Bias Amplification via Walks on Expanders

Omshi Samal

1 Background

A linear code $C \subseteq F_2^n$ is defined to be *ϵ-balanced* when

$$\forall c \in C, \frac{1-\epsilon}{2} \leq \text{weight}(c) := \frac{1}{n}\sum_{i=1}^{n} c_i \leq \frac{1+\epsilon}{2}.$$

If C is a ϵ-balanced code, then $\text{dist}(C) = \min_{c \in C, c \neq 0} \text{weight}(c) \geq \frac{1-\epsilon}{2}$. Moreover, for any string $f : [n] \to \mathbb{F}_2$ in the code C, we have

$$\text{Bias}(f) = \left| \mathbb{E}_{v \in [n]} \left[(-1)^{f(v)} \right] \right| \leq \epsilon.$$

So, if we have an ϵ-balanced code C (which then has distance $\geq \frac{1-\epsilon}{2}$) and we want to increase its distance (that is, reduce ϵ), we can think of that as *amplifying the bias* of the code [[RR23]]. One way to amplify the bias is by taking walks over expanders. Given an expander G on n vertices, let W_k be the set of all walks of length k on G. Then for any string $f : [n] \to \mathbb{F}_2$, we can construct a longer string $g : [|W_k|] \to \mathbb{F}_2$ as:

$$g(w) = \oplus_{i \in [k]} f(w_i)$$

where w_i is the i-th index of walk $w \in W_k$.

As it turns out, if we take graph G to be a Ramanujan expander of degree d and expansion λ, and have $\text{Bias}(f) \leq \sqrt{\lambda}$, then $\text{Bias}(g) = O(\lambda^{k/2})$. However, this procedure is quite costly as now the length of g is $|W_k|$; so we have that if the distance of the amplified code is taken to be $\frac{1-\epsilon}{2}$, the rate of this amplified code is $\Omega(\epsilon^4)$. Tashma's construction [[Ta-17]] uses a pseudorandom set of walks $V_k \subset W_k$ to get codes with rate $\Omega(\epsilon^{2+o(1)})$, while distance is still $\frac{1-\epsilon}{2}$, which is close to the optimal given by Gilbert-Varshamov bound.

2 Main Question and Approach

A natural question to ask is if a *random* subset of walks has the sampling properties required for bias amplification. We know that $|W_k| = O(d^k)$ where d is the degree of the graph G. Would a random subset $V_k \subset W_k$ of size $O(d^{k/2})$ still function as a good enough parity sampler? As it turns out, in order to answer this question, we need to evaluate whether

$$\mathbb{E}_{c \in C} \left[|\text{bias}_{W_k}(c) - \text{bias}_{V_k}(c)| \right] \leq \lambda^{k/2}.$$

In order to simplify the implementation a bit, instead of constructing an ϵ-balanced code C, we just take a (larger) set S such that $\forall s \in S, \text{Bias}(s) \leq \epsilon$. For getting a random subset of walks over expanders, we construct Ramanujan expanders using the Lubotzky-Philips-Sarnack construction [LPS88].

Unfortunately there's quite a few caveats with trying to experimentally test this, as we are limited by computing power, and so we are only able to test for walks of very short lengths ($k \leq 4$). We also have to estimate the desired expectation by taking very small random samples from S. However, these tests still suggest that parity-sampling properties remain intact for random small subsets of random walks on expanders.

3 Program Structure

The program is modularized into two primary files: `lps_graph.py` for the construction of the expanders and `sets_and_walks_helpers.py` for generating the biased sets and walks.

3.1 The LPSGraph Class (`lps_graph.py`)

This class implements the construction Lubotzky-Phillips-Sarnak Ramanujan expander graphs.

- **Initialization:** Supports direct input of primes $p, q \equiv 1 \pmod 4$ or parametric initialization where the class searches for suitable primes based on desired sparsity (β), size (n), and expansion ($l = \lambda$).
- **`LPS_generators()`:** Solves the quaternion equation $a_0^2 + a_1^2 + a_2^2 + a_3^2 = p$ to find $p + 1$ generators in $GL(2, q)$.
- **`build_graph()`:** Constructs the Cayley graph of $GL(2, q)$ using the computed generators. The graph is connected if $(\frac{p}{q}) = 1$. If $(\frac{p}{q}) = -1$, then it takes the component containing the identity vertex.
- **`graph_is_ramanujan()`:** A straightforward tool that uses `scipy.sparse.linalg.eigsh` to compute the eigenvalues and verify if the graph satisfies the Ramanujan bound.

3.2 Sampling Helpers (`sets_and_walks_helpers.py`)

This module provides the machinery to generate test cases (biased sets) and perform the walks.

- **`generate_biased_set()`:** Given n and ϵ, creates a subset of $\mathbb{F}_2^n$ with Hamming weights restricted to a range $[\frac{n(1-\epsilon)}{2}, \frac{n(1+\epsilon)}{2}]$. This is the set S. As for our usage, the sets are going to be quite large, this function returns a generator or *yields* set elements one-by-one instead of returning the whole set at once.
- **`get_all_walks()`:** This function gets all possible walks of some given length k over some given graph G which is passed as a SageMath graph object. The walks are returned as a list of tuples.
- **`bias_amp_func_using_walk()` and `vectorized_bias_amp_func_using_walk()`:** Given a set of walks W and a string s, both the functions calculate and return the (amplified) bias of the resulting string p:

  $$\text{Bias}(p) = \left| \frac{1}{|W|} \sum_{w \in W} \frac{1}{n} \sum_{i \in w} s(i) \right|$$

  Note that for the `vectorized_bias_amp_func_using_walk()`, both the inputs (the string s, and the set of walks W) *must* be NumPy arrays.

4 Installation and Usage Guide

The program is written in **Python 3** and relies heavily on the **SageMath** library for algebraic structures. As such, SageMath (Version 10.7) needs to be installed as per the instructions on their website [[The25]]. We run Python (Version 3.13.3) within the SageMath environment, so none of the additional dependencies (like NumPy etc.) need to be installed separately.

4.1 Basic Usage

To run any script:

```
sage --python lps_graph.py
sage --python sets_and_walks_helpers.py
sage --python experiments.py
```

4.2 Experiments

The `experiments.py` file contains some basic experiments on different types of expander graphs and comparing the resulting amplified bias of the full walk set and smaller random subsets. The main function is `run_experiment()` which takes in a biased set and an expander graph, as well as the size of the random subset of walks to compare with. The results are printed and written to a file in `\results`.

4.3 Example Usage

```python
from lps_graph import LPSGraph
from sets_and_walks_helpers import *

# 1. Construct a (5+1)-regular LPS graph
lps = LPSGraph(p=5, q=13)
G = lps.get_graph()
expansion = lps.expansion

# 2. Relabel vertices from matrices in PGL(2, q) to integers
# This is necessary to be able to index the string using vertices
G.relabel()

# 3. Generate biased set
S = generate_biased_set(eps=expansion, n=G.order(), sample_size=10000)

# 4. Sample a subset of 100 walks of length 10
walk_subset = get_random_subset_of_walks(G, k=10, set_size=100)

# 5. Calculate bias
avg_amplified_bias = 0
for biased_vec in S:
    result = bias_amp_func_using_walk(biased_vec, walk_subset)
    avg_amplified_bias += result

avg_amplified_bias = result/10000
print(f"Amplified Bias: {avg_amplified_bias}")
```

4.4 Troubleshooting and Tips

- The two primary files `lps_graph.py` and `sets_and_walks_helpers.py` contain global variables `MAXIMUM_GRAPH_SIZE`$= 10^6$ and `MAXIMUM_SET_SIZE`$= 10^7$. These exist to abort the program if the requested graphs or sets are too large to prevent memory issues from arising. However, these should be adjusted based on the user's system's specifications.

- Note that for certain parameter values, some scripts can take a long time to run, so using `nohup` in MacOS to run the programs in the background if needed is recommended. It is straightforward to use as:

```
    nohup sage --python SCRIPT_NAME.py > LOG_FILENAME.log > 2>&1

```

References

[LPS88] Alexander Lubotzky, Ralph Phillips, and Peter Sarnak. "Ramanujan graphs". In: *Combinatorica* 8.3 (1988), pp. 261–277.

[RR23] Silas Richelson and Sourya Roy. "Analyzing ta-shma's code via the expander mixing lemma". In: *IEEE Transactions on Information Theory* 70.2 (2023), pp. 1040–1049.

[Ta-17] Amnon Ta-Shma. "Explicit, almost optimal, epsilon-balanced codes". In: *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*. 2017, pp. 238–251.

[The25] The Sage Development Team. *SageMath Installation Guide*. Accessed: 2025-12-29. 2025. URL: `https://doc.sagemath.org/html/en/installation/index.html`.