An Experimental Investigation of Set Intersection Algorithms for Text Searching

JÉRÉMY BARBAY
Universidad de Chile
and
ALEJANDRO LÓPEZ-ORTIZ, TYLER LU, and ALEJANDRO SALINGER
University of Waterloo

The intersection of large ordered sets is a common problem in the context of the evaluation of boolean queries to a search engine. In this article, we propose several improved algorithms for computing the intersection of sorted arrays, and in particular for searching sorted arrays in the intersection context. We perform an experimental comparison with the algorithms from the previous studies from Demaine, López-Ortiz, and Munro [ALENEX 2001] and from Baeza-Yates and Salinger [SPIRE 2005]; in addition, we implement and test the intersection algorithm from Barbay and Kenyon [SODA 2002] and its randomized variant [SAGA 2003]. We consider both the random data set from Baeza-Yates and Salinger, the Google queries used by Demaine et al., a corpus provided by Google, and a larger corpus from the TREC Terabyte 2006 efficiency query stream, along with its own query log. We measure the performance both in terms of the number of comparisons and searches performed, and in terms of the CPU time on two different architectures. Our results confirm or improve the results from both previous studies in their respective context (comparison model on real data, and CPU measures on random data) and extend them to new contexts. In particular, we show that value-based search algorithms perform well in posting lists in terms of the number of comparisons performed.

Categories and Subject Descriptors: F.2.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity—*Nonnumerical Algorithms and Problems: Computations on discrete structures*

General Terms: Algorithms, Design, Experimentation, Performance, Theory

Additional Key Words and Phrases: Database queries, set intersection

A preliminary version of this paper appeared in [Barbay et al. 2006].
First author's address: Departamento de Ciencias de la Computación, Universidad de Chile, Av. Blanco Encalada 2120, Tercer Piso, Santiago, C.P. 837-0459, Chile.
Second, third and fourth authors' address: David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada.

DOI 10.1145/10.1145/1498698.1564507 http://doi.acm.org/10.1145/10.1145/1498698.1564507

ACM Reference Format:
Barbay, J., López-ortiz, A., Lu, T., and Salinger, A. 2009. An experimental investigation of set intersection algorithms for text searching. ACM J. Exp. Algor. 14, Article 3.7 (July 2009), 24 pages. DOI = 10.1145/10.1145/1498698.1564507 http://doi.acm.org/10.1145/10.1145/1498698.1564507

1. INTRODUCTION

The intersection of large ordered sets is a common problem in the context of the evaluation of relational queries to databases, as well as boolean queries to a search engine. The worst-case complexity of this problem has long been well understood, dating back to the algorithm by Hwang and Lin from over 3 decades ago [Hwang and Lin 1971, 1972], and the average case has been studied in the case of the intersection of two sets, when the elements are uniformly distributed [de la Vega et al. 1998].

In 2000, Demaine et al. [2000] introduced a new intersection algorithm, termed `Adaptive`, which intersects all the sets in parallel so as to compute the intersection in time proportional to the shortest proof of the result set. In a subsequent study [Demaine et al. 2001], they compared its performance in practice, relative to a straightforward implementation of an intersection algorithm, and proposed a new and better adaptive algorithm, which outperformed both in practice. They measured the number of comparisons performed on the index of a collection of plain text from web pages. In 2002, Barbay and Kenyon [2002] introduced another intersection algorithm, which adapts to the correlation between the terms of the query, and one year later, Barbay [2003] introduced a randomized variant. To the best of our knowledge, neither of these algorithms were implemented before our study. In 2004, Baeza-Yates [2004] introduced an intersection algorithm based on an alternative technique. Baeza-Yates and Salinger [2005] measured the performance of the algorithm in terms of CPU time, on pairs of random arrays.

In this article, we consider the number of comparisons and searches performed, as well as the CPU time on two different architectures (RISC and CISC), on three different data sets: (i) a random data set similar to the one considered by Baeza-Yates and Salinger [2005], (ii) the query log used by Demaine et al. [2001] on a larger data set provided by Google, and (iii) the GOV2 corpus, of size 361GB, with a larger query log, both from the TREC Terabyte 2006 efficiency query stream. This combines the previous studies and allows us to compare all the aforementioned algorithms on common platforms. We propose several variants for the intersection and search in sorted arrays in the context of their intersection:

—We propose a variant of the algorithm from [Baeza-Yates 2004], which performs the intersection of more than two sorted arrays without sorting the intermediary results. This variant is significantly faster than the original algorithm on real instances, both in terms of the number of comparisons performed and in terms of CPU time.

—We reduce the number of comparisons performed by each intersection algorithm by introducing value-based search algorithms, and we further improve their performance by introducing an adaptive value-based search algorithm.

—We show that a variant of binary search optimizes cache usage over the original version, when the arrays are too large to fit in memory.

The article is structured as follows: In Section 2, we describe the data sets and the architectures on which we evaluated the various algorithms discussed. In Section 3, we describe in detail the intersection and search algorithms studied. In Section 4, we present and analyze our experimental measures in the various contexts. We conclude in Section 5 with a summary of our experiments.

2. EXPERIMENTAL SET-UP

In this article, we measure the performance of the algorithms from [Demaine et al. 2001], from [Barbay and Kenyon 2002] and from [Baeza-Yates 2004], which were previously studied in different contexts (random or practical) and under different measures (CPU or number of comparisons), so they had not, until now, been directly compared. We perform this comparison under each of the previous settings, as well as using a larger corpus, on which the performance of algorithms is more sensitive to cache effects.

2.1 Data Sets

2.1.1 *Random, Uniformly Distributed Data.* We compare the performance of the algorithms on pairs of sorted sets generated in the same way as Baeza-Yates and Salinger [2005]: sequences of integer random numbers, uniformly distributed in the range $[1, 10^9]$. The length n of the longest sequence varies from 1,000 to 22,000, by steps of 3,000. The length m of the shortest sequence varies from 100 to 400, by steps of 100.

For each algorithm and each pair of sizes (n, m), we generate 20 instances. We measure the number of comparisons once for each algorithm and instance, and we average the running time over 1,000 executions. Each execution, for a given combination of algorithm and instance, is separated from the next one with the same combination by the execution of all the other algorithms on all the instances. This ensures a realistic simulation of the cache behavior.

2.1.2 *Google Corpus and Query Log.* We compare the performance of the intersection algorithms to answer real queries on a sample web corpus, both provided by Google. This is the same query log used by Demaine et al. [2001], but on a substantially larger and more recent data set.

The set of Web pages contains 678,760 text documents in 6.85GB of text. As the documents or web pages of the corpus were not given a numerical identifier *a priori*, we numbered the documents as they were stored, by assigning them a sequential number indicating their order in the indexing process. The resulting inverted word index has 2,604,335 alphanumeric keywords with HTML markup removed.

Table I. The distribution of the sizes of TREC queries: On average, 4.42 keywords per query

# of keywords (k)	1	2	3	4	5	6	7	8	9
# of queries	105	778	12,66	12,17	793	414	198	98	53
# of keywords (k)	10	11	12	13	14	15	16	17	18
# of queries	44	14	7	4	5	2	0	1	1

The query log corresponds to 5,000 entries. For more details on the query log, we refer the reader to [Demaine et al. 2001], where its properties are discussed in detail.

2.1.3 *TREC GOV2 Corpus and Query Log.* We consider a larger Web corpus and an associated query log, which form the data set TREC GOV2. This Web corpus was collected by the TREC competition in information retrieval, through a partial crawl of U.S. government Web sites.

The GOV2 Web corpus corresponds to approximately 361GB of text, which once indexed associates 38,515,138 keywords to the references of 25,197,524 documents. Each document is on average 13.37KB long, most are in HTML, but some are in PDF. The document numbering scheme is such that certain groups of documents have numbers close to each other. As a result, this creates gaps in the numbering scheme where certain numbers between document groups do not appear.

The query log provided with the TREC GOV2 corpus corresponds to 100,000 queries with click-through to `.gov` domains. We randomly selected a sample of 5,000 queries for our simulations. There were 105 queries involving only one keyword, and 305 queries where a keyword did not appear in the inverted word index. This leaves 4,590 nontrivial queries, which corresponds to a query log of similar size to the one used on the Google data set. The average size of a query is 4.42 keywords. Table I shows the number of keywords distribution in the queries: Most queries have less than 11 keywords.

2.2 Machines and Compilers

We implemented the algorithms in `C++`, and we ran our experiments on two architectures. For each architecture, we measured only the performance of the intersection on sorted arrays once they have been loaded in memory (and eventually cached on the swap partition of the hard drive). In particular, we did not measure the performance of the indexing structure, which retrieves those arrays from the index on the hard drive.

2.2.1 *The* `INTEL` *platform.* For all data sets, we used a PC running `Linux version 2.4.20-31.9` on a processor `Intel Pentium 4`, at 2.66GHz with a low level 1 cache of $8K$, a level 2 cache of 512K, 1GB of memory, and a swap partition of size 4.16GB. We measured the CPU time using the `rdtscl` function, specific to the Pentium, which measures the number of processor cycles, and hence includes the time taken by hard-drive accesses to the swapped partition, and by cache misses. The programs were compiled on this machine using `gcc 3.2.2` with the optimization option `-O3`.

For the largest data set, we also measured the CPU time using the `times` function, from the `sys/times.h` library, to allow the comparison with the equivalent measures on the other platform, which does not support the `rdtscl` function.

2.2.2 *The* `SUN` *platform.* For very large instances, we ran additional simulations using an `UltraSparc III` server from Sun running Unix on 8 processors at 900MHz, with 16GB of RAM. As the largest sorted array uses 216MB, and as each instance is composed of at most 18 arrays, no instance uses more than 4GB, hence all intersection instances hold in main memory on this machine. This is a RISC architecture, which means, in particular, that certain multiplications and divisions may not be directly supported by the processor but computed through function calls.

The CPU time was measured on this machine using the `times` function from the `sys/times.h` library, which returns the elapsed real time, including time taken by cache misses. The programs were compiled on this machine using `gcc 2.95.2` with the optimization option `-O3`.

3. ALGORITHMS

In this article, we define search and melding algorithms separately so that we can study the impact of new search algorithms on all melding algorithms and find the best combination over all possible ones.

3.1 Melding Algorithms

Various algorithms for the intersection of k sets have been introduced in the literature [Barbay and Kenyon 2002, Demaine et al. 2000, 2001, Baeza-Yates 2004; Baeza-Yates and Salinger 2005; Barbay 2003]. Among those, we do not consider the naïve algorithm, which traverses each array linearly, as both theoretical and experimental analysis show that its performance in the comparison model is significantly worse than the ones studied here. For similar reasons, we do not consider either the `Adaptive` intersection algorithm, proposed by Demaine et al. [2000], or the algorithm proposed by Hwang and Lin [Demaine et al. 2001]. Instead, we focus on four main algorithms, some of them with minor variants.

3.1.1 *SvS and Swapping SvS.* `SvS` is a straightforward algorithm widely used, which intersects the sets two at a time in increasing order by size, starting with the two smallest (see Algorithm 1). It performs a binary search to determine if an element in the first set appears in the second set. We also consider variants of it, which replace the binary search with various other searches.

Demaine et al. considered the variant `Swapping_SvS`, where the searched element is picked from the set with the least remaining elements, instead of the first (initially smallest) set in `SvS`. This algorithm was first proposed by Hwang and Lin [1971]: It performs better when the size of the second set is substantially reduced after a search although experiments show that this does not happen often.

Algorithm 1. Pseudocode for `SvS`

```
SvS(set, k)
Sort the sets by size (|set[0]| ≤ |set[1]| ≤ ... ≤ |set[k]|).
Let the smallest set set[0] be the candidate answer set.
for each set S from set do initialize ℓ[S] = 0.
for each set S from set do
    for each element e in the candidate answer set do
        search for e in S in the range ℓ[S] to |S|,
        and update ℓ[S] to the rank of e in S.
        If e was not found then
            remove e from candidate answer set,
            and advance e to the next element in the answer set.
        end if
    end for
end for
```

Algorithm 2. Pseudocode for `Small_Adaptive`

```
Small_Adaptive(set, k)
while no set is empty do
    Sort the sets by increasing number of remaining elements.
    Pick an eliminator e = set[0][0] from the smallest set.
    elimset ← 1.
    repeat
        search for e in set[elimset].
        increment elimset.
    until s = k or e is not found in set[elimset]
    if s = k then
        add e to answer.
    end if
end while
```

3.1.2 *Small Adaptive.* `Small_Adaptive` is a hybrid algorithm, which combines the best properties of `SvS` and `Adaptive` (see Algorithm 2). For each element in the smallest set, it performs a galloping search on the second smallest set. If a common element is found, a new search is performed in the remaining $k-2$ sets to determine if the element is indeed in the intersection of all sets, otherwise a new search is performed. Observe that the algorithm computes the intersection from left to right, producing the answer in increasing order. After each step, each set has an already examined range and an unexamined range. `Small_Adaptive` selects the two sets with the smallest unexamined range and repeats the process described above until there is a set that has been fully examined.

3.1.3 *Sequential and Random Sequential.* Barbay and Kenyon [2002] introduced a fourth algorithm, called `Sequential`, which is optimal for a different measure of difficulty, based on the nondeterministic complexity of the instance. It cycles through the sets, performing one entire gallop search at a time in each (as opposed to a single galloping *step* in `Adaptive`), so that it performs at

Algorithm 3. Pseudocode for `Sequential`

```
Sequential(set, k)
 1: Choose an eliminator e = set[0][0], in the set elimset ← 0.
 2: Consider the first set, i ← 1.
 3: while the eliminator e ≠ ∞ do
 4:     search in set[i] for e.
 5:     if the search found e then
 6:         increase the occurrence counter.
 7:         if the value of occurrence counter is k then output e end if
 8:     end if
 9:     if the value of the occurrence counter is k, or e was not found then
10:         update the eliminator to e ← set[i][succ(e)].
11:     end if
12:     Consider the next set in cyclic order i ← i + 1 mod k.
13: end while
```

most k searches for each comparison performed by an optimal nondeterministic algorithm: Its pseudocode is given in Algorithm 3.

A randomized variant [Barbay 2003], `RSequential`, performs less comparisons than `Sequential` on average on instances where the searched elements are present in roughly half of the arrays, rather than in almost all or almost none of the arrays. The difference with `Sequential` corresponds to a single line, the choice of the next set where to search for the "eliminator" (line 12 in Algorithm 3): `Sequential` takes the next set available while `RSequential` chooses one at random among all the sets not yet known to contain the eliminator.

3.1.4 *Baeza-Yates and Baeza-Yates Sorted.* `BaezaYates` algorithm was originally intended for the intersection of two sorted lists. It takes the median element of the smaller list and searches for it in the larger list. The element is added to the result set if present in the larger list. The median of the smaller list and the rank insertion of the median in the larger set divide the problem into two subproblems. The algorithm solves recursively the instances formed by each pair of subsets, always taking the median of the smaller subset and searching for it in the larger subset. If any of the subsets is empty, it does nothing. In order to use this algorithm on instances with more than two lists, Baeza-Yates [2004] suggests to intersect the lists two-by-two, intersecting the smallest lists first. As the intersection algorithm works for sorted lists and the result of the intersection may not be sorted, the result set needs to be sorted before intersecting it with the next list, which would be highly inefficient. The pseudocode for `BaezaYates` algorithm is shown in Algorithm 4.

To avoid the cost of sorting each intermediate result set, we introduce `So_BaezaYates`, a minor variant of `BaezaYates`, which does not move the elements found from the input to the result set as soon as it finds them, but only at the last recursive step. This ensures that the elements are added to the result set in order and trades the cost of explicitly sorting the intermediate results with the cost of keeping slightly larger subsets.

Algorithm 4. Pseudocode for `BaezaYates`

```
BaezaYates(set, k)
  1: Sort the sets by size (|set[0]| ≤ |set[1]| ≤ ... ≤ |set[k]|).
  2: Let the smallest set set[0] be the candidate answer set.
  3: for each set set[i], i = 1...k do
  4:     candidate ← BYintersect(candidate, set[i], 0, |candidate| − 1, 0, |set[i]| − 1)
  5:     sort the candidate set.
  6: end for

BYintersect(setA, setB, minA, maxA, minB, maxB)
  1: if setA or setB are empty then return Ø endif.
  2: Let m = (minA + maxA)/2 and let medianA be the element at setA[m].
  3: Search for medianA in setB.
  4: if medianA was found then
  5:     add medianA to result.
  6: end if
  7: Let r be the insertion rank of medianA in setB.
  8: Solve the intersection recursively on both sides of r and m in each set.
```

Each of those algorithms has linear time worst-case behavior in the sum of the sizes of the arrays, and each performs better than the others on a set of instances. Note that `BaezaYates`, `So_BaezaYates`, `Small_Adaptive`, and `SvS` take active advantage of the difference of sizes of the sets, and that `Small_Adaptive` is the only one that takes advantage of how this size varies as the algorithm eliminates elements, while `Sequential` and `RSequential` ignore this information.

3.2 Search Algorithms

We extend the set of search algorithms tested to value-based algorithms, such as `Interpolation`, `Extrapolation`, or `Extrapol_Ahead` and to some cache-oblivious search algorithms, such as `Rounded_Binary`.

3.2.1 *Binary Search and Variants.* Binary search is well known in the literature. The adequate implementation[1] finds the insertion rank p of a key x in a sorted set A of size n in $1 + \log_2 n$ comparisons. In the context of the intersection of sorted arrays, several elements are searched in each array, and in many applications those elements are of increasing size, so that the position of the last lookup during the previous search is a lower bound for the position of the currently searched element. While using this lower bound reduces the number of comparisons (we call this `Adaptive_Binary`), it yields a slower CPU performance when the array is very large and partially cached. `Total_Binary` ignores this lower bound and uses the cache more efficiently.

[1]It can be implemented in two different ways, each of them optimizing a different performance measure, the number of two-way comparisons, closer to CPU time, and the number of three-way comparisons, closer to the running time in the context of hierarchical memory. As the latter implementation performed poorly on all contexts, we discuss here only the one optimizing the number of two-way comparisons.

We test a third variant, `Rounded_Binary`, which represents a trade-off between `Adaptive_Binary` and `Total_Binary`: It performs the same comparisons as `Total_Binary` so long as the compared elements are larger as the lower bound obtained from the previous search, at which point it switches to a more sophisticated mode taking advantage both of the positions of the previous comparisons and of the lower bound. This variant always performs more comparisons than `Adaptive_Binary`, and less than `Total_Binary`, but it performs better in terms of CPU on instances where the array searched is very large due to cache effects.

3.2.2 *Galloping Search.* Originally introduced by Bentley and Yao [1976], unbounded search is the problem of searching for the insertion rank p of a key x in a sorted set A of unbounded size. The algorithm probes the i keys with index $\{1, 3, 7, 15, \ldots, 2^i - 1\}$ in sequence till it finds a key $A[2^i - 1]$ larger than x, and then performs a binary search in A between positions $2^{i-1} - 1$ and $2^i - 1$. This technique is sometimes called one sided binary search [Skiena 1997], exponential search [Chen 2003], doubling search [Barbay and Kenyon 2002], or galloping [Demaine et al. 2000, 2001]: We will use this last name for our implementation, `Galloping` search. It solves the unbounded search problem in $2\log_2(p+1)$ comparisons.

3.2.3 *Interpolation and Extrapolation Search.* `Interpolation` search has long been known to perform significantly better in terms of comparisons over binary search on data randomly drawn from a uniform distribution, and recent developments suggest that interpolation search is also a reasonable technique for nonuniform data [Demaine et al. 2004]. Searching for an element of value e in an array $\texttt{set}[i]$ on the range a to b, the algorithm probes position $I(a, b, e)$ defined as follows:

$$I(a, b, e) = \left\lfloor \frac{e - \texttt{set}[i][\texttt{a}]}{\texttt{set}[i][b] - \texttt{set}[i][\texttt{a}]}(b - a) \right\rfloor + a$$

We propose a variant, `Extrapolation` search, which involves extrapolating on the current and previous positions in `set[i]`. Specifically, the extrapolation step probes the index $I(p_i', p_i, e)$, where p_i' is the previous extrapolation probe. This has the advantage of using "explored data" as the basis for calculating the expected index: This strategy is similar to galloping, which uses the previous jump value as the basis for the next jump (i.e., the value of the next jump is the double of the value of the current jump).

3.2.4 *Extrapolation Look Ahead Search.* We propose another search algorithm, `Extrapol_Ahead`, which is similar to extrapolation, but rather than basing the extrapolation on the current and previous positions, we base it on the current position and a position that is further ahead. Thus, our probe index is calculated by $I(p_i, p_i{+}l, e)$ where l is a positive integer that essentially measures the degree to which the extrapolation uses local information. The algorithm uses the local distribution as a representative sample of the distribution between $\texttt{set}[i][p_i]$ and the eliminator: A large value of l corresponds to an algorithm using more global information, while a small value of l correspond to an algorithm using only local information. If the index of the successor succ(e) of e in set[i] is not far from p_i, then the distribution between set[i][p_i] and set[i][$p_i + l$] is expected to be similar to the distribution between set[i][p_i] and set[i][succ(e)] and the estimate will be fairly accurate. Thus, if the set is bursty, or piecewise uniform, we would expect this strategy to outperform interpolation because the set is locally representative. On the other hand, if the set comes from a random uniform distribution, then we would expect interpolation to be better because, in this case, using a larger range to interpolate is more accurate than using a smaller range.

4. EXPERIMENTAL RESULTS

In each of the contexts defined in Section 2, we test all the algorithms defined in Section 3, and we measure their performance in terms of the number of searches and comparisons performed, and in terms of CPU time. The CPU times for the Random and Google data sets correspond only to measures on the INTEL platform, as the instances are too small for the execution time to be measured on the SUN platform. Both platforms are considered for the larger TREC GOV2 data set.

Note that the number of searches for a fixed merging algorithm does not depend on which search algorithm is used (they all return the same position), and that the number of comparisons performed does not depend on the architecture. Despite the fact that the CPU time on a particular instance can slightly vary from one execution to another, we verified on small samples (50 queries from the TREC data set, all queries from the Google data set) that the CPU measures over a single run yield the same conclusion than averaging over 50 runs: Hence, we report our results on larger samples with a single run.

4.1 Experiments on Random, Uniformly Distributed Data

In the context of randomly generated data, we only measure the performance of the algorithms with two lists, in a similar way to the study by Baeza-Yates and Salinger [2005], which compare the CPU performance on random data of the combinations BaezaYates using Adaptive_Binary, Small_Adaptive using Galloping, and of the naïve linear algorithm; BaezaYates using Adaptive_Binary was the best combination. We test a larger set of algorithms, on random data generated in a similar way, and we measure both the performance in CPU time and the number of comparisons and searches. Note that RSequential behaves exactly the same as Sequential on two arrays and thus is not represented.

We show on the plots the number of comparisons and CPU times for different intersection and search algorithms as a function of the size n of the largest list when the size of the smallest list m is fixed, for various values of m. The standard deviation is usually very low, hence we omit in the figures with more than two plots on them.

4.1.1 *Comparison with Baeza-Yates and Salinger.* In terms of CPU time, our results agree with Baeza-Yates and Salinger's study [2005]: Both

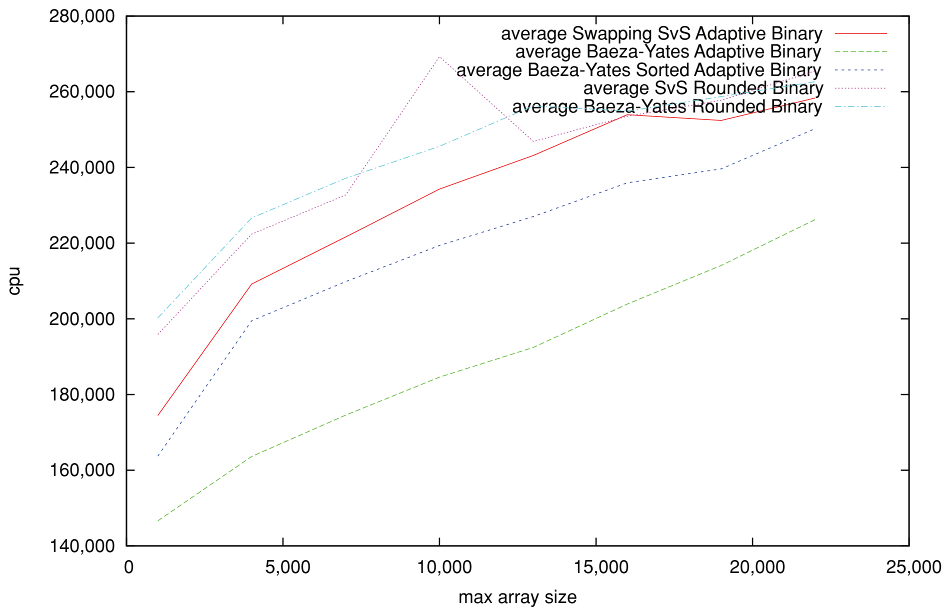

Fig. 1. CPU times for the five best combinations of algorithms on random generated instances. `BaezaYates` using `Adaptive_Binary` performs the best for all size ratios, closely followed by `Swapping_SvS` and `SvS` using `Galloping`.

`BaezaYates` and `So_BaezaYates` using `Adaptive_Binary` outperform any other combination of algorithms. Figure 1 shows the performance of the five best combinations of algorithms on this data set. As Figure 2 shows, none of the other search algorithms perform better than the initial choice proposed by Baeza-Yates and Salinger.

The superiority of `Adaptive_Binary` over all search algorithms when using `BaezaYates` or `So_BaezaYates` is easily explained: Value-based search algorithms, such as `Interpolation`, are too costly in CPU time, and adaptive search algorithms, such as `Galloping` or `Extrapol_Ahead`, are inefficient when the searched position is in the middle of the array, on average. The superiority of `BaezaYates` among melding algorithms is relative, as `SvS` and `Swapping_SvS` perform well for almost any search algorithm. The difference in CPU performance between `BaezaYates` and `So_BaezaYates` using `Adaptive_Binary`, `SvS`, `Swapping_SvS`, or `Small_Adaptive` using `Galloping` is minimal (see Table II).

4.1.2 *Number of Searches and Comparisons.* In terms of the number of searches, `BaezaYates`, `SvS`, `Swapping_SvS`, and `Small_Adaptive` perform the best, while `Sequential` and `So_BaezaYates` perform much more searches (see Table II). The difference of performance between `BaezaYates` and `So_BaezaYates` is easily explained: `BaezaYates` performs one more comparison per search to reduce the domain by one more value, which increases the number of comparisons but reduces the number of searches in comparison to `So_BaezaYates`. The difference of performance between `Sequential` and the other algorithms is due to the fact that `Sequential` always chooses the new

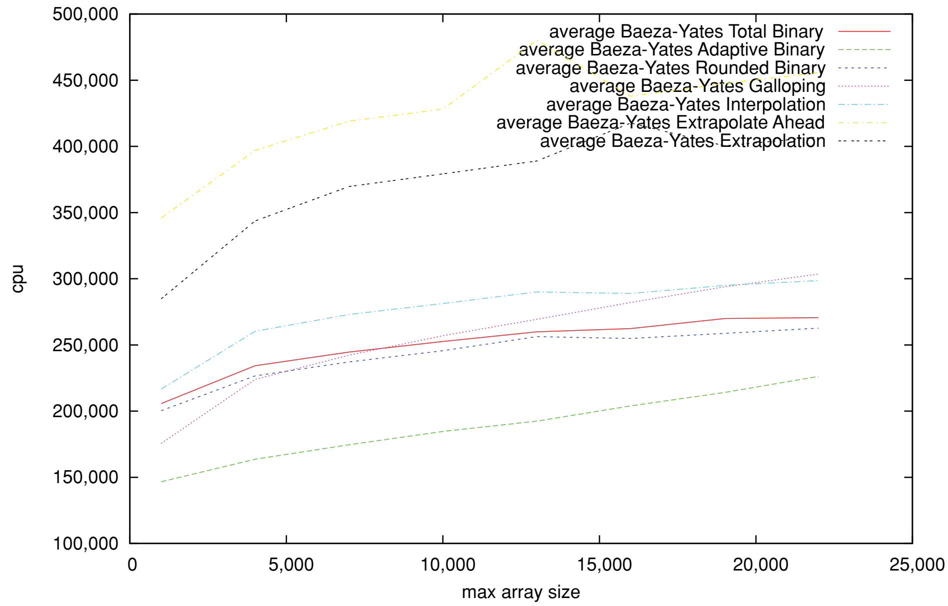

Fig. 2. CPU times for all search algorithms in combination with BaezaYates. The best search algorithm is the one proposed originally, Adaptive_Binary.

Table II. Total number of searches and comparisons and total running time performed by each algorithm on the Random data set, when associated with the search algorithm performing the best with it. The number of searches and comparisons are correlated, although the difference in terms of the number of searches performed between BaezaYates and So_BaezaYates does not correspond to the difference in the number of comparison performed. The CPU times are not correlated with the two other measures

Algorithm	Searches	Comparisons		Runtime	
SvS	200	1,024	(Extrapol_Ahead)	242,986	(Rounded_Binary)
Swapping_SvS	200	1,024	(Extrapol_Ahead)	230,916	(Adaptive_Binary)
Small_Adaptive	200	1,024	(Extrapol_Ahead)	435,828	(Galloping)
BaezaYates	199	1,066	(Interpolation)	188,258	(Adaptive_Binary)
So_BaezaYates	328	1,064	(Interpolation)	218,156	(Adaptive_Binary)
Sequential	385	1,198	(Extrapol_Ahead)	327,075	(Adaptive_Binary)

eliminator on the array previously searched: In the context where the elements of the array are uniformly drawn and of very different size, it always results in a worse performance than choosing the eliminator from the smallest array.

In terms of the number of comparisons, the use of value-based search algorithms, such as Interpolation, Extrapolation, or Extrapol_Ahead, results in a better performance for any melding algorithm: Those algorithms outperform other search algorithms on the uniform distribution of elements in the arrays.

The best combinations regarding the number of comparisons performed are Swapping_SvS using Extrapol_Ahead and BaezaYates using Interpolation, even though Figure 3 shows that Swapping_SvS with Extrapol_Ahead has a small advantage over BaezaYates with Interpolation.

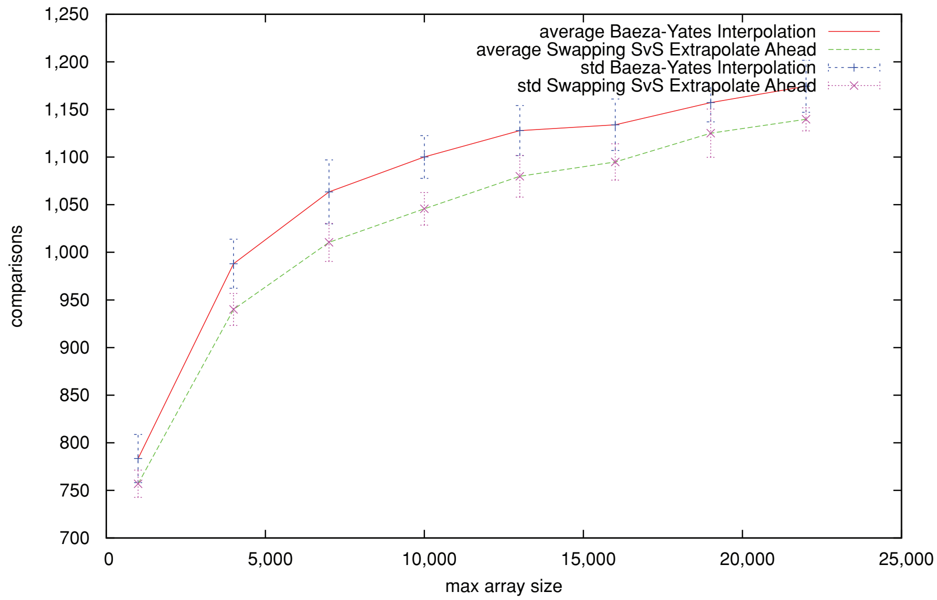

Fig. 3. Number of comparisons for `BaezaYates` using `Interpolation` and `Swapping_SvS` using `Extrapol_Ahead` on the Random data set. `Swapping_SvS` with `Extrapol_Ahead` performs visibly better.

Fixing the size of the smallest list to other values does not alter the relative ranking (see Figure 4), so we only report the data for $m = 200$. For completeness, we summarize the results across all algorithms on the whole random data set in Table III.

4.2 Experiments on the Google Data Set

Demaine et al. [2001] studied the combinations of algorithms `Small_Adaptive` using `Galloping`, `SvS`, and `Swapping_SvS` using `Adaptive_Binary`, and found the combination `Small_Adaptive` using `Galloping` to outperform the others in terms of the number of comparisons performed on a set of queries provided by Google on the index of their own Web-crawl.

We measured the performance of each combinations of algorithms on the same queries, but on the index of a larger Web crawl, also provided by Google. Similar to the results given by Demaine et al., we show on the plots the number of comparisons and CPU times as a function of the number k of keywords in the queries, which corresponds to the number of arrays forming the instance. The standard deviation of the two by two difference of performance on each instance, not represented here, was always very low. We omit the standard deviation of the average performance of each algorithm on instances composed of k arrays: it mostly represents the variation of difficulty among queries with k keywords, and not the stability of the results.

4.2.1 *Comparison with Demaine et al.* Considering the same algorithms studied by Demaine et al. [2001], our results agree with the previous study:

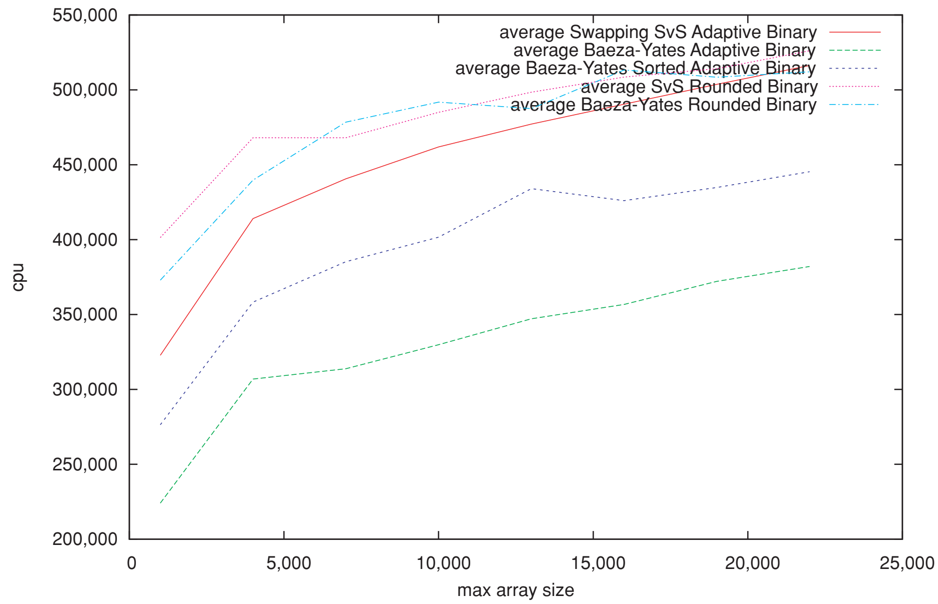

Fig. 4. CPU times for the five best combinations of algorithms on the Random data set with the smallest list of size 400. The order of the algorithms is the same than when the smallest list has size 200: `BaezaYates` using `Adaptive_Binary` performs the best for all size ratios.

`Small_Adaptive` using `Galloping` performs less comparisons than the other algorithms, but in fact, `Small_Adaptive` does not behave much differently from `SvS` and `Swapping_SvS`, as the combinations `SvS` using `Galloping` and `Swapping_SvS` using `Galloping` performs almost equally: The improvement in the number of comparisons performed is mainly due to the usage of the `Galloping` search algorithm (see Figure 5). This similarity of performance is likely to come from the fact that there are 2.286 keywords per query on average: `SvS`, `Swapping_SvS`, and `Small_Adaptive` behave the same on instances which consist of only two arrays.

The number of comparisons performed is further reduced by the use of value-based search algorithms. All intersection algorithms benefit from the use of `Interpolation`, and all except `BaezaYates` and `So_BaezaYates` benefit even more from the use of `Extrapol_Ahead`, the interpolation search variant that we introduced (see Figure 5). As a result, the best combination of search and melding algorithms regarding the number of comparison performed are `Small_Adaptive`, `SvS` and `Swapping_SvS` using `Extrapol_Ahead`, and results in an important improvement over the best solution proposed by Demaine et al.

4.2.2 *Study of Barbay and Kenyon's algorithm.* The algorithm proposed by Barbay and Kenyon [2002] and its randomized variant [Barbay 2003] both perform noticeably more comparisons than the other intersection algorithms measured, independently of the search algorithm chosen (see Table IV). This high number of comparisons is correlated with the high number of searches

Table III. Total number of comparisons and CPU times performed by each algorithm over the Random data set. In bold, the best performance in terms of the number of comparisons, for various melding algorithms in combination with Extrapol_Ahead, and the best performance in terms of CPU: BaezaYates using Adaptive_Binary

Comparisons	SvS	Swapping_SvS	Seq.	BY	So_BY	Small_Adapt.
Total_Binary	2,815	2,815	4,397	2,811	4,501	2,815
Adaptive_Binary	2,469	2,469	2,632	1,620	1,620	2,469
Rounded_Binary	2,623	2,623	3,997	2,629	4,190	2,623
Galloping	2,087	2,087	2,237	2,410	2,373	2,087
Interpolation	1,067	1,067	1,242	1,066	1,064	1,067
Extrapolation	1,281	1,281	1,444	1,261	1,262	1,281
Extrapol_Ahead	**1,024**	**1,024**	1,198	1,085	1,073	**1,024**
CPU	**SvS**	**Swapping_SvS**	**Seq.**	**BY**	**So_BY**	**Small_Adapt.**
Total_Binary	262,397	254,008	457,540	250,018	402,544	677,318
Adaptive_Binary	255,064	230,916	327,075	**188,258**	218,156	444,476
Rounded_Binary	242,986	246,871	436,438	242,773	391,347	443,064
Galloping	245,333	244,216	332,311	255,945	286,040	435,828
Interpolation	279,127	280,624	374,779	275,463	304,616	466,446
Extrapolation	375,585	371,444	464,203	373,947	401,933	547,751
Extrapol_Ahead	413,209	404,841	576,109	426,452	506,075	584,941

performed: The algorithms fails to find a shorter proof by cycling through the arrays.

The searches performed by Sequential are shorter on average than other similar algorithms: The ratio between the number of comparisons and the number of searches is even smaller than for other algorithms, such as SvS (see Table IV). This is probably explained by the fact that Sequential performs many searches of average size, as opposed to algorithms such as SvS, which perform many small searches in the smallest arrays, but a few rather large ones in the other arrays.

Note that the number of comparisons (and ratio) of BaezaYates and So_BaezaYates using Galloping is not representative: When using Adaptive_Binary search, which is better suited to their behavior, the performance in terms of the number of comparisons is much better (see Table IV). The melding algorithm So_BaezaYates is more efficient in terms of the number of comparisons than BaezaYates, although it performs more searches, which still results in a slightly smaller number of comparisons per searches: This corresponds to the additional comparison performed by BaezaYates to check if the searched element is present in the searched array.

4.2.3 *Real Time on Real Data.* The CPU performance is correlated to the number of comparisons for all melding and search algorithms, except for the value-based search algorithms (see Figure 6). The fact that Interpolation generally performs more comparisons than Extrapol_Ahead (see Table V) but uses less CPU time indicates that the cost of the extra memory accesses performed by Extrapol_Ahead is more significant than the reduction in the number of comparisons: It might result in an additional cache miss, since it is at distance $\lg n$ of the previous access, where n is the number of remaining element in the array.

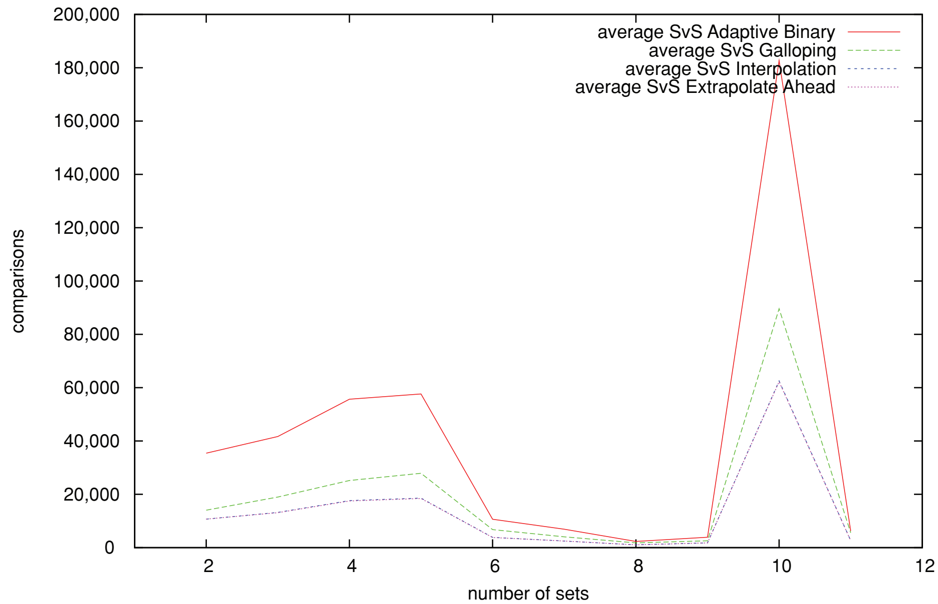

Fig. 5. Number of comparisons for SvS using Adaptive_Binary, Galloping, Interpolation, or Extrapol_Ahead on the Google data set. Galloping and Interpolation successively improve on Adaptive_Binary search. The performance of Extrapol_Ahead is almost indistinguishable from Interpolation's, although Table V shows that it does perform slightly better. Swapping_SvS and Small_Adaptive show the same behavior.

Table IV. Number of comparisons and searches performed on the Google data set. The average cost of a search (the log of its length), here measured in number of comparisons, is smaller for Sequential and RSequential than for SvS, Swapping_SvS, or Small_Adaptive

Algorithm	Comparisons	Searches	Ratio
SvS using Galloping	16,884	3,542	4.77
Swapping_SvS using Galloping	16,884	3,541	4.77
Small_Adaptive using Galloping	16,884	3,542	4.77
Sequential using Galloping	25,440	5,801	**4.39**
RSequential using Galloping	24,518	5,873	**4.17**
BaezaYates using Galloping	24,285	3,327	7.30
So_BaezaYates using Galloping	20,935	5,209	4.02
BaezaYates using Adaptive_Binary	18,543	3,327	5.57
So_BaezaYates using Adaptive_Binary	15,689	5,209	3.01

For completeness, we summarize the results across all algorithms on the whole data set in Table V.

4.3 Experiments on the TREC GOV2 Data Set

As for the Google data set, we measured the number of searches and comparisons performed and the CPU time used by the algorithms. As in the previous section, we show on the plots the number of comparisons and CPU times for

Table V. Total number of comparisons and CPU times (in millions of cycles) performed by each algorithm over the Google data set. In bold, the best performance in terms of number of comparisons, SvS and Swapping_SvS using Extrapol_Ahead, and in terms of CPU times, SvS using Galloping

Comparisons	SvS	Swapping_SvS	Seq.	BY	So_BY	Small_Adapt.	RSeq.
Total_Binary	58,217	58,209	93,087	57,594	83,710	58,217	94,400
Adaptive_Binary	39,221	39,221	55,817	18,543	15,689	39,225	54,210
Rounded_Binary	54,674	54,671	87,267	54,286	78,511	54,679	88,509
Galloping	16,884	16,884	25,440	24,285	20,935	16,884	24,518
Interpolation	12,184	12,184	17,843	15,352	12,386	12,185	17,398
Extrapolation	13,426	13,426	19,672	17,455	14,428	13,427	19,100
Extrapol_Ahead	**12,125**	**12,125**	17,701	16,179	13,145	12,126	17,279
CPU	SvS	Swapping_SvS	Seq.	BY	So_BY	Small_Adapt.	RSeq.
Total_Binary	5.142	4.976	8.674	5.426	7.140	8.325	15.446
Adaptive_Binary	3.762	3.937	6.704	3.284	3.113	7.208	13.401
Rounded_Binary	4.684	4.831	8.260	5.327	6.908	7.995	14.873
Galloping	**2.791**	2.874	4.808	3.953	3.769	5.980	11.525
Interpolation	3.338	3.434	5.640	4.182	4.046	6.577	11.992
Extrapolation	4.229	4.248	6.617	5.426	5.258	7.493	13.104
Extrapol_Ahead	5.480	5.424	8.641	6.637	7.279	8.614	15.036

different melding and search algorithms as a function of the number of arrays forming the instances.

We restricted our study to the most promising algorithms from the study on Google data set: In particular, we did not consider the melding algorithm RSequential on the TREC GOV2 data set. The fact that the data set is larger allows us to compare the CPU performance of the algorithms on two different architectures: The SUN station has much more memory but a reduced set of instructions which makes multiplication and divisions much more costly; while the INTEL station has a larger set of instructions but much less memory, so that part of the arrays will be cached on the swap partition of the hard drive.

4.3.1 *Comparison with Demaine et al.* In terms of the number of comparisons performed, the melding algorithm Small_Adaptive outperforms all the other melding algorithms, in combination with any search algorithm, which confirms and extends the results reported by Demaine et al. [2001] (see Table VI). As for the Google data set, the value-based search algorithm Extrapol_Ahead improves the performance of each melding algorithm, and in particular, the performance of Small_Adaptive (see Table VI). However, unlike the Google data set, the performance of Interpolation is similar to that of Galloping. This decrease in performance is mainly due to the fact that the numbering scheme of TREC documents left many “gaps,” which contributes to the nonuniformity of posting sets.

4.3.2 *Study of Barbay and Kenyon's Algorithm.* As for the Google data set, the algorithm Sequential [Barbay and Kenyon 2002] is much worse than the other melding algorithms for any fixed search algorithm, in terms of the number of comparisons or searches performed, as well as in terms of CPU time (see Figure 7). This just hints that the instances from the TREC GOV2 data

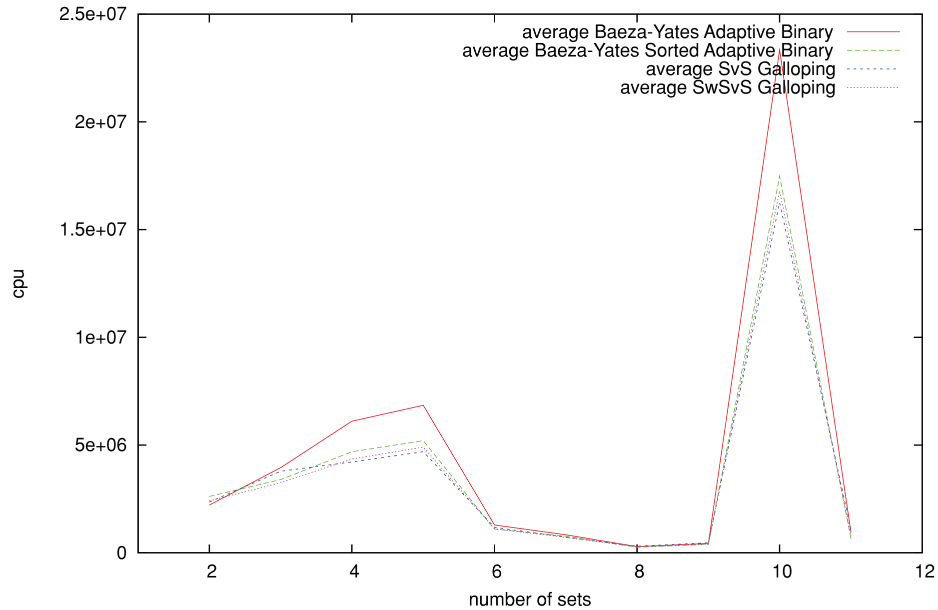

Fig. 6. CPU times for the four best combinations: SvS and Swapping_SvS using Galloping search, BaezaYates, and So_BaezaYates using Adaptive_Binary search on Google data set. SvS, Swapping_SvS, and So_BaezaYates perform very similarly, but BaezaYates performs slightly worse.

set are not too different from those from the Google data set, just larger, both in terms of the sizes of the arrays and in the number of arrays.

4.3.3 *Impact of the Cache.* In contrast to the measures on the Google data set, the number of comparisons is not always correlated to the CPU timings, even for comparison-based search algorithms. In particular, when using the melding algorithms Small_Adaptive or Sequential, the search algorithm Rounded_Binary performs more comparisons than Adaptive_Binary but uses less CPU (see Figure 9). This indicates that Rounded_Binary generates less cache misses, summing to a better overall time.

The same is not true with the other melding algorithms, perhaps because the search queries generated by those algorithms are either shorter (in which case no optimization of the cache is needed), or much larger (in which case cache misses happen at a different level).

4.3.4 *Impact of Architecture Differences.* Not surprisingly, the cache optimization of the Rounded_Binary search algorithm does not give it any advantage on a machine where all the data fits in memory, such as to SUN platform: There all the binary variants perform very similarly (see Figure 10).

We were also able to measure a quantitative difference between the two architectures: The difference of CPU performance between the comparison and value-based search algorithms, such as Galloping and Interpolation, is much larger on the SUN platform than on the INTEL platform, regardless of the melding algorithm considered (see Figures 11 and 12). In general, the hardware cost

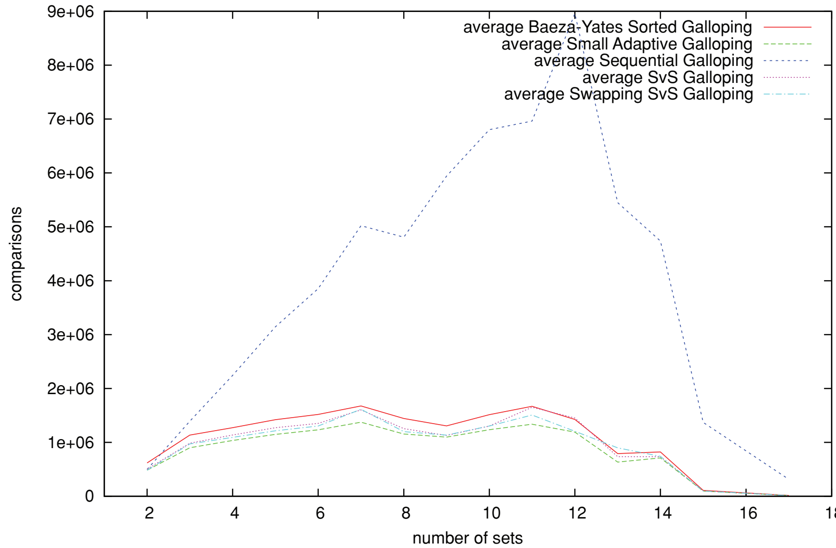

Fig. 7. Number of comparisons performed by various melding algorithms combined with `Galloping` on the TREC GOV2 data set. The difference of performance from `Sequential` is even worse than on the Google data set.

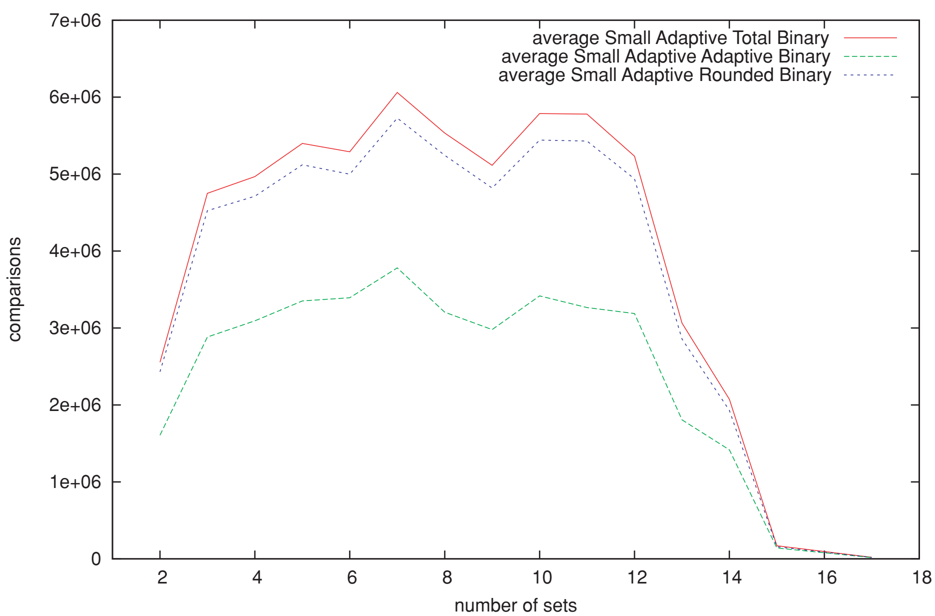

Fig. 8. Number of comparisons performed by variants of binary search combined with `Small_Adaptive` on the TREC GOV2 data set. `Rounded_Binary` and `Total_Binary` perform roughly the same, while `Adaptive_Binary` performs much better.

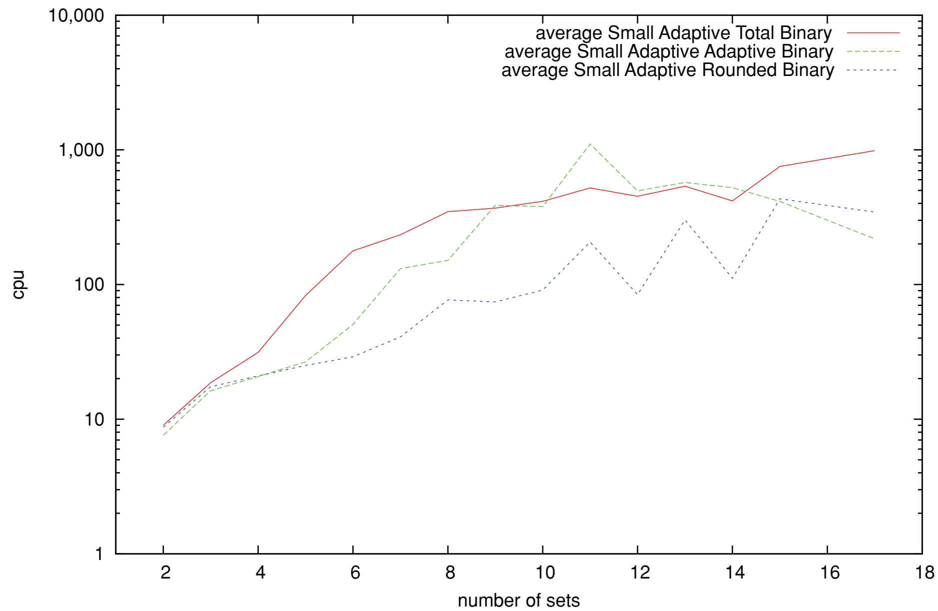

Fig. 9. CPU performance of the various variants of binary search on the `INTEL` platform, in combination with `Small_Adaptive`. The variant `Rounded_Binary` is better in CPU time, thanks to its optimization of the cache.

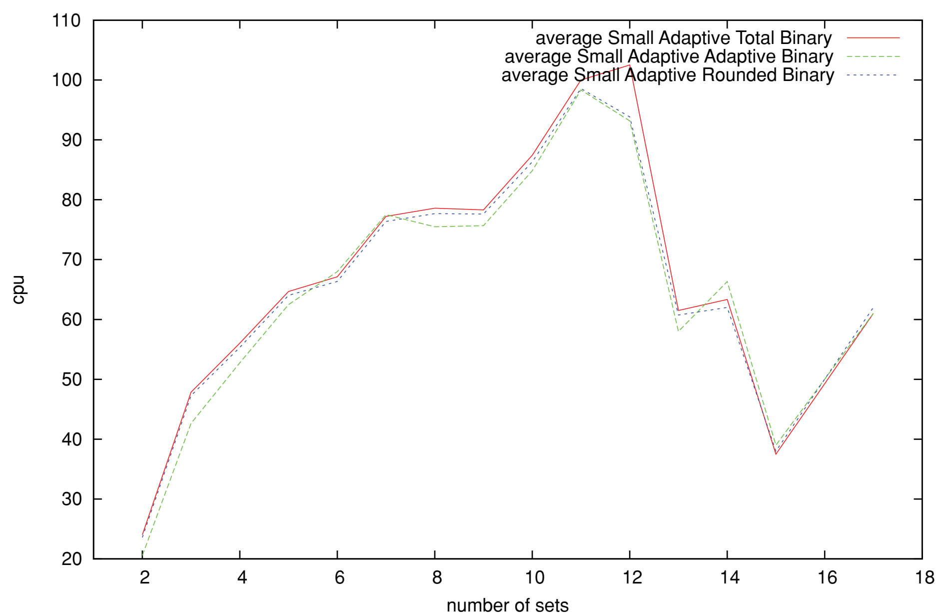

Fig. 10. CPU performance of the various variants of binary search on the `SUN` platform, in combination with `Small_Adaptive`. The binary searches are performing roughly the same.

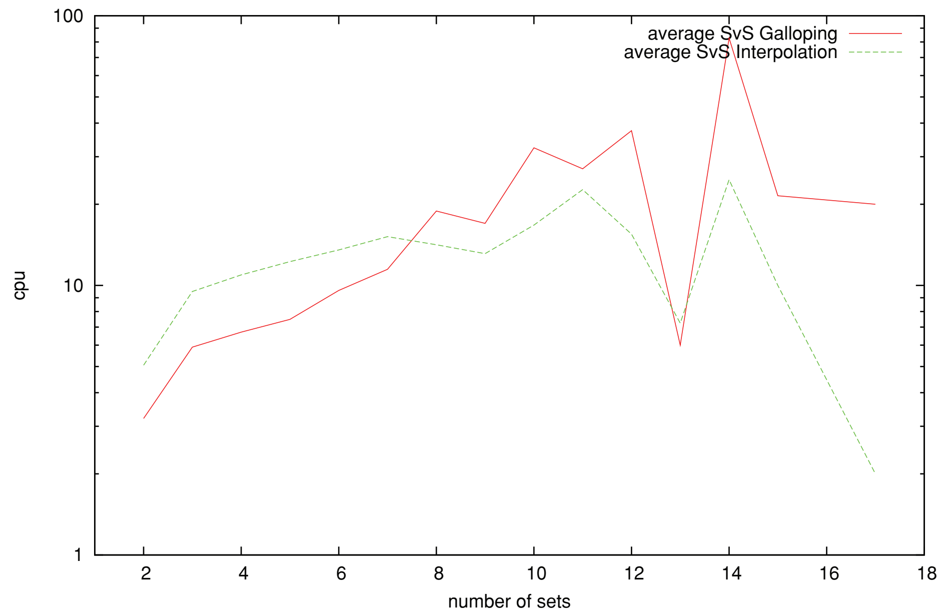

Fig. 11. CPU performance of `Galloping` compared to `Interpolation`, both combined with `SvS`, when solving the TREC GOV2 data set on the `INTEL` platform. The advantage is not clear, but in total, `Galloping` is performing a little better (see Table VI).

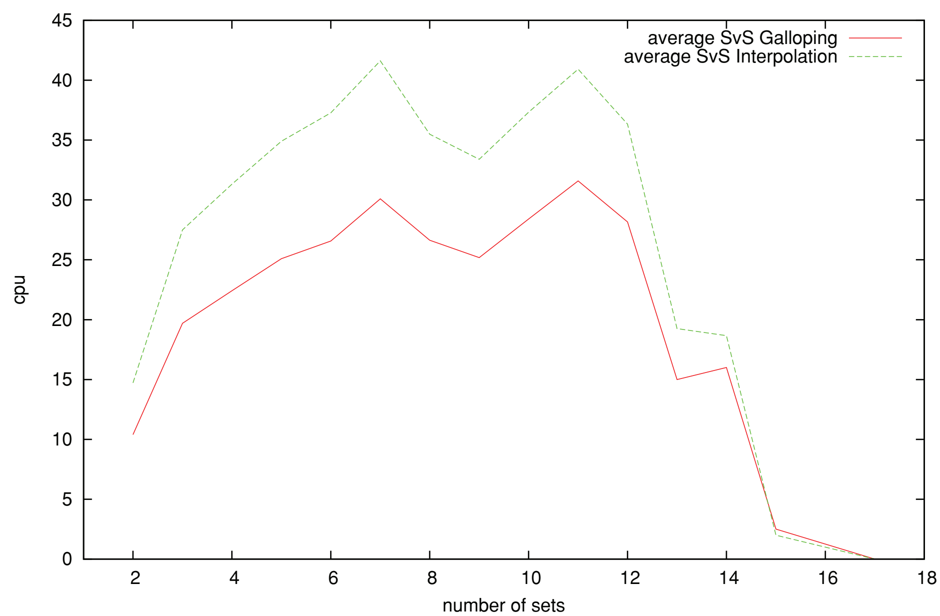

Fig. 12. On `SUN`, CPU performance of `Galloping` compared to `Interpolation`, both combined with `SvS`, when solving the TREC GOV2 data set on the `SUN` platform. `Interpolation` is definitely performing worse.

Table VI. Total number of comparisons (in billions) performed by each algorithm over the TREC GOV2 data set. In bold, the best results, obtained for Small_Adaptive using Extrapol_Ahead.

	SvS	Swapping_SvS	Seq.	BY	So_BY	Small_Adapt.
Adaptive_Binary	13.41	13.44	28.66	7.87	4.12	13.32
Total_Binary	21.70	21.64	39.90	22.43	28.73	21.54
Rounded_Binary	20.46	20.57	37.83	21.43	27.15	20.44
Galloping	4.468	4.473	10.57	9.40	5.52	4.44
Interpolation	4.60	4.61	11.13	8.55	4.76	4.57
Extrapolation	4.25	4.26	9.84	8.61	4.78	4.23
Extrapol_Ahead	3.76	3.77	8.09	8.05	4.23	**3.74**

Table VII. Total CPU time performed by each algorithm over the TREC GOV2 data set. In bold, the smallest CPU times on the INTEL platform, obtained using Swapping_SvS; and on the SUN platform, obtained using SvS, both in combination with Galloping search.

INTEL	SvS	Swapping_SvS	Seq.	BY	So_BY	Small_Adapt.
Adaptive_Binary	117,303	57,686	901,254	53,363	36,273	180,957
Total_Binary	360,526	81,227	598,387	93,341	88,081	320,692
Rounded_Binary	64,910	63,693	169,797	75,730	83,717	108,728
Galloping	33,255	**30,686**	132,245	55,088	40,462	59,081
Interpolation	47,883	49,060	127,338	67,066	54,331	75,162
Extrapolation	49,694	50,570	136,946	77,592	63,244	78,606
Extrapol_Ahead	61,731	62,021	155,396	87,303	81,922	88,674
SUN	**SvS**	**Swapping_SvS**	**Seq.**	**BY**	**So_BY**	**Small_Adapt.**
Adaptive_Binary	153,887	159,169	409,576	112,401	98,411	230,258
Total_Binary	180,854	182,974	354,558	184,239	227,041	244,521
Rounded_Binary	175,343	180,150	348,563	182,170	223,368	241,526
Galloping	**96,907**	102,197	219,816	125,904	111,422	162,243
Interpolation	134,960	140,272	327,509	157,669	142,653	200,471
Extrapolation	142,385	147,886	328,316	185,944	171,270	208,057
Extrapol_Ahead	158,138	163,545	338,525	194,108	192,490	223,195

of interpolation search seems higher on a SUN architecture than on an Intel architecture. We speculate that this might be caused by differences in RISC vs CISC instruction set but the question remains to be studied further.

For completeness, we summarize the results across all algorithms on the whole TREC GOV2 data set in Tables VI and VII.

5. CONCLUSIONS

To summarize our results:

—In terms of the number of searches performed, the best melding algorithms are Small_Adaptive, SvS and Swapping_SvS on random data, and Small_Adaptive on real data.

—In terms of the number of comparisons performed, the best combinations on random data consist in one of the melding algorithms Small_Adaptive, SvS, and Swapping_SvS associated with the search algorithm Extrapol_Ahead. On real data, Small_Adaptive leads over the others under this measure and performs best when combined with Extrapol_Ahead, which improves on previous results [Demaine et al. 2001].

—In terms of CPU time, the best performance on random data corresponds to the BaezaYates algorithm using Adaptive_Binary search (which confirms previous results [Baeza-Yates and Salinger 2005]), closely followed by the SvS algorithm using Galloping search. On real data, the algorithm SvS leads over the others when used in combination with Galloping search, as previously observed.

In terms of the number of searches or comparisons performed, the poor performance of sophisticated algorithms, such as Sequential, designed to exploit short certificates of the intersection [Barbay and Kenyon 2002], or of its randomized variant [Barbay 2003], both on random and real data, indicates the regularity of the instances in both settings: Most instances have a long certificate. On the other hand, the difference of performance of the intersection algorithm BaezaYates on random and real data shows that real data are far from randomly uniform. In particular, the good performance of the Extrapol_Ahead search algorithm shows that value-based search algorithms are not only performing well on sorted arrays of random elements, but also on posting lists.

In terms of CPU time, the architecture differences between the platforms led to both quantitative results variations (the gaps between the performance of some algorithms was larger on the RISC architecture than on the CISC architecture) and qualitative result variations (Rounded_Binary optimizes the cache on the architecture with the smallest amount of memory, but not on the other one). The difference of size between the Google and the GOV2 data set led to qualitative changes in the CPU performance between the variants of binary search, as the variants optimized for cache effects performed better than others on the largest data set, and worst on the smallest. As those search algorithms are outperformed, both in number of comparison performed and in CPU time, by more sophisticated algorithms, this does not yield any qualitative change, but it does hint that optimizing the best search algorithm in CPU time, such as Galloping, so that it takes a better advantage of the cache, might yield even better CPU performance.

Finally, the best solution to compute the intersection of sorted arrays corresponding to conjunctive queries in an indexed search engines seems to be one of the simplest melding algorithm SvS, already used in practice but improved by replacing the use of the Adaptive_Binary search algorithm by an adaptive search algorithm, Galloping search.

ACKNOWLEDGMENTS

We would like to thank Stefan Buettcher for interesting discussions and for giving us access to the TREC GOV2 corpus and query log, Google for making their corpus and query log available, Mike Patterson for his help concerning the simulations on the SUN platform, Mirela Andronescu for her help concerning the PERL scripts processing the data, and Joshua Tam for his initial contribution to the coding of the algorithms as an undergraduate research assistant.

REFERENCES

BAEZA-YATES, R. A. 2004. A fast set intersection algorithm for sorted sequences. In *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*. Springer, Berlin, 400–408.

BAEZA-YATES, R. A. AND SALINGER, A. 2005. Experimental analysis of a fast intersection algorithm for sorted sequences. In *Proceedings of the 12th International Conference on String Processing and Information Retrieval (SPIRE)*. Springer, Berlin, 13–24.

BARBAY, J. 2003. Optimality of randomized algorithms for the intersection problem. In *Proceedings of the Symposium on Stochastic Algorithms, Foundations and Applications (SAGA)*. Springer, Berlin, 26–38.

BARBAY, J. AND KENYON, C. 2002. Adaptive intersection and t-threshold problems. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics, Philadelphia, 390–399.

BARBAY, J. AND KENYON, C. 2008. Alternation and redundancy analysis of the intersection problem. *ACM Trans. Algorithms 4,* 1, 1–18.

BARBAY, J., LÓPEZ-ORTIZ, A., AND LU, T. 2006. Faster adaptive set intersections for text searching. In *Proceedings of the 5th International Workshop on Experimental Algorithms (WEA)*. Springer, Berlin, 146–157.

BENTLEY, J. L. AND YAO, A. C.-C. 1976. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett. 5,* 3, 82–87.

CHEN, D. Z. 2003. Cse 413 – analysis of algorithms – fall. Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN.

DE LA VEGA, W. F., FRIEZE, A. M., AND SANTHA, M. 1998. Average-case analysis of the merging algorithm of Hwang and Lin. *Algorithmica 22,* 4, 483–489.

DEMAINE, E. D., JONES, T. R., AND PATRASCU, M. 2004. Interpolation search for non-independent data. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, Philadelphia, 529–530.

DEMAINE, E. D., LÓPEZ-ORTIZ, A., AND MUNRO, J. I. 2000. Adaptive set intersections, unions, and differences. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, Philadelphia, 743–752.

DEMAINE, E. D., LÓPEZ-ORTIZ, A., AND MUNRO, J. I. 2001. Experiments on adaptive set intersections for text retrieval systems. In *Proceedings of the 3rd Workshop on Algorithm Engineering and Experiments*. SIAM, Philadelphia. Lecture Notes in Computer Science (LNCS), vol. 2153. Washington DC, 5–6.

HWANG, F. K. AND LIN, S. 1971. Optimal merging of 2 elements with n elements. *Acta Inf.*, 145–158.

HWANG, F. K. AND LIN, S. 1972. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM J. Comput. 1,* 1, 31–39.

SKIENA, S. S. 1997. *The Algorithm Design Manual*. TELOS, State University of New York, Stony Brook, NY.

Received September 2006; revised February 2009; accepted March 2009