CSC2420 Spring 2017: Algorithm Design, Analysis and Theory
Lecture 10

Allan Borodin

March 20, 2017

Announcements and todays agenda

- Announcements
  1. Assignment 2 is due next Monday, March 27
  2. I have posted the first question for Assignment 3.
- Todays agenda
  1. Streaming algorithms for other "counting problems"
     1. Heavy hitters (frequently occuring elements in a data stream)
     2. Counting the number of distinct and unique elements
  2. The semi-streaming model
  3. Relation of the streaming model to other models.
  4. Return to offline algorithms
     1. Discussion of randomized primality testing.
     2. Discussion of weighted majority algorithm
     3. The Lovasz Local lemma and the Moser-Tardos algorithm for finding a satisfying instance of an exact k-SAT formula in which every clause C shares a variable with at most $d < 2^k/e$ other clauses.

A general class of counting based problems

In what follows, I will not discuss the precise data structures used to represent and process elements efficiently (in time and space). It may also be the case that at the end of processing a stream, the answer may have to be checked or some other post processing might be done. And although I am focusing on one pass streaming algorithms, many streaming papers present results for a small nunber of passes.

A general class of counting based problems

In what follows, I will not discuss the precise data structures used to represent and process elements efficiently (in time and space). It may also be the case that at the end of processing a stream, the answer may have to be checked or some other post processing might be done. And although I am focusing on one pass streaming algorithms, many streaming papers present results for a small nunber of passes.

A general problem of interest in the streaming model is the following:

- We have a stream $(a_1, d_1) \ldots, (a_n, d_n)$ where $a_i \in M$ with $|M| = m$, and $d_i \in \mathbb{N}$. We can assume $M = \{1, 2, \ldots m\}$ and either measure time and space in terms of bit operations or unit cost for elements and counts stored in wordsr elements
- The input (a_i, d_i) means that the count for element a_i has increased (resp. decreased) by d_i if $d_i > 0$ (resp., $d_i < 0$).
- When $d_i > 0$ (for all i) this is called the cash register model and the more general model is called the turnstile model.

Simple occurence counting problems

We will focus on $d_i = 1$ (for all i) which is a simple occurence count model.

- In the simple occurence count, we may want to keep track of frequently occuring elements
- There are two common variants of "frequently occuring elements". For a given k, we may want to know:
  1. Which elements, if any, occur at least n/k times?
  2. What are the k most frequently occuring items?
- On the other hand we may want to know how many elements occur only once in the stream or (as a case of the frequency moments problems) how many distinct items occur in the stream.

Special case: finding a majority element if one exists

If we want to know the frequency of occurence of any element j, we could simply maintain a counter for each element. This would require m counters which (in the desire for small memory) might be prohibitive.

How much space do we need to determine if the stream has a majority element?

Special case: finding a majority element if one exists

If we want to know the frequency of occurence of any element j, we could simply maintain a counter for each element. This would require m counters which (in the desire for small memory) might be prohibitive.

How much space do we need to determine if the stream has a majority element? What input parameters should a space bound depend on?

Special case: finding a majority element if one exists

If we want to know the frequency of occurence of any element j, we could simply maintain a counter for each element. This would require m counters which (in the desire for small memory) might be prohibitive.

How much space do we need to determine if the stream has a majority element? What input parameters should a space bound depend on?
The majority algorithm (first proposed by Boyer and Moore [1980] but not as a streaming algorithm) finds a majority element if one exists as follows:

Majority algorithm

Initialize $c := 0$
For $i = 1 \ldots n$
    If $c = 0$
        Then $Candidate := a_i$ and $c := 1$
        Else If $a_i = Candidate$
            Then $c := c + 1$
            Else $c := c - 1$
EndFor

Majority algorithm and an extension

Lets say that a majority element occurs more then $n/2$ times, so as to not worry about the case that the number of occurrences is exactly $n/2$.

Claim: If there is a majority element, then it will be the value of *Candidate*. Of course, the element in *Candidiate* may not be a majority element.

While we might not expect a stream to have a majority element, for a reasonably large k, we might expect that some elements will occur frequently. In the *heavy hitters problem* we are given a k and asked to find elements (if any) that occur at least n/k times. (Setting $k = 2 - \delta$ for any $\delta > 0$ is the special case of the majority element.)

In general we cannot expect to have a small space streaming algorithm that will exactly solve the heavy hitters problem. Instead we will try for a streaming algorithm tht solves the ϵ-heavy hitters problem where we produce a set C of elements that satisfies:

- Every element that occurs at least n/k times is in C and
- Every element in C occurs at least $n/k - \epsilon n$ times.

A counter-based ϵ-heavy hitters algorithm

Twenty years after the Majority algorithm, a "natural" generalization of Majority was articulated in a couple of papers. In fact the following algorithm was a rediscovery of an algorithm by Misra and Gries [1982]. Think of the Majority where $k = 2$ where we use 1 counter. Here we use $k - 1$ counters.

Misra and Gries heavy hitters as a streaming algorithm

Initialize $C := \varnothing$
For $i = 1 \ldots n$
  If $a_i = C_j$ % C_j is an element of the set C with count c_j
    Then $c_j := c_j + 1$ $c_j = 0$ is implicitly set for elements not in C
    Else If $|C| = \ell < k - 1$
      Then $C_\ell = a_i$; $c_\ell := 1$ % Start a new counter
      Else For $j = 1 \ldots k - 1$,
          $c_j := c_j - 1$
          If $c_j = 0$, then $C := C \setminus C_j$ % Make room for new candidates
          EndFor
EndFor

ϵ-heavy hitters continued

If we set $k = \frac{1}{\epsilon}$, we guarantee that the count associated with each element of C is at most ϵn below its true count.

There are other counter-based variants of the previous algorithm. Metwally et al [2005] show how to adapt the following algorithm to both the heavy hitters and k most frequently occuring elements problems.

The Metwaly et al "space-saving" algorithm maintain k (or $1/\epsilon$) counters and now we initialize the counters with the first k occuring elements. And as before when a new element arrives, if it already is one of the k elements being "monitored" in a counter, then that count is incremented. Otherwise, the new element replaces the element having the lowest count c (at this time) and the count for the new element is set (surprisingly) to $c + 1$. The key property is that the minimum counter value *min* (at the end) will be at most n/k. (Note that we can assume the k counters are occupied since there are $m > k$ distinct elements to make the problem interesting). Furthermore, any element coccuring more than *min* times must be maintained in one of the counters.

Other approaches for solving frequently occuing elements

There are two other common types of algorithms for heavy hitters, one based on approximating quantiles, and the other based on "sketches" which are randomized projections of the input stream viewed as a vector.

Other approaches for solving frequently occuing elements

There are two other common types of algorithms for heavy hitters, one based on approximating quantiles, and the other based on "sketches" which are randomized projections of the input stream viewed as a vector.

One nice approach used in some applications is the *count-min sketch* which can be thought of as an example of "compressed sensing".

A count-min sketch uses a small number (ℓ) of pairwise independent hashing functions $h_1, \ldots, h_\ell$, and a "medium" number (b) of buckets into which elements are hashed. In large data stream applications, n could be in the hundres of millions (or billions) and b might be in the low thousands while ℓ can be thought of as a small constant. We think of ℓ and b as being indepedent of n.

The sketch supports two operations on elements x, namely *Increment*(x) and *Count*(x) where *Count*(x) is attempting to count the number of times that *Increment*(x) has been applied.

Count-min sketch continued

The idea of the count-min sketch is quite simple but (like other applications of hashing) very useful.

Count-min sketch continued

The idea of the count-min sketch is quite simple but (like other applications of hashing) very useful.

Whenever an element x appears, we increment the bucket count in $h_i(x)$ for each of the ℓ hashing functions $\{h_i\}$. When the stream is completed, the count Z_i for bucket $h_i(x)$ would either be the correct count $c(x)$ for element x (if there were no collisions with other y such that $h_i(x) = h_i(y)$) or (more likely) be an overestimate of the true count for x. Thus we can take the $\min_i\{Z_i(x)\}$ as our estimate of $c(x)$ and know that we can only have an overestimate.

Count-min sketch continued

The idea of the count-min sketch is quite simple but (like other applications of hashing) very useful.

Whenever an element x appears, we increment the bucket count in $h_i(x)$ for each of the ℓ hashing functions $\{h_i\}$. When the stream is completed, the count Z_i for bucket $h_i(x)$ would either be the correct count $c(x)$ for element x (if there were no collisions with other y such that $h_i(x) = h_i(y)$) or (more likely) be an overestimate of the true count for x. Thus we can take the $\min_i\{Z_i(x)\}$ as our estimate of $c(x)$ and know that we can only have an overestimate.

The question then is what is the probability δ that the min-count value $Z = \min_i Z_i$ for an element x will be (say) more than an ϵn overestimate. This will clearly depend on how we set the parameters ℓ and b.

Count-min sketch continued

The idea of the count-min sketch is quite simple but (like other applications of hashing) very useful.

Whenever an element x appears, we increment the bucket count in $h_i(x)$ for each of the ℓ hashing functions $\{h_i\}$. When the stream is completed, the count Z_i for bucket $h_i(x)$ would either be the correct count $c(x)$ for element x (if there were no collisions with other y such that $h_i(x) = h_i(y)$) or (more likely) be an overestimate of the true count for x. Thus we can take the $\min_i\{Z_i(x)\}$ as our estimate of $c(x)$ and know that we can only have an overestimate.

The question then is what is the probability δ that the min-count value $Z = \min_i Z_i$ for an element x will be (say) more than an ϵn overestimate. This will clearly depend on how we set the parameters ℓ and b.

How would you set b and ℓ so that:

$$Prob[Z > c(x) + \epsilon n] \leq \delta$$

Computing the number of distinct elements and unique elements in a stream

We want to approximate the number D_n of distinct elements and the number U_n of elements that occurs just once in a stream of n elements. Here I am following the presentation in the 2009 lecture notes by Muthukrishnan.

Computing the number of distinct elements and unique elements in a stream

We want to approximate the number D_n of distinct elements and the number U_n of elements that occurs just once in a stream of n elements. Here I am following the presentation in the 2009 lecture notes by Muthukrishnan.

The approach is to estimate U_n/D_n and D_n.

We want to sample (close to) uniformly from the distinct elements; that is, with probablity that each element is selected with probability $1/\approx D_n$.

Conceptually, if we have k uniformly chosen permutations μ_j (of the universe M), then for each such permutation we will keep track of the element s (and its count) whose rank $\mu_j(s)$ (in the permutation) is the best so far. Letting c_j be the count at the end of the stream for a given μ_j, we estimate U_n/D_n by $EST = \frac{|\{j:c_j=1\}|}{k}$.

Computing the number of distinct elements and unique elements in a stream

We want to approximate the number D_n of distinct elements and the number U_n of elements that occurs just once in a stream of n elements. Here I am following the presentation in the 2009 lecture notes by Muthukrishnan.

The approach is to estimate U_n/D_n and D_n.

We want to sample (close to) uniformly from the distinct elements; that is, with probablity that each element is selected with probability $1/\approx D_n$.

Conceptually, if we have k uniformly chosen permutations μ_j (of the universe M), then for each such permutation we will keep track of the element s (and its count) whose rank $\mu_j(s)$ (in the permutation) is the best so far. Letting c_j be the count at the end of the stream for a given μ_j, we estimate U_n/D_n by $EST = \frac{|\{j:c_j=1\}|}{k}$.

Setting $k = O(\log(1/\delta)/\epsilon^2)$, it is shown that :

$$Prob[(1-\epsilon)U_n/D_n \leq Est \leq (1+\epsilon)U_n/D_n] \geq 1-\delta$$

Computing U_n/D_n continued

Ignoring the fact that storing the permutation would already take too much space, we do the following to compute each c_j:

Computing the counts for elements found to have the lowest rank

$s := a_1; c_j = 1$
For $i = 2 \ldots n$
  If $\mu_j(a_i) < \mu_j(s)$
    Then $s := a_i; c_j := 1$
    Else If $\mu_j(a_i) = \mu_j(s)$ then $c_j := c_j + 1$;
    % Otherwise $\mu_j(a_i) > \mu_j(s)$ and nothing is done

Instead of using random permuations of the universe M, it is sufficient to have a family of approximate *min-wise hashing functions* h_j that satisfy : $\forall S \subseteq M \forall s \in S \; |j : [h_j(s) = \min_{x \in S}|$ is at least a $1/(1+\epsilon)$ fraction of the number of hash functions in the family. It is known how to specify such functions using $O(\log^2 m)$ bits.

Estimating D_n

Suppose we could determine $D_n < t$ for any $t = (1+\epsilon)^j$ for $j = 1, \ldots \log_{1+\epsilon} m$. Then we could estimate D_n within a factor of $(1 \pm \epsilon)$.

To determine if $D_n < t$, it suffices to hash $\{1, \ldots, m\}$ into $\{1, \ldots, t\}$ for a sufficient number of independent hash functions h_j. If $c_j =$ the number of elements x such that $h_j(x) = 1$, then first estimate that $D_j < t$ iff $c = 0$.

One gets a good estimate by taking the majority answer for sufficiently many h_j.

Some results regarding the "semi-streaming" model

Most of the work to date in streaming has concerned computing statistics and counting and the desired space bound is at most some logarithmic function of the parameters involved (i.e. n and m).

Some results regarding the "semi-streaming" model

Most of the work to date in streaming has concerned computing statistics and counting and the desired space bound is at most some logarithmic function of the parameters involved (i.e. n and m).

In 2005, Feigenbaum et al introduced the *semi-streaming model* in order to study graph problems in a streaming model. This is still the same streaming model but now the stream elements are either edges or vertices of a graph. (We may or may not know in advance $m = |E|$ or $n = |V|$.)

Since the usual goal is to produce a solution (i.e., a max matching, a max independent set, a densest subset of vertices, a colouring , etc), we need at least space n to present a solution. Semi-streaming algorithms aim to (approximately) solve graph problems using space $\tilde{O}(n)$ rather than space $O(m)$. (Here the soft $\tilde{O}$ hides polylog factors.)

The semi-streaming model is studied with regard to both a single pass and a small number of passes.

What is known about the bipartite matching problem in the semi-streaming model?

- As far as I know, in the edge arrival model, there is no known worst case semi-streaming algorithm (even randomized) that achieves an approximation better than the $\frac{1}{2}$ approximation achieved by any maximal matching algorithm.
- There is a slightly improved approximation ratio due to Konrad et al [2014] if one allows a random ordering of the input edges.
- In the vertex arrival model, the randomized Ranking algorithm can be simulated by a randomized semi-streaming algorithm.
- Surprisingly, Goel et al [2011] show that there is a *deterministic* semi-streaming algorithm that achieves the $1 - \frac{1}{e}$ KVV approximation ratio. We remind oursleves that no deterministic online algorithm can do better than $\frac{1}{2}$.
- It is also interesting to note that the size of a maximium matching (in an arbitrary undirected graph) can be approximated (within a poly logarithmic factor) by a streaming algorithm using poly logarithmic space assuming the input stream is a random ordering of the edges.

An aside; the edge model vs the vertex model

With regard to the edge vs vertex input models for graph problems, is there any relation between these input models with respect to various one pass algorithms (e.g. priority, online adversarial, ROM, i.i.d., streaming)?

- In streaming, the issue of the input model makes sense for any graph problem. But for the online and priority models which require irrevocable decisions about each input item, the issue only makes sense if the graph problem can be formulated as either a decision on edges or a decision of vertices (e.g. matching, shortest path problems).
- With respect to the online models, I do not know how to simulate an edge input algorithm by a vertex input algorithm or conversely how to simulate a vertex input algorithm by an edge input algorithm. What are the advatanges and disadvatages of each input model?
- In the streaming and online models we have results that give evidence that for bipartite matching the vertex model obtains better results.
- In the priority framework, any algorithm for the vertex model can be transformed into an algorithm for the edge model.

How do the streaming models relate to other models of computation

Given that we can think of streaming algorithms as online algorithms, it is natural to compare the streaming model with the competitive analysis view of online algorithms (i.e., one pass algorithms making irrevocable decisions about each item).

It should also be clear that we can adapt the streaming model to allow random input streams (e.g., the i.i.d. distributional models and the ROM model) and there are some results along these lines. However, we have only considered the worst case adversarial (streaming) model.

How do the streaming models relate to other models of computation

Given that we can think of streaming algorithms as online algorithms, it is natural to compare the streaming model with the competitive analysis view of online algorithms (i.e., one pass algorithms making irrevocable decisions about each item).

It should also be clear that we can adapt the streaming model to allow random input streams (e.g., the i.i.d. distributional models and the ROM model) and there are some results along these lines. However, we have only considered the worst case adversarial (streaming) model.

The defining difference in these two online models is that the streaming model limits space while the competitive analysis online model requires irrevocable decisions about each input item and these decisions constitute the solution of the algorithm. The competitive analysis model does not readily apply to (say) counting problems. (About what are we making irrevocable decisions?)

Semi-streaming model vs. competitive analysis model

Thus to compare the models, we should restrict ourselves to search and optimization problems using the semi-streaming modeling.

As just noited, bipartite matching in the vertex input model can be solved *deterministically* by a semi-streaming algorithm with approximation ratio $1 - \frac{1}{e}$ and we know that in the competitive analysis world we cannot deterministically do better than $\frac{1}{2}$.

The intuitition is that the streaming model is a more permissive model in that it does not have to make irrevocable decisions. And it almost seems like any compeititve anaylsis online algorithm (deterministic or randomized) can be simulated by a streaming algorithm which is certainly the case for say the Ranking algorithm.

But a competitive analysis algorithm does not have to maintain the $\tilde{O}(n)$ space bound and it could be remembering all the edges it has seen thus far and moreover, use even more space in determining its decisions. However, it is not clear if such an online graph algorithm could exploit this.

Streaming algorithms becoming parallel algorithms

Streaming algorithms and algorithmic frameworks for parallel computation (such a Map Reduce) are both approaches that are meant for very large datasets.

Looking back at a number of the streaming algorithms we see that they often are running a number of different processes in paralell and use one or only a few passes over the data. For example, the algorithms for say heavy hitters maintain several counters (or hash buckets).

This aspect of many streaming algorithms (with one or few passes) lends itself to parallelization. As a specific examle, Bahmani et al [2012] design streaming algorithms for the densest subgraph problem and then implement those algorithms within the distributed MapReduce paradigm.

Map Reduce

Map Reduce algorithms do computation on a large number of servers interconnected by a fast network. (There is no shared memory.) Each server performs computations (on the data they hold) and then exchange data.
Map Reduce algorithms (as say inplemented in Hadoop) operate on (key,value) pairs in rounds, each round consisting of three stages:

- Map: Transforms a (key,value) into one or several new (key,value) pairs.
- Shuffle: All the values associated with a given key are sent to the same (perhaps virtual) machine. This aspect is carried out automatically by the system.
- Reduce: All values associated with a given key get batched into a multiset of (key,value) pairs

See the models specified in Feldman et al [2010], Karloff et al [2010] and Beame et al [2013].

Primality testing

- I now want to briefly turn attention to one of the most influential randomized algorithms, namely a poly time randomized algorithm for primality (or perhaps better called compositeness) testing. Let $PRIME = \{N | N \text{ is a prime number}\}$ where N is represented in say binary (or any base other than unary) so that $n = |N| = O(\log N)$.
- History of polynomial time algorithms:
  1. Vaughan 1972 showed that *PRIMES* is in *NP*. Note that co-PRIMES (i.e. the composites) are easily seen to be in *NP*.
  2. One sided error randomized algorithms (for compositeness) by Solovay and Strassen and independently Rabin in 1974. That is, $Prob[ALG \text{ says } N \text{ prime } | N \text{ composite}] \leq \delta < 1$ and $Prob[ALG \text{ says } N \text{ composite } | N \text{ prime}] = 0$
  3. The Rabin test is related to an algorithm by Miller that gives a deterministic polynomial time algorithm assuming a conjecture that would follow from (the unproven) ERH. The Rabin test is now called the Miller-Rabin test.
  4. Goldwasser and Killian establish a 0-sided randomized algorithm.
  5. In 2002, Agarwal, Kayal and Saxena show that primality is in deterministic polynomial time.

Why consider randomized tests when there is a deterministic algorithm?

- Even though there is now a deterministic algorithm, it is not nearly as efficient as the 1-sided error algorithms which are used in practice. These randomized results spurred interest in the topic (and other number theoretic algorithms) and had a major role in cryptographic protocols (which often need random large primes). Moreover, these algorithms became the impetus for major developments in randomized algorithms.
- While many of our previous algorithms (excluding the streaming algorithm for F_k) might be considered reasonably natural (or natural extensions of a deterministic algorithm), the primality tests require some understanding of the subject matter (i.e. a little number theory) and these algorithms are not something that immediately comes to mind.

Some basic number theory we need

- $Z_N^* = \{a \in Z_N : gcd(a, N) = 1\}$ is a (commutative) group under multiplication mod N.
- If N is prime, then
  1. For $a \neq 0(modN), a^{N-1} = 1(modN)$.
  2. Z_N^* is a cyclic group; that is there exists a generator g such that $\{g, g^2, g^3, \ldots, g^{N-1}\}$ (all mod N) is the set Z_N^*. This implies that $g^i \neq 1(modN)$ for any $1 \leq i < N - 1$.
  3. There are exactly two square roots of 1 in Z_N^*, namely 1 and -1.
- The Chinese Remainder Theorem: Whenever N_1 and N_2 are relatively prime (i.e. $gcd(N_1, N_2) = 1$), then for all $v_1 < N_1$ and $v_2 < N_2$, there exists a unique $w < N_1 \cdot N_2$ such that $v_1 = w(modN_1)$ and $v_2 = w(modN_2)$.

A simple but "not quite" correct algorithm

We also need two basic computational facts.

1. a^i mod N can be computed efficiently.
2. $gcd(a, b)$ can be efficiently computed.

The following is a simple algorithm that works except for an annoying set of numbers called Carmichael numbers.

Simple algorithm ignoring Carmichael numbers

Choose $a \in Z_N$ uniformly at random.
If $gcd(a, N) \neq 1$, then Output Composite
If a^{N-1} mod $N \neq 1$, then Output Composite
Else Output Prime

When does the simple algorithm work?

- $S = \{a | gcd(a, N) = 1 \text{ and } a^{N-1} = 1\}$ is a subgroup of Z_N^*
- If there exists an $a \in Z_N^*$ such that $gcd(a, N) = 1$ but $a^{N-1} \neq 1$, then S is a proper subgroup of Z_N^*.
- By Lagrange's theorem, if S is a proper subgroup, $|S|$ must divide the order of the group so that $|S| \leq \frac{N-1}{2}$
- Thus the simple algorithm would be a 1-sided error algorithm with probabiltiy $< \frac{1}{2}$ of saying Prime when N is Composite.
- The only composite numbers that give us trouble are the Carmichael numbers (also known as *false primes*) for which $a^{N-1} mod N = 1$ for all a such that $gcd(a, N) = 1$.
- It was only recently (relatively speaking) that in 1994 it was proven that there are an infinite number of Carmichael numbers.
- The first three Carmicahel numbers are 561, 1105, 1729

Miller-Rabin 1-sided error algorithm

Let $N-1 = 2^t u$ with u odd %Since wlg. N is odd, $t \geq 1$
Randomly choose non zero $a \in Z_N$ %Hoping that a will be composite certificate
If $gcd(a, N) \neq 1$ then report Composite
$x_0 = a^u$ %All computation is done mod N
For $i = 1 \ldots t$
    $x_i := x_{i-1}^2$
    If $x_i = 1$ and $x^{i-1} \notin \{-1, 1\}$, then report Composite
End For
If $x_t \neq 1$, then report Composite $\%x_t = x^{N-1}$
Else report Prime

Analysis sketch of Miller-Rabin

- Let S be the set of $a \in N$ that pass (i.e. fool) the Rabin-Miller test.
- S is a subgroup of Z_N^*. We want to show that S is a proper subgroup and then as before by Langrange we will be done.
- It suffices then to find one element $w \in Z_N^*$ that will not pass the Miller-Rabin test.
  Case 1: N is not Carmichael and then we are done.
  Case 2: N is Carmichael and hence N cannot be a prime power.
  - $N = N_1 \cdot N_2$ and $gcd(N_1, N_2) = 1$ and of course odd
  - The non-certificates must include some b such that $b^{2^i u} = -1(modN)$ and hence $b^{2^i u} = -1(modN_1)$
  - By the Chinese Remainder Theorem, there exists $w = v(modN_1)$ and $w = 1(modN_2)$
  - Hence $w^{2^i u} = -1(modN_1)$ and $w^{2^i u} = 1(modN_2)$
  - This implies $w^{2^i u} \notin \{-1, 1\} \pmod N$

New topic: the weighted majority algorithm

I am following a survey type paper by Arora, Hazan and Kale [2008]. To quote from their paper: "We feel that this meta-algorithm and its analysis should be viewed as a basic tool taught to all algorithms students together with divide-and-conquer, dynamic programming, random sampling, and the like".

- The weighted majority algorithm and generalizations
  The "classical" WMA pertains to the following situation:
  Suppose we have say n expert weathermen (or maybe "expert" stock market forecasters) and at every time t, they give a binary prediction (rain or no rain, Raptors win or lose, dow jones up or down, Canadian dollar goes up or down, Trump will tweet).
- Now some or all of these experts may actually be getting their opinions from the same sources (or each other) and hence these predictions can be highly correlated.
- Without any knowledge of the subject matter (and why should I be any different from the "experts") I want to try to make predictions that will be nearly as good (over time t) as the BEST expert.

The weighted majority algorithm

The WM algorithm

Set $w_i(0) = 1$ for all i

For $t = 0...$

Our $(t+1)^{st}$ predication is

0: if $\sum_{\{i:\text{ expert } i \text{ predicts } 0\}} w_i(t) \geq (1/2)\sum_i w_i(t)$

1: if $\sum_{\{i:\text{ expert } i \text{ predicts } 1\}} w_i(t) \geq (1/2)\sum_i w_i(t)$; arbitrary o.w.

% We vote with weighted majority; arbitrary if tie

For $i = 1..n$

If expert i made a mistake on $(t+1)^{st}$ prediction

then $w_i(t+1) = (1-\epsilon)w_i(t)$;

else $w_i(t+1) = w_i(t)$

End If

End For

End For

How good is our uninformed MW prediction?

Theorem : Perfomance of WM

Theorem: Let $m_i(t)$ be the number of mistakes of expert i after the first t forecasts, and let $M(t)$ be the number of our mistakes. Then for any expert i (including the best expert) $M(t) \leq \frac{2 \ln n}{\epsilon} + 2(1+\epsilon)m_i(t)$.

- That is, we are "essentially" within a multiplicative factor of 2 plus an additive term of the best expert (without knowing anything).
- Using randomization, the factor of 2 can be removed. That is, instead of taking the weighted majority opinion, in each iteration t, choose the prediction of the i^{th} expert with probability $w_i(t)/\sum_i w_i(t)$

Theorem: Performance of Randomized WM

For any expert i, $\mathbf{E}[M(t)] \leq \frac{\ln n}{\epsilon} + (1+\epsilon)m_i(t)$

Proof of deterministic WM

Let's assume that $\epsilon \leq 1/2$. It follows that

$$-\epsilon - \epsilon^2 \leq \ln(1-\epsilon) < -\epsilon$$

Let $w_{i,t}$ be the weight of the i^{th} expert at time t and let $m_i(t)$ be the number of mistakes made by expert i. . Consider the potential function $\Phi(t) = \sum_i w_{i,t}$. Clearly

$$\Phi(t) \geq w_{i,t} = (1-\epsilon)^{m_i(t)}$$

We now need an upper bound on $\Phi(t)$. Since each time the WM algoriithm makes a mistake, at least half of the algorithms make a mistake so that $\Phi(t) \leq (1-\epsilon/2)\Phi(t-1)$. Starting with $\Phi(0) = n$, by induction

$$\Phi(t) \leq n \cdot (1-\epsilon/2)^{M(t)}$$

Putting the two inequalities together and taking logarithms

$$\ln(1-\epsilon)m_i(t) \leq \ln n + M(t)\ln(1-\epsilon/2)$$

The argument is completed by rearranging, using the above facts concerning $\ln(1-\epsilon)$ and then dividing by $\epsilon/2$.

What is the meaning of the randomized impovement?

- In many applications of randomization we can argue that randomization is (provably) necessary and in other applications, it may not be provable so far but current experience argues that the best algorithm in theory and practice is randomized.
- For some algorithms (and especially online algorithms) analyzed in terms of worst case performance, there is some debate on what randomization is actually accomplishing.
- In a [1996] article Blum states that "Intuitively, the advantage of the randomized approach is that it dilutes the worst case". He continues to explain that in the determinstic algorithm, slightly more than half of the total weight could have predicted incorrectly, causing the algorithm to make a mistake and yet only reducing the total weight by $1/4$ (when $\epsilon = 1/2$). But in the randomized version, there is still a .5 probability that the algorithm will predict correctly. Convincing?

An opposing viewpoint

- In the blog LessWrong this view is strongly rejected. Here the writer makes the following comments: "We should be especially suspicious that the randomized algorithm guesses with probability proportional to the expert weight assigned. This seems strongly reminiscent of betting with 70% probability on blue, when the environment is a random mix of 70% blue and 30% red cards. We know the best bet and yet we only sometimes make this best bet, at other times betting on a condition we believe to be less probable.
  Yet we thereby prove a smaller upper bound on the expected error. Is there an algebraic error in the second proof? Are we extracting useful work from a noise source? Is our knowledge harming us so much that we can do better through ignorance?" The writer asks: "So what's the *gotcha* ... the improved upper bound proven for the randomized algorithm did not come from the randomized algorithm making systematically better predictions - doing superior cognitive work, being more intelligent - but because we arbitrarily declared that an intelligent adversary could read our mind in one case but not in the other."

Further defense of the randomized approach

- Blum's article expresses a second benefit of the randomized approach: "Therefore the algorithm can be naturally applied when predictions are 'strategies' or other sorts of things that cannot easily be combined together. Moreover, if the 'experts' are programs to be run or functions to be evaluated, then this view speeds up prediction since only one expert needs to be examined in order to produce the algorithm's prediction ...."
- We also know (in another context) that ROM ordering can beat any deterministic priority order say for the online bipartite matching problem.

Generalizing: The Multiplicative Weights algorithm

The Weighted Majority algorithm can be generalized to the multiplicative weights algorithm. If the i^{th} expert or decision is chosen on day t, it incurs a real valued cost/profit $m_i(t) \in [-1, 1]$. The algorithm then updates $w_i(t+1) = (1 - \epsilon m_i(t))w_i(t)$. Let $\epsilon \leq 1/2$ and $\mathbf{\Phi}(t) = \sum_i w_i(t)$. On day t, we randomly select expert i with probability $w_i(t)/\Phi(t)$.

Performance of The MW algorithm

The expected cost of the MW algorithm after T rounds is $\sum_{t=1}^{T} \mathbf{m}(t) \cdot \mathbf{p}(t) \leq \frac{\ln n}{\epsilon} + \sum_{t=1}^{T} m_i(t) + \epsilon \sum_{t=1}^{T} |m_i(t)|$

Reinterpreting in terms of gains instead of losses

We can have a vector $\mathbf{m}(t)$ of gains instead of losses and then use the "cost vector" $-\mathbf{m}(t)$ in the MW algorithm resulting in:

Performance of The MW algorithm for gains

$\sum_{t=1}^{T} \mathbf{m}(t) \cdot \mathbf{p}(t) \geq -\frac{\ln n}{\epsilon} + \sum_{t=1}^{T} m_i(t) - \epsilon \sum_{t=1}^{T} |m_i(t)|$

By taking convex combinations, an immediate corollary is

Performance wrt. a fixed distribution p

$\sum_{t=1}^{T} \mathbf{m}(t) \cdot \mathbf{p}(t) \geq -\frac{\ln n}{\epsilon} + \sum_{t=1}^{T} \mathbf{m}(t) - \epsilon |\mathbf{m}(t)|)\mathbf{p}$

An application to learning a linear binary classifier

Instead of the online application of following expert advice, let us now think of "time" as rounds in an iterative procedure. In particular, we would like to compute a linear binary classifier (when it exists).

- We are trying to classsify objects characterized by n features; that is by points $\mathbf{a}$ in $\Re^n$. We are given m labelled examples $(\mathbf{a}_1, \ell_1), \ldots, (\mathbf{a}_m, \ell_m)$ where $\ell_j \in \{-1, +1\}$
- We are going to assume that these examples can be "well classified" by a linear classifier in the sense that there exists a non negative vector $\mathbf{x}^* \in \Re^n$ (with $x_i \geq 0$) such that $sign(\mathbf{a}_j \cdot \mathbf{x}^*) = \ell_j$ for all j.
- This is equivalent to saying $\ell_j \mathbf{a}_j \cdot \mathbf{x}^* \geq 0$ and furthermore (to explain the "well") we will say that $\ell_j \mathbf{a}_j \cdot \mathbf{x}^* \geq \delta$ for some $\delta > 0$.
- The goal now is to learn some linear classifer; ie a non negative $\mathbf{x} \in \Re^n$ such that $\ell_j \mathbf{a}_j \cdot \mathbf{x}^* \geq 0$. Without loss of generality, we can assume that $\sum_i x_i = 1$.
- Letting $\mathbf{b}_j = \ell_j \mathbf{a}_j$, this can now be veiwed as a reasonably general LP (search) problem.

Littlestone's Winnow algorithm for learning a linear classifier

- Litlestone [1987] used the multiplicative weights approach to solve this linear classification problem.
- Let $\rho = \max_j ||\mathbf{b}_j||_\infty$ and let $\epsilon = \delta/(2\rho)$
- The idea is to run the MW algorthm with the decisions given by the n features and gains specified by the m examples. The gain for feature i with respect to the j^{th} example is defined as $(\mathbf{b}_j)_i/\rho$ which is in [-1,1]. The $\mathbf{x}$ we are seeking is the distribution $\mathbf{p}$ in MW.

The Winnow algorithm

Initialize $\mathbf{p}$
While there are points not yet satisfied
  Let $\mathbf{b}_j \cdot \mathbf{p} < 0$ % a constraint not satisfied
  Use MW to upate $\mathbf{p}$
End While

Bound on number of iterations

The Winnow algorithm will terminate in at most $\lceil 4\rho^2 \ln n/\delta^2 \rceil$ iterations.

Some additional remarks on Multiplicative Weights

The survey by Arora, Hazan and Kale [2012] discusses other modifications of the MW paradigm and numerous applications. In terms of applications, they sketch results for

- Aporoximately solving (in the sense of property testing) the decision problem for an LP; there that is given linear constraints expressed by $A\mathbf{x} \geq \mathbf{b}$, the decision problem is to see if such a non-negative $\mathbf{x}$ exists (or more generally, if $\mathbf{x}$ is in some given convex set). The algorithm either returns a $\mathbf{x} : \mathbf{A_i x} \geq \mathbf{b_i} - \delta$ for all i and some additive approximation δ or says that the given LP was infeasible.
- Solving zero sum games approximately.
- The AdaBoost algorithm of Shapire and Freund
- Some other specific applications including a class of online algorithms.

The Lovász Local Lemma (LLL)

- Suppose we have a set of "bad" random events $E_1, \ldots, E_m$ with $Prob[E_i] \leq p < 1$ for each i. Then if these events are independent we can easily bound the probability that none of the events has occurred; namely, it is $(1-p)^m > 0$.
- Suppose now that these events are not independent but rather just have limited dependence. Namely suppose that each E_i is dependent on at most r other events. Then the Lovász local Lemma (LLL) states that if $e \cdot p \cdot (r+1)$ is at most 1, then there is a non zero probability that none of the bad events E_i occurred.
- As stated this is a non-constructive result in that it does not provide a joint event in which none of the bad events occured.
- There are a number of applications of LLL including (Leighton, Maggs, Rao) routing, the restricted machines version of the Maxmin "Santa Claus" problem and as we shall now see, solving exact k-SAT under suitable conditions on the clauses.

A somewhat canonical application of the LLL

- Let $F = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ be a an exact k CNF formula. From our previous discussion of the exact Max-k-Sat problem and the naive randomized algorithm, it is easy to see that if $m < 2^k$, then F must be satisfiable. ($E[\text{clauases satisfied}] = \frac{2^k-1}{2^k} m > m - 1$ when $m < 2^k$.)
- Suppose instead that we have an arbitrary number of clauses but now for each clause C, at most r other clauses share a variable with C.
- If we let E_i denote the event that C_i is not satisfied for a random uniform assignment and hence having probability $1/(2^k)$, then we are interested in having a non zero probability that none of the E_i occurred (i.e. that F is satisfiable).
- The LLL tells us that if $r + 1 \leq \frac{2^k}{e}$, then F is satisfiable.
- As informally but nicely stated in Gebauer et al [2009]: "In an unsatisable CNF formula, clauses have to interleave; the larger the clauses, the more interleaving is required."

A constructive algorithm for the previous proof of satisfiability

- Here we will follow a somewhat weaker version (for $r \leq 2^k/8$) proven by Moser [2009] and then improved by Moser and G. Tardos [2010] to give the tight LLL bound. This proof was succinctly explained in a blog by Lance Fortnow
- This is a constructive proof in that there is a randomized algorithm (which can be de-randomized) that with high probability (given the limited dependence) will terminate and produce a satisfying assignment in $O(mlogm)$ evaluations of the formula.
- Both the algorithm and the analysis are very elegant. In essence, the algorithm can be thought of as a local search search algorithm and it seems that this kind of analysis (an information theoretic argument using Kolmogorov complexity to bound convergence) should be more widely applicable.

The Moser algorithm

We are given an exact k-CNF formula F with m variables such that for every clause C, at most $r \leq 2^k/8$ other clauses share a variable with C.

Algorithm for finding a satisfying truth assignment

Let τ be a random assignment
Procedure SOLVE
  While there is clause C not satisfied
    Let C be the lexicographically first such clause and Call FIX(C)
  End While

Procedure FIX(C)
  Randomly set all the variables occuring in C
  While there is a neighbouring unsatisfied clause D
    Let D be the lexicographically first such clause and Call FIX(D)
  End While

Sketch of Moser algorithm analysis

- Suppose the algorithm makes at least s recursive calls to FIX. Then $n + s * k$ random bits describes the algorithm computation up to the s^{th} call at which time we have some true assignment τ'.
- That is, the computation (if it halts in s calls is described by the n bits to describe the initial τ and the k bits for each of the s calls to FIX.
- Using Kolmogorov complexity, we state the fact that most random strings cannot be compressed.
- Now we say that r is sufficiently small if $k - \log r - c > 0$ for some constant c, Then the main idea is to describe these $n + s * k$ bits in a compressed way if s is large enough and r is small enough.

Moser proof continued

- Claim: Any C that is satisfied before Fix(C) is called in SOLVE remains satisfied.
- Claim: Working backwards from τ' we can recover the original (uniformly random) $n + s * k$ bits using $n + m \log m + s(\log r + c)$ bits which is possible if we know the clauses being fixed since then we know which k bits are being flipped. That is, we have n for τ', $m \log m$ for calls to FIX in SOLVE and $\log r + c$ for each recursive call where the constant c is used to indicate the end of a recursive call.
- By the basic fact of Kolmogorov complexity we must have $n + m \log m + s(logr + c) \geq n + s * k$ or equivalently $s(k - \log r - c) \leq m \log m$ which in turn implies $r < 2^{k-c}$ in order for s to be positive so that $s = O(m \log m)$.
- Together with G. Tardos, the bound is shown to match the bound of the Lovász local Lemma (LLL) and that the algorithm can be de-randomized.