Faster Adaptive Set Intersections for Text Searching

Jérémy Barbay, Alejandro López-Ortiz, and Tyler Lu

David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, ON N2L 3G1, Canada
{jbarbay,alopez-o,ttlu}@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 engineer a better algorithm for this task, which improves over those proposed by Demaine, Munro and López-Ortiz [SODA 2000/ALENEX 2001], by using a variant of interpolation search. More specifically, our contributions are threefold. First, we corroborate and complete the practical study from Demaine *et al.* on comparison based intersection algorithms. Second, we show that in practice replacing binary search and galloping (one-sided binary) search by interpolation search improves the performance of each main intersection algorithms. Third, we introduce and test variants of interpolation search: this results in an even better intersection algorithm.
Topics. *Evaluation of Algorithms for Realistic Environments, Implementation, Testing, Evaluation and Fine-tuning of Algorithms, Information Retrieval.*

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]. In 2000, Demaine *et al.* improved over this by proposing a faster method for computing the intersection of k sorted sets [6] using an adaptive algorithm. Their algorithm has optimal worst-case behaviour on a much finer analysis than simply worst-case input size. We refer the reader to [6] for the precise details on the adaptive measure used.

In a followup study they showed that the adaptive theoretical optimal algorithm is not always best in practice in the context of search engines [7]. In that study, they compared a straightforward implementation of an intersection algorithm, termed `SvS`, with their adaptive algorithm, termed `Adaptive`, and showed that on the given data `Adaptive` is superior only for queries involving two or three terms, while thereafter `SvS` outperforms it by a constant factor. Their study uses what at the time was a sizable collection of plain text from web pages. Using this data set, Demaine *et al.* engineered an algorithm, termed `Small Adaptive`, that combines the best aspects of both `Adaptive` and `SvS`. They showed experimentally that on the given data set this algorithm outperforms both the `Adaptive` and `SvS` algorithm.

In this paper we revisit that study. Our contributions are threefold. First, we corroborate the practical study from [7] by considering a much larger web crawl and extend their study to include two more recent algorithms, introduced in [3]. The results are similar to those of the original study: the algorithm termed `Small Adaptive` is the one which performs the best. Second, we study the impact of replacing binary searches and galloping (one-sided binary) searches by interpolation searches, for each of the main intersection algorithms. Our results show that this improves the performance of each intersection algorithm. The optimal algorithm, `Interpolation Small Adaptive`, is based on `Small Adaptive`, and our results show that the relative performance of the intersection algorithms are the same when using interpolation search than when using binary search and galloping. Third, we introduce several parameterized variants of extrapolation search, which combine the concepts of interpolation search and galloping, taking advantage of both. We evaluate the performance of each of those variants using `Small Adaptive` as a base, and we identify the best variant, termed `Extrapolate Ahead Small Adaptive`, which at each step computes the position of the next comparison using the values of elements at distance l of each other, and which performs the best when l is logarithmic on the size of the set. This results in an intersection algorithm which performs even better in practice than simply introducing interpolation.

The paper is structured as follows: in the next section we describe the data set on which we evaluated the various algorithms discussed. In Section 3 we describe in detail the intersection algorithms studied, and the basis of the interpolation algorithms. In Section 4 we present our experimental results. We conclude in Section 5 with a summary of the results.

2 Dataset

The intersection of sets in the context of search engines is a driving application for this work. Thus we test our algorithms using a web crawl together with a representative query log from a search engine. Each set corresponds to a keyword occurring in a query, and the elements of each set refer to integer document identifiers of those web pages containing the keyword. We use a sample web corpus from Google of 6.85 gigabytes of text as well as a 5000 entry query log, also from Google. The query log is the same as in [7], while the web crawl is a substantially larger and more recent data set. In the past we empirically verified that the relative performance of the algorithms did not change when run on corpora varying in size by orders of magnitude. Our results using this new larger set are consistent with this observation.

The web corpus was indexed into an inverted word index, which lists a set of document identifiers in increasing order for each word appearing in the corpus. The total number of web pages indexed is approximately 600,000. The size of the resulting inverted word index is 1.06 gigabytes with HTML markup removed, and the number of words in the index is 2,604,335. Note that words consists of only alphanumerical characters.

In the sample query log from Google, we do not consider queries that contain words not found in our index nor queries that consists of a single keyword since no set intersection need be performed in this case. We refer the reader to [7] for a more thorough discussion on the query log.

3 Algorithms

3.1 Intersection Algorithms

Various algorithms for the intersection of k sets have been introduced in the literature [3, 6, 7]. In this study we focus on four particular ones, described below. We do not consider, however, the most naïve sequential (linear merging) algorithm as both theoretical and experimental analysis show that its performance is significantly worse than the ones studied here.

Algorithm 1 Pseudo-code for `Adaptive`

```
Choose eliminator e = set[0][0], in the set elimset ← 0.
Consider the first set, i ← 1
while the eliminator e ≠ ∞ do
  perform one step of the galloping search in set[i].
  if the gallop overshot then
    binary search in set[i] for e.
    if e was found then
      increase the occurrence counter, and let i ← i + 1 mod k, i ≠ elimset.
      if the value of occurrence counter is k then
        output e and let e ← set[i][succ(e)],   elimset ← i
        i ← i + 1 mod k, i ≠ elimset.
      else
        set e to the first element in set[i] which is larger than e.
        update the set elimset ← i and consider the next set i ← i+1 mod k, i ≠
        elimset.
      end if
    end if
  end if
end while
```

The theoretical study in [6] introduced an information theoretical optimum algorithm, which was implemented in [7] under the name `Adaptive`. This algorithm performs a search in *all* other sets for an element from one set, using a one-sided binary search or "galloping" search. The element being searched for is updated using a greedy technique. For the details we refer the reader to [6].

The experimental study in [7] introduced more algorithms, simulating fourteen different algorithms to study their practical performance on a query set provided by Google and a data set obtained through their own web crawl. Of those, we focus on two particular ones: `SvS` and `Small Adaptive`. `SvS` is a straightfor-

Algorithm 2 Pseudo-code for `SvS`

Sort the sets by size ($|set[0]| \leq |set[1]| \leq \ldots \leq |set[k]|$).
Let the smallest set $s[0]$ be the candidate answer set.
for each set $s[i]$, $i = 1 \ldots k$ **do** initialize $\ell[k] = 0$.
for each set $s[i]$, $i = 1 \ldots k$ **do**
  for each element e in the candidate answer set **do**
    binary search for e in $s[i]$ in the range $\ell[i]$ to $|s[i]|$,
    and update $\ell[i]$ to the last position probed in the previous step.
    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

ward algorithm widely used in practice, which intersects the sets two at a time in increasing order by size, starting with the two smallest. It uses a binary search procedure to determine if an element in the first set appears in the second set.

Algorithm 3 Pseudo-code for `Small Adaptive`

Sort the sets by size ($|set[0]| \leq |set[1]| \leq \ldots \leq |set[k]|$).
Choose an eliminator $e = set[0][0]$ in the set $elimset \leftarrow 0$.
Consider the first set, $i \leftarrow 1$.
while the eliminator $e \neq \infty$ **do**
  gallop once in $set[i]$.
  if the gallop overshot **then**
    binary search in $set[i]$ for e.
    if e was found **then**
      increase the occurrence counter and let $i \leftarrow i + 1 \bmod k, i \neq elimset$.
      if the value of occurrence counter is k **then**
        add e to answer.
        resort the sets, and let $e \leftarrow set[0][succ(e)]$, $elimset \leftarrow 0$, $i \leftarrow 1$
      end if
    else
      resort the sets.
      if $i = 0$ or $i = 1$ **then** consider the set $i \leftarrow 1 - i$,
      else consider the first set: $elimset \leftarrow 0$, $i \leftarrow 1$. **end if**
    end if
  end if
end while

`Small Adaptive` is a hybrid algorithm, which combines the best properties of `SvS` and `Adaptive`. For each element in the smallest set, it performs a galloping one-sided 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 4 Pseudo-code for `Sequential`

```
Choose an eliminator e = set[0][0], in the set elimset ← 0.
Consider the first set, i ← 1.
while the eliminator e ≠ ∞ do
  Gallop for e in set[i] till overshot
  binary search in set[i] for e
  if the binary search found e then
    increase the occurrence counter.
    if the value of occurrence counter is k then output e end if
  end if
  if the value of the occurrence counter is k, or e was not found then
    update the eliminator to e ← set[i][succ(e)].
  end if
  Consider the next set in cyclic order i ← i + 1 mod k.
end while
```

The theoretical study in [3] introduces 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.

The pseudo-code for the algorithms described above is given in Algorithms 1 to 4. Each of those algorithms has linear time worst case behaviour, and each performs better than the others on at least one instance. `Adaptive` performs well on instances with an intersection certificate that can be encoded in a small amount of space, while `Sequential` performs well on instances whose intersection certificate contains a small number of comparisons. `SvS` reduces the number of sets by intersecting the two smallest sets, searching for the elements of the smallest set in the larger set; `Small Adaptive` performs similarly so long as no element is found to be in the intersection of the two sets, at which point it checks for it in the other sets, and after which it updates which sets are the smallest. Note that `Small Adaptive` and `SvS` are the only algorithms taking active advantage of the difference of sizes of the sets, and that `Small Adaptive` is the only one which takes advantage of how this size varies as the algorithm eliminates elements: `Adaptive` and `Sequential` ignore this information.

All of these algorithms are based on galloping and binary search, and use only comparisons between the elements: we study the impact on the performance of replacing those searches with interpolation search, or a suitably engineered variant of interpolation search, as described in the next section.

3.2 Search Algorithms

Interpolation search has long been known to perform significantly better than binary search on data randomly drawn from a uniform distribution, hence it is only natural to test if this holds using web crawled data. Moreover, recent developments suggest that interpolation search is also a reasonable technique for non-uniform data [5]. Our experiments, which we describe in the next Section, confirm this conjecture.

Recall that interpolation search for an element of value e in an array $set[i]$ on the range a to b probes a position as given by the formula:

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

In each of `Adaptive`, `Small Adaptive` and `Sequential` we replace each galloping step by an interpolation probe, and we replace binary search with interpolation search. In essence, the two changes are equivalent to performing an interpolation search in $set[i]$ for the eliminator. The index probed is $I(\ell[i], n_i)$, where $\ell[i]$ is the current position in $set[i]$ and n_i is the size $|set[i]|$ of $set[i]$.

4 Experimental Results

We compare the performance of each of the four algorithms described in the previous section by focusing on the number of comparisons performed by the algorithms. For large data sets such as in search engines, the run time is dominated by external memory accesses. It has long been known that the number of comparisons by an algorithm generally shows high correlation with the number of I/O operations, so we follow this convention. Our model has certain other simplifications; for example posting sets are likely to be stored in a compressed form, albeit one suitable for random access. We posit that most such refinements and other system specific improvements are likely orthogonal to the *relative* performance of the search algorithms presented here.

4.1 Comparing Intersection Algorithms

Here we present the part of our study which corroborates the study of [7], as we measure the performance of the algorithms on a larger data set, and completes it as we compare one more intersection algorithm (`Sequential`).

Figures 1 and 2 show that, when using binary search, `Small Adaptive` is much better than `Sequential`. This is likely due to the fact that `Small Adaptive` searches for the eliminator in sets of increasing order, and that it does not cycle through the sets when choosing an eliminator.

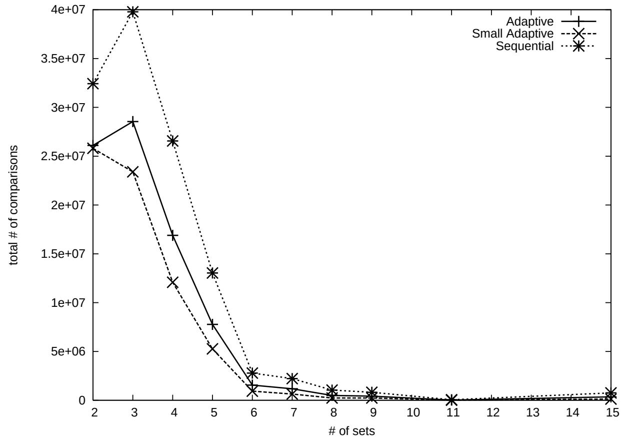

Fig. 1. Performance of various Intersection algorithms when using binary search.

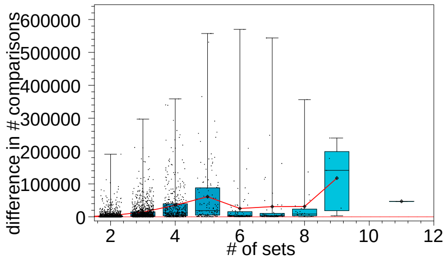

Fig. 2. **`Small Adaptive`** (high wins) vs. **`Sequential`** (low wins). The algorithm **`Small Adaptive`** is always better.

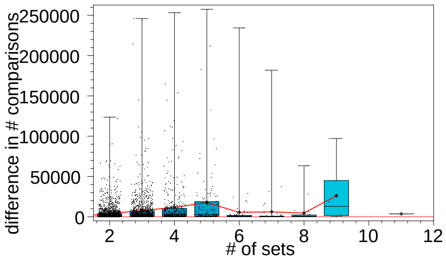

Fig. 3. **`Interpolation Small Adaptive`** (high wins) vs. **`Small Adaptive`** (low wins). It improves on all instances, and is consistent over k.

4.2 Comparing Interpolation and binary search

Here we present a first approach of the impact of replacing the binary searches and galloping by interpolation searches in the intersection algorithms. It is well known that interpolation search outperforms binary search, on average on arrays whose elements are well behaved (uniformly distributed). Thus it is expected that replacing binary search by interpolation search would improve the performance of the intersection algorithms. As gallop search [4] is a local search algorithm, it is not necessarily outperformed by interpolation search: we show here that in practice it is.

Figure 3, 4 and 5 show the clear advantage of using interpolation search over binary search, as each of the three intersection algorithm using interpolation search has a clear advantage over its variant using binary search, outperforming it on almost all instances.

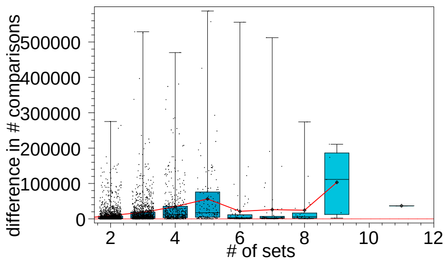

Fig. 4. `Interpolation Sequential` (high wins) vs. `Sequential` (low wins). Interpolation search provides roughly a two-fold improvement.

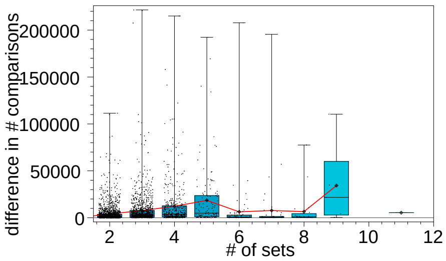

Fig. 5. `Interpolation Adaptive` (high wins) vs. `Adaptive` (low wins). The improvement is more noticeable if k is smaller.

As a side note, the study of the *ratio* of the performances (not shown here because of space limitations) shows that the ratio between the performance of `Interpolation Adaptive` and `Adaptive`, while always larger than one, decreases when k increases. This is likely due to the fact that the algorithm continually cycles through the sets trying to find a set which does not contain the eliminator [7]. Thus, the overhead caused by the cycling, which performs one interpolation going through each set (as opposed to galloping), is dominating the number of comparisons when k is relative large. Note that, in contrast, since `Small Adaptive` does not cycle through the sets, the average ratio between the performance of `Small Adaptive` and `Interpolation Small Adaptive` stays fairly constant with respect to k.

The experiments suggest that web crawled data is amenable to interpolation search, and hence using this technique gives a noticeable reduction in the number of comparisons required.

4.3 Introducing and Comparing Extrapolation Variants

In this section, we introduce an adaptation of interpolation search, which we named *extrapolation*, and some variants of it. We test those variants on our data set. Interestingly, our experimental results show that the difference in performance between search algorithms is independent of the intersection algorithm chosen. Since `Small Adaptive` is the fastest algorithm among those tested in [7] (when using binary search) and in our measures (when using binary search as well as when using interpolation search), we use it as a reference to show the performance of different interpolation techniques (See Figure 6).

The first variant, which we call `Extrapolation Small Adaptive`, involves extrapolating on the current and previous positions in $set[i]$. Specifically, the extrapolation step probes the index $I(p'_i, p_i)$, 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

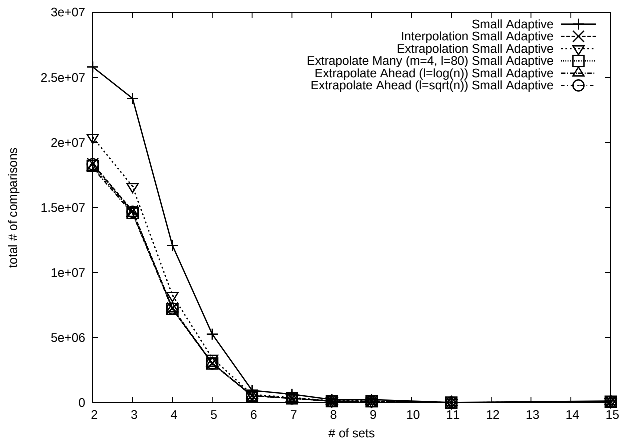

Fig. 6. Relative performance of search algorithms in `Small Adaptive`: binary search is outperformed by both interpolation search and Extrapolation-based algorithms.

next jump is the double of the value of the current jump). Figure 7 shows that extrapolation alone does worse than interpolation. Those results suggest that using the previous "explored data" for extrapolation is not as accurate as using a standard interpolation probe, given by $I(p_i, n_i)$, on the remaining elements in $set[i]$.

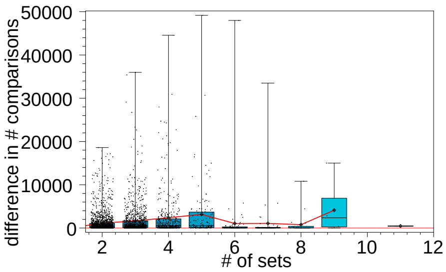

Fig. 7. `Extrapolation` (low wins) vs `Interpolation Small Adaptive` (high wins). Using extrapolation on previous explored data is less accurate.

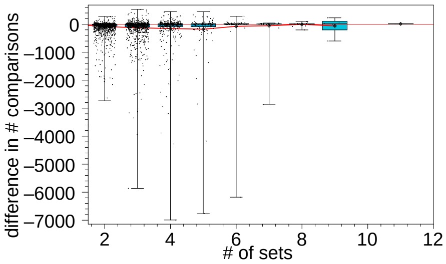

Fig. 8. `Extrapolate Ahead` ($l = 50$) (low wins) vs. `Interpolation Small Adaptive` (high wins). The look-ahead range improves the performance.

The second variant, `Extrapolate Ahead Small Adaptive`, is similar to `Extrapolation Small Adaptive`, 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 $I(p_i, p_i + l)$ 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 $set[i][p_i]$ 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. Figure 8 shows that for $l = 50$, `Extrapolate Ahead Small Adaptive` performs as well as `Interpolation Small Adaptive`, and that their performance stays close when it is worse. Figure 10 shows a similar result for $l = \sqrt{n_i}$.

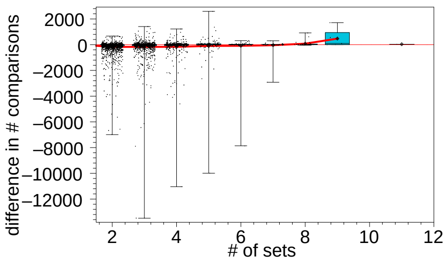

Fig. 9. `Extrapolate Ahead` ($l = \lg n_i$) (low wins) vs. `Interpolation Small Adaptive` (high wins). More complex results with a logarithmic look-ahead range.

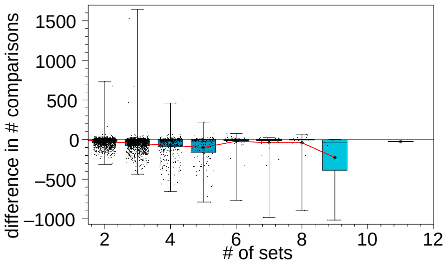

Fig. 10. `Extrapolate Ahead` ($l = \sqrt{n_i}$) (low wins) vs. `Interpolation Small Adaptive` (high wins). Deterioration of performance for a polynomial look-ahead range.

Figure 9 shows that choosing a smaller value for the look-ahead range l, such as $l = \log_2 n_i$, deteriorates slightly the performance: the algorithm has a much less precise approximation of the distribution of the values in the array.

The third variant involves extrapolating many times, which we call `Extrapolate Many Small Adaptive`. We calculate the index by taking the average of several extrapolations, which is based on the current position and several positions ahead. That is, our probe index can be calculated by $\frac{1}{m}\sum_{j=1}^{m} I(p_i, p_i + j\frac{l}{m})$, where m is the number of times we extrapolate and l is the farthest reach of the extrapolations. This has the advantage of a more accurate extrapolation and could result in less comparisons. Figures 11 and 12 show that it is not the case, as `Interpolation Small Adaptive` is still better, if only by a small margin. This is perhaps due to the fact that the extrapolations with larger values of j in $I(p_i, p_i + j\frac{l}{m})$ is more accurate than those with smaller values of j, thus when taking the average of all extrapolations, the ones with small values of j contribute more to the inaccuracy of the estimate.

5 Conclusions and Open Questions

We showed that using binary search, the intersection algorithm `Small Adaptive` outperforms all the other intersection algorithms including `Sequential` the most

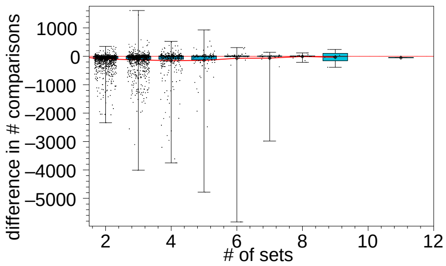

Fig. 11. `Extrapolate Many` (m=4, l=80) (low wins) vs `Interpolation Small Adaptive` (high wins).

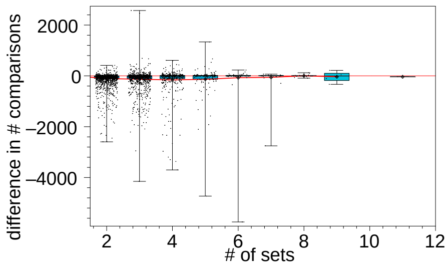

Fig. 12. `Extrapolate Many` (m=8, l=80) (low wins) vs `Interpolation Small Adaptive` (high wins).

recent intersection algorithm proposed in the theory community, which heretofore had not been compared in practice. Our results also confirm the superiority of `Small Adaptive` over all other algorithms as reported in [7], even on a data set substantially larger than the one used in that study. Considering variants of those intersection algorithms using interpolation search instead of binary search and galloping, we showed that for any fixed intersection technique, such as `Small Adaptive`, using interpolation search always improves the performance. Finally, we combine the two concepts of interpolation search and galloping to define the extrapolation search and several variants of it. Comparing the practical performance of these on the intersection algorithm `Small Adaptive`, we found one particularly effective, which results in an even better intersection algorithm, termed `Extrapolate Ahead Small Adaptive`, which at each step computes the position of the next comparison using the values of elements at distance l of each other, and which performs the best when $l = \lg n_i$.

Algorithm	# of comparisons
`Sequential`	119479075
`Adaptive`	83326341
`Small Adaptive`	68706234
`Interpolation Sequential`	55275738
`Interpolation Adaptive`	58558408
`Interpolation Small Adaptive`	44525318
`Extrapolation Small Adaptive`	50018852
`Extrapolate Many Small Adaptive` ($m = 4$, $l = 80$)	44119573
`Extrapolate Many Small Adaptive` ($m = 8$, $l = 80$)	44087712
`Extrapolate Ahead Small Adaptive` ($l = 50$)	44133783
Extrapolate Ahead Small Adaptive ($l = \lg n$)	43930174
`Extrapolate Ahead Small Adaptive` ($l = \sqrt{n}$)	44379689

Table 1. Total number of comparisons performed by each algorithm over thedata set. `Extrapolate Ahead Small Adaptive` with look-ahead range $l = \lg(n)$ is best.

For completeness we summarize the results across all algorithms on the whole data set in Table 1.

We would like to highlight some further experiments and open questions. First, it would be interesting to run the experiments over other data, such as the TREC corpus, particularly on the web slice of the collection. Second, to measure actual running times as opposed to the on number of comparisons alone. We expect that I/O and caching effects would have a significant impact on the reported times of each algorithm.

References

1. Ricardo A. Baeza-Yates. A Fast Set Intersection Algorithm for Sorted Sequences. In *Proceedings of 15th Annual Symposium on Combinatorial Pattern Matching* (CPM) 400-408, 2004.
2. Ricardo A. Baeza-Yates, Alejandro Salinger. Experimental Analysis of a Fast Intersection Algorithm for Sorted Sequences. In *Proceedings of 12th International Conference on String Processing and Information Retrieval* (SPIRE) 13-24, 2005.
3. Jérémy Barbay and Claire Kenyon. Adaptive Intersection and t-Threshold Problems. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA) 390-399, January 2002.
4. Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information processing letters*, 5(3):82–87, 1976.
5. Erik D. Demaine, Thouis R. Jones, Mihai Patrascu. Interpolation search for non-independent data. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), 529-530, January 2004.
6. Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), 743-752, January 2000.
7. Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Experiments on Adaptive set intersections for text retrieval systems. In *Proceedings of the 3rd Workshop on Algorithm Engineering and Experiments* (ALENEX), 91-104, January 2001.
8. V. Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4) 441-476, December 1992
9. W. Frakes and R. Baeza-Yates. *Information Retrieval.* Prentice Hall, 1992
10. G. Gonnet, L. Rogers, and G. George. An algorithmic and complexity analysis of interpolation search. *Acta Informatica*, 13(1) 39-52, 1980
11. Frank K. Hwang, Shen Lin. Optimal Merging of 2 Elements with n Elements. *Acta Informatica*, v.1, pp. 145-158, 1971
12. Frank K. Hwang, Shen Lin. A Simple Algorithm for Merging Two Disjoint Linearly-Ordered Sets. *SIAM Journal of Computing*, v.1, pp. 31-39, 1972.
13. Frank K. Hwang. Optimal Merging of 3 Elements with n Elements. SIAM Journal of Computing. v.9, pp. 298-320, 1980
14. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the 1st Symposium on Discrete Algorithms* (SODA),, 319-327, 1990
15. Y. Perl, A. Itai, and H. Avni. Interpolation search–A $\log\log n$ search. *CACM*, 21(7) 550-554, 1978