University of California, Los Angeles
CS 281 Computability and Complexity

Instructor: Alexander Sherstov
Scribe: Glenn Sun
Date: November 10, 2020

LECTURE
12

Randomized Complexity Classes

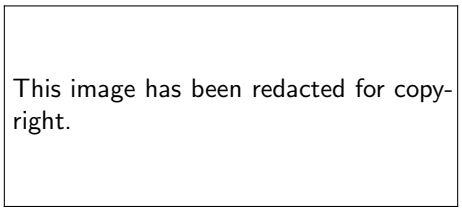
Following our introduction to randomized algorithms in the previous lecture, we begin by giving two more canonical examples of randomized algorithms that highlight common techniques in the field. Then, we formally define the complexity classes associated with randomness and prove their relationships with each other.

12.1 More examples of randomized algorithms

The next problem we tackle will be perfect matching in bipartite graphs. Recall that a perfect matching is a subset of the edge set that uses every vertex exactly once. The problem of determining whether or not a bipartite graph has a perfect matching is indeed possible in deterministic polynomial time. For example, one may reduce it to a max-flow problem and solve it using the theory of network flows. The Edmonds–Karp algorithm for max-flow often found in textbooks runs in $O(nm^2)$ time, and currently best known algorithms can solve max-flow in $O(nm)$ time [4]. Because m is can be as large as $O(n^2)$, one may regard these as taking $O(n^5)$ time and $O(n^3)$ time, respectively.

PROPOSITION 12.1. *There exists a randomized algorithm that determines whether a bipartite graph contains a perfect matching using $O(n^\omega)$ time with 99% accuracy, where $\omega = 2.37...$ is the currently best known matrix multiplication constant.*

Proof. Let G be a bipartite graph and consider the matrix representation of G where rows correspond to vertices on the left and columns correspond to vertices on the right, with 0s and 1s as entries denoting whether or not an edge exists between the two vertices. This is sometimes called the *biadjacency matrix*. Replace each 1 in this matrix with a different variable x_{ij}, and call the resulting matrix M. That is,



Recall that if S_n denotes the symmetric group of order n (i.e. the permutations of $\{1, \ldots, n\}$), then

$$\det M = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) M_{1\sigma(1)} \cdots M_{n\sigma(n)}.$$

Amazingly, $\det M$ is a nonzero polynomial if and only if G has a perfect matching. This is because for every $\sigma \in S_n$, the term $\operatorname{sgn}(\sigma) M_{1\sigma(1)} \cdots M_{n\sigma(n)}$ is non-zero (i.e. is exactly $\operatorname{sgn}(\sigma) x_{1\sigma(1)} \cdots x_{n\sigma(n)}$) if and only if the corresponding matrix entries in the biadjacency matrix were all 1, meaning that there is an edge between the ith vertex on the left and the $\sigma(i)$th vertex on the right for all i, which is exactly the definition of a perfect matching. No two terms in the summation cancel each other out because although $\operatorname{sgn}(\sigma)$ can be either 1 or -1, the variables will always be different.

The algorithm is then very simple. Choose each x_{ij} at random from a sufficiently large finite field $\mathbb{F}_p$. Then plug those values into M and compute the determinant, and claim that there is a perfect matching if $\det M \neq 0$, or that there is no perfect matching if $\det M = 0$.

A technicality is finding a sufficiently large finite field. We will show that finding a prime p in the range $[100n, 200n]$ gives us the desired accuracy. This range indeed contains a prime by Bertrand's postulate (see last lecture). We may simply check each number in the range to see if it is prime.

Let us analyze the correctness. When G has no perfect matching, $\det M$ is the zero polynomial, so it will always evaluate to 0 on any input and we will always answer correctly. On the other hand, when G has a perfect matching, the bad event is when our random choices form a root of the polynomial $\det M$. Note that each non-zero term in $\det M$ has degree n, so the entire polynomial has degree n. Then by the Schwartz-Zippel lemma, the bad event happens with probability at most $\frac{n}{|\mathbb{F}_p|} \leq \frac{n}{100n} = \frac{1}{100}$, which achieves the desired 99% accuracy.

Lastly, let us quickly discuss the running time. Checking if a number is prime takes $O(\log^c n)$ time by the AKS primality test [1], but even doing simple divisibility checks takes just $O(\sqrt{n})$ time. So finding the finite field takes $O(n\sqrt{n})$ time at most. Hence the bulk of the computation comes from computing the determinant of M. It turns out that computing determinant can be reduced to matrix multiplication through LU factorization [3], and the current best known algorithm for matrix multiplication takes $O(n^{2.373\cdots})$ time [2]. □

The fact that the bulk of the complexity comes from matrix multiplication (or finding determinant) is extremely helpful, because matrix multiplication is an extremely well-studied problem with not only fast algorithms but parallelizable algorithms. Although the $O(n^{2.373\cdots})$ algorithm has an unrealistically high constant coefficient, even Gaussian elimination for determinant takes just $O(n^3)$ time, which matches the best known algorithm for max-flow asymptotically and beats its constant coefficient by an astronomical amount. Strassen's algorithm for matrix multiplication takes $O(n^{2.807\cdots})$ time, beating all known flow-based methods, and is also realistically feasible to implement and run.

This result again highlighted the technique of using polynomials and the Schwartz–Zippel lemma, which we saw used in a basic sense in the previous lecture. Although the application of this technique was a little more advanced, the same core idea shows the power of using polynomials in randomized algorithms.

Our last example is 2SAT, the language of 2CNF formulas with a satisfying assignment. In the midterm exam, we showed that 2SAT may be solved deterministically in linear time. There are many ways to do so, for example, by reducing the problem to directed graph and running the strongly connected components algorithm. Obviously, linear time is a lower

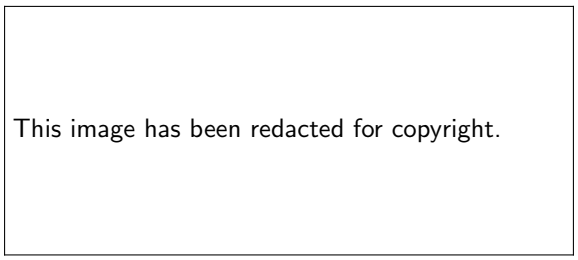
bound for this problem, so our randomized solution will not be better. We present it just to illustrate another common technique in randomized algorithms: the random walk.

PROPOSITION 12.2. *There exists a randomized algorithm that solves 2SAT in $O(n^2)$ time with 99% accuracy.*

Proof. Consider the following algorithm: start with an arbitrary assignment, and as long as the assignment is not a satisfying assignment, pick an arbitrary unsatisfied clause and flip one of the variables at random, for a maximum of $100n^2$ iterations. If we never reach a satisfying assignment, we claim that the 2SAT instance is not satisfiable. Clearly this runs in $O(n^2)$ time.

To analyze the algorithm, it is clearly always correct on no-instances, since we will never reach a satisfying assignment. For yes-instances, let $x^* = (x_1^*, \ldots, x_n^*)$ be a satisfying assignment, and for $i = 0, 1, \ldots, n$, let A_i be the set of assignments that differ from x^* in exactly i places. Although there may be more than one satisfying assignment, we will just fix one satisfying assignment as an upper bound on the probability of failure.

As previously alluded to, we interpret our algorithm as a random walk between the sets $A_0, \ldots, A_n$, and the goal is to reach A_0 (i.e. the satisfying assignment x^*). We have the following picture:



The transitions between the sets are characterized as follows:

- From A_0, the algorithm terminates.
- From A_n, the algorithm moves to A_{n-1} with probability 1. This is because we flip one variable, all of which were wrong, so now one variable is correct.
- From A_i where $0 < i < n$, the algorithm moves to A_{i-1} with probability at least $\frac{1}{2}$, and to A_{i+1} with the complement probability. The exact probability depends on the actual assignment, but in any violating clause, at least one of the two variables must be wrong. We flip one at random, so we flip a wrong variable to correct with probability at least $\frac{1}{2}$, and the other way around with the complement probability.

Define T_i to be the expected number of iterations it takes to reach A_0 from A_i, or rather, because this expectation may depend on the actual assignment in question by the third bullet above, let T_i be the maximum such expectation over all $x \in A_i$. Then the following three equations follow directly from the above three bullets:

$$\begin{cases} T_0 = 0 & (1) \\ T_n = 1 + T_{n-1} & (2) \\ T_i \leq 1 + \frac{1}{2}T_{i-1} + \frac{1}{2}T_{i+1} \text{ for } 0 < i < n & (3) \end{cases}$$

This is $n+1$ linear (in)equalities in $n+1$ variables, so we may solve the system in a number of standard ways. For instance, add together equation (1), $\frac{n}{2}$ times equation (2), and i times equation (3) for each $0 < i < n$. We get that $T_0 + T_1 + 2T_2 + \cdots + (n-1)T_{n-1} + \frac{n}{2}T_n \leq \frac{n^2}{2} + \frac{1}{2}T_0 + T_1 + 2T_2 + \cdots + (n-1)T_{n-1} + \frac{n-1}{2}T_n$. Most terms cancel out, and multiplying by 2 yields $T_n \leq n^2$.

Recall that T_n is an upper bound on the expected time it takes to reach x^*. Then by Markov's inequality, the probability of not reaching x^* after $100n^2$ iterations is bounded by $\frac{1}{100}$, achieving the desired 99% success probability. □

The theory of random walks is rich and far deeper than we can cover in a single example. One amazing classical result is that any random walk on an infinite 1- or 2-dimensional grid passes through every point with probability 1, or equivalently, returns to its starting point with probability 1. In other words, we are guaranteed the success of many random walk solutions to problems, provided that enough time is given. Unfortunately, this result does not hold in higher dimensions. This fact was first shown by George Pólya [5], and Shizuo Kakutani later succinctly communicated the result with the catchphrase: "A drunk man will find his way home, but a drunk bird may get lost forever."

12.2 Randomized complexity classes

With knowledge of a few examples of randomized algorithms, we now turn our attention to categorizing them into complexity classes. There are two main complexity classes that we will study. Denote the indicator function of a language L by $\mathbf{1}_L$.

Definition 12.3 (BPP, RP). A language L is in the class BPP if there exists a polynomial time Turing machine M such that for all $x \in \{0,1\}^*$,

$$\mathbb{P}(M(x,r) = \mathbf{1}_L(x)) \geq \frac{2}{3}.$$

Similarly, a language L is in the class RP if there exists a polynomial time Turing machine M such that

$$\mathbb{P}(M(x,r) = \mathbf{1}_L(x)) \geq \begin{cases} \frac{2}{3} & \text{if } x \in L \\ 1 & \text{if } x \notin L \end{cases}.$$

We call $r \in \{0,1\}^*$ the random string, and it should be polynomial length. In practice, these bits may be obtained from thermal noise in a computer or other sources of randomness. Although M is a deterministic Turing machine, M can output different results on the same input x when given different random strings, achieving the desired random behavior.

It is critically important that only r is probabilistic, not the input x. In other words, these classes require a good chance of correctness on *all* inputs. No matter what the input is, running the machine many times changing only the random string produces the correct output a $\frac{2}{3}$ fraction of the time.

For RP, notice that the accuracy requirement is different for $x \in L$ and $x \notin L$. RP algorithms have *one-sided error*, in particular, they do not make errors on inputs $x \notin L$. Stated differently, RP algorithms do not have false positives (you can trust it if it claims $x \in L$), but may have false negatives. In contrast, BPP algorithms have *two-sided error*.

It is not hard to check that co RP, the class of complements of languages in RP, can be alternatively characterized by switching the roles of $\frac{2}{3}$ and 1. That is, co RP algorithms

do not make errors on inputs $x \in L$, do not have false negatives, and potentially have false positives. Recall that our matrix multiplication verification algorithm from last lecture was always correct when $AB = C$, placing that language in co RP. In contrast, one can easily check that the two examples from this lecture are in RP.

As trivia, BPP stands for "bounded-error probabilistic polynomial time" and RP stands for "randomized polynomial time". One might also wonder if the constant $\frac{2}{3}$ is significant. After all, we did achieve 99% accuracy on our two examples today. We will resolve this question in the next section.

PROPOSITION 12.4. *The following containments are correct:*

1. $\mathrm{P} \subset \mathrm{RP} \subset \mathrm{BPP}$ *and* $\mathrm{P} \subset \mathrm{co\,RP} \subset \mathrm{BPP}$.
2. $\mathrm{RP} \subset \mathrm{NP}$ *and* $\mathrm{co\,RP} \subset \mathrm{co\,NP}$.

Proof. Item (1) is trivial. Any problem in P can be solved correctly in polynomial time with probability 1, placing it both RP and co RP. Any problem in RP or co RP must only admit one-sided error, so it is also in BPP, which allows two-sided error.

To show (2), that RP $\subset$ NP, consider the random string as the certificate. In particular, because $\mathbb{P}(M(x, r) = \mathbf{1}_L(x)) > \frac{2}{3} > 0$ for $x \in L$, there exists at least one random string r such that $M(x, r) = \mathbf{1}_L(x)$. This exactly matches the definition of NP. In particular, we have certificates for yes-instances and we reject all no-instances, just as required. □

The technique of showing existence by instead showing non-zero probability is called the *probabilistic method*. We will see many more applications of this idea in future lectures, where we will be able to appreciate the true power of the the probabilistic method.

One might wonder what the upper bound on BPP is. Trivially, BPP $\subset$ PSPACE because one may simply try every random string while reusing space, then output the majority answer. Some people conjecture that in fact BPP = P, but the current best known upper bound is BPP $\subset \Sigma_2 \cap \Pi_2$. This is Sipser–Gács theorem, whose proof we delay to the next lecture because it is a substantially non-trivial result.

12.3 Error reduction

As alluded to, one extremely useful fact about RP and BPP is that the constant of $\frac{2}{3}$ in their definition does not matter at all. In RP, it is equivalent to require a success probability anywhere from $\frac{1}{\operatorname{poly} n}$ to $1 - e^{-\operatorname{poly} n}$, and in BPP it is equivalent to require a success probability anywhere from $\frac{1}{2} + \frac{1}{\operatorname{poly} n}$ to $1 - e^{-\operatorname{poly} n}$. (By $\operatorname{poly} n$, with the convention that $n = |x|$, we mean n^c for some $c \in \mathbb{R}$, or sometimes n^c for all $c \in \mathbb{R}$, depending on context.)

What this means is that any problem in RP or BPP can be computed *more* accurately as the input size increases, and in fact the accuracy improves exponentially as the input grows. This is the best that anyone could ever hope for, and it makes RP and BPP algorithms extremely practical, because most real-world applications do not care about errors that are this rare. On the other hand, to show that a problem belongs on RP or BPP, we only need to solve it with a fairly low success probability.

We start with the proof for RP, since it is simpler.

THEOREM 12.5 (Error reduction for RP). *Let L be a language, and suppose there exists a polynomial time Turing machine M such that*

$$\mathbb{P}(M(x,r) = \mathbf{1}_L(x)) \geq \begin{cases} \frac{1}{\operatorname{poly} n} & \text{if } x \in L \\ 1 & \text{if } x \notin L \end{cases}.$$

Then there exists a polynomial time Turing machine M' such that

$$\mathbb{P}(M'(x,r) = \mathbf{1}_L(x)) \geq \begin{cases} 1 - e^{-\operatorname{poly} n} & \text{if } x \in L \\ 1 & \text{if } x \notin L \end{cases}.$$

Proof. Because M does not make errors on $x \in L$, we may run M multiple times on independent random strings, and we know that if any run ever rejects, then the correct answer must be to reject. Otherwise, we accept. The chance that M is wrong every time we run it is very small, so the only question is how many times do we need to run M?

Denote the number of times we run M by N. Note that the inequality $e^x \geq 1 + x$ implies $1 - x \leq e^{-x}$. Then, the probability of being wrong all N times is given by

$$\begin{aligned} \mathbb{P}(M'(x,r) \neq \mathbf{1}_L(x)) &\leq \left(1 - \frac{1}{\operatorname{poly} n}\right)^N \\ &\leq (e^{-1/\operatorname{poly} n})^N. \end{aligned}$$

From here, it is easy to see that setting N to a polynomial reduces the probability of error to $e^{-\operatorname{poly} n}$. This gives us our polynomial running time for M', as well as our $1 - e^{-\operatorname{poly} n}$ probability of success, as was to be shown. ∎

The inequality that $1 - x \leq e^{-x}$ is important and extremely widely used in the field of randomized algorithms, because multiplying terms of the form e^{-x} is much easier than terms of the form $1 - x$. It allows us to derive cleaner bounds without sacrificing accuracy, since it is very tight when x is small.

For BPP, this simple inequality does not work because the two-sided error requires a different approach to bounding the probability of the bad event. The alternative tool we will use is the Chernoff bound, which we discussed in the previous lecture.

THEOREM 12.6 (Error reduction for BPP). *Let L be a language, and suppose there exists a polynomial time Turing machine M such that for all $x \in \{0,1\}^*$,*

$$\mathbb{P}(M(x,r) = \mathbf{1}_L(x)) \geq \frac{1}{2} + \frac{1}{\operatorname{poly} n}.$$

Then there exists a polynomial time Turing machine M' such that for all $x \in \{0,1\}^*$,

$$\mathbb{P}(M'(x,r) = \mathbf{1}_L(x)) \geq 1 - e^{-\operatorname{poly} n}.$$

Proof. Since M may make errors on both sides, there is only one sensible thing to do. Let M' be the Turing machine that runs M for a total of N times on independent random strings and returns the majority answer. Again, let us see what N has to be.

Recall that the Chernoff bound says that if $X_1, \ldots, X_N$ are independent indicator random variables with probability of success p, then for all $\delta > 0$,

$$\mathbb{P}\left(\left|\frac{X_1 + \cdots + X_N}{N} - p\right| \geq \delta\right) \leq 2e^{-2\delta^2 N}.$$

In other words, the probability that the empirical mean deviates from the theoretical mean decreases exponentially in both the error and the number of trials. If we let $X_1, \ldots, X_N$ denote whether or not the result is correct each time we run M, note that

$$\begin{aligned}
\mathbb{P}(M(x,r) \neq \mathbf{1}_L(x)) &= \mathbb{P}\left(X_1 + \cdots + X_N \leq \frac{N}{2}\right) \\
&= \mathbb{P}\left(\frac{X_1 + \cdots + X_N}{N} - p \leq \frac{1}{2} - p\right) \\
&\leq \mathbb{P}\left(\left|\frac{X_1 + \cdots + X_N}{N} - p\right| \geq p - \frac{1}{2}\right) \\
&\leq 2e^{-2(p-\frac{1}{2})^2 N} \\
&\leq 2e^{-2(1/\operatorname{poly} n)^2 N}.
\end{aligned}$$

Because $(\frac{1}{\operatorname{poly} n})^2$ is still just the inverse of a polynomial, setting N equal to larger polynomial gives us the desired $e^{-\operatorname{poly} n}$ probability of failure. (The multiplicative constant 2 in front can be absorbed into the $\operatorname{poly} n$ term.) □

In the next lecture, we will wrap up our discussion of randomness by proving the Sipser–Gács theorem that $\mathrm{BPP} \subset \Sigma_2 \cap \Pi_2$, as well as covering some other miscellaneous topics in randomness.

References

[1] M. Agrawal, N. Kayal, and N. Saxena. Primes is in p. *Annals of Mathematics*, 160, 09 2002.

[2] J. Alman and V. V. Williams. A refined laser method and faster matrix multiplication, 2020.

[3] M. Lavrov. Determinant and matrix multiplication complexity? Mathematics Stack Exchange.

[4] J. B. Orlin. Max flows in $o(nm)$ time, or better. In *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '13, page 765–774, New York, NY, USA, 2013. Association for Computing Machinery.

[5] G. Pólya. Über eine aufgabe der wahrscheinlichkeitsrechnung betreffend die irrfahrt im straßennetz. *Mathematische Annalen*, 84:149–160.