

Sort Me if You Can: How to Sort Dynamic Data

Aris Anagnostopoulos^{1*}, Ravi Kumar², Mohammad Mahdian², and Eli Upfal^{3**}

¹ Sapienza University of Rome, Rome, 00185, Italy. aris@cs.brown.edu

² Yahoo! Research, Sunnyvale, CA 94089, USA.

{[ravikumar](mailto:ravikumar@yahoo-inc.com), [mahdian](mailto:mahdian@yahoo-inc.com)}@yahoo-inc.com

³ Brown University, Providence, RI 02912, USA. eli@cs.brown.edu

Abstract. We formulate and study a new computational model for dynamic data. In this model the data changes gradually and the goal of an algorithm is to compute the solution to some problem on the data at each time step, under the constraint that it only has a limited access to the data each time. As the data is constantly changing and the algorithm might be unaware of these changes, it cannot be expected to always output the exact right solution; we are interested in algorithms that guarantee to output an approximate solution. In particular, we focus on the fundamental problems of sorting and selection, where the true ordering of the elements changes slowly. We provide algorithms with performance close to the optimal in expectation and with high probability.

1 Introduction

In the classic paradigm, an algorithm received all the input at the start of the computation and computed a function of that input. As computing became more interactive, researchers developed the theory of online algorithms, focusing on the tradeoff between the timely availability of the input and the performance of the algorithm. In this paper we study another important aspect of online, interactive computing: computing and maintaining global information on a data set that is constantly *changing*. While algorithms and models to study dynamic data have been in vogue, our work formulates and studies a new model of computing in the presence of constantly changing data.

For concreteness we present our work through one specific motivation, the popular online voting website Bix (bix.com), owned by Yahoo!; this partially inspired us to study the particular problem of sorting. We comment later on more general applications. The Bix website hosts online contests for various themes such as the most entertaining sport or the most dangerous animal or the best presidential nominee, in which users vote to select the best amongst a pre-specified set of candidates. For a given contest, Bix displays a pair of candidates to a user visiting the website and asks the user to rank-order this pair. As the contest progresses, Bix aggregates all the pairwise comparisons provided by users to pick the leader (or the top few leaders) of the contest thus far; the goal is to reflect the current aggregated opinion as faithfully as possible. For simplicity, we will ignore issues such as malicious user behavior and assume each user is able to

* Part of this work was done while the author was at Yahoo! Research.

** Supported in part by NSF award DMI-0600384, ONR Award N000140610607, and Yahoo! Faculty Research Grant.

compare any pair of candidates. In fact, we will assume something more general: each user has access to the global total order (“the public opinion”) and when Bix shows a pair of candidates, the user consults this total order to rank-order the given pair.

There are two factors that make this setting both interesting and challenging. First, as the contest progresses, users’ voting patterns might change, perhaps slowly, at an aggregate level. This can be caused by an intrinsic shift in public opinion about the candidates or factors external to the contest. While one cannot assume there is a fixed total order that the contest is trying to uncover, it is reasonable to assume that the total order changes slowly over time. Second, whenever a user visits their website, Bix has to choose a pair of candidates to show to the user in order to elicit the comparison. A visiting user is thus a valuable resource and hence Bix has to judiciously utilize this by showing a pair of candidates that yields the most value. Note that this is not a trivial problem: for example, it is not hard to show that asking the user to rank a random pair of candidates is quite “wasteful” and leads to considerably weaker guarantees.⁴

One way to model the above scenario is as follows. We have a set of n elements and an underlying total order π^t , at time t , on the elements. The ordering slowly changes over time and we model the slow change by requiring that the change from π^t and π^{t+1} is local. The goal is to design an algorithm that, at any point in time, tracks the top few elements of the underlying total order or more generally, maintains a total order $\tilde{\pi}^t$ that is close to π^t . The only capability available to the algorithm is pairwise comparison probes: at any time t , given one or more pair of elements, it can obtain the pairwise ranking of them according to the underlying total order currently in effect, (i.e., π^t). Clearly, there is a tradeoff between the number of probes that can be made at time t and the quality of $\tilde{\pi}^t$ (e.g., if the number of probes is large enough, then $\tilde{\pi}^t = \pi^t$ is easily achievable.)

Another motivation for the sorting problem is that of ranking in settings such as web search, recommendation systems, and online ad selection. A significant factor in ranking is the use of historic data. However, what may have been a good ranking in the past may not remain so perpetually, and the ranking changes are typically gradual over time (e.g., the query “vacation spots” might connote differently depending on the time of the year). The ranking system would like to track the changing perception of ranking by selecting what feedback (in the form of clicks) to request from the user. In addition to the above applications, which are mostly in the Internet domain, the problem has applications in sociology under the topic of the method of “paired comparisons” in the measurement of social values [5, Ch. 7].⁵

Of course, except for the aforementioned motivations for the sorting problem, similar issues arise in scenarios other than sorting. Consider, for example, a web crawler, whose goal is to track the highest quality pages on the web. The notion of quality, however, is (slowly) time-varying and the crawling algorithm, which is usually resource-constrained, has only limited access to the web graph at any point in time. The goal of the crawler would then be to track pages whose quality is reasonably close to the

⁴ In the language of the model defined in Section 2, this algorithm leads to a guarantee of $O(n^2)$ for the Kendall tau distance (only a constant factor better than an oblivious algorithm that always outputs the same ranking), whereas we are able to achieve $O(n \ln \ln n)$.

⁵ We thank Matthew Salganik for pointing out this application.

current best. Another graph application is maintaining routing tables with fastest (least congested) routes. The load on routes changes gradually, and the router receives new information on route's load only when a packet is sent along that route. Yet another setting can be that of a company that wants to track popular social network users with lots of friends, so as to use this information for viral marketing. Social networking systems such as Facebook allow to query and find the contacts of a given user (unless the user explicitly disallows) but limit the number of queries so as to prevent abuse. Overall, our setting is fairly general and can capture real-life scenarios such as continually updated remote databases, hashing, load balancing, polling, etc.

A general framework. The nature of the problems described above suggests the following general framework to study dynamic data. Let \mathcal{U} and \mathcal{V} be (possibly infinite) universe of objects. Let $f : \mathcal{U} \rightarrow \mathcal{V}$ be a function. Let $d : \mathcal{U} \times \mathcal{U} \rightarrow \mathbb{R}^+$ and $d' : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{R}^+$ be pairwise distance functions. $U^t \in \mathcal{U}$ is the object at time t while $V^t \in \mathcal{V}$ will be the estimate of the output of function f at time t .

(1) We have an implicit sequence of objects U^1, U^2, \dots such that $d(U^t, U^{t+1})$ is small, that is, the object changes slowly over time. The change can be arbitrary, or stochastic (which is the case that we consider in this paper).

(2) At each t , portions of the object U^t can be accessed by a certain number of probes.

(3) The goal is to output a sequence V^1, V^2, \dots such that for each t , $d'(f(U^t), V^t)$ is small, that is, we have a good approximation to the function of the true object at each point in time.

In the case of the Bix sorting problem that is the main focus of this paper, $\mathcal{U} = \mathcal{V} = S_n$, the set of permutations on n elements, $d = d'$ is the Kendall tau distance, and f is the identity function. For the selection problems we have that \mathcal{V} is the set of elements, d' is the absolute rank difference between two elements, and f is an element. The slow changing of the objects in (1) is captured by permitting, say, only pairwise exchanges (corresponding to Kendall distance of 1) and the access to the object in (2) is captured by rank-ordering a given pair of elements according to the current total order. Even though in this paper we only focus on ranking and selection problems, this framework applies to many other settings as well, such as graph algorithms [1].

Related work. Models for dealing with dynamic and uncertain data have been extensively studied in the algorithmic community, from various points of view. This includes the multi-arm bandit algorithms that deal with explore-exploit tradeoffs, online algorithms that deal with future information, dynamic graph algorithms that deal with fast updates in response to graph changes, data stream algorithms that deal with limited computational resources such as space, stochastic optimization algorithms, etc. However, none of these captures the two crucial aspects of the above scenario: the slow changing of the underlying object and the probe model of exposing only a limited portion of the object to an algorithm. We are only aware of one other work that studies similar tradeoffs, namely the work of Slivkins and Upfal [4], which studies them in the more restricted setting of the multi-armed bandit model.

Our results. For the problem of maintaining a sorted order using a single probe at each time step when the permutation changes slowly and randomly and where the notion of distance is Kendall tau (number of pairwise disagreements), we give an algorithm that guarantees that for every time step t , the distance between the underlying true ordering and the ordering maintained by the algorithm is at most $O(n \ln \ln n)$, in expectation and with high probability. This builds upon an algorithm that has a distance guarantee of $O(n \ln n)$, in expectation and with high probability. We also show an $\Omega(n)$ lower bound on the expected distance between the true ordering and the order maintained by any algorithm.

To show the upper bound result, we first develop an algorithm that is based on periodically running the quicksort algorithm on the data. We use quicksort-specific properties to show that this algorithm can guarantee a distance of $O(n \ln n)$. We then give a more sophisticated algorithm that runs a copy of the above quicksort-based algorithm in parallel with multiple copies of faster though less accurate “local quicksorts.” These local quicksorts will be able to give us the desired distance guarantee of $O(n \ln \ln n)$ in the first few runs; however, their weakness is that they could accumulate the errors and lead to considerably worse distance guarantees later. This weakness is overcome by occasionally resetting the algorithm using the slower quicksort, which is run in parallel.

We then consider selection problems: finding an element of a given rank. We provide algorithms that track the target elements to within distance 1. The basic idea is similar to the one we used for sorting: we adapt a static algorithm to the online setting by repeated executions. Furthermore, to ensure that the result returned is always close to the true value, we decompose the algorithm into two processes that are executed independently and in parallel, where the slower process prepares the data structures that the faster process uses over and over to compute the output. For the special case of finding the minimum element, we give a simpler algorithm by modeling the evolution of the process as a Markov chain.

2 Sorting Dynamic Elements

Consider a set $U = \{u_1, \dots, u_n\}$. Throughout most of this paper, our focus is on the problem of sorting the elements of U . In a static setting, where the correct ordering of the elements of U is given by a permutation π , there are numerous well-known sorting algorithms that can find the permutation π after comparing $O(n \ln n)$ pairs in U [2]. We are interested in a dynamic setting, where the true ordering π changes over time. To make this precise, consider a discretized time horizon with time steps indexed by positive integers. Let π^t be the true ordering at time t . We assume that the true ordering changes gradually, and we model this by assuming that for every $t > 1$, π^t is obtained from π^{t-1} by swapping a random pair of *consecutive* elements.

Our objective is to give an algorithm that can estimate the true ordering π^t . Unlike the familiar notion of algorithms that terminate in finite time, the algorithms we study run for ever; we often refer to them as *protocols*. In every time step t , the algorithm can select two elements of U to compare. The ordering of these two elements according to π^t is given to the algorithm, and then the algorithm computes an estimate $\hat{\pi}^t$ of the true ordering. The algorithm has memory, that is, it is allowed to store any information, and the information will be carried over to the next time step. Note that our definition

assumes that the rate of comparisons performed by the algorithm is equal to the rate of change in the true ordering (i.e., one swap and one comparison probe per time step). This assumption is merely for simplicity: all our results work in the more general setting where corresponding to every change in the true ordering, the algorithm is allowed to perform a number of comparisons given by a parameter β (β can be more or less than 1); the proofs are essentially exactly the same, with the bounds multiplied by functions of β . Furthermore, notice that we did not impose any constraint on either the amount of memory required by the algorithm or its running time. While such constraints seem natural in practice, it turns out that the running time and the memory are not major concerns, at least for the algorithms that we propose in this paper. Also, we need to specify whether the algorithm knows the initial ordering π^1 . For convenience, we assume that the algorithm knows π^1 , although our results hold without this assumption as well.⁶

Notice that unlike in the static setting where the algorithm can find the permutation π after finite time, in the dynamic setting the algorithm can never expect to find the exact true ordering π^t . Therefore, we need a way to measure how close the estimate is to the true ordering. For this purpose, we use the classical *Kendall tau* distance function between permutations. For a permutation π we write $x <_\pi y$ if x is ordered before y according to permutation π . The Kendall tau distance $\text{KT}(\pi_1, \pi_2)$ between permutations π_1 and π_2 is defined as follows:

$$\text{KT}(\pi_1, \pi_2) = |\{(x, y) : x <_{\pi_1} y \wedge y <_{\pi_2} x\}|.$$

The maximum Kendall tau distance between two permutations (and in fact the distance between two random permutations) is $O(n^2)$. In fact, no algorithm can guarantee that in every time step the distance between π^t and $\tilde{\pi}^t$ is less than $O(n)$ (Section 2.1). Our main result in Section 2.3 shows that there is an algorithm that can guarantee with high probability that this distance is at most $O(n \ln \ln n)$. We start with an easier result of $O(n \ln n)$ in Section 2.2, which will be used in our main result.

2.1 Lower Bound

We first prove an $\Omega(n)$ lower bound on the expected Kendall tau distance between the estimated order computed by any algorithm for our problem and the actual order at any time t .

Theorem 1. *For every $t > n/8$, $\text{KT}(\tilde{\pi}^t, \pi^t) = \Omega(n)$ in expectation and whp.⁷*

Proof. Consider the time interval $I = [t - n/8, t]$. Let B be the set of items involved in any comparison by the algorithm in this interval, $|B| \leq n/4$. Let $\bar{B} = U \setminus B$. At any time $\tau \in I$, the elements of B are adjacent in π^τ to up to $n/2$ elements in \bar{B} . Thus, for any $\tau \in [t - n/8, t]$, there are at least $n/4$ pairs of adjacent elements of \bar{B} in π^τ , each of these pairs is swapped with probability c/n , where $c > 0$ is a constant. Thus, during the interval $[t - n/8, t]$ the expected number of pairs in \bar{B} that are swapped is $(c/32)n$. The expected number of pairs that are swapped and then swapped back

⁶ We only need to be careful to require $t \geq n \ln n$ in our upper bounds (Theorems 2 and 3) if the algorithm does not know π^1 .

⁷ We say that an event holds “with high probability”, abbreviated whp., if it holds with probability at least $1 - n^{-c}$ for some constant c , for sufficiently large n .

is $O(c^2 n^2/n^2) = O(1)$. Since no element of \bar{B} is involved in any comparison the algorithm cannot identify swapped between pairs in \bar{B} , implying the result. \square

2.2 Algorithm with $O(n \ln n)$ Distance Guarantee

In this section we first give an algorithm that guarantees the following: for every time step t , the distance between the orderings π^t and $\tilde{\pi}^t$ is $O(n \ln n)$, with high probability. We will use this result in the next section to get an improved bound of $O(n \ln \ln n)$.

The algorithm proceeds in phases, where each phase consists of $O(n \ln n)$ time steps (in expectation and whp.). In each phase, the algorithm runs a randomized quicksort algorithm to sort all elements. At any time step, the algorithm outputs the ordering that is obtained at the end of the *last* phase. Notice that since this algorithm outputs the same permutation for $O(n \ln n)$ steps, it cannot provide a distance guarantee better than $O(n \ln n)$. The following theorem shows that the distance guarantee of this algorithm is in fact $\Theta(n \ln n)$ whp.

Theorem 2. *For every t , $\text{KT}(\tilde{\pi}^t, \pi^t) = O(n \ln n)$ in expectation and whp.*

Before proving the above theorem, we note that in our algorithm, the quicksort algorithm *may not* be replaced by an arbitrary $O(n \ln n)$ sorting algorithm. The reason being, in our setting, the algorithm can receive inconsistent data (since the true ordering is changing), and such inconsistencies can lead to large errors in general. In the case of quicksort, we will use its specific properties to argue that the inconsistencies can result in only a small number of *additional* errors (these errors will correspond to the set B in the following proof).

Proof (of Theorem 2). Consider one phase of the algorithm from time t_0 to t_1 . We have that $t_1 - t_0 = \Theta(n \ln n)$, in expectation and whp.

To bound the Kendall tau distance we have to bound the number of pairs (u_i, u_j) that are ordered differently in the two permutations $\tilde{\pi}^t$ and π^t . We divide these pairs into two disjoint sets, A and B , where the set A contain the pairs for which the algorithm's order at time t_1 is in accordance with the true ordering at some time point $t \in [t_0, t_1]$:

$$A = \{(u_i, u_j) \mid u_i <_{\tilde{\pi}^{t_1}} u_j, u_i >_{\pi^{t_1}} u_j, \exists t \in [t_0, t_1] \text{ s. t. } u_i <_{\pi^t} u_j\},$$

and the set B contains the pairs for which there was a disagreement between the algorithm's order estimate (at time t_1) and the true order throughout the execution of the algorithm in this phase:

$$B = \{(u_i, u_j) \mid u_i <_{\tilde{\pi}^{t_1}} u_j, \forall t \in [t_0, t_1] u_i >_{\pi^t} u_j\}.$$

Since $\text{KT}(\tilde{\pi}^t, \pi^t) = |A \cup B| = |A| + |B|$, Lemmas 1 and 2 will complete the proof. \square

First we bound the cardinality of A by the running time of the algorithm.

Lemma 1. $|A| = O(n \ln n)$ in expectation and whp.

Proof. For the set A , note that if we let A' be the set of pairs for which the true order changed in $[t_0, t_1]$, that is,

$$A' = \{(u_i, u_j) \mid u_i <_{\pi^{t_1}} u_j, \exists t \in [t_0, t_1] \text{ s. t. } u_i >_{\pi^t} u_j\},$$

then we have that $A \subseteq A'$. Now notice that since the true order of the pair (u_i, u_j) was swapped during $[t_0, t_1]$, it has to be the case that at some point in $[t_0, t_1]$, the pair

(u_i, u_j) was chosen to swap. Since only one pair swaps its ordering at each timestep and since $t_1 - t_0 = O(n \ln n)$ in expectation and whp., we have that $|A| \leq |A'| \leq t_1 - t_0 = O(n \ln n)$ in expectation and whp. \square

For the set B the counting is more involved. By definition, for a pair $(u_i, u_j) \in B$ we have that $u_i > u_j$ according to the true ordering during $(t_0, t_1]$, however, at t_1 the algorithm concluded otherwise. This means that during one of the recursive calls of the quicksort algorithm, elements u_i and u_j belonged to the same subarray that was then sorted, a pivot element u_k was chosen ($u_k \neq u_i, u_j$), and after element u_k was compared with all the elements of the subarray, the result was $u_i < u_k$ and $u_j > u_k$. For this to have happened, the element u_k would have to be swapped with *each* of the elements u_i and u_j at least once while it was a pivot. After the element u_k terminates being a pivot, the algorithm's perception of the ordering between u_i and u_j does not change. (Note that the above arguments crucially rely on the fact that the algorithm is quicksort.)

From the previous discussion we see that if we can bound the number of swaps of the pivot elements during the period they were acting as pivots, then we will be able to bound the number of pairs in the set B . Since the probability that a pivot element is chosen at a given time step is small (at most $2/n$), we expect the set B to be small. We prove this formally below.

Lemma 2. $|B| = O(n \ln n)$ in expectation and whp.

Proof. We will charge the error due to pair (u_i, u_j) to the corresponding pivot u_k . Let X_i be the number of steps that element u_i was a pivot during $[t_0, t_1]$; note that $X_i \leq n$. Let \mathcal{E} be the event that

$$\sum_{i=1}^n X_i \leq c_0 n \ln n, \quad (1)$$

for some constant $c_0 > 0$. Since the running time of quicksort is $O(n \ln n)$ in expectation and whp., \mathcal{E} holds whp. Also, the running time of quicksort is $O(n^2)$ in the worst case. The event $\neg\mathcal{E}$ will only contribute a negligible (inverse polynomial) amount to the calculations below; therefore, for ease of exposition, we will condition on \mathcal{E} being true for the rest of the proof.

Since $X_i \leq n$ and $\sum X_i \leq c_0 n \ln n$, by convexity, $\sum X_i^2$ is maximized if $c_0 \ln n$ of the X_i 's are equal to n and the rest are equal to 0. Hence,

$$\sum_{i=1}^n X_i^2 \leq c_0 n^2 \ln n. \quad (2)$$

Let Y_i be the number of steps that element i was a pivot element and it was chosen to swap. Given X_i , we have that $Y_i \sim \text{Binomial}(X_i, p)$ where $p = 2/n$ (with the exception of the case that the pivot is or becomes the first or last element in the order, in which case $p = 1/n$). We argued earlier that for the pair (u_i, u_j) to become misordered, the corresponding pivot was swapped with both u_i and u_j . Therefore, if a pivot swapped Y_i times, then it could have led to at most Y_i^2 misordered pairs. We can then bound the

number of pairs in the set B by $S \triangleq \sum_{i=1}^n Y_i^2 \geq |B|$. The proof is complete if we upper bound $\mathbf{E}[S]$. Now,

$$\begin{aligned} \mathbf{E}[S] &= \mathbf{E} \left[\sum_{i=1}^n Y_i^2 \right] = \mathbf{E} \left[\mathbf{E} \left[\sum_{i=1}^n Y_i^2 \mid X_i \right] \right] = \mathbf{E} \left[\sum_{i=1}^n \mathbf{E} [Y_i^2 \mid X_i] \right] \\ &= \mathbf{E} \left[\sum_{i=1}^n (\mathbf{Var} [Y_i \mid X_i] + \mathbf{E}[Y_i \mid X_i]^2) \right] = \mathbf{E} \left[\sum_{i=1}^n (X_i p(1-p) + X_i^2 p^2) \right] \\ &= \mathbf{E} \left[p(1-p) \sum_{i=1}^n X_i + p^2 \sum_{i=1}^n X_i^2 \right] \stackrel{(1),(2)}{\leq} (2c_0(1-p) + 4c_0) \ln n \leq c_1 \ln n, \end{aligned}$$

for some constant $c_1 > 0$.

To bound the probability that the set B is large, first note that given X_i 's, the Y_i 's are independent binomial random variables. We apply Azuma's inequality and finish the proof.⁸ For some sufficiently large constant $c_2 > 0$, we have

$$\Pr(S - \mathbf{E}[S] > c_2 n \ln n) \leq \exp \left(-\frac{2c_2^2 n^2 \ln^2 n}{\sum_{i=1}^n X_i^2} \right) \stackrel{(2)}{\leq} n^{-2c_2^2/c_1}.$$

The following lemma is also proved similarly and will be used later. The proof will appear in the full version of the paper.

Lemma 3. *Given an element u_i , the number of pairs (u_i, u_j) that the permutations π^{t_1} and $\tilde{\pi}^{t_1}$ rank differently is bounded by $\tilde{c} \ln n$ in expectation and whp., for some constant \tilde{c} and sufficiently large n .*

2.3 Main Result

Now we present a more complicated protocol that maintains an error of $O(n \ln \ln n)$. The main idea is that after the quicksort execution, which due to its running time has as a result an error of $O(n \ln n)$, the rank of each element in the algorithm's estimate is within $O(\ln n)$ of its actual rank. Thus, by performing several ($O(n/\ln n)$) local sorts in blocks of size $m = \Theta(\ln n)$ we can correct the ordering. The total running time is $O(n/\ln n) \cdot (\ln n) \ln \ln n$, therefore after all the sorts terminate the total error will be bounded by $O(n \ln \ln n)$.

There are some issues that we have to address though. First, since elements might have moved to neighboring blocks, we make the blocks overlapping thus allowing the comparison of all neighboring elements (see Figure 1). First we sort the first m elements. From the resulting order we maintain the first $m/2$ of the elements. The second half of the block is sorted along with the next $m/2$ elements. Again we maintain the first $m/2$ elements and proceed in the same way.

⁸ The following is a consequence of Azuma's inequality [3]. Assume that $0 < X_i < d_i$ are independent random variables, and let $S = \sum_{i=1}^n X_i$. Then

$$\Pr(S - \mathbf{E}[S] > \lambda) \leq \exp \left(-2\lambda^2 / \sum_{i=1}^n d_i^2 \right).$$

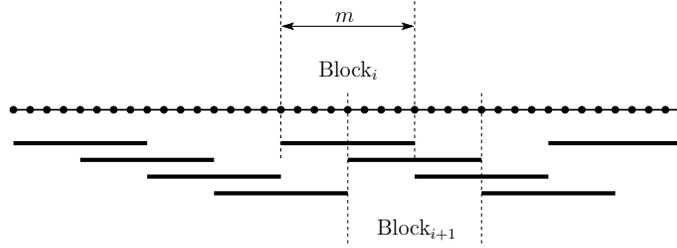


Fig. 1. The set of elements ordered according to $\tilde{\pi}^{t_0}$, and the partition into blocks.

Second, while we would like to sequentially execute a full set of local quicksorts after the termination of the previous one so as to maintain the error of $O(n \ln \ln n)$, eventually elements will move far. Thus it is necessary to occasionally execute a full quicksort to recover the global order. The problem, however, is that during the execution of the global quicksort the error will become $n \ln n$. Therefore, we use the following idea: execute two sets of quicksorts independently. During the odd timesteps we execute a regular quicksort, and after its termination we restart, as in Section 2.2. The previous analysis applies to this case as well with the difference that in every step there are two pairs whose order swaps. During the even steps, we execute the set of $\Theta(n / \ln n)$ quicksorts on overlapping blocks of length $m = \Theta(\ln n)$. The input to the set of quicksorts is the output of the last full quicksort that has terminated. After the termination of the set of quicksorts we rerun them, again with the same input. The two processes are executed independently with their own data structures. In every time step, the “output” of the protocol is the output of the latest successfully completed set of quicksorts. In Figure 2 we present a schematic representation of the algorithm. The proof of the following theorem will appear in the full version of the paper.

Theorem 3. For every t , $\text{KT}(\tilde{\pi}^t, \pi^t) = O(n \ln \ln n)$ in expectation and whp.

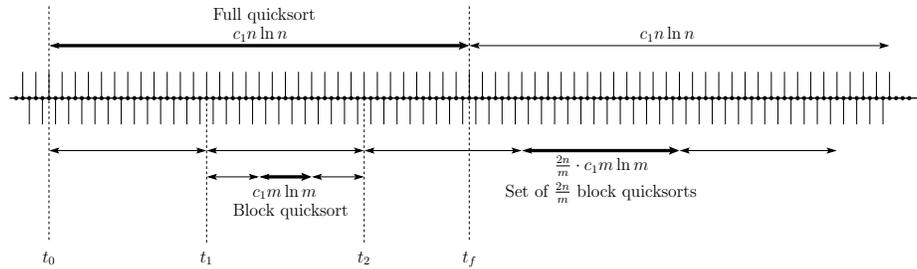


Fig. 2. The periods of the execution of the sorting algorithm.

3 Selection Problems

As we mentioned earlier, the dynamic data setting can capture several scenarios. In this section we illustrate this by providing two more examples. First we show a simple algorithm for finding the element with minimum rank; for a fairly realistic application of this setting, consider the social-network example presented in the introduction. We then present a more general algorithm that can be used to find the element of a given rank. By combining this algorithm with the previous result on sorting, one can find the top- k ranked elements.

3.1 Finding the Minimum

Assume the same dynamic perturbation model as before where each pair swaps in every time step with probability $\alpha/(n-1)$, where $\alpha > 0$ is a constant ($\alpha = 1$ in the simplest case). Instead of sorting all the elements we only want to estimate the smallest element. The following simple algorithm outputs at any given step an element that is either the minimum or very close to minimum. The algorithm maintains the current minimum estimate m and in every step compares it with an element u_i chosen uniformly at random from all the elements. If $u_i < m$, it replaces m with u_i . The basic idea to prove the following theorem is to model the process as a Markov chain (details omitted in this version).

Theorem 4. *Let m_t be the rank of the estimate at time t . In the steady state $\Pr(m_t \geq i) \leq \left(\frac{\alpha}{1+\alpha}\right)^i$, and $\mathbf{E}[m_t] = 1 + \alpha$.*

3.2 Finding the Element of a Given Rank

In this section we give an algorithm for solving the problem of finding the element of rank k for $k = 1, 2, \dots, n$. Given k , our goal is to find an element u_i that minimizes the distance $|\pi^t(u_i) - k|$, where $\pi^t(u_i)$ is the rank of u_i at time t . For $k = 1$ the problem is that of finding the minimum, while for $k = \lceil n/2 \rceil$ the problem is that of finding the median. To make the exposition clearer we present the case of the median; the algorithm and the proof can be easily generalized for any k . Figure 3 is a dynamic version of the median algorithm in [3], with a few modifications to adapt it to our dynamic setting. As in the case of the elaborate sorting algorithm, we run two algorithms in an interleaved manner. In the odd steps we prepare a set C that will contain the median in any step of the next execution of the while loop. In the even steps we use the set C computed in the previous step to output the median estimate. During a sorting phase of the set C , the output estimate is the element $\tilde{\mu}$ computed in the previous run of the sorting. Let t_0 be the last time point of the first period. We show that the difference in the rank of the element returned by the algorithm is negligible.

Theorem 5. *Let $\tilde{\mu}$ be the output of algorithm Median. For every time $t \geq t_0 = O(n)$ we have that $\Pr(|\pi(\tilde{\mu}) - \frac{n}{2}| = 0) \geq 1 - o(1)$, and $\mathbf{E}[|\pi(\tilde{\mu}) - \frac{n}{2}|] = o(1)$.*

Proof. The proof is based on the proof of the static version presented, for example, in [3]; we will outline only the modification specific to the dynamic case.

1. **Algorithm** Median(U)
2. **Input:** A set of elements U
3. **while (true)**
4. **Execute in odd steps:**
5. Pick a (multi-)set R of $\frac{n}{36 \ln n}$ elements from U chosen independently uniformly at random with replacement
6. quicksort(R)
7. Let d be the $(\frac{n}{72 \ln n} - \sqrt{n})$ th smallest element in the sorted set R
8. Let u be the $(\frac{n}{72 \ln n} + \sqrt{n})$ th smallest element in the sorted set R
9. By comparing every element in U to d and u , compute the set $C = \{x \in U : d \leq x \leq u\}$ and the number $\ell_d = |\{x \in U : x < d\}|$
10. **Execute in even steps using the set C computed last**
11. quicksort(C)
12. $\tilde{\mu} \leftarrow (\lfloor n/2 \rfloor - \ell_d + 1)$ th element in the sorted order of C
13. **end while**

Fig. 3. Algorithm for computing the median.

We partition time into periods of length $\Theta(n)$, where each period corresponds to a full execution of steps 4–9 in Figure 3. (Executing the full set of steps 4–9 (odd time steps) requires time $\Theta(n)$ while the set of steps 10–12 (even time steps) requires $\Theta(\sqrt{n} \ln^2 n)$, whp.) In the odd time steps of a period we compute a set C to be used to compute the median in the next period. In the even time steps we use the set C computed in the previous period.

We first note that the length of each period is linear with high probability, therefore in a given period the rank of a given element (and in particular that of the median) changes by a constant in expectation. Furthermore, with high probability no element moves more than $c \ln n$ places during a period, for some constant c .

The analysis of the static case as presented in [3] reduces to proving that the following two facts (adapted to our case) hold whp.:

1. The set C computed at a given period contains all the elements that are medians during the next period.
2. $|C| = O(\sqrt{n} \ln n)$ whp.

With a similar argument as the one used in [3] and taking into account that the rank of element d during two periods does not change more than $2c \ln n$ whp., we have that in order to maintain $\pi^t(d) < \lceil n/2 \rceil$ it suffices that at most $\frac{n}{72 \ln n} - \sqrt{n}$ samples in R had rank smaller than $\frac{n}{2} - 2c \ln n$, when they were selected. We define $X_i = 1$ if the i th sample had rank smaller than $\frac{n}{2} - 2c \ln n$, and 0 otherwise. Then we have that $\Pr(X_i = 1) = \frac{1}{2} - \frac{2c \ln n}{n}$ and $\mathbf{E}[\sum X_i] = \frac{n}{72 \ln n} - \frac{c}{18}$. We can apply a Chernoff bound and obtain:

$$\begin{aligned} \Pr\left(\sum_{i=1}^{|R|} X_i < \frac{n}{72 \ln n} - \sqrt{n}\right) &= \Pr\left(\sum X_i - \mathbf{E}\left[\sum X_i\right] < \frac{c}{18} - \sqrt{n}\right) \\ &\leq e^{-\frac{72 \ln n}{n}(\sqrt{n} - \frac{c}{18})^2} \leq \frac{1}{n^3}. \end{aligned}$$

A similar argument shows that we maintain that $\pi^t(u) > \lceil n/2 \rceil$ throughout the execution of the entire period, therefore, the set C created at step 8 will contain whp. all elements that are medians in the next period.

Next we need to show that $|C| = O(\sqrt{n} \ln n)$. The argument must take into account that the elements change ranks and is similar to the one just presented. The reader can refer to [3] for the type of events that need to be defined and we do not repeat here the details. Thus C can be sorted in $O(\sqrt{n} \ln^2 n)$ time.

We have now established that whp. $|C| = O(\sqrt{n} \ln n)$ and that it contains all elements that are medians during the next period. Since sorting in step 8 takes $O(\sqrt{n} \ln^2 n)$ steps, the probability that either the median at the beginning of a sorting phase, or the $O(\ln n)$ pivots that it is compared to during the sorting move during the sorting phase is bounded by $O(\ln^3 n / \sqrt{n})$. Thus, with probability $1 - O(\ln^3 n / \sqrt{n})$ the sorting returns the correct median at that step. The probability that the median change place during the next sorting round (before a new median is computed) is bounded by $O((\sqrt{n} \ln^2 n) / n)$. Thus, at any given step, with probability $1 - o(n^{-1/2+\epsilon})$ the algorithm returns the correct median. The expectation result is obtained by observing that when the output is not the correct median, its distance to the correct median is with high probability $O(\ln n)$. \square

4 Conclusions

In this paper we study a new computational paradigm for dynamically changing data. This paradigm is rich enough to capture many natural problems that arise in online voting, crawling, social networks, etc. In this model the data gradually changes over time and the goal of an algorithm is to compute some property of it by probing, under the constraint that the amount of access to the data at each time step is limited. In this simple framework, we consider the fundamental problems of sorting and selection, where the true ordering slowly changes over time and the algorithm can probe the true ordering once each time step using a pair of elements it chooses. We obtain an algorithm that maintains, at each time step, an ordering that is at most $O(n \ln \ln n)$ -Kendall-tau distance away from the true ordering, with high probability. For selection problems, we provide algorithms that track the target element to within distance 1. Revisiting classical algorithmic problems in this paradigm will be an interesting direction for future line of research [1].

References

1. A. Anagnostopoulos, R. Kumar, M. Mahdian, and E. Upfal. Dynamic graph algorithms. Manuscript, 2009.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.
3. M. Mitzenmacher and E. Upfal. *Probability and Computing*. Cambridge University Press, 2005.
4. A. Slivkins and E. Upfal. Adapting to a changing environment: The Brownian restless bandits. In *Proc. 21st Annual Conference on Learning Theory*, pages 343–354, 2008.
5. L. L. Thurstone. *The Measurement of Values*. The University of Chicago Press, 1959.