Tolerant Algorithms

Seminar on Advanced Algorithms and Data Structures - Student Report

Gabriel Giamarchi

October 26, 2018

1 Introduction

We all know programming is hard.
Programming without bugs is even harder.
What about programming without bugs on a buggy machine ?
We will attack this unsettling question by discussing the results obtained in [7] (inspired by legal contracts, let's refer to [7] as *the paper* from now on).

Approaching the previous question can be done in many ways; the first thing to do is to define which model of errors we will work in, as well as what question we really want to ask in such a model. Then, we will see some of the so called "tolerant algorithms" presented in [7] as the goal of the paper is to offer concrete algorithmic solutions. I will notably skip covering a tolerant sorting algorithm both for conciseness and because better results have been obtained recently (see [6] for an algorithm in expected time $O(n \log(n))$!).

2 The Model

2.1 Model description

The model used in the paper is made of two important parts.
Firstly, it uses an abstraction called an *oracle*. An *oracle* is able to answer the queries that an algorithm needs to perform on its input data. For instance, this could simply be comparisons (for a sorting algorithm). In this model, we assume that **the only access algorithms have to their input data is via queries to an oracle**.

Secondly, these queries are *persistently unreliable*; This means that any query q can be answered wrongly by the oracle with a certain probability p_q (what "wrong" means is left to the specific problem we are studying withing this model). Additionally, repeating queries will persistently give the same answer: we cannot simply repeat queries and take the mode to diminish error probability. We assume all probabilities p_q are bounded by some probability p (in what follows, p will consistently refer to the bound on oracle query failure probability).

This model might seem a bit surprising at first; Surely any computer which offers *persistently* wrong primitives (oracle queries) is a very flawed machine and should not be used for computation ! However, one soon realizes that this model is very appropriate in many cases. In my opinion, it is one of the main things to take away from the paper (see section 2.3).

For instance, we all remember the havoc that can be wrought by floating point arithmetic; wouldn't it be nice to use algorithms tolerant to faulty arithmetic and be safer from horrible

cancellation issues ? (this is actually what is done in numerical algorithms; moving terms around to prevent the "noisy queries" from affecting us. In a way, this paper does the same for other algorithms)
Another very interesting use case of the model lies in problems making use of expensive (thus not repeatable) and unreliable comparisons. This could arises if we organize competitions, or ask experts to rank items.
A last example is: It could be much easier to make an efficient system if we allow some errors. For instance, we could imagine getting rid of some cache consistency mechanisms (do not mention this to a systems person) to get a much simpler and faster machine ! Of course, very tolerant algorithm would be needed to correctly compute results on such a computer.

All of the previous examples are only interesting if we can indeed compute correct results despite the presence of persistent errors. Thus, the main question is: Can we solve problems of this form, and if yes, with what probability ?

2.2 Remark

The question in the last paragraph is formalized in the paper as "Can we still approximately solve our task with probability $1 - f(p)$ for a function f that goes to 0 as p goes to 0 ?". This seems a bit surprising, because for $p = 0$ clearly the probability of success with any correct algorithm should be 1, and continuity of the success probability as a whole seems intuitive. In fact one could argue as follows:

Let ALG be any *correct* algorithm (that works with probability 1 given $p = 0$) we could map each computation of ALG to the sequence $(q_i)_{i\in\{1,2,\dots,N\}}$ of queries posed to the oracle. Clearly if every query returns the correct result we are back in the case where no errors are present and as such:

$$P[\text{ALG gives the correct answer}] \geq (1-p)^N \approx 1 - Np \quad \text{for small } p \text{ by Taylor expansion}$$

Since most of the time the number of queries used also corresponds to the quantity used to quantify the runtime of the algorithm (ex: number of comparisons), by writing this runtime as some function $R(n)$ where n is the size of the input we get:

$$P[\text{ALG gives the correct answer}] \approx 1 - R(n)p$$

Great ! We found a function $f(p)$ that goes to zero as p goes to zero and solved the whole thing !
Of course, the small print is that the function f has to be **independent of n** !

The (even better) case where f is not independent of n but rather **also** goes to zero as n goes to infinity is interesting too because it represents a class of Tolerant Algorithms that give correct answers *with high probability*. This is not what is studied in this paper but we'll make a short remark about it later on.

We will now look at a few different problems following this model and try to show Tolerant Algorithms and suitable function $f(p)$ for them.

2.3 Main Takeaways

Before jumping in the description and analysis of some Tolerant Algorithms, I want to emphasise some key ideas to take away from this paper.

The model: Although the idea of studying algorithms that can succeed in performing their task with noisy data, or even noisy primitives is not a new one (see [4] for instance), the model presented in [2] (the model presented at the beginning) has sparked further research and merits attention. I think the model itself is quite flexible because of the abstraction offered by the concept of an oracle. It is possible to convert a problem in which an algorithm has access to noisy data into a problem where an algorithm uses queries to a flawed oracle with access to perfect data.

Thus the model offers a broad, flexible framework in which unreliability of computing primitives, as well as data, can be studied.

A simple principle: The second thing that I tried to do was to identify a "design principle" that is useful for the conception of tolerant algorithms. We'll try to highlight this red thread as most as possible through some of the algorithms presented.

The idea is: **Get rid of what you do not want**.
This can simply be seen as the dual of "getting what you want", but we'll see in the following sections how it can be useful for algorithmic design and analysis.

As a first example of this principle, let's consider the tolerant maximum finding algorithm presented in the paper. The first algorithm one probably imagines for finding the maximum of n numbers is a very good one: Simply go through each number and keep in memory the *best maximum seen so far*. I would categorize this as *getting what you want*; We want a maximum and iterate through the numbers in search of better versions of what we want.

The tolerant algorithm which follows will instead try to *get rid of as many non-maximum elements as possible*, and from this idea gets resilience to unreliable answers of the oracle.

3 Maximum of n numbers

Given an unordered set of n distinct numbers $a_1, a_2, \ldots, a_n$, find the maximum using an oracle able to answer any query regarding the comparisons between a_i and a_j.

Let's see the algorithm presented in the paper (Algorithm (1)) and the accompanying theorem:

Theorem 1. *For any constant* $1/2 > \lambda > 0$ *there exists a constant* $C = C(\lambda)$ *such that for any* $p \leq 1/64$ *the algorithm* LinearMax *succeeds with probability at least* $1 - C \cdot p^{\frac{1}{2}-\lambda}$

Algorithm (1) looks absolutely incomprehensible, but by cleaning things a bit (getting rid of what we don't want !) we can begin to get a better view of what the main idea is. Let's abstract a few things:

- We define $\kappa < 1$ as the factor present throughout the algorithm ($\frac{15}{16}$ in the previous algorithm).
- We move the core of the algorithm to a separate procedure: KnockoutRound
- We also write the ideas behind loop conditions in plain text

This results in the sketch: "Algorithm" 2

Algorithm 1 Returns a probable maximum of the set M

procedure LINEARMAX
  $n = |M|$
  while $|M| \geq \max\{n^{1-\lambda}, 100/\sqrt{p}\}$ **do**
    $j \leftarrow 0$
    Select $i \in \mathbb{N}$ such that $n(\frac{15}{16})^{i-1} \geq |M| > n(\frac{15}{16})^{i}$
    while $j < 100 \cdot i \cdot \log(1/p)$ and $|M| > n \cdot (\frac{15}{16})^{i}$ **do**
      Select a set S of $s_i = 4i$ elements from M uniformly at random
      Remove all elements in S from M
      Compare all elements from S with M
      Let M be the list of elements larger than at least $\frac{3}{4} \cdot s_i$ elements of S
      $j \leftarrow j + 1$
    if $|M| > n \cdot \frac{15}{16}^{i}$ **then**
      Output "error" and exit
  while $|M| \geq 100/\sqrt{p}$ **do**
    $j \leftarrow 0$
    Select i' such that $n(\frac{16}{15})^{i'} \geq |M| > n(\frac{16}{15})^{i'-1}$
    $i = i' - \log_{16/15} \frac{100}{\sqrt{p}} + 1$
    while $j < 100 \cdot i \cdot \log(1/p)$ and $|M| > (\frac{16}{15})^{i-1}$ **do**
      Select a set S of $s_i = 4i$ elements from M uniformly at random
      Remove all elements in S from M
      Compare all elements from S with M
      Let M be the list of elements larger than at least $\frac{3}{4} \cdot s_i$ elements of S
      $j \leftarrow j + 1$
    if $|M| > n \cdot (\frac{15}{16})^{i-1}$ **then**
      Output "error" and exit
  Finish by a regular linear maximum finding algorithm on the remaining elements of M

3.1 Conceptual sketch of Algorithm 1

Algorithm 2 Conceptual version of algorithm 1

procedure KnockoutRound(M,i)
  Select a set S of $s_i = 4i$ elements from M uniformly at random
  Remove all elements in S from M
  Compare all elements from S with M
  Let M be the list of elements larger than at least $\frac{3}{4} \cdot s_i$ elements of S
procedure LinearMax
  $n = |M|$
  while $|M| \geq n^{1-\lambda}$ **do** ▷ Phase 1
    $j \leftarrow 0$
    Select $i \in \mathbb{N}$ such that $n(\kappa)^{i-1} \geq |M| > n\kappa^i$
    while $j < 100 \cdot i \cdot \log(1/p)$ and $|M| > n \cdot (\kappa)^i$ **do**
      KnockoutRound(M,i)
      $j \leftarrow j + 1$
    if $|M|$ has not been reduced by at least a factor κ **then**
      Output "error" and exit
  while $|M| \geq 100/\sqrt{p}$ **do** ▷ Phase 2
    $j \leftarrow 0$
    $i \leftarrow \lceil \log_{\frac{1}{\kappa}}(\frac{|M|\sqrt{p}}{100}) \rceil + 1$
    while $j < 100 \cdot i \cdot \log(1/p)$ and $\kappa^i |M| > \kappa$ **do**
      KnockoutRound(M,i)
      $j \leftarrow j + 1$
    if $|M|$ has not been reduced by at least a factor κ **then**
      Output "error" and exit
  ▷ Phase 3
  Finish by a regular linear maximum finding algorithm on the remaining elements of M

This is somewhat more manageable now ! We notice the following things:

1. The main procedure is "knockout round", in which we select a sampling set S from the elements of M, and remove elements which *probably* **are not** the minimum. We hope to reduce the size of $|M|$ in knockout round by some factor $\kappa < 1$.

2. There are 3 phases; The first one when $|M|$ is arbitrarily big, the second when $|M|$ is in $O(n^{1-\lambda})$ and the third one when $|M|$ is in $O(1)$.

3. In the first phase, the size of the sampling set increases (as $|M|$ decreases i increases). In the second phase the size of the sampling set decreases.

4. The "if" statements are there to simplify analysis; If we cannot reduce the size of $|M|$ after many knockout rounds we simply fail and exit. This way, the runtime analysis is simplified but the error probability is somewhat increased.

3.2 Analysis

Let's try to see a rough overview of the analysis of LinearMax, as it highlights some important points about mysterious constants in the algorithm and shows where the main "tolerance" of the algorithm lies (as you might have guessed, it's in KnockoutRound). For those who would like to directly skip ahead, I recommend to go to section 3.2.1

We will begin by seeing why the runtime of LinearMax is in $O(n)$. Then, we'll see how error probabilities of each phase can be bounded.

Essentially phase 1 (where $|M|$ can be arbitrarily big) will have an error probability bounded by $C \cdot p^{\frac{1}{2}-\lambda}$ for some big constant C.
In the second phase, the error probability will be bounded by $C \cdot p^{\frac{1}{2}}$.
By the following remark, phase 3 fits within these bounds too.

Preliminary remark: Phase 3 (regular maximum-finding) does not pose a problem with the error bounds above. In phase 3, $|M| < \frac{100}{\sqrt{p}}$ and any regular linear time maximum finding algorithm will use $O(|M|)$ oracle queries. By the thought experiment (see section 2.2) shown at the beginning we have an error probability bounded by the number of queries used times p which is smaller than $\frac{100p}{\sqrt{p}} \in O(p^{1/2})$. This explains why we need to bound $|M|$ by something in $\frac{1}{\sqrt{p}}$!

Runtime analysis

Definition: We call a loop iteration of phase 1 or 2 a *stage* of phase 1 or 2. Note that in a particular stage the value of i is fixed for all inner loop iterations.

The runtime analysis is based on the following lemma, which bounds the number of calls of KnockoutRound we can make in each stage of phase 1 or 2. Intuitively, we say that KnockoutRound is *expected* to remove many elements of M, thus the probability that we are not able to reduce the size of $|M|$ by at least a factor κ after many iterations is very low.

Lemma 1. *The probability that in a fixed stage i of Phase 1, after more than $100 \cdot k$ iterations of the body of the inner loop, $|M| > n \cdot (\frac{15}{16})^i$ holds is at most $(\frac{1}{2})^k$. Similarly, the probability that in a fixed stage i of Phase 2, after more than $100 \cdot k$ iterations of the body of the inner loop, $|M| > (\frac{16}{15})^{i'-1}$ holds is at most $(\frac{1}{2})^k$ (note that i' refers to algorithm 1).*

The proof uses the assumption $p < 1/64$, but if $p \geq 1/64$ we could adapt the constants in the algorithm (factor κ for instance) to get a similar result.

By using this lemma, the total number of inner loops (divided by 100) is bounded by a geometric distribution of parameter $\frac{1}{2}$. Therefore the total **expected** number of loops (calls to KnockoutRound) in each stage is in $O(1)$.

Furthermore, in each stage of phase 1 or 2 the total number of oracle queries used is at most $|M| \cdot 4i$. By using bounds on the values of $|M|$ and i in each phase of the algorithm and linearity of expectation it is possible to conclude that the first 2 phases pose an **expected** number of queries to the oracle in $O(n)$. Of course, this quantity in phase 3 is trivially in $O(n)$ (this holds exactly, by opposition to "in expectation").

Taken together, these remarks imply the following lemma:

Lemma 2. *The asymptotic **expected** run time complexity (measured in number of comparisons) of* LinearMax *is in $O(n)$*

Error probability analysis

The more interesting question is the analysis of the error probability. At any *stage* of phase 1 or 2 there are 3 possible sources or errors:

1. The maximum is in the sample set S (which we always remove from M)
2. The maximum is (due to faulty oracle queries) reported to be smaller than $\frac{1}{4}$ of the elements of S.
3. One of the two error triggering if-statements are entered.

Analysis of the third cause of error basically follows from using lemma 1 again. This gives an error probability in $O(p) \subset O(\sqrt{p})$ (since $p < 1$). The intuition behind this is: The error triggering if-statements happen if we somehow fail to significantly reduce the size of M. Since we

are choosing sampling sets uniformly at random, we *expect* to remove a significant portion of M in each KnockoutRound call. Thus the overall probability that the size of M remains big is quite small.

Analysis of the first and second points is more interesting because they both teach us something about the ideas behind LinearMax.

For the first point, the quantity to study is $|S|/|M|$, namely the probability of (wrongly) selecting the maximum. The reason why this quantity stays low in both phases (and the reason why we are able to achieve good bounds of the probability of this kind of error happening) is:

- In the first phase, the sampling set S keeps a relatively small size (in $O(\log(n))$) compared to the size of M which is in $n^{1-\lambda}$.
- In the second phase, while the size of M is much lower, we are very careful to pick i carefully (now decreasing with $|M|$) to ensure that $|S|/|M|$ stays low. This is why we choose the value of i we do on line 18 of algorithm (2).

Remark This first kind of error explains the need for the second phase. We have to be choose less and less elements near the end to avoid accidentally sampling the maximum. One could ask: "Why remove the sampling set from M ?" Firstly, we never use the size reduction of M due to removing S in the analysis, which means that we don't *need* to remove S (this comes from the fact that $|S| \in O(\log(n)) << |M|$). Thus, it gives us the nice property that comparisons in different stages are independent "for free".

3.2.1 The important bit

The second point is the most important, since this is where we consider errors due to faulty oracle queries. As expected, the *tolerance* of this algorithm comes from the fact that it's much harder to mistakenly discard the maximum in a KnockoutRound (if we don't sample it), since many queries would need to be wrong. In fact, at least, $\frac{1}{4}$ of the comparisons would need to be wrong for this to happen. By the following lemma, the probability of this happening is small:

Lemma 3. *Let $1 \geq p \geq 0$ be the failure probability of a comparison. Let $k > 0$ be a multiple of 4. The probability that at least $k/4$ out of k comparisons fail is at most $(4p)^{\frac{k}{4}}$.*

Since in each stage of phase 1 or 2 we compare with $k = 4i$ elements, i (in both phases) iterates between 1 and $O(\log(n))$ and we perform a maximum of $100i\log(1/p)$ iterations, we get that the probability of this kind of error happening is at most (the paper shows this bound in one line but I don't think it's trivial):

$$
\begin{aligned}
&\sum_{i=1}^{O(\log(n))} 100i\log(1/p)\cdot(4p)^i \\
&\leq 100\log(1/p)\sum_{i=1}^{\infty} i\cdot(4p)^i \qquad \text{(Known series, essentially the derivative of a geom. series)} \\
&= C\log(1/p)\frac{4p}{(1-4p)^2} \quad \text{(Using the closed form of the series above)} \\
&\leq \bar{C}p\log(1/p) \quad \text{(By using } p < 1/64 \text{ we bound } (p-1)^{-1}\text{)} \\
&\leq \tilde{C}\cdot\sqrt{p} \quad \text{(see below for explanation)}
\end{aligned}
$$

The last bound can be observed in the following way:

$$\lim_{p\to 0} p\log(1/p) = \lim_{p\to 0} \frac{\log(1/p)}{(1/p)}$$
$$= \lim_{n\to\infty} \frac{\log(n)}{n} \text{ with } n := (1/p)$$

Furthermore, since $\forall \epsilon > 0,\ \log(n) \in O(n^\epsilon)$ the function of n: $\frac{\log(n)}{n}$ is in $O(\frac{n^\epsilon}{n})$ which is $O(p^{1-\epsilon})$ with the definition $n = (1/p)$ above. By choosing $\epsilon = 1/2$ we get the desired inequality. Of course, this result (in the limit p goes to zero) only implies the bound in a neighborhood of $p = 0$ by continuity of both functions. However, one can choose the constant $\tilde{C}$ big enough to expand this neighborhood until it includes the values of p we allow.

3.3 Remark

Summarizing the previous results, the surprising thing is that we get an *expected* runtime in $O(n)$ (asymptotically the same as the best possible algorithm), but an error probability in $1 - C \cdot p^{\frac{1}{2}-\lambda} = 1 - f(p)$ which is asymptotically better than the worst case estimate of section 2.2. Essentially, without any asymptotic hit on the runtime, we get some fault tolerance "for free".

This is not just isolated to this problem and model in particular, for instance using the same model, a tolerant algorithm is studied in [5] for the problem of finding the longest increasing subsequence, and in particular the number of comparisons (oracle queries) used by the algorithm reaches (asymptotically) the lower bound of any $O(\log(n))$ approximation algorithm (since with errors, the problem cannot be solved exactly). This lower-bound is the same, even **in the absence of errors !**.

In a different model, namely with noisy queries that are not *persistently* unreliable anymore, a more general problem ("select one of the smallest k elements") is studied in [9]. An algorithm which succeeds *with high probability* (see section 2.2) is presented. Again, the surprising result of obtaining some fault tolerance at no significant asymptotic complexity cost is encountered in some cases.

These results raise the following interesting question: "Does there exist a consistent method, or underlying principles, that would allow us to convert any algorithm into a more *tolerant* version of itself ?". Of course, this is vague and obtaining such broad results seem quite hard, but the next section shows that one can get interesting results by applying elementary probability theory, in some cases. Furthermore, some results have already been obtained, as in [1] for a very restricted instance of the previous question.

4 Link between Runtime and Tolerance

The purpose of this section is to show, in the case of Tolerant Algorithms, a trade-off between runtime and success probability. This section comes from R. Penninger's Ph.D. thesis and not the article directly but I wanted to show it to illustrate that simple probabilistic ideas (here Markov's inequality) can be sometimes used to prove non trivial results.

Given that n represents the size of the input, $T(n)$ is an upper bound on the *expected* runtime of some Tolerant Algorithm ALG, we have the following theorem:

Theorem 2. *Given a tolerant algorithm* ALG *that computes the correct result with probability* $1 - f(n,p)$ *and a positive function* $g(n,p)$*, we can construct a tolerant algorithm* ALG* *that solves the same computational task as algorithm* ALG*, *with probability* $1 - f(n,p) - g(n,p)$*. The worst-case running time of* ALG* *is* $T(n)/g(n,p)$*.*

Proof. We simply define ALG* by executing ALG and stopping with an error if we go above the time bound of $T(n)/g(n,p)$. The time-bound then trivially holds, and we are left with calculating the success probability.

Define events E_1 := "ALG fails on the input" and E_2 := "ALG needs more than $T(n)/g(n,p)$". Clearly $P[E_1] \leq f(n,p)$.
Now, define the random variable T as the runtime of ALG. By Markov's inequality (the only place where something **really** happens in the proof) we have:

$$\begin{aligned} P[E_2] &\leq P[T \geq T(n)/g(n,p)] \\ &\leq P[T \geq \mathbb{E}[T]/g(n,p)] \quad \text{since } T(n) \text{ is an upper bound on } \mathbb{E}(T) \\ &\leq \frac{\mathbb{E}[T]}{\mathbb{E}[T]/g(n,p)} \quad \text{by Markov's inequality} \\ &= g(n,p) \end{aligned}$$

Finally $P[\text{ALG}^* \text{ fails to compute correct result}] \leq P[E_1 \cup E_2] \leq P[E_1] + P[E_2] \leq f(n,p) + g(n,p)$ which gives our desired result.

□

5 Tolerant Search

Let us turn our attention to a different problem now, namely identifying the position at which we should insert a key b in an **accurately sorted** sequence of keys $a_1 < a_2 < \cdots < a_n < a_{n+1} = \infty$. We suppose that b is different from all a_i, and we are provided with an oracle giving us noisy answers to the questions $b < a_i$. We write the oracle answers as a relation $<_E$. Our model is:

$$\forall j \in \{1, \ldots, n\} \quad b <_E a_j = \begin{cases} \neg(b < a_j) \text{ with probability } p \\ b < a_j \text{ with probability } 1-p \end{cases}$$

We also assume that $p < 1/2$, otherwise we are better off not using the oracle at all !
Thinking about this problem is greatly simplified if we begin with suitable *definitions*. To get them we are going to apply the "get rid of what you don't want" principle to the problem description itself. What we **do not want** is for many elements a_i to be in conflict (via the unreliable queries to the oracle) with our estimation of where we should insert b. We define the following two functions:

$$\begin{aligned} &\text{rank}(b) := \text{smallest } i \text{ such that } b < a_i \\ &\text{conf}(j) := |\{0 < i < j \mid a_i >_E b\}| + |\{j > i > n+1 \mid a_i <_E b\}| \\ &\qquad\qquad \text{(the number of } a_i\text{s that disagree with assuming "}\mathit{rank}(b) = j\text{")} \end{aligned}$$

We have the following property:

$$\text{conf}(j) = \begin{cases} \text{conf}(j-1) + 1 \text{ if } b <_E a_{j-1} \\ \text{conf}(j-1) - 1 \text{ if } b >_E a_{j-1} \end{cases} \tag{1}$$

This way, the conf function tells us how high the thing which we **do not want** is, and the only thing left to do is to *get rid of* (minimize) it.

5.1 Algorithm

This suggests that we can build a simple algorithm that builds the vector conf(j) in $O(n)$ time and then outputs the position j_0 which minimizes conf(j). For instance:

Algorithm 3 Gives an approximation of rank(b)

```
1: procedure SEARCH
2:     ∀i = 1, ..., n + 1   conf(i) := 0
3:     conf(1) := |{0 < i < n + 1 | a_i <_E b}|        ▷ This can be done in time O(n)
4:     for j ∈ {2, ..., n + 1} do
5:         if b <_E a_{j−1} then
6:             conf(j) = conf(j − 1) + 1
7:         else
8:             conf(j) = conf(j − 1) − 1
9:     RETURN the first position j_0 which minimizes conf(j)    ▷ This can be done in time O(n)
```

Intuitively the algorithm 3 returns a position j_0 that maximizes the probability that $rank(j) = j_0$ (a maximum likelihood position for b). Of course it's an important thing to prove for the analysis, and this is done in [2] (look inside definition 2 on pages 2-3 with an argument for the sorting problem that can be adapted to this case)
The interesting question is: What is the probability that $j_0 = \text{rank}(b)$; that is: What is the probability that we report the correct result ?

5.2 Analysis via random walk argument

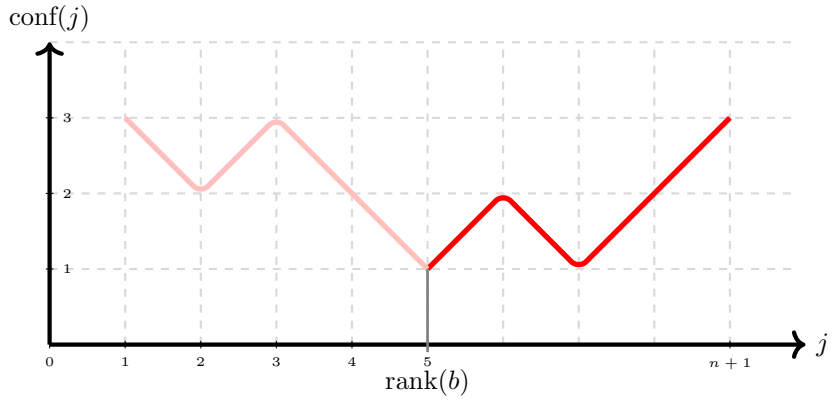
This is the point at which we realize the definition of conf(j) is quite clever, not only does it suggest the algorithm but also the means of its analysis. Namely, the sequence conf(j), $j \geq \text{rank}(b)$ and equation 1 can be seen as a *random walk* which increases with a probability $q := (1-p)$ (because b is smaller than all a_i with $j > \text{rank}(b)$) and decreases with probability p !

Thus the probability that conf(j_0) is smaller than all values conf(j), $j > \text{rank}(b)$ is the probability that we increase the value of a random walk by 1 in the first step times the probability that a random walk starting at 1 never decreases below 1, which is qP["A random walk going down with probability p and up with probability q starting at 1 never decreases below 1"].
In the next section we show that this is equal to $q(1 - \frac{p}{q}) = (1 - 2p)$.

We can make a similar argument for the values of conf(j), $j < \text{rank}(b)$, which gives us:

$$P[j_0 \text{ (position reported by Alg. 3)} = \text{rank}(b)] \geq (1-2p)^2$$

Here is a little figure to help with visualization.



5.3 Studying random walks with Generating Functions

What is the probability that a random walk which goes up with probability q and down with probability p starting at 1 always stays ≥ 1 ?

I suggest a different proof than the one presented in R. Penninger's Ph.D. thesis since he uses induction which gives little insight on how to obtain the formula in the first place. I will use generating functions, because they give correct results, which we can then fully prove (by induction for instance !), and we'll even get interesting remarks for "free" !

Let's begin by replacing our question with the equivalent question that a similar random walk starting at 0 always stays ≥ 0. We formalize our random walk as a sum $S_n := \sum_{i=1}^{n} X_i$ of random variables of the form:

$$X_j := \begin{cases} +1 \text{ if we go up on step j} \\ -1 \text{ if we go down on step j} \end{cases}$$

Let's now define a random variable N which takes the value of the smallest i such that $S_i = -1$ (first passage below zero) and which has value ∞ if this i does not exist. Additionally, let $p_i := P[N = i]$. To study the sequence p_i we build a so called "generating function" which can be seen as replacing the infinite dimensional vector p_i by another one, namely the generating function (For a detailed overview of generating functions I recommend the corresponding chapter in [8]):

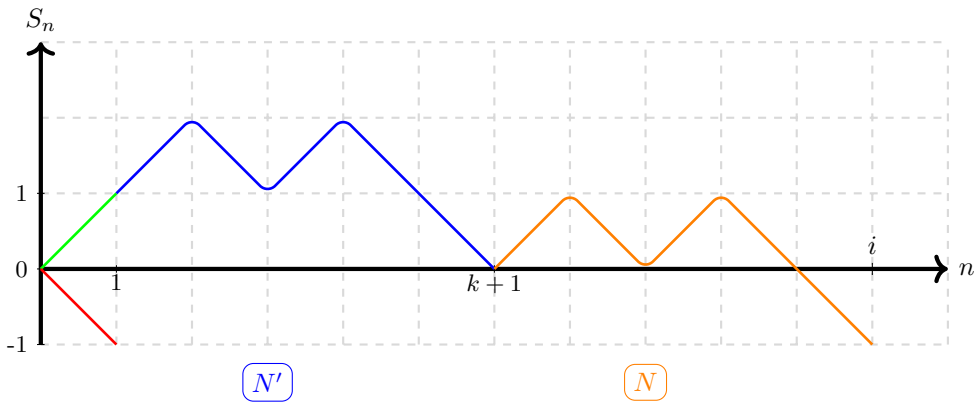
$$G(z) := \sum_{n=0}^{\infty} p_n z^n$$

Now we'll get a recursion formula for the p_is and see how that can be translated to an equation for $G(z)$. We perform a case distinction on the first step of the random walk to find $P[N = i]$.

Since we begin at zero, clearly $p_0 = 0$.
If the first step X_1 of our random walk is -1 then $N = 1$, and the reverse also holds.
If the first step is $+1$ then to get to -1 we need to go first to 0 (in some number of steps k) and then from 0 to -1 in $i - k - 1$ steps. The key observation is that since the random walk is a memoryless process the variable:

$$N' := \text{ smallest } k \text{ such that } S_k = 0 \text{ starting at } S_0 = 1$$

has the same distribution as N !
The picture is:



This gives us that:

$$P[X = 1] = p_1 = p \tag{2}$$

$$P[X = i] = p_i = p \cdot 0 + q \cdot \sum_{k=0}^{i-1} P[N' = k] \cdot P[N = i - 1 - k] = q \cdot \sum_{k=0}^{i-1} p_k \cdot p_{i-1-k}$$

The convolution product in the previous equation looks scary, but it dramatically easy to express with our generating function ! Since:

$$\begin{aligned} G(z) &= p_0 + p_1 z + p_2 z^2 + \dots \\ \Rightarrow G(z) \cdot G(z) &= (p_0 p_0) + (p_0 p_1 + p_1 p_0) z + (p_0 p_2 + p_1 p_1 + p_2 p_0) z^2 + \dots \\ \Rightarrow G(z)^2 &= \sum_{i=0}^{\infty} z^i \sum_{k=0}^{i} p_k p_{i-k} \\ \Rightarrow z G(z)^2 &= \sum_{j=1}^{\infty} z^j \sum_{k=0}^{j-1} p_k p_{j-1-k} \end{aligned}$$

Using this and the equations 2 we find the following equation for $G(z)$:

$$G(z) = pz + qzG(z)^2 \tag{3}$$

Solving 3 gives us:

$$G(z) = \frac{1 - \sqrt{1 - 4pqz^2}}{2zq} \tag{4}$$

Note: We needed to take the solution with minus because we must have $G(0) = p_0 = 0$.

This generating function is now a powerful tool to study the whole sequence p_i; for instance we can get any coefficient we want by expanding into a series or (numerically for instance) by contour integrals in the complex plane.

For our purpose, we are interested in $1 - \sum_{i=0}^{\infty} p_i$. This represents the probability of the random walk staying semipositive (the original question of this section !). This is simply $1 - G(1)$ which we calculate as being (using $p + q = 1 \Rightarrow 1 - 4pq = (p - q)^2$):

$$G(1) = \begin{cases} 0 \text{ when } p \geq q \\ 1 - p/q \text{ when } p < q \end{cases} \tag{5}$$

We already have the result used in the paper (we are in the case $p < q$) but we get another interesting result known as **Gambler's Ruin** in the case $p \geq q$. Namely, even if $p = q = 0.5$ a random walk has probability **zero** of always staying semipositive; which means that a gambler playing with a finite amount against an infinitely rich casino for an infinite amount of time, will eventually loose all her/his money, **even if the game is fair** !

Moral of the story, don't gamble, or at least calculate the generating functions before doing so.

6 Linear Programming

The last topic I want to mention is *Linear Programming* (we'll abbreviate it as LP). The paper finishes by showing a tolerant algorithm that solves LP-type problems in two dimensions in time $O(n \log(n))$ (*with high probability*) and has correctness probability $1 - f(p)$. The algorithm and its analysis are too long to present here but I will try to show some interesting facts about it.

Firstly, LP problems are very interesting because they provide a common language in which **a lot** of questions can be expressed. If we were able to build tolerant algorithms for n-dimensional LP-problems this would be a big step forward in answering the question at the end of section 3.3. Furthermore, the unreliable queries to the oracle in this problem are faulty geometrical primitives (deciding on which side of a line a point is for instance). This shows a different type of query than comparisons. We all know programming such primitives which are not affected by numerical issues in tough corner cases is quite difficult, hence the interest of tolerant algorithms for such queries.

I will shortly hint at the way the algorithm is designed, because it ties in (pun intended) with the thread of "getting rid of unhelpful things". The tolerant algorithm presented is an adaptation of an existing LP algorithm called **Megiddo's algorithm**. Let's see the main idea behind Megiddo's algorithm.

6.1 Megiddo's algorithm

We will talk about a 2D version of Megiddo's algorithm to simplify the explanation and because it is the case studied in the paper, but there exist d-dimensional versions based on the same important idea.

As a reminder, a linear program has the form:

$$\begin{cases} \text{maximize} & c^T x \\ \text{with constraints} & Ax \leq b \end{cases} \tag{6}$$

With $x, c \in \mathbb{R}^d, b \in \mathbb{R}^n, A \in \mathbb{R}^{n \times d}$.
We can always rotate the whole system (in linear time in n) to align the *objective function* (the function to maximize) with any direction (an axis for instance). Each row of the matrix A is an element of the dual space of $\mathbb{R}^d$ and can be seen (together with the corresponding element of b) as restricting the allowed subset of $\mathbb{R}^d$ (called *feasible region*) to be one one side of a hyperplane. Fortunately for us, in 2D hyperplanes are simply lines, which facilitates visualization !

In the following picture, the feasible region has been coloured and we try to minimize the x_1-coordinate.

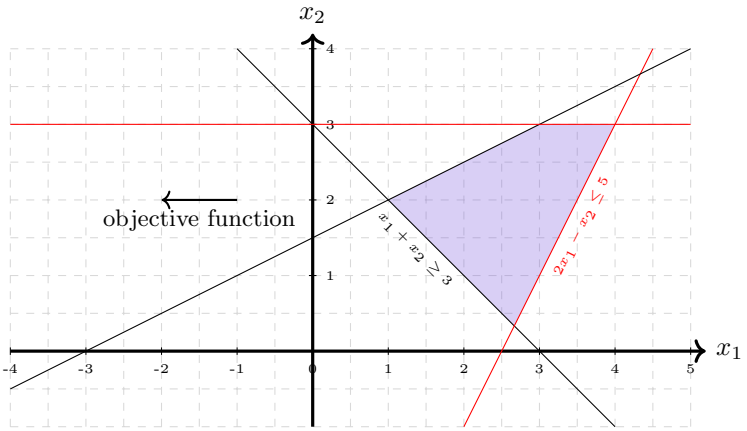


Figure adapted from https://tex.stackexchange.com/questions/75933/how-to-draw-the-region-of-inequality, Tom Bombadil's answer

We see the following things:

1. The optimal point "gets stuck" in a corner of the feasible region, and thus touches some of the restricting hyperplanes (making the corresponding constraints *tight*). In effect this renders the other constraints irrelevant (drawn in red above).

2. Since we are in 2D we need only to identify 2 *tight* constraints (let's ignore degenerate cases) to identify a solution !

The idea behind Megiddo's algorithms is to **discard the irrelevant constraints**, which is done (in higher dimensions than 2) by recursively solving linear programs or by looking at the constraints equations (in 2D). (I recommend the following survey of some LP-problem solving methods for an overview of the algorithm: [3])

Again, we return to the principle of section 2.3. I think this is one reason why Megiddo's algorithm specifically was suited to be converted to a tolerant algorithm.

The difficulty in adapting this algorithm to be tolerant lies in still being able to discard irrelevant constraints without touching those which will be *tight*. Again, a sampling set technique similar to the one used in LinearMax is used, but the analysis is much more complicated.

6.2 Remarks

Actually, not only is the sampling set technique reused, but the algorithm LinearMax is also slightly modified and used as a subroutine. This offers the perspective that we might be able to build new tolerant algorithms in a modular fashion.

Concerning runtimes, the tolerant LP algorithm has runtime $O(n \log(n))$ which is slightly worst than Megiddo's algorithm (in time complexity $O(n)$ with the general case of d dimensions being $O(2^{2^d} n)$). Would it be possible to achieve a runtime of $O(n)$ *with* tolerance, like the surprising results for LinearMax ? Also, it would be interesting to succeed in converting the d-dimensional version of Megiddo's algorithm into a tolerant version of itself.

7 Conclusion

The concrete tolerant algorithms presented in the paper are of interest in many applications, both inside and outside of the computer world (finding the maximum applies to tournament organization for instance), as the model used is very flexible. Also, I think that the idea itself of designing algorithms resilient to faulty primitives is of interest, even more so in a time in which building faster computers becomes increasingly difficult. In any case, the model used in the paper being the basis of present day research testifies to its interest.

Finally, I believe an interesting area of further study would be to build *tolerant data structures*; data structures which are resilient to unreliability of the oracle they query to gather access to underlying data. For instance, could we build a tree or heap structure for which comparisons are unreliable ? Such structures might allow us to build tolerant algorithms in a modular way, and greatly ease conversion of existing algorithms.

References

[1] Braverman, M., Efremenko, K., Gelles, R., and Yitayew, M. A. Optimal short-circuit resilient formulas. *arXiv preprint arXiv:1807.05014* (2018).

[2] Braverman, M., and Mossel, E. Noisy sorting without resampling. In *SODA* (2008).

[3] Dyer, M., Megiddo, N., and Welzl, E. 38 linear programming. https://www.inf.ethz.ch/personal/emo/PublFiles/LpSurvey03.pdf.

[4] Feige, U., Raghavan, P., Peleg, D., and Upfal, E. Computing with noisy information. *SIAM Journal on Computing 23*, 5 (1994), 1001–1018.

[5] Geissmann, B. Longest increasing subsequence under persistent comparison errors. *arXiv preprint arXiv:1808.03307* (2018).

[6] Geissmann, B., Leucci, S., Liu, C., and Penna, P. Optimal sorting with persistent comparison errors. *CoRR abs/1804.07575* (2018).

[7] Klein, R., Penninger, R., Sohler, C., and Woodruff, D. P. Tolerant algorithms. In *European Symposium on Algorithms* (2011), Springer, pp. 736–747.

[8] Knuth, D. E. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[9] Leucci, S., and Liu, C. A nearly optimal algorithm for approximate minimum selection with unreliable comparisons. *CoRR abs/1805.02033* (2018).