Adaptive Search over Sorted Sets

Biagio Bonasera[a], Emilio Ferrara[b], Giacomo Fiumara[a], Francesco Pagano[c], Alessandro Provetti[a,*]

[a] *Dept. of Mathematics and Informatics, Univ. of Messina, V.le F. Stagno D'Alcontres 31, I-98166 Messina, Italy*
[b] *Center for Complex Networks and Systems Research, School of Informatics and Computing. Indiana Univ., Bloomington, USA*
[c] *Dept. of Informatics, Univ. of Milan, Via Comelico, 39. I-20135 Milan, Italy*

Abstract

We revisit the classical algorithms for searching over sorted sets to introduce an algorithm refinement, called Adaptive Search, that combines the good features of Interpolation search and those of Binary search. W.r.t. Interpolation search, only a constant number of extra comparisons is introduced. Yet, under diverse input data distributions our algorithm shows costs comparable to that of Interpolation Search, i.e., $O(\log \log n)$ while the worst-case cost is always in $O(\log n)$, as with Binary search. On benchmarks drawn from large datasets, both synthetic and real-life, Adaptive Search scores better times and lesser memory accesses even than Santoro and Sidney's Interpolation-Binary Search.

Keywords: Sorting, Searching sorted sets

1. Introduction

We revisit the classical algorithms for searching over sorted sets to introduce a new algorithm, called Adaptive Search (AS), that combines the good features of Interpolation search and those of Binary search [1].
The membership problem can be formally defined as follows.
instance:

- $\mathcal{S} = \{a_1, a_2, ..., a_n\}$, a set of n distinct, sorted elements, with $a_i < a_{i+1}$, $1 \leq i \leq n-1$;
- an element *key*

question: Does *key* belong to the set represented by $\mathcal{S}$ ($key \in \mathcal{S}$) ?

There exist two classical algorithms for searching over sorted sets: binary search (BS) [1] and interpolation search (IS) [2]; both take advantage of the ordering of the instance to minimize the number of keys that must be accessed.

*Corresponding author
Email addresses: ferrarae@indiana.edu (Emilio Ferrara), gfiumara@unime.it (Giacomo Fiumara), francesco.pagano@unimi.it (Francesco Pagano), ale@unime.it (Alessandro Provetti)

In BS, the worst-case computational cost is $O(\log n)$; this result is independent of data distribution over the instance. Notice that in search the worst-case is rather important as it corresponds to an unsuccessful membership query.

Vice versa, the Interpolation Search algorithm is more efficient than BS when the elements of $\mathcal{S}$ are distributed uniformly or *quasi-uniformly* [1] over the $[a_1, a_n]$ interval; the computational cost is in $O(\log \log n)$.

Unfortunately, Interpolation Search degrades to $O(n)$ when data is not uniformly distributed (in the sense above). This is particularly inconvenient when searching over indexes of large databases, where it is crucial to minimize the number of accesses[2].

In this work we propose an algorithm, called Adaptive Search (AS) that refines Interpolation Search and minimizes the number of memory accesses needed to complete a search. AS is **adaptive** to the values by means of a *mixed* behavior: it combines the independence from the distribution of BS with the good average costs of IS.

W.r.t. Interpolation search, AS requires only a constant number of extra comparisons. Yet, under several relevant input data distributions our algorithm shows average case costs comparable to those of interpolation, i.e., $O(\log \log n)$, while the worst-case cost remains in $O(\log n)$, as with Binary search.

Comparison with a more recent literature is also encouraging: both on synthetic and real datasets AS has better times and lesser memory accesses than Santoro and Sidney's Interpolation-Binary Search [3]. Also, it is easier to implement and more broadly applicable that the approach of Demaine et al.[4] to searching non-independent data.

2. The Adaptive Search algorithm

Given an ordered set $\mathcal{S}$, allocated on an array A, and an element *key* that is searched, we define the following:

A[bot]: the minimum element of the subset (at the beginning, $bot = 1$);

A[top]: the maximum element of the subset (at the beginning, $top = |\mathcal{S}|$);

A[next]: interpolation element, i.e. what IS would choose, and

A[med]: the el. halfway between *bot* and *top*, i.e., what BS would choose.

Our algorithm consists, essentially, of a while cycle. At each iteration, we consider $\mathcal{S} = \{A[bot], .., A[top]\}$ and we set:

$$next = bot + \left\lfloor \frac{key - A[bot]}{A[top] - A[bot]} * (top - bot) \right\rfloor$$

Variable *next* defined above contains the index value that bounds the array segment on which our AS algorithm will recur on. As with interpolation, the instance is now *clipped:*

[1] By quasi-uniform data distribution we intended, informally, that the distance between two consecutive values of $\mathcal{S}$ does not vary much.

[2] In this discussion we do not consider the advanced techniques, viz. the exploitation of locality, that underlie search over large database indexes.

$$\mathcal{S}' = \begin{cases} \{A[bot], ..., A[next]\} \text{ if } A[bot] \leq key \leq A[next] \\ \\ \{A[next], ..., A[top]\} \text{ otherwise} \end{cases}$$

To do so, we set the new boundaries of the segment containing $\mathcal{S}'$:

$$\begin{cases} top = next \text{ if } A[bot] \leq key \leq A[next] \\ \\ bot = next \text{ otherwise} \end{cases}$$

The computation is now restricted to the segment that would have been considered by IS. Next, the median point is computed over such restricted segment, *rather than on the whole input.* Vice versa, if interpolation returns a shorter interval than BS would have, we keep the result of the interpolation step:

if $|\mathcal{S}'| > \frac{|\mathcal{S}|}{2}$ then $next = med = bot + \frac{top-bot}{2}$;
elseif $key = A[next]$ then key is found and we terminate;
elseif $key > A[next]$ then $bot = next + 1$;
else $top = next - 1$ (must be $key < A[next]$).

At the end of the iteration, $\mathcal{S}' = \{A[bot], ..., A[top]\}$, and, clearly, $|\mathcal{S}'| < \frac{|\mathcal{S}|}{2}$. Finally:

if $A[bot] < key < A[top]$ then iterate search on $\mathcal{S}'$;
else $key \notin \mathcal{S}$ and we terminate with *no.*

From the point of view of computational costs, we could summarize the following: our algorithm may spend up to double number of operations than IS in carefully finding out the best halving of the search segment, which in turn will mean that less iterations shall be needed to complete. By means of standard cost analysis techniques, we have the following results:

- Best Case: *key* is found, with a constant number of comparisons: $\Theta(1)$;
- Worst Case: the intervals between values are unevenly distributed; hence, the interval found by the BS technique is always the shortest. As a result, AS will execute essentially the same search as BS, with equal $O(\log n)$ time complexity (but more operations at each level), and
- Average Case: we consider the average case to be when there is some degree of uniformity in the distance between two consecutive values of $\mathcal{S}$. In such cases, AS executes exactly as IS so its cost is in $O(\log \log n)$.

3. Relation with literature

Only after our solution was conceived and implemented, have we become aware of an earlier work by Santoro and Sidney [3] who devised a similar solution that combines (but does not *blend)* together interpolation and binary search. Although the asymptotic complexity is the same, there are some marked differences between their solution and ours, let's discuss them now.

Santoro-Sidney's algorithm, called Interpolation-Binary Search, is based on the idea that interpolation search is useful, from the point of view of costs, only when the array searched

is larger than a given threshold. When the considered array segment is smaller than a user-defined threshold, binary search is applied unconditionally. Vice-versa, above the threshold an interpolation search step is applied, followed eventually by a binary search step.

Unlike IBS, our algorithm makes, at each level of its iteration, a choice about which *clipping* of $\mathcal{S}$ to apply. Hence, it is possible to show that for any input AS will not take more elementary operations than IBS.

We have sought a statistical confirmation of this fact by running a set of experiment over random-generated ordered sets; the results are presented in detail in the next Section[3]. We limited the testing of IBS to queries with parameter $\theta = 2$, which the authors suggested would work best. For all parameter settings and for all data distributions considered AS outperformed IBS albeit the difference could sometimes be statistically insignificant.

Two other works that address search over sorted sets have considered slight variations of the specification, that of Melhlhorn and Tsakalidis [5] and that of Demaine et al. [4]. The former considered an extended data structure, the Interpolation Search Tree (IST) to optimize the dictionary operations, not just search, over the sorted set. As such, their solution is not comparable to ours as it seeks to optimize insertion and deletion times rather than speed up search.

The latter, i.e., Demaine's interpolation search for *non independent* data is also not directly comparable to our work, but deserves a careful analysis. They define a deterministic metric of "well-behaved" or *smooth* data that enables searching along the lines of interpolation search. Specifically, they define

$$\Delta = \frac{max(x_i - x_{i-1})}{min(x_i - x_{i-1})}$$

i.e., the ratio between the largest and smallest *gap* between two adjacent elements of $\mathcal{S}$, as the key parameter in measuring the well-behavedness of the input. A data structure is needed that maintains a dynamic data set, that evenly divide the interval $(x_1, ..., x_n)$ into *n bins,* named $B_1, \ldots B_n$; each of them represents a range of size $\frac{x_n - x_1}{n}$.

Each bin B_i stores in a balanced binary search tree (BST) its elements, *plus* the nearest neighbors above and below that set. Hence, searching for an element *key* proceeds by interpolating on *key* to find which B_i it may lay in, i.e.,

$$i = \frac{(key - x_1)}{(x_n - x_1)}$$

then performing a search in the BST associated to B_i. For their solution, Demaine et al. prove the following results:

- the worst-case search time is $O(\log n)$ and thus $O(\log \min\{\Delta, n\})$, and
- the algorithm reproduces the $O(\log \log n)$ performance of interpolation search on data drawn independently from the uniform distribution.

[3]The instances and the test times are available from the companion Web site.

4. Experimental validation

We have implemented AS, along with the other algorithms mentioned so far, in order to test its efficiency, on real data, vis-à-vis those in the literature. The testing platform consists of a Java implementation running on a PC with JRE 1.7, Windows 2003 server R2, dual Opteron CPU with 4GBs of RAM. The tests consisted of running a number of searches corresponding to 1/1000 of the size of the dataset; keys where randomly chosen, with at least 80% of them successful. The results were normalized w.r.t. the number of queries.

4.1. Validation across distributions

As a first step, we considered random-generated benchmark instances (ordered arrays) of Java double data type, double-precision 64-bit IEEE 754 floating point values. Instances were randomly generated, with the following distribution types

1. *uniform sparsity:* the gap between two consecutive values is fixed across the instance. As unrealistic as it is, this case is useful in assessing whether AS introduces overheads.
2. *increasing sparsity:* the gap is actually growing, so the elements towards the end (i.e., the highest integer values) are more distant from each other than those at the beginning.
3. *stepwise sparsity:* the instance has zones with distinct, but fixed, gap sizes; the gap size grows towards the end of the array.
4. *Paretian:* the "80-20" rule applied to the summation of the values inside the instance, i.e., the summation of the first 80% of elements is equal to the summation of the last 20%.

For each parameter setting we generated and tested 10 random instances, then computed the average. Also, values are normalized w.r.t. the number of queries, so as to make them comparable across instance sizes. The results, presented in Table 1 compare the number of accesses, iterations and times of the four algorithms we considered.

On aggregate, AS outperformed IBS2 as well, albeit the difference could sometimes be statistically insignificant. The distributions were designed to stress-test AS in an unfavorable setting, where quicker implementations of BS could easily make up for the extra number of iterations. Even though on uniform- and increasing-sparsity instances Binary search can still run slightly faster than AS, on aggregation AS yields a huge advantage over all other algorithms, especially in terms of number of accesses and iterations.

The full benchmark results and the source codes (in Java) will be made available on a dedicated Web site[4].

4.2. A real-life benchmark dataset

To perform our analysis on real-life data with mixed or alternating distributions we used a public dataset on Facebook friendship released by Gjoka [6]; it contains a graph of about 957 thousands vertices (each representing a user) and 58.4 millions edges (each representing a friendship relation). Since each user is identified by a unique integer, and the dataset is ordered by user-id, it represents an ideal benchmark for testing our Adaptive Search algorithm as it gives to us one instance of about 1 million ordered integers. Also, the dataset can be split up in 9 distinct sub-instances of 100k elements each. The collected user-ids depend on several factors and human

[4] `http://logic-ai.di.unimi.it/`

		Accesses		Iterations		Times	
	Sizes :	10^5	10^6	10^5	10^6	10^5	10^6
Sparsity	Algo.						
uniform	*BS*	14,728	18,467	15,790	19,155	4.074,941	756,956
	IS	4,743	4,919	2,888	3,068	2.553,635	700,328
	*IBS*2	19,644	23,397	26,922	33,613	25.273,348	3.639,200
	AS	6,054	6,290	2,887	3,065	3.150,028	1.004,239
increasing	*BS*	14,741	18,479	15,828	19,156	462,659	831,015
	IS	19,613	26,619	18,906	25,978	948,907	2.994,142
	*IBS*2	19,338	23,502	22,581	29,177	1.345,126	3.499,578
	AS	11,198	12,160	5,460	6,016	596,080	1.744,790
stepwise	*BS*	14,795	18,505	15,957	19,171	445,794	753,501
	IS	232,945	329,222	256,465	351,515	10.386,056	35.041,154
	*IBS*2	20,304	24,202	24,777	31,096	1.485,665	3.604,115
	AS	12,055	12,968	6,129	6,708	652,009	1.505,453
Paretian	*BS*	14,793	18,476	15,917	19,157	457,496	916,074
	IS	17,536	21,702	16,028	20,209	839,989	2.768,519
	*IBS*2	20,252	24,180	25,339	31,904	1.509,791	3.900,253
	AS	10,338	11,003	5,097	5,536	564,157	1.516,632

Table 1: Averaged and normalized benchmark values over random instances with distinct data distributions. Times are in milliseconds.

intervention, e.g., users *leaving* Facebook and thus having their ids removed, so subinstances turn out to have distinct data distributions. Moreover, the gaps between two consecutive user-ids depends also on how the sample was collected.

These intuitions are confirmed by statistical analysis of the distribution of the gaps w.r.t. a *null model* generated with gaps of the same average and standard deviation. For the whole dataset we found that Spearman's rank correlation coefficient is equal to $4.95 \cdot 10^{-5}$ and Pearson's to $1.66 \cdot 10^{-4}$; this indicates that the distribution is *nonuniform.*

We used the same platform and the same set-up as before for the testing; the first test considered the whole Gjoka's dataset and the aggregated results (averaged over 10 runs) are in Table 2. As per the synthetic benchmarks, we ran a number of searches corresponding to 1/1000 of the size of the dataset; keys where randomly chosen, with at least 80% of them successful.

Algorithm	*Accesses*	*Iterations*	*Time*(*ms*)
BS	18,439	19,136	6,236,787
IS	501,346	499,730	74,035,808
*IBS*2	24,097	31,474	28,205,959
AS	8,349	4,044	4,791,845

Table 2: Benchmarks values over Gjoka's dataset

Subsequently, we have sought to confirm these results over similar datasets having diverse value distributions. To do so, we repeated the test on 9 sub-instances of Gjoka's, each corre-

sponding to 100k consecutive keys, i.e., positions (not values) 0–99.999, 100.000–199.999 and so on. In fact, the L_2 (Euclidean) distance from a uniform distribution of *gaps* between two consecutive values, varies widely. Nevertheless, our AS algorithm performed well on each subset, as it is reported in Table 3.

Instance	1	2	3	4	5	6	7	8	9
IS	1,662	1,473	1,494	1,463	1,488	1,470	1,483	1,487	1,489
BS	621	522	3,623	300	300	300	300	300	300
IBS	2,177	2,028	1,951	2,044	2,043	2,051	2,057	2,047	2,068
AS	889	711	860	307	310	311	309	309	309

Table 3: Memory accesses over 9 subinstances of Gjoka's dataset

Instanceno.	1	2	3	4	5	6	7	8	9
IS	1,783	1,567	1,613	1,570	1,602	1,578	1,589	1,593	1,605
BS	422	351	3,454	100	100	100	100	100	100
IBS	2,834	2,633	2,466	2,630	2,618	2,636	2,644	2,643	2,661
AS	417	355	453	100	100	100	100	100	100

Table 4: Iterations over 9 subinstances of Gjoka's dataset

Instance no.	1	2	3	4	5
IS	1,180,339	433,851	474,331	447,672	453,487
BS	682,630	252,233	1,862,590	126,926	127,868
IBS	5,104,407	2,600,980	2,533,918	2,723,071	2,523,096
AS	427,276	326,454	393,219	135,930	135,005
Instance no.	6	7	8	9	$Sum(1..9)$
IS	453,342	481,395	71,925	70,068	4,066,410
BS	127,822	131,044	136,434	130,819	3,578,366
IBS	2,541,275	2,588,292	242,335	254,291	21,111,665
AS	134,669	136,642	141,063	140,683	1,970,941

Table 5: Times, in milliseconds over 9 subinstances of Gjoka's dataset

At this point of our validation, we can exclude that the results suffer from any possible *positive bias* of the benchmark.

5. Conclusions

Even though we have considered only the simplest instance of search, i.e., ordered sets of integers, it turns out that this case is of great practical interest when we consider large dataset extracted from, e.g., crawling Web pages or Online Social Networks, where users/resources are

identified by simple integer keys. This is notably the case with Facebook, which assign to each subscriber a user-id consisting of a *progressive* integer. On such type of data, our solution shows a marked improvement over the literature. The results of experiments described in the previous section lead us to draw the following conclusions:

1. The performances of our AS algorithm vis-à-vis those IS and BS are very good and improve as n grows;
2. The number of accesses needed by AS is less than those of BS. The cost analysis of IS suggests that on certain instances, i.e., when sparsity grows, our algorithm needs between $\log n$ and $2 \log n$ accesses.
3. our method for selecting the search interval succeeds in preventing the irregularities of data distribution from affecting performances; indeed, the number of accesses required remains $\cong \log \log n$.
4. while the asymptotic complexity of our AS algorithm is the same as Santoro's IBS, we have found that -on relatively diverse benchmarks- AS often needs half or less of the memory accesses than IBS.
5. even though we could not yet run a complete study on large datasets, we have indication that the results presented here are likely to be confirmed for search dictionaries (considered by [5, 7]).

An interesting open question is whether instances that elicit the worst case ($2 \log n$ comparisons) for AS can actually be found, and how likely they are to appear within real datasets.

Acknowledgments

Thanks to Minas Gjoka for making the dataset studied in this work available.

References

[1] T. H.Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 3rd Edition, The MIT press & McGraw-Hill, 2009.

[2] A. Andersson, C. Matsson, Dynamic interpolation search in $o(\log \log n)$ time, in: Proc. of Int'l Colloquium on Automata, Languages and Programs (ICALP), 1993, pp. 15–27.

[3] N. Santoro, J. B. Sidney, Interpolation-binary search., Inf. Process. Lett. 20 (4) (1985) 179–181.

[4] E. D. Demaine, T. R. Jones, M. Patrascu, Interpolation search for non-independent data., in: SODA: ACM-SIAM Symposium on Discrete Algorithms (SODA), SIAM press, 2004, pp. 529–530.

[5] K. Mehlhorn, A. K. Tsakalidis, Dynamic interpolation search, Journal of the ACM 40 (3) (1993) 621–634.

[6] M. Gjoka, M. Kurant, C. Butts, A. Markopoulou, Practical recommendations on crawling online social networks, IEEE Journal on Selected Areas in Communications 29 (9) (2011) 1872–1892.

[7] A. Andersson, T. Hagerup, J. Hastad, O. Petersson, Tight bounds for searching a sorted array of strings, in: SIAM J. on Computing 30:55, 2001, pp. 1552–1578.