Faster Set Intersection Algorithms for Text Searching*
University of Waterloo Technical Report CS-2007-13

Jérémy Barbay, Alejandro López-Ortiz, Tyler Lu, and Alejandro Salinger

David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, ON N2L 3G1, Canada
{jbarbay,alopez-o,ttlu,ajsalinger}@uwaterloo.ca

Abstract. 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 paper 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.

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 three decades ago [12, 13], and the average case has been studied in the case of the intersection of two sets, when the elements are uniformly distributed [8].

In 2000, Demaine *et al.* [10] introduced a new intersection algorithm, termed `Adaptive`, which intersects all the sets in parallel to find the shortest proof of the result. In a subsequent study [11], 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 [4] introduced another intersection algorithm, which adapts to the correlation between the terms of the query, and one year later Barbay [3] introduced a randomized variant. To the best of our knowledge, neither of these algorithms were implemented before our study. In 2004, Baeza-Yates [1] introduced an intersection algorithm, based on an alternative technique. Baeza-Yates and Salinger [2] measured the performance of the algorithm in terms of CPU time, on pairs of random arrays.

In this paper 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

* A preliminary version of this paper appeared in [5].

by Baeza-Yates and Salinger [2], (ii) the query log used by Demaine *et al.* [11] on a larger data-set provided by Google, and (iii) and 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 [1], 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 the performances by introducing an adaptive value-based search algorithm.
- We show that a variant of binary search optimizes the cache better than the original version, when the arrays are too large to fit in memory.

The paper is structured as follows: in the next Section 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

The practical studies of intersection algorithms from Demaine *et al.* [11] and from Baeza-Yates and Salinger [2] compare the performance of different algorithms in different contexts (random or practical) and using different measures (CPU or number of comparisons), so these results are not directly comparable. In this paper we measure the performance of these algorithms in the same setting as these studies, as well as in a larger corpus, on which the performance of algorithms is more sensible to cache effects.

2.1 Data-sets

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 [2]: 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 and measure the performance once for the number of comparisons performed and $1,000$ times for the running time

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.* [11], but on a substantially larger and more recent data set.

The set of webpages contains $678,760$ text documents in 6.85 gigabytes 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 indexed, by assigning a number indicating its order in the indexing process. The resulting inverted word index has $2,604,335$ alphanumerical keywords with HTML markup removed.

The query log corresponds to $5,000$ entries. For more details on the query log we refer the reader to Demaine *et al.* [11], where its properties are discussed in detail.

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 US government websites.

The GOV2 web corpus corresponds to approximately 361 GB of text, which once indexed associates $25,197,524$ references to $38,515,138$ keywords. Each document is on average 13.37 KB 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 didn't exist in the inverted word index. This leaves 4590 non-trivial 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 2.1 shows the number of keywords distribution in the queries: most queries have less than 11 keywords.

# of keywords (k)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
# of queries	105	778	1266	1217	793	414	198	98	53	44	14	7	4	5	2	0	1	1

Table 1. The distribution of the sizes of TREC queries: on average, 4.42 keywords per query.

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.

The `INTEL` platform: For all data-sets we used a PC running `Linux version 2.4.20-31.9` on a processor `Intel(R) Pentium(R) 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.

The `SUN` platform: For very large instances we ran additional simulations using an `UltraSparc III` server from Sun running Unix on 8 processors at 900 MHz, 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 multiplications and divisions are not 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`.

Algorithm 1 Pseudo-code for `SvS`

`SvS(set, k)`

```
 1: Sort the sets by size (|set[0]| ≤ |set[1]| ≤ ... ≤ |set[k]|).
 2: Let the smallest set s[0] be the candidate answer set.
 3: for each set s[i], i = 1...k do initialize ℓ[k] = 0.
 4: for each set s[i], i = 1...k do
 5:    for each element e in the candidate answer set do
 6:       search for e in s[i] in the range ℓ[i] to |s[i]|,
 7:       and update ℓ[i] to the last position probed in the previous step.
 8:       if e was not found then
 9:          remove e from candidate answer set,
10:          and advance e to the next element in the answer set.
11:       end if
12:    end for
13: end for
```

3 Algorithms

In this paper 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 [1, 2, 3, 4, 10, 11]. 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.* [10], nor the algorithm proposed by Hwang *et al.* [11]. Instead we focus on four main algorithms, some of them with minor variants.

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 *et al.* [12]: 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.

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.

Algorithm 2 Pseudo-code for `Small_Adaptive`

`Small_Adaptive(set, k)`

```
 1: while no set is empty do
 2:    Sort the sets by increasing number of remaining elements.
 3:    Pick an eliminator e = set[0][0] from the smallest set.
 4:    elimset ← 1.
 5:    repeat
 6:       search for e in set[elimset].
 7:       increment elimset;
 8:    until s = k or e is not found in set[elimset]
 9:    if s = k then
10:       add e to answer.
11:    end if
12: end while
```

Sequential and Random Sequential: Barbay and Kenyon [4] introduced a fourth algorithm, called `Sequential`, which is optimal for a different measure of difficulty, based on the non-deterministic 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 most k searches for each comparison performed by an optimal non-deterministic algorithm: its pseudo-code is given in Algorithm 3.

A randomized variant [3], `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.

Algorithm 3 Pseudo-code 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
```

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 sub-problems. 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 [1] 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. The pseudo-code for `BaezaYates` algorithm is shown in Algorithm 4.

We introduce `So_BaezaYates`, a minor variant of `BaezaYates` which avoids the cost of sorting each intermediate result set to produce the result set in order. The only difference with `BaezaYates` algorithm is that it does not add the found elements to the result set as it finds them, but it only adds them when the median of the smaller list is in the first position of the subset, that is, in each last recursion step for the left subsets. This ensures that the elements are added to the result set in order and saves the cost of explicitly sorting the result, but the algorithm performs more comparisons than the original `BaezaYates` algorithm since `So_BaezaYates` considers the elements found for more recursive calls than `BaezaYates`.

Algorithm 4 Pseudo-code for `BaezaYates`

`BaezaYates`(`set`, `k`)

1: Sort the sets by size ($|\texttt{set}[0]| \leq |\texttt{set}[1]| \leq \ldots \leq |\texttt{set}[k]|$).
2: Let the smallest set $\texttt{set}[0]$ be the candidate answer set.
3: **for** each set $\texttt{set}[i]$, $i = 1 \ldots k$ **do**
4: $\quad \texttt{candidate} \leftarrow \texttt{BYintersect}(\texttt{candidate}, \texttt{set}[i], 0, |\texttt{candidate}| - 1, 0, |\texttt{set}[i]| - 1)$
5: $\quad$ sort the candidate set.
6: **end for**

`BYintersect`(`setA`, `setB`, `minA`, `maxA`, `minB`, `maxB`)

1: **if** `setA` or `setB` are empty **then** return $\emptyset$ **endif**.
2: Let $\texttt{m} = (\texttt{minA} + \texttt{maxA})/2$ and let `medianA` be the element at $\texttt{setA}[\texttt{m}]$.
3: Search for `medianA` in `setB`.
4: **if** `medianA` was found **then**
5: $\quad$ 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`.

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 comparison (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.

We test a third variant, `Rounded_Binary`, which represents a trade-off between `Adaptive_Binary` and `Total_Binary`: it performs the same comparisons than `Total_Binary` so long as the compared elements are larger than 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.

Galloping Search: Originally introduced by Bentley and Yao [6], *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* [14], *exponential search* [7], *doubling search* [4], or *galloping* [10, 11]: we will use this last name for our implementation, `Galloping` search. It solves the unbounded search problem in $2 \log_2 (p+1)$ comparisons.

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 non-uniform data [9]. 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:

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

We propose a variant, `Extrapolation` search, which involves extrapolating on the current and previous positions in $\texttt{set}[i]$. 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).

[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 other implementation performed poorly on all contexts, we discuss here only the one optimizing the number of two-way comparisons.

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 $\texttt{succ}(e)$ of e in $\texttt{set}[i]$ is not far from p_i, then the distribution between $\texttt{set}[i][p_i]$ and $\texttt{set}[i][p_i + l]$ is expected to be similar to the distribution between $\texttt{set}[i][p_i]$ and $\texttt{set}[i][\texttt{succ}(e)]$, 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 [2], 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 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.

Comparison with Baeza-Yates and Salinger [2]: In terms of CPU time, our results agree with Baeza-Yates and Salinger's study: both `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.

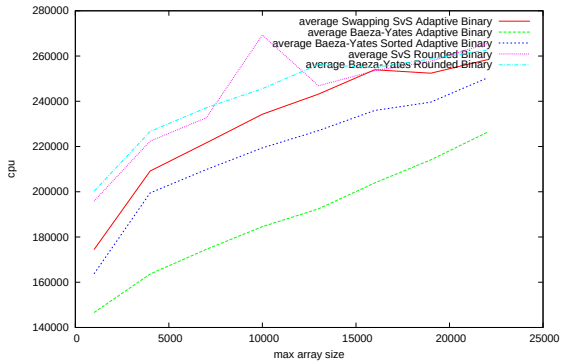

Fig. 1. CPU times for the five best combinations of algorithms on random generated instances. `BaezaYates` using `Adaptive_Binary` performs the best for all size ratios, closely followed by `Swapping_SvS` and `SvS` using `Galloping`.

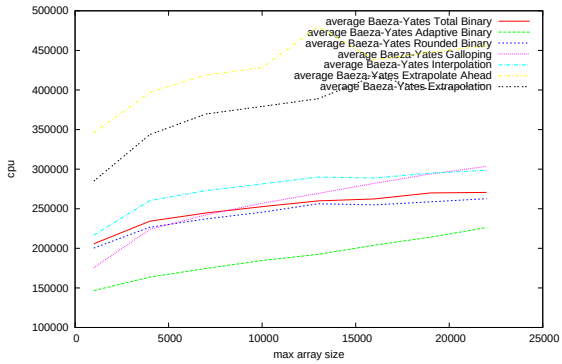

Fig. 2. CPU times for all search algorithms in combination with `BaezaYates`. The best search algorithm is the one proposed originally, `Adaptive_Binary`.

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 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 again Table 2). 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 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`.

Fixing the size of the smallest list to other sizes 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 3.

Algorithm	Searches	Comparisons		Runtime	
SvS	200	1024	(Extrapol_Ahead)	242986	(Rounded_Binary)
Swapping_SvS	200	1024	(Extrapol_Ahead)	230916	(Adaptive_Binary)
Small_Adaptive	200	1024	(Extrapol_Ahead)	435828	(Galloping)
BaezaYates	199	1066	(Interpolation)	188258	(Adaptive_Binary)
So_BaezaYates	328	1064	(Interpolation)	218156	(Adaptive_Binary)
Sequential	385	1198	(Extrapol_Ahead)	327075	(Adaptive_Binary)

Table 2. 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 corresponds to the difference in the number of comparison performed. The CPU times are not correlated with the two other measures.

	SvS		Swapping_SvS		Sequential		BaezaYates		So_BaezaYates		Small_Adaptive	
	cmp	cpu	cmp	cpu	cmp	cpu	cmp	cpu	cmp	cpu	cmp	cpu
Total_Binary	2815	262397	2815	254008	4397	457540	2811	250018	4501	402544	2815	677318
Adaptive_Binary	2469	255064	2469	230916	2632	327075	1620	**188258**	1620	218156	2469	444476
Rounded_Binary	2623	242986	2623	246871	3997	436438	2629	242773	4190	391347	2623	443064
Galloping	2087	245333	2087	244216	2237	332311	2410	255945	2373	286040	2087	435828
Interpolation	1067	279127	1067	280624	1242	374779	1066	275463	1064	304616	1067	466446
Extrapolation	1281	375585	1281	371444	1444	464203	1261	373947	1262	401933	1281	547751
Extrapol_Ahead	**1024**	413209	**1024**	404841	1198	576109	1085	426452	1073	506075	**1024**	584941

Table 3. 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.

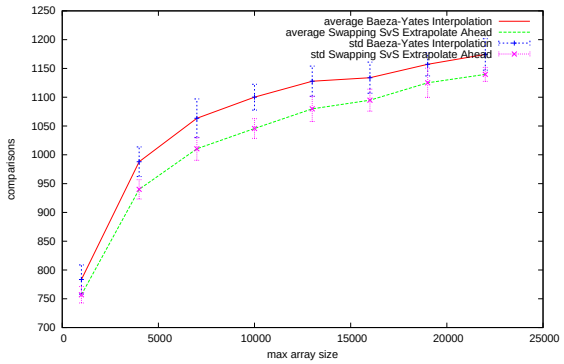

Fig. 3. Number of comparisons for BaezaYates using Interpolation and Swapping_SvS using Extrapol_Ahead on the Random data-set. Swapping_SvS with Extrapol_Ahead performs visibly better.

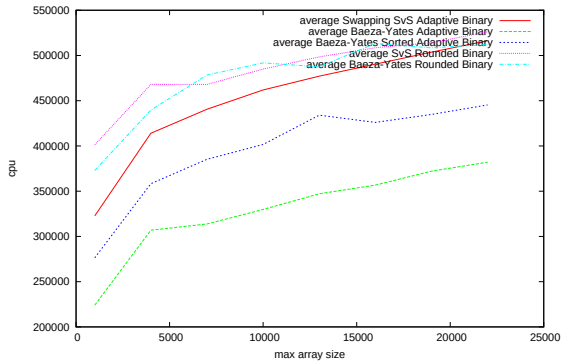

Fig. 4. CPU times for the five best combinations of algorithms on the Random data-set with the smallest list of size 400. The order of the algorithms is the same than when the smallest list has size 200:BaezaYates using Adaptive_Binary performs the best for all size ratios.

4.2 Experiments on the Google data-set

Solving the queries provided by Google on the index of their own web-crawl, Demaine *et al.* [11] 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.

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. Similarly 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 performances 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.

Comparison with Demaine *et al.* [11]: Considering the same algorithms studied by Demaine *et al.*, our results agree with the previous study: `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 with 2.286 keywords per query on average: `SvS`, `Swapping_SvS` and `Small_Adaptive` behave the same on instances which consist of only two arrays.

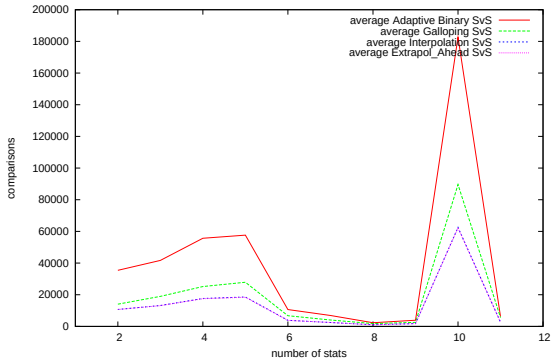

Fig. 5. Number of comparisons for `SvS` using `Adaptive_Binary`, `Galloping`, `Interpolation` or `Extrapol_Ahead` on the Google data-set. `Galloping` and `Interpolation` successively improve on `Adaptive_Binary` search. The performance of `Extrapol_Ahead` is almost indistinguishable from `Interpolation`'s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

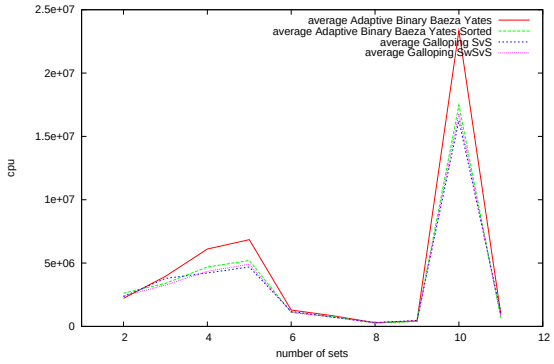

Fig. 6. CPU times for the four best combinations: `SvS` and `Swapping_SvS` using `Galloping` search, and `BaezaYates` and `So_BaezaYates` using `Adaptive_Binary` search on Google data-set. `SvS`, `Swapping_SvS` and `So_BaezaYates` perform very similarly, but `BaezaYates` performs slightly worse.

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` benefits even more from the use of `Extrapol_Ahead`, the variants that we introduced (see again 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.*.

Study of Barbay and Kenyon's [4] algorithm: The algorithm proposed by Barbay and Kenyon [4] and its randomized variant [3] are both performing noticeably more comparisons than the other intersection algorithms measured, independently of the search algorithm chosen (see Table 4). This high number of comparisons is correlated with the high number of searches 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 again Table 4). 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,

Algorithm	Comparisons	Searches	Ratio
`SvS` using `Galloping`	16884	3542	4.77
`Swapping_SvS` using `Galloping`	16884	3541	4.77
`Small_Adaptive` using `Galloping`	16884	3542	4.77
`Sequential` using `Galloping`	25440	5801	**4.39**
`RSequential` using `Galloping`	24518	5873	**4.17**
`BaezaYates` using `Galloping`	24285	3327	7.30
`So_BaezaYates` using `Galloping`	20935	5209	4.02
`BaezaYates` using `Adaptive_Binary`	18543	3327	5.57
`So_BaezaYates` using `Adaptive_Binary`	15689	5209	3.01

Table 4. 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`.

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 performances in terms of the number of comparisons are much better (see again Table 4). 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.

Real time on real data: The CPU performances are correlated with the number of comparisons for all melding and search algorithms, except for the value based search algorithms, such as `Extrapol_Ahead` (see Figure 6). The fact that `Interpolation` performs more comparisons than `Extrapol_Ahead` (see Figure 8), 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.

For completeness we summarize the results across all algorithms on the whole data set in Table 5.

	SvS		Swapping_SvS		Sequential		BaezaYates		So_BaezaYates		Small_Adaptive		RSequential	
	cmp	cpu	cmp	cpu	cmp	cpu	cmp	cpu	cmp	cpu	cmp	cpu	cmp	cpu
Total_Binary	58217	5.142	58209	4.976	93087	8.674	57594	5.426	83710	7.140	58217	8.325	94400	15.446
Adaptive_Binary	39221	3.762	39221	3.937	55817	6.704	18543	3.284	15689	3.113	39225	7.208	54210	13.401
Rounded_Binary	54674	4.684	54671	4.831	87267	8.260	54286	5.327	78511	6.908	54679	7.995	88509	14.873
Galloping	16884	**2.791**	16884	2.874	25440	4.808	24285	3.953	20935	3.769	16884	5.980	24518	11.525
Interpolation	12184	3.338	12184	3.434	17843	5.640	15352	4.182	12386	4.046	12185	6.577	17398	11.992
Extrapolation	13426	4.229	13426	4.248	19672	6.617	17455	5.426	14428	5.258	13427	7.493	19100	13.104
Extrapol_Ahead	**12125**	5.480	**12125**	5.424	17701	8.641	16179	6.637	13145	7.279	12126	8.614	17279	15.036

Table 5. 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.

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 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 performances of the algorithms on two different architectures: the SUN station has much more memory but a limited 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.

Comparison with Demaine *et al.* [11]: 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.* [11] (see Table 6). 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 (again, see Table 6). 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 non-uniformity of posting sets.

Study of Barbay and Kenyon's [4] algorithm: As for the Google data-set, the algorithm Sequential 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-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.

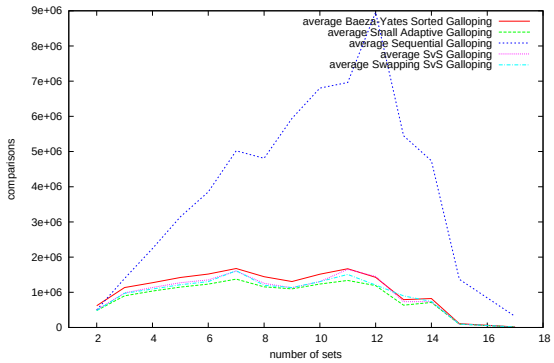

Fig. 7. Number of comparisons performed by various melding algorithm combined with `Galloping` on the TREC GOV2 data-set. The difference of performance from `Sequential` is even worse than on the Google data-set.

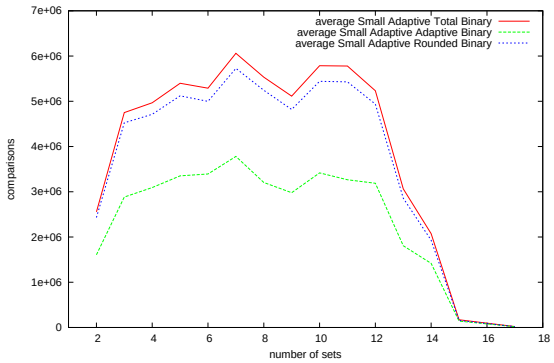

Fig. 8. Number of comparisons performed by variants of binary search combined with `Small_Adaptive` on the TREC GOV2 data-set. `Rounded_Binary` and `Total_Binary` perform roughly the same, while `Adaptive_Binary` performs much better.

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 over-all 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).

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 on platform `SUN`: then 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, and this independently of the melding algorithm considered (see Figure 11 and 12). In general, all interpolation variants perform poorly on the `SUN` platform, and somewhat less poorly on the `INTEL` platform. The explanation is twofold:

- The RISC processor of the `SUN` platform does not support multiplications natively, which are performed through an online routine, whereas the CISC/RISC processor of the `INTEL` platform does support them: this increases the difference of computational cost between interpolation variants and other search algorithms.
- As the `SUN` platform we tested on has significantly more memory than the `INTEL` platform, less cache misses occur and it is less important on the `SUN` platform than on the `INTEL` platform to reduce the number of memory accesses: this reduces the main advantage of interpolation variants.

For completeness we summarize the results across all algorithms on the whole TREC GOV2 data set in Tables 6 and 7.

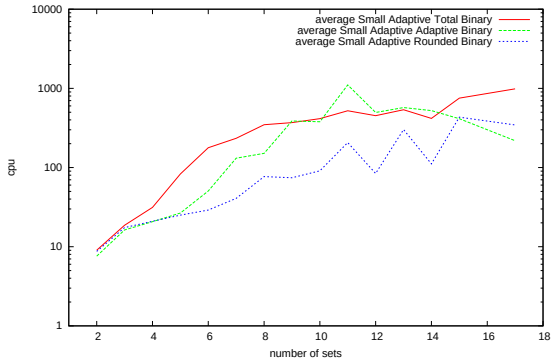

Fig. 9. CPU performance of the various variants of binary search on the `INTEL` platform, in combination with `Small_Adaptive`. The variant `Rounded_Binary` is better in CPU time, thanks to its optimization of the cache.

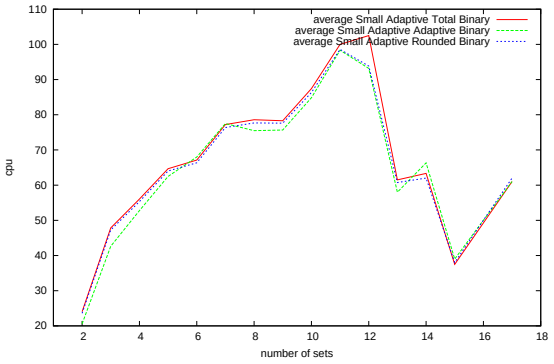

Fig. 10. CPU performance of the various variants of binary search on the `SUN` platform, in combination with `Small_Adaptive`. The binary searches are performing roughly the same.

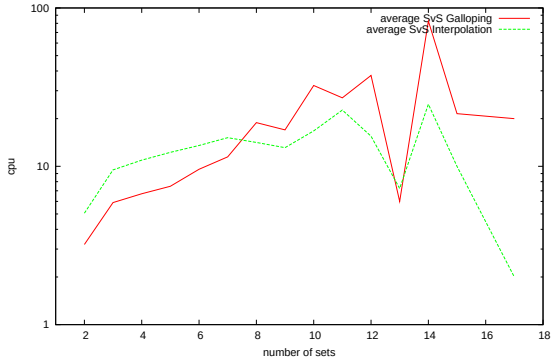

Fig. 11. CPU performances of `Galloping` compared to `Interpolation`, both combined with `SvS`, when solving the TREC GOV2 data-set on the `INTEL` platform. The advantage is not clear, but in total `Galloping` is performing a little better (see Table 6.

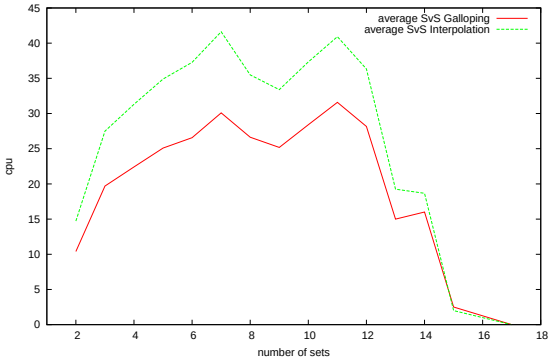

Fig. 12. On `SUN` CPU performances of `Galloping` compared to `Interpolation`, both combined with `SvS`, when solving the TREC GOV2 data-set on the `SUN` platform. `Interpolation` is definitely performing worse.

	SvS	Swapping_SvS	Sequential	BaezaYates	So_BaezaYates	Small_Adaptive
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 6. 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		Sequential		BaezaYates		So_BaezaYates		Small_Adaptive	
	INTEL	SUN	INTEL	SUN	INTEL	SUN	INTEL	SUN	INTEL	SUN	INTEL	SUN
Adaptive_Binary	117303	153887	57686	159169	901254	409576	53363	112401	36273	98411	180957	230258
Total_Binary	360526	180854	81227	182974	598387	354558	93341	184239	88081	227041	320692	244521
Rounded_Binary	64910	175343	63693	180150	169797	348563	75730	182170	83717	223368	108728	241526
Galloping	33255	**96907**	**30686**	102197	132245	219816	55088	125904	40462	111422	59081	162243
Interpolation	47883	134960	49060	140272	127338	327509	67066	157669	54331	142653	75162	200471
Extrapolation	49694	142385	50570	147886	136946	328316	77592	185944	63244	171270	78606	208057
Extrapol_Ahead	61731	158138	62021	163545	155396	338525	87303	194108	81922	192490	88674	223195

Table 7. 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.

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` takes the advantage on the others and forms the best combination when combined with `Extrapol_Ahead`, which improves on the previous results [11].
- In terms of CPU time, the best performance on random data corresponds to the `BaezaYates` algorithm using `Adaptive_Binary` search (which confirms previous results [2]), closely followed by the `SvS` algorithm using `Galloping` search. On real data, the algorithm `SvS` takes the advantage and forms the best solution, still in combination with `Galloping` search.

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 [4], or of its randomized variant [3], 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 performances.

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.

Acknowledgements: 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.

Bibliography

[1] R. A. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 3109 of *Lecture Notes in Computer Science (LNCS)*, pages 400–408. Springer, 2004.

[2] R. A. Baeza-Yates and A. Salinger. Experimental analysis of a fast intersection algorithm for sorted sequences. In *Proceedings of 12th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 13–24, 2005.

[3] J. Barbay. Optimality of randomized algorithms for the intersection problem. In *Proceedings of the Symposium on Stochastic Algorithms, Foundations and Applications (SAGA)*, volume 2827 of *Lecture Notes in Computer Science (LNCS)*, pages 26–38. Springer, 2003.

[4] J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 390–399. Society for Industrial and Applied Mathematics (SIAM), 2002.

[5] J. Barbay, A. López-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. In *Proceedings of the 5th International Workshop on Experimental Algorithms (WEA)*, volume 4007 of *Lecture Notes in Computer Science (LNCS)*, pages 146–157. Springer Berlin / Heidelberg, 2006.

[6] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Information processing letters*, 5(3):82–87, 1976.

[7] D. Z. Chen. Cse 413 - analysis of algorithms - fall. Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556 USA, 2003.

[8] W. F. de la Vega, A. M. Frieze, and M. Santha. Average-case analysis of the merging algorithm of hwang and lin. *Algorithmica*, 22(4):483–489, 1998.

[9] E. D. Demaine, T. R. Jones, and M. Patrascu. Interpolation search for non-independent data. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 529–530, 2004.

[10] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 743–752, 2000.

[11] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In *Proceedings of the 3rd Workshop on Algorithm Engineering and Experiments*, volume 2153 of *Lecture Notes in Computer Science (LNCS)*, pages 5–6, Washington DC, January 2001.

[12] F. K. Hwang and S. Lin. Optimal merging of 2 elements with n elements. *Acta Informatica*, pages 145–158, 1971.

[13] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal of Computing*, 1(1):31–39, 1972.

[14] S. S. Skiena. *The Algorithm Design Manual*. TELOS, State University of New York, Stony Brook, NY, 1997.