Large Scale Percent-Sorted Decision Randomized Algorithm

Nathan Hourt and Jason Parham

May 6^{th}, 2013

Abstract

For a list, L, of n elements (with n being very large) to be α-approximately-sorted (with $0 \leq \alpha \leq 1$), this implies that at least $\lceil n\alpha \rceil$ of the elements are sorted by some objective standard and that the rest of the elements are unsorted. The exact definition of sortedness of an element is an application domain concern, and in this paper we make no assumptions about how element sortedness is defined nor the type of information in the list. Therefore, the decision of sortedness is abstracted away from the authors and embodied in an oracle function. This oracle can examine a given list element, ϵ, along with other elements in the list to determine, in constant time, whether ϵ is sorted. The authors feel that an application developer could reasonably implement such an oracle using application-specific knowledge on how unsorted elements appear in the list.

We will describe an algorithm which estimates the exact sortedness (β) of the list. The algorithm uses the aforementioned oracle to give an estimate of β, β', which is close to β with high probability. It should be noted that we are only interested in a value of β that is large ($0.9 \leq \beta$) so that unsorted elements are relatively rare, which is conducive to the formulation of the oracle described above. If such an oracle cannot be created, then this algorithm and analysis will not apply.

The output of this algorithm, β', can be compared with α to determine if a list is α-approximately-sorted, assuming α is not too close to β. If α is too close to β, this randomized algorithm cannot guarantee an estimate accurate enough to decide α-approximately-sortedness reliably.

1 Introduction

Given that an oracle can reasonably exist for an α-approximately-sorted list, we focus our attention on the complexity of a randomized algorithm that could decide such a list. Specifically, we focus our attention on a randomized algorithm that is *sub-linear*. Our algorithm will sample a sub-linear number of elements in an α-approximately-sorted list and determine if it is either 1.) α-approximately-sorted with a high probability 2.) not α-approximately-sorted with a high probability or 3.) inconclusive. For the algorithm to be inconclusive, the queried α must be very close to the actual percent-sortedness of the list. The algorithm is better designed for estimating the percent-sortedness of the list with a high probability as opposed to a preceise decision problem.

We show using Chernoff bounds that, using our algorithm, the difference between the actual percent-sortedness and the estimated percent-sortedness is tightly bounded with high probability. We follow up our analysis of the algorithm with an empirical experiment of the algorithm with a list of 10^{10} numbers. Using our experimentation, we can get an estimation of the percent-sortedness of the list within acceptable limits. The experiment is implemented in C++, which was chosen for computational speed and ease of random access of a stored blob of $\Omega(10^{10})$ numbers.

The rest of the paper is as follows: Section 2 gives the sub-linear, randomized algorithm that we later analyze and implement. Section 3 gives our theoretical analysis of the algorithm using a Chernoff bound to lower-bound the probability that the algorithm gives a wrong answer.

Section 4 gives our experimental results of the algorithm implemented in C++ for lists of variable percent-sortedness and for lists of $25 \cdot 2^{30} \approx 2.6 \cdot 10^{10}$ numbers. Section 5 concludes the paper with a synopsis of the results from the experiment and theoretical analysis. An appendix has also been included with the C++ source code used during the experimentation.

2 Algorithm

Given a list, L, of n elements we would like to determine the sortedness of L. We assume the existence of an oracle function, $Q(\epsilon)$ with $\epsilon \in L$, that returns in constant time $True$ if ϵ is sorted in L, $False$ otherwise.

Let P_Q be the probability of the oracle Q correctly deciding if ϵ_i is sorted, where $0.5 < P_Q \leq 1$.

The algorithm is as follows:

1. Let s be a counter initialized to 0.
2. Loop over i from $1..\lceil \sqrt{n} \rceil$.
   (a) Pick an element, ϵ_i, from L uniformly at random with replacement
   (b) Increment s by 1 if $Q(\epsilon_i)$ is $True$.
3. Calculate $\beta' = \dfrac{s}{\lceil \sqrt{n} \rceil}$
4. Adjust β' to account for P_Q: $\beta' = \frac{\beta'}{2P_Q - 1}$
5. Return β'.

3 Analysis

We will analyze the algorithm using a Chernoff bound. The Chernoff bound that we want to show will upper bound the error probability. We will assume that we have a list, L, of n elements and an oracle function, $Q(\epsilon)$ with $\epsilon \in L$, that returns in constant time $True$ if ϵ is sorted in L, $False$ otherwise.

Let β be the actual, perfect knowledge percent sorted for the list L; i.e. if L has n elements of which exactly ϕ are sorted, then $\beta = \frac{\phi}{n}$.

Let β' be the algorithm's calculated, estimated value of β.

Let P_Q be the probability of the oracle Q correctly deciding if ϵ_i is sorted, where $0.5 < P_Q \leq 1$.

Let $X_i = \begin{cases} 1, & \text{if } \epsilon_i \text{ is sorted} \quad \text{with probability } \beta P_Q + (1-\beta)(1-P_Q) = 2\beta P_Q - \beta - P_Q + 1 \\ 0, & \text{otherwise} \end{cases}$

$E[X_i] = 2\beta P_Q - \beta - P_Q + 1 = \mu$

Therefore, $X_i \in \{0, 1\}$ and $X = \displaystyle\sum_{i=1}^{\lceil \sqrt{n} \rceil} (X_i)$

It should be noted that $\beta' = \dfrac{X}{\lceil \sqrt{n} \rceil (2P_Q - 1)}$ and $X = \beta' \lceil \sqrt{n} \rceil (2P_Q - 1)$

Chernoff Bound:

$$\Pr\left(\left| \sum_{i=1}^{n} (X_i) - \mu n \right| \geq nt \right) \leq 2e^{\left[\frac{-2n^2t^2}{\sum_{i=1}^{n} (a_i - b_i)^2} \right]}$$

$$\Pr\left(\left| \sum_{i=1}^{\lceil \sqrt{n} \rceil} (X_i) - \mu \lceil \sqrt{n} \rceil \right| \geq \lceil \sqrt{n} \rceil t \right) \leq 2e^{\left[\frac{-2(\lceil \sqrt{n} \rceil)^2 t^2}{\sum_{i=1}^{\lceil \sqrt{n} \rceil} (a_i - b_i)^2} \right]}$$

$$\Pr\left(\left|\sum_{i=1}^{\lceil\sqrt{n}\,\rceil}(X_i) - \mu\lceil\sqrt{n}\,\rceil\right| \geq \lceil\sqrt{n}\,\rceil t\right) \leq 2e^{\left[\frac{-2(\lceil\sqrt{n}\,\rceil)^2t^2}{\lceil\sqrt{n}\,\rceil}\right]}$$

$$\Pr\left(\left|X - \mu\lceil\sqrt{n}\,\rceil\right| \geq \lceil\sqrt{n}\,\rceil t\right) \leq 2e^{\left[-2\lceil\sqrt{n}\,\rceil t^2\right]}$$

$$\Pr\left(\left|X - \mu\lceil\sqrt{n}\,\rceil\right| \geq \lceil\sqrt{n}\,\rceil t\right) \leq 2e^{\left[-2\lceil\sqrt{n}\,\rceil t^2\right]} = \delta$$

We can replace t with a function of a tunable variable δ if we wish, which allows us to set δ to a tolerable error probability and see how close β' is to β with probability at least $(1-\delta)$.

$$\delta = 2e^{\left[-2\lceil\sqrt{n}\,\rceil t^2\right]}$$

$$\delta/2 = e^{\left[-2\lceil\sqrt{n}\,\rceil t^2\right]}$$

$$ln(\delta/2) = -2\lceil\sqrt{n}\,\rceil t^2$$

$$-\frac{ln(\delta/2)}{2\lceil\sqrt{n}\,\rceil} = t^2$$

$$t = \sqrt{-\frac{ln(\delta/2)}{2\lceil\sqrt{n}\,\rceil}}$$

$$t = \sqrt{\frac{ln(2/\delta)}{2\lceil\sqrt{n}\,\rceil}}$$

Factoring t back into the Chernoff Bound:

$$\Pr\left(\left|X - \mu\lceil\sqrt{n}\,\rceil\right| \geq \lceil\sqrt{n}\,\rceil\sqrt{\frac{ln(2/\delta)}{2\lceil\sqrt{n}\,\rceil}}\right) \leq \delta$$

$$\Pr\left(\left|\beta'\lceil\sqrt{n}\,\rceil(2P_Q - 1) - \mu\lceil\sqrt{n}\,\rceil\right| \geq \lceil\sqrt{n}\,\rceil\sqrt{\frac{ln(2/\delta)}{2\lceil\sqrt{n}\,\rceil}}\right) \leq \delta$$

$$\Pr\left(\left|\lceil\sqrt{n}\,\rceil(\beta'(2P_Q - 1) - \mu)\right| \geq \lceil\sqrt{n}\,\rceil\sqrt{\frac{ln(2/\delta)}{2\lceil\sqrt{n}\,\rceil}}\right) \leq \delta$$

$$\Pr\left(\left|\lceil\sqrt{n}\,\rceil\right| \cdot |\beta'(2P_Q - 1) - \mu| \geq \lceil\sqrt{n}\,\rceil\sqrt{\frac{ln(2/\delta)}{2\lceil\sqrt{n}\,\rceil}}\right) \leq \delta$$

$$\Pr\left(\lceil\sqrt{n}\,\rceil \cdot |\beta'(2P_Q - 1) - \mu| \geq \lceil\sqrt{n}\,\rceil\sqrt{\frac{ln(2/\delta)}{2\lceil\sqrt{n}\,\rceil}}\right) \leq \delta, \text{ with } n > 0 \Rightarrow \lceil\sqrt{n}\,\rceil > 0$$

$$\Pr\left(|\beta'(2P_Q - 1) - \mu| \geq \sqrt{\frac{ln(2/\delta)}{2\lceil\sqrt{n}\,\rceil}}\right) \leq \delta$$

$$\Pr\left(|\beta'(2P_Q - 1) - (2\beta P_Q - \beta - P_Q + 1)| \geq \sqrt{\frac{ln(2/\delta)}{2\lceil\sqrt{n}\,\rceil}}\right) \leq \delta$$

$$\Pr\left(|\beta'(2P_Q - 1) - 2\beta P_Q + \beta + P_Q - 1| \geq \sqrt{\frac{ln(2/\delta)}{2\lceil\sqrt{n}\,\rceil}}\right) \leq \delta$$

$$\Pr\left(|\beta'(2P_Q - 1) - \beta(2P_Q - 1) + P_Q - 1| \geq \sqrt{\frac{ln(2/\delta)}{2\lceil\sqrt{n}\,\rceil}}\right) \leq \delta$$

$$\Pr\left(|(\beta' - \beta)(2P_Q - 1) + P_Q - 1| \geq \sqrt{\frac{ln(2/\delta)}{2\lceil\sqrt{n}\,\rceil}}\right) \leq \delta$$

$$\Pr\left(\left|\frac{1}{2P_Q - 1}\right| \cdot |(\beta' - \beta)(2P_Q - 1) + P_Q - 1| \geq \left|\frac{1}{2P_Q - 1}\right| \cdot \sqrt{\frac{ln(2/\delta)}{2\lceil\sqrt{n}\,\rceil}}\right) \leq \delta$$

$$\Pr\left(\left|\frac{(\beta'-\beta)(2P_Q-1)}{2P_Q-1}+\frac{P_Q-1}{2P_Q-1}\right|\geq\frac{\sqrt{\frac{ln(2/\delta)}{2\lceil\sqrt{n}\rceil}}}{|2P_Q-1|}\right)\leq\delta$$

$$\Pr\left(\left|(\beta'-\beta)+\frac{P_Q-1}{2P_Q-1}\right|\geq\frac{\sqrt{\frac{ln(2/\delta)}{2\lceil\sqrt{n}\rceil}}}{|2P_Q-1|}\right)\leq\delta$$

$$\Pr\left(\left|(\beta'-\beta)+\frac{P_Q-1}{2P_Q-1}\right|\geq\frac{\sqrt{\frac{ln(2/\delta)}{2\lceil\sqrt{n}\rceil}}}{2P_Q-1}\right)\leq\delta, \text{ with } 0.5<P_Q\leq 1$$

We note that the potential error in the oracle creates an extra term in the bound, $\frac{P_Q-1}{2P_Q-1}$ which is added into the $\beta'-\beta$ difference. This term helps counteract the oracle's error, and should be added to β' to get as close as possible to β. In the conclusions below, when we refer to β' we mean this adjusted value of $\beta'+\frac{P_Q-1}{2P_Q-1}$.

And finally, we consider whether this bound makes sense; δ works the way we would expect: as we bound the probability that β' deviates further from β, δ goes down. Moreover, as we increase the size of the list, our bound loosens. This makes intuitive sense, and so the bound above behaves correctly.

4 Experimentation

To test the algorithm, we implemented it in C++ using floating point numbers as values in the list. We ran tests at the 100GiB scale, using a list of $26843545600 = 25 \cdot 2^{30} \approx 2.6 \cdot 10^{10}$ floats. Two tests were performed on each of four lists, each list having a different β. All tests were performed with $\delta = 0.001$ and $P_Q > 0.97$.

The lists used to test the algorithm were generated using floats in the range $[0, 1000]$. The floats are stored in sorted order by letting each list element be the product of its index and $\frac{1000}{n}$. This gives a list of steadily increasing elements. Unsorted elements are introduced independently and uniformly at random, with each element (as it is created and added to the list) being unsorted with probability $1-\beta$. If an element is selected to be unsorted, a number is chosen uniformly at random from the set $\{1, 2, 3, 4, 5\}$, and either added to or subtracted from (each option with 50% probability) the element.

The oracle is created based on the observation that unsorted elements differ from their neighbors by at least 1 whereas sorted elements have a very small difference from their neigh-

β	P_Q	β'	Adjusted β'	Difference	Bound
0.850002	0.98	0.870463 0.87094	0.84963 0.850107	0.000372231 0.000104606	0.00501691
0.919999	0.99	0.934143 0.934205	0.923939 0.924001	0.00394017 0.00400246	0.00491452
0.949999	0.997	0.953868 0.953137	0.95085 0.950119	0.000850688 0.000119995	0.00484531
0.98	0.999	0.979944 0.979913	0.979944 0.979913	0.0000562668 0.0000867844	0.00481623

Table 1: Experimental Results

bors, so an element is regarded by the oracle as unsorted if the difference between it and both of its neighbors is .5 or more. This oracle may fail if it is asked to identify an unsorted element which is adjacent to another unsorted element, and then only if the adjacent unsorted element is within .5 difference with the inspected element. This probability is extremely low in our list, which contains unsorted elements uniformly at random. The probability grows, however, as β decreases but remains very low for all β we are interested in.

A brief clarification of the columns in Table 1. β is the exact sortedness of the list; P_Q is the oracle's probability of success which we chose arbitrarily based on the intuition that it should be extremely high for the values of β we are interested in. Neither β nor P_Q change between the two experiments on each list. The β' column refers to the actual output from the algorithm, whereas the *Adjusted* β' column stores $\beta' + \frac{P_Q - 1}{2P_Q - 1}$. *Difference* is $|\beta -$ Adjusted $\beta'|$, and *Bound* is the Chernoff bound which *Difference* should exceed with probability at most δ. *Bound* is also the same between the two tests on a certain list.

A few notes on selecting P_Q. We realize that our method of choosing P_Q for the above experiments was very informal; in general a more exact method may be advisable. However, the probability of the oracle failing should be incredibly small, so the exact value of P_Q should not be particularly important. In the above experiment with $\beta \approx 0.85$, after running the algorithm with a $P_Q = 0.99$ we observed that the β' value found was well beneath 0.9 and so we changed P_Q to be 0.98 based on that observation, knowing that a lower β spells a higher error rate for our oracle. It turns out that the 0.98 probability gave a β' somewhat closer to β than 0.99 did, although in general β will not be known for comparison. It should be noted that selecting P_Q will affect the accuracy of the bounds, and if P_Q does not accurately reflect the performance of the oracle, the final estimate for β may not be within these bounds.

As can be seen from the table, the algorithm performs as expected and gives good results. By using an oracle which works with very high probability, and adjusting for the error it introduces when it fails, we can get a very close approximation of β in sub-linear time.

5 Conclusion

We have shown that our algorithm for estimating the sortedness of an α-approximately-sorted gives a result which is very close to correct in sub-linear time, assuming a constant time oracle can be constructed which identifies elements as being sorted or unsorted within the list.

The oracle must be correct with a high probability ($P_Q > 0.97$). The Chernoff bound begins to deteriorate when P_Q is not very close to 1, which is expected and backed up in our experiments; we make this restriction at the cost of a much simpler Chernoff bound. Furthermore, our analysis breaks down if $\beta' > 2P_Q - 1$; therefore, the algorithm requires that the oracle be nearly infallible, giving a correct result with high probability. This is not an unreasonable requirement and the authors predict that such an oracle can reasonably be designed and implemented.

A final consideration is the problem of whether a given list is at least α-approximately sorted. The algorithm simply returns an estimation for β, which is the actual percent-sortedness of the list. Therefore, our estimated value, β', has a very high probability of being close to β: β' will be within $\frac{\sqrt{\frac{ln(2/\delta)}{2\lceil\sqrt{n}\rceil}}}{2P_Q - 1}$ of β with a very high probability. If an α is given within this range of β, our algorithm cannot give a definitive answer as to whether the list is α-approximately-sorted.

If such a situation exists where α is within this *"danger"* interval, then the reason for running this randomized algorithm should be examined. The purpose of this algorithm is to estimate β' with high probability to be within a certain interval from β; its purpose is not to give the exact value of β, which would be infeasible for a Monte Carlo randomized algorithm. As long as α is sufficiently distant from β and β', this algorithm can give a reliably correct answer for the decision problem of an α-approximately-sorted list. If α is too close to β, this algorithm (and probably any other Monte Carlo algorithm) will not give a reliable answer.