2

The Power of Randomness

2.1 Polynomial Identity Testing

Before we study the derandomization of randomized algorithms, we will need some algorithms to derandomize. This section introduces one such algorithm. It solves the following computational problem.

Computational Problem 2.1. Identity Testing: given two multivariate polynomials, $p(x_1, \ldots, x_n)$ and $q(x_1, \ldots, x_n)$, decide whether $p = q$.

This definition requires some clarification. Specifically, we need to say what we mean by:

- "polynomials": A *(multivariate) polynomial* is an finite expression of the form

$$p(x_1, \ldots, x_n) = \sum_{i_1,\ldots,i_n \in \mathbb{N}} c_{i_1,\ldots,i_n} x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}.$$

  We need to specify what space the coefficients of the polynomials will come from; they could be the integers, reals, rationals, etc. In general, we will assume that the coefficients are chosen from a *field* (a set with addition and multiplication, where every nonzero element has a multiplicative inverse) or more generally an *(integral) domain* (where the product of two nonzero elements is always nonzero). Examples of fields include $\mathbb{Q}$ (the rationals), $\mathbb{R}$ (the reals), $\mathbb{Z}_p$ (integers modulo p) where p is prime. An integral domain that is not a field is $\mathbb{Z}$ (the integers), but every integral domain is contained in its *field of fractions*, which is $\mathbb{Q}$ in the case of $\mathbb{Z}$. $\mathbb{Z}_n$ for composite n is not even an integral domain. We remark that there does exist a finite field $\mathbb{F}_q$ of size $q = p^k$ for every prime p and positive integer k, and in fact this field is unique (up to isomorphism); but $\mathbb{F}_q$ is only equal to $\mathbb{Z}_q$ in case q is prime (i.e. $k = 1$). For more background on algebra, see the references in the chapter notes.
  For a polynomial $p(x_1, \ldots, x_n) = \sum_{i_1,\ldots,i_n} c_{i_1,\ldots,i_n} x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}$, we define its *degree* (a.k.a. *total degree*) to be the maximum sum of the exponents $i_1 + \cdots + i_n$ over its monomials

with nonzero coefficients $c_{i_1,\ldots,i_n}$. Its *degree in x_j* is the maximum of i_j over its monomials with nonzero coefficients.

- "$p = q$": What does it mean for two polynomials to be equal? There are two natural choices: the polynomials are the same as *functions* (they have the same output for every point in the domain), or the polynomials are the same as *formal polynomials* (the coefficients for each monomial are the same).
  These two definitions are equivalent over the integers (or more generally over infinite domains), but they are *not* equivalent over finite fields. For example, consider
  $$p(x) = \prod_{\alpha \in \mathbb{F}} (x - \alpha).$$
  for a finite field $\mathbb{F}$.[1] It is easy to see that $p(\alpha) \neq 0$ for all $\alpha \in \mathbb{F}$, but $p \neq 0$ as a formal polynomial. For us, *equality refers to equality as formal polynomials.*
- "given": What does it mean to be given a polynomial? There are several possibilities here:
  (1) As a list of coefficients: this trivializes the problem of Identity Testing, as we can just compare.
  (2) As an "oracle": a black box that, given any point in the domain, gives the value of the polynomial.
  (3) As an *arithmetic formula*: a sequence of symbols like $(x_1+x_2)(x_3+x_7+6x_5)x_3(x_5-x_6) + x_2x_4(2x_3 + 3x_5)$ that describes the polynomial. Observe that while we can solve Identity Testing by expanding the polynomials and grouping terms, but the expanded polynomials may have length exponential in the length of the formula, and thus the algorithm is not efficient.
  More general than formulas are circuits. An *arithmetic circuit* consists of a directed acyclic graph, consisting of *input nodes*, which have indegree 0 and are labeled by input variables or constants, and *computation nodes*, which have indegree 2 and are labelled by operations ($+$ or $\times$) specifying how to compute a value given the values at its children; one of the computation nodes is designated as the *output node*. Observe that every arithmetic circuit defines a polynomial in its input variables $x_1, \ldots, x_n$. Arithmetic formulas are equivalent to arithmetic circuits where the underlying graph is a tree.

The randomized algorithm we describe will work for both the 2nd and 3rd formulations above (oracles and arithmetic circuits/formulas). It will be convenient to work with the following equivalent version of the problem.

Computational Problem 2.2. Identity Testing (reformulation): Given a polynomial $p(x_1, \ldots, x_n)$, is $p = 0$?

That is, we consider the special case of the original problem where $q = 0$. Any solution for the general version of course implies one for the special case; conversely, we can solve the general version by applying the special case to the polynomial $p' = p - q$.

[1] When expanded and terms are collected, this polynomial p can be shown to simply equal $x^{|\mathbb{F}|} - x$.

Algorithm 2.3 (Identity Testing).
Input: A multivariate polynomial $p(x_1, \ldots, x_n)$ of degree at most d over a field/domain $\mathbb{F}$.

(1) Let $S \subseteq \mathbb{F}$ be any set of size $2d$.
(2) Choose $\alpha_1, \ldots, \alpha_n \overset{\text{R}}{\leftarrow} S$.
(3) Evaluate $p(\alpha_1, \ldots, \alpha_n)$. If the result is 0, accept. Otherwise, reject.

It is clear that if $p = 0$, the algorithm will always accept. The correctness in case $p \neq 0$ is based on the following simple but very useful lemma.

Lemma 2.4 (Schwartz–Zippel Lemma). If p is a nonzero polynomial of degree d over a field (or integral domain) $\mathbb{F}$ and $S \subseteq \mathbb{F}$, then

$$\Pr_{\alpha_1,\ldots,\alpha_n \overset{\text{R}}{\leftarrow} S} [p(\alpha_1, \ldots, \alpha_n) = 0] \leq \frac{d}{|S|}.$$

In the univariate case ($n = 1$), this amounts to the familiar fact that a polynomial with coefficients in a field and degree d has at most d roots. The proof for multivariate polynomials proceeds by induction on n, and we leave it as an exercise (Problem 2.1).

By the Schwartz-Zippel lemma, the algorithm will err with probability at most $1/2$ when $p \neq 0$. This error probability can be reduced by repeating the algorithm many times (or by increasing $|S|$). Note that the error probability is only over the coin tosses of the algorithm, not over the input polynomial p. This is what we mean when we say *randomized algorithm*; it should work on a worst-case input with high probability over the coin tosses of the algorithm. Algorithms whose correctness (or efficiency) only holds for randomly chosen inputs are called *heuristics*, and their study is called *average-case analysis.*

Note that we need a few things to ensure that our algorithm will work.

- First, we need is a bound on the degree of the polynomial. We can get this in different ways depending on how the polynomial is represented. For example, for arithmetic formulas, the degree is bounded by the length of the formula. For arithmetic circuits, the degree is at most exponential in the size (or even depth) of the circuit.
- We also must be able to evaluate p when the variables take arbitrary values in some set S of size $2d$. For starters, this requires that the domain $\mathbb{F}$ is of size at least $2d$. We should also have an explicit representation of the domain $\mathbb{F}$ enabling us to write down and manipulate field elements (e.g. the prime p in case $\mathbb{F} = \mathbb{Z}_p$). Then, if we are given p as an oracle, we have the ability to evaluate p by definition. If we are given p as an arithmetic formula or circuit, then we can do a bottom-up, gate-by-gate evaluation. However, over infinite domains (like $\mathbb{Z}$), there is subtlety — the bit-length of the numbers can grow exponentially large. Problem 2.4 gives a method for coping with this.

Since these two conditions are satisfied, we have a polynomial-time randomized algorithm for IDENTITY TESTING for polynomials given as arithmetic formulas over $\mathbb{Z}$ (or even circuits, by Problem 2.4). There are no known subexponential-time *deterministic* algorithms for this problem, even for formulas in $\Sigma\Pi\Sigma$ form (i.e. a sum of terms, each of which is the product of linear functions in the input variables). A deterministic polynomial-time algorithm for $\Sigma\Pi\Sigma$ formulas where the outermost sum has only a constant number of terms was obtained quite recently (2005).

2.1.1 Application to Perfect Matching

Now we will see an application of IDENTITY TESTING to an important graph-theoretic problem.

Definition 2.5. Let $G = (V, E)$, then a *matching* on G is a set $E' \subset E$ such that no two edges in E' have a common endpoint. A *perfect matching* is a matching such that every vertex is incident to an edge in the matching.

Computational Problem 2.6. PERFECT MATCHING: given a graph G, decide whether there is a perfect matching in G.

Unlike IDENTITY TESTING, PERFECT MATCHING has deterministic polynomial-time algorithms — e.g. using alternating paths, or by reduction to MAX FLOW in the bipartite case. However, both of these algorithms seem to be inherently sequential in nature. With randomization, we can obtain an efficient parallel algorithm.

Algorithm 2.7 (PERFECT MATCHING in bipartite graphs).
Input: a bipartite graph G with n vertices on each side.

We construct an $n \times n$ matrix A where

$$A_{i,j}(x) = \begin{cases} x_{i,j} & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases},$$

where $x_{i,j}$ is a formal variable.

Consider the multivariate polynomial

$$\det(A(x)) = \sum_{\sigma \in S_n} (-1)^{\text{sign}(\sigma)} \prod_i A_{i,\sigma(i)},$$

where S_n denotes the set of permutations on $[n]$. Note that the σ'th term is nonzero if and only if the σ defines a perfect matching. That is, $(i, \sigma(i)) \in E$ for all $1 \le i \le n$. So $\det(A(x)) = 0$ iff G has no perfect matching. Moreover its degree is bounded by n, and given values $\alpha_{i,j}$ for the $x_{i,j}$'s we can evaluate $\det(A(\alpha))$ efficiently in parallel (in polylogarithmic time using a polynomial number of processors) using an efficient parallel algorithm for determinant.

So to test for a perfect matching efficiently in parallel, just run the IDENTITY TESTING algorithm with, say, $S = \{1, \ldots, 2n\} \subset \mathbb{Z}$, to test whether $\det(A(x)) = 0$.

Some remarks:

- The above also provides the most efficient *sequential* algorithm for Perfect Matching, using the fact that Determinant has the same time complexity as Matrix Multiplication, which is known to be at most $O(n^{2.38})$.
- More sophisticated versions of the algorithm apply to non-bipartite graphs, and enable *finding* perfect matchings in the same parallel or sequential time complexity (where the result for sequential time is quite recent).
- Identity Testing has been also used to obtain a randomized algorithm for Primality, which was derandomized fairly recently (2002) to obtain the celebrated deterministic polynomial-time algorithm for Primality. See Problem 2.5.

2.2 The Computational Model and Complexity Classes

2.2.1 Models of Randomized Computation

To develop a rigorous theory of randomized algorithms, we need to use a precise model of computation. There are several possible ways to augmenting a standard deterministic computational model (e.g. Turing machine or RAM model), such as:

(1) The algorithm has access to a "black box" that provides it with (unbiased and independent) random bits on request, with each request taking one time step. This is the model we will use.

(2) The algorithm has access to a black box that, given a number n in binary, returns a number chosen uniformly at random from $\{1, \ldots, n\}$. This model is often more convenient for describing algorithms. Problem 2.2 shows that it is equivalent to Model 1, in the sense that any problem that can be solved in polynomial time on one model can also be solved in polynomial time on the other.

(3) The algorithm is provided with an infinite tape (i.e. sequence of memory locations) that is that is initially filled with random bits. For polynomial-time algorithms, this is equivalent to the Model 1. However, for space-bounded algorithms, this model seems stronger, as it provides the algorithm with free storage of its random bits (i.e. not counted towards its working memory). Model 1 is considered to be the "right" model for space-bounded algorithms. It can be shown to be equivalent to allowing the algorithm *one-way* access to an infinite tape of random bits.

2.2.2 Complexity Classes

We will now define complexity classes that capture the power of efficient randomized algorithms. As is common in complexity theory, these classes are defined in terms of decision problems, where the set of inputs where the answer should be "yes" is specified by a *language* $L \subseteq \{0,1\}^*$. However, the definitions generalize in natural ways to other types of computational problems, such as computing functions or solving search problems.

Recall that we say an algorithm A runs in time $t : \mathbb{N} \to \mathbb{N}$ if A takes at most $t(|x|)$ steps on every input x, and it runs in *polynomial time* if it runs time $t(n) = O(n^c)$ for a constant c. Polynomial time is a theoretical approximation to feasible computation, with the advantage that it is robust to reasonable changes in the model of computation and representation of the inputs.

Definition 2.8. **P** is the class of languages L for which there exists a deterministic polynomial-time algorithm A such that

- $x \in L \Rightarrow A(x)$ accepts.
- $x \notin L \Rightarrow A(x)$ rejects.

For a randomized algorithm A, we say that A runs in time $t : \mathbb{N} \to \mathbb{N}$ if A takes at most $t(|x|)$ steps on every input x *and every sequence of random coin tosses.*

Definition 2.9. **RP** is the class of languages L for which there exists a probabilistic polynomial-time algorithm A such that

- $x \in L \Rightarrow \Pr[A(x) \text{ accepts}] \geq 1/2$.
- $x \notin L \Rightarrow \Pr[A(x) \text{ accepts}] = 0$.

That is, **RP** algorithms may have *false negatives*; the algorithm may sometimes say "no" even if the answer is "yes", albeit with bounded probability. But the definition does not allow for false positives. Thus **RP** captures efficient randomized computation with *one-sided error*. **RP** stands for "random polynomial time". Note that the error probability of an **RP** algorithm can be reduced to $2^{-p(n)}$ for any polynomial p by running the algorithm $p(n)$ times independently and accepting the input iff at least one of the trials accepts. By the same reasoning, the $1/2$ in the definition is arbitrary, and any constant $\alpha \in (0,1)$ or even $\alpha = 1/\text{poly}(n)$ would yield the same class of languages.

A central question in this course is whether randomization enables us to solve more problems in polynomial time (e.g. decide more languages):

Open Problem 2.10. Does **P** = **RP**?

Similarly, we can consider algorithms that may have false positives but no false negatives.

Definition 2.11. **co-RP** is the class of languages L whose complement $\bar{L}$ is in **RP**. Equivalently, $L \in$ **co-RP** if there exists a probabilistic polynomial-time algorithm A such that

- $x \in L \Rightarrow \Pr[A(x) \text{ accepts}] = 1$.
- $x \notin L \Rightarrow \Pr[A(x) \text{ accepts}] \leq 1/2$.

That is, in **co-RP** we may err on NO instances, whereas in **RP** we may err on YES instances.

Using the IDENTITY TESTING algorithm we saw earlier, we can deduce that IDENTITY TESTING for arithmetic formulas is in **co-RP**. In Problem 2.4, this is generalized to arithmetic circuits, and thus we have:

Theorem 2.12. The language

$$\mathrm{ACIT}_{\mathbb{Z}} = \{C : C(x_1, \ldots, x_n) \text{ an arithmetic circuit over } \mathbb{Z} \text{ s.t. } C = 0\}$$

is in **co-RP**.

It is common to also allow two-sided error in randomized algorithms:

Definition 2.13. **BPP** is the class of languages L for which there exists a probabilistic polynomial-time algorithm A such that

- $x \in L \Rightarrow \Pr[A(x) \text{ accepts}] \geq 2/3$.
- $x \notin L \Rightarrow \Pr[A(x) \text{ accepts}] \leq 1/3$.

Just as with **RP**, the error probability of **BPP** algorithms can be reduced from 1/3 (or even $1/2 - 1/\mathrm{poly}(n)$) to exponentially small by repetitions, this time taking a majority vote of the outcomes. Proving this requires some facts from probability theory, which we will review in the next section.

The cumbersome notation **BPP** stands for 'bounded-error probabilistic polynomial-time," due to the unfortunate fact that **PP** ("probabilistic polynomial-time") refers to the definition where the inputs in L are accepted with probability greater than 1/2 and inputs not in L are accepted with probability at most 1/2. Despite its name, **PP** is not a reasonable model for randomized algorithms, as it takes exponentially many repetitions to reduce the error probability. **BPP** is considered the standard complexity class associated with probabilistic polynomial-time algorithms, and thus the main question of this course is:

Open Problem 2.14. Does **BPP** = **P**?

So far, we have considered randomized algorithms that can output an incorrect answer if they are unlucky in their coin tosses; these are called "Monte Carlo" algorithms. It is sometimes preferable to have "Las Vegas" algorithms, which always output the correct answer, but may run for a longer time if they are unlucky in their coin tosses. For this, we say that A has *expected running time* $t : \mathbb{N} \to \mathbb{N}$ if for every input x, the expectation of the number of steps taken by $A(x)$ is at most $t(|x|)$, where the expectation is taken over the coin tosses of A.

Definition 2.15. **ZPP** is the class of languages L for which there exists a probabilistic algorithm A that always decides L correctly and runs in expected polynomial time.

ZPP stands for "zero-error probabilistic polynomial time". The following relation between **ZPP** and **RP** is left as an exercise.

Fact 2.16 (Problem 2.3). **ZPP** = **RP** $\cap$ **co-RP**.

We do not know any other relations between the classes associated with probabilistic polynomial time.

Open Problem 2.17. Are any of the inclusions $\mathbf{P} \subset \mathbf{ZPP} \subset \mathbf{RP} \subset \mathbf{BPP}$ proper?

One can similarly define randomized complexity classes associated with complexity measures other than time such as space or parallel computation. For example:

Definition 2.18. **RNC** is the class of languages L such that exists a probabilistic parallel algorithm A that runs in polylogarithmic time on polynomially many processors, such that

- $x \in L \Rightarrow \Pr[A(x) \text{ accepts}] \geq 1/2$.
- $x \notin L \Rightarrow \Pr[A(x) \text{ accepts}] = 0$.

A formal model of a parallel algorithm is beyond the scope of this course, but can be found in standard texts on algorithms or parallel computation. We have seen:

Theorem 2.19. Perfect Matching in bipartite graphs, i.e. the language $\mathrm{PM} = \{G : G \text{ a bipartite graph with a perfect matching}\}$, is in **RNC**.

2.2.3 Tail Inequalities and Error Reduction

In the previous section, we claimed that we can reduce the error of a **BPP** algorithm by taking independent repetitions and ruling by majority vote. The intuition that this should work is based on the Law of Large Numbers: if we repeat the algorithm many times, the fraction of correct answers should approach its expectation, which is greater than 1/2 (and thus majority rule will be correct). For complexity purposes, we need quantitative forms of this fact, which bound how many repetitions are needed to achieve a desired probability of correctness.

First, we recall a basic inequality which says that it is unlikely for (a single instantiation of) a random variable to exceed its expectation by a large factor.

Lemma 2.20 (Markov's Inequality). If X is a nonnegative random variable, then for any $\alpha > 0$,

$$\Pr[X \geq \alpha] \leq \frac{\mathrm{E}[X]}{\alpha}$$

Markov's Inequality alone does not give very tight concentration around the expectation; to get even a 50% probability, we need to look at deviations by a factor of 2. To get tight concentration, we need to take independent copies of a random variable. There are a variety of different tail inequalities that apply for this setting; they are collectively referred to as *Chernoff Bounds.*

Theorem 2.21 (A Chernoff Bound). Let $X_1, \ldots, X_t$ be independent random variables taking values in the interval $[0,1]$, let $X = (\sum_i X_i)/t$, and $\mu = \mathrm{E}[X]$. Then

$$\Pr[|X - \mu| \geq \varepsilon] \leq 2\exp(-t\varepsilon^2/2).$$

Thus, the probability that the average deviates significantly from the expectation vanishes exponentially with the number of repetitions t. We leave the proof of this Chernoff Bound as an exercise (Problem 2.7).

Now let's apply the Chernoff Bound to analyze error-reduction for **BPP** algorithms.

Proposition 2.22. The following are equivalent:

(1) $L \in \mathbf{BPP}$.
(2) For every polynomial p, L has a probabilistic polynomial-time algorithm with two-sided error at most $2^{-p(n)}$.
(3) There exists a polynomial q such that L has a probabilistic polynomial-time algorithm with two-sided error at most $1/2 - 1/q(n)$.

Proof. Clearly, (2)$\Rightarrow$ (1) $\Rightarrow$ (3). Thus, we prove (3) $\Rightarrow$ (2).

Given an algorithm A with error probability at most $1/2 - 1/q(n)$, consider an algorithm A' that on an input x of length n, runs A for $t(n)$ independent repetitions and rules according to the majority, where $t(n)$ is a polynomial to be determined later.

We now compute the error probability of A' on an input x of length n. Let X_i be an indicator random variable that is 1 iff the i'th execution of $A(x)$ outputs the correct answer, and let $X = (\sum_i X_i)/t$ be the average of these indicators, where $t = t(n)$. Note that $A'(x)$ is correct when $X > 1/2$. By the error probability of A and linearity of expectations, we have $\mathrm{E}[X] \geq 1/2 + 1/q$, where $q = q(n)$. Thus, applying the Chernoff Bound with $\varepsilon = 1/q$, we have:

$$\Pr[X \leq 1/2] \leq 2 \cdot e^{-t/2q^2} < 2^{-p(n)},$$

for $t(n) = 2p(n)q(n)^2$ and sufficiently large n. □

2.3 Sampling and Approximation Problems

2.3.1 Sampling

The power of randomization is well-known to statisticians. If we want to estimate the mean of some quantity over a large population, we can do so very efficiently by taking the average over a small random sample.

Formally, here is the computational problem we are interested in solving.

Computational Problem 2.23. SAMPLING (aka $[+\varepsilon]$-APPROX ORACLE AVERAGE): Given oracle access to a function $f : \{0,1\}^m \to [0,1]$, estimate $\mu(f) \overset{\text{def}}{=} \mathrm{E}[f(U_m)]$ to within an additive error of ε. That is, output an answer in the interval $[\mu - \varepsilon, \mu + \varepsilon]$.

And here is the algorithm:

Algorithm 2.24 ($[+\varepsilon]$-APPROX ORACLE AVERAGE). For an appropriate choice of t, choose $x_1, \ldots, x_t \overset{\text{R}}{\leftarrow} \{0,1\}^m$, query the oracle to obtain $f(x_1), \ldots, f(x_t)$, and output $(\sum_i f(x_i))/t$.

By the Chernoff Bound (Theorem 2.21), we only need to take $t = O(\log(1/\delta)/\varepsilon^2)$ samples to have additive error at most ε with probability at least $1-\delta$. Note that for constant ε and δ, the sample size is independent of the size of the population (2^m), and we have running time $\mathrm{poly}(m)$ even for $\varepsilon = 1/\mathrm{poly}(m)$ and $\delta = 2^{-\mathrm{poly}(m)}$.

For this problem, we can *prove* that no deterministic algorithm can be nearly as efficient.

Proposition 2.25. Any deterministic algorithm solving [+(1/4)]-Approx Oracle Average must make at least $2^m/2$ queries to its oracle.

Proof. Suppose we have a deterministic algorithm A that makes fewer than $2^n/2$ queries. Let Q be the set of queries made by A when all of its queries are answered by 0. Now define two functions

$$\begin{aligned} f_0(x) &= 0 \qquad \forall x \\ f_1(x) &= \begin{cases} 0 & x \in Q \\ 1 & x \notin Q \end{cases} \end{aligned}$$

Then A gives the same answer on both f_0 and f_1 (since all the oracle queries return 0 in both cases), but $\mu(f_0) = 0$ and $\mu(f_1) > 1/2$, so the answer must have error greater than 1/4 for at least one of the functions. □

Thus, randomization provides an exponential savings for approximating the average of a function on a large domain. However, this does not show that $\mathbf{BPP} \neq \mathbf{P}$. There are two reasons for this:

(1) [$+\varepsilon$]-Approx Oracle Average is not a decision problem, and indeed it is not clear how to define languages that capture the complexity of approximation problems. However, below we will see how a slightly more general notion of decision problem does allow us to capture approximation problems such as this one.

(2) More fundamentally, it does not involve the standard model of input as used in the definitions of **P** and **BPP**. Rather than the input being a string that is explicitly given to the algorithm (where we measure complexity in terms of the length of the string), the input is an exponential-sized oracle to which the algorithm is given random access. Even though this is not the classical notion of input, it is an interesting one that has received a lot of attention in recent years, because it allows for algorithms whose running time is sublinear (or even polylogarithmic) in the actual size of the input (e.g. 2^m in the example here). As in the example here, typically such algorithms require randomization and provide approximate answers.

2.3.2 Promise Problems

Now we will try to find a variant of the [$+\varepsilon$]-Approx Oracle Average problem that is closer to the **P** vs. **BPP** question. First, to obtain the standard notion of input, we consider functions that are presented in a concise form, as Boolean circuits $C : \{0,1\}^m \to \{0,1\}$ (analogous to the algebraic circuits defined last lecture, but now the inputs take on Boolean values and the computation gates are $\wedge$, $\vee$, and $\neg$).

Next, we need a more general notion of decision problem than languages:

Definition 2.26. A *promise problem* Π consists of a pair (Π_Y, Π_N) of disjoint sets of strings, where Π_Y is the set of YES instances and Π_N is the set of NO instances. The corresponding computational problem is: given a string that is "promised" to be in $\Pi_Y \cup \Pi_N$, decide which is the case.

All of the complexity classes we have seen have natural promise-problem analogues, which we denote by **prP**, **prRP**, **prBPP**, etc. For example:

Definition 2.27. **prBPP** is the class of promise problems Π for which there exists a probabilistic polynomial-time algorithm A such that

- $x \in \Pi_Y \Rightarrow \Pr[A(x) \text{ accepts}] \geq 2/3$.
- $x \in \Pi_N \Rightarrow \Pr[A(x) \text{ accepts}] \leq 1/3$.

Since every language L corresponds to the promise problem $(L, \overline{L})$, any result proven for every promise problem in some promise-class also holds for every language in the corresponding language class. In particular, if every **prBPP** algorithm can be derandomized, so can every **BPP** algorithm:

Proposition 2.28. **prBPP = prP $\Rightarrow$ BPP = P**.

Now we can consider the following problem.

Computational Problem 2.29. $[+\varepsilon]$-APPROX CIRCUIT AVERAGE is the promise problem CA^ε, defined as:

$$\begin{aligned}\mathrm{CA}^\varepsilon_Y &= \{(C,p) : \mu(C) > p + \varepsilon\}\\ \mathrm{CA}^\varepsilon_N &= \{(C,p) : \mu(C) \leq p\}\end{aligned}$$

Here ε can be a constant or a function of the input length $n = |(C,p)|$.

It turns out that this problem completely captures the power of probabilistic polynomial-time algorithms.

Theorem 2.30. For every function ε such that $1/\mathrm{poly}(n) \leq \varepsilon(n) \leq 1 - 1/2^{n^{o(1)}}$, $[+\varepsilon]$-APPROX CIRCUIT AVERAGE is **prBPP**-complete. That is, it is in **prBPP** and every promise problem in **prBPP** reduces to it.

Proof. [Sketch]
Inclusion in **prBPP**: Follows from Algorithm 2.24 and the fact that boolean circuits can be evaluated in polynomial time.

Hardness for **prBPP**: Given any promise problem $\Pi \in$ **prBPP**, we have a probabilistic polynomial-time algorithm A that decides Π with 2-sided error at most 2^{-n} on inputs of length n.

We can view the output of $A(x;r)$ as a function of its input x and its coin tosses r. Note that if x is of length n, then we may assume that r is of length at most $\mathrm{poly}(n)$ without loss of generality (because an algorithm that runs in time at most t can toss at most t coins). For any n, there is a $\mathrm{poly}(n)$-sized circuit $C(x;r)$ that simulates the computation of A for inputs x of length n and coin tosses r of length $\mathrm{poly}(n)$, and moreover C can be constructed in time $\mathrm{poly}(n)$. (See any text on complexity theory for a proof.) Let $C_x(r)$ be the circuit C with x hardwired in. Then the map $x \mapsto (C_x, 1/2^n)$ is a polynomial-time reduction from Π to $[+\varepsilon]$-Approx Circuit Average. Indeed, if $x \in \Pi_N$, then A accepts with probability at most $1/2^n$, so $\mu(C_x) \le 1/2^n$. And if $x \in \Pi_Y$, then $\mu(C_x) \ge 1 - 1/2^n > 1/2^n + \varepsilon(n')$, where $n' = |(C_x, 1/2^n)| = \mathrm{poly}(n)$ and we take n sufficiently large. □

Consequently, derandomizing this one algorithm is equivalent to derandomizing all of **prBPP**:

Corollary 2.31. $[+\varepsilon]$-Approx Circuit Average is in **prP** if and only if **prBPP** = **prP**.

Note that the proof of Proposition 2.25 does not extend to $[+\varepsilon]$-Approx Circuit Average. Indeed, it's not even clear how to define the notion of "query" for an algorithm that is given a circuit C explicitly; it can do arbitrary computations that involve the internal structure of the circuit. Moreover, even if we restrict attention to algorithms that only use the input circuit C as if it were an oracle (other than computing the input length $|(C,p)|$ to know how long it can run), there is no guarantee that the function f_1 constructed in the proof of Proposition 2.25 has a small circuit.

2.3.3 Approximate Counting to within Relative Error

Note that $[+\varepsilon]$-Approx Circuit Average can be viewed as the problem of approximately counting the number of satisfying assignments of a circuit $C : \{0,1\}^m \to \{0,1\}$ to within additive error $\varepsilon \cdot 2^m$, and a solution to this problem may give useless information for circuits that don't have very many satisfying assignments (e.g. circuits with fewer than $2^{m/2}$ satisfying assignments). Thus people typically study approximate counting to within *relative error*. For example, given a circuit C, output a number that is within a $(1+\varepsilon)$ factor of its number of satisfying assignments, $\#C$. Or the following essentially equivalent decision problem:

Computational Problem 2.32. $[\times(1+\varepsilon)]$-Approx #CSAT is the following promise problem:

$$\begin{aligned}\mathrm{CSAT}^{\varepsilon}_{Y} &= \{(C,N) : \#C > (1+\varepsilon)\cdot N\}\\ \mathrm{CSAT}^{\varepsilon}_{N} &= \{(C,N) : \#C \le N\}\end{aligned}$$

Unfortunately, this problem is **NP**-hard for general circuits (consider $N = 0$), so we do not expect a **prBPP** algorithm. However, there is a very pretty randomized algorithm if we restrict to DNF formulas.

Computational Problem 2.33. $[\times(1+\varepsilon)]$-Approx #DNF is the restriction of $[\times(1+\varepsilon)]$-Approx #CSAT to C to formulas in *disjunctive normal form* (DNF) (i.e. an OR of clauses, where each clause is an AND of variables or their negations).

Theorem 2.34. For every function $\varepsilon(n) \geq 1/\text{poly}(n)$, $[\times(1+\varepsilon)]$-Approx #DNF is in **prBPP**.

Proof. It suffices to give a probabilistic polynomial-time algorithm that estimates the number of satisfying assignments to within a $1 \pm \varepsilon$ factor. Let $\varphi(x_1, \ldots, x_m)$ be the input DNF formula.

A first approach would be to apply random sampling as we have used above: Pick t random assignments uniformly from $\{0,1\}^m$ and evaluate φ on each. If k of the assignments satisfy φ, output $(k/t) \cdot 2^m$. However, if $\#\varphi$ is small (e.g. $2^{m/2}$), random sampling will be unlikely to hit any satisfying assignments, and our estimate will be 0

The idea to get around this difficulty is to embed the set of satisfying assignments, A, in a smaller set B so that sampling can be useful. Specifically, we will define sets A' and B satisfying the following properties:

(1) $|A'| = |A|$
(2) $A' \subseteq B$
(3) $|A'| \geq |B|/\text{poly}(n)$, where $n = |\varphi|$.
(4) We can decide membership in A' in polynomial time.
(5) $|B|$ computable in polynomial time.
(6) We can sample uniformly at random from B in polynomial time.

Letting ℓ be the number of clauses, we define A' and B as follows:

$$\begin{aligned} B &= \Big\{(i,\alpha) \in [\ell] \times \{0,1\}^m : \alpha \text{ satisfies the } i^{\text{th}} \text{ clause}\Big\} \\ A' &= \Big\{(i,\alpha) \in B : \alpha \text{ does not satisfy any clauses before the } i^{\text{th}} \text{ clause}\Big\} \end{aligned}$$

Now we verify the desired properties:

(1) Clearly $|A| = |A'|$ since A' only contains pairs (i, α) such that the first satisfying clause in α is the i^{th} one.
(2) Also, the size of A' and B can differ by at most a factor of ℓ by construction since for A' we only look at the first satisfying clause and there can only be $m-1$ more elements in B per assignment α.
(3) It is easy to decide membership in A' in linear time.
(4) $|B| = \sum_{i=1}^{\ell} 2^{m-m_i}$, where m_i is the number of literals in clause i.
(5) We can randomly sample from B as follows. First pick a clause with probability proportional to the number of satisfying assignments it has (2^{m-m_i}). Then, fixing those variables in the clause (e.g. if x_j is in the clause, set $x_j = 1$, and if $\neg x_j$ is in the clause, set $x_j = 0$), assign the rest of the variables uniformly at random.

Putting this together, we deduce the following algorithm:

Algorithm 2.35 ($[\times(1+\varepsilon)]$-Approx #DNF).
Input: a DNF formula $\varphi(x_1, \ldots, x_m)$ with ℓ clauses

(1) Generate a random sample of t points in $B = \{(i,\alpha) \in [\ell] \times \{0,1\}^m : \alpha \text{ satisfies the } i^{\text{th}} \text{ clause}\}$, for an appropriate choice of $t = O(1/(\varepsilon/\ell)^2)$ to be determined below.

(2) Let $\hat{\mu}$ be the fraction of sample points that land in $A' = \{(i,\alpha) \in B : \alpha \text{ does not satisfy any clauses before the } i^{\text{th}} \text{ clause}\}$.

(3) Output $\hat{\mu} \cdot |B|$.

By the Chernoff bound, we have $\hat{\mu} \in [|A'|/|B| \pm \varepsilon/\ell]$ with high probability (where we write $[\alpha \pm \beta]$ to denote the interval $[\alpha-\beta, \alpha+\beta]$). Thus, with high probability the output of the algorithm satisfies:

$$\hat{\mu} \cdot |B| \in [|A'| \pm \varepsilon|B|/\ell] \subseteq [|A| \pm \varepsilon|A|].$$

□

There is no deterministic polynomial-time algorithm known for this problem:

Open Problem 2.36. Give a deterministic polynomial-time algorithm for approximately counting the number of satisfying assignments to a DNF formula.

However, when we study pseudorandom generators in Chapter **??**, we will see a quasipolynomial-time derandomization of the above algorithm (i.e. one in time $2^{\text{polylog}(n)}$) (Problem **??**).

2.3.4 MaxCut

We give an example of another algorithm problem for which random sampling is a useful tool.

Definition 2.37. For a graph $G = (V, E)$ and $S, T \subseteq V$, define $\text{cut}(S,T) = \{\{u,v\} \in E : u \in S, v \in T\}$, and $\text{cut}(S) = \text{cut}(S, V \setminus S)$.

Computational Problem 2.38. MaxCut (search version): Given G, find the largest cut in G, i.e. the set S maximizing $|\text{cut}(S)|$.

Solving this problem optimally is **NP**-hard (in contrast to MinCut, which is known to be in **P**). However, there is a simple randomized algorithm that finds a cut of expected size at least $|E|/2$ (which is of course at least $1/2$ the optimal):

Algorithm 2.39 (MaxCut approximation).
Input: a graph $G = (V, E)$

Output a random subset $S \subseteq V$. That is, place each vertex v in S independently with probability $1/2$.

To analyze this algorithm, consider any edge $e = (u, v)$. Then the probability that e crosses the cut is 1/2. By linearity of expectations, we have:

$$\mathrm{E}[|\mathrm{cut}(S)|] = \sum_{e \in E} \Pr[e \text{ is cut}] = |E|/2.$$

This also serves as a proof, via the probabilistic method, that every graph (without self-loops) has a cut of size at least $|E|/2$.

In Chapter 3, we will see how to derandomize this algorithm. We note that there is a much more sophisticated randomized algorithm that finds a cut whose expected size is within a factor of .878 of the largest cut in the graph (and this algorithm can also be derandomized).

2.4 Random Walks and S-T Connectivity

2.4.1 Graph Connectivity

One of the most basic problems in computer science is that of deciding connectivity in graphs, i.e.

Computational Problem 2.40. S-T Connectivity: Given a directed graph G and two vertices s and t, is there a path from s to t in G?

This problem can of course be solved in linear time using breadth-first or depth-first search. However, these algorithms also require linear space. It turns out that S-T Connectivity can in fact be solved using much less workspace. (When measuring the space complexity of algorithms, we do not count the space for the (read-only) input and (write-only) output.)

Theorem 2.41. There is an algorithm deciding S-T Connectivity using space $O(\log^2 n)$ (and time $O(n)^{\log n}$).

Proof. The following recursive algorithm IsPath(G, u, v, k) decides whether there is a path of length at most k from u to v.

Algorithm 2.42 (Recursive S-T Connectivity).
IsPath(G, u, v, k):

```
(1) If $k = 0$, accept if $u = v$.
(2) If $k = 1$, accept if $u = v$ or $(u, v)$ is an edge in $G$.
(3) Otherwise, loop through all vertices $w$ in $G$ and accept if both IsPath$(G, u, w, \lceil k/2 \rceil)$ and
    IsPath$(G, w, v, \lfloor k/2 \rfloor)$ accept for some $w$.
```

We can solve S-T Connectivity by running IsPath(G, s, t, n), where n is the number of vertices in the graph. The algorithm has $\log n$ levels of recursion and uses $\log n$ space per level of recursion (to store the vertex w), for a total space bound of $\log^2 n$. Similarly, the algorithm uses linear time per level of recursion, for a total time bound of $O(n)^{\log n}$. □

It is not known how to improve the space bound in Theorem 2.41 or to get the running time down to polynomial while maintaining space $n^{o(1)}$. For *undirected graphs*, however, we can do much better using a randomized algorithm. Specifically, we can place the problem in the following class:

Definition 2.43. A language L is in **RL** if there exists a randomized algorithm A that always halts, uses space at most $O(\log n)$ on inputs of length n, and satisfies:

- $x \in L \Rightarrow \Pr[A(x) \text{ accepts}] \geq 1/2$.
- $x \notin L \Rightarrow \Pr[A(x) \text{ accepts}] = 0$.

Recall that our model of a randomized space-bounded machine is one that has access to a coin-tossing box (rather than an infinite tape of random bits), and thus must explicitly store in its workspace any random bits it needs to remember. The requirement that A always halts ensures that its running time is at most $2^{O(\log n)} = \text{poly}(n)$, because otherwise there would be a loop in its configuration space. Similarly to **RL**, we can define **L** (deterministic logspace), **co-RL** (one-sided error with errors only on NO instances), and **BPL** (two-sided error).

Now we can state the theorem for undirected graphs.

Computational Problem 2.44. UNDIRECTED S-T CONNECTIVITY: Given an undirected graph G and two vertices s and t, is there a path from s to t in G?

Theorem 2.45. UNDIRECTED S-T CONNECTIVITY is in **RL**.

Proof. [Sketch] The algorithm simply does a polynomial-length random walk starting at s.

Algorithm 2.46 (UNDIRECTED S-T CONNECTIVITY **via Random Walks).**

Input: (G, s, t), where $G = (V, E)$ has n vertices

```
(1) Let $v = s$.
(2) Repeat up to $n^4$ times:
    (a) If $v = t$, halt and accept.
    (b) Else let $v \overset{\text{R}}{\leftarrow} \{w : (v, w) \in E\}$.
(3) Reject (if we haven't visited $t$ yet).
```

Notice that this algorithm only requires space $O(\log n)$, to maintain the current vertex v as well as a counter for the number of steps taken. Clearly, it never accepts when there isn't a path from s to t. In the next section, we will prove that if G is a d-regular graph, then a random walk of length $\tilde{O}(d^2 n^3)$ from s will hit t with high probability. Note that this suffices for Theorem 2.45, because make an arbitrary undirected graph 3-regular while preserving s-t connectivity by replacing each vertex v with a cycle of length $\deg(v)$. In fact, the algorithm actually works as described above for general undirected graphs and even directed graphs in which each connected component is Eulerian (indeg(v) = outdeg(v) for every vertex), but we will not prove it here. But it does not work for arbitrary directed graphs. Indeed, it is not difficult to construct directed graphs in which there is a path from s to t but a random walk from s takes exponential time to hit t (Problem 2.9). $\square$

This algorithm, dating from the 1970's, was derandomized only in 2005. We will cover this result in Section 4.4. However, the general question of derandomizing space-bounded algorithms remains open.

Open Problem 2.47. Does **RL** = **L**? Does **BPL** = **L**?

2.4.2 Random Walks on Graphs

For generality that will be useful later, many of the definitions in this section will be given for *directed multigraphs* (which we will refer to as *digraphs* for short). By multigraph, we mean that we allow G to have parallel edges and self-loops. We call such a digraph *d-regular* if every vertex has indegree d and outdegree d. To analyze the random-walk algorithm of the previous section, it suffices to prove a bound on the *hitting time* of random walks.

Definition 2.48. For a digraph $G = (V, E)$, we define its *hitting time* as

$$\mathrm{hit}(G) = \max_{i,j \in V} \min_{t} \{\Pr[\text{a random walk of length } t \text{ started at } i \text{ visits } j] \geq 1/2\}.$$

We note that hit(G) is often defined as the maximum over vertices i and j of the *expected* time for a random walk from i to visit j. The two definitions are the same upto a factor of 2, and the above is more convenient for our purposes.

We will prove:

Theorem 2.49. For every connected and d-regular undirected graph G on n vertices, we have hit$(G) = O(d^2 n^3 \log n)$.

There are combinatorial methods for proving the above theorem, but we will prove it using a linear-algebraic approach, as the same methods will be very useful in our study of expander graphs. For an n-vertex digraph G, we define its *random-walk transition matrix*, or *random-walk matrix* for short, to be the $n \times n$ matrix M where $M_{i,j}$ is the probability of going from vertex i to vertex j in one step. That is, $M_{i,j}$ is the number of edges from i to j divided by the outdegree of i. In case G is d-regular, M is simply the adjacency matrix of G divided by d. Notice that for every probability distribution $\pi \in \mathbb{R}^n$ on the vertices of G (written as a row vector), the vector πM is the probability distribution obtained by selecting a vertex i according to π and then taking one step of the random walk to end at a vertex j. This is because $(\pi M)_j = \sum_i \pi_i M_{i,j}$.

In our application, we start at a probability distribution π concentrated at vertex s, and are interested in the distribution πM^k we get after taking k steps on the graph. Specifically, we'd like to show that it places nonnegligible mass on vertex t for $k = \mathrm{poly}(n)$. We will do this by showing that it in fact converges to the uniform distribution $u = (1/n, 1/n, \ldots, 1/n) \in \mathbb{R}^n$ within a polynomial number of steps. Note that $uM = u$ by the regularity of G, so convergence to u is possible (and will be guaranteed given some additional conditions on G).

We will measure the rate of convergence in ℓ_2 norm. For vectors $x, y \in \mathbb{R}^n$, we will use the standard inner product $\langle x, y\rangle = \sum_i x_i y_i$, and ℓ_2 norm $\|x\| = \sqrt{\langle x, x\rangle}$. We write $x \perp y$ to mean that x and y are orthogonal, i.e. $\langle x, y\rangle = 0$. We want to determine how large k needs to be so that $\|\pi M^k - u\|$ is "small". This is referred to as the *mixing time* of the random walk. Mixing time can be defined with respect to various distance measures and the ℓ_2 norm is not the most natural one, but it has the advantage that we will be able to show that the distance decreases noticeably in every step. This is captured by the following quantity.

Definition 2.50. For a regular digraph G with random-walk matrix M, we define

$$\lambda(G) \stackrel{\text{def}}{=} \max_{\pi} \frac{\|\pi M - u\|}{\|\pi - u\|} = \max_{x \perp u} \frac{\|xM\|}{\|x\|},$$

where the first maximization is over all *probability distributions* $\pi \in [0,1]^n$ and the second is over all vectors $x \in \mathbb{R}^n$ such that $x \perp u$. We write $\gamma(G) \stackrel{\text{def}}{=} 1 - \lambda(G)$.

To see that the first definition of $\lambda(G)$ is smaller than or equal to the second, note that for any probability distribution π, the vector $x = (\pi - u)$ is orthogonal to uniform (i.e. the sum of its entries is zero). For the converse, observe that given any vector $x \perp u$, the vector $\pi = u + \alpha x$ is a probability distribution for a sufficiently small α. It can be shown that $\lambda(G) \in [0, 1]$. (For undirected regular graphs, this follows from Problem 2.11.)

The following lemma is immediate from the definition of M.

Lemma 2.51. Let G be a regular digraph with random-walk matrix M. For every initial probability distribution π on the vertices of G and every $k \in \mathbb{N}$, we have

$$\|\pi M^k - u\| \leq \lambda(G)^k \cdot \|\pi - u\| \leq \lambda(G)^k.$$

Thus a smaller value of $\lambda(G)$ (equivalently, a larger value of $\gamma(G)$) means that the random walk mixes more quickly. Specifically, for $k = \ln(n/\varepsilon)/\gamma(G)$, it follows that every entry of πM^k has probability mass at least $1/n - (1 - \gamma(G))^k \geq (1 - \varepsilon)/n$. So the mixing time of the random walk on G is at most $O((\log n)/\gamma(G))$, and this holds with respect to any reasonable distance measure. Note that $O(1/\gamma(G))$ steps does not suffice, because a distribution with ℓ_2 distance ε from uniform could just assign equal probability mass to $1/\varepsilon^2$ vertices (and thus be very far from uniform in any intuitive sense).

Corollary 2.52. $\mathrm{hit}(G) = O(n \log n/\gamma(G))$.

Proof. As argued above, a walk of length $k = O(\log n/\gamma(G))$ has a probability of at least $1/2n$ of ending at j. Thus, if we do $O(n)$ such walks, we will hit j with probability at least $1/2$. □

Thus we are left with the task of showing that $\gamma(G) \geq 1/\mathrm{poly}(n)$. This is done in Problem 2.11, using a connection with eigenvalues described in the next section.

2.4.3 Eigenvalues

Recall that $v \in \mathbb{R}^n$ is an *eigenvector* of $n \times n$ matrix M if $vM = \lambda v$ for some $\lambda \in \mathbb{R}$, which is called the corresponding *eigenvalue*. A useful feature of *symmetric* matrices is that they can be described entirely in terms of their eigenvectors and eigenvalues.

Theorem 2.53 (Spectral Thm for Symmetric Matrices). If M is a symmetric $n \times n$ real matrix with distinct eigenvalues $\mu_1, \ldots, \mu_k$, then the subspaces $W_i = \{x : x \text{ is an eigenvector of eigenvalue } \mu_i\}$ are orthogonal (i.e. $x \in W_i,\ y \in W_j \Rightarrow x \perp y$ if $i \neq j$) and span $\mathbb{R}^n$ (i.e. $\mathbb{R}^n = W_1 + \cdots + W_k$). We refer to the dimension of W_i as the *multiplicity* of eigenvalue μ_i. In particular, $\mathbb{R}^n$ has a basis consisting of orthogonal eigenvectors $v_1, \ldots, v_n$ having respective eigenvalues $\lambda_1, \ldots, \lambda_n$, where the number of times μ_i occurs among the λ_j's exactly equals the multiplicity of μ_i.

Notice that if G is a undirected regular graph, then its random-walk matrix M is symmetric. We know that $uM = u$, so the uniform distribution is an eigenvector of eigenvalue 1. Let $v_2, \ldots, v_n$ and $\lambda_2, \ldots, \lambda_n$ be the remaining eigenvectors and eigenvalues, respectively. Given any probability distribution π, we can write it as $\pi = u + c_2v_2 + \cdots + c_nv_n$. Then the probability distribution after k steps on the random walk is

$$\pi M^k = u + \lambda_2^k c_2 v_2 + \cdots + \lambda_n^k c_n v_n.$$

In Problem 2.11, it is shown that all of the λ_i's have absolute value at most 1. Notice that if they all have have magnitude strictly smaller than 1, then πM^k indeed converges to u. Thus it is not surprising that our measure of mixing rate, $\lambda(G)$, equals the absolute value of the second largest eigenvalue.

Lemma 2.54. Let G be an undirected graph with random-walk matrix M. Let $1 = \lambda_1 \geq |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n|$ be the eigenvalues of M. Then $\lambda(G) = |\lambda_2|$.

Proof. Let $u = v_1, v_2, \ldots, v_n$ be the basis of orthogonal eigenvectors corresponding to the λ_i's. Given any vector $x \perp u$, we can write $x = c_2v_2 + \cdots + c_nv_n$. Then:

$$\begin{aligned}
\|xM\|^2 &= \|\lambda_2 c_2 v_2 + \cdots + \lambda_n c_n v_n\|^2 \\
&= \lambda_2^2 c_2^2 \|v_2\|^2 + \cdots + \lambda_n^2 c_n^2 \|v_n\|^2 \\
&\leq |\lambda_2|^2 \cdot (c_2^2\|v_2\|^2 + \cdots + c_n^2\|v_n\|^2) \\
&= |\lambda_2|^2 \cdot \|x\|^2
\end{aligned}$$

Equality is achieved with $x = v_2$. □

Thus, bounding $\lambda(G)$ amounts to bounding the eigenvalues of G. Due to this connection, $\gamma(G) = 1 - \lambda(G)$ is often referred to as the *spectral gap*, as it is the gap between the largest eigenvalue and the second largest.

In Problem 2.11, it is shown that:

Theorem 2.55. If G is a connected, nonbipartite, and regular undirected graph, then $\gamma(G) = \Omega(1/(dn)^2)$.

Combining Theorem 2.55 with Corollary 2.52, we deduce Theorem 2.49. (The nonbipartite assumption in Theorem 2.55 can be achieved by adding a self-loop to each vertex, which only increases the hitting time.) We note that the bounds presented here are not tight.

2.4.4 Markov Chain Monte Carlo

Random walks are a very widely used tool in the design of randomized algorithms. In particular, they are the heart of the "Markov Chain Monte Carlo" method, which is widely used in statistical physics and for solving approximate counting problems. In these applications, the goal is to generate a random sample from an *exponentially* large space, such as an (almost) uniformly random perfect matching for a given bipartite graph G. (It turns out that this is equivalent to approximately counting the number of perfect matchings in G.) The approach is to do a random walk on an appropriate (regular) graph $\hat{G}$ defined on the space (e.g. by doing random local changes on the current perfect matching). Even though $\hat{G}$ is typically of size exponential in the input size $n = |G|$, in many cases it can be proven to have mixing time $\mathrm{poly}(n) = \mathrm{polylog}(|\hat{G}|)$, a property referred to as *rapid mixing*. These Markov Chain Monte Carlo methods provide some of the best examples of problems where randomization yields algorithms that are exponentially faster than all known deterministic algorithms.

2.5 Exercises

~~**Problem 2.1 (Schwartz–Zippel lemma).** Prove Lemma 2.4: If $p(x_1, \ldots, x_n)$ is a nonzero poly-~~ nomial of degree d over a a field (or integral domain) $\mathbb{F}$ and $S \subseteq \mathbb{F}$, then

$$\Pr_{\alpha_1,\ldots,\alpha_n \xleftarrow{\mathrm{R}} S}[p(\alpha_1, \ldots, \alpha_n) = 0] \leq \frac{d}{|S|}.$$

You may use the fact that every nonzero *univariate* polynomial of degree d over $\mathbb{F}$ has at most d roots.

Problem 2.2 (Robustness of the model). Suppose we modify our model of randomized computation to allow the algorithm to obtain a random element of $\{1, \ldots, m\}$ for any number m whose binary representation it has already computed (as opposed to just allowing it access to random bits). Show that this would not change the classes **BPP** and **RP**.

Problem 2.3 (Zero error vs. 1-sided error). Prove that **ZPP** = **RP** $\cap$ **co-RP**.

Problem 2.4 (Identity Testing for integer circuits). In this problem, you will show how to do Identity Testing for arithmetic *circuits* over the integers. The Prime Number Theorem says that the number of primes less than T is $(1 \pm o(1)) \cdot T/\ln T$, where the $o(1)$ tends to 0 as $T \to \infty$. You may use this fact in the problem below.

(1) Show that if N is a nonzero integer and $M \overset{\text{R}}{\leftarrow} \{1, \ldots, \log^2 N\}$, then

$$\Pr[N \not\equiv 0 \pmod{M}] = \Omega(1/\text{loglog}N).$$

(2) Use the above to prove Theorem 2.12: Identity Testing for arithmetic *circuits* over $\mathbb{Z}$ is in **co-RP**.

Problem 2.5 (Identity Testing via Modular Reduction). In this problem, you will analyze an alternative to the algorithm seen in class, which directly handles polynomials of degree larger than the field size. It is based on the same idea as Problem 2.4, using the fact that polynomials over a field have many of the same algebraic properties as the integers.

The following definitions and facts may be useful: A polynomial $p(x)$ over a field $\mathbb{F}$ is called *irreducible* if it has no nontrivial factors (i.e. factors other than constants from $\mathbb{F}$ or constant multiples of p). Analogously to prime factorization of integers, every polynomial over $\mathbb{F}$ can be factored into irreducible polynomials and this factorization is unique (up to reordering and constant multiples). It is known that the number of irreducible polynomials of degree at most d over a field $\mathbb{F}$ is at least $\mathbb{F}^{d+1}/2d$. (This is similar to the Prime Number Theorem for integers mentioned in Problem 2.4, but is easier to prove.) For polynomials $p(x)$ and $q(x)$, $p(x)$ mod $q(x)$ is the remainder when p is divided by q. (More background on polynomials over finite fields can be found in the references listed in Section 2.6.)

In this problem, we consider a version of the Identity Testing problem where a polynomial $p(x_1, \ldots, x_n)$ over finite field $\mathbb{F}$ is presented as a formula built up from elements of $\mathbb{F}$ and the variables $x_1, \ldots, x_n$ using addition, multiplication, and *exponentiation* with exponents given in *binary*. We also assume that we are given a representation of $\mathbb{F}$ enabling addition, multiplication, and division in $\mathbb{F}$ to be done quickly.

(1) Let $p(x)$ be a univariate polynomial of degree $\leq D$ over a field $\mathbb{F}$. Prove that there is a constant c such that if $p(x)$ is nonzero (as a formal polynomial) and $q(x)$ is a randomly selected polynomial of degree at most $d = c \log D$, then the probability that $p(x)$ mod $q(x)$ is nonzero is at least $1/c \log D$. Deduce a randomized, polynomial-time identity test for *univariate* polynomials presented in the above form.
(2) Obtain an identity test for multivariate polynomials by reduction to the univariate case.

Problem 2.6. (PRIMALITY)

(1) Show that for every positive integer n, the polynomial identity $(x+1)^n \equiv x^n + 1 (\bmod n)$ holds iff n is prime.
(2) Obtain a **co-RP** algorithm for the language PRIMALITY$= \{n : n \text{ prime}\}$ using Part 1 together with the previous problem. (In your analysis, remember that the integers modulo n are a field only when n is prime.)

Problem 2.7 (A Chernoff Bound). Let $X_1, \ldots, X_t$ be independent $[0,1]$-valued random variables, and $X = \sum_{i=1}^{t} X_i$.

(1) Show that for every $r \in [0, 1/2]$, $\mathrm{E}[e^{rX}] \leq e^{r\,\mathrm{E}[X]+r^2 t}$. (Hint: $1 + x \leq e^x \leq 1 + x + x^2$ for all $x \in [0, 1/2]$.)
(2) Deduce the following Chernoff Bound: $\Pr\left[X \geq \mathrm{E}[X] + \varepsilon t\right] \leq e^{-\varepsilon^2 t/4}$. Where did you use the independence of the X_i's?

Problem 2.8 (Necessity of Randomness for Identity Testing*). In this problem, we consider the "oracle version" of the identity testing problem, where an arbitrary polynomial $p : \mathbb{F}^m \to \mathbb{F}$ of degree d is given as an oracle (ie black box) and the problem is to test whether $p = 0$. Show that any deterministic algorithm that solves this problem when $m = d = n$ must make at least 2^n queries to the oracle (in contrast to the randomized identity testing algorithm from class, which makes only one query provided that $|\mathbb{F}| \geq 2n$).

Is this a proof that $\mathbf{P} \neq \mathbf{RP}$? Explain.

Problem 2.9 (Random Walks on Directed Graphs). Show that for every n, there exists a digraph G with n vertices, outdegree 2, and $\mathrm{hit}(G) = 2^{\Omega(n)}$.

Problem 2.10. Let G be a regular digraph with random-walk matrix M.

(1) Show that $\lambda(G)$ is the square root of the absolute value of the second-largest eigenvalue of the symmetric matrix MM^T.
(2) Describe the graph for which MM^T is the random-walk matrix.

Problem 2.11 (Spectral Graph Theory). Let M be the random-walk matrix for a d-regular *undirected* graph $G = (V, E)$ on n vertices. We allow G to have self-loops and multiple edges. Recall that the uniform distribution (or all-ones vector) is an eigenvector of M of eigenvalue $\lambda_1 = 1$. Prove the following statements. (Hint: for intuition, it may help to think about what the statements mean for the behavior of the random walk on G.)

(1) All eigenvalues of M have absolute value at most 1.
(2) G is disconnected $\iff$ 1 is an eigenvalue of multiplicity at least 2.
(3) Suppose G is connected. Then G is bipartite $\iff$ -1 is an eigenvalue of M.
(4) G connected $\Rightarrow$ all eigenvalues of M other than λ_1 are $\leq 1 - 1/\mathrm{poly}(n, d)$. To do this, it may help to first show that the second largest eigenvalue of M (not necessarily in absolute value) equals

$$\max_x \langle Ax, x\rangle = 1 - \frac{1}{d} \cdot \min_x \sum_{(i,j) \in E} (x_i - x_j)^2,$$

where the maximum/minimum is taken over all vectors x of length 1 such that $\sum_i x_i = 0$, and $\langle x, y\rangle = \sum_i x_i y_i$ is the standard inner product. For intuition, consider restricting the above maximum/minimum to $x \in \{+\alpha, -\beta\}^n$ for $\alpha, \beta > 0$.
(5) G connected and nonbipartite $\Rightarrow$ all eigenvalues of M (other than 1) have absolute value at most $1 - 1/\mathrm{poly}(n, d)$ and thus $\lambda(G) \leq 1 - 1/poly(n, d)$.
(6*) Extra credit: Establish the (tight) bound $1 - \Omega(1/d \cdot D \cdot n)$ in Part 4, where D is the diameter of the graph, and show that a simple graph satisfies $D \leq O(n/d)$. (The $1 - \Omega(1/d \cdot D \cdot n)$ bound also holds for Part 5, but you do not need to prove it here.)

2.6 Chapter Notes and References

Recommended textbooks on randomized algorithms are Motwani–Raghavan [MR] and Mitzenmacher–Upfal [MU]. The randomized algorithm for Identity Testing was independently discovered by DeMillo and Lipton [DL], Schwartz [Sch], and Zippel [Zip]. A deterministic polynomial-time Identity Testing algorithm for formulas in $\Sigma\Pi\Sigma$ form with a constant number of terms was given by Kayal and Saxena [KS], improving a previous quasipolynomial-time algorithm of Dvir and Shpilka [DS]. Problem 2.8 is from [LV].

Recommended textbooks on abstract algebra and finite fields are [Art, LN].

The randomized algorithm for Perfect Matching is due to Lovász, who also showed how to extend the algorithm to non-bipartite graphs. An efficient parallel randomized algorithm for *finding* a perfect matching was given by Karp, Upfal, and Wigderson [KUW] (see also [MVV]). A randomized algorithm for finding a perfect matching in the same sequential time complexity as Lovász's algorithm was given recently by Mucha and Sankowski [MS] (see also [Har]).

For more on parallel algorithms, we refer to the textbook by Leighton [Lei]. The Identity Testing and Primality algorithms of Problems 2.5 and 2.6 are due to Agrawal and Biswas [AB]. Agrawal, Kayal, and Saxena [AKS1] derandomized the Primality algorithm to prove that Primality is in **P**.

The randomized complexity classes **RP**, **BPP**, **ZPP**, and **PP** were formally defined by Gill [Gil], who conjectured that $\mathbf{BPP} \neq \mathbf{P}$ (in fact $\mathbf{ZPP} \neq \mathbf{P}$). Chernoff Bounds are named after H. Chernoff [Che]; the version in Theorem 2.21 is due to Hoeffding [Hoe] and is sometimes referred to as Hoeffding's Inequality. For some other Chernoff Bounds, see [MR]. Problem 2.3 is due to Rabin (cf. [Gil]).

The computational perspective on sampling, as introduced in Section 2.3.1, is surveyed in [Gol1, Gol2]. SAMPLING is perhaps the simplest example of a computational problem where randomization enables algorithms with running time sublinear in the size of the input. Such *sublinear-time algorithms* are now known for a wide variety of interesting computational problems; see the surveys [Ron, Rub].

Promise problems were introduced by Even, Selman, and Yacobi [ESY]. For survey of their role in complexity theory, see Goldreich [Gol5].

The randomized algorithm for $[\times(1+\varepsilon)]$-APPROX #DNF is due to Karp and Luby [KLM]. A 1/2-approximation algorithm for MAXCUT was first given in [SG]; that algorithm can be viewed as a natural derandomization of Algorithm 2.39. (See Algorithm 3.17.) The .878-approximation algorithm was given by Goemans and Williamson [GW].

The $O(\log^2 n)$-space algorithm for S-T CONNECTIVITY is due to Savitch [Sav]. Using the fact that S-T CONNECTIVITY (for *directed* graphs) is complete for nondeterministic logspace (**NL**), this result is equivalent to the fact that $\mathbf{NL} \subseteq \mathbf{L}^2$, where $\mathbf{L}^c$ is the class of languages that can be decided deterministic space $O(\log^2 n)$. The latter formulation (and its generalization $\mathbf{NSPACE}(s(n)) \subseteq \mathbf{DSPACE}(s(n)^2)$) is known as *Savitch's Theorem.* The randomized algorithm for UNDIRECTED S-T CONNECTIVITY was given by Aleliunas, Karp, Lipton, Lovász, and Rackoff [AKL$^+$], and was recently derandomized by Reingold [Rei] (see Section 4.4). For more background on random walks, mixing time, and the Markov Chain Monte Carlo Method, we refer the reader to [MU, Ran].

The bound on hitting time given in Theorem 2.49 is not tight; for example, it can be improved to $\Theta(n^2)$ for regular graphs that are simple (have no self-loops or parallel edges) [KLNS].

Even though we will focus primarily on undirected graphs (for example, in our study of expanders in Chapter 4), much of what we do generalizes to regular digraphs, or more generally to digraphs where every vertex has the same indegree as outdegree (i.e. where each connected component is Eulerian). See e.g. [Mih, Fil, RTV]. Problem 2.10 is from [Fil].

The Spectral Theorem (Thm. 2.53) can be found in any standard textbook on linear algebra. Problem 2.11, Part 5 is from [Lov2]. Spectral Graph Theory is a rich subject, with many applications beyond the scope of this course; see the survey by Spielman [Spi] and references therein.

One significant omission from this chapter is the usefulness of randomness for *verifying proofs.* Recall that **NP** is the class of languages having membership proofs that can be verified in **P**. Thus it is natural to consider proof verification that is probabilistic, leading to the class **MA**, as well as a larger class **AM**, where the proof itself can depend on the randomness chosen by the verifier. (These are both subclasses of the class **IP** of languages having *interactive proof systems.*) There are languages, such as GRAPH NONISOMORPHISM, that are in **AM** but are not known to be in **NP** [GMW]. "Derandomizing" these proof systems (e.g. proving $\mathbf{AM} = \mathbf{NP}$) would show that GRAPH NONISOMORPHISM is in **NP**, i.e. that there are short proofs that two graphs are nonisomorphic. For more about interactive proofs, see [Vad, Gol6, AB].