10.2 Probabilistic Algorithms

Adam DeJans

December 6, 2017

Outline

What is a Probabilistic Algorithm?

Probabilistic Turing Machines

- Probability of a branch
- Probability that M accepts w
- M recognizes language A with error probability ϵ

BPP

- Bounded Probabilistic Polynomial-time
- Why 1/3?
- An Illustration
- Relating the Illustration to Turing-Machines

The Amplification Lemma

Proof Idea

Complexity Classes

Interesting Note

Famous Problem in BPP

Primality

Algorithm to solve $PRIMES$

If p is an odd prime number, P($PRIME$ accepts p) $= 1$.

If p is not a prime number, then P(accepts p) $\leq 2^{-k}$

What is a Probabilistic Algorithm?

- Also known as Randomized Algorithms
- A **probabilistic algorithm** is an algorithm designed to use the outcome of a random process.
  - In other words, part of the logic for the algorithm uses randomness
  - Typically, such an algorithm would contain an instruction to flip a coin and the result of that coin flip would influence the algorithms subsequent execution and output.
- Often the algorithm has access to a pseudo-random number generator

Probabilistic Algorithms

- The algorithm uses random bits to help make choices (in hope of getting better performance)

- Certain types of problems seem to be more easily solvable by probabilistic algorithms than by deterministic algorithms.

Probabilistic Turing Machines

We set up a model of computation - probabilistic Turing machines - which allow us to talk about complexity classes for probabilistic algorithms.

Definition: A **probabilistic Turing machine** is a type of non-deterministic Turing machine where we always have 1 or 2 possible moves at each point. If there is 1 move, we call it a deterministic move, and if there are 2 moves, we call it a coin toss. We have accept or reject possibilities as before.

We consider machines which run in $poly(n)$ on all branches of its computation.

Probability of a branch

For a branch b of M on w, we say the **probability** of b is

$$P(b) = 2^{-k}$$

where k is the number of coin-flip steps that occur on branch b.

Probability that M accepts w

We define the probability that M accepts w to be

$$P(M \text{ accepts } w) = \sum_{b \text{ is an accepting branch}} P(b)$$

This is indeed the obvious definition: what is the probability of following b if we actually tossed coins at each coin toss step? At each step there is $\frac{1}{2}$ chance of going off b.

We see that the machine will accept the input with certain probability. Accept some with 99%, 0%, 2%, 50%. We want to say that the probability does the right thing on every input, but with small probability of failing (the error).

M recognizes language A with error probability ϵ

For a language A, we say that probabilistic Turing-machine M **decides** A **with error probability** ϵ if for $w \in A$,

$$P(M \text{ accepts } w) \geq 1 - \epsilon.$$

If $w \notin A$, then

$$P(M \text{ rejects } w) \geq 1 - \epsilon.$$

(i.e., it accepts with small probability, $P(M \text{ accepts } w \leq \epsilon.)$

For instance, if a machine accepts with 1% error, then it accepts things in the language with 99% probability.

BPP

There is a forbidden behavior to probabilistic Turing-machines: the machine is not allowed to be unsure. For instance, accept/reject an input with probability $\frac{1}{2}$. It has to lean overwhelmingly one way or another way. How overwhelming do you want to be? We have a parameter k, which can apply universally to adjust the error possibility. By repeating an algorithm many times, we can decrease error (*Amplification Lemma*).

Bounded Probabilistic Polynomial-time

Bounded Probabilistic Polynomial-time (BPP) is defined to be the class of languages that are recognized by probabilistic polynomial time Turing machines with an error probability of $\frac{1}{3}$.

A problem in BPP:

- Can be solved by an algorithm that is allowed to make random decisions (coin-flips)
- Guaranteed to run in polynomial time
- On a given run of the algorithm, it has (at most) $\frac{1}{3}$ probability of giving an incorrect answer
- These algorithms are known as probabilistic algorithms

Why 1/3?

When we discuss bounded in BPP, bounded means bounded below $\frac{1}{2}$. The $\frac{1}{3}$ looks like an arbitrary number, and it is! In fact, it can be any constant between 0 and $\frac{1}{2}$ (as long as it is independent of the input).

Why?

- If the algorithm is run many times, the probability of the probabilistic Turing-machine being wrong the majority of the time decreases exponentially
- Therefore, these kinds of algorithms can become mroe accurate by running it several times (and then taking the majority vote of the results)

An Illustration

- Let the error probability be $\frac{1}{3}$
- We have a box containing many red and blue balls
  - $2/3$ of the balls are one color
  - $1/3$ of the balls are the other color
  - We do not know which color is $2/3$ and which color is $1/3$
- To find out, we start taking samples at random and keep track of which color ball we pulled from the box
- The color that comes up most frequently during a large sampling will most likely be the majority color originally in the box

Relating the Illustration to Turing-Machines

- The blue and red balls correspond to branches in a probabilistic (polynomial time) Turing-machine. Lets call it M_1
- We can assign each color:
  - Red $=$ accepting
  - Blue $=$ rejecting
- The sampling can be done by running M_1 using another probabilistic Turing-machine, denote this as M_2, with a better error probability
- M_2's error probability is exponentially small if it runs M_2 a polynomial number of times and outputs the result that occurs most often.

How do we guarantee that we can find such a M_2? This leads us to the *Amplification Lemma*.

The Amplification Lemma

Let ϵ be a fixed constant strictly between 0 and $\frac{1}{2}$. Then for any polynomial $poly(n)$ a probabilistic polynomial time Turing machine M that operates with error probability ϵ has an equivalent probabilistic polynomial time Turing machine M' that operates with an error probability of $2^{-poly(n)}$.

Proof Idea

M' on w runs M on w $poly(n)$ times and outputs the majority answer.

Complexity Classes

Since a P-algorithm gives the correct answer with error of 0% it is clear that $P \subseteq BPP$.

The relationship between BPP and NP is unknown: it is not known if BPP is a subset of NP, or if NP is a subset of BPP, or if they are incomparable. BPP is known to be a subset of PSCPACE.

It is however unknown wheter vice versa also holds.

It is also known that either P=BPP or P$\neq$NP or both.

Interesting Note

Many problems were known to be in BPP, but not known to be in P. The number of such problems is decreasing. In fact, most people *believe* $P = BPP$, because of pseudorandomness. If there was some way to compute a value of the coin toss in a way that would act as good as a truly random coin toss, with a bit more work one could prove $P = BPP$. A lot of progress has been made constructing pseudo-random generators, but they require assumptions such as $P \neq NP$.

Famous Problem in BPP

For a long time, one of the most famous problems that was known to be in BPP but not known to be in P was PRIMES. However, in 2002, Agrawal and his students showed that PRIMES is in P.

Let's take a look at the PRIMES problem.

Primality

We'll use primality testing as an example of a probabilistic algorithm. Let

$$PRIMES = \{p \mid p \text{ is a prime number in binary}\}.$$

We'll give a probabilistic, polynomial-time algorithm for PRIMES. We'll sketch the idea, without going through the details. Recall, our algorithm is probabilistic in the sense that for each input the running time is polynomial, but there is a small chance that it will be wrong.

Theorem (Fermat's Little Theorem)
For any prime p and a relatively prime to p,

$$a^{p-1} \equiv 1 \quad (mod\ p).$$

Think of the preceding theorem as providing a type of "test" for primality called a *Fermat test*. When we say that p passes the Fermat test at a, we mean that $a^{p-1} \equiv 1\ (mod\ p)$.

$PSEUDOPRIME =$ "On input p:

1. Select $a_1, \ldots, a_k$ randomly in Z_p^+.
2. Compute a_i^{p-1} mod p for each i.
3. If all computed values are 1, *accept*; otherwise, *reject*."

Correctness: If p is pseudoprime, it passes all tests and the algorithm accepts with certainty. If p isn't pseudoprime, it passes at most half of all tests. In that case, it passes each randomly selected test with probability at most $\frac{1}{2}$. The probability that it passes all k randomly selected tests is thus at most 2^{-k}. The algorithm operates in polynomial time because modular exponentiation is computable in polynomial time.

Notice that this does not solve $PRIMES$ as there are numbers, known as *Carmichael numbers*, which are non-primes that pass all Fermat tests, for all values of a.

To convert the above algorithm to one which solves $PRIMES$ we use the underlying principle that the number 1 has exactly two square roots, namely 1 and -1, modulo any prime p. Meanwhile, for many composite numbers (including all Carmichael numbers), the number 1 has four or more square roots.

If a number passes the Fermat test at a, the algorithm finds one of its square roots of 1 at random and determines whether that square root is 1 or -1. If it isn't, we know that the number isn't prime.

We can obtain square roots of 1 if p passes the Fermat test at a because a^{p-1} mod $p = 1$, and so $a^{(p-1)/2}$ mod p is a square root of 1. If that value is still 1, we may repeatedly divide the exponent by 2, so long as the resulting exponent remains an integer, and see whether the first number that is different from 1 is -1 or some other number.

Algorithm to solve $PRIMES$

$PRIME$ = "On input p:

1. If p is even, *accept* if $p = 2$; otherwise, reject.
2. Select $a_1, \ldots, a_k$ randomly in Z_p^+.
3. For each i from 1 to k:
4. Compute a_i^{p-1} mod p and *reject* if different from 1.
5. Let $p - 1 = s \cdot 2^l$ where s is odd.
6. Compute the sequence $a_i^{s \cdot 2^0}, a_i^{s \cdot 2^1}, a_i^{s \cdot 2^2}, \ldots, a_i^{s \cdot 2^l}$
7. If some element of this sequence is not 1, find the last element that is not 1 and *reject* if that element is not -1.

8. All tests have passed at this point, so *accept*."

It's clear that $PRIME$ is correct when the input p is even, so we focus on justifying the correctness when p is odd. We say that a_i is a **(compositeness) witness** if the algorithm rejects at either stage 4 or 7, using a_i.

We use the following three facts:

- Fact 1: Fermats Little Theorem:
- Fact 2: Any non-Carmichael composite number fails at least half of all Fermat tests (for at least half of all values of a).
- Fact 3: For every Carmichael composite n, there is some $b \neq 1, -1$ such that $b^2 = 1 \mod n$ (that is, 1 has a nontrivial square root, mod n). No prime has such a square root.

If p is an odd prime number, P($PRIME$ accepts p) $= 1$.

- Show that, if the algorithm rejects, then p must be composite.
- Reject because of Fermat: Then not prime, by Fact 1 (primes pass).
- Reject because of Carmichael: Then 1 has a nontrivial square root b, mod n, so n isnt prime, by Fact 3.
  - Let b be the last term in the sequence that isnt congruent to 1 mod p.
  - b^2 is the next one, and is 1 mod p, so b is a square root of 1, mod p.

If p is not a prime number, then P(accepts p) $\leq 2^{-k}$

- Suppose p is a composite.
- If p is not a Carmichael number, then at least half of the possible choices of a fail the Fermat test (by Fact 2).
- If p is a Carmichael number, then Fact 3 says that some b fails the Carmichael test (is a nontrivial square root).
- Actually, when we generate b using a as above, at least half of the possible choices of a generate bs that fail the Carmichael test.
- Why: Technical argument, in Sipser, p. 374-375.

[1] [2] [3]

References

[1] Michael Sisper. *Introduction to the Theory of Computation*. PWS Publishing Company, first edition, 1997.

[2] NAME. *Notes on Theory of Computation - Lecture 22*. Publisher, YEAR.

[3] Stephany Coffman-Wolph. *Presentation*. Publisher, 2007.

Questions?