Sorting and Selection on Dynamic Data[*]

Aris Anagnostopoulos[†] Ravi Kumar[‡] Mohammad Mahdian[‡] Eli Upfal[§]

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 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 receives all the input at the start of the computation and computes 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 online voting website Bix (`bix.com`), owned by Yahoo!;[1] 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

[*]The research leading to these results has received funding from the EU FP7 Project N. 255403 – SNAPS and by the NSF award IIS-0905553.

[†]Department of Informatics and System Sciences, Sapienza University of Rome, Italy, `aris@dis.uniroma1.it`

[‡]Yahoo! Research, Sunnyvale, CA, {`ravikumar,mahdian`}`@yahoo-inc.com`

[§]Dept. Computer Science, Brown University, Providence RI, `eli@cs.brown.edu`

[1]Yahoo! acquired the Bix contest site in the end of 2006 and kept it operative until June 2009 when it decided to terminate it as part of its prioritization efforts.

issues such as malicious user behavior and assume that each user is able to 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 the 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.[2]

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 [13, Ch. 7].[3]

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 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

[2] 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 $O(n \ln \ln n)$.

[3] We thank Matthew Salganik for pointing out this application.

use this information for viral marketing. Social networking systems such as Facebook and LinkedIn allow one 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.

1.1 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) universes of objects. Let $f : \mathcal{U} \to \mathcal{V}$ be a function. Let $d : \mathcal{U} \times \mathcal{U} \to \mathbb{R}^+$ and $d' : \mathcal{V} \times \mathcal{V} \to \mathbb{R}^+$ 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 $U^1, U^2, \ldots$ 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 considered in this paper).

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

(3) The goal is to output a sequence $V^1, V^2, \ldots$ 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, which 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 the element at a particular rank. The slow changing of the objects in (1) is captured by permitting, say, only pairwise swaps (corresponding to a 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].

1.2 Related Work

Models for dealing with dynamic and uncertain data have been extensively studied in the algorithmic community, from various points of view. 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 now discuss some of the models most related to ours.

Dynamic graph algorithms [5]. In the setting of dynamic graph algorithms a graph changes over time and the goal is to keep track of the changes so as to be able to efficiently answer graph queries. The main difference is that here when a change is performed to a graph it is known to the algorithm; instead in our setting the algorithm does not know the change but it has to perform queries to learn it. In addition, the expensive resource in our setting is the number of queries, while we do not pose any additional restrictions on the time/space complexity of the algorithms.

Multi-armed bandit algorithms [6]. This is a setting for studying exploration-exploitation trade-offs. In the standard multi-armed bandit setting there exist k slot machines (one-armed bandits). Pulling a lever in a slot machine gives a reward, which depends on the machine, and reveals information about the machine. The objective is to select which machines to query so as to maximize the total reward. Similar to our case, the number of queries in every time step is limited. In particular, similar tradeoffs with those in our setting are studied in the work of Slivkins and Upfal [11] where the distribution of the rewards changes over time. The setting studied in the current paper is more general as the underlying structure can be arbitrary.

Data stream algorithms [8]. Here the algorithm observes a sequence of events (for example, edge addition or deletion) and has to maintain an approximate solution. Here there are some limited computational resources, typically space, and the algorithm should maintain an approximate solution under the resource's limitations, while being able to observe the entire stream of changes. Instead our limited resource is the number of queries to the data.

Property testing [9, 10]. Here the goal is to find whether some structure (e.g., a graph) contains some property or is far from containing the property using a limited number of queries. As opposed to our setting, the model is static. Furthermore, in the dynamic-data model the underlying problem does not have to be a YES/NO question and there are no restrictions placed on the input (such that if the structure does not satisfy the property then it is far from satisfying it).

Online algorithms [2]. In online algorithms the input is revealed over time and the algorithm must make decisions without the knowledge of future requests. Again, the main difference is that all the underlying changes are revealed to the algorithm, as opposed to our setting where changes have to be discovered by performing the appropriate queries.

Stochastic optimization algorithms [12]. In stochastic optimization, decisions should be made anticipating future events, which take place according to distributional assumptions. Yet again, the algorithm is able to learn the changes as they take place as opposed to the dynamic-data setting.

1.3 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 the Kendall tau (number of pairwise disagreements), first we show an $\Omega(n)$ lower bound on the expected distance between the true ordering and the order maintained by any algorithm. Our conjecture is that this lower bound is tight. Subsequently, 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.

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 properties specific to quicksort 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.

1.4 Roadmap

In Section 2 we present the results for sorting. We present the precise model and in Section 2.1 we give the lower bound of $\Omega(n)$. In Section 2.2 we give the simple algorithm guaranteeing an error of $O(n \ln n)$ and in Section 2.3 we present our main result, which gives an error of $O(n \ln \ln n)$. In Section 3 we study selection problems. In Section 3.1 we give the simple algorithm for finding the minimum element and in Section 3.2 we present the algorithms for finding the median (or any element of a given rank). Finally, we conclude in Section 4.

2 Sorting Dynamic Elements

Consider a set $U = \{u_1, \ldots, u_n\}$. 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 [3]. 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 constant number $\alpha \geq 1$ of random pairs 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 $\tilde{\pi}^t$ of the true ordering. The algorithm has a memory, that is, it is allowed to store any information, and the information will be carried over to the next time step. Note 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.[4]

Notice that unlike in the static setting where the algorithm can find the permutation π after a 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

[4]We only need to be careful to require $t \geq cn \ln n$, for a constant c, in our upper bounds (Theorems 2 and 7) if the algorithm does not know π^1.

permutation π we write $x <_\pi y$ if x is ordered before y according to permutation π. The Kendall tau distance $\mathrm{KT}(\pi_1, \pi_2)$ between permutations π_1 and π_2 is defined as follows:

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

The maximum Kendall tau distance between two permutations (and furthermore, the distance between two random permutations) is $\Theta(n^2)$. 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 $t > n/8$, $\mathrm{KT}(\tilde{\pi}^t, \pi^t) = \Omega(n)$ in expectation and whp.*[5]

Proof. We prove the result for $\alpha = 1$, then it clearly holds for higher rates of change (the proof has to be modified only slightly, to prove it rigorously). The intuition of the proof is as follows. Consider the time interval $I = [t - n/100, t]$. The algorithm compares pairs involving at most $n/50$ elements in this time interval. Therefore, every time the nature[6] swaps a pair of consecutive elements of the permutation π, there is a constant probability that this pair does not involve any of the elements touched by the algorithm. This means that there is a linear number of swaps that the algorithm does not "know" about.

To formalize this idea, we use the principle of deferred decisions. First, we change the process as follows: in every step, after the algorithm selects a pair and asks for their comparison, the nature first picks *two* disjoint pairs of consecutive elements in π (i.e., $\pi(i), \pi(i+1)$ and $\pi(j), \pi(j+1)$ for i and j such that $\{i, i+1\} \cap \{j, j+1\} = \emptyset$) uniformly at random, and then selects one of these two pairs at random and swaps them. We call this random experiment process B. Clearly, the outcome of process B is exactly the same as the outcome of the original process, since choosing two pairs and then randomly choosing one of them is equivalent to choosing just one pair at random. The choice of the two pairs and the selection of one of them is called the nature's *decision* in this step.

Next, we change the process again by deferring some of nature's decisions. The idea is to fix all the decisions that involve at least one of the elements touched by the algorithm and defer the rest. However, since each swap by the nature will affect which pairs are candidates for swaps in the future, we also need to fix the decisions involving overlapping pairs picked by the nature. So, the process described below maintains the invariant that the deferred decisions at any time constitute a *random* set of *disjoint* pairs from among the elements that are not involved in any of the algorithm's comparisons or nature's fixed decisions so far.

The process, which will be denoted by C, is as follows: initially, the set of *touched* elements is empty, and the set of *deferred decisions* is also empty. In every time step during the interval I, the

[5]We say that an event holds *with high probability*, abbreviated whp., if it holds with probability that tends to 0 as $n \to \infty$.

[6]We use the term "nature" to refer to the agent or the mechanism performing the random changes in the underlying ranking.

algorithm selects a pair of elements to compare. If any of these elements is previously untouched, we mark it as touched and for any of the deferred decisions, we flip a coin with the appropriate probability to determine if the decision involves that element. If it does, we fix the decision according to the appropriate conditional distribution, i.e., we pick both pairs involved in the decision, and the one among them that should be swapped. These pairs can involve other previously untouched elements. We mark all these elements as touched, and again, flip coins to determine if any of the deferred decisions should involve any of those elements. This process is continued until for all the deferred decisions and all the elements newly marked as touched, we determine that the decision does not involve the element. After this, the query asked by the algorithm is answered (note that this can be done since the deferred decisions are irrelevant to the query asked by the algorithm), and then the nature needs to make a new decision (i.e., pick two more pairs). Again, we flip a coin to determine if any of these two pairs overlaps with the the set of touched elements, and if the outcome of the coin flip determines that it does, we fix this decision, update the set of touched elements to include the elements involved in the newly fixed decision, and iterate as before. Also, we flip a coin to determine if the new decision overlaps with any of the deferred decisions, and if it does, we fix both of the decisions by picking the corresponding pairs from the appropriate distribution and selecting one of the two pairs in each decision for the swap.

At the end of the interval I (i.e., at time t), this process has fixed some of the nature's decisions and has deferred the rest. At this point, we start a second phase and fix all the deferred decisions by picking a random set of disjoint pairs (two pairs for each deferred decision) and swapping a random one of the two pairs for each decision. Clearly, at the end of this phase, the distribution of the state of the process is exactly the same as process B.

The last step of the proof is to show that the number of deferred decisions at the end of the first phase of process C is $\Theta(n)$ in expectation. This is done as follows: a decision in process C will be deferred until the end of phase 1, if and only if the corresponding decision in process B consists of two pairs that are disjoint from all the other pairs picked by the nature and all queries asked by the algorithm. The set of all such elements is of size at most $6n/100$, since in each round the algorithm is picking 2 elements and the nature is picking 4. Let us now compute the probability that a decision is defered. A decision in process B consist of two pairs. For each of these pairs, we need to randomly pick the first element of the pair among any of the $n-1$ possible choices. Out of these $n-1$ pairs, fewer than $12n/100$ would lead to a pair that includes a "touched" element (since there are $6n/100$ touched elements, and we cannot pick the element before any of the touched elements either). So, at least a $(1-12/100)$ fraction of the choices would lead to a pair that does not include a touched element. This gives an overall probability of at least $(1-12/100)^2 > 1/2$ that a given decision is deferred. Thus, in expectation, at least half of the decisions are deferred until the end of phase 1. However, the algorithm has to output a permutation at the end of phase 1 before the deferred decisions are fixed. Taking the probability only over the random choices in the second phase, the expected distance between the permutations $\tilde{\pi}^t$ and π^t is at least half the number of deferred decisions. Thus, the overall expected distance between $\tilde{\pi}^t$ and π^t is at least $n/400 = \Omega(n)$. □

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

In this section, we give an algorithm that guarantees the following: for every time step t, the distance between the orderings π^t and $\tilde{\pi}^t$ is $O(n \ln n)$, whp. 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 next theorem shows that the distance guarantee of this algorithm is in fact $O(n \ln n)$ whp.

Theorem 2. *For every t, $\mathrm{KT}(\tilde{\pi}^t, \pi^t) = O(n \ln n)$ 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 that 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).

We also need to clarify what we mean by a randomized quicksort algorithm. The randomized quicksort algorithm picks a random element as the *pivot*, compares all other elements against this pivot and divides them into two sets S_1 and S_2, where S_1 are those elements that are less than the pivot and S_2 are those that are greater than the pivot. Then, it recursively sorts the set S_1, and after the completion of this part, it recursively sorts S_2. This is the natural way to implement the classical randomized quicksort algorithm [3], but while in the classical framework of sorting, it is permissible if the recursive run of the algorithm on S_1 is interleaved with the recursive run on S_2, in our setting it is not.

Before proving Theorem 2, we give the following proposition according to which, randomized quicksort is executed in time $O(n \ln n)$ in expectation and whp. under the dynamic-data model.

Proposition 3. *The running time of the standard randomized quicksort algorithm in the dynamic-data model is $O(n \ln n)$ in expectation and whp.*

Proof. The standard proof for the runtime of quicksort consists of the following main steps [4, Section 2.4]:

1. Call a pivot element *good* if it separates the corresponding array to two parts such that each of them has at least a constant fraction γ of the elements. This implies that for a given path in the quicksort execution tree the total number of good pivot elements is $O(\ln n)$.

2. Since a pivot element has a constant probability to be good, the number of *bad* (i.e., not good) pivots in a given path follows a negative binomial distribution and using a Chernoff bound we obtain that this number of bad pivots is a constant times the number of good pivot elements whp. This means that the total length of each path is $O(\ln n)$ whp.

3. Since there are at most n different search tree paths, a union bound shows that the running time is $O(n \ln n)$ whp. (and in expectation).

In the case of dynamic data, the proof is almost the same. For step 1, we define a pivot element to be good if at the moment it is chosen it splits (according to the true permutation) the corresponding array into two parts each containing at least a fraction γ of the elements. Since elements might swap when the pivot is partitioning the array, a good pivot might split the array into two parts such that one might have fewer than a γ fraction of the total elements of the array.

However, whp. each part will contain at least a fraction $\gamma/2$ of the elements, thus the number of good pivots in a given path is $O(\ln n)$ whp. The second and the third steps continue holding true unmodified. □

We can now prove Theorem 2.

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 $\tilde{\pi}^{t_1}$ and π^{t_1}. We divide these pairs into two disjoint sets, A and B, where the set A contains 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 $\mathrm{KT}(\tilde{\pi}^t, \pi^t) = |A \cup B| = |A| + |B|$, Lemmas 4 and 5 will complete the proof. □

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

Lemma 4. *$|A| = O(n \ln n)$ 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 $[t_0, t_1)$, i.e.,

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

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 α pairs swap their 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. □

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 $u_i < u_k < u_j$. 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 for a swap 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 5. *$|B| = O(n \ln n)$ 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_k be the number of steps that element u_k was a pivot during $[t_0, t_1)$; note that $X_k \le n$. Let $\mathcal{E}$ be the event that

$$\sum_{k=1}^{n} X_k \le c_0 n \ln n, \tag{1}$$

for some constant $c_0 > 0$. Note that since the randomized quicksort algorithm has exactly one pivot at any time step, $\sum_{k=1}^{n} X_k$ is the running time of the algorithm. Since, by Proposition 3, the running time of quicksort is $O(n \ln n)$ in expectation and whp., $\mathcal{E}$ holds whp., if we let c_0 be sufficiently large. 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 $X_k \le n$ and $\sum X_k \le c_0 n \ln n$, by convexity, $\sum X_k^2$ is maximized if $c_0 \ln n$ of the X_k's are equal to n and the rest are equal to 0. Hence,

$$\sum_{k=1}^{n} X_k^2 \le c_0 n^2 \ln n. \tag{2}$$

Let Y_k be the number of steps that element u_k was a pivot element and it was chosen to swap. Given X_k, we have that $Y_k \sim \text{Binomial}(X_k, p)$ where $p = 2\alpha/n$ (with the exception of the case that the pivot is or becomes the first or last element in the order, in which case $p = \alpha/n$). 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_k times, then it could have led to at most $\binom{Y_k}{2} \le Y_k^2$ misordered pairs. We can then bound the number of pairs in the set B by $S \triangleq \sum_{k=1}^{n} Y_k^2 \ge |B|$. The proof is complete if we upper bound $\mathbf{E}[S]$. Now,

$$\begin{aligned}
\mathbf{E}[S] &= \mathbf{E}\left[\sum_{k=1}^{n} Y_k^2\right] = \mathbf{E}\left[\mathbf{E}\left[\sum_{k=1}^{n} Y_k^2 \middle| X_k\right]\right] = \mathbf{E}\left[\sum_{k=1}^{n} \mathbf{E}\left[Y_k^2 | X_k\right]\right] \\
&= \mathbf{E}\left[\sum_{k=1}^{n} (\mathbf{Var}\left[Y_k | X_k\right] + \mathbf{E}[Y_k | X_k]^2)\right] = \mathbf{E}\left[\sum_{k=1}^{n} (X_k p(1-p) + X_k^2 p^2)\right] \\
&= \mathbf{E}\left[p(1-p)\sum_{k=1}^{n} X_k + p^2 \sum_{k=1}^{n} X_k^2\right] \overset{(1),(2)}{\le} (2\alpha c_0(1-p) + 4\alpha^2 c_0)\ln n \le c_1 \ln n,
\end{aligned}$$

for some constant $c_1 > 0$.

To bound the probability that the set B is large, first note that given X_k's, the Y_k's are independent binomial random variables. We apply Azuma's inequality and finish the proof.[7] For

[7]The following is a consequence of Azuma's inequality [7]. Assume that $0 < X_i < d_i$ are independent random variables, and let $S = \sum_{i=1}^{n} X_i$. Then

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

some sufficiently large constant $c_2 > 0$, we have

$$\mathbf{Pr}(S - \mathbf{E}[S] > c_2 n \ln n) \leq \exp\left(-\frac{2c_2^2 n^2 \ln^2 n}{\sum_{k=1}^{n} X_k^2}\right) \overset{(2)}{\leq} n^{-2c_2^2/c_0}.$$

□

The following lemma is also proved similarly and will be used later.

Lemma 6. *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 $c_3 \ln n$ in expectation and whp., for some constant c_3 and sufficiently large n.*

Proof. Similarly to the previous proof, we partition the set of incorrectly ordered pairs to two sets, A and B, A containing elements that are incorrectly ordered with u_i but at some point during the period $[t_0, t_1)$ were correctly ordered, and B elements that have been ordered incorrectly with u_i throughout the entire period $[t_0, t_1]$:

$$\begin{aligned} A = \{&u_j : u_i <_{\tilde{\pi}^{t_1}} u_j, u_i >_{\pi^{t_1}} u_j, \exists t \in [t_0, t_1) : u_i <_{\pi^t} u_j\} \\ &\bigcup \{u_j : u_j <_{\tilde{\pi}^{t_1}} u_i, u_j >_{\pi^{t_1}} u_i, \exists t \in [t_0, t_1) : u_j <_{\pi^t} u_i\}, \end{aligned}$$

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

Note that the size of set A can be bounded by the number of times element u_i was chosen to switch. Since at every timestep it is chosen with probability at most $2\alpha/n$ and since the total running time is bounded by $c_0 n \ln n$, the expected number of times that element u_i is chosen to switch is bounded by $2\alpha c_0 \ln n$, therefore, by applying the Chernoff bound, we obtain that the size of set A is bounded by $4\alpha c_0 \ln n$ in expectation and whp.

The size of set B can also be bounded by a similar way as before. In order for element u_i and element u_j to be ranked incorrectly even though they had always the same relative rank in the true ordering (say $u_i < u_j$), it must be the case that some pivot element, u_k, at some point was $u_k < u_i$ and while it was a pivot it became $u_k > u_j > u_i$ (or vice versa). The difference from the situation before is that we need to count only the pivots along the path (in the quicksort execution tree) of element u_i. If a pivot switched Y_k times, then it has added at most Y_k elements to the set B.

As before, we assume that the event $\mathcal{E}$ holds. We let X_k be the number of steps that element u_k was a pivot element and Y_k be the number of steps that element u_k was a pivot and switched order with another adjacent element. Let $P_i \subseteq U$ be the set of elements that acted as pivots along the path in the quicksort tree of element u_i. Then, by properties of the quicksort algorithm, we have

$$\mathbf{E}\left[\sum_{k: u_k \in P_i} X_k\right] = O(n), \tag{3}$$

and

$$\sum_{k: u_k \in P_i} X_k \leq c_4 n \ln n, \tag{4}$$

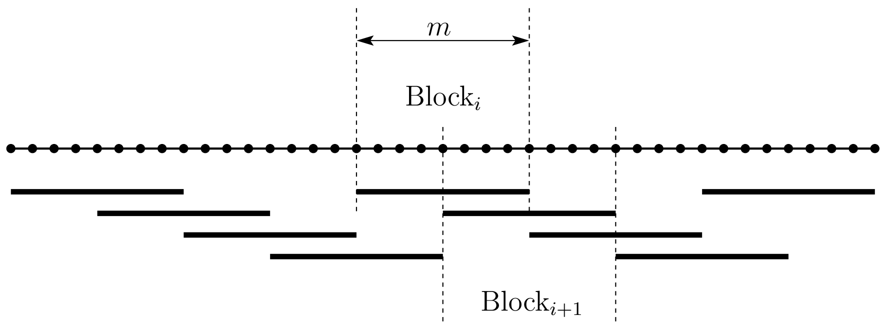

Figure 1: The partition into blocks.

whp., for a constant c_4. As before, since $X_k, Y_k \sim \text{Binomial}(X_k, 2\alpha/n)$ (again with the exception of the first and last elements where the probability is α/n), we have

$$\mathbf{E}[|B|] \le \mathbf{E}\left[\mathbf{E}\left[\sum_{k:u_k\in P_i} Y_k \middle| X_k\right]\right] \le \mathbf{E}\left[\sum_{k:u_k\in P_i} X_k \frac{2\alpha}{n}\right] = O(1),$$

by (3), and $\mathbf{E}[|B|] = O(\ln n)$, whp., by (4). Since $|B|$ is bounded above by $\text{Binomial}(\sum_{k:u_k\in P_i} X_k, 2\alpha/n)$, by the Chernoff bound, $|B| = O(\ln n)$, whp. Therefore the size of $A \cup B$, is $O(\ln n)$, whp. and the proof is complete for $c_3 = 2c_0 + c_4$. □

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 to exploit the fact that after the quicksort execution, which due to its running time has an error of $\Omega(n \ln n)$, the rank of each element in the algorithm's estimate is within $O(\ln n)$ of its actual rank. Therefore, for sorting such an ordering, it is not necessary to run an $O(n \ln n)$ algorithm from scratch; instead, we can use sorting algorithms that are faster than $O(n \ln n)$. In particular, by performing several ($O(n/\ln n)$) local quicksorts on blocks of size $m = \Theta(\ln n)$ we can correct the ordering. The total running time of this sorting algorithm is $O(n/\ln n) \cdot (\ln n) \ln \ln n$, therefore after this step terminates, the total error will be $O(n \ln \ln n)$.

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.

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 2α 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

```
1. **Function** BlockSort($\rho$)
2.   **Input:** Permutation $\rho =$ Output of the full quicksort at time $t_0$
3.   $B$ : Array of size $m$
4.   **for** $j = 1$ to $\frac{m}{2}$
5.     $B(j) \leftarrow \rho^{-1}(j)$
6.   **end for**
7.   **for** $i = 1$ to $\frac{2n}{m} - 1$
8.     **for** $j = 1$ to $\frac{m}{2}$
9.       $B\left(\frac{m}{2} + j\right) \leftarrow \rho^{-1}\left(i\frac{m}{2} + j\right)$
10.    **end for**
11.    quicksort($B$)
12.    **for** $j = 1$ to $\frac{m}{2}$
13.      $\sigma^{-1}\left((i-1)\frac{m}{2} + j\right) \leftarrow B(j)$
14.      $B(j) \leftarrow B\left(\frac{m}{2} + j\right)$
15.    **end for**
16.  **end for**
17.  **for** $j = 1$ to $\frac{m}{2}$
18.    $\sigma^{-1}\left(n - \frac{m}{2} + j\right) \leftarrow B(j)$
19.  **end for**
```

Figure 2: A set of block sorts that lasts for $\Theta(n \ln \ln n)$ steps. The algorithm is executed only during even time steps. The input permutation ρ is the latest output from a full (global) quicksort (the output of the full quick soft at time t_0 in Figure 3 if the BlockSort algorithm starts execution at time t_1). In every step the output of the algorithm is the output of the last BlockSort that is completed.

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. We present the even steps of the algorithm Figure 2, while in Figure 3 we present a schematic representation.

The next theorem proves the main result of the paper.

Theorem 7. *For every t, $\mathrm{KT}(\tilde{\pi}^t, \pi^t) = O(n \ln \ln n)$ in expectation and whp.*

Proof. We consider a period of execution of the algorithm $[t_0, t_f]$, which is of length at most $c_5 n \ln n$ (for $c_5 = 2c_0$) whp., and during which, at the odd time steps a full quicksort is executed, while during the even time step a series of sets of block quicksorts is executed. We have analyzed the behavior of the protocol during the odd steps, so now we want to analyze the behavior of the protocol during the even steps. We focus on a single set of quicksort runs, which has a duration of $(2n/m - 1) \cdot c_5 m \ln m = O(n \ln \ln n)$. Let $t_1 \in [t_0, t_f)$ be the starting time and t_2 be the ending time of the sequence of $2n/m - 1$ quicksorts (see Figure 3).

Recall that while this set starts executing at time t_1, its input will be the ordering ρ that is the output of the full quicksort algorithm at time t_0. So, first we want to show that for every element u_i, the true rank at time t_1, $\pi^{t_1}(u_i)$, and the rank according to the output of the full quicksort at

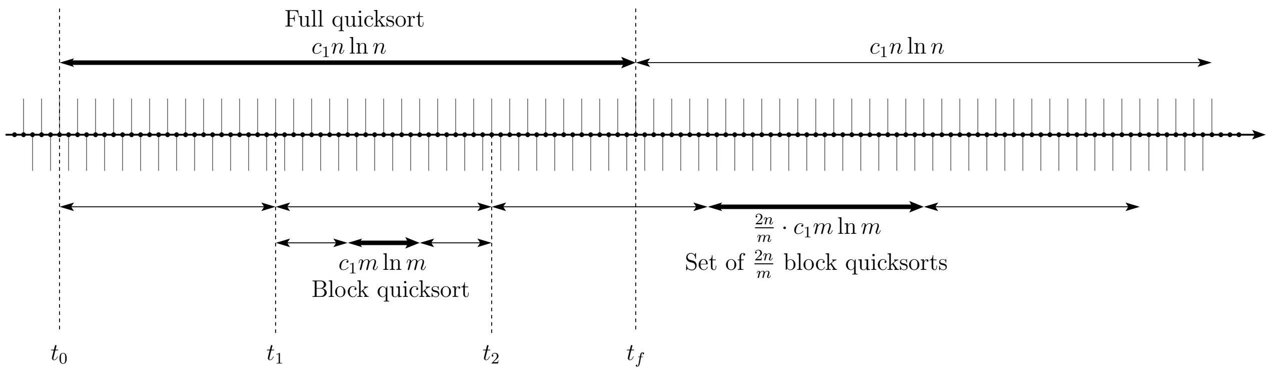

Figure 3: The periods of the execution of the sorting algorithm.

time t_0, $\rho(u_i)$, are within $O(\ln n)$ to each other. Let us, therefore, generally consider a time point $t \in [t_0, t_f]$. We then have whp.

$$\begin{aligned}\left|\pi^t(u_i) - \rho(u_i)\right| &\leq \left|\pi^t(u_i) - \pi^{t_0}(u_i)\right| + \left|\pi^{t_0}(u_i) - \rho(u_i)\right| \\ &\leq c_6 \ln n + c_7 \ln n,\end{aligned}$$

for appropriate constants c_6 and c_7. To see why the above expression is true, the first inequality follows from the Kendall tau being a metric. Let us now see why the second inequality is true. The first term expresses the difference in the (true) rank of element u_i between time points t_0 and t. Since $t \in [t_0, t_f]$ and since $t_f - t_0 \leq c_5 n \ln n$ we conclude that in expectation the rank of element u_i changes (by one) at most $2\alpha c_5 \ln n$ times, since the probability for element u_i to be selected to swap is at most $2\alpha/n$ in every time step. Therefore, by an application of the Chernoff bound, we conclude that for some constant c_6 the rank of element u_i does not change more than $c_6 \ln n$ whp. The second term of the inequality is bounded by making use of Lemma 6 (for $c_7 \geq 2c_3$, since in Lemma 6 we assumed that the quicksort algorithm is executed in all time steps while here it is executed only in the odd time steps): if the two ranks $\pi^{t_0}(u_i)$ and $\rho(u_i)$ differ by some value d, then it has to be the case that element u_i is incorrectly ordered with at least d other elements at time t_0.

Therefore, by applying a union bound over all elements and all time points $t \in [t_0, t_f]$, we can conclude that event $\tilde{\mathcal{E}}$ is true whp., where $\tilde{\mathcal{E}}$ is the event that "for all elements u_i and all time steps $t \in [t_0, t_f]$, we have that $\left|\pi^t(u_i) - \rho(u_i)\right| \leq c_8 \ln n$," for some constant c_8. Also we define the block size to be $m = 6c_8 \ln n$.

We will estimate the error at time t_2, and prove that it is at most $O(n \ln \ln n)$. Since the running time of each BlockSort is at most $O(n \ln \ln n)$, this implies that the error until the end of the next BlockSort is at most $O(n \ln \ln n)$. As in the proof of the simpler algorithm, we assume that the event $\tilde{\mathcal{E}}$ is true (since $\tilde{\mathcal{E}}$ holds whp., the expected error introduced if $\tilde{\mathcal{E}}$ does not hold is $o(n)$) and count the number of pairs (u_i, u_j) ordered incorrectly at time t_2. We divide these pairs into three groups:

1. (u_i, u_j) whose ordering (according to π) has changed at least once during $[t_1, t_2]$.

2. (u_i, u_j) for which there is a pivot u_k in one of the quicksorts runs during $[t_1, t_2]$ that was swapped with both u_i and u_j while it was the pivot.

3. all other (u_i, u_j)'s that are misordered at time t_2, that is, all pairs whose ordering does not change during $[t_1, t_2]$, and no quicksort pivot has swapped its order with both of them.

As in the previous section, since in each time step only one pair is swapped in π, the number of pairs (u_i, u_j) in the first group is bounded by $t_2 - t_1$, which is $O(n \ln \ln n)$ whp. For the second group, we use an argument similar to the one used in the previous section to bound $|B|$: since each element u_k is a pivot for at most $2m = O(\ln n)$ steps, the number of pairs (u_i, u_j) that a pivot u_k can be swapped with can be bounded by Y_k^2, where $Y_k \sim \text{Binomial}(O(\ln n), 2\alpha/n)$. Therefore, the expected number of pairs in the second group is at most $\mathbf{E}[\sum_k Y_k^2] = nO(\ln n/n) = O(\ln n)$. Furthermore, for each constant c, the probability that Y_k is greater than c is at most $O((\ln n/n)^c)$, and hence, whp., each Y_k is at most a constant. Also, $\sum_k Y_k \sim \text{Binomial}(O(n \ln n), 2\alpha/n)$, and therefore by the Chernoff bound, whp., it is at most $O(\ln n)$. Putting these together, we obtain $\sum_k Y_k^2 \leq \max_k\{Y_k\} \cdot \sum_k Y_k = O(\ln n)$, whp.

Finally, we bound the number of pairs in the third group. In fact, we will show that there is no such pair. Let (u_i, u_j) be a pair in the third group, and assume, without loss of generality, that $\rho(u_i) < \rho(u_j)$, that is, u_i is placed before u_j in the input to the algorithm BlockSort. By the argument in the proof of Theorem 2, since no pivot has swapped with both u_i and u_j, if they ever end up in the same quicksort block, they are ordered correctly at time t_2. Therefore, u_i and u_j must never end up in the same block. Thus, their ordering in the output of BlockSort is the same as their ordering in the input, i.e., u_i is ordered before u_j in $\tilde{\pi}^{t_2}$. This means that since u_i and u_j are misordered at time t_2 and their correct ordering (i.e., according to π) has not changed during $[t_1, t_2]$, u_j must be ordered before u_i in π^t for every $t \in [t_1, t_2]$ ($\pi^t(u_j) < \pi^t(u_i)$). Given that the event $\tilde{\mathcal{E}}$ is true, this implies:

$$|\rho(u_i) - \rho(u_j)| = \rho(u_j) - \rho(u_i) \leq \rho(u_j) - \pi^t(u_j) + \pi^t(u_i) - \rho(u_i) \leq 2c_8 \ln n.$$

Therefore, since the length of each block is $m = 6c_8 \ln n$, the only way u_i and u_j are not sorted in the same quicksort block is if at some point in BlockSort, u_i is selected to be included in a block B while u_j is not (which happens if the right limit of a block B is at some rank $r \in [\rho(u_i), \rho(u_j))$), and in the output of the quicksort on this block, u_i is among the first $m/2$ elements (and hence is not included in the next block, which would contain u_j). For this to happen, from the first $\frac{2}{3}m$ elements in the block B according to their ordering before running quicksort, at least $m/6$ must be ordered after u_i by the quicksort algorithm on this block (otherwise, u_i would not be among the first $m/2$ elements in the output of quicksort). Let u_k be any such element. Since u_k is among the first $2m/3$ elements of the block B, we must have $\rho(u_k) \leq r - m/3$ (recall that r is the right limit of the block B). Since the event $\tilde{\mathcal{E}}$ holds, we must have:

$$\pi^t(u_k) \leq r - m/3 + c_8 \ln n = r - c_8 \ln n, \tag{5}$$

for any $t \in [t_1, t_2]$. On the other hand, since u_j is ordered before u_i according to π^t, we have

$$\pi^t(u_i) > \pi^t(u_j) \geq \rho(u_j) - c_8 \ln n > r - c_8 \ln n. \tag{6}$$

By inequalities (5) and (6), the element u_k is before the element u_i in the ordering at any time t while the quicksort algorithm is running. So, the only way that the quicksort algorithm can make a "mistake" and rank u_i before u_k is if at some point during the running of this quicksort, a pivot swaps with both u_i and u_k. However, with high probability at most a constant number of elements in the block B are chosen for a swap while the quicksort on this block is in progress (since B contains $O(\ln n)$ elements and (by Proposition 3) quicksort lasts for $O(\ln n \ln \ln n)$ steps, the probability that more than c elements of this block are chosen for a swap while quicksort is running is at most $O(((\ln^2 n \ln \ln n)/n)^c))$. Therefore, whp. the quicksort does not make such a mistake for all the $m/6$ possible u_k's. This means that whp. there is no pair in the third group.

Putting everything together, we showed that whp. the number of pairs that are misordered at time t_2 is at most $O(n \ln \ln n) + O(\ln n) + 0 = O(n \ln \ln n)$. $\square$

As me mentioned previously, we assume that the algorithms know the initial permutation π^1; in that case Theorems 2 and 7 hold for every t. If π^1 is unknown then the algorithms do not have sufficient information during the first period. However, we are interested in the long-term behavior of the process and our results continue to hold after a full period, that is, for $t \geq cn \ln n$, for a constant c.

Naturally, the reader might wonder if by applying our technique one more time one can improve the bound from $O(n \ln \ln n)$ to $O(n \ln \ln \ln n)$. While such a result might be obtainable, it would require new techniques, since we cannot prove a result similar to Lemma 6 for the algorithm in this section. The reason is that the probability that an element moves more than $O(\ln \ln n)$ steps from its original position can be bounded by $1/\text{polylog}(n)$, instead of $1/\text{poly}(n)$, as it was the case in Lemma 6. This bound is insufficient to show that with high probability, the rank of *every* element in π and $\tilde{\pi}$ differ by at most $O(\ln \ln n)$.

3 Selection Problems

As we mentioned earlier, the dynamic data setting can capture many scenarios. In this section, we illustrate this by providing two more examples. First we show a simple algorithm for finding the element with minimum (or maximum) 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 a slightly different dynamic perturbation model than 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 except for m. If $u_i < m$, it replaces m with u_i.

Theorem 8. *Let m_t be the rank of the estimate at time t. In the steady state $\mathbf{Pr}(m_t \geq i) \leq \left(\frac{\alpha}{1+\alpha}\right)^i$, and $\mathbf{E}[m_t] \leq 1 + \alpha$.*

Proof. We can model the evolution of the rank $\pi(m)$ as a Markov chain on the nonnegative integers. The evolution of the value $\pi(m) - 1$ is dominated by the following Markov chain with states labeled from 0 to $n-1$: with probability $1/(n-1)$, the chain moves to state 0 (if it is not already there)

```
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**
```

Figure 4: Algorithm for computing the median. In every step the output of the algorithm is the latest element $\tilde{\mu}$ that has been computed in step 12.

and with probability $\alpha/(n-1)$, it moves from state i to state $i+1$, otherwise it remains at state i. It is easy to verify that the stationary probability of being at state $0 \leq i < n-1$ is

$$p_i = \frac{1}{1+\alpha}\left(\frac{\alpha}{1+\alpha}\right)^i,$$

and that for $i = n-1$ the probability is

$$p_{n-1} = \frac{1}{1+\alpha}\left(\frac{\alpha}{1+\alpha}\right)^{n-2}.$$

□

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 $k = 1, 2, \ldots, n$. Given k, our goal is to find an element u_i that minimizes the distance $\left|\pi^t(u_i) - k\right|$, 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 4 is a dynamic version of the median algorithm in [7], 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, while in the even steps we use the last set C computed in the odd steps to output the median estimate. At time t during a sorting phase of the set C, the output estimate $\tilde{\mu}_t$ is the element $\tilde{\mu}$ computed in the previous run of the sorting. We now show that the difference in the rank of the element returned by the algorithm is negligible.

Theorem 9. *Let $\tilde{\mu}_t$ be the output of algorithm Median at time t. For any time step t after the algorithm is run at least once (i.e., after $\Theta(n)$ steps), we have that $\mathbf{Pr}\left(\left|\pi(\tilde{\mu}_t) - \frac{n}{2}\right| = 0\right) \geq 1 - o(1)$, and $\mathbf{E}\left[\left|\pi(\tilde{\mu}_t) - \frac{n}{2}\right|\right] = o(1)$.*

Proof. The proof is based on the proof of the static version presented, for example, in [7].

We partition time into periods of length $\Theta(n)$, where each period corresponds to a full execution of steps 4–9 in Figure 4. (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., as we will see later.) 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, no element's rank changes more than $c\ln n$ during a period, for some constant c, whp. Also, since the quicksort call in line 6 of the algorithm takes $O(n)$ time and each element is a pivot for at most $O(n/\ln n)$ steps during this call, the probability that a pivot passes over more than one element during this call is at most $O(1/\ln n)$, and therefore by the analysis of quicksort in Section 2.2, with probability $1 - o(1)$, apart from pairs that change their relative position while line 6 is in progress, quicksort does not make any additional mistake.

The analysis of the static case as presented in [7] 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.

If those two facts hold then step 11 can be executed in sublinear time and in addition the algorithm can output an estimate in step 12.

We prove the first fact by using a similar argument as the one used in [7], but adapted to the case of dynamic data: For set C to contain all the elements that are medians in the entire next period, it means that for every time step t we have that $\pi^t(d) < \lceil n/2 \rceil$. 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 least $\frac{n}{72\ln n} - \sqrt{n}$ samples in R had rank smaller than $\frac{n}{2} - 2c\ln n$, when they were selected. We define $X_i = 1$ if the ith sample had rank smaller than $\frac{n}{2} - 2c\ln n$, and 0 otherwise. Then we have that $\mathbf{Pr}(X_i = 1) = \frac{1}{2} - \frac{2c\ln n}{n}$ and $\mathbf{E}\left[\sum X_i\right] = \frac{n}{72\ln n} - \frac{c}{18}$. We can apply the Chernoff bound and obtain:

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

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. We note that even though some of the elements that are included in the set C in step 9 of the algorithm may become less than d or more than u in the subsequent period when the set C is in use (or conversely, elements that were less than d or more than u might move to $[d, u]$), as long as these elements are not the median at any point (which whp. is true, since a median of C leaving $[d, u]$ or an element outside $[d, u]$ becoming the new median would require this element to move $|C|/2$ positions, which will not happen whp.), this does not affect our calculation of the median.

Next we show the second fact, that $|C| = O(\sqrt{n} \ln n)$. Again we adapt the argument of [7] to our case. Assume that we show that in the beginning of the period, call it time t_0, we have that

$$\pi^{t_0}(u) < \frac{n}{2} + 72\sqrt{n} \ln n - c \ln n \tag{7}$$

whp. This, combined with the fact that the rank of element u during a period does not change more than $c \ln n$ whp., implies that for the entire period we have that

$$\pi^t(u) < \frac{n}{2} + 72\sqrt{n} \ln n,$$

which in turn means that during the entire period fewer than $72\sqrt{n} \ln n$ elements of C have rank higher than $\lceil n/2 \rceil$. A similar argument can show that whp. during the entire period fewer than $72\sqrt{n} \ln n$ elements of C have rank lower than $\lceil n/2 \rceil$. These two facts imply that $|C| \le 144\sqrt{n} \ln n$.

From the discussion of the previous paragraph, it remains to show that whp. at time t_0 Equation (7) holds. For this to happen it means that the set R has fewer than $\frac{n}{72 \ln n} - \sqrt{n}$ samples among the largest

$$\frac{n}{2} - 72\sqrt{n} \ln n - c \ln n$$

elements. Now define X_i to be 1 if the ith sample of R is among the $\frac{n}{2} - 72\sqrt{n} \ln n - c \ln n$ largest elements, and 0 otherwise. Then we have that

$$\mathbf{Pr}(X_i = 1) = \frac{1}{2} - \frac{72 \ln n}{\sqrt{n}} - \frac{c \ln n}{n},$$

and

$$\mathbf{E}\left[\sum_{i=1}^{|R|} X_i\right] = \frac{n}{72 \ln n} - 2\sqrt{n} - \frac{c}{36}.$$

Then the probability that Equation (7) does not hold is bounded by

$$\begin{aligned}\mathbf{Pr}\left(\sum_{i=1}^{|R|} X_i \ge \frac{n}{72 \ln n} - \sqrt{n}\right) &= \mathbf{Pr}\left(\sum X_i - \mathbf{E}\left[\sum X_i\right] \ge \sqrt{n} + \frac{c}{36}\right) \\ &\le e^{-\frac{2(\sqrt{n} + \frac{c}{36})^2}{\frac{n}{36 \ln n}}} \\ &\le \frac{1}{n^3}.\end{aligned}$$

We have now established that whp. $|C| = O(\sqrt{n} \ln n)$ and that C contains all elements that are medians during the next period. Since sorting in step 8 takes $O(\sqrt{n} \ln^2 n)$ steps whp., 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 changes 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, E. Upfal, and F. Vandin. Algorithms on dynamic graphs. Manuscript, 2010.

[2] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis.* Cambridge University Press, 1998.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001.

[4] D. P. Dubhashi and A. Panconesi. *Concentration of Measure for the Analysis of Randomzed Algorithms.* Cambridge University Press, 2009.

[5] D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 8. CRC Press, 1999.

[6] R. Kleinberg. *Online Decision Problems with large Strategy Sets.* PhD thesis, MIT, 2005.

[7] M. Mitzenmacher and E. Upfal. *Probability and Computing.* Cambridge University Press, 2005.

[8] S. Muthukrishnan. *Data Streams: Algorithms and Applications.* Now Publishers Inc., 2005.

[9] D. Ron. Property testing: A learning theory perspective. In *Foundations and Trends in Machine Learning*, volume 1, pages 307–402. 2008.

[10] D. Ron. Algorithmic and analysis techniques in property testing. In *Foundations and Trends in Theoretical Computer Science*, volume 5, pages 73–205. 2009.

[11] 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.

[12] C. Swamy and D. B. Shmoys. Approximation algorithms for 2-stage stochastic optimization problems. *SIGACT News*, 37(1):33–46, 2006.

[13] L. L. Thurstone. *The Measurement of Values.* The University of Chicago Press, 1959.