Dynamic Data: Model, Sorting, Selection

Andrew Moreland – andymo@stanford.edu

May 15, 2014

---

1 Introduction

This paper is intended as an introduction to and explanation of *Sorting and Selection on Dynamic Data*[1], a paper published by Anagnostopoulos et al. in an attempt to address the problems that the authors encountered at Bix, a crowd-polling site that was operated by Yahoo in the mid-to-late 2000s. One of the unique issues that Bix faced was the problem of sorting a dataset that did not have an explicit total ordering, and which changed over time. In contrast to the classic problem of sorting integers, it is hard to compare the popularity of arbitrary real-life objects. So, Anagnostopoulos et al. extracted the essential features of the data they were working with into the "Dynamic Data Model" and then designed sorting and order-selection algorithms for this model. In particular, they present sorting algorithms which are able to achieve expected $O(n \log(n))$ and $O(n \log \log(n))$ error-bounds on dynamic data.

These theoretical bounds are interesting, but it's also valuable to measure how these algorithms perform in practice. In addition to walking through the results of Anagnostopoulos et al., we present the results of experiments run on a Dynamic Data simulator. We show that the theoretical bounds predicted in the paper hold up very well in practice by measuring various facets of the simpler sorting algorithm across a wide range of problem parameters.

2 Dynamic Data Model

Anagnostopoulos et al. propose the Dynamic Data Model for problems that share the following characteristics:

1. Time can be discretized into short timesteps
2. There is a domain S of objects that we are interested in
3. There exists an objective total-order on S, π
4. We are allowed to make limited pairwise comparison queries on π

5. π changes slowly as time progresses

This model captures the essence of the problems that Bix faced: they were able to ask their users to make comparisons between a pair of objects drawn from a predefined set (e.g. famous actors) on which there more-or-less exists a total ordering. Moreover, the popularity of actors does change but not especially quickly in the grand scheme of things, so it's reasonable to require that π change slowly.

3 Sorting Problem: Formal Statement

At this point we will introduce the formal statement of the main problem that the authors of the paper solve.

1. We have a domain, S, of objects that we'd like to sort
2. We are allowed to make a constant number α of pairwise queries per timestep t_i
3. "Nature", a non-adversarial random process makes α pairwise swaps of neighbors on the total order π per t_i.
4. We want to minimize the Kendall τ-Distance (defined in next section) between $\hat{\pi}$, our ordering, and π, the real order
5. Our algorithm runs for an arbitrarily long time period

This problem statement captures the main elements of the sorting problem on dynamic data: most of the parts of the statement are straight-forward. Perhaps the most unintuitive element of the model is that the algorithm does not terminate. Recall that the original context of this problem was the polling website Bix. Bix always wanted to know the correct ordering of the elements that were being compared by their users. So, it doesn't make sense to stop at any arbitrary point and declare the algorithm's generated order "final" because the Nature process continues running. Our solution must be able to continually adapt to the transpositions that Nature makes, so we need to keep executing it.

3.1 Kendall τ-Distance

Since the Kendall τ-Distance is the metric that our algorithm will attempt to minimize, it is worth spending some time here defining and exploring this metric.

First, Kendall τ-Distance is defined for permutations π_1, π_2 over some domain S as:

$$K(\pi_1, \pi_2) = |\{(i,j) : i < j, (\pi_1(i) < \pi_1(j) \wedge \pi_2(i) > \pi_2(j)) \vee (\pi_1(i) > \pi_1(j) \wedge \pi_2(i) < \pi - 2(j))\}|$$

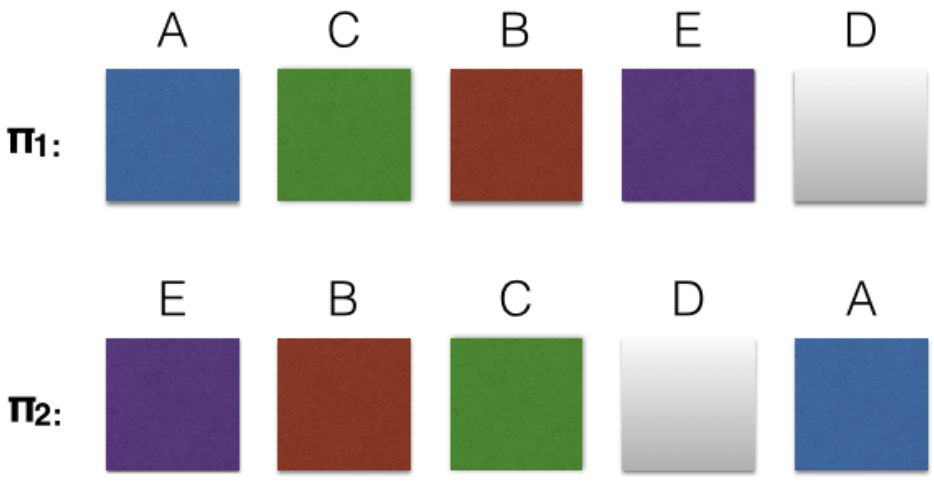

Figure 1: In this example, element A contributes 4 *to the distance, C contributes* 2 *and B contributes* 1*. So,* $K(\pi_1, \pi_2) = 4 + 2 + 1 = 7$

This is a dense definition, so we will unpack it into an intuitive definition. Essentially, we we count the number of pairs of elements of S which ordered differently by the permutations. See Figure 1 for an example computation.

Notably, if we let $n = |S|$ (a convention we will use for the rest of the paper), then we see that the number of pairs of elements is $\Theta(n^2)$. This means that in the absolute worst case, two permutations can have τ-Distance $\Theta(n^2)$. The lower bound is trivial: if two permutations are identical, then the τ-distance is clearly just 0.

Finally, we should note that this definition of distance implies that the distance between $\pi^{(t)}$ and $\pi^{(t+1)}$ (π at time t and π at time $t+1$) is at most α as Nature is limited to swapping neighbors in the ordering. Each of these at most α swaps leads to a distance increase of 1. This does a good job of capturing the the "slow change" of π that we wanted to model.

3.2 Feasibility of the Problem

At this point it is worth asking whether or not this is a reasonable problem at all. In order to sort the elements of S we have to use some sort of comparison-based sorting algorithm. Importantly, we are only allowed a fixed α number of comparisons between elements per discrete time step. Unfortunately, Nature is allowed to make α pairwise swaps per time step as well. A reasonable person might worry that Nature's random swaps could, perhaps with high probability, interfere with our sorting algorithm to the extent that we are unable to guarantee significantly better than the worst case τ-distance $\theta(n^2)$ between our output $\hat{\pi}$ and the true order π.

Imagine that we are executing bubble sort in a context where $\alpha = 1$. Suppose that during time t_i we compare the first and second items of S under our current ordering $\hat{\pi}$, a_1 and a_2. We might find that $a_1 > a_2$, so, we would decide to swap a_1 with a_2 in our ordering. However,

during this time step Nature could randomly decide that it wants to swap the true order of a_1 and a_2. As a result, after we move to time t_{i+1} we note that we have made no improvement in $K(\pi, \hat{pi})$. This is worrying. One thing that is encouraging, however, is that for us to have made no progress in this case Nature must have randomly permuted precisely the elements we chose to swap. For sufficiently large n, this will happen with very low probability in general! Hopefully the probability of interference is actually low enough that we can construct a sorting algorithm which is able to do better than $\theta(n^2)$ τ-Distance.

3.3 Lower Bound on Distance

Before we start to build such an algorithm we should explore how well any algorithm we construct could possibly perform. In the paper, Anagnostopoulos et al. provide a lower bound:

Lemma 3.1. *For any sorting algorithm for the Dynamic Data model, at any time $t > n/8$ the expected distance $K(\pi, \hat{\pi})$ between the optimal ordering and the output ordering is $\Omega(n)$ with high probability.*

Intuitively this makes a lot of sense. For instance, if we consider a time interval $I = [t, t + \frac{n}{100}]$ (for sufficiently large n that this is not miniscule), and assume that $\alpha = 1$, then we can see that our algorithm is only able to see the relative ordering of at most $\frac{n}{50}$ elements since it examines 2 per timestep. Moreover, at each discrete time step there is a constant probability γ that the pair of elements that Nature chooses to swap is completely disjoint from these $\frac{n}{50}$ elements. This means that, in expectation, we expect our algorithm to be completely ignorant of the ordering of $\gamma\frac{n}{50}$ elements in S. Note that this directly gives the $\Omega(n)$ bound.

We can formalize this intuition by using the Principle of Deferred Decisions, but we have chosen to omit this proof because it follows essentially the same reasoning as the intuition and the details are somewhat painful to work out. A rough sketch is that we can show that if Nature wishes, it is able to defer a linear number of its random swaps to the end of any time interval without changing the operation of the algorithm or the distributions of the permutations. The set of these deferred decisions must be completely disjoint from the set of elements touched by the algorithm during its execution. This means, as we intuited, that there are a linear number of swaps that the algorithm cannot observe.

This lower bound is important because it gives us a target for the performance of our algorithms.

4 Dynamic Quicksort

Now that we know what kind of error bound we are shooting for we can begin to develop an algorithm to solve the sorting problem. The algorithm that the authors propose for solving the problem on dynamic data is basically just a run-of-the-mill Randomized Quicksort. Psuedocode for Randomized Quicksort is presented in Algorithm 1.

Algorithm 1 Randomized Quicksort

```
function QUICKSORT(arr)
    if len(arr) <= 1 then
        return arr
    end if
    p ← e ∈ arr at random
    larr ← []
    rarr ← []
    for all a ∈ arr do
        if e ≤ p then
            larr+ = e
        else
            rarr+ = e
        end if
    end for
    return QUICKSORT(larr) :: p :: QUICKSORT(rarr)
end function
```

Quicksort is extended into Dynamic Quicksort by executing it repeatedly. When an interation of Quicksort terminates at time step t_i, the permutation $\hat{\pi}^{(t_i)}$ is updated. Until the next quicksort phase terminates at time t_j, when our algorithm is queried for the ordering of two elements, $\hat{\pi}^{(t_i)}$ is used. Since we know that quicksort executing on static data takes $O(n \log n)$ timesteps in expectation, we can assume that each phase of quicksort executing on dynamic data will take $\Omega(n \log n)$ timesteps in expectation. This means that for $\Omega(n \log n)$ timesteps, we expect to output an identical permutation. If we apply the reasoning of Lemma 3.1, we see that we cannot expect a τ-distance bound superior to $O(n \log n)$ from Dynamic quicksort. It turns out that this is the bound that we will achieve.

Theorem 4.1. *For every t, $KT(\pi^{(t)}, \hat{\pi^{(t)}}) = O(n \log n)$ in expectation and with high probability.*

Note that in this Theorem, it is not permitted to replace Quicksort with an arbitrary comparison based sort that achieves $O(n \log n)$ time bounds. We will use specific properties of quicksort in order to prove this Theorem.

The first step in proving Theorem 4.1 is to show that randomized quicksort operating on dynamic data takes uses roughly the same number of comparisons as quicksort operating on static data.

Proposition 4.2. *The running time of randomized quicksort in the dynamic-data model is $O(n \log n)$ in expectation and with high probability.*

Proof. In order to show this, we ask that the reader recall the standard method of analysis for randomized quicksort. In this analysis, we define a pivot chosen for the partition step of quicksort to be *good* if it partitions the array into subarrays of size at least γn, where γ is some

constant fraction. Simple analysis shows that this gives an $O(\log n)$ bound on the number of good pivots in any path down the recursion tree.

Next, we note that any pivot has a constant probability of being good. This means that the number of bad pivots in a path is distributed according to the negative binomial distribution, since once $O(\log(n))$ good pivots are chosen we can no longer see bad pivots. We can bound the number of bad pivots to a constant multiple of the number of good pivots by applying a Chernoff bound to the negative binomial distribution. [1] This means that the total length of each path is at $O(\log n)$ with high probability because the bad pivots add at most a factor of $\log(n)$ to the length.

Since there are at most n paths in our tree, we can use a union bound to show that the number of comparisons we make is $O(n \log n)$ in expectation and with high probability.

To extend this proof to dynamic data, we note that if a pivot chosen during the partitioning phase is good, initially γn elements should be on either side of it in the sorted array. However, since the partitioning step takes $O(n)$ time to execute, we expect some drift in the order. Applying the reasoning behind Lemma 3.1, we expect that we should still retain at least $\frac{\gamma}{2}n$ elements in each of the subarrays by the end of the partitioning phase. Since this is another constant fraction, the pivot remains good with high probability. This means that our proof for the static case holds for the dynamic case as well.

□

Now, consider the execution of randomized quicksort between time t_0, and its completion at time t_1. We expect $O(n \log n)$ steps in its execution, which corresponds to $O(n \log n)$ comparisons between elements of S. If we assume that $\alpha = 1$ (alpha only effects constants in the following proofs) then we should expect roughly $O(n \log n)$ timesteps. This insight allows us to bound the $K(\hat{\pi}, \pi)$ at any timestep.

First, assume that at t_0, $K(\hat{\pi}^{(t_0)}, \pi^{(t_0)}) = O(n \log n)$. Since we continue to use this permutation until the next time quicksort completes, it suffices to bound the amount that our error increases. With high probability, the number of steps until the next permutation is computed is $O(n \log n)$. Since Nature can only introduce 1 error per time step (because $\alpha = 1$), $K(\hat{\pi}^{(t_0)}, \pi^{(t_j)}) = O(n \log n)$ for $j \in [t_0, t_1]$.

Now, we need to show that we achieve $O(n \log n)$ τ-distance immediately after the execution of a quicksort phase. Because quicksort is a correct algorithm on static data, we can see that all of the errors that Dynamic Quicksort makes must be due to Nature shifting data underneath it. We will divide the incorrectly ordered pairs (a, b) into two classes:

Class A: This class consists of pairs (a, b) which were correctly sorted by Dynamic Quicksort at some time t_j in the interval $[t_0, t_1]$. If (WLOG) $\pi^{(t_1)}(a) < \pi^{(t_1)}(b)$ but $\hat{\pi}^{(t_1)}(a) > \hat{\pi}^{(t_1)}(b)$, then we see that at some time t_k after t_j Nature must have swapped a and b.

[1] Chernoff bounds are a useful probabilistic tool. When applied to $X \sim Bin(n, p)$ with $p > 1/2$, the statement is: $P[X \leq n/2] \leq e^{-\frac{1}{2p}n(p-\frac{1}{2})^2}$. We can rewrite the negative binomial distribution in terms of a binomial distribution in order to get our bound.

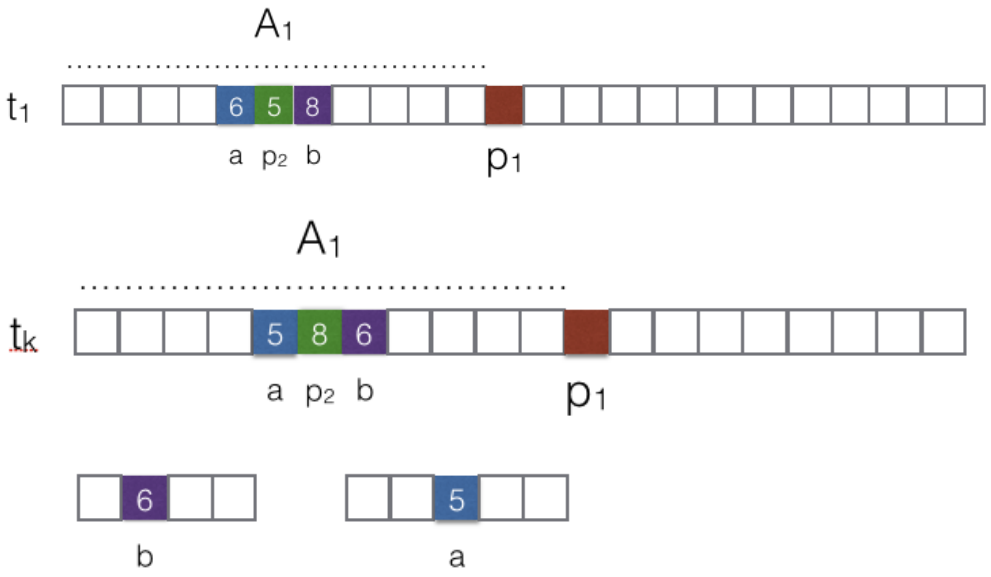

Figure 2: At timestep t_1 we see that $a <_\pi b$, i.e. that $\pi(a) < \pi(b)$. Suppose that a swaps with p_2, a pivot chosen to partition subarray A_1. During the partitioning step, a will be assigned to the right subarray. If b is then swapped with p_2, b will be assigned to the left subarray. Note that from now on, our quicksort execution will believe that $b < a$, but we see that at no point has that been the case.

Class B: This class consists of pairs (a, b) which were never corrected sorted by Dynamic Quicksort. Assume that $\pi(a) < \pi(b)$ for all of $[t_0, t_1]$. If dynamic quicksort outputs $\hat{\pi}^{(t_1)}(a) > \hat{\pi}^{(t_1)}(b)$, then we see that during some partition phase a and b must have ended up in the incorrect subarrays. This can only happen if each were transposed with the pivot for that phase, p. An example of this error is shown in Figure 2.

These classes of errors are exhaustive, so if we manage to bound each of these classes in a suitable way then we will have bounded the overall error that creeps into the ordering that a phase of quicksort computes.

Lemma 4.3. *If t_1 is the end of a quicksort phase, $K(\hat{\pi}^{(t_1)}, \pi^{(t_1)}) = O(n \log n)$ with in expectation and with high probability.*

4.1 Bounding the Errors

Lemma 4.4. *Class A contributes $O(n \log n)$ in expectation to $K(\hat{\pi}^{(t_1)}, \pi^{(t_1)})$ with high probability.*

Proof. The first class is easy to bound. In expectation there will be at most $O(n \log n)$ timesteps during the execution of the algorithm. Nature is allowed to make one pairwise swap during each of these steps. This means that at most $O(n \log n)$ errors can be introduced in the part of the domain that have been "fixed" prior to termination of a quicksort phase. □

Lemma 4.5. *Class B contributes $O(n \log n)$ in expectation to $K(\hat{\pi}^{(t_1)}, \pi^{(t_1)})$ with high probability.*

Proof. The key insight that is needed to prove this lemma is the observation that in order for (a, b) to be a member of class B, both a and b must have been swapped with a pivot during some partitioning phase. Therefore, it suffices to show that pivots are not swapped too often.

Let X_k denote the number of steps that pivot k is active in the execution of the algorithm. Since exactly one pivot is active during any point in the execution of the algorithm, $\sum_k X_k$ is precisely the number of time steps a quicksort phase takes to execute. Since, with high probability, a quicksort phase requires $O(n \log n)$ timesteps, we will condition on this being true. Therefore:

$$\sum_k X_k = O(n \log n) \leq c_0 n \log n \tag{1}$$

Since any given pivot is active only during a single linear time partitioning phase, $X_k \leq n$ for all k. By convexity, we see that:

$$\sum_k X_k^2 \leq c_0 n^2 \log n \tag{2}$$

since the sum is maximized when it's a combination of "maximized" elements. This corresponds to $O(\log n)$ X_k set to $O(n)$.

Now, define Y_k to be the number of time steps for which element u_K was an active pivot and was swapped by Nature with another element. The probability that any element is swapped during a timestep is $2/n$, since exactly one pair is swapped. Clearly $Y_k \sim \text{Binomial}(X_k, p)$ if we condition on X_k, and let $p = 2/n$.

Recall that a pair (a, b) is in class B only if each of a and b are swapped with some pivot. Observe that $\binom{Y_k}{2}$ corresponds to the maximum number of pairs that u_k could have been swapped with while active. Now,

$$\binom{Y_k}{2} \leq Y_k^2$$

so Y_k^2 serves as an upper bound on $|B|$. If we define S to be $\sum_k Y_k^2$, then we can bound B as follows:

$$E[S] = E[\sum_k Y_k^2] = E[E[\sum Y_k^2 | X_k]] = E[\sum_k E[Y_k^2 | X_k]]$$

by linearity of expectation. Now,

$$= E[\sum_k (Var[Y_k|X_k] + E[Y_k|X_k])^2] = E[\sum_k (X_k p(1-p) + X_k^2 p^2)]$$

by definition of expectation for a binomial variable. Finally,

$$= E[p(1-p)\sum_k X_k + p^2 \sum_k X_k^2] \leq^{(1),(2)} (2c_0(1-p) + 4c_0)\log n \leq c_1 \log n$$

for some constant c_1. This gives the expectation that we need. Now we want to bound the probability that B strays too far from its expectation. Here the authors apply Azuma's inequality[2], which is a result for bounded-difference Martingales. [2]

We can directly apply this inequality to bound the probability that B exceeds $O(n \log n)$ by more than a constant multiple:

$$P[S - E[S] > c_2 n \log n] \leq exp(\frac{-2c_2^2 n^2 \log^2 n}{\sum_k X_k^2}) \leq n^{-2c_2^2/c_0}$$

which is very small for sufficiently large c_2.

This shows that class B is size $O(n \log n)$ in expectation and with high probability. □

Together Lemma 4.4 and Lemma 4.5 prove Lemma 4.3, which concludes our proof of Theorem 4.1, so we have that at any time t our Dynamic Quicksort outputs an ordering which is $O(n \log n)$ τ-Distance from the correct ordering.

5 Improved Dynamic Quicksort

The main result of the paper is a sorting algorithm for the Dynamic Data Model with $O(n \log \log n)$ expected τ-Distance. Most of the analysis to prove this algorithm is the same (at least in spirit)

[2] Assume that for all i, $0 < X_i < d_i$ are independent random variables, and let $S = \sum_{i=1}^n X_i$. Then:

$$P[S - E[S] > \lambda] < exp(-2\lambda^2 / \sum_i d_i^2)$$

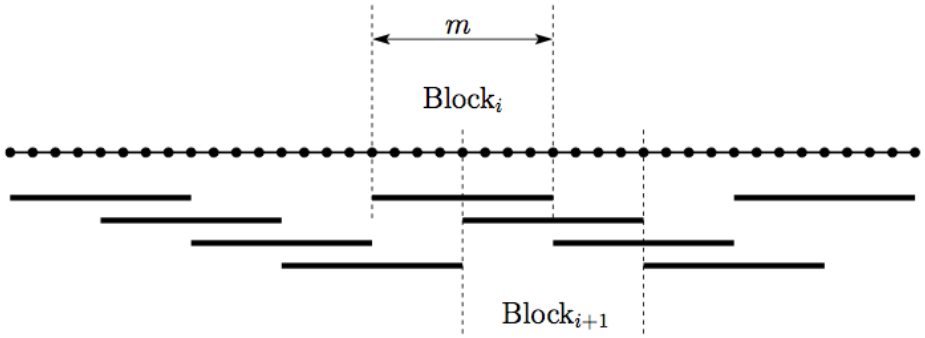

Figure 3: Each block is size $O(\log n)$. Small quicksorts are run on each of the blocks in between steps of a larger global quicksort.

as what we did to prove the $O(n \log n)$ bound. The way that the algorithm works is that we interleave a global quicksort over the entire domain with smaller quicksorts on overlapping subdomains of size $O(\log n)$. This exploits a corollary of Theorem 4.1 which is that during the execution of global quicksort, we expect the average element to stray at most $O(\log n)$ from its correct position. The smaller quicksorts work to counteract this drift while the global quicksort avoids any large-scale errors. An example of the partioning scheme used in order to perform the smaller quicksorts is in Figure 3. In order to answer queries during the continuous execution of these quicksorts, we simply output the most recently completed result. The paper provides a proof that this results in an output permutation with an $O(n \log \log n)$ τ-Distance from the correct ordering at any time step.

6 Experiments

In order to explore the performance of Dynamic Quicksort, we implemented a simulator for the Dynamic Data environment in Java and then implemented Quicksort on top of it. We ran several experiments, the results of which are shown in the following figures. Overall, it seems that the predicted bounds are very accurate in practice.

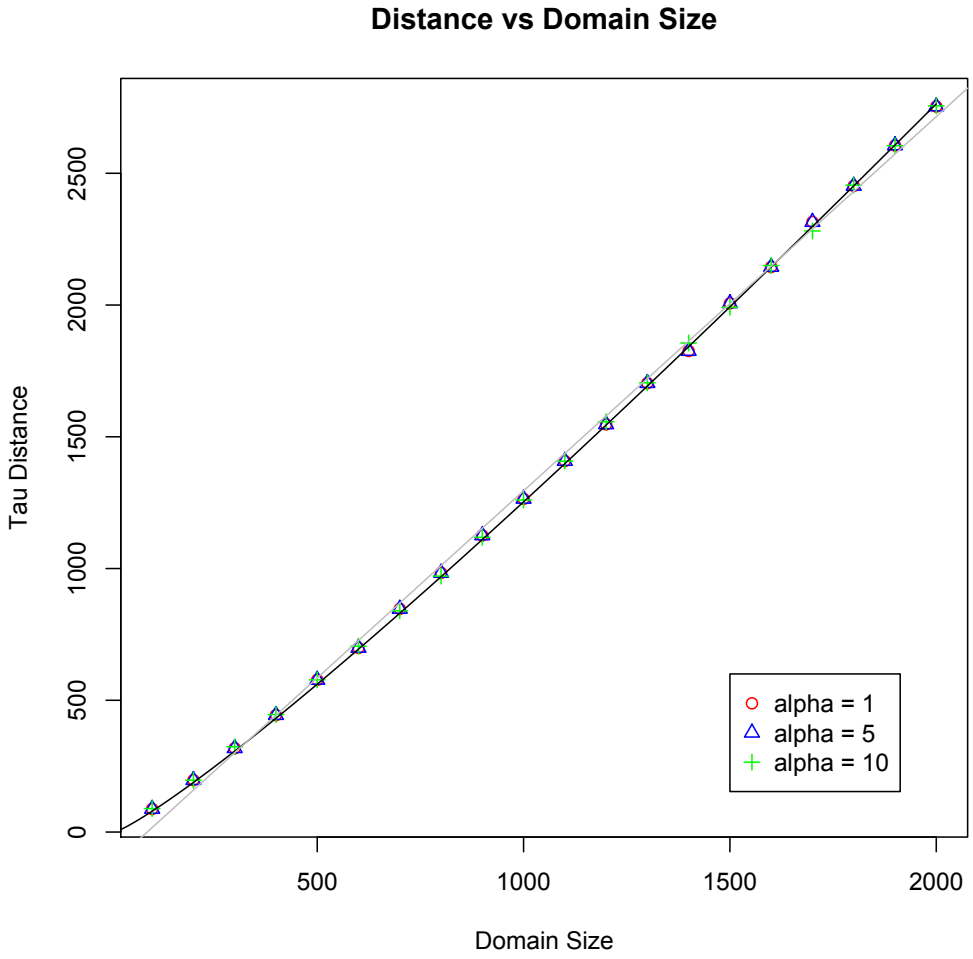

Figure 4: We executed Dynamic Quicksort with various values of α on domains of various size. Each trial was repeated 100 times. The black curve is an $n \log n$ curve fit to the data. The grey curve is linear. Observe that the $n \log n$ curve has a superior fit to the curve's shape, particularly at the start and end. Note also that the results are stable even through variations in α.

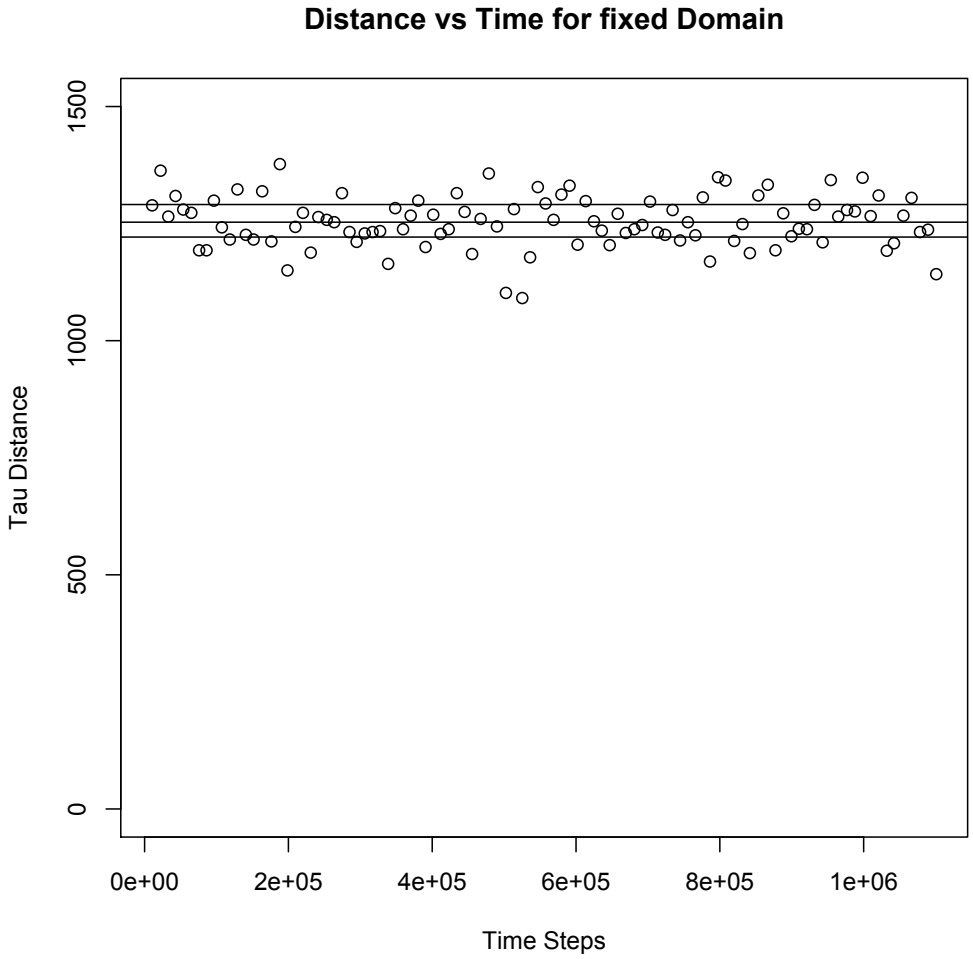

Figure 5: This shows the τ-Distance of the permutation when queried at various time steps throughout its execution. This is a single run of the algorithm on a domain of size 1000 *with* 100 *sorting passes and* $\alpha = 1$*. The bars are* $25th$*,* $50th$ *and* $75th$ *percentile lines. Note that the distance is grouped fairly tightly around the mean for the duration of the execution and shows no increasing or decreasing drift.*

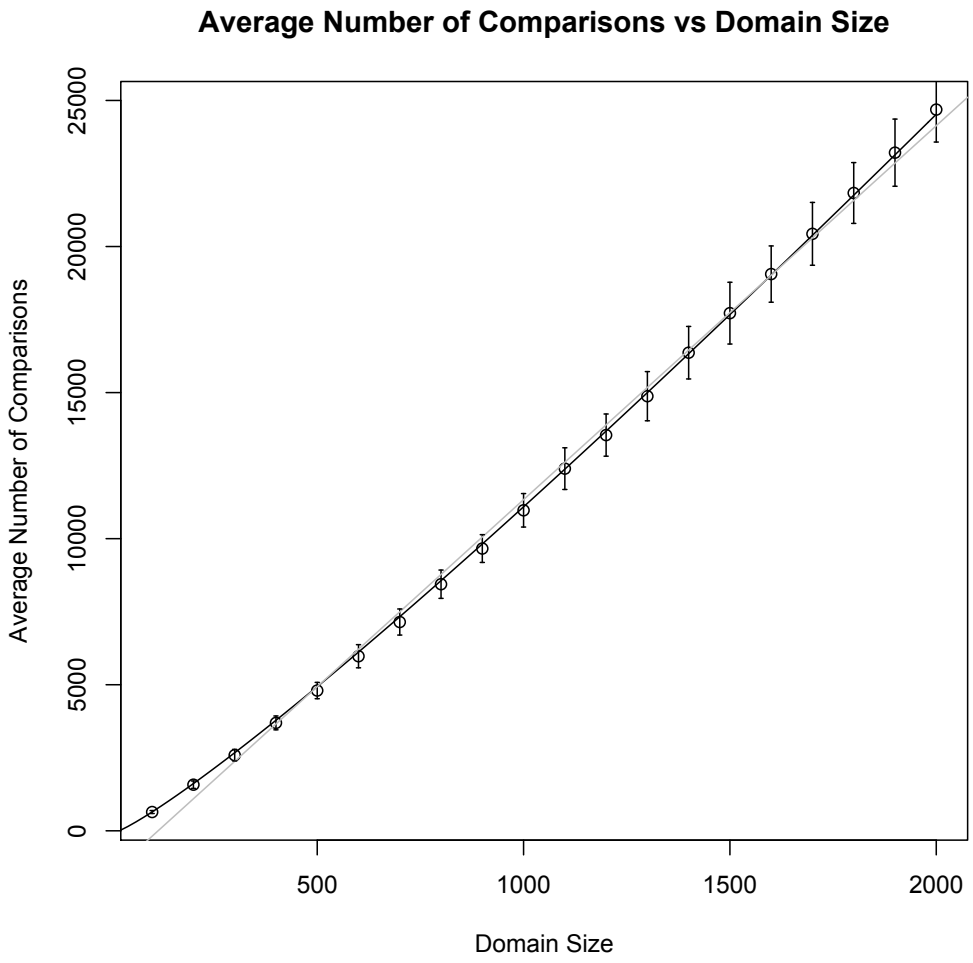

Figure 6: This shows the number of comparisons made by the Dynamic Quicksort algorithm during each of the 100 *sorting passes per data point.* α *was fixed at* 1*. The fit-lines are as before, and the bars show* 1 *standard deviation in each direction. Again, we see an* $O(n \log n)$ *trend as predicted.*

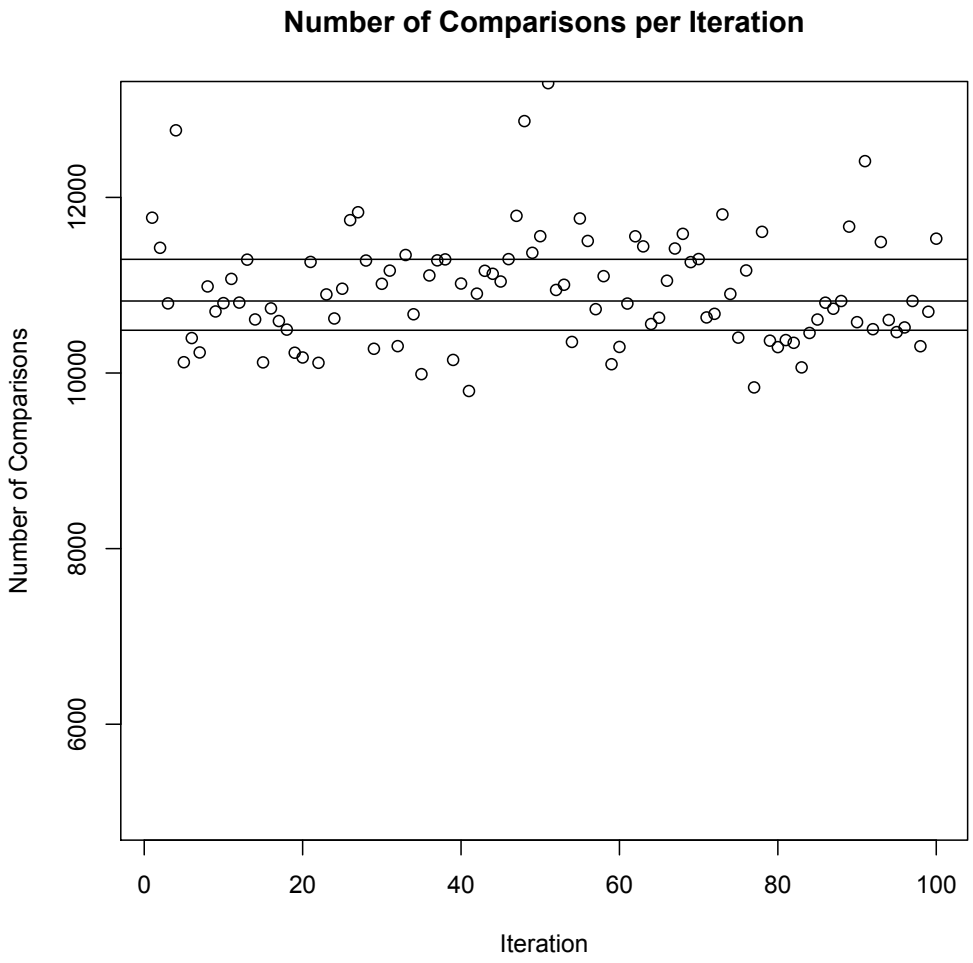

Figure 7: This shows the number of comparisons per iteration of a sorting pass for a domain of size 1000 *and* $\alpha = 1$*. These are again* $25th$*,* $50th$*,* $75th$ *percentile lines. Note that there are some extreme outliers, but it does not appear that there is a slanted trend in either direction. This is as predicted.*

7 Conclusion

In this paper we have thoroughly explored the Dynamic Data Model proposed by Anagnostopoulos et al. and sorting algorithms based on it. The main takeaways from the paper are:

1. Using yesterday's results can allow us to answer questions on dynamically changing data (see Google's PageRank)
2. Breaking apart a problem into smaller pieces is often a good idea
3. Running time is not the only important metric for an algorithm

If the reader is interested in learning more, Anagnostopoulos has another paper that explores a similar model for dynamic graphs. The full paper also uses Dynamic Quicksort as a primitive for constructing k-th order statistic algorithms.

Overall, this paper gives a very viable (and practical) solution to the problem of sorting on dynamic data.

8 Acknowledgements

Thanks to Professor Roughgarden and Rishi Gupta for their comments on the presentation that this paper was based off of.

9 References

[1] A. Anagnostopoulos, R. Kumar, M. Mahdian, and E. Upfal. *Sorting and Selection on Dynamic Data*. In *Theoretical Computer Science*, Vol 412, 2011.

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