Does randomness solve problems?

Russell Impagliazzo
UCSD

Well, what’s the answer?

YES, randomness solves problems

BUT the randomness (probably) doesn’t have to be random to solve problems.

What do we mean by a problem? What is a solution?

For our purposes, a computational problem has the following form:

Input: Can be represented as a binary sequence x. We define n to be the length of x, the number of bits needed to write x.

Output: also coded as binary sequence y

Correctness: Some mathematical relationship holds between x and y

Algorithm: A series of well defined computational steps that, on input

Randomized algorithms vs. Deterministic algorithms

A randomized algorithm has an instruction: Pick a bit b at random, equally likely to be 0 or 1, as well as all of the standard operations that a deterministic algorithm can perform.

It is equivalent to pick a supply of random bits r=r_1...r_T at the start, and say that r_i will be the result of the i’th call to the ``coin flip’’ operation. So we can think of a randomized algorithm R(x) as a deterministic algorithm A applied to two inputs,

The real input x, and the random input r

R(x)= A(x,r) for a

When would randomness always solve a problem?

We could consider randomness ESSENTIAL to solve a problem in the worst-case

If there were a problem and a randomized algorithm that has high success probability on every instance and is much faster on large inputs than ANY deterministic algorithm for the same problem.

To formalize this, we let BPP be the class of problems solvable by randomized algorithms in time that is polynomial (n, n^2, n^3 etc) in the input size, and gets the right answer with high probability on every instance, and P the similar class for deterministic algorithms.

Randomness helps if BPP is a strict superset of P

When would randomness solve some instances?

Instead of only looking at algorithms that solve all instances of a problem,

We could look at algorithms that solve some instances but not others. We let Promise-BPP be the class of problems and subsets of instances where a randomized algorithm can solve the problem in poly time with high probability of correctness on those instances, and Promise-P the same.

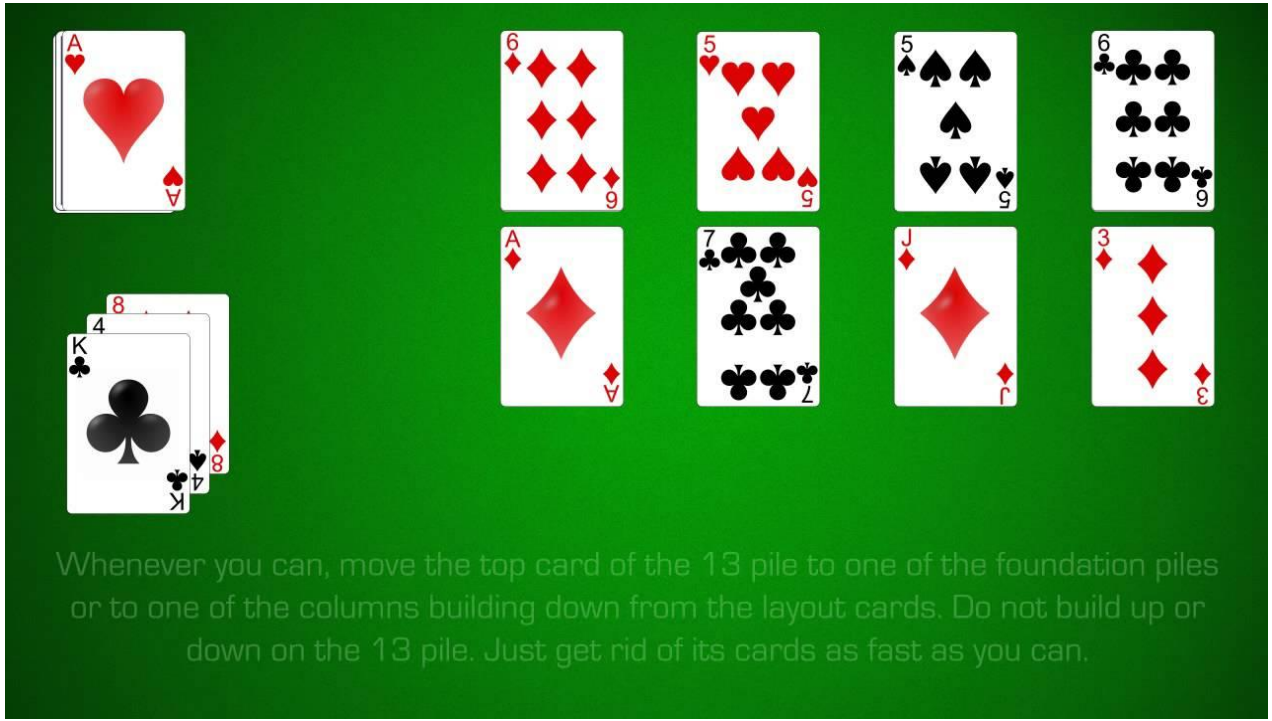
Then Promise-BPP is a strict superset of Promise-P if there are some instances of some problems where randomized algorithms out-perform deterministic ones.

Helping for some instances

If Promise BPP= Promise-P, then BPP=P, but the other direction isn’t known.

Why would anyone think randomness might help?

Stanislaw Ulam was a physicist who worked for the Manhattan Project. While recovering from surgery, he was in the hospital for a number of weeks.



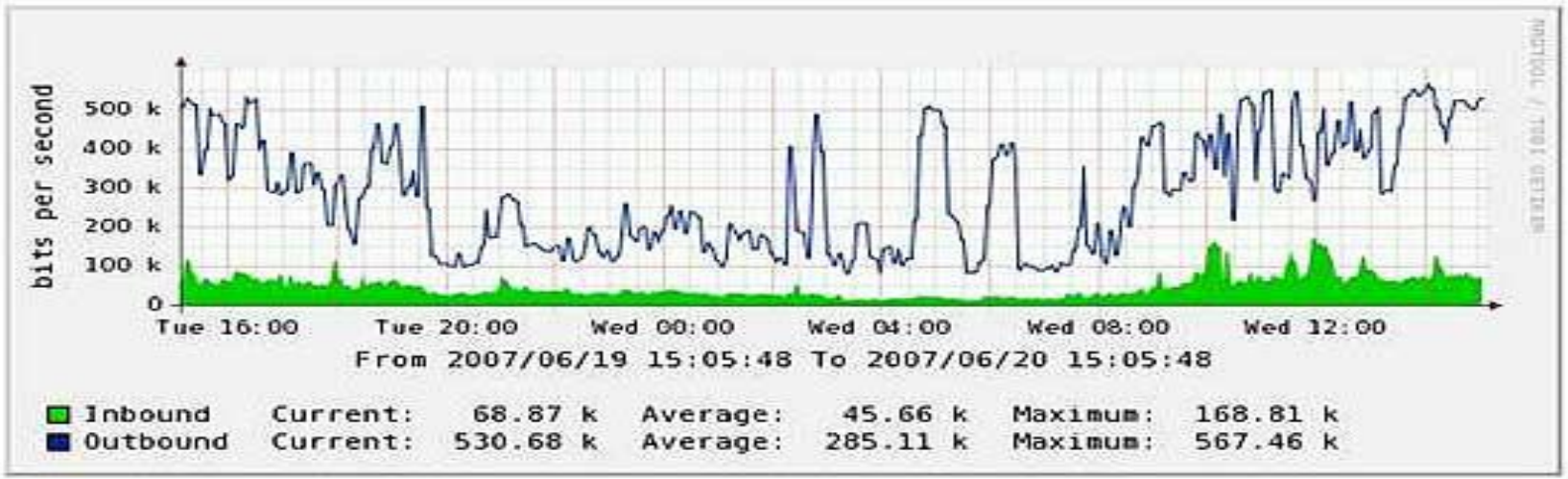
While he was in the hospital he killed time playing canfield, a complex solitaire game

He first tried to figure out the odds of winning canfield mathematically, but got frustrated and decided that it would be easier to simulate thousands of games on a computer. (The Manhattan Project had one of the first general purpose computers.) Later, when the Manhattan Project needed to perform calculations involving huge numbers of possible interactions of atomic particles, he'd remember this idea.

His colleague, Nicholas Metropolis, dubbed this algorithmic technique ``The Monte Carol method'' after an uncle who loved to go to Monte Carlo to gamble.

Monte Carlo integration

One way to view the Monte Carlo technique is as a form of numerical integration for irregular high-dimensional functions. Measuring a bursty function at random times is more accurate than measuring at fixed intervals.



Circuit Approximate Probability Problem

CAPP is a discrete version of the ``bursty function'' problem.

A Boolean circuit is a fixed sequence of Boolean operations (and, or , not) performed on input bits and intermediate gates. For example,

(x and y) or (z and not x) is one Boolean circuit, with three inputs and three operations. Given a circuit, C(x_1,..x_n), CAPP asks for an estimate of

Prob[C(x_1,..x_n)=1] for randomly chosen x_i's, which is accurate to within an additive error of say .1 (The exact number doesn't matter)

Completeness of CAPP

CAPP can be solved with high probability by the randomized algorithm that picks several random x's and outputs the average value of C(x).

CKR observed that CAPP is ``Promise-BPP-complete'', in that we can solve CAPP deterministically if and only if Promise-BPP=Promise-P.

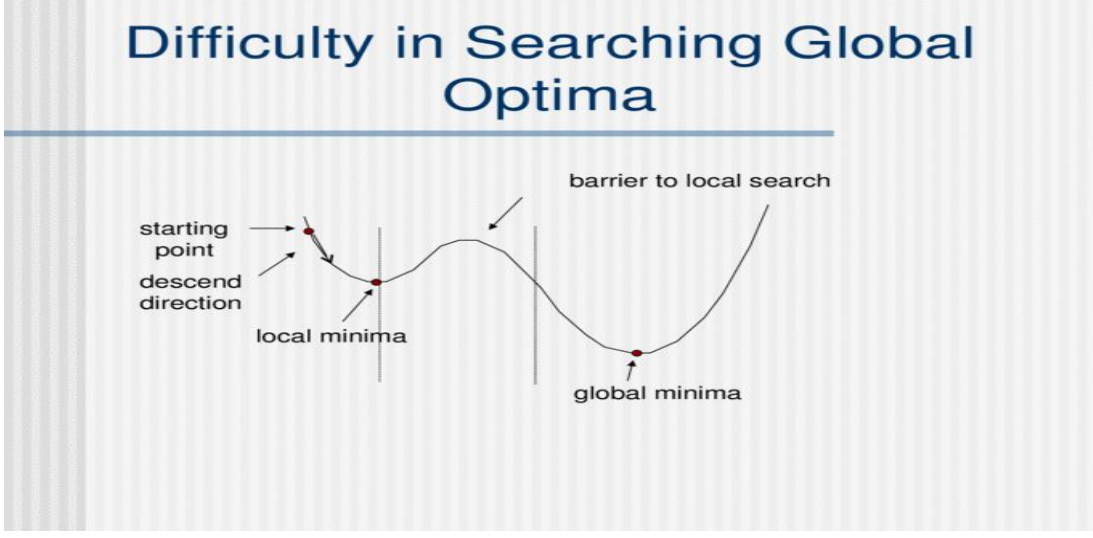
(The idea is to code the behaviour of any probabilistic algorithm on any fixed input as a circuit; estimating the probability this circuit outputs 1 gives the most likely output for the randomized algorithm on this input.)

Metropolis algorithm

Nicholas Metropolis also gave his name (if nothing else) to the Metropolis algorithm (actually due to Arianna and Marshall Rosenbluth, and Augusta and Edward Teller, pictured below). With many variants such as simulated annealing, Metropolis is a highly successful optimization heuristic. Here the goal is to search through a huge number of possible solutions to a problem to find the best one, according to an objective function. Such problems are typically NP-complete.

Why the Metropolis algorithm is useful

The Metropolis algorithm balances greedy optimization, going in directions that improve the objective, with random moves that might make the solution worse, but allow the algorithm to escape local optima that are not global optima. It works well on many but not all instances of optimization problems.



Primality testing

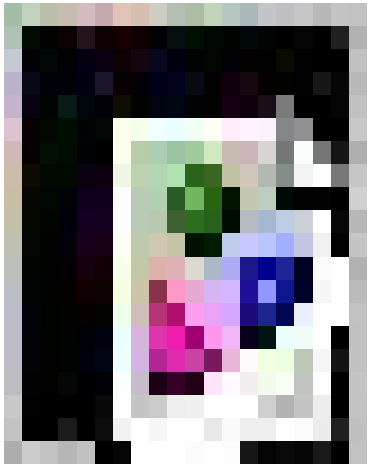
A new flavor of randomized algorithms became prominent in the mid-70's. Testing whether an integer is prime was one of the most important ``unclassified'' problems, and the obvious approach (factoring a composite number) is still believed computationally difficult. But another way to show a number is not prime is to show that the conclusion of Fermat's little theorem, x^{p-1} mod p = x mod p for all primes p.

Randomized primality testing

Miller: ERH implies Primality is deterministic polynomial time decidable

Rabin, Solovay-Strassen: Unconditionally, primality is randomized polynomial time decideable






Derandomized primality testing

But in 2002, Agrawal, Kayal and Saxena ``derandomized'' a related randomized polynomial testing algorithm, showing primality testing is also in P

Polynomial identity testing

Schwartz, Zippel, and DeMillo and Lipton introduced another classical randomized algorithm at the same time as primality testing. While less glamorous, PIT is probably more useful.

Given an equation where both sides are polynomials in several variables (thought of as taking real values), is the equation always true? The equation can be given as an algebraic circuit, like a Boolean circuit, but using addition and multiplication rather than ands and ors.

PIT randomized algorithm

There are two random steps: Plug in random values for the variables, then check to the equation mod a random number (to keep the values of sub-expressions from blowing up.)

While some special cases of PIT have been derandomized (and in fact AKS was based on derandomizing a special case of PIT), there is no non-trivial deterministic algorithm for the general cases.

Random self-reducibility

Another general use of randomness is to make a worst-case instance look like a typical instance. This method is used frequently in computational geometry, where some algorithms break down if say, three points are very close to being on a line. Perturbing the input randomly can put points in general position while preserving relevant features.

Intuitions about when randomness helps

Depending on which type of randomized algorithm people use, their intuitions about when randomness should be helpful vary.

A real enthusiast might think randomness is helpful for all instances of all hard problems. Let's call this position the zealot's view.

Someone who primarily uses random self-reducibility arguments might think that randomness helps make the worst-case look like an easier typical case. We could call this position the averaging view.

More intuitions

Someone using randomness for heuristics might think that randomness is useful for the typical case of hard problems, but not the worst-case. Let's call this the heuristic view.

Someone using randomness for algebraic problems might think that problems where randomness is useful are the highly structured problems, so randomness might be useful for many instances of some problems, but not useful for generic hard problems. Let's call this the structured view.

Finally, someone might think randomness is never actually needed, that we can derandomize every algorithm. Let's call this the skeptical view.

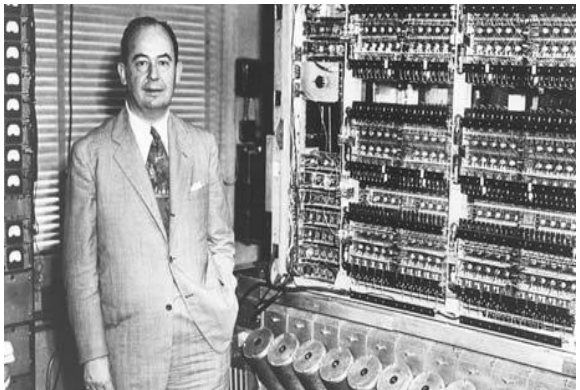
Ruling out the middle positions.

While all but the zealot's position seem plausible to me, we haven't ruled out either of the extreme positions. It is still possible that EXP=BPP, so all exponentially hard problems can be solved using randomness, an extreme zealot position. It is very likely (and I'll say why in a while) that the skeptic's position is true, ie., Promise-BPP= Promise-P. But at least some formalizations of the most moderate positions-- the structuralist and the heuristic -- have been proved false!

Are randomized algorithms randomized?

While randomized algorithms have been used since the dawn of computing, randomness has almost never been used in algorithms.

Instead, pseudo-random sequences, series of bits generated deterministically or from a small ``seed'', have been used to supply the supposedly random choices of the algorithms. This started in the Manhattan Project, with von Neumann suggesting a pseudo-random generator based on the middle digits of the squares of numbers.



What makes a pseudo-random generator good?

While identifying ``bad'' properties of pseudo-random sequences is an interesting combinatorial problem, it isn't clear when a generator is strong enough to fool any algorithm. Yao, building on work by Shamir, Blum, and Micali, gave the first complexity-theoretic definition of strong PRG, and designed one based on cryptographic hardness assumptions.

Yao’s criteria

Yao’s criteria is similar to ``taste tests’’, or the Turing test. If you can’t tell two things apart, then whatever one is useful for the other is useful for.

Say you have a pseudo-random generator G that takes a very small seed s and produces a pseudo-random string z=G(s). Then if there’s no circuit C so that |Prob[C(G(s)=1] - Prob[C(r)=1]| is non-negligible, then it immediately follows that G can be used to solve CAPP and hence every Promise-BPP problem.

Non-constructive existence proofs

Another use of randomness in TCS is in non-constructive proofs that some objects exist. This probabilistic method was developed in combinatorics by Paul Erdos.

In particular, Riordan and Shannon used it to prove that most Boolean functions require very large circuits to compute, exponential in the input size, by comparing the number of such functions to the number of small circuits.

Explicit hard functions

While most functions are exponentially hard for circuits, until last year the best lower bound we had for a function in EXP was a 1984 3n lower bound by N. Blum.

In a dramatic breakthrough, the new lower bound proved by GHKK'16 is

(3.012) n.

While particular functions have been shown to have large circuit complexity, such functions themselves have high complexity. The smallest classes we have lower bounds for are in the second level of the exponential hierarchy, stronger than NEXP, the analog of NP at the exponential level.

Derandomization and explicit hard functions

Since most functions are hard, picking one at random would be a method to find a hard function, whereas a similar deterministic method for finding a hard function would put that function in EXP.

So we can view the quest for a constructive hard function as a derandomization question. Over the years, we've discovered tighter and tighter formal connections between these problems, to the extent that progress on one inevitably entails progress on the other.

Hardness vs. Randomness

Nisan and Wigderson made the break-through connection in one direction.

As later improved by BFNW and IW, if there are any functions in time

2^{O(n)} that require exponential circuit size, then we can use them to construct strong pseudo-random generators in the Yao sense, and Promise-BPP=PromiseP.

So if the skeptic viewpoint is wrong, then circuits can speed up EVERY exponentially hard problem! In other words, the ``off-line'' difficulty of problems, where you are allowed to spend lots of time to optimize your algorithm for a particular size, is always much better than ``on-line'' computation.

Average-case simulations of probabilistic algorithms

Combining the NW hardness to randomness paradigm with random self-reducibility ideas and ``program correctness checking'', IW show that randomized algorithms are either super-powerful, or can be derandomized on ``typical instances''.

Either EXP=BPP (extreme zealot position) or there's a sub-exponential time deterministic simulation of any randomized algorithm that works on ``typical instances''. (In fact, it is computationally hard to find instances on which the simulation fails!).

Interpretation

To me, this result rules out the strong versions of the heuristic and structuralist view. Both possibilities apply to any type of problem, structured or not. Either randomness helps every hard problem, or it can be removed for any problem on all but hard to find inputs.

If randomness is needed, it is either needed everywhere, or only on extremely rare instances. So it is not really possible that randomness only helps typical instances, as the heuristic view would suggest.

So those are two very plausible views that have been eliminated, whereas some rather implausible ones remain.

Implications of derandomization

In the other direction, using some rather strange indirect arguments, IKW show that derandomization of CAPP implies that problems in NEXP require super-polynomial circuit size. So any progress on derandomizing Promise-BPP entails progress on the circuit lower bound question, and vice versa.

Consequences for derandomizing PIT

Perhaps even more surprising, KI, CIKK show that derandomizing the classic PIT algorithm would imply a similar arithmetic circuit lower bound. This explains why progress on derandomizing PIT has been much slower than derandomizing primality testing. (There are also results in the other direction; algebraic circuit lower bounds would partially derandomize important cases of PIT) So again, derandomization is inherently tied to proving lower bounds.

Conclusion

While we don’t absolutely know that randomized algorithms can be all derandomized, if not, the consequences are incredibly surprising (to me, at least).

But to make substantial improvements in either general derandomization or circuit lower bounds, we have to make substantial improvements in both!

This is both a challenge and an opportunity.