- Consider as an oracle an extremely large book of randomly generated numbers. This oracle could be used to simulate any probabilistic algorithm, so $P = BPP$ relative to this oracle. On the other hand, if one assigns the task to determine whether a given string of numbers exists in some range in the book, this question is in NP but not in P.
- Another example of an oracle would be an extremely large book, in which most of the pages contained the answer to the problem at hand, but for which the n^{th} page was blank for every natural number n that could be quickly created by any short deterministic algorithm. This type of oracle could be used to create a scenario in which $P \neq BPP$ and $P \neq NP$.
- A third example, this time of an *advice function* rather than an oracle, would be if the proctor wrote a long random string on the board before starting the exam (with the length of the string depending on the length of the exam). This can be used to show the inclusion $BPP \subset P/poly$.

By using written oracles instead of computer oracles, it also became more obvious that the oracles were non-interactive (i.e. subsequent responses by the oracle did not depend on earlier queries).

3.10. Moser's entropy compression argument

There are many situations in combinatorics in which one is running some sort of iteration algorithm to continually "improve" some object A; each loop of the algorithm replaces A with some better version A' of itself, until some desired property of A is attained and the algorithm halts. In order for such arguments to yield a useful conclusion, it is often necessary that the algorithm halts in a finite amount of time, or (even better), in a bounded amount of time[10].

[10] In general, one cannot use infinitary iteration tools, such as *transfinite induction* or *Zorn's lemma* (Section 2.4), in combinatorial settings, because the iteration processes used to improve some target object A often degrade some other finitary quantity B in the process, and an infinite iteration would then have the undesirable effect of making B infinite.

A basic strategy to ensure termination of an algorithm is to exploit a *monotonicity property*, or more precisely to show that some key quantity keeps increasing (or keeps decreasing) with each loop of the algorithm, while simultaneously staying bounded. (Or, as the economist Herbert Stein was fond of saying, "If something cannot go on forever, it must stop.")

Here are four common flavours of this monotonicity strategy:

- The *mass increment argument*. This is perhaps the most familiar way to ensure termination: make each improved object A' "heavier" than the previous one A by some non-trivial amount (e.g. by ensuring that the cardinality of A' is strictly greater than that of A, thus $|A'| \geq |A| + 1$). Dually, one can try to force the amount of "mass" remaining "outside" of A in some sense to decrease at every stage of the iteration. If there is a good upper bound on the "mass" of A that stays essentially fixed throughout the iteration process, and a lower bound on the mass increment at each stage, then the argument terminates. Many "greedy algorithm" arguments are of this type. The proof of the Hahn decomposition theorem (Theorem 1.2.2) also falls into this category. The general strategy here is to keep looking for useful pieces of mass outside of A, and add them to A to form A', thus exploiting the additivity properties of mass. Eventually no further usable mass remains to be added (i.e. A is *maximal* in some L^1 sense), and this should force some desirable property on A.
- The *density increment argument*. This is a variant of the mass increment argument, in which one increments the "density" of A rather than the "mass". For instance, A might be contained in some ambient space P, and one seeks to improve A to A' (and P to P') in such a way that the density of the new object in the new ambient space is better than that of the previous object (e.g. $|A'|/|P'| \geq |A|/|P| + c$ for some $c > 0$). On the other hand, the density of A is clearly bounded above by 1. As long as one has a sufficiently good lower bound on the density increment at each stage, one

can conclude an upper bound on the number of iterations in the algorithm. The prototypical example of this is Roth's proof of his theorem[**Ro1953**] that every set of integers of positive upper density contains an arithmetic progression of length three. The general strategy here is to keep looking for useful density fluctuations inside A, and then "zoom in" to a region of increased density by reducing A and P appropriately. Eventually no further usable density fluctuation remains (i.e. A is *uniformly distributed*), and this should force some desirable property on A.

- The *energy increment argument*. This is an "L^2" analogue of the "L^1"-based mass increment argument (or the "L^∞"-based density increment argument), in which one seeks to increments the amount of "energy" that A captures from some reference object X, or (equivalently) to decrement the amount of energy of X which is still "orthogonal" to A. Here A and X are related somehow to a Hilbert space, and the energy involves the norm on that space. A classic example of this type of argument is the existence of orthogonal projections onto closed subspaces of a Hilbert space; this leads among other things to the construction of *conditional expectation* in measure theory, which then underlies a number of arguments in ergodic theory, as discussed for instance in Section 2.8 of *Poincaré's Legacies, Vol. I.* Another basic example is the standard proof of the *Szemerédi regularity lemma* (where the "energy" is often referred to as the "index"). These examples are related; see Section 4.2 for further discussion. The general strategy here is to keep looking for useful pieces of energy orthogonal to A, and add them to A to form A', thus exploiting square-additivity properties of energy, such as Pythagoras' theorem. Eventually, no further usable energy outside of A remains to be added (i.e. A is *maximal* in some L^2 sense), and this should force some desirable property on A.

- The *rank reduction argument*. Here, one seeks to make each new object A' to have a lower "rank", "dimension", or "order" than the previous one. A classic example here is the proof of the linear algebra fact that given any finite set of vectors, there exists a linearly independent subset which spans the same subspace; the proof of the more general *Steinitz exchange lemma* is in the same spirit. The general strategy here is to keep looking for "collisions" or "dependencies" within A, and use them to collapse A to an object A' of lower rank. Eventually, no further usable collisions within A remain, and this should force some desirable property on A.

Much of my own work in additive combinatorics relies heavily on at least one of these types of arguments (and, in some cases, on a nested combination of two or more of them). Many arguments in nonlinear partial differential equations also have a similar flavour, relying on various *monotonicity formulae* for solutions to such equations, though the objective in PDE is usually slightly different, in that one wants to keep control of a solution as one approaches a singularity (or as some time or space coordinate goes off to infinity), rather than to ensure termination of an algorithm. (On the other hand, many arguments in the theory of *concentration compactness*, which is used heavily in PDE, does have the same algorithm-terminating flavour as the combinatorial arguments; see Section 2.1 of *Structure and Randomness* for more discussion.)

Recently, a new species of monotonicity argument was introduced by Moser[**Mo2009**], as the primary tool in his elegant new proof of the *Lovász local lemma*. This argument could be dubbed an *entropy compression argument*, and only applies to *probabilistic algorithms* which require a certain collection R of random "bits" or other random choices as part of the input, thus each loop of the algorithm takes an object A (which may also have been generated randomly) and some portion of the random string R to (deterministically) create a better object A' (and a shorter random string R', formed by throwing away those bits of R that were used in the loop). The key point is to design the algorithm to be partially *reversible*, in the sense that given A' and

R' and some additional data H' that logs the cumulative *history* of the algorithm up to this point, one can reconstruct A together with the remaining portion R not already contained in R'. Thus, each stage of the argument *compresses* the information-theoretic content of the string $A + R$ into the string $A' + R' + H'$ in a lossless fashion. However, a random variable such as $A + R$ cannot be compressed losslessly into a string of expected size smaller than the *Shannon entropy* of that variable. Thus, if one has a good lower bound on the entropy of $A + R$, and if the length of $A' + R' + H'$ is significantly less than that of $A + R$ (i.e. we need the marginal growth in the length of the history file H' per iteration to be less than the marginal amount of randomness used per iteration), then there is a limit as to how many times the algorithm can be run, much as there is a limit as to how many times a random data file can be compressed before no further length reduction occurs.

It is interesting to compare this method with the ones discussed earlier. In the previous methods, the failure of the algorithm to halt led to a new iteration of the object A which was "heavier", "denser", captured more "energy", or "lower rank" than the previous instance of A. Here, the failure of the algorithm to halt leads to new information that can be used to "compress" A (or more precisely, the full state $A + R$) into a smaller amount of space. I don't know yet of any application of this new type of termination strategy to the fields I work in, but one could imagine that it could eventually be of use (perhaps to show that solutions to PDE with sufficiently "random" initial data can avoid singularity formation?), so I thought I would discuss (a special case of) it here.

Rather than deal with the Lovász local lemma in full generality, I will work with a special case of this lemma involving the *k-satisfiability problem* (in *conjunctive normal form*). Here, one is given a set of *boolean variables* $x_1, \ldots, x_n$ together with their negations $\neg x_1, \ldots, \neg x_n$; we refer to the $2n$ variables and their negations collectively as *literals*. We fix an integer $k \geq 2$, and define a (length k) *clause* to be a *disjunction* of k literals, for instance

$$x_3 \vee \neg x_5 \vee x_9$$

is a clause of length three, which is true unless x_3 is false, x_5 is true, and x_9 is false. We define the *support* of a clause to be the set of variables that are involved in the clause, thus for instance $x_3 \vee \neg x_5 \vee x_9$ has support $\{x_3, x_5, x_9\}$. To avoid degeneracy we assume that no clause uses a variable more than once (or equivalently, all supports have cardinality exactly k), thus for instance we do not consider $x_3 \vee x_3 \vee x_9$ or $x_3 \vee \neg x_3 \vee x_9$ to be clauses.

Note that the failure of a clause reveals complete information about all k of the boolean variables in the support; this will be an important fact later on.

The *k-satisfiability problem* is the following: given a set S of clauses of length k involving n boolean variables $x_1, \ldots, x_n$, is there a way to assign truth values to each of the $x_1, \ldots, x_n$, so that all of the clauses are simultaneously satisfied?

For general S, this problem is easy for $k = 2$ (essentially equivalent to the problem of 2-colouring a graph), but NP-complete for $k \geq 3$ (this is the famous *Cook-Levin theorem*). But the problem becomes simpler if one makes some more assumptions on the set S of clauses. For instance, if the clauses in S have disjoint supports, then they can be satisfied independently of each other, and so one easily has a positive answer to the satisfiability problem in this case. (Indeed, one only needs each clause in S to have *one* variable in its support that is disjoint from all the other supports in order to make this argument work.)

Now suppose that the clauses S are not completely disjoint, but have a limited amount of overlap; thus *most* clauses in S have disjoint supports, but not all. With too much overlap, of course, one expects satisfability to fail (e.g. if S is the set of *all* length k clauses). But with a sufficiently small amount of overlap, one still has satisfiability:

Theorem 3.10.1 (Lovász local lemma, special case)**.** *Suppose that S is a set of length k clauses, such that the support of each clause s in S intersects at most 2^{k-C} supports of clauses in S (including s itself), where C is a sufficiently large absolute constant. Then the clauses in S are simultaneously satisfiable.*

One of the reasons that this result is powerful is that the bounds here are uniform in the number n of variables. Apart from the loss of C, this result is sharp; consider for instance the set S of all 2^k clauses with support $\{x_1, \ldots, x_k\}$, which is clearly unsatisfiable.

The standard proof of this theorem proceeds by assigning each of the n boolean variables $x_1, \ldots, x_n$ a truth value $a_1, \ldots, a_n \in \{\text{true}, \text{false}\}$ independently at random (with each truth value occurring with an equal probability of $1/2$); then each of the clauses in S has a positive zero probability of holding (in fact, the probability is $1 - 2^{-k}$). Furthermore, if E_s denotes the event that a clause $s \in S$ is satisfied, then the E_s are mostly independent of each other; indeed, each event E_s is independent of all but most 2^{k-C} other events $E_{s'}$. Applying the *Lovász local lemma*, one concludes that the E_s simultaneously hold with positive probability (if C is a little bit larger than $\log_2 e$), and the claim follows.

The textbook proof of the Lovász local lemma is short but non-constructive; in particular, it does not easily offer any quick way to compute an actual satisfying assignment for $x_1, \ldots, x_n$, only saying that such an assignment exists. Moser's argument, by contrast, gives a simple and natural algorithm to locate such an assignment (and thus prove Theorem 3.10.1). (The constant C becomes 3 rather than $\log_2 e$, although the $\log_2 e$ bound has since been recovered in a paper of Moser and Tardos.)

As with the usual proof, one begins by randomly assigning truth values $a_1, \ldots, a_n \in \{\text{true}, \text{false}\}$ to $x_1, \ldots, x_n$; call this random assignment $A = (a_1, \ldots, a_n)$. If A satisfied all the clauses in S, we would be done. However, it is likely that there will be some non-empty subset T of clauses in S which are not satisfied by A.

We would now like to modify A in such a manner to reduce the number $|T|$ of violated clauses. If, for instance, we could always find a modification A' of A whose set T' of violated clauses was strictly smaller than T (assuming of course that T is non-empty), then we could iterate and be done (this is basically a mass decrement argument). One obvious way to try to achieve this is to pick a clause s in T that is violated by A, and modify the values of A on the support

of s to create a modified set A' that satisfies s, which is easily accomplished; in fact, any non-trivial modification of A on the support will work here. In order to maximize the amount of entropy in the system (which is what one wants to do for an entropy compression argument), we will choose this modification of A' *randomly*; in particular, we will use k fresh random bits to replace the k bits of A in the support of s. (By doing so, there is a small probability (2^{-k}) that we in fact do not change A at all, but the argument is (very) slightly simpler if we do not bother to try to eliminate this case.)

If all the clauses had disjoint supports, then this strategy would work without difficulty. But when the supports are not disjoint, one has a problem: every time one modifies A to "fix" a clause s by modifying the variables on the support of s, one may cause other clauses s' whose supports overlap those of s to fail, thus potentially increasing the size of T by as much as $2^{k-C} - 1$. One could then try fixing all the clauses which were broken by the first fix, but it appears that the number of clauses needed to repair could grow indefinitely with this procedure, and one might never terminate in a state in which all clauses are simultaneously satisfied.

The key observation of Moser, as alluded earlier, is that each failure of a clause s for an assignment A reveals k bits of information about A, namely that the exact values that A assigns to the support of s. The plan is then to use each failure of a clause as a part of a *compression protocol* that compresses A (plus some other data) losslessly into a smaller amount of space. A crucial point is that at each stage of the process, the clause one is trying to fix is almost always going to be one that overlapped the clause that one had just previously fixed. Thus the total number of possibilities for each clause, given the previous clauses, is basically 2^{k-C}, which requires only $k - C$ bits of storage, compared with the k bits of entropy that have been eliminated. This is what is going to force the algorithm to terminate in finite time (with positive probability).

Let's make the details more precise. We will need the following objects:

- A truth assignment A of n truth values $a_1, \ldots, a_n$, which is initially assigned randomly, but which will be modified as the algorithm progresses;
- A long random string R of bits, from which we will make future random choices, with each random bit being removed from R as it is read.

We also need a recursive algorithm Fix(s), which modifies the string A to satisfy a clause s in S (and, as a bonus, may also make A obey some other clauses in S that it did not previously satisfy). It is defined recursively:

- Step 1. If A already satisfies s, do nothing (i.e. leave A unchanged).
- Step 2. Otherwise, read off k random bits from R (thus shortening R by k bits), and use these to replace the k bits of A on the support of s in the obvious manner (ordering the support of s by some fixed ordering, and assigning the j^{th} bit from R to the j^{th} variable in the support for $1 \leq j \leq k$).
- Step 3. Next, find all the clauses s' in S whose supports intersect s, and which A now violates; this is a collection of at most 2^{k-C} clauses, possibly including s itself. Order these clauses s' in some arbitrary fashion, and then apply Fix(s') to each such clause in turn. (Thus the original algorithm Fix(s) is put "on hold" on some CPU stack while all the child processes Fix(s') are executed; once all of the child processes are complete, Fix(s) then terminates also.)

An easy induction shows that if Fix(s) terminates, then the resulting modification of A will satisfy s; and furthermore, any other clause s' in S which was already satisfied by A before Fix(s) was called, will continue to be satisfied by A after Fix(s) is called. Thus, Fix(s) can only serve to decrease the number of unsatisfied clauses T in S, and so one can fix all the clauses by calling Fix(s) once for each clause in T - provided that these algorithms all terminate.

Each time Step 2 of the Fix algorithm is called, the assignment A changes to a new assignment A', and the random string R changes to a shorter string R'. Is this process reversible? Yes - provided that one

knows what clause s was being fixed by this instance of the algorithm. Indeed, if s, A', R' are known, then A can be recovered by changing the assignment of A' on the support of s to the only set of choices that violates s, while R can be recovered from R' by appending to R' the bits of A on the support of s.

This type of reversibility does not seem very useful for an entropy compression argument, because while R' is shorter than R by k bits, it requires about $\log |S|$ bits to store the clause s. So the map $A+R \mapsto A' + R' + s$ is only a compression if $\log |S| < k$, which is not what is being assumed here (and in any case the satisfiability of S in the case $\log |S| < k$ is trivial from the union bound).

The key trick is that while it does indeed take $\log |S|$ bits to store any given clause s, there is an economy of scale: after many recursive applications of the fix algorithm, the *marginal* amount of bits needed to store s drops to merely $k - C + O(1)$, which is less than k if C is large enough, and which will therefore make the entropy compression argument work.

Let's see why this is the case. Observe that the clauses s for which the above algorithm $\mathrm{Fix}(s)$ is called come in two categories. Firstly, there are those s which came from the original list T of failed clauses. Each of these will require $O(\log |S|)$ bits to store - but there are only $|T|$ of them. Since $|T| \leq |S|$, the net amount of storage space required for these clauses is $O(|S| \log |S|)$ at most. Actually, one can just store the subset T of S using $|S|$ bits (one for each element of S, to record whether it lies in T or not).

Of more interest is the other category of clauses s, in which $\mathrm{Fix}(s)$ is called recursively from some previously invoked call $\mathrm{Fix}(s')$ to the fix algorithm. But then s is one of the at most 2^{k-C} clauses in S whose support intersects that of s'. Thus one can encode s using s' and a number between 1 and 2^{k-C}, representing the position of s (with respect to some arbitrarily chosen fixed ordering of S) in the list of all clauses in S whose supports intersect that of s'. Let us call this number the *index* of the call $\mathrm{Fix}(s)$.

Now imagine that while the Fix routine is called, a running *log file* (or history) H of the routine is kept, which records s each time one of the original $|T|$ calls $\mathrm{Fix}(s)$ with $s \in T$ is invoked, and also

records the index of any other call Fix(s) made during the recursive procedure. Finally, we assume that this log file records a termination symbol whenever a Fix routine terminates. By performing a *stack trace*, one sees that whenever a Fix routine is called, the clause s that is being repaired by that routine can be deduced from an inspection of the log file H up to that point.

As a consequence, at any intermediate stage in the process of all these fix calls, the original state $A + R$ of the assignment and the random string of bits can be deduced from the current state $A' + R'$ of these objects, plus the history H' up to that point.

Now suppose for contradiction that S is not satisfiable; thus the stack of fix calls can never completely terminate. We trace through this stack for M steps, where M is some large number to be chosen later. After these steps, the random string R has shortened by an amount of Mk; if we set R to initially have length Mk, then the string is now completely empty, $R' = \emptyset$. On the other hand, the history H' has size at most $O(|S|) + M(k - C + O(1))$, since it takes $|S|$ bits to store the initial clauses in T, $O(|S|) + O(M)$ bits to record all the instances when Step 1 occurs, and every subsequent call to Fix generates a $k - C$-bit number, plus possibly a termination symbol of size $O(1)$. Thus we have a lossless compression algorithm $A + R \mapsto A' + H'$ from $n + Mk$ completely random bits to $n + O(|S|) + M(k-C+O(1))$ bits (recall that A and R were chosen randomly, and independently of each other). But since $n + Mk$ random bits cannot be compressed losslessly into any smaller space, we have the entropy bound

$$n + O(|S|) + M(k - C + O(1)) \geq n + Mk \tag{3.49}$$

which leads to a contradiction if M is large enough (and if C is larger than an absolute constant). This proves Theorem 3.10.1.

Remark 3.10.2. Observe that the above argument in fact gives an explicit bound on M, and with a small bit of additional effort, it can be converted into a probabilistic algorithm that (with high probability) computes a satisfying assignment for S in time polynomial in $|S|$ and n.

Remark 3.10.3. One can replace the usage of randomness and Shannon entropy in the above argument with *Kolmogorov complexity* instead; thus, one sets $A+R$ to be a string of $n+Mk$ bits which cannot be computed by any algorithm of length $n + O(|S| \log |S|) + M(k - C + O(1))$, the existence of which is guaranteed as soon as (3.49) is violated; the proof now becomes deterministic, except of course for the problem of building the high-complexity string, which by their definition can only be constructed quickly by probabilistic methods.

Notes. This article first appeared at `terrytao.wordpress.com/2009/08/05`, but is based on an earlier blog post by Lance Fortnow at

`blog.computationalcomplexity.org/2009/06`.

Thanks to harrison, Heinrich, nh, and anonymous commenters for corrections.

There was some discussion online about the tightness of bounds in the argument.

3.11. The AKS primality test

The *Agrawal-Kayal-Saxena (AKS) primality test*, discovered in 2002, is the first provably deterministic algorithm to determine the primality of a given number with a run time which is guaranteed to be polynomial in the number of digits, thus, given a large number n, the algorithm will correctly determine whether that number is prime or not in time $O(\log^{O(1)} n)$. (Many previous primality testing algorithms existed, but they were either probabilistic in nature, had a running time slower than polynomial, or the correctness could not be guaranteed without additional hypotheses such as GRH.)

In this article I sketch the details of the test (and the proof that it works) here. (Of course, full details can be found in the original paper[**AgKaSa2004**], which is nine pages in length and almost entirely elementary in nature.) It relies on polynomial identities that are true modulo n when n is prime, but cannot hold for n non-prime as they would generate a large number of additional polynomial identities, eventually violating the *factor theorem* (which asserts that a polynomial identity of degree at most d can be obeyed by at most d