Doctoral mini-course on expander graphs
December 5, 2024 – Lecture 2

1 Introduction

We recall that d is an integer. An (n, d, λ)-graph is an n-vertex graph which is d-regular and whose second largest eigenvalue in absolute value is equal to λ.
We are interested in *families of expanders graphs*, that is families of (n, d, λ)-graphs with d and $\lambda = o(d)$ fixed, and n arbitrarily large. We know that we can construct such (n, d, λ)-graphs with $\lambda = O(\sqrt{d})$ in a strongly explicit way.
In this lecture we will see two applications of expander graphs in algorithms and complexity. We start with a number of useful facts about random walks in graphs.

2 Random walks

We start at a given vertex x_0 of a graph G, and pick a neighbor x_1 of x_0 uniformly at random, then a neighbor x_2 of x_1 at random, and so on.
Then $x_0, x_1, \dots, x_t$ is called a *random walk of length t* starting at x_0.

If we start with a probability distribution σ on $V(G)$ instead of a single vertex x_0, then after t steps we have a new probability distribution σ^t.

Let π be the probability distribution on $V(G)$ such that $\pi(v) = d(v)/2m$ (the probability of taking a vertex is proportional to its degree). Note that when G is regular, π is just the uniform distribution.

π is called the *stationary distribution*; it is easy to see that if we start with π, then after any number of steps of the random walk the distribution is still π (it is enough to show that it is true after a single step : $\mathbb{P}(v) = \sum_{uv \in E} d(u)/2m \cdot 1/d(u) = d(v)/2m = \pi(v)$).

Theorem 1. *If G is connected and non-bipartite, then for any initial distribution σ, $\sigma^t \to \pi$ (in total variation distance) as $t \to \infty$.*

In practice what we would like to bound is the number of steps until the random walks is close to the uniform distribution (sampling from such a random walk is sometimes the only way to sample from very large families of objects).

We can view a probability distribution on the vertex set $V(G)$ of a graph G as some vector of $\mathbb{R}^n$ (recall that n always denotes the number of vertices of G). If we take a random vertex v of G according to some probability distribution $x \in \mathbb{R}^n$, and then move to some random neighbor of v (uniformly at random among all neighbors of v), then what is the new probability distribution ? Well it is precisely $(\frac{1}{d}A_G)x$, where A_G denotes the adjacency matrix of G. In particular after a random walk of t steps the distribution is equal to $(\frac{1}{d}A_G)^t x$.

Theorem 2. *Let G be an (n, d, λ)-graph, and let $\sigma \in \mathbb{R}^n$ be a probability distribution on $V(G)$. Then for any integer t,*

$$\|\sigma^t - \pi\|_1 = \|(\tfrac{1}{d}A_G)^t\sigma - \pi\|_1 \leq \sqrt{n}(\lambda/d)^t.$$

So whenever $\lambda \leq d/2$ (for instance), for any $\varepsilon > 0$, after $t = \Omega(\log n + \log \varepsilon^{-1})$ steps, σ^t is ε-close to the uniform distribution (even if the original distribution σ is concentrated on a single vertex). Note that this provides another proof that expander graphs have diameter $O(\log n)$.

We will also need the following related results.

Theorem 3. *Let G be an (n, d, λ)-graph. Suppose we take a vertex x_0 uniformly at random in G, and then perform a random walk $x_0, \ldots, x_t$ of length t starting at x_0. Then for any subset $S \subseteq V(G)$, the probability that $x_0, x_1, \ldots, x_t$ are all in S is at most $(|S|/n + \lambda/d)^t$. Moreover, if $|S|/n \geq 6\lambda/d$, then this probability is at least $(|S|/n - 2\lambda/d)^t$.*

Note that both bounds are very close to $(|S|/n)^{t+1}$ which corresponds to the setting were all the points are taken uniformly at random

3 Randomized algorithms

We consider *Monte Carlo* algorithms, which are a class of randomized algorithms whose running time is independent of the random bits. These algorithms can make some errors, but with bounded probability (say $\frac{1}{2}$). We can view such an algorithm as taking a string of k random bits (for some k) and then running deterministically according to the chosen string.

For simplicity we will only consider *one-sided* algorithms, that is algorithms that do not err on positive instances, but might be wrong on negative instances, or symmetrically algorithms that do not err on negative instances, but might be wrong on positive instances. A typical example is the *Polynomial Identity Testing* problem where one is given two multivariate polynomials of bounded total degree over some finite field as a black-box, and the goal is to decide whether the polynomials are equal. The classical randomized algorithm evaluates the two polynomials on some random elements and decides that the two polynomial are equal if their values agree on the chosen random elements. If they are indeed equal, then the algorithm gives the correct answer and if they are distinct there is a good probability that the two values of the polynomials would be different on the random elements (assuming the field is larger than the degree). Note that finding a deterministic algorithm (which also runs in polynomial time) for this problem is a major open problem: it is one of the very few problems for which an efficient randomized algorithm is known, but no deterministic algorithm has been found.

In the definition above we assumed that the randomized algorithm we consider errs with probability at most $\frac{1}{2}$. But what if we want this probability to be much smaller, say at most ε ? Say algorithm $\mathcal{A}$ always gives the correct answer on positive instances. Then a simple trick is to repeat the execution of the algorithm t times (on fresh new random bits at every execution). If $\mathcal{A}$ always outputs yes, we answer yes. If $\mathcal{A}$ answers no at least once, we answer no. Let $\mathcal{B}$ be the resulting algorithm. If the instance is positive, then $\mathcal{A}$ answers yes at every execution and thus $\mathcal{B}$ answers yes (so we still have a one-sided algorithm). If the instance is negative, then the probability that $\mathcal{A}$ answered yes t times is at most $\frac{1}{2^t}$, which is at most ε for $t = \log \varepsilon^{-1}$.

This is all very nice, but true random bits are very expensive and we would like to avoid using too many of them. Say $\mathcal{A}$ uses k random bits. Then $\mathcal{B}$ will use $t \cdot k = k \log \varepsilon^{-1}$ bits, and we would like to be much more efficient than that.

For this we will use the results of the previous section. Consider an (n, d, λ)-graph G with $n = 2^k$ and $\lambda \leq d/4$. Note that the vertices of G are in one-to-one correspondence with the sequences of k bits, and thus all the possible executions of $\mathcal{A}$ (on some fixed input).

We start by taking a vertex x_0 of G uniformly at random, and then we perform a random $x_0, \ldots, x_t$ of length t starting at x_0 (note that this costs

$k + t \log d = k + O(t)$ random bits).

We execute $\mathcal{A}$ on the vertices $x_0, \ldots, x_t$ and again answer yes if and only all the executions give yes as an answer. Let $\mathcal{B}'$ be the new algorithm. Again if the instance is positive, $\mathcal{B}'$ answers correctly. If the instance is negative, the probability of error is the probability that all vertices $x_0, \ldots, x_t$ of G lie in the subset $S \subseteq V(G)$ corresponding to the executions of $\mathcal{A}$ where $\mathcal{A}$ answers incorrectly. By definition $|S| \leq n/2$ and thus it follows from Theorem 3 that the probability of error is at most $(1/2 + \lambda/d)^t \leq (3/4)^t$. So if $t = \log \varepsilon^{-1}$ the probability of error is at most ε. Note that we do not need to construct G (whose size in 2^k) to run the algorithm, we only need to select a random starting vertex x_0, and then a sequence of random neighbors, and for this a strongly explicit description of G (telling us the list of neighbors of each vertex in time polynomial in k) is enough.

To sum up: if we simply repeat the original algorithm on new (independent) random bits, it costs us $tk = k \log \varepsilon^{-1}$ random bits to obtain an error of at most ϵ. If we use a random walk in an expander grah to do the sampling it costs us only $k + O(\log \varepsilon^{-1})$ random bits.

Note that we have been considering only one-sided randomized algorithms throughout, but everything works equally well with two-sided algorithms (you just have to use the majority of the outcomes of the executions of the algorithm, and use a more general version of Theorem 3 where each x_i is prescribed to be in some set S_i).

The moral of this section: random walks starting from a random vertex in an expander graph behave very much like independent random samples. This can be made more precise, with for instance a Chernoff-type bound on the concentration of the sum of random variable taken from such a random walk.

4 Hardness of approximation

Recall that the class NP is the class of problems for which positive instances have certificates that can be checked in polynomial time (in the size of the instance). For instance if a graph G has a proper 3-coloring, the certificate is the 3-coloring itself: a verifier can check that for all pairs of adjacent vertices, the colors assigned to the two vertices are distinct, in polynomial time.

A major result in computational complexity from the 1990s, the PCP theorem (where PCP stands for Probabilistically Checkable Proof), states that problems in NP have another very useful type of certificates, still of polynomial size: here the verifier starts by looking at the instance and using some randomized algorithm (which only relies on $O(\log n)$ random bits) decides a constant number of random locations. Then some prover presents a certificate of polynomial size, and the verifier only checks the constant number of locations it chose, and based only on this constant number of bits, decides to accept or reject the proof. If the instance is positive, then the verifier must accept the proof, and if the instance is negative, then the verifier must reject with probability at least $\frac{1}{2}$, for any certificate.

The PCP theorem can equivalently be stated as a result on the hardness of approximating combinatorial problems. Recall that a *clique* is a graph is a set of pairwise adjacent vertices. The *clique number* of G, denoted by $\omega(G)$, is the maximum size of a clique in G.

Theorem 4. *There exist constants $0 < a < b < 1$ such that given an n-vertex graph G for which*

(1) $\omega(G) \leq an$, or

(2) $\omega(G) \geq bn$,

then unless P=NP, there is no polynomial time algorithm that can make the distinction between the two cases (1) and (2).

To explain the connection, we briefly mention the problem 3-SAT: we have a formula of the type $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_4 \vee \neg x_5) \wedge \ldots$, where there are 3 literals per clause and the x_i's are boolean variables, and the goal is to decide whether the formula can be satisfied (so all clauses have to be satisfied) by some assignment of values to the x_i's. By a classical hardness reduction between the clique problem and 3-SAT, it can be shown that Theorem 4 is equivalent to the statement that there is some $\varepsilon > 0$ such that if we are given a 3-SAT formula in which (1) all clauses can be satisfied, or (2) only a fraction of at most $1 - \varepsilon$ of the clauses can be satisfied, then no polynomial time algorithm can make the distinction between (1) and (2), unless P=NP. Now you can reduce any problem in NP to such gap version of 3-SAT, and this version has a simple probabilistically verifiable certificate, consisting of the values of the variables x_i. The verifier just checks a constant number of clauses selected uniformy at random and says that the formula is satisfiable if all these clauses are satisfied. If (1) all clauses can be satisfied, then the verifier is always correct and if (2) only a fraction of at most $1-\varepsilon$ of the clauses can be satisfied then in any assignment of variables, a fraction at least ε of the clauses will be violated and the verifier will check such a clause with constant probability, and thus see that the certificate is incorrect.

Theorem 4 says that we cannot approximate the clique number of a graph in polynomial time within some (small) constant multiplicative factor, unless $\mathsf{P}=\mathsf{NP}$. Next week we will see how to amplify this gap to an arbitrarily large constant, and then even to some polynomial in n, as a direct application of Theorem 3. We will also see another key application of expander graphs in algorithms and complexity.