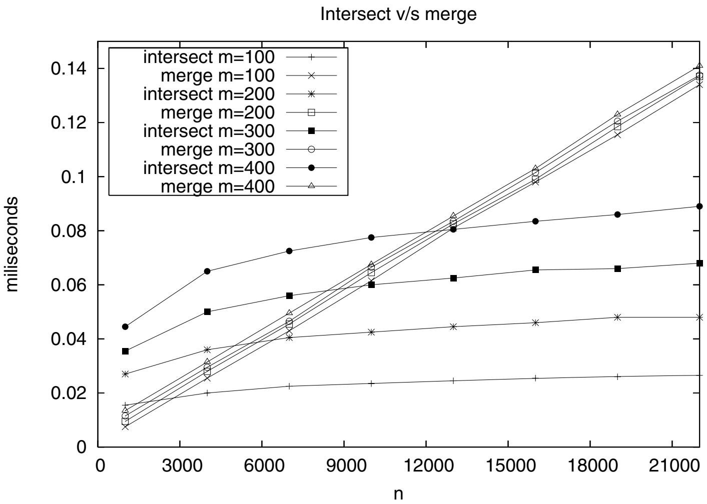

Fig. 6. Experimental results for Intersect and Merge for different values of n and m

Figure 6 shows a comparison between Intersect and Merge. We can see that Intersect is better than Merge when n increases and that the time increases for larger values of m.

Figure 7 shows a comparison between the times of Intersect and Adaptive. We can see that the times of both algorithms follow the same tendency and that Intersect is marginally better than Adaptive.

Figure 8 shows the results obtained with the Intersect algorithm and the optimization described at the end of the last section. For this comparison, we also added the computation of the overlap of both sequences to Merge.

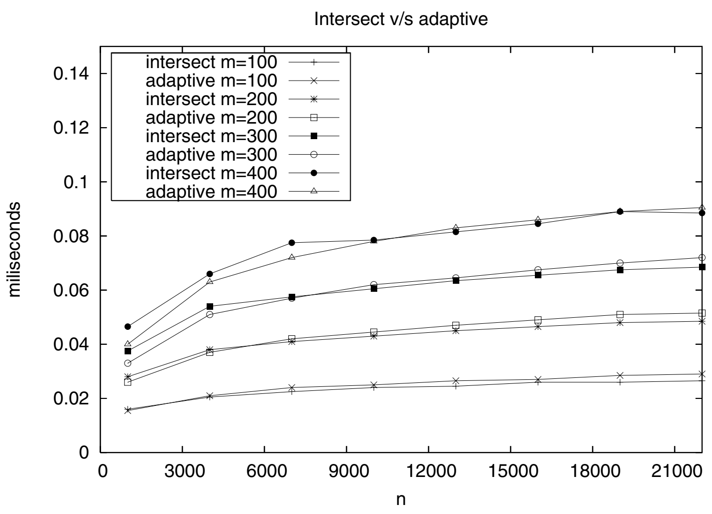

Fig. 7. Experimental results for Intersect and Adaptive, for different values of n and m

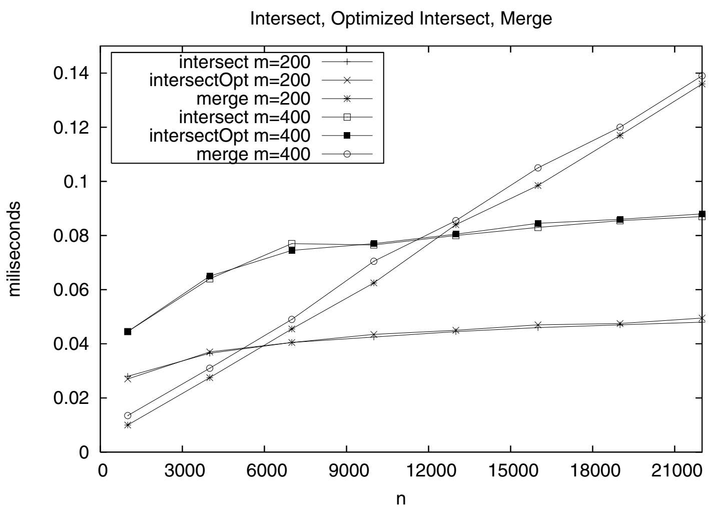

Fig. 8. Experimental results for Intersect, optimized Intersect and Merge, for different values of n and $m = 200$ and $m = 400$

We can see that there is no big difference between the original and the optimized algorithm, and moreover, the original algorithm was a bit faster than the optimized one. The reason why the optimization did not result in an improvement can be the uniform distribution of the test data. As the random numbers are uniformly distributed, in most cases the overlap of both sets covers a big part of Q. Then, the optimization does not produce any improvement and it only results in a time overhead due to the overlap search.

3.5 Multiple Sequences

When there are more than two lists, there are several possible heuristics depending on the list sizes. One possible algorithm is two process just pairs of sequences. For three lists the best solution will be to intersect the two shortest lists and then intersect the result with the longer list. For four lists or more the heuristic will depend on the partial answers, and hence has to be adaptive. In general, doing a balanced merging tree that avoids the long lists until the end will perform well. On the other hand, in practice we will not have more than 6 to 8 lists. Hence, if intersecting the two shortest lists gives a very small answer, might be better to intersect that to the next shortest list, and so on. In general the optimal algorithm will depend in the partial answers and hence we would need a dynamic programming algorithm to obtain it.

Other possibility is to extend our algorithm to the case of L lists. That is, we search the median of the shortest list on the other $L-1$ lists. If we find the median in the $L-1$ lists, we add that to the result. Next, we solve recursively for all the elements that are less than the median and for all the elements that are larger than the median. The complexity in this case is similar to the two sequence case using m as the length of the shortest list and n as the length of the rest of the lists.

4 Hybrid Algorithms

We can see from the experimental results obtained for the basic algorithm that there is a section of values of n where Merge is better than Intersect. Hence, a natural idea is to combine both algorithms in one hybrid algorithm that runs each of them when convenient. However this will depend on the data, the implementation, and the actual hardware and software platform used. So the following discussion is based on our context but can be replicated, possibly with different results, for other cases.

In order to know where is the cutting point to use one algorithm instead of the other, we measured for each value of n the time of both algorithms with different values of m until we identified the value of m where Merge was faster than Intersect. These values of m form a straight line as a function of n, which we can observe in Fig. 9. This straight line is approximated by $m = 0.033n + 8.884$, with a correlation of $r^2 = 0.999$.

The hybrid algorithm works by running Merge whenever $m > 0.033n + 8.884$, and running Intersect otherwise. The condition is evaluated on each step of the recursion.

When we modify the algorithm, the cutting point changes. We would like to find the optimal hybrid algorithm. Using the same idea again, we found the straight line that defines the values where Merge is better than the hybrid algorithm. This straight line can be approximated by $m = 0.028n + 32.5$, with $r^2 = 0.992$. Hence, we define the algorithm Hybrid2, which runs Merge whenever $m > 0.028n + 32.5$ and runs Intersect otherwise. Finally, we combined both

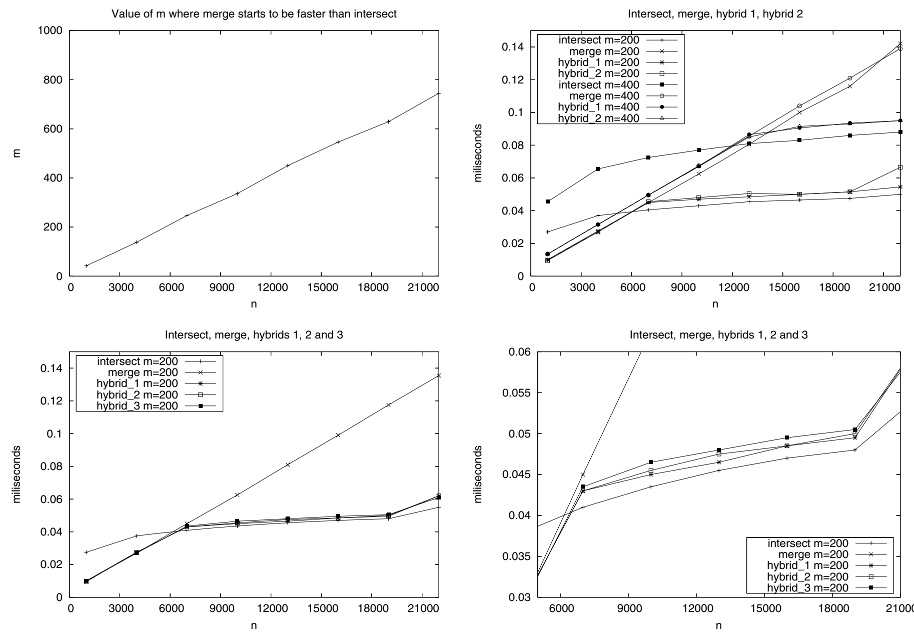

Fig. 9. Up: on the left, value of m from which Merge is faster than Intersect. On the right, a comparison between the original algorithm, Merge and the hybrids 1 and 2 for $m = 200$ and $m = 400$. Down: comparison between Intersect, Merge and the three hybrids for $m = 200$. The plot on the right is a zoom of the one on the left.

hybrids, creating a third version where the cutting line between Merge and Intersect is the average between the lines of the hybrids 1 and 2. The resulting straight line is $m = 0.031n + 20.696$. Figure 9 shows the cutting line between the original algorithm and Merge, and the results obtained with the hybrid algorithms. The optimal algorithm would be on theory the Hybrid.i when i tends to infinity, as we are looking for a fixed point algorithm.

We can observe that the hybrid algorithms registered lower times than the original algorithm in the section where the latter is slower than Merge. However, in the other section the original algorithm is faster than the hybrids, due to the fact that in practice we have to evaluate the cutting point in each step of the recursion. Among the hybrid algorithms, we can see that the first one is slightly faster than the second one, and that this one is faster than the third one. An idea to reduce the time in the section that the original algorithm is faster than the hybrids is to create a new hybrid algorithm that runs Merge when it is convenient and that then runs the original algorithm, without evaluating the relation between m and n in order to run Merge. This algorithm shows the same times than Intersect in the section where the latter is better than Merge, combining the advantages of both algorithms in the best way. Figure 10 show the results obtained with this new hybrid algorithm.

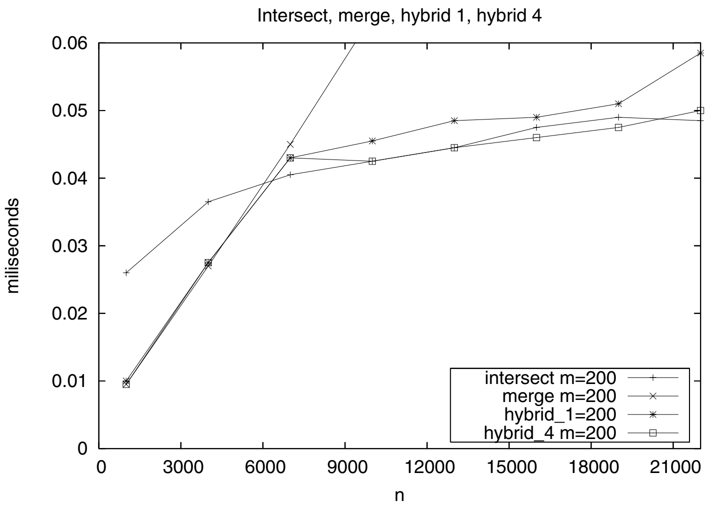

Fig. 10. Experimental results for Intersect, Merge and the hybrids 1 and 4 for different values of n and for $m = 200$

5 Application to Query Processing in Inverted Indexes

5.1 Context

Inverted indexes are used in most text retrieval systems [3]. Logically, they are a vocabulary (set of unique words found in the text) and a list of references per word to its occurrences (typically a document identifier and a list of word positions in each document). In simple systems (Boolean model), the lists are sorted by document identifier, and there is no ranking (that is, there is no notion of relevance of a document). In that setting, our basic algorithm applies directly to compute Boolean operations on document identifiers: union is equivalent to merging, intersection is the complement operation (we only keep the repeated elements), and subtraction implies deleting the repeated elements. In practice, long lists are not stored sequentially, but in blocks. Nevertheless, these blocks are large, and the set operations can be performed in a block-by-block basis.

In complex systems ranking is used. Ranking is typically based in word statistics (number of word occurrences per document and the inverse of the number of documents having it). Both values can be precomputed and the reference lists are then stored by decreasing intra-document word frequency order to have first the most relevant documents. Lists are then processed by decreasing inverse extra-document word frequency order (that is, we process the shorter lists first), to obtain first the most relevant documents. However, in this case we cannot always have a document identifier mapping such that lists are sorted by that order.

The previous scheme was used initially on the Web, but as the Web grew, the ranking deteriorated because word statistics do not always represent the content and quality of a Web page and also can be “spammed” by repeating and adding (almost) invisible words. In 1998, Brin and Page [11] described a search engine (which was the starting point of Google) that used links to rate the quality of a page. This is called a global ranking based in popularity, and is independent of the query posed. It is out of the scope of this paper to explain Pagerank, but it models a random Web surfer and the ranking of a page is the probability of the Web surfer visiting it. This probability induces a total order that can be used as document identifier. Hence, in a pure link based search engine we can use our intersection algorithm as before. However, nowadays hybrid ranking schemes that combine link and word evidence are used. In spite of this, a link based mapping still gives good results as approximates well the true ranking (which can be corrected while is being computed).

Another important type of query is sentence search. In this case we use the word position to know if a word follows or precedes a word. Hence, as usually sentences are small, after we find the Web pages that have all of them, we can process the first two words to find adjacent pairs and then those with the third word and so on. This is like to compute a particular intersection where instead of finding repeated elements we try to find correlative elements (i and $i+1$), and therefore we can use again our algorithm as word positions are sorted. The same is true for proximity search. In this case, we can have a range k of possible valid positions (that is $i \pm k$) or to use a different ranking weight depending on the proximity.

Finally, in the context of the Web, our algorithm is in practice much faster because the uniform distribution assumption is pessimistic. In the Web, the distribution of word occurrences is quite biased. The same is true with query frequencies. Both distributions follow a power law (a generalized Zipf distribution) [3,5]. However, the correlation of both distributions is very small [4]. That implies that the average length of the lists involved in the query are not that biased. That means that the average lengths of the lists, n and m, when sampled, will satisfy $n = \Theta(m)$ (uniform), rather than $n = m + O(1)$ (power law). Nevertheless, in both cases our algorithm makes an improvement. Now we study this case experimentally.

5.2 Sequence Lengths with Zipf's Distribution

Now we study the behavior of the Intersect algorithm depending of the ratio between the lengths of the two sequences when these lengths follow a Zipf distribution and the correlation between both sets is zero (ideal case). For this experiment, we took two random numbers, a and b, uniformly distributed between 0 and 1,000. With these numbers we computed the lengths of the sequences D and Q as $n = K/a^\alpha$ and $m = K/b^\alpha$, respectively, with $K = 10^9$ and $\alpha = 1.8$ (a typical value for word occurrence distribution in English), making sure that $n > m$. We did 1,000 measurements, using 80 different sequences for each of them, and repeating 1,000 times each run.

Figure 11 shows the times obtained with both algorithms as a function of n/m, in normal scale and logarithmic scale.

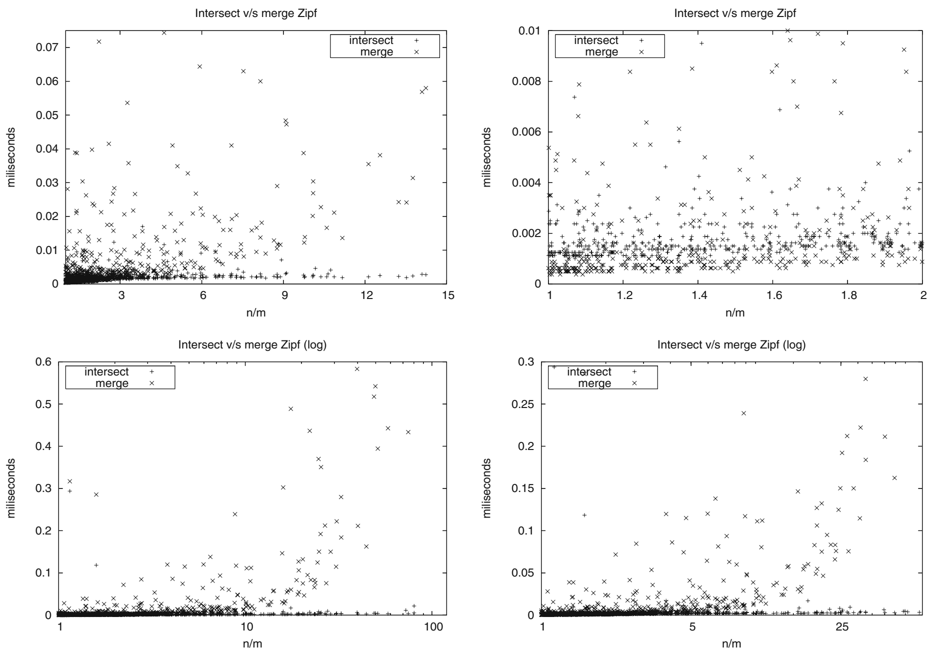

Fig. 11. Up: times for Intersect and Merge as a function of the ratio between the lengths of the sequences when they follow a Zipf distribution. The plot on the right is a zoom of the one on the left. Down: times for Intersect and Merge in logarithmic scale. The plot on the right is a zoom of the one on the left.

We can see that the times of Intersect are lower than the times of Merge when n is much greater than m. When we decrease the ratio between n and m, it is not so clear anymore which of the algorithms is faster. When $n/m < 2$, in most cases the times of Merge are better.

6 Concluding Remarks

We have presented a simple set intersection algorithm that performs quite well in average and does not inspect all the elements involved. It can be seen as a natural hybrid of binary search and merging. Our experiments show that the algorithm is also faster than Merge in practice when one of the sequences is much larger than the other one. This improvement is more evident when n increases. In addition, our algorithm surpasses Adaptive [13,14] for every ratio between the sizes of the sequences. The hybrid algorithm that combines our algorithm with Merge according to the empiric information obtained, takes advantage of both algorithms and became the most efficient one.

In practice, queries are short (on average 2 to 3 words [5]) so there is almost no need to do multiset intersection and if so, they can be easily handled by pairing the smaller sets firsts, which seems to be the most used algorithm [14].

In addition, we do not need to compute the complete result, as most people only look at less than two result pages [5]. Moreover, computing the complete result is too costly if one or more words occur several millions of times as happens in the Web and that is why most search engines use an intersection query as default. Hence, lazy evaluation strategies are used. If we use the straight classical merging algorithm, this naturally obtains first the most relevant Web pages. For our algorithm, it is not so simple, because although we have to process first the left side of the recursive problem, the Web pages obtained do not necessarily appear in the correct order. A simple solution is to process the smaller set from left to right doing binary search in the larger set. However this variant is efficient only for small m, achieving a complexity of $O(m \lg n)$ comparisons. An optimistic variant can use a prediction on the number of pages in the result and use an intermediate adaptive scheme that divides the smaller sets in non-symmetric parts with a bias to the left side. Hence, it is interesting to study the best way to compute partial results efficiently.

As the correlation between both sets in practice is between 0.2 and 0.6, depending on the Web text used (Zipf distribution with α between 1.6 and 2.0) and the queries (Zipf distribution with a lower value of α, for example 1.4), we would like to extend our experimental results to this case. However, we already saw that in both extremes (correlation 0 or 1), the algorithm studied is competitive.

References

1. Baeza-Yates, R.A.: Efficient Text Searching. PhD thesis, Dept. of Computer Science. University of Waterloo (May 1989); Also as Research Report CS-89-17
2. Baeza-Yates, R.A., Bradford, P.G., Culberson, J.C., Rawlins, G.J.E.: The Complexity of Multiple Searching (1993) (unpublished manuscript)
3. Baeza-Yates, R.A., Ribeiro-Neto, B.: Modern Information Retrieval, 513 pages. ACM Press/Addison-Wesley, England (1999)
4. Baeza-Yates, R.A., Saint-Jean, F.: A three level search engine index based in query log distribution. In: Nascimento, M.A., de Moura, E.S., Oliveira, A.L. (eds.) SPIRE 2003. LNCS, vol. 2857, pp. 56–65. Springer, Heidelberg (2003)
5. Baeza-Yates, R.A.: Query usage mining in search engines. In: Scime, A. (ed.) Web Mining: Applications and Techniques. Idea Group, USA (2004)
6. Baeza-Yates, R.A.: A fast set intersection algorithm for sorted sequences. In: Sahinalp, S.C., Muthukrishnan, S., Dogrusöz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 400–408. Springer, Heidelberg (2004)
7. Baeza-Yates, R.A., Salinge, A.: Experimental analysis of a fast intersection algorithm for sorted sequences. In: Consens, M.P., Navarro, G. (eds.) SPIRE 2005. LNCS, vol. 3772, pp. 13–24. Springer, Heidelberg (2005)
8. Barbay, J., Kenyon, C.: Adaptive Intersection and t-Threshold Problems. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, January 2002, pp. 390–399 (2002)
9. Barbay, J., López-Ortiz, A., Lu, T., Salinger, A.: An experimental investigation of set intersection algorithms for text searching. Journal of Experimental Algorithms (JEA) 14(3), 7–24 (2009)
10. Bentley, J.L., Yao, A.C.-C.: An Almost Optimal Algorithm for Unbounded Searching. Information Processing Letters 5, 82–87 (1976)

11. Brin, S., Page, L.: The anatomy of a large-scale hypertextual Web search engine. In: 7th WWW Conference, Brisbane, Australia (April 1998)
12. Culpepper, J., Moffat, A.: Compact set representation for information retrieval. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 137–148. Springer, Heidelberg (2007)
13. Demaine, E.D., López-Ortiz, A., Munro, J.I.: Adaptive set intersections, unions, and differences. In: Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, January 2000, pp. 743–752 (2000)
14. Demaine, E.D., López-Ortiz, A., Munro, J.I.: Experiments on adaptive set intersections for text retrieval systems. In: Buchsbaum, A.L., Snoeyink, J. (eds.) ALENEX 2001. LNCS, vol. 2153, pp. 91–104. Springer, Heidelberg (2001)
15. Dietz, P., Mehlhorn, K., Raman, R., Uhrig, C.: Lower Bounds for Set Intersection Queries. In: Proceedings of the 4^{th} Annual Symposium on Discrete Algorithms, pp. 194–201 (1993)
16. Dobkin, D., Lipton, R.: On the Complexity of Computations Under Varying Sets of Primitives. Journal of Computer and Systems Sciences 18, 86–91 (1979)
17. Fernandez de la Vega, W., Kannan, S., Santha, M.: Two probabilistic results on merging. SIAM J. on Computing 22(2), 261–271 (1993)
18. Fernandez de la Vega, W., Frieze, A.M., Santha, M.: Average case analysis of the merging algorithm of Hwang and Lin. Algorithmica 22(4), 483–489 (1998)
19. Hwang, F.K., Lin, S.: A Simple algorithm for merging two disjoint linearly ordered lists. SIAM J. on Computing 1, 31–39 (1972)
20. Rawlins, G.J.E.: Compared to What?: An Introduction the the Analysis of Algorithms. Computer Science Press/W.H. Freeman (1992)
21. Sanders, P., Transier, F.: Intersection in integer inverted indices. In: ALENEX 2007, pp. 71–83 (2007)