Deterministic Algorithm for the t-Threshold Set Problem

Jérémy Barbay[1] and Claire Kenyon[2]

[1] Department of Computer Science,
University of British Columbia,
201-2366 Main Mall, Vancouver, B.C.
V6T 1Z4 Canada
[2] Laboratoire d'Informatique (LIX),
École Polytechnique,
91128 Palaiseau Cedex - France

Abstract. Given k sorted arrays, the *t-Threshold* problem, which is motivated by indexed search engines, consists of finding the elements which are present in at least t of the arrays. We present a new deterministic algorithm for it and prove that, asymptotically in the sizes of the arrays, it is optimal in the alternation model used to study adaptive algorithms. We define the *Opt-Threshold* problem as finding the smallest non empty t-threshold set, which is equivalent to find the largest t such that the t-threshold set is non empty, and propose a naive algorithm to solve it.
Keywords: adaptive algorithm, t-threshold-set, opt-threshold set.

1 Introduction

We consider search engines where *queries* are composed of k keywords, and where each keyword is associated to a sorted array of references to entries in some database. The answer to a t-threshold query is the set of references which appear in at least t of the k arrays [3]. The answer to an opt-threshold query is the non-empty set of references maximizing the number of arrays they appears in. The algorithms studied are in the *comparison model*, where comparisons are the only operations permitted on references.

The analysis of the complexity of those problems in the worst case, when only the number of arrays k and their sizes $n_1, \ldots, n_k$ are fixed, is trivial and does not permit to distinguish between most algorithms. We propose a *finer analysis* using a difficulty measure δ on the instances: we analyze the complexity of the problem in the worst case among instances of same number of arrays, size of arrays, and difficulty. This type of analysis was already performed on adaptive algorithms for the union problem [6], intersection problem [3, 6], several sorting problems [5, 8, 11], and for the computation of the convex hull [9].

The t-threshold problem has been studied before, as a generalization of the intersection problem, but the algorithm was complicated and of complexity $O(t\delta \log k \log \sum_i n_i)$, which is not optimal because of the $\log k$ factor, and because when the arrays are of very different sizes the complexity increases more quickly

than the lower bound [3]. Moreover this type of query is a parametrized relaxation of the intersection, and is not quite practical for general purpose search engines.

To answer those problems, in section 2, we give a simpler and better deterministic algorithm for t-Threshold, with time complexity $O(\delta \sum_{i=1}^{k} \log(n_i/\delta+1)+\delta k \log(k-t+1))$, which is, asymptotically in the sizes of the arrays, optimal for $t \geq k/2$ (no lower bound is known for $t < k/2$). We also discuss the notion of the *opt-threshold set* of an instance, defined as the smallest non-empty threshold set, or identically as the non empty t-threshold set with maximal t. This would permit opt-threshold queries, which seem more practical than conjunctive or even t-threshold queries for a search engine. Finally in section 3, we present some perspectives for the domain, and especially address the issue of the practical testing of those algorithms.

2 t-Threshold

Let $\mathcal{U}$ be a totally ordered space. Let $-\infty$ and $+\infty$ be such that all elements of $\mathcal{U}$ are strictly larger than $-\infty$ and strictly smaller than $+\infty$.

Definition 1 (Instance, Signature, t-Threshold Set [3, definition 4.1]). *An* instance *consists of k sorted arrays $A_1, \ldots, A_k$ of sizes $n_1, \ldots, n_k$, whose elements come from $\mathcal{U}$. Its* signature *is $(k, n_1, \ldots, n_k)$. An instance has signature* at most *$(k, n_1, \ldots, n_k)$ if it has signature $(k', n'_1, \ldots, n'_{k'})$ with $k' \leq k$ and $n'_i \leq n_i$ for all $i \leq k'$. The output of the* t-Threshold *problem is the set $T_t(A_1, \ldots, A_k)$ containing exactly the elements present in at least t arrays.*

Note that the 1-threshold set is the union of the arrays and that the k-threshold set is their intersection.

Example 1. For instance the following set of arrays forms an instance of signature $(4, 5, 7, 6, 5)$ where the first number corresponds to the number of arrays, and the following numbers correspond to the sizes of those arrays. The 1-threshold set is the union of all the arrays; the 2-threshold set is $\{3, 4, 5, 7, 10, 11\}$; the 3-threshold set is $\{5\}$; and the 4-threshold set is the intersection and is empty.

$A =$	3	4	5	6	7		
$B =$	5	6	7	10	11	12	13
$C =$	0	1	2	10	11	14	
$D =$	3	4	5	8	9		

2.1 Algorithm

The algorithm that we propose here for the t-threshold set uses an `unbounded search` algorithm, which looks for an element x in a sorted array A of unknown size, starting at position *init*. It returns a value p such that $A[p-1] < x \leq A[p]$, called the *insertion point* of x in A. This algorithm has already been studied before, it can be implemented using the doubling search and binary search algorithms [1, 3, 6, 7, 10], and is then of complexity $2\lceil \log_2(p - init) \rceil$, and can be

implemented directly [4] to improve the complexity by a constant factor of less than 2.

In [3, algorithm 3], we gave an algorithm to compute the t-threshold set which performed unbounded searches in parallel in all arrays. Its complexity was close to the lower bound in the worst case for $n = \sum_i n_i$ and k fixed, but not in the worst case for an arbitrary signature $(k, n_1, \ldots, n_k)$. Here, we give a different algorithm which performs unbounded searches one at a time, whose complexity is better for an arbitrary signature $(k, n_1, \ldots, n_k)$ fixed.

Given $t \in \{1, \ldots, k\}$, and k non-empty sorted sets $A_1, \ldots, A_k$ of sizes $n_1, \ldots, n_k$, algorithm 1 computes the t-threshold set $T = T_t(A_1, \ldots, A_k)$. For simplicity, we assume that all arrays contain the element $-\infty$ at position 0 and the element $+\infty$ at position $n_i + 1$.

The algorithm has two nested loops. The outer loop goes through potential elements of T (in variable m) in increasing order. Given a candidate element m, for each array i, the variable $contain_i \in \{\texttt{MAYBE}, \texttt{YES}, \texttt{NO}\}$ expresses the current knowledge of the algorithm regarding the question: "does m belong to A_i?". To answer the question "does m belong to T?", the algorithm looks for m in arrays marked `MAYBE` until either t arrays are marked `YES`, in which case we can conclude that $m \in T$, or $k - t + 1$ arrays are marked `NO`, in which case we can conclude that $m \notin T$. In each pass in the inner loop the algorithms searches for m in one array A_s which potentially contains m.

Invariant: at the start of the inner loop, for every i, p_i denotes the first potential position for m in A_i, i.e. it is known that $A_i[p_i - 1] < m$. To decide if m is present in an array A_s, the algorithm effects an unbounded search for m from the position p_s. The position returned is the *insertion* position of m: in particular all elements before this position are smaller than m, and so p_i is updated to this position, thus maintaining the invariant.

The algorithm updates m each time it enters the outer loop, at which point the goal is to compute t-threshold on the sets $A_i[p_i \ldots n_i]$. It would seem natural to take the next candidate m as $\min_i A_i[p_i]$, but one can do better than that to gain efficiency: observe that if m is in T, then any set of $k - t + 1$ arrays must contain at least one copy of m; hence it is enough to define m as $\min_{i \in H} A_i[p_i]$, where H is any set of arrays of size $k-t+1$. A heap data structure is appropriate for H, as it permits to maintain $\min_{i \in H} A_i[p_i]$ with a minimum of comparisons[1].

Once m is defined, the arrays whose index are not in the heap, which potentially contain m, are marked `MAYBE`, the arrays in the heap whose first element is equal to m are marked `YES` and removed from the heap, and the arrays still left in the heap, which cannot contain m, are marked `NO`.

Example 2. On the following set of arrays for instance, if $t = 3$ and $m = 5$, when the algorithm found it in B and D, looked for it but didn't find it in C, and will

[1] See remark 1 page 7 for a comment on the use of an array instead of a heap.

look for it in A, the internal state looks like this:

	MAYBE	$A =$	3	4	5	6	7			$p_A = 1$
	YES	$B =$	**5**	6	7	10	11	12	13	$p_B = 1$
	NO	$C =$	~~0~~	~~1~~	~~2~~	10	11	14		$p_C = 4$
→	YES	$D =$	~~3~~	~~4~~	**5**	8	9			$p_D = 3$

The arrow indicates the array whose marking has just been updated, the first column gives the markings and the last column the positions.

After finding $m = 5$ in A the algorithm will get out from the inner loop deciding that 5 is in the 3-threshold set. It will then increment p_A, p_B and p_D, and will choose another value for m:

- it will complete the heap to 2 indexes (here with D hence $H = \{C, D\}$),
- will set m to the smallest first element of those arrays (here $m{=}D_4 = 8$),
- will mark arrays out of the heap with MAYBE (here A and B),
- will mark D with YES and take it out from the heap,
- and finally will mark C with NO.

→	MAYBE	$A =$	~~3~~	~~4~~	~~5~~	6	7			$p_A = 4$
→	MAYBE	$B =$	~~5~~	6	7	10	11	12	13	$p_B = 2$
→	NO	$C =$	~~0~~	~~1~~	~~2~~	10	11	14		$p_C = 4$
→	YES	$D =$	~~3~~	~~4~~	~~5~~	**8**	9			$p_D = 4$

Algorithm 1 Threshold Set$(t, A_1, \ldots, A_k)$

Given k, $t \in \{1, \ldots, k\}$, and k sorted sets $A_1, \ldots, A_k$, all of which contain $+\infty$, the algorithm computes the t-threshold set $T = T_t(A_1, \ldots, A_k)$.

for all i **do** $p_i \leftarrow 1$ **end for**
$T \leftarrow \emptyset$; $H \leftarrow \emptyset$; $s \leftarrow 1$
repeat
  Complete H so that it contains $k - t + 1$ indices of arrays.
  $m \leftarrow \min\{A_i[p_i]$ s.t. $i \in H\}$
  for all $i \notin H$ **do** $contain_i \leftarrow$ MAYBE **end for**
  for all $i \in H$ s.t. $A_i[p_i] = m$ **do** $contain_i \leftarrow$ YES; remove i from H **end for**
  for all $i \in H$ s.t. $A_i[p_i] \neq m$ **do** $contain_i \leftarrow$ NO **end for**
  while #YES $< t$ and #NO $< k - t + 1$ **do**
    Let A_s be the next array marked MAYBE in the cyclic order.
    $p_s \leftarrow$ Unbounded Search(m, A_s, p_s)
    if $A_s[p_s] \neq m$ **then** $contain_s \leftarrow$ NO **else** $contain_s \leftarrow$ YES **end if**
  end while
  if #YES $\geq t$ **then** $T \leftarrow T \cup \{m\}$ **end if**
  for all i such that $contain_i =$ YES **do** $p_i \leftarrow p_i + 1$ **end for**
until $m = +\infty$
return T

2.2 Analysis

Any instance can be decomposed into blocks of consecutive elements, in order to put in perspective its structure and capture some aspect of its difficulty. We present here one way to do it with the t-partition certificate. This permits to analyze the complexity of the algorithm (Theorem 1), and to prove that, asymptotically in the sizes of the arrays, no other algorithm can perform better on instances of similar structures (see the lower bound in [3], improved in [1, chap. 4]).

Definition 2 (t-Partition-Certificate). *A* t-partition-certificate *is a partition of* $\mathcal{U}$ *into intervals such that each singleton interval is an element of* T_t*, and all other intervals have an empty intersection with at least* $k - t + 1$ *arrays.*

The *non-deterministic complexity* is the minimal number of comparisons needed by a non-deterministic algorithm to check the result of the instance. It is a weak lower bound on the complexity of a randomized or deterministic algorithm. For $t = k$, the t-threshold set is exactly the intersection, and the minimal size of a t-partition-certificate is exactly the non-deterministic complexity of the instance plus one [3], which forms a natural measure of difficulty.

Imagine a function which indicates, for each element $x \in \mathcal{U}$, either the indices of t arrays containing x (if x is in the t-threshold set) or the indices of $k - t + 1$ arrays not containing x (if x is *not* in the t-threshold set). For each element several answers are acceptable, but elements in the same interval of a partition-certificate of the instance will have common answers. The minimal number of times such a function *alternates* answers, for x scanning $\mathcal{U}$ in increasing order, is also the minimal size of a partition-certificate of the instance (minus one), which is called t*-alternation*.

Definition 3 (t-Alternation δ_t). *The* t-alternation $\delta_t(A_1, \ldots, A_k)$ *of an instance* $(A_1, \ldots, A_k)$ *is the minimal number of intervals composing a* t*-partition-certificate of this instance.*

For general values of t the non-deterministic complexity nd does not correspond exactly to the minimum size δ of a t-partition certificate, but the values are still related:

- To any comparison-based proof, one can associate a t-partition-certificate, and the size of this partition-certificate is at most the size of the proof plus one, hence $\delta - 1 \leq nd$;
- conversely, from any t-partition-certificate, one can deduce a comparison-based proof whose length (number of comparisons) is at most $(k - t + 1)$ times the t-alternation minus 1, hence $nd \leq (k - t + 1)(delta - 1)$.

Example 3. The partition-certificates are easier to see if we represent each array by a line where elements of value x are drawn at abscissa x:

$A =$ 3 4 **5** 6 7
$B =$ **5** 6 7 10 11 12 13
$C =$ 0 1 2 10 11 14
$D =$ 3 4 **5** 8 9

For instance, the 4-threshold set of the last instance (A, B, C, D) is empty, and the 3-threshold set is $T_3(A, B, C, D) = \{5\}$:

- the partition $\{\,(-\infty, 3),\ [3, 10),\ [10, +\infty)\,\}$ is a 4-partition-certificate of minimal size for this instance, hence its 4-alternation is $\delta_4(A, B, C, D) = 3$;
- the partition $\{\,(-\infty, 3),\ [3, 5),\ \{\mathbf{5}\},\ (5, 8),\ [8, 10),\ [10, +\infty)\,\}$ is a 3-partition-certificate of minimal size for this instance, hence the 3-alternation of this instance is $\delta_3(A, B, C, D)$=6, and its 3-threshold set is $T_3(A, B, C, D)$=$\{5\}$.

Note that a t-partition certificate of minimal size can be generated by a greedy algorithm, but that it is not needed in an efficient algorithm to compute the t-threshold set. Algorithm 1 and its analysis improve the previous upper bound $O(\delta t \log k \log \sum_i n_i)$ [3] for the t-threshold set computational complexity. This is partly due to the algorithm, which performs one unbounded search at a time instead of performing several in parallel; and partly due to a better analysis.

Theorem 1. *Algorithm 1 performs* $O(\delta \sum_i \log(n_i/\delta{+}1) + \delta k \log(k - t + 1))$ *comparisons on an instance of signature* $(k, n_1, \ldots, n_k)$ *and* t*-alternation* δ*.*

This upper bound has to be compared to the lower bound $\Omega(\delta \sum_{i=2}^{k} \log n_i/\delta)$ (see [3, Corollary 3.1] improved in [1, chap. 4]), valid when $t \geq k/2$. The ratio is then of $1 + k \log(k-t+1)/\sum_{i=2}^{k} \log n_i/\delta$. This is equal to 1 if $t = k$, and smaller than 2 in most cases, in particular as long as $\forall_i n_i \geq \delta(k - t + 1)$. Otherwise it is $O(\log(k - t + 1))$, which is reasonably small for queries to search engines on the web, where k is the number of keywords input by the user.

Proof. Consider an instance $(A_1, \ldots, A_k)$ of signature $(k, n_1, \ldots, n_k)$ and of t-alternation δ. Let $(I_j)_{1 \leq\ j \leq \delta}$ be a corresponding t-partition-certificate. We call *search comparisons* the comparisons performed during an unbounded search, and *heap comparisons* the other comparisons.

During the execution of algorithm 1, the sequence of values taken by m is strictly increasing. For $j = 1, \ldots, \delta$ we say that a comparison is performed during *phase* j if $m \in I_j$ when the comparison is performed.

For each phase $j = 1, \ldots, \delta$, by definition either I_j is a singleton whose element is present in at least t arrays or there is at least $k - t + 1$ arrays which do not intersect I_j. As the algorithm searches the arrays in a fixed order, it performs at most one search in each array before to update m to a value not in I_j, and move to the next phase. Hence in each phase at most one search is performed for each array.

For each phase $j = 1, \ldots, \delta$ and each array A_i, let g_j^i be the increment of p_i between the instants before and after the unbounded search during phase j in array A_i. The algorithm performs $2 \log_2(g_j^i + 1)$ search comparisons in array A_i during phase j, and a total of $2 \sum_j \log_2(g_j^i + 1)$ during the execution of the algorithm. This is smaller than $2\delta \log_2(\sum_j g_j^i/\delta{+}1)$ by concavity of the function $\log_2(x + 1)$. These g_j^i elements are "jumped" by the algorithm and will not be compared anymore: so $\sum_j\ g_j^i{\leq} n_i$ and the algorithm performs less than $2\delta \log_2(n_i/\delta + 1)$ search comparisons in array A_i. Summing over all arrays

gives the upper bound $2\delta \sum_i \log_2(n_i/\delta+1)$ of the number of search comparisons performed by the algorithm.

The heap H contains at most $k-t+1$ elements, so each action on H costs at most $\log_2(k-t+1)$ comparisons. During a positive phase (when $m \in T_t$) there is only one iteration of the outer loop, hence a maximum of $k-t+1$ additions, of total cost at most $(k-t+1)\log_2(k-t+1)$, accounting for $2(k-t+1)\log_2(k-t+1)$ comparisons for additions and removals.

During a negative phase, there can be several iterations of the outer loop, each such that $\#\texttt{NO} = k-t+1$. Let's note $\#H$ the number of indexes in H during an iteration of the outer loop: it is the value of $\#NO$ before it enters the inner loop. At the next iteration of the main loop the algorithm must add $k-t+1-\#H$ indexes to complement the heap, which is exactly the number of negative unbounded searches it performed to obtain $\#\texttt{NO} = k-t+1$. As the algorithm performs at most $k-1$ unbounded searches per phase, the number of addition to the heap during such a phase is at most $k-1$, of total cost including removals at most $2(k-1)\log_2(k-t+1)$. Hence the algorithm performs in total less than $2\delta(k-1)\log_2(k-t+1)$ heap comparisons.

The total number of search and heap comparisons performed by the algorithm is smaller than $2\delta \sum_i \log_2(n_i/\delta+1) + 2\delta(k-1)\log_2(k-t+1)$, which is in $O(\delta \sum_i \log(n_i/\delta+1) + \delta(k-1)\log(k-t+1))$. □

Remark 1. Note that the use of an array instead of a heap for H would permit to save comparisons in many cases, in particular in positive phases where computing the min would cost only $k-t$ comparisons. But during a negative phase, there can be up to $k-1$ iterations of the external loop, each with only one unbounded search but also one min computation. In this case an array costs $(k-1)(k-t+1)$ comparisons while the heap costs only $2(k-1)\log_2(k-t+1)$. Hence the choice of a heap to implement H.

2.3 From *t*-Threshold to Opt-Threshold

The t-threshold set is a relaxation of the intersection: it is less constrained as t is getting smaller. By definition, the sets are then increasing and $T_k \subset \ldots \subset T_1$. For conjunctive queries whose corresponding intersection is empty, the $(k-1)$-threshold set would be a more informative answer. The same reasoning can be applied to any t-threshold query for $t > 1$: if the answer to the query is empty then the $(t-1)$-threshold query answer is more informative.

We define the opt-threshold set as the smallest non empty threshold set. It is always defined, as the 1-threshold set is never empty because it is the union of the arrays, whose sizes are required to be positive. The multiplicity of the instance is simply the value of t for which the opt-threshold set is the t-threshold set.

Definition 4 (Opt-Threshold Set). *The* multiplicity *$opt(A_1, \ldots, A_k)$ of an instance is the maximum number of arrays with non-empty intersection. The* Opt-Threshold Set *of an instance of multiplicity t is the sorted array $T_t(A_1, .., A_k)$.*

The opt-threshold set is the most adequate answer to a simple query composed of k words on an indexed search engine. When some elements of the database match all the words of the query then it corresponds to an intersection, otherwise it corresponds to the set of elements maximizing the number of words matched.

Example 4. The multiplicity of example 1 is $opt(A, B, C, D) = 3$, and the opt-threshold set is $T_3 = \{5\}$ (see example 3 for the proof).

A naive algorithm to solve this problem would be to use algorithm 1 to compute iteratively the t-threshold set for t decreasing from k to 2 till a non-empty threshold set is computed. This is better than performing a binary search or a doubling search on the optimal value of t, as the t-alternation can vary widely from one value of t to the next, and hence the complexity of algorithm 1.

Proposition 1. *The naive algorithm performs* $O(\delta(k-t+1)\sum_i \log(n_i/\delta+1))$ *comparisons on an instance of signature* $(k, n_1, \ldots, n_k)$*, multiplicity* t *and* t*-alternation* δ*.*

A more sophisticated algorithm would compute each t-partition certificate recursively from a $t+1$-partition certificate, till obtaining a non-empty threshold set, but this saves only a constant factor in the worst case complexity: such an algorithm can still be forced to search $k - t + 1$ times in the same array for elements of the same interval of a partition certificate.

The algorithm is optimal if $t = k$. Otherwise its complexity is at a factor of $O(k - t + 1)$ of the lower bound, which can be considered small for queries to search engines on the web.

3 Perspectives

An obvious perspective concerns the opt-threshold set problem: is the naive algorithm optimal ? Is the t-alternation for an optimal value of t an adequate measure of difficulty for this problem ? We thought for a while that there was an algorithm solving this problem with the same complexity than for the t-threshold set, but were proved wrong: the problem is still open...

The worst instances used in the lower bounds [3] are pathological: δ_t elements are in a number of arrays maximizing the number of comparisons needed to find it ($k-1$ for the intersection, and $t-1$ in general for the t-threshold set). A measure taking into account the *multiplicity* of each element would be more precise. For the intersection the optimal algorithm has to be randomized: for instance if an element is present in exactly half of the arrays, a randomized algorithm will look for it in 2 arrays on average before concluding, while a deterministic algorithm will look for it in $k/2$ arrays [2]. Those results do not apply to the t-threshold directly, whose analysis needs a more sophisticated generalization of the measure of difficulty.

Jason Hartline suggested an interesting extension of the t-threshold and opt-threshold problems, in which a weight is associated to each array, and a score is defined for each reference by summing the weights of the arrays containing it. The

problems consist then in returning references scoring above the threshold t, or the maximum score attained by a reference. It is easy to extend the algorithms to solve those new problems, and extending their analysis and lower bounds seems feasible.

More complex queries can be defined by combining t-threshold functions. We recursively define a *concept* as a t-threshold or an opt-threshold function on concepts or keywords. We define in turn a *concept query* by a concept, and we define the answer to such a query as the set of references matching this concept. Preliminary work shows that the lower bounds and algorithms studied on the t-threshold set problem can be generalized to those more complex queries.

Testing algorithms for the intersection is feasible on data from any actual search engine (see [7]). On the other hand, testing accurately algorithms for the t-threshold set or for opt-threshold set won't be so easy. With usual search engines which perform only intersections, users restrict their queries to a small number of keywords, in order to limit the risk to obtain an empty answer. A realistic test would require the implementation of a search engine using an opt-threshold set algorithm, and real testing by users.

Acknowledgements: Thanks to the anonymous reviewers for helping to point out a mistake concerning the opt-threshold set problem. Jérémy Barbay wishes also to thank the "Institut National de Recherche en Informatique et Automatisme" of France for funding his postdoctoral position, and Joël Friedman for inviting him at UBC.

References

1. J. Barbay. *Analyse fine: bornes infrieures et algorithmes de calculs d'intersection pour moteurs de recherche*. PhD thesis, Universit Paris-Sud, Laboratoire de Recherche en Informatique, Septembre 2002.
2. J. Barbay. Optimality of randomized algorithms for the intersection problem. In *2nd Symposium on Stochastic Algorithms, Foundations and applications*. LNCS, Springer-Verlag, 2003.
3. J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *Proceedings of the* 13^{th} *ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pages 390–399. ACM-SIAM, ACM, January 2002.
4. J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Information processing letters*, 5(3):82–87, 1976.
5. C. Cool and D. Kim. Best sorting algorithm for nearly sorted lists. *Communication of ACM*, 23:620–624, 1980.
6. E. D. Demaine, A. Lpez-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the* 11^{th} *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 743–752, 2000.
7. E. D. Demaine, A. Lpez-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, Lecture Notes in Computer Science*, pages 5–6, Washington DC, January 2001.
8. V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. ACM Computing Surveys, 1992. 24(4):441–476.

9. D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 1986. 15(1):287–299.
10. K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, chapter 4.2 Nearly Optimal Binary Search Tree, pages 184–185. Springer-Verlag, 1984.
11. O. Petersson and A. Moffat. A framework for adaptive sorting. *Discrete Applied Mathematics*, 59:153–179, 1995.