Competitive Search for Longest Empty Intervals

Peter Damaschke[*]

Abstract

A problem arising in statistical data analysis and pattern recognition is to find a longest interval free of data points, given a set of data points in the unit interval. We use the inverse length of the empty interval as a parameter in the complexity bounds, since it is small in statistically relevant cases. For sorted point sets we get nearly optimal strategies. While the asymptotic complexities are trivial, achieving an optimal number of operations appears to be difficult. Constant factors can be of practical interest for huge data sets. We derive deterministic and randomized upper and lower bounds. Matching bounds and smooth trade-offs between the different operations (reads, comparisons, subtractions) are open questions. For unsorted point sets, the complexity is at least linear. Therefore we also use statistical inference to get approximate solutions in sublinear time.

1 Introduction

Given a set of n data points in a finite-size part of a geometric space, we call a subset of this space (with prescribed shape) free of data points an empty region. Searching for largest empty regions is a natural problem in, e.g., data mining [10]. It has been considered for rectangles in the plane [1, 2, 5, 8, 11] and boxes in d dimensions. Usually, the complexity of algorithms is expressed as a function of input size n. However, empty regions are statistically relevant only if they are large compared to the expected size if the data point set were drawn from a uniform distribution. Then, large empty regions may be found faster than in the worst case. Thus it is sensible to measure complexity as a function of both n and a parameter inverse to the size of the empty region. Here we study, as a first step, the 1-dimensional case: empty intervals between n data points in the unit interval. While the worst-case complexity is trivially $\Theta(n)$, the parameterized problem has a different nature. Still, its optimal asymptotic complexity is easy to determine by standard arguments, but the exact number of operations appears to be a surprisingly difficult question. Constant factors make a difference in practice, when huge data sets are processed. Analyzing the number of operations (e.g., comparisons) without ignoring constant factors is quite common for sorting, searching, and order statistics.

We state our problem LONGEST EMPTY INTERVAL more formally. A sorted set of real numbers $0 = x_0 < x_1 < \ldots < x_n = 1$ is given. An *empty interval* is an interval delimited by two consecutive x_i, x_{i+1}. We can access x_i through index i in constant time. (The x_i are either stored in an array or delivered by an oracle.) Our goal is to find a longest empty interval, that is, one with largest difference $\max_i(x_{i+1} - x_i)$. This can be trivially done by n read operations (*reads* for short), subtractions, and comparisons, respectively, and linear time is optimal due to an obvious adversary argument. Define $r := 1/\max_i(x_{i+1} - x_i)$. Supposing that a "very long" empty interval is expected, with $r \ll n$, we want an algorithm that takes advantage of the small r.

Throughout the paper, logarithms are base 2. We call the x_k values *data points*. In our complexity bounds we neglect minor-order terms. To avoid clumsy notation we also silently suppress factors $1 + o(1)$ where $o(1)$ tends to 0 as n grows.

We show that LONGEST EMPTY INTERVAL can be solved optimally with $r \log(n/r)$ reads. However, in order to keep the number of other operations within $O(r \log(n/r))$ we need some more reads. We have to add factor 2 (deterministic) or 1.4427 (randomized). We also study the case of unsorted data point sets, called LONGEST EMPTY INTERVAL (UNSORTED). Amazingly, n and r almost switch their roles: We give an algorithm with roughly $n \log r$ comparisons, while the number of reads is trivially n. We remark that a rather obvious RAM algorithm using n equidistant buckets solves LONGEST EMPTY INTERVAL (UNSORTED) in $O(n)$ time, but for comparison-based algorithms $\Theta(n \log r)$ is optimal, and the simple scheme also fails for similar problems in higher dimensions. The problem is also known as *max gap* and has an $\Omega(n \log n)$ lower bound in the algebraic decision tree model [3, 9]. Our algorithms do not assume prior knowledge of r. Another practical advantage is their simplicity, however, several details leading to the constant factors are a bit tricky, and there remain gaps between the current upper and lower bounds. In the unsorted case, approximate solutions, i.e., large regions with few data points, can still be obtained in sublinear time. We give a grid-based method to analyze the performance of an obvious sampling method.

[*]Department of Computer Science and Engineering, Chalmers University, 41296 Göteborg, Sweden, email: `ptr@cs.chalmers.se`. Supported by the Swedish Research Council (Vetenskapsrådet), grant no. 2007-6437, "Combinatorial inference algorithms – parameterization and clustering".

The last section informally discusses extensions to other geometric set families.

We conclude the introduction with some motivations and further related literature.

In the sorted case one may argue that the longest empty interval *could have been* computed on the fly, when the set has been sorted, and this makes up a minor part of the calculations. But what if distances in our huge sorted point sets *have not been* computed earlier, simply because there was no interest in such analysis? Then we want to solve the *actual* problem as efficiently as possible. There may also arise machine learning problems where we know that some unknown "empirical" function is monotone, values are not explicitly stored but can be queried by experiments, and we are mainly interested in large jumps of this function. In fact, LONGEST EMPTY INTERVAL exhibits striking similarities to a well-known problem in combinatorial search: competitive group testing [6, 7].

In [4] we gave algorithms for finding at most s disjoint intervals of maximum total length that contain at most p data points (s, p are fixed parameters). Finding longest empty intervals in sorted point sets is part of the preprocessing. Then, it is proved that the optimal solutions are composed of such intervals from a certain candidate set whose size depends only on s and p, and it can be computed by dynamic programming. Only the time for preprocessing depends on n, therefore we save a significant fraction of the overall running time by log-time preprocessing. In range prediction applications as in [4], the data points come as previously sorted sets.

2 The Sorted Case

Theorem 1 LONGEST EMPTY INTERVAL *can be solved using $r\log(n/r)$ reads, and this bound is optimal.*

Proof. The optimality argument is omitted.

The proposed algorithm maintains, in a linked list, the ordered sequence of data points x_k already read. In every step we take two consecutive data points in this list with currently largest distance, say x_a and x_b, read the data point $x_{\lfloor (a+b)/2 \rfloor}$ and insert it in our list. We stop as soon as $a+1=b$. Since $x_b - x_a$ is the maximal distance in the sequence, we have found the longest empty interval at this moment.

To analyze the number of reads, think of this splitting process as a binary tree of segments of data points, in the obvious sense. One read is associated with every non-leaf node. Consider the tree upon termination of the algorithm. A *long* node represents an interval of length at least $1/r$, other nodes are called *short.* We prune the tree as follows. Any pair of short leaf siblings is removed, making their parent a leaf. The parent node is always long, since the algorithm has considered intervals by decreasing lengths and stopped at $1/r$. After pruning, one read is associated with every long node. Since the leaves represent pairwise disjoint intervals, at most r leaves are long nodes. Every long non-leaf node is on some path from the root to some long leaf (otherwise we could continue pruning). It follows that all reads are associated with nodes on paths to at most r of the leaves. The path length in the tree is trivially bounded by $\log n$. At most r nodes have depth $\log r$, and the remaining subpaths from level $\log r$ to the leaves have length at most $\log n - \log r$. Since at most r such paths exist, we get the claimed bound. $\square$

However we have to worry about the other operations, too. Upon every read we also need two subtractions to get the lengths of the two new intervals. Thus, the method needs $2r\log(n/r)$ subtractions. The catch is that we need to know the longest interval for the next split. Using a heap for at most r interval lengths (the current leaves of the tree), we make, for every read, up to $4\log r$ length comparisons to include the two new interval lengths in the heap (and also $5\log r$ copy operations in the heap). Thus the method in this form costs $4r\log r\log(n/r)$ comparisons. An optimal number of reads is good if data access is very expensive, e.g., if data reside in some external memory. But usually the costs of reads, comparisons, and subtractions should be similar. Thus we will next aim at $O(r\log(n/r))$ operations in total, with small constant factors. We now propose a method that still uses binary search, but on the range of values rather than indices. The number of reads is only doubled.

Theorem 2 LONGEST EMPTY INTERVAL *can be solved using $2r\log(n/r)$ reads, $2r\log(n/r)$ comparisons, and $O(r)$ subtractions.*

Proof. In the jth phase ($j = 1, 2, 3\ldots$), we declare every $i/2^j$ (i odd, $0 < i < 2^j$) a *grid point.* For every new grid point g, binary search finds k with $x_k \le g < x_{k+1}$. We call $[x_k, x_{k+1}]$ the *empty interval around g.* We compute the lengths of empty intervals around all grid points and determine the longest one.

Let p be the exponent with $1/2^p \le 1/r < 1/2^{p-1}$. Then, a longest empty interval (of length $1/r$) contains a grid point in phase p. Since we have computed the lengths of empty intervals around all grid points, $1/r$ is among these values, and it is the maximum length. Since every empty interval without grid points is entirely between two consecutive grid points, its length is at most $1/2^p \le 1/r$, hence we know at this moment that a longest empty interval is found.

In order to find the empty interval around any new grid point introduced in phase j, it suffices to do binary search on the data points between the two neighbored old grid points. (Recall that we already know the indices of the leftmost and rightmost data point in this range.)

Since all these search spaces do not overlap, we perform 2^{j-1} binary search procedures on a total of n elements in phase j. By concavity of log, the total number of search steps in phase j is maximized if all search spaces have equal size $n/2^{j-1}$. Summation over all phases yields the number of operations: $\sum_{j=1}^{p} 2^{j-1}(\log \frac{n}{2^{j-1}} + O(1)) = 2^p(\log n - p + O(1))$.

The worst case is $1/r < 1/2^{p-1}$, with an arbitrarily small difference. Now $2^p < 2r$ yields the upper bound of $2r\log(n/r)$ search steps. Every search step requires one read and one comparison. Subtractions are only used to compute the lengths of empty intervals around the $O(r)$ grid points. Only $O(r)$ comparisons are needed to determine the maximum length among them. $\square$

The worst case in the above analysis suggests that randomization on the grid size might improve the constant factor in the number of reads. In fact, we obtain:

Theorem 3 Longest Empty Interval *can be solved using an expected number of* $(1/\ln 2)r\log(n/r)$ *reads,* $(1/\ln 2)r\log(n/r)$ *comparisons, and* $O(r)$ *subtractions. (Remark:* $1/\ln 2 < 1.4427$*.)*

Proof. We sample a random $t \in [1, 2)$ according to some probability density function q that we specify below, multiply the grid point distances by t, and continue deterministically as in Theorem 2. For formal clarity: We construct the grid on an interval of length t including $[0, 1]$, but then we ignore all grid points outside $[0, 1]$.

As in Theorem 2, let p be the exponent with $1/2^p \leq 1/r < 1/2^{p-1}$. If $t \leq 2^p/r$ then we also have $t/2^p \leq 1/r < t/2^{p-1}$. Now we argue, as in Theorem 2, that an empty interval of length $1/r$ is identified in phase p. However, since grid points outside the unit interval are ignored, we perform only $2^{j-1}/t$ binary search procedures on disjoint subsets of a set of n elements, in phase j. The total number of search steps in phase j is maximized if all search spaces have equal size $tn/2^{j-1}$. Summing over all phases we get $\frac{1}{t}\sum_{j=1}^{p} 2^{j-1}\left(\log \frac{tn}{2^{j-1}} + O(1)\right) = \frac{2^p}{t}(\log n - p + O(1))$.

If $t > 2^p/r$ then $t/2^{p+1} \leq 1/r < t/2^p$. Still we can argue as above, but with $p+1$ in the role of p, which yields the result $(2/t)2^p(\log n - p + O(1))$.

Define $x := 2^p/r$, and note that $1 \leq x < 2$. We express the number of reads as $(x/t)r\log(n/r)$ if $t \leq 2^p/r$, and $2(x/t)r\log(n/r)$ if $t > 2^p/r$. Specifically, we use density $q(t) = 1/(t\ln 2)$ for sampling. (In fact, q is a density function, due to $\int_1^2 dt/t = \ln 2$). Thus we obtain in front of $r\log(n/r)$ the following expected factor: $x\left(\int_1^x \frac{1}{t}q(t)dt + 2\int_x^2 \frac{1}{t}q(t)dt\right) = \frac{x}{\ln 2}\left(\int_1^x \frac{1}{t^2}dt + 2\int_x^2 \frac{1}{t^2}dt\right) = \frac{x}{\ln 2}\left(\frac{1}{1} - \frac{1}{x} + \frac{2}{x} - \frac{2}{2}\right) = \frac{1}{\ln 2}$. The other bounds follow as in Theorem 2. $\square$

It remains open, even in the randomized case, whether $r\log(n/r)$ reads are sufficient together with $O(1)r\log(n/r)$ other operations. More generally, a smooth trade-off between reads and comparisons would be nice. Apparently this would require to "bridge" somehow between binary search on indices and values.

3 The Unsorted Case

In order to solve Longest Empty Interval (unsorted), we have to read all n data points x_i, since any missing x_i could fall into the largest empty interval of the rest of the data set. Hence the number of reads is not interesting. We focus on comparisons and subtractions. Trivially, sorting the x_i solves the problem by $n\log n$ comparisons and n subtractions, but for $r \ll n$ we can avoid sorting and save almost a $\log n$ factor:

Theorem 4 Longest Empty Interval (unsorted) *can be solved using* $n(\log r + 3) + 4r$ *comparisons and* $O(r)$ *subtractions, and* $n\log r$ *is a lower bound for the number of comparisons.*

Proof. Again we perform binary search on $[0, 1]$, inserting grid points $i/2^j$ (i odd) in phase j, but this time we divide the data points recursively into subsets situated between any two neighbored grid points. If j phases are needed, this costs altogether nj comparisons between data points and grid points. After each phase we check which of the mentioned subsets became empty. This step is simple: To every new grid point we attach a discrete variable that tells us whether some data point went to the left and to the right subset. As soon as we get some empty subset(s) in our partitioning, we know that the largest empty interval is formed by the rightmost data point in some nonempty subset and the leftmost data point in the next nonempty subset to the right. All candidates are found by n comparisons in total, because the linear order of subsets is known, and minimum resp. maximum search is done on disjoint subsets. If j is the final phase, at most 2^j subtractions yield the interval lengths, and 2^j further comparisons return the result.

Once more, let p be the exponent with $1/2^p \leq 1/r < 1/2^{p-1}$. We detect an empty subset when two grid points hit the largest empty interval, which happens in phase $j \leq p+1$. Hence $j < \log r + 2$, furthermore $2^j \leq 2^{p+1} < 4r$. Summation of comparisons in binary search and candidate selection yields the bound. The lower bound argument is omitted. $\square$

It is not possible to find exactly the largest empty interval in sublinear time. On the other hand, for statistical inference and data mining, a relaxed optimization goal is still appropriate: Find a large interval containing at most a given fraction of data points (as in [4]). Then we can sample from the data points and estimate the point numbers in intervals. The question is how reliable the inferred "sparse" intervals are.

For technical reasons we further modify the problem statement in two ways, without changing its "essence": Firstly, instead of a huge set of data points we assume an unknown *continuous* probability distribution on $[0, 1]$ to sample from. Secondly, instead of searching for an interval with given probability mass q and maximum length L, we search for an interval with given L and minimum q. (Note that the length of an interval is "observable", whereas probability mass can only be estimated.) Now we can measure the performance simply by the competitive ratio q_A/q, where q_A is the probability mass of the interval selected by the algorithm, and q is the minimal probability mass among all intervals of length L. We get the following trade-off, with $\delta = q_A/q - 1$:

Theorem 5 *Given some $L < 1$ and an unknown probability distribution on the unit interval, let q be the minimum probability mass of the intervals of length L. Then one can, in $O(m \log m)$ time, sample an expected number of m points and specify an interval of length L with probability mass smaller than $(1 + \delta)q$, subject to an error probability less than* $\frac{h}{q}(1 + 1/\delta)\exp(-mq\frac{(\delta-2/h)^2}{4+2\delta})$, *for any positive δ and h.*

Proof. Sample m points and take an interval A of length L with least number of sampled points. The probability of $q_A > (1+\delta)q$ is limited by a union bound, applied to a finite "grid" of intervals G so that every too heavy A contains some G. Details are omitted. $\square$

After a slight refinement of the proof we can replace factor $\frac{h}{q}$ with the smaller $\frac{h}{L}$. The free parameter h may be choosen so as to minimize the error bound. In particular, taking $h = mq\delta$ gives the best asymptotics for large m. Here we obtain $m(1 + \delta)\exp(-mq\frac{\delta^2}{4+2\delta})$. For a given sample size m, the bound can also be used to compute $1 + \delta$ that are achievable with high probability, depending on q. For very small q, these δ are large, however, the "absolute" probability mass $q(1+\delta)$ of the returned interval is more interesting than the competitive ratio in this case.

4 Further Research: Other Geometric Set Families

It remains to improve the various complexity and probability bounds and to close the gaps. In this paper we have focused on intervals, but the ideas are much more general. In the final remarks we sketch some extensions to be considered in further research. The k longest empty intervals, as needed in [4], can be found by slight modifications of our strategies. Bounds are similar, when $1/r$ is redefined as the length of the k-th longest empty interval. For analoguous problems in d-space, e.g., the largest empty (axis-parallel) box in $[0, 1]^d$, a scheme as in Theorem 4 still works, with some relaxation: Since we lack total order, we cannot get optimal results in $o(n \log n)$ time, but $(1 - 1/s)$-approximations in $O(n(\log r + \log s))$ time, where $1/r$ is the volume of the result. Hidden factors depend on d. The sampling approach of Theorem 5 works similarly for other geometric set families $\mathcal{F}$, too, once there is an efficient algorithm for finding large sets in $\mathcal{F}$ with few data points. A technical difficulty of the analysis is to define suitable "grids": For any probability distribution we need a finite family $\mathcal{G}$ so that every set of $\mathcal{F}$ has a subset in $\mathcal{G}$ with small loss of probability mass. Granularity can be chosen so as to minimize the union bound. The cardinality of $\mathcal{G}$ appears as a factor, but loss affects the negative exponent in the exp term. For unions of s intervals (s fixed) we can proceed as in Theorem 5, only the loss is multiplied by s, since we cut intervals at each end, and the cardinality of $\mathcal{G}$ goes as $(h/q)^{2s-1}$. For boxes in d-space we can simply use slices in the d axis directions. For families $\mathcal{F}$ like disks or balls, grid construction is possible, too, but more complex, since "heavy" borders of sets in $\mathcal{F}$ must be sliced.

References

[1] A. Aggarwal, S. Suri. Fast algorithms for computing the largest empty rectangle, *Symp. on Comput. Geometry 1987*, 278-290

[2] M.J. Atallah, G.N. Frederickson. A note on finding a maximum empty rectangle, *Discrete Applied Math.* 13 (1986), 87-91

[3] M. Ben-Or. Lower bounds for algebraic computation trees. *15th ACM STOC 1983*, 80-86

[4] A. Bergkvist, P. Damaschke. Fast algorithms for finding disjoint subsequences with extremal densities, *Pattern Recognition* 39 (2006), 2281-2292, abstract in: *16th ISAAC 2005*, *LNCS* 3827, 714-723

[5] B. Chazelle, L.R.S. Drysdale, D.T. Lee. Computing the largest empty rectangle, *SIAM J. Comp.* 15 (1986), 550-555

[6] D.Z. Du, H. Park. On competitive group testing, *SIAM Journal Comp.* 23 (1994), 1019-1025

[7] D.Z. Du, G. Xue, S.Z. Sun, S.W. Cheng. Modifications of competitive group testing, *SIAM Journal Comp.* 23 (1994), 82-96

[8] J. Edmonds, J. Gryz, D. Liang, R.J. Miller. Mining for empty rectangles in large data sets, *Theor. Computer Science* 296 (2003), 435-452

[9] D.T. Lee, Y.F. Wu. Geometric complexity of some location problems, *Algorithmica* 1 (1986), 193-211

[10] B. Liu, L.P. Ku, W. Hsu. Discovering interesting holes in data, *15th IJCAI 1997*, 930-935

[11] M. Orlowski. A new algorithm for the largest empty rectangle problem, *Algorithmica* 5 (1990), 65-73