Efficient Algorithms and New Characterizations for CSP Sparsification

Sanjeev Khanna* Aaron (Louie) Putterman† Madhu Sudan‡

November 5, 2024

Abstract

CSP sparsification, introduced by Kogan and Krauthgamer (ITCS 2015), considers the following question: how much can an instance of a constraint satisfaction problem be sparsified (by retaining a reweighted subset of the constraints) while still roughly capturing the weight of constraints satisfied by *every* assignment. CSP sparsification captures as a special case several well-studied problems including graph cut-sparsification, hypergraph cut-sparsification, hypergraph XOR-sparsification, and corresponds to a general class of hypergraph sparsification problems where an arbitrary 0/1-valued *splitting function* is used to define the notion of cutting a hyperedge (see, for instance, Veldt-Benson-Kleinberg SIAM Review 2022). The main question here is to understand, for a given constraint predicate $P : \Sigma^r \to \{0,1\}$ (where variables are assigned values in Σ), the smallest constant c such that $\widetilde{O}(n^c)$ sized sparsifiers exist for every instance of a constraint satisfaction problem over P. A recent work of Khanna, Putterman and Sudan (SODA 2024) [KPS24] showed *existence* of near-linear size sparsifiers for new classes of CSPs. In this work (1) we significantly extend the class of CSPs for which nearly linear-size sparsifications can be shown to exist while also extending the scope to settings with non-linear-sized sparsifications; (2) we give a polynomial-time algorithm to extract such sparsifications for all the problems we study including the first efficient sparsification algorithms for the problems studied in [KPS24].

Our results captured in item (1) lead to two new classifications: First we get a complete classification of all symmetric Boolean predicates P (i.e., on the Boolean domain $\Sigma = \{0,1\}$) that allow nearly-linear-size sparsifications. This classification reveals an inherent, and previously unsuspected, number-theoretic phenomenon that determines near-linear size sparsifiability. Second, we also completely classify the set of Boolean predicates P that allow non-trivial ($o(n^r)$-size) sparsifications, thus answering an open question from the work of Kogan and Krauthgamer.

The constructive aspect of our result is an arguably unexpected strengthening of [KPS24]. Their work roughly seemed to suggest that sparsifications can be found by solving problems related to finding the minimum distance of linear codes. These problems remain unsolved to this date and our work finds a different path to achieve poly-time sparsification, resolving an open problem from their work. As a consquence we also get the first efficient algorithms to spectrally sparsify Cayley graphs over $\mathbb{F}_2^n$ in time polynomial in the number of generators. Our techniques build on [KPS24] which proves the existence of nearly-linear size sparsifiers for CSPs where the unsatisfying assignments of the underlying predicate P are given by a linear equation over a finite field. Our main contributions are to extend this framework to *higher-degree equations* over *general Abelian groups* (both elements are crucial for our classification results) as well as designing polynomial-time sparsification algorithms for all problems in our framework.

*School of Engineering and Applied Sciences, University of Pennsylvania, Philadelphia, PA. Email: sanjeev@cis.upenn.edu. Supported in part by NSF awards CCF-1934876 and CCF-2008305.

†School of Engineering and Applied Sciences, Harvard University, Cambridge, Massachusetts, USA. Supported in part by the Simons Investigator Awards of Madhu Sudan and Salil Vadhan, NSF Award CCF 2152413 and a Hudson River Trading PhD Research Scholarship. Email: aputterman@g.harvard.edu.

‡School of Engineering and Applied Sciences, Harvard University, Cambridge, Massachusetts, USA. Supported in part by a Simons Investigator Award and NSF Award CCF 2152413. Email: madhu@cs.harvard.edu.

Contents

1 Introduction

In this work we study the problem of "CSP sparsification" and give the first efficient algorithms and characterizations for sparsifying many classes of CSPs. Our contributions yield immediate, novel results in efficient Cayley graph sparsification [KPS24], efficient hedge-graph sparsification [GKP17] and sparsifying hypergraphs with cardinality-based splitting functions [VBK22]. We start by introducing CSP sparsification more precisely before moving on to the motivation, our results, and the applications of these results.

1.1 CSP sparsification

CSP sparsification was introduced by Kogan and Krauthgamer [KK15] as a broad extension of the notion of cut sparsification in graphs. A cut sparsifier of a graph is a weighted subgraph of a given graph on the same set of vertices that roughly preserves the size of every cut. The seminal works of Karger [Kar93] and Benczúr and Karger [BK96] showed that every undirected graph admits a cut sparsifier of size nearly linear in the number of vertices, thus potentially compressing graph representations by a nearly linear factor while preserving some significant information. CSP sparsification aims to extend this study to broader classes of structures than just graphs, and aims to preserve a broader class of queries than cuts. (We remark here that this is just one of several directions of generalizations. Other notable directions such as spectral sparsification ([BSS09, ST11, Lee23, JLS23, JLLS23]) and general submodular hypergraph sparsification ([KK23, JLLS23]) are not covered in this work.)

In this work, a CSP problem, $\mathrm{CSP}(P)$, is specified by a predicate $P : \Sigma^r \to \{0, 1\}$. An instance Φ of $\mathrm{CSP}(P)$ on n variables is given by m weighted constraints $(w_1, C_1), \ldots, (w_m, C_m)$ where each constraint C_j applies the constraint P to a specified sequence $(i_1(j), \ldots, i_r(j)) \in [n]^r$ of r out of the n variables. An assignment to the variables is given by $a \in \Sigma^n$ and a satisfies constraint C_j if $P(a_{i_1(j)}, \ldots, a_{i_r(j)}) = 1$. In CSP sparsification, an instance Φ is viewed as the specification of a structure that we wish to compress with the goal that the compressed representation approximately preserves the total weight of satisfied constraints for *every assignment* $a \in \Sigma^n$. A *sparsification* of Φ is obtained by retaining a subset of the constraints, possibly assigning them new weights, and the size of the sparsification is the number of constraints retained.

1.2 Motivation

Beyond its immediate interest as a natural combinatorial question, CSP sparsification is motivated by the need to simplify large complex networks represented by hypergraphs. While graphs model pairwise interactions, hypergraphs are needed to model interactions between larger subsets of entities. The work of Veldt, Benson and Kleinberg [VBK22] describes a broad collection of settings where the central interactions require this greater flexibility to model. Furthermore, while graphs admit only one interesting notion of cutting an edge, with hyperedges one can imagine more complex notions. Typically, this is referred to as *generalized hypergraph sparsification* (see for instance, [VBK22]), where each hyperedge $e \subseteq V$ is equipped with a splitting function $g_e : 2^e \to \mathbb{R}^+$. Given a cut $S \subseteq V$, the contribution of a hyperedge e to the cut is exactly $g_e(S \cap e)$, and the total cut size is $\sum_{e \in E} g_e(S \cap e)$. For instance, by setting the splitting function to be 0 on inputs e and $\emptyset$, and 1 otherwise, this models the standard notion of cuts in hypergraphs. By allowing more general notions of hyperedge cuts, say, that is, g_e is an arbitrary function of $S \cap e$, this model captures a variety of applications, ranging from scientific computing on sparse matrices [BDKS16], to clustering and machine learning [YNY+19, ZHS06], to modelling transistors and other circuitry elements in circuit

design [AK95, Law73], and even to human behavior and biological interactions (see [VBK22]). In each of these applications, the ability to sparsify the hypergraph while still preserving cut-sizes is a key building block, as these dramatically decrease the memory footprint (and complexity) of the networks being optimized.

Within this general framework, boolean CSP sparsification captures the important and expressive case where the splitting functions are $\{0,1\}$-valued, (i.e., each $g_e : 2^e \to \{0,1\}$). This richness of expressibility in CSPs leads to the question (raised in [KK15]) of for which CSPs (i.e., for which predicates P) is CSP(P) sparsifiable to nearly linear size, or more generally, what is the best (possibly non-integral) exponent ℓ such that CSP(P) is sparsifiable to instances of size roughly n^ℓ on instances with n variables.[1] In Kogan and Krauthgamer's work [KK15], they showed that for any r-CNF constraint, CSPs admit sparsifiers of size $\widetilde{O}(rn/\epsilon^2)$. Filtser and Krauthgamer [FK17] later gave a complete characterization of Boolean CSPs on two variables ($|\Sigma| = r = 2$) establishing a dichotomy result that shows that each predicate P either allows for near-linear size sparsification or requires quadratic size sparsifiers (which is trivially achievable since that is the number of distinct constraints). Their result was extended to binary (r=2) CSPs over all finite alphabets Σ by Butti and Živný [BZ20], thus giving the first classification for an infinite subclass of CSPs. A recent work of the authors [KPS24] extended the classification of [KK15] to the case of ternary Boolean CSPs ($r = 3$, $\Sigma = \{0,1\}$) but only identified the predicates that allow nearly linear sparsification while showing the rest required at least quadratic size. Note that the trivial sparsification in this case has size $O(n^3)$. The central notion developed in their work is sparsifiers of linear systems of equations over finite fields (aka "code sparsifiers"). Their results are non-constructive in that they only show existence of nearly-linear size sparsifiers, and did not give a polynomial time algorithm to find them. In fact, their work highlighted some natural obstacles to achieving efficient sparsification.

In this work we extend the work of [KPS24] in two directions: (1) We give a polynomial time algorithm for constructing sparsifiers of linear systems of equations, thereby making the results of [KPS24] constructive. (2) We extend the (constructive) sparsification from linear equations over finite fields to linear and higher degree equations over abelian groups. These generalizations allow us to get new (efficiently constructive) dichotomies for sparsifiability of some (infinite) subclasses of CSPs.

1.3 Our Results

We start by describing our new results, and defer the new technical ingredients to the following section.

Efficient Code Sparsification Our main technique extends a technique called "code sparsification" introduced in [KPS24]. The primary structure of interest in code sparsification is a linear code $C \subset \mathbb{F}_q^m$ over some finite field $\mathbb{F}_q$. A *sparsifier* for C is a restriction $C|_S$ (also called "puncturing") of the code C to a subset $S \subseteq [m]$ of the coordinates along with weights on the coordinates so that the weighted Hamming weight of each codeword in $C|_S$ is approximately the same as the weight of the corresponding codeword in C. [KPS24] show that for every code of dimension n there *exists* a sparsifier of size nearly-linear in n. Code sparsification turns out to be a powerful tool for CSP

[1] In the literature on CSP decision or maximization problems, a number of further subtleties arise. Ideally one would like sparsifications for *families* or predicates as opposed to a single predicate. There is a difference between whether constraints must be application of the predicate to distinct variables or we allow repetition of variables in a constraint. In the Boolean setting there is a difference between predicates being applied to variables versus literals. None of these issues is relevant in the case of CSP *sparsification* since the task is rich enough to allow easy reductions among all these problems.

sparsification and indeed is used in [KPS24] to get nearly linear-size sparsifiers for many classes of CSPs. However, the result of [KPS24] is only *existential* and indeed seems to require solving NP-hard problems to get algorithmic code sparsification. We remedy this problem by giving an *efficient* algorithmic implementation of code-sparsification:

Theorem 1.1. *For any code $C \subset \mathbb{F}_q^m$ of dimension n and a parameter $\epsilon \in (0,1)$, there is a polynomial time (in $n, m, \log(q), \epsilon^{-1}$) randomized algorithm for computing (with high probability) a $(1 \pm \epsilon)$ code-sparsifier $C|_S$ of C, with $|S| = \widetilde{O}(n/\epsilon^2)$.*

This settles a key open question of [KPS24] regarding tractability of computing code-sparsifiers and leads to efficient algorithms for the sparsifiers constructed in their work. For instance it leads to the first efficient algorithm to sparsify a Cayley graph over $\mathbb{F}_2^n$ into another Cayley graph with roughly the same spectrum, in the sense of [ST11], where efficiency is with respect to the number of generators (and not the size of the graph)!

Corollary 1.2. *For any Cayley graph G over $\mathbb{F}_2^n$ with associated generating set $Q \subseteq \mathbb{F}_2^n$ with $|Q| = m$ and a parameter $\epsilon \in (0,1)$ of our choosing, there is a polynomial time (in n, m, ϵ^{-1}) randomized algorithm for computing a $(1 \pm \epsilon)$* Cayley-graph spectral-sparsifier *$\tilde{G} = \mathrm{Cay}(\mathbb{F}_2^n, \hat{Q})$, where $|\hat{Q}| = \widetilde{O}(n/\epsilon^2)$ is a re-weighted subset of the generators.*

We elaborate on this result in Section 11.1, and also show a novel generalization to Cayley graphs over $\mathbb{Z}_q^n$ in Section 11.2.

Another corollary of this result is an efficient algorithm for "hedge-graph sparsification", a topic of extensive study in the literature [GKP17, JLM+23, FGK+24], where lack of sub-modularity had thus far been a barrier to efficient sparsfication. We resolve this in Section 11.3.

CSP Sparsification Next, we discuss our contributions to the CSP sparsification regime. We start by formalizing the notion of sparsifying CSPs. An instance of CSP(P) on n variables is given by a collection of m weighted constraints $(w_1, C_1), \ldots (w_m, C_m)$ where C_j is given by a sequence $(i_1(j), \ldots, i_r(j)) \in [n]^r$. For $a \in \Sigma^n$, let $C_j(a) = P(a_{i_1(j),\ldots,i_r(j)})$ denote whether a satisfies the jth constraint. We let $\Phi(a)$ denote the total weight of constraints satisfied by a, i.e.,

$$\Phi(a) = \sum_{j=1}^{m} w_j P(a_{i_1(j)}, \ldots, a_{i_r(j)}).$$

Definition 1.1 (CSP Sparsification)**.** *For $P : \Sigma^r \to \{0,1\}$, $\epsilon > 0$ and $s \in \mathbb{Z}^+$ we say that an instance $\hat{\Phi} \in CSP(P)$ is an* (ϵ, s)-sparsifier *for $\Phi \in CSP(P)$ if there are at most s constraints $\left\{\hat{C}_i\right\}_{i \in [s]}$ in $\hat{\Phi}$, each of which satisfies $\hat{C}_i \in \{C_1, \ldots C_m\}$ and for every $a \in \Sigma^n$ we have $(1 - \epsilon)\Phi(a) \leq \hat{\Phi}(a) \leq (1 + \epsilon)\Phi(a)$. (Note that the constraints of $\hat{\Phi}$ may have very different weights than corresponding constraints in Φ.)*

For $P : \Sigma^r \to \{0,1\}$, $\epsilon > 0$ and $s : \mathbb{Z}^+ \to \mathbb{Z}^+$, we say that CSP($P$) is $(\epsilon, s(\cdot))$-sparsifiable *if for every $n \in \mathbb{Z}^+$, every instance $\Phi \in CSP(P)$ with n variables has an $(\epsilon, s(\epsilon, n))$-sparsifier. Furthermore we say that CSP(P) is* $(\epsilon, s(\cdot))$-efficiently sparsifiable *if there is a probabilistic polynomial time algorithm to compute such a sparsification.*

Essentially, the requirement of a sparsifier is that the constraints in the sparsifier $\hat{\Phi}$ are a (suitably re-weighted) subset of the original constraints in Φ such that the value attained on each assignment is approximately preserved. The only freedom we get is in assigning new weights to

these constraints. For brevity, we will say that CSP(P) is sparsifiable to *near-linear size* if CSP(P) is $(\epsilon, s(\epsilon, n))$-sparsifiable for some function $s(\epsilon, n) = \widetilde{O}_\epsilon(n)$.

Our first theorem completely characterizes which symmetric Boolean predicates admit nearly linear-size sparsification. Recall that a predicate $P : \Sigma^r \to \{0,1\}$ is Boolean if $\Sigma = \{0,1\}$. We say P is *symmetric* if $P(a_1, \ldots, a_r) = P(a_{\pi(1)}, \ldots, a_{\pi(r)})$ for every $a_1, \ldots, a_r \in \Sigma$ and every permutation $\pi : [r] \to [r]$. Hence a Boolean predicate P is symmetric if and only if there exists $P_0 : \{0, \ldots, r\} \to \{0,1\}$ such that $P(a_1, \ldots, a_r) = P_0(a_1 + \cdots + a_r)$ for every $a_1, \ldots, a_r \in \{0,1\}$. Define a symmetric Boolean predicate P to be *periodic* if the zeroes of P_0 form an arithmetic progression. That is, P given by $P(a_1, \ldots, a_r) = P_0(a_1 + \cdots + a_r)$ is periodic if and only if there exists c, d such that $P_0(\alpha) = 0$ for some $\alpha \in \{0, \ldots, r\}$ if and only if $\alpha \in \{c + i \cdot d \mid i \in \mathbb{Z}\}$, where the addition and multiplication in $c + i \cdot d$ are over the integers. We say P is *aperiodic* otherwise.

Theorem 1.3. *Let $P : \{0,1\}^r \to \{0,1\}$ be a symmetric predicate. Then if P is periodic, CSP(P) is $(\epsilon, \widetilde{O}(n/\epsilon^2))$-efficiently sparsifiable for every $\epsilon \in (0,1)$. On the other hand, if P is not periodic then for every $0 < \epsilon < 1$, CSP(P) is not $(\epsilon, o(n^2))$-sparsifiable.*

Remark 1.4. *Note that symmetric CSPs exactly capture the $\{0,1\}$-valued case of so-called "cardinality-based splitting functions" where the splitting function $g_e(S \cap e)$ depends only on the carindality of $|S \cap e|$. Such splitting functions appear broadly in the clustering literature [LM17, LM18, VBK20, VBK21, LVS+21, ZLS22] and in summarization of complex data sets [GK10, LB11, TIWB14]. This result leads to an exact (and efficient) characterization of when such splitting functions allow for sparsifiers of near-linear size.*

As noted earlier, this generalizes all known results on nearly-linear CSP sparsification while adding efficiency to previous results. Note that the class of symmetric predicates is quite general, capturing cut functions in graphs and hypergraphs, as well as parity functions. In particular, this theorem provides a concise reason explaining why hypergraphs, graphs, and parity functions all admit linear-size sparsifiers, whereas a priori, it may have seemed quite arbitrary for these natural choices to all yield linear-size sparsifiers. Prior to this work there was no reason to believe that the sparsifiability of symmetric predicates would be inherently tied to the periodicity of the pattern of unsatisfying assignments. Our results thus reveal this phenomenon for the first time and establish a formal connection between periodicity and sparsifiability.

Another interesting consequence of this theorem is that it implies that there are "discrete jumps" in the sparsifiability of predicates. That is to say, either symmetric predicates are sparsifiable to near-linear size, or one can do no better than quadratic size; there is no intermediate regime with for instance sparsifiers of size $O(n^{1.5})$. This has been observed in the field of sparsification in many regimes, for instance near-linear size sparsifiers exist for graphs [BK96], undirected hypergraphs [CKN20], and codes [KPS24], while requiring quadratic size sparsifiers for directed graphs. No problems were known for which intermediate complexity helped. Our results formally validate this phenomenon for the class of symmetric CSPs.

Our next theorem classifies Boolean predicates that have a non-trivial sparsification. Note that every instance of CSP(P) on n variables for $P : \{0,1\}^r \to \{0,1\}$ has $O(n^r)$ distinct constraints. Thus trivially CSP(P) is $(0, O(n^r))$-sparsifiable. We define a sparsification to be *non-trivial* if it achieves $s(n) = o(n^r)$. Our next theorem shows that CSP(P) is non-trivially sparsifiable if and only if P does not have only one satisfying assignment.

Theorem 1.5. *Let $P : \{0,1\}^r \to \{0,1\}$. If $|P^{-1}(1)| \neq 1$ then CSP(P) is $(\epsilon, \widetilde{O}_r(n^{r-1}/\epsilon^2))$ efficiently-sparsifiable for every $\epsilon > 0$. Otherwise, for every $0 < \epsilon < 1$, CSP(P) is not $(\epsilon, o(n^r))$ sparsifiable.*

This settles an open question posed by [KK15] who asked for such a classification. Surprisingly, this theorem may be viewed as a "mostly positive" statement by showing that almost every predicate, specifically every one with more than one satisfying assignment, is non-trivially sparsifiable. We also stress that in our result the existence of sparsifiability is accompanied by an efficient (randomized) algorithm to construct such sparsifiers.

Finally we extend the characterization of nearly linear-size sparsifiable ternary Boolean CSPs from [KPS24] to get a complete classification (in particular, separating CSPs that admit a near-quadratic sparsifier from those that require cubic size). To describe the theorem we need the notion of the projection of a predicate. Given predicates $P : \{0,1\}^r \to \{0,1\}$ and $Q : \{0,1\}^c \to \{0,1\}$ we say that P has a *projection* to Q if there exists a function $\rho : \{X_1, \ldots, X_r\} \to \{0,1\} \cup \{Y_1, \neg Y_1, \ldots, Y_c, \neg Y_c\}$ such that $Q(Y_1, \ldots, Y_c) = P(\rho(X_1), \ldots, \rho(X_r))$. Let $\mathrm{AND}_c(Y_1, \ldots, Y_c) = Y_1 \wedge \cdots \wedge Y_c$.

Theorem 1.6. *For a predicate $P : \{0,1\}^3 \to \{0,1\}$ let c be the largest integer such that P has a projection to AND_c. Then, CSP(P) is $(\epsilon, \widetilde{\Theta}(n^c))$ efficiently-sparsifiable for every $\epsilon \in (0,1)$, and moreover, for every $\epsilon \in (0,1)$, CSP(P) is not $(\epsilon, o(n^c))$ sparsifiable.*

As mentioned before, this theorem continues adding evidence to the conjecture that sparsifier sizes always come in integral gaps. In particular, it shows that for predicates of arity 3, optimal sparsifier sizes are either $\widetilde{\Theta}(n), \widetilde{\Theta}(n^2)$ or $\Theta(n^3)$.

1.4 Technical Theorems

As mentioned before, our main technique extends a technique called *code sparsification* introduced in [KPS24]. We present two "orthogonal" improvements to this technique of code sparsification. The first is to make code sparsifiers *algorithmic*, and the second is to *extend* these notions *beyond codes*. Getting an algorithm version of the existence result of [KPS24] is non-trivial — their work roughly decomposed the coordinates of a linear codes into "dense" and "sparse" coordinates, where the former support a subcode of relatively high dimension (relative to their size). Applying this decomposition recursively until all coordinates are regular within their partition and then sampling an appropriate number of coordinates in each partition yields their sparsification. The partition of coordinates into "dense" and "sparse" ones is unfortunately not easy to compute (at least to our knowledge). For instance, the support of a minimum weight codeword can be dense in some codes and finding such codewords, or even approximating the size of their support is NP-hard in the worst case, and means there is no clear way to implement the sparsifiers of [KPS24] in less than exponential time. One of the main contributions of this work is to find an alternate path to this decomposition. We elaborate more on this in Section 2.1, but the approach is to efficiently find a small superset of the dense coordinates of a code. We build on insights from Benczúr and Karger [BK96] to develop this alternate path which is not as obvious in the coding setting as in the graph-theoretic one. Fortunately, not only does this approach extend to codes, but also to all the extensions of codes that we need to sparsify in this paper. In the rest of this section we focus only on the existential aspects of the results in the dicussion, while the theorems assert the algorithmic parts.

Turning to the second direction of improvements in this paper, as noted earlier, code sparsifiers already help with CSP sparsifications, but for our purpose of classifying symmetric Boolean CSPs they only go part of the distance. Specifically, towards our classification, it is relatively straightforward to show that aperiodic symmetric predicates P do not allow for nearly linear sparsifiers. Applying the code sparsifiers of [KPS24] we can show that periodic predicates P with the *period being a prime* are sparsifiable to nearly linear size. However, this leaves a big gap where P is

periodic with a composite period – we neither get near-linear sparsifiers in this case nor get to rule them out. To capture all periodic CSPs we extend the sparsifiers of [KPS24] from codes over fields to roughly what might be called "codes over groups". We don't formalize this concept but instead describe the specific theorem we prove based on this extension.

We say that a predicate $P : \{0,1\}^r \to \{0,1\}$ is an *affine predicate* over a group $(G, \odot)$ if there exist $a_1, \ldots, a_r, b \in G$ such that $P(x_1, \ldots, x_r) = 1$ if and only if $a_i x_1 \odot a_2 x_2 \odot \cdots \odot a_r x_r \neq b$. (Note that $a_i x_i = a_i$ if $x_i = 1$ and 0 otherwise, where 0 is the identity element of the group G.) We say that P is an *affine Abelian predicate* if there exists a finite Abelian group A such that P is an affine predicate over A.

Theorem 1.7. *If $P : \{0,1\}^r \to \{0,1\}$ is an affine Abelian predicate over an Abelian group A, then CSP(P) is $(\epsilon, \widetilde{O}(n \cdot \min(r^4, \log^2(|A|))/\epsilon^2)$-efficiently-sparsifiable for every $\epsilon > 0$.*[2]

In our language, the main technical result of [KPS24] can be stated as being the special case corresponding to A being $\mathbb{Z}_p$ the group of additions modulo a prime p. We note that the class of affine Abelian predicates is much richer and has nice closure properties. For instance if P_1 and P_2 are affine Abelian predicates then so is $P_1 \vee P_2$, thus establishing that the predicates covered by Theorem 1.7 have a nice closure property. (Even an existential sparsification result would not follow from the [KPS24] result if P_1 and P_2 were predicates over different prime groups $\mathbb{Z}_p$ and $\mathbb{Z}_q$, whereas with Abelian predicates we can now simply operate over $\mathbb{Z}_p \times \mathbb{Z}_q$.) In particular, Theorem 1.7 immediately yields the characterization of symmetric Boolean CSPs that allow a near-linear sparsification (Theorem 1.3), as well as their efficient construction. We also note here that Theorem 1.7 does not follow immediately from the techniques of [KPS24]. We elaborate more on this in Section 2, but briefly — the entire analysis of [KPS24] is coding-theoretic with notions like dimension and distance of codes playing a fundamental role, and linear algebra over $\mathbb{F}_q$ being the main engine. In our case we have to switch to a more lattice-theoretic approach and notions like dimension have to be replaced with more involved counting arguments, while some of the simpler linear algebra is replaced with gcd computations. We complement this with an efficient algorithm for computing a type of decomposition for the "code", which yields the constructive aspect. While the final proof is not much more complicated each step requires extracting abstractions that were not obvious in their work.

We also show below that the requirement that the underlying group is Abelian can not be relaxed.

Theorem 1.8. *For every non-Abelian group G, there exists a predicate $P : \{0,1\}^4 \to \{0,1\}$ and an $\epsilon_0 > 0$ such that P is an affine predicate over G and CSP(P) is not $(\epsilon_0, o(n^2))$-sparsifiable.*

Turning to our second theorem (Theorem 1.5), observe that the main result of [KPS24] and Theorem 1.7 do not seem to yield non-trivial but superquadratic size sparsifiers. For many natural classes of problems, the best sparsifiers are superquadratic in size and so extending the techniques to address such questions is of importance (in particular to prove a statement like Theorem 1.5). To this end we give a different extension of Theorem 1.7 to higher degree polynomials.

For an Abelian group A (possibly infinite) We say that $P : \{0,1\}^r \to \{0,1\}$ is a degree ℓ polynomial over A if there exists a degree ℓ polynomial $Q \in A[X_1, \ldots, X_r]$ such that for every $x \in \{0,1\}^r \subseteq \mathbb{Z}_q^r$ we have $P(x) = 1$ if and only if $Q(x) \neq 0$.[3]

[2] We note that A does not have to be a finite group in this theorem, though in all the applications to CSPs we only use finite Abelian groups.

[3] A polynomial in $A[X_1, \ldots, X_r]$ is just a formal sum of monomials in $X_1 \ldots, X_r$ with coefficients from A. This polynomial naturally gives a function from $\mathbb{Z}^r \to G$ by evaluating the monomials over integers, and then adding/subtracting the appropriate number of copies of the coefficient.

Theorem 1.9. *If $P : \{0,1\}^r \to \{0,1\}$ is a degree ℓ polynomial over an Abelian group A then CSP(P) is $(\epsilon, \widetilde{O}(n^\ell \min(r^{4\ell}, \log^2(|A|))/\epsilon^2)$-efficiently-sparsifiable for every $\epsilon > 0$.*

While the extension to higher degree polynomials is relatively straightforward, it vastly increases the expressive power of these algebraic CSPs. The positive result in Theorem 1.5 turns out to be a natural extension obtained by proving that every predicate P with more than one satisfying assignment can be written as the OR of polynomials of degree at most $r-1$ over $\mathbb{F}_2$.

1.5 Conclusions

CSP sparsification [KK15] is a powerful unifying abstraction that captures many central and often individually studied, sparsification questions. By studying them jointly we thus get a more global picture of the world of sparsifiability. Our work shows that it is possible to classify large subclasses completely and prove discrete jumps in the level of sparsifiability within these broad subclasses. The classifications also show an algorithmic jump, namely that when sparsifiers exist they can be found algorithmically while the non-existence results are information-theoretic. In the process, these sparsification results yield applications to sparsifying generalized hypergraphs (particularly, with cardinality-based splitting functions), and towards deriving efficient constructions of fundamental combinatorial objects such as Cayley-graph sparsifiers.

Our work builds on the code sparsification technique of [KPS24], but extends it conceptually as well as algorithmically. In particular, we give a broad framework for upper-bounding the size of sparsifiers by interpreting predicates as polynomials over arbitrary groups. In fact, every known CSP sparsification result follows from our framework (specifically from Theorem 1.9). Indeed already the code sparsification framework of [KPS24] (which we generalize) captured the cut sparsifiers of [BK96], hypergraph cut sparsifiers, and r-CNF sparsifiers [KK15, CKN20], the sparsifiers for Boolean predicates of arity 2 [FK17] and even those over arbitrary alphabets [BZ20]. (The final claim was not pointed out in [KPS24] so we include a proof of the sparsification result in [BZ20] using our framework in Appendix B.) Our framework further generalizes theirs and so captures all the above mentioned works and the code sparsifiers of [KPS24] while giving new results for symmetric Boolean predicates, classifying general functions with non-trivial sparsifiability, as well as giving efficient algorithms for finding these sparsifiers.

For future directions we note that there is a noticeable gap between the upper bound and lower bound techniques, even for near-linear size sparsifiability of asymmetric predicates and for characterizing the size needs of symmetric CSPs, when the size lower bound is quadratic. The only known lower bounds on sparsifiability come from the notion of a projection to an AND predicate. A hope at this stage may be that every function that does not have a projection to AND_c can be expressed as a degree $c-1$ polynomial. Note that we proved this to be true when $c = 2$ and P is a symmetric predicate. Unfortunately this seems to fail beyond this setting. In Appendix D we describe an asymmetric predicate that is not expressible as a degree 1 polynomial but also does not have a projection to AND_2 and in Appendix E we show that there is a symmetric predicate which does not have a projection to AND_3 but also *seems* to not be expressible as a degree 2 polynomial. Thus finding new lower bounds on sparsifiability (other than projection to AND_c) or finding new reasons for sparsifiability (not captured by Theorem 1.9) seem to be necessary for future progress.

Addendum: In a recent breakthrough subsequent to our work, Brakensiek and Guruswami [BG24] have given a complete characterization of the sparsifiability of every CSP (over all alphabets) in terms of the "non-redundancy" of the predicate, a quantity that has been the subject of some prior work in CSP dichotomy classifications. They do not give a general method to determine the

non-redundancy of a predicate, or to analyze its growth as a function of n, the number of variables, however they do give bounds in specific instances. In particular their work shows the existence of predicates for which the best sparsifications are of size n^α for some $1.5 \leq \alpha \leq 1.6$ (and so not an integer). Their work is not algorithmic and sparsifications are only guaranteed to exist.

Organization: In Section 2, we provide an in-depth overview of the techniques used to prove our results and show how these immediately imply Theorem 1.6. Section 3 summarizes some useful definitions and previously known results that we utilize in our work. Section 4 is dedicated to proving a decomposition theorem and counting bound for codewords of a specific weight, and we use this key result in Section 5 where we prove a version of Theorem 1.7 only for groups of the form $\mathbb{Z}_q$ and a dependence on $\log^2(q)$. In Section 6, we generalize this algorithm to all Abelian groups, proving Theorem 1.7 in its entirety. In Section 7, we show a complement to this theorem, namely, there are affine predicates over a non-Abelian group that are *not* sparsifiable which yields Theorem 1.8. In Section 8, we show how to extend our sparsification framework to Boolean symmetric CSPs, proving Theorem 1.3. Finally, in Section 9, we prove Theorem 1.9, and then use it as a key building block to prove Theorem 1.5. In Section 10, we then use Theorem 1.5 to prove Theorem 1.6.

2 A Detailed Overview of Techniques

We start by reviewing, in Section 2.1 the work of Khanna, Putterman and Sudan [KPS24] to highlight the challenges of getting efficient algorithms, and working over Abelian groups as opposed to finite fields. We then describe our efficient algorithm implementation of their strategy in Section 2.2. Then, in Section 2.3 we describe our approach to extending their work to general (finite) cyclic groups. In Section 2.4 we explain we extend this work to all finite Abelian groups and then to all Abelian (even infinite) groups. While the CSP applications don't need it, we also remove the logarithmic dependence on m, the number of constraints, since m can be exponentially large in general linear systems we look at. Section 2.4 also includes an overview of the steps needed to achieve this. We then show to get our classification of all Boolean symmetric CSPs in Section 2.5. Finally, in Section 2.6 show how to extend the entire approach to sparsification of higher degree polynomial constraints and how to use this to get a classification all non-trivially sparsifiable Boolean predicates.

2.1 Previous work of [KPS24]

Recall that the work of [KPS24] showed that for any linear subspace (i.e. a code) $\mathcal{C} \subseteq \mathbb{F}_q^m$ of dimension n, there exists a weighted subset $S \subseteq [m], w : S \to \mathbb{R}^+$ of size $\widetilde{O}(n \log(q)/\epsilon^2)$ such that for any codeword $c \in \mathcal{C}$,

$$(1-\epsilon)\mathrm{wt}(v) \leq \mathrm{wt}_w(v|_S) \leq (1+\epsilon)\mathrm{wt}(v).$$

In this context, $\mathrm{wt}_w(x)$ is meant to be the "weighted hamming weight", i.e., $\mathrm{wt}_w(x) = \sum_{i=1}^m w(i)\mathbb{1}_{x_i \neq 0}$.

Their sparsification relies on a counting bound they prove that asserts that for every code $\mathcal{C} \subseteq \mathbb{F}_q^m$ of dimension n and every integer d either (1) $\mathcal{C}$ has the property that for every positive integer α it has at most $n^\alpha \cdot q^\alpha$ codewords of weight at most αd, or (2) there is a subset of coordinates $S \subseteq [m]$ and positive integer t such that $|S| \leq d \cdot t$ and $\mathcal{C}$ has more than t independent codewords supported on S. If condition (1) holds for a judicious choice of d, then uniformly sampling coordinates in $[m]$ at rate roughly $1/d$ sparsifies the code $\mathcal{C}$. But if condition 1 does not hold, then for S being the maximal set with property (2), they show that sampling coordinates i of $[m] \setminus S$ at rate $1/d$ and

weighting them with weight $w(i) = d$ and retaining all coordinates of S with weight 1 leads to a good sparsifier. (The full sparsification to nearly-linear size in n applies this idea recursively.)

The key thus is this counting bound, which is in turn proved by a "contraction" argument. This argument defines a randomized procedure that outputs a random codeword. The claim is that this if no set S satisfies condition (2) then any fixed codeword c of weight at most αd is output with probability at least $n^{-\alpha} \cdot q^{-\alpha}$. This immediately implies the counting lemma above.

The randomized procedure maintains a code $\mathcal{C}'$ that is obtained from $\mathcal{C}$ after a sequence of contractions as follows: (a) Sample a random coordinate j in the support $\mathrm{supp}(\mathcal{C}')$ of $\mathcal{C}'$ (where $\mathrm{supp}(\mathcal{C}')$ is the set of coordinates where there exists a codeword of $\mathcal{C}'$ that is non-zero). (b) Contract $\mathcal{C}'$ on coordinate j, i.e., remove all codewords with j in their support from $\mathcal{C}'$. This procedure is repeated until $\dim(\mathcal{C}') \leq \alpha$ at which point we output a random codeword of $\mathcal{C}'$. To see that this satisfies the claim, one assumes that no set S satisfies condition (2) and fixes a random codeword $c \in \mathcal{C}$ of weight at most αd and show that each step of contraction (as in (a) above) preserves membership of $c \in \mathcal{C}'$ with probability at least $\alpha d/|\mathrm{supp}(\mathcal{C}')|$ and the product of this quantity over the iterations of contraction telescopes to $n^{-\alpha}$. When the algorithm terminates $\mathcal{C}'$ has dimension at most α and so has at most q^{α} codewords. Thus if c is still a codeword of $\mathcal{C}'$ then it will be output with probability $q^{-\alpha}$.

This concludes our summary of the work of [KPS24]. Before turning to our work we highlight the challenges towards making their analysis algorithmic, and to extending it to settings beyond linear codes over $\mathbb{F}_q$.

Algorithmic challenge: The description above including that of contraction is needlessly wasteful — in order to maintain the code $\mathcal{C}'$ one does not need to maintain the full set of codewords of $\mathcal{C}'$. It suffices to maintain a basis and the contraction can be easily explained as a step of Gaussian elimination on this basis. (Indeed this is already done in [KPS24].) The key challenge to making this argument algorithmic is that of finding a set S satisfying condition (2) when it exists. For instance if $\mathcal{C}$ restricted to S has dimension $t = 1$, finding S corresponds to finding a codeword of minimum weight in $\mathcal{C}$, a well-known NP-hard task [Var97] even to approximate [DMS03]. This is the key algorithmic barrier that we need to overcome in this work.

Beyond Fields: Turning to extensions beyond $\mathbb{F}_q$, we first note that such an extension is necessary for us to get a classification. For instance the predicate $P(x_1, \ldots, x_6) = 1 \Leftrightarrow \sum_i x_i \notin \{0, 6\}$ needs to works with linear structures ("modules") over $\mathbb{Z}_6$. The entire analysis above involving notions like dimension and basis is linear algebraic, and Gaussian elimination only works in this setting. It is well-known that elements of linear algebra sometimes completely break down beyond the setting of fields, leading to obstacles such as inability to prove strong lower bounds for ACC circuits [BBR94], while opening up paths for surprising designs [Gro97] and codes [Efr12]. A priori it is not clear if there is a way to extend the analysis to, say, additive sets over $\mathcal{C} \subseteq \mathbb{Z}_6^m$.

Beyond linear-size sparsifications: Finally while the methods of [KPS24] are great when it comes to proving the existence of linear-sized sparsifications, they fail to distinguish between different settings where the best sparsification is not trivial (of size n^r for an r-ary predicate P) but super-linear. They either offer sparsifications of (nearly) linear size or revert to trivial sparsifications. A challenge before our work is to develop techniques to identify settings with sparsification complexity in between the linear and the trivial regime.

2.2 Towards an Efficient Algorithm

As explained above finding a small set S of coordinates that supports many codewords is not known to be algorithmically feasible (and quite possibly is NP-hard). The key starting point of our work is the observation that we do not have to *exactly* recover the set S of rows to remove. Instead, as long as we can recover some *superset* T of these rows S which is not too large, the rest of the sparsification algorithm will work unhindered. A similar insight is also seen in [BK96] when creating their efficient sparsification routine for graphs. In the graph setting, [BK96] define a notion of strength of an edge and show that simply storing $2d\log(n)$ spanning forests suffices for capturing all edges of strength $\leq d$. After this, one can then show that sampling the rest of the edges in a graph at rate roughly $\log(n)/d$ will preserve all cuts in the graph to a factor of $(1 \pm \epsilon)$. At a high-level, our approach is in a similar spirit, though we need a different combinatorial object.

Instead of considering spanning forests, we introduce the notion of "maximum spanning subsets". For a matrix G generating the code $\mathcal{C}$, our goal is to select a subset of rows T_1 such that the number of distinct codewords in $\text{Span}(G|_{T_1})$ is equal to the number of distinct codewords in $\text{Span}(G)$. We show that if one iteratively calculates and stores $2d\log n$ disjoint maximum spanning subsets (yielding a set T) we efficiently compute a set T that is not too large and satisfies $S \subseteq T$. Note that our notion of largeness is much weaker. S was promised to have size at most $d \cdot t$ (where t is the dimension of S) whereas our T has size at most $2dn\log n$. But this turns out to be good enough for efficient sparsification!

The key analysis step is showing that such a set T contains S: This is done by looking at how much of S has been collected after each block of removal of cd disjoint maximum spanning subsets. Suppose the initial dimension of S is t. We note that if the dimension of S is still at least t/c after this block of removals, then each iteration within the block must have recovered at least t/c new coordinates from S, and this is a contradiction to the claim that S only had dimension t to start with. We conclude the dimension of S drops by a constant factor in each block and so $\log n$ blocks of removal of maximum disjoint spanning subsets completely covers all of S. This allows us to make the counting lemma of [KPS24] algorithmic and thus get efficient algorithms for all their settings.

2.3 Sparsifying additive sets over $\mathbb{Z}_q$ for general q

We now discuss sparsifying more general linear spaces, and start by identifying three specific properties of the contraction procedure from [KPS24] that we would like to emulate in our setting.

> Property 1: If we contract a code $\mathcal{C}'$ on coordinate j, then a codeword c remains in the contracted code if and only if $c_j = 0$.
>
> Property 2: There is a bounded number of contractions one can do before $\mathcal{C}'$ has dimension at most α.
>
> Property 3: Each contraction preserves the linear structure; i.e., $\mathcal{C}'$ is always a linear code. Furthermore $\mathcal{C}'$ is specified by at most n vectors in $\mathbb{F}_q^m$ and this specification can be maintained under contraction in polynomial time.

Roughly, the first two items are necessary to ensure that we can lower-bound the probability that a codeword c remains in the span of the generating matrix after repeated contractions. The final item is necessary to ensure that the contraction is well-behaved and can be manipulated in polynomial time, in the sense that one can contract on an already contracted code in polynomial time.

The natural analog of linear codes in $\mathbb{F}_q^m$ is additive sets over $\mathcal{C} \subseteq \mathbb{Z}_q^m$. There is no immediate notion of a basis, though one can start with a generating set of elements of $\mathcal{C}$ given by a matrix $G \in \mathbb{Z}_q^{m \times n}$ such that $\mathcal{C} = \{G.x | x \in \mathbb{Z}^n\}$. But this generating set, or even its size, is not unique — for example the set of vectors[4] generated by $(3,0)$ and $(0,2)$ in $\mathbb{Z}_6^2$ is the same as the set generated by $(3,2)$. Nevertheless, our goal is to sparsify such a matrix to one of size $\widetilde{O}(n)$.

A natural choice for contraction would be to pick a random coordinate j where some element of $\mathcal{C}'$ is non-zero and then to remove from $\mathcal{C}'$ all elements that are non-zero on the jth coordinate, and this is what we do. This immediately yields Property 1. It also preserves the additive structure, so the first part of Property 3 follows. To make the representation explicit we give an algorithm based on GCD computation (reminiscent of the Hermite Normal form computation). (See [Mic14b]) The tricky part is to argue Property 2. Here we can no longer count on reduction in dimension since we no longer have a vector space to work with. However group theory comes to our rescue — we notice that $\mathcal{C}$ is a subgroup of $\mathbb{Z}_q^m$ of size at most q^n and after each contraction it shrinks to a strict subgroup. Since subgroups have size at most half of any group containing them, it follows that there can be at most $n \log q$ iterations of contraction.

Armed with this contraction procedure we are able to establish a type of codeword-counting bound as in [Kar93, KPS24]. Specifically, we prove the following result, which plays a key role in our later sparsification method.

Theorem 2.1 (Informal version of Theorem 4.4)**.** *For every additive set $\mathcal{C} \subseteq \mathbb{Z}_q^m$ of size at most $q^n s$, and for every integer $d \geq 1$, there exists a set $S \subseteq [m]$ of size at most $n \log(q) \cdot d$, such that upon removing these rows, for any integer $\alpha \geq 1$, the resulting additive set has at most $\binom{n \log(q)}{\alpha} \cdot q^{\alpha+1}$ distinct elements of weight $\leq \alpha d$.*

The formal proof of this statement appears in Section 4. As an immediate consequence of this result, we are able to directly take advantage of some of the tools used in [KPS24] to conclude the existence of $(\epsilon, \widetilde{O}(n \log^2(m) \log^2(q)/\epsilon^2))$-sparsifiers for *unweighted* codes $\mathcal{C} \subseteq \mathbb{Z}_q^m$, with associated generating matrix $G \in \mathbb{Z}_q^{m \times n}$. This is formally stated and proved as Theorem 5.13.

However, as in [KPS24], this result is imperfect for several reasons. First, the dependence on $\log(m)$ in the sparsifier size is unaffordable, as m can be exponentially large in n, resulting in losses of polynomial factors of n in the sparsifier size. (We can't afford to simply re-invoke the sparsification result as is because the sparsified code now has associated weights.) Second, we need an analog to the efficient procedure to find a small superset T of the set S alluded to in Theorem 2.1 to get an efficient algorithm. The latter procedure turns out to be not too different than in the $\mathbb{F}_q$ case though leads to additional $\log q$ factor losses in the bound. Finally, we still only have a result for cyclic groups and not general Abelian groups (including infinite ones). We address these issues briefly next before moving on to the extensions to higher degree constraints.

2.4 Getting near-linear sized sparsifiers over all Abelian groups

To address the dependence on m, we create analogs of some of the key pieces used in the framework of [KPS24], and also introduce new techniques. To recap, the framework from [KPS24] first uses a simple algorithm to make quadratic size code-sparsifiers for codes of originally *unbounded* length. This algorithm operates by sampling each coordinate i of the code at rate $p_i = \frac{n}{\epsilon^2 \min_{c \in \mathcal{C}: c_i \neq 0} \mathrm{wt}(c)}$. That is, the sampling rate for coordinate i is inversely proportional to the minimum weight codeword which is non-zero in its ith coordinate. Unfortunately, this quantity is in fact NP-hard to calculate

[4] For lack of a better word, we abuse terminology here in referring to elements of $\mathbb{Z}_q^m$ as vectors even though they don't form a vector space.

(even to approximate), and further, the analysis used to show that these sampling rates preserve codeword weights is tuned specifically for the case of fields. To bypass this, we take advantage of the fact that we do not actually require *exactly* quadratic size sparsifiers, and in particular, any polynomially-bounded (in n) size sparsifier suffices. Thus, we can simply invoke the result discussed above for creating $(\epsilon, \widetilde{O}(n \log^2(m) \log^2(q)/\epsilon^2))$ sparsifiers, which will yield sparsifiers of polynomial size. As we will see later, this procedure can be made to run in polynomial time, and thus we can bypass the NP-hardness of this first "quadratic-size sparsifier" step used in [KPS24].

The second component of this algorithm is the so-called "weight-decomposition" step from [KPS24]. Roughly, the previous step returns a polynomially-bounded size sparsifier. But, each coordinate in this sparsifier can have weight which is potentially unbounded in n, and the original algorithm we defined *only works on unweighted codes*. If we naively try to take this weighted code and turn it into an unweighted code by repeating each coordinate a number of times proportional to its weight, we will end up back where we started, with a code of possibly exponential length.

Thus, the objective is to break the code up into several parts, each of which corresponds to coordinates with weights in a given range $[\alpha^i, \alpha^{i+1}]$, for some value α. As in [KPS24], the key insight comes from the fact that if a given codeword is non-zero in coordinates with large weight, it suffices to simply approximately preserve the weight of the codeword on these large weight coordinates, and ignore the lower weight coordinates. On the large weight coordinates, the ratio of the weights of the coordinates is bounded, so we can afford to simply repeat each coordinate a number of times proportional to its weight, and then sparsify this unweighted code.

However, the tricky part comes in showing how to cleanly restructure the code such that once we have preserved the codewords on coordinates of large weight, they are removed from the code, and instead it is only the codewords which were all 0 on these large weight coordinates that remain to be sparsified. Our key insight here is that it turns out that this corresponds *exactly* with performing a contraction on the coordinates of the code with large weight. This returns a new code where the only surviving codewords are those which were all 0 on the large weight coordinates. In particular, the number of remaining distinct codewords has sufficiently decreased such that we can take the union of all of the sparsifiers of all the different weight levels, while still not exceeding near-linear size. This leads to the proof of Theorem 5.17 in the general, non-field setting.

Affine Abelian Contraction Algorithm After establishing the result for $\mathbb{Z}_q$, the difficulty now comes in extending the algorithm to general Abelian groups A. We do this in Section 6, where we show a generalization of the contraction argument to Abelian groups, and then we are immediately able to conclude the existence of "code" sparsifiers in this regime, and by equivalence, CSP sparsifiers for affine, Abelian predicates by using the same framework established for $\mathbb{Z}_q$; we discuss the generalization of this contraction argument below.

In the more general Abelian group setting, our first difficulty comes in associating a "code" with the Abelian group A. In the setting of $\mathbb{Z}_q$, such an association is natural, as we simply place the coefficients of our affine equation as the entries in a row of the generating matrix G. In the Abelian setting, we instead take advantage of the Fundamental Theorem of Finite Abelian groups [Pin10]. Roughly speaking, this theorem states that for any Abelian group A, we can say that A is isomorphic to a group of the form

$$\mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_u}.$$

In particular, for any element $a \in A$, we can create a (linear) bijection sending a to a tuple $(d_1^{(a)}, \ldots d_u^{(a)}) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_u}$. Then, our result proceeds by creating a generating matrix G over these tuples, that is, each entry of G is an entry in $\mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_u}$. The benefit

now comes from the fact that arithmetic in *each individual entry of a tuple* operates the same as arithmetic over $\mathbb{Z}_q$. In particular, the contraction algorithm we defined for arbitrary $\mathbb{Z}_q$ extends to $\mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_u}$ by running the contraction algorithm once for each $\mathbb{Z}_{q_i}$ for $i \in [u]$ (once for each entry in the tuple). The correctness of this contraction algorithm then (largely) follows from the correctness of the algorithm over $\mathbb{Z}_q$, and so we can conclude the existence of a similar decomposition theorem / counting bound. This then allows us to conclude Theorem 1.7.

2.5 Sparsifying Symmetric CSPs

As a direct consequence of our sparsifiability result for affine predicates over $\mathbb{Z}_q$, we are able to precisely identify the symmetric, Boolean predicates P for which CSP(P) is sparsifiable to near-linear size, formally stated as Theorem 1.3.

The formal proof of this statement appears in Section 8. To prove this theorem, we show that for any symmetric predicate $P : \{0,1\}^r \to \{0,1\}$, if the zeros of P are periodic, then one can write a simple affine equation expressing the zeros of P. As before, we take *periodic* to mean that if $P_0(x) = 0, P_0(y) = 0$ for $x, y \in \{0, \dots r\}$, then for $z = 2x - y$, or $z = 2y - x$, we have $P_0(z) = 0$. If this is the case, we show there will be an equation of the form

$$P(x) = \mathbf{1}[\sum_{i=1}^{r} a_i x_i + b \neq 0 \mod q],$$

where q is some integer $\leq r+1$. In this context, q will be the *period* of P, meaning that the levels of the predicate P which evaluate to 0 will be exactly q apart. It is inherent in this formulation that the period of P may be a composite number, hence requiring the generalization of the contraction method to non-fields. We can consider for instance the predicate $P : \{0,1\}^6 \to \{0,1\}$, where $P(x) = 0$ if and only if $|x| = 1, 5$. It follows then that we can express

$$P(x) = \mathbf{1}[\sum_{i=1}^{6} x_i + 3 \neq 0] \mod 4,$$

and one can check there is no similar way to express the predicate as such an affine equation modulo a prime.

We complement the result above by showing that if the zeros of a predicate P are *not* periodic, then there must exist a projection to an AND of arity at least 2. Indeed, any single witness to non-periodicity, i.e. a triple of the form $c_1, c_2, c_3 = 2c_2 - c_1$, where P is 0 on strings with c_1 or c_2 1's, but is 1 on c_3 1's is enough to conclude the existence of a projection to an AND of arity 2. Thus, periodicity of the zeros of a symmetric predicate is a necessary and sufficient condition for being sparsifiable to near-linear size.

2.6 Non-trivial sparsification for almost all predicates

Finally, we prove a broad generalization of a result from Filtser and Krauthgamer [FK17]. [FK17] showed that when dealing with predicates of the form $P : \{0,1\}^2 \to \{0,1\}$, all CSP instances with predicate P are sparsifiable to near-linear size in the number of variables if and only if $|P^{-1}(1)| \neq 1$, i.e. the predicate P does not have only one satisfying assignment.

We extend this result, and show that for any r (i.e. not only $r = 2$ as in [FK17]), if a predicate $P : \{0,1\}^r \to \{0,1\}$ satisfies $|P^{-1}(1)| \neq 1$, then we can sparsify CSP instances with predicate P to size $\widetilde{O}(n^{r-1})$. In fact, our result is slightly stronger than this, as it even applies to a CSP with *multiple* predicates. We show that for any CSP with predicates $P_1, \dots P_m$ such that each P_i satisfies

$|P_i^{-1}(1)| \neq 1$, the entire system can be sparsified to size $\widetilde{O}(n^{r-1})$. Our result is formally stated as Theorem 1.5.

This result is tight as there exist predicates with at least 2 satisfying assignments that require sparsifiers of size $\Omega(n^{r-1})$. Indeed, consider for instance the predicate $P : \{0,1\}^r \to \{0,1\}$ such that $P(1, x_1, \dots x_{r-1}) = 1$, and $P(0, x_1, \dots x_{r-1}) = \text{AND}(x_1, \dots x_{r-1})$. This predicate will have $2^{r-1} + 1$ satisfying assignments, but also has a projection to an AND of arity $r - 1$, and thus requires sparsifiers of size $\Omega(n^{r-1})$.

Deriving the sparsification result is non-trivial. Indeed, the sparsification procedures created in this paper and the prior work of [KPS24] both rely on being able to represent a predicate as the non-zero assignments to an affine predicate. So, in order to generalize our sparsification method to a broader class of predicates, we show that for any predicate $P : \{0,1\}^r \to \{0,1\}$, if there exists a degree ℓ polynomial Q over $\mathbb{F}_q$ such that $\forall y \in \{0,1\}^r, P(y) = \mathbf{1}[Q(y) \neq 0]$, we can then sparsify CSPs with predicate P to size $\widetilde{O}(n^\ell/\epsilon^2)$. To see why this result follows intuitively, for a degree ℓ polynomial on n variables, there are $O(n^\ell)$ possible terms in this polynomial. Naturally, the polynomial is already *linear* over all of these terms, so if we make a new variable for each term, we can represent the polynomial as a linear equation over all of the new variables (but this leads to a degradation in sparsifier size as we are now sparsifying over a larger universe of variables).

Beyond this point, the difficulty comes in showing that if a predicate has at least two satisfying assignments, there exists a polynomial of degree $r - 1$ which exactly captures this predicate. Unfortunately, we are unable to show this fact exactly: instead, we show that for any two assignments $y_1, y_2 \in \{0,1\}^r$, one can create a polynomial P_{y_1,y_2} over $\mathbb{Z}_2$ of degree $r - 1$ which is 1 only on y_1, y_2. It follows then that for a general predicate $P : \{0,1\}^r \to \{0,1\}$ with satisfying assignments $y_1, \dots y_s$, one can write

$$P = P_{y_1,y_2} \vee P_{y_1,y_3} \vee \cdots \vee P_{y_1,y_s}.$$

Thus, all that remains is to show that one can simulate the OR of several predicates without blowing up the size of the sparsifier. To do this, we take advantage of our result on the sparsifiability of Abelian groups. Indeed, for an example such as the one above, we instead operate over the Abelian group $(\mathbb{Z}_2)^s$. The elements of this group look like tuples

$$(x_0, x_1, \dots x_{s-1}).$$

Ultimately, we then create the predicate over $(\mathbb{Z}_2)^s$ which looks like

$$P(x) = (P_{y_1,y_2}(x), P_{y_1,y_3}(x), \dots, P_{y_1,y_s}(x)) = (P_{y_1,y_2}, P_{y_1,y_3}, \dots, P_{y_1,y_s})(x).$$

Note that because each polynomial $P_{y_1,y_i}(x)$ only takes values from $\mathbb{Z}_2$, the above polynomial is zero (i.e. the zero tuple) if and only if every polynomial P_{y_1,y_i} evaluates to 0. Otherwise, if any individual polynomial evaluates to 1 on x, the output is non-zero. Finally, we can conclude by invoking our result on the sparsifiability affine Abelian groups.

One consequence of this result is that it gives us a *complete* characterization of the sparsifiability of CSPs with predicates on 3 variables. Indeed, in the work of [KPS24], it was shown that any predicate P of arity 3 which does not have a projection to AND is sparsifiable to near-linear size (in n). Using the above result, we know that any predicate P of arity 3 which has at least 2 satisfying assignments is sparsifiable to quadratic size. Thus, the only predicate which is not sparsifiable to quadratic size is the predicate with exactly one satisfying assignment, namely, $\text{AND}(x_1, x_2, x_3)$ up to negations. For this predicate, one can clearly see a sparsification lower bound of size $\Omega(n^3)$. Thus, piecing this all together, we give a *complete* characterization of the sparsifiability of predicates of arity 3. That is,

Theorem 1.6. *For a predicate $P : \{0,1\}^3 \to \{0,1\}$ let c be the largest integer such that P has a projection to AND_c. Then, CSP(P) is $(\epsilon, \widetilde{\Theta}(n^c))$ efficiently-sparsifiable for every $\epsilon \in (0,1)$, and moreover, for every $\epsilon \in (0,1)$, CSP(P) is not $(\epsilon, o(n^c))$ sparsifiable.*

3 Preliminaries

In this work, when we use $\mathbb{F}_q$, this refers to the field with q elements (and as such q must necessarily be prime or a prime power). When we say $\mathbb{Z}_q$, this is simply the group of integers $\mod q$, and as such we do not require that q is prime or a prime power (and in fact, the focus will be on the case where q is composite).

Definition 3.1. *We say that a* code $\mathcal{C} \subseteq \mathbb{Z}_q^m$ *is simply a subspace of $\mathbb{Z}_q^m$ closed under linear combinations. Thus, we can associate with any such code $\mathcal{C}$ a* generating matrix $G \in \mathbb{Z}_q^{m \times n}$ *such that $\mathcal{C} = Span(G)$ (specifically, $\mathcal{C} = \{Gx : x \in \mathbb{Z}_q^n\}$).*

The coordinates *of the code are $[m]$. When we refer to the* number of coordinates*, this is interchangeable with the* length *of the code, which is exactly m.*

Note that typically code are generated over fields, but in this work we do not restrict ourself to this setting.

In this work, we will be concerned with code sparsifiers as defined below.

Definition 3.2. *For a code $\mathcal{C} \subseteq \mathbb{Z}_q^m$ with associated generating matrix $G \subseteq \mathbb{Z}_q^{m \times n}$, a $(1 \pm \epsilon)$-sparsifier for $\mathcal{C}$ is a subset $S \subseteq m$, along with a set of weights $w_S : S \to \mathbb{R}^+$ such that for any $x \in \mathbb{Z}_q^n$*

$$(1-\epsilon)\mathrm{wt}(Gx) \leq \mathrm{wt}_S(G|_S x) \leq (1+\epsilon)\mathrm{wt}(Gx).$$

Here, wt_S is meant to imply that if the codeword is non-zero in its coordinate corresponding to an element $i \in S$, then it contributes $w_S(i)$ to the weight. We will often denote $G|_S$ with the corresponding weights as $\widetilde{G}$.

We next present a few simple results for code sparsification that we will use frequently.

Claim 3.1. *For a vertical decomposition of a generating matrix,*

$$G = \begin{bmatrix} G_1 \\ G_2 \\ \vdots \\ G_k \end{bmatrix},$$

if we have a $(1 \pm \epsilon)$ sparsifier to codeword weights in each G_i, then their union is a $(1 \pm \epsilon)$ sparsifier for G.

Proof. Consider any codeword $c \in \mathrm{Span}(G)$. Let c_i denote the restriction to each G_i in the vertical decomposition. It follows that if in the sparsifier $\mathrm{wt}(\hat{c}_i) \in (1 \pm \epsilon)\mathrm{wt}(c_i)$, then $\mathrm{wt}(\hat{c}) = \sum_i \mathrm{wt}(\hat{c}_i) \in (1 \pm \epsilon) \sum_i \mathrm{wt}(c_i) = (1 \pm \epsilon)\mathrm{wt}(c)$. □

Claim 3.2. *Suppose $\mathcal{C}'$ is $(1 \pm \delta)$ sparsifier of $\mathcal{C}$, and $\mathcal{C}''$ is a $(1 \pm \epsilon)$ sparsifier of $\mathcal{C}'$, then $\mathcal{C}''$ is a $(1-\epsilon)(1-\delta), (1+\epsilon)(1+\delta)$ sparsifier to $\mathcal{C}$ (i.e. preserves the weight of any codeword to a factor $(1-\epsilon)(1-\delta)$ below and $(1+\epsilon)(1+\delta)$ above).*

Proof. Consider any codeword $\mathcal{C}x$. We know that $(1-\epsilon)\mathrm{wt}(\mathcal{C}x) \leq \mathrm{wt}(\mathcal{C}'x) \leq (1+\epsilon)\mathrm{wt}(\mathcal{C}x)$. Additionally, $(1-\delta)\mathrm{wt}(\mathcal{C}'x) \leq \mathrm{wt}(\mathcal{C}''x) \leq (1+\delta)\mathrm{wt}(\mathcal{C}'x)$. Composing these two facts, we get our claim. □

Claim 3.3. ([FHHP11]) *Let $X_1, \ldots X_\ell$ be random variables such that X_i takes on value $1/p_i$ with probability p_i, and is 0 otherwise. Also, suppose that $\min_i p_i \geq p$. Then, with probability at least $1 - 2e^{-0.38\epsilon^2\ell p}$,*

$$\sum_i X_i \in (1 \pm \epsilon)\ell.$$

We will use the following result from the work of Khanna, Putterman, and Sudan [KPS24]:

Theorem 3.4. *For a predicate $P : \{0,1\}^r \to \{0,1\}$, if there exists $\pi : \{x_1, \ldots x_r\} \to \{0, 1, a, \neg a, b, \neg b\}$ such that $P(\pi(x_1), \ldots \pi(x_r)) = AND(a, b)$, then there exist CSPs with predicate P that require sparsifiers of size $\Omega(n^2/r^2)$.*

While the result [KPS24] is specifically only stated for ANDs of arity 2, note that their argument generalizes to ANDs of arity r. I.e., if there exists a projection to ANDs of arity ℓ, then there exist instances which require sparsifiers of size $\Omega(n^\ell)$ for any constant $0 < \epsilon < 1$.

We will frequently make use of the following facts about the Euclidean algorithm:

Fact 3.5 (Euclidean Algorithm)**.** (See among many others, [Wei]) There exists an algorithm which upon being given a list of integers $y_1, \ldots y_r$:

1. Sets $x_1 = y_1, \ldots x_r = y_r$.
2. Only performs operations of the form $x_i \leftarrow x_i + \ell \cdot x_j$ (for $\ell \in \mathbb{Z}$).
3. Terminates with $x_1 = \mathrm{GCD}(y_1, \ldots y_r)$.

Note that while many algorithms are only stated for computing the GCD of any two numbers, one can simply calculate the GCD of more numbers iteratively, adding one new numbers to the computation in each iteration. I.e., in the case of 3 numbers, one can compute $\mathrm{GCD}(x_1, \mathrm{GCD}(x_2, x_3))$.

4 A Decomposition Theorem for Codes over $\mathbb{Z}_q$

We start by working towards a proof of Theorem 1.7 for the special case of finite cyclic groups, i.e., groups of the form $\mathbb{Z}_q$ for some (potentially composite) integer q. In this section, we will prove a decomposition theorem (Theorem 4.4) for general linear spaces over $\mathbb{Z}_q$ and follow this with an efficient algorithm for computing this decomposition (Theorem 4.6). This theorem is analogous to Theorem 2.1 in [KPS24], with the key difference being our decomposition is also algorithmic. This forms the basis of all of our algorithmic sparsification results. We will use this efficient decomposition in Section 5 to prove Theorem 1.7 over $\mathbb{Z}_q$.

To do this, we first define a contraction algorithm on the generating matrix of this linear space. We show that with high probability a codeword of low-weight will survive this contraction procedure, and remain in the span of the generating matrix, provided the "support" of the generating matrix never becomes too small. Using this, we are able to prove a decomposition theorem about linear spaces over $\mathbb{Z}_q$. Either there exists a linear subspace with small "support" that contains many distinct codewords, or the contraction procedure shows that there can not be too many codewords of light weight.

Definition 4.1. *For a linear space $\mathcal{C} \subseteq \mathbb{Z}_q^m$, we say that the **support** of $\mathcal{C}$ is the set of coordinates that are not always 0. That is,*

$$\mathrm{Supp}(\mathcal{C}) = \{i \in [m] : \exists c \in \mathcal{C} : c_i \neq 0\}.$$

Definition 4.2. *Using the definition of support, we can likewise define the **density** of a linear space. For $\mathcal{C} \subseteq \mathbb{F}_q^m$,*

$$\mathrm{Density}(\mathcal{C}) = \frac{\log_2 |\mathcal{C}|}{|\mathrm{Supp}(\mathcal{C})|}.$$

Definition 4.3. *For a linear subspace $\mathcal{C} \subseteq \mathbb{Z}_q^m$, a **subcode** of $\mathcal{C}$ is any subspace $\mathcal{C}' \subseteq \mathcal{C}$ which is closed under linear combinations.*

We are now ready to define our contraction algorithm.

Algorithm 1: Contract(G, j)

1 Run the Euclidean GCD algorithm (Fact 3.5) on the jth row of generating matrix G (treating the entries as being over $\mathbb{Z}$), using column operations to get the GCD of the jth row into a column, say this column is v. Swap v with the first column.
2 Add multiples of v to cancel out the jth entry of every other column in the jth coordinate. I.e., for $i \in \{2, \dots n\}$, $G_i \leftarrow G_i - ((G_i)_j / v_j) \cdot v$.
3 Replace column v with $\eta \cdot v$, where $\eta = \min\{\ell \in [1, \dots, q] : v_j \cdot \ell = 0\}$.
4 **return** G

Algorithm 2: Repeated Contractions(G, α, q)

1 Input generating matrix $G \in \mathbb{Z}_q^{m \times n}$.
2 **while** *more than $q^{\alpha+1}$ distinct codewords in the span of G* **do**
3 Choose a random non-zero row j of G.
4 Set $G = \mathrm{Contract}(G, j)$.
5 **end**

Claim 4.1. *Consider a codeword $c \in \mathcal{C}$ where $\mathcal{C}$ is the span of G. If we run Contract(G, j) on a coordinate j such that $c_j = 0$, then c will still be in the span of G after the contraction.*

Proof. First, we show that after line 1 in the Algorithm 1, c is still in the span. Indeed, the Euclidean GCD algorithm only ever makes column operations of the form $v_1 + r \cdot v_2 \to v_1$ (see Fact 3.5). This means that every step of the algorithm is invertible, so in particular, after every step of the Euclidean GCD algorithm, the span of the matrix G will not have changed.

Now, in line 2 of Algorithm 1, this again does not change the span of G. Indeed, we can simply undo this step by adding back multiples of v to undo the subtraction. Since the span of G hasn't changed after this step, c must still be in the span. Further, note that this step is possible *because* v_j is the GCD of the jth row of the generating matrix. I.e., there must be some multiple of v_j which cancels out every other entry in the jth row.

Finally, in line 3, the span of G actually changes. However, since c was zero in its jth coordinate, it could only contain an integer multiple of η times the column v (again where η is defined as $\eta = \min\{\ell \in [q] : v_j \cdot \ell = 0\}$), as otherwise it would be non-zero in this coordinate. Hence c will still be in the span of the code after this step, as it is only codewords that are non-zero in coordinate j that are being removed. □

Claim 4.2. *After calling Contract(G, j) on a non-zero coordinate j, the number of distinct codewords in the span of G decreases by at least a factor of 2.*

Proof. Consider the matrix G after line 2 of Algorithm 1. By the previous proof, we know that the span of G has not changed by line 2. Now, after scaling up v by η, we completely zero out row j of G. Previously, for any codeword in the code which was zero in coordinate j, we could correspondingly add a single multiple of column v to it to create a new codeword that was non-zero in its jth coordinate. Thus, the number of codewords which were non-zero in coordinate j was at least as large as the number which were zero. After scaling up column v, codewords which were non-zero in this coordinate are no longer in the span, so the number of distinct codewords has gone down by at least a factor of 2. □

Claim 4.3. *Let $\mathcal{C}$ be the span of $G \subseteq (\mathbb{Z}_q)^{m \times n}$, and let d be an integer. Suppose every subcode $\mathcal{C}' \subseteq \mathcal{C}$ satisfies* $\text{Density}(\mathcal{C}') < \frac{1}{d}$. *Then for any $\alpha \in \mathbb{Z}^+$, the number of distinct codewords of weight $\leq \alpha d$ is at most $\binom{n \log(q)}{\alpha} \cdot q^{\alpha+1}$.*

Proof. Consider an arbitrary codeword c in the span of G of weight $\leq \alpha d$. Let us consider what happens when we run Algorithm 2. In the worst case, we remove only a factor of 2 of the codewords in each iteration. Thus, suppose that this is indeed the case. Suppose that the code starts with q^n distinct codewords. Now, after ℓ iterations, this means in the worst case there are $2^{n \log(q) - \ell}$ distinct codewords remaining. Note that after running these contractions, the resulting generating matrix defines a subcode $\mathcal{C}'$ of our original space. Then, by our assumption on the density of every subcode, at this stage, the number of non-zero rows in the generating matrix must be at least $(n \log(q) - \ell)d$. Thus, the probability that c remains in the span of the generating matrix after the next contraction is at least

$$\Pr[c \text{ survives iteration}] \geq 1 - \frac{\alpha d}{(n \log(q) - \ell)d} = 1 - \frac{\alpha}{n \log(q) - \ell},$$

where this is simply the probability that we sample a coordinate in which $c_j = 0$.

Now, we can consider the probability that c survives all iterations. Indeed,

$$\begin{aligned} \Pr[c \text{ survives all iterations}] &\geq \frac{n \log(q) - \alpha}{n \log(q)} \cdot \frac{n \log(q) - \alpha - 1}{n \log(q) - 1} \cdot \cdots \cdot \frac{\alpha + 1 - \alpha}{\alpha + 1} \\ &= \binom{n \log(q)}{\alpha}^{-1}. \end{aligned}$$

At this point, the number of remaining codewords is at most $q^{\alpha+1}$. Hence, the total number of codewords of weight $\leq \alpha d$ is at most $\binom{n \log(q)}{\alpha} \cdot q^{\alpha+1}$. □

Theorem 4.4. *For a code $\mathcal{C} \subseteq \mathbb{Z}_q^n$ with at most q^n distinct codewords, and any arbitrary integer $d \geq 1$, at least one of the following is true:*

1. *There exists $\mathcal{C}' \subseteq \mathcal{C}$ such that* $\text{Density}(\mathcal{C}') \geq \frac{1}{d}$.
2. *For any integer $\alpha \geq 1$, the number of distinct codewords of weight $\leq \alpha d$ is at most $\binom{n \log(q)}{\alpha} \cdot q^{\alpha+1}$.*

Proof. Suppose condition 1 does not hold, and invoke Claim 4.3. □

Corollary 4.5. *For any linear code $\mathcal{C}$ over $\mathbb{Z}_q^m$, with $\leq q^n$ distinct codewords, and for any integer $d \geq 1$, there exists a set of at most $n \log(q) \cdot d$ rows, such that upon removing these rows, for any integer $\alpha \geq 1$, the resulting code has at most $\binom{n \log(q)}{\alpha} \cdot q^{\alpha+1}$ distinct codewords of weight $\leq \alpha d$.*

Proof. We use Theorem 4.4. Suppose for the code $\mathcal{C}$ that condition 2 does not hold. Then, condition 1 holds, which means there exists a set $n'd$ rows, with at least $2^{n'}$ distinct codewords that are completely contained on these rows. Now, by removing these rows, this yields a new code where the number of distinct codewords has decreased by a factor of $2^{n'}$. Thus, if removing these rows yields a new code $\mathcal{C}'$, this new code has at most $q^n/2^{n'}$ distinct codewords. Now, if condition 2 holds, for $\mathcal{C}'$, we are done. Otherwise, we can again apply this decomposition, removing another $n''d$ rows to yield a new code with at most $2^{n\log(q)-n'-n''}$ distinct codewords. In total, as long as we remove rows in accordance with condition 1, we can only ever remove $n\log(q)d$ rows total, before there are no more distinct codewords remaining, and condition 2 must hold.

Thus, there exists a set of at most $n\log(q)\cdot d$ rows, such that upon their removal, the new code satisfies condition 2. □

4.1 Efficient Algorithms for Computing Decomposition

In this subsection, we will show how to *efficiently* find (a superset of) the rows necessary for the decomposition guaranteed to exist. In particular, while we may not be able to exactly identify the set of rows S of the generating matrix which need to be removed, we will be able to find a set T of $\leq 2n\log(q)\log(n)\cdot d$ rows, such that $S\subseteq T$. It follows then that if we remove the set of rows T, then uniform sampling should suffice for preserving the weights of codewords.

We summarize this in the following theorem:

Theorem 4.6. *For any linear code $\mathcal{C}$ over $\mathbb{Z}_q^m$, with $\leq q^n$ distinct codewords, and for any integer $d\geq 1$, there exists a set S of at most $n\log(q)\cdot d$ rows, such that upon removing these rows, for any integer $\alpha\geq 1$, the resulting code has at most $\binom{n\log(q)}{\alpha}\cdot q^{\alpha+1}$ distinct codewords of weight $\leq \alpha d$. Further, there is an efficient algorithm for recovering a set $T\subseteq[m]$ such that $S\subseteq T$, and $|T|\leq nd\log(q)(\log(n)+\log(q))$.*

Roughly speaking, the method we use in this section should be thought of as a linear algebraic analog to storing spanning forests of a graph. [BK96] showed that if one stores $2\log(n)\cdot d$ spanning forests of a graph, then one can sample the rest of the graph at rate roughly $\frac{1}{d}$ while still preserving the sizes of all cuts in the graph. In our case, the natural analog of a spanning forest will be a subset T_1 of the rows of the generating matrix G, such that the number of distinct codewords in the span of $G|_{T_1}$ is the same as in G. We define this notion more formally below:

Definition 4.4. *For a generating matrix $G\in\mathbb{Z}_q^{m\times n}$, we say that a **maximum spanning subset** $T\subseteq[m]$ is a subset of the rows of the generating matrix G such that the number of distinct codewords in the span of $G|_T$ is the same as in the span of G.*

Claim 4.7. *In order to efficiently construct a maximum spanning subset of $G\in\mathbb{Z}_q^{m\times n}$ of size $\leq n\log(q)$, it suffices to be able to* efficiently *compute the number of distinct codewords in the span of an arbitrary generating matrix $H\in\mathbb{Z}_q^{m'\times n'}$.*

Proof. Consider the following simple algorithm:

Algorithm 3: BuildMaxSpanningSubset(H)

```
Initialize T = ∅.
Let k = 0.
for i ∈ [m] do
    If the number of distinct codewords in the span of H|_{T∪{i}} is ≥ k, then set T = T ∪ {i},
      and k ← the number of distinct codewords in the span of H|_{T∪{i}}.
end
return T
```

Clearly, this is efficient if the procedure for checking the number of distinct codewords in the span of $G|_T$ is efficient. Further, it is clear to see that this yields a spanning subset, as any row which is not kept does not increase the number of codewords. So, it remains to prove the size bound. For this, we simply remark that if adding a row to $G|_T$ increases the number of distinct codewords, then it *at least* doubles the number of distinct codewords. To see why, suppose we have a set T and a set $T \cup \{i\}$ such that $G|_{T\cup\{i\}}$ has more distinct codewords in its span than $G|_T$. Consider any two messages $x_1, x_2 \in \mathbb{Z}_q^n$ such that $G|_T x_1 = G|_T x_2$, yet $G|_{T\cup\{i\}} x_1 \neq G|_{T\cup\{i\}} x_2$. It must be the case then that $G|_{T\cup\{i\}}(x_1 - x_2)$ is non-zero only in the last coordinate (corresponding to row i), as the two vectors are equal for all the rows in T. To conclude, we can then see that for any codeword $z = G|_T x$, there are at least two *distinct* corresponding codewords in the span of $G|_{T\cup\{i\}}$, i.e., $G|_{T\cup\{i\}} x$ and $G|_{T\cup\{i\}}(x + x_1 - x_2)$.

Thus, each row which increases the number of codewords increases it by at least a factor of 2. Because there can be at most q^n distinct codewords, this means the size of the spanning subset will be at most $n \log(q)$. □

Claim 4.8. *There is an efficient algorithm for exactly calculating the number of distinct codewords in the span of a generating matrix $G \in \mathbb{Z}_q^{m\times n}$.*

Proof. First, note that from the analysis of the contraction algorithm, we can assume without loss of generality that G is given to us in a form where the first row has a single non-zero entry in the first column (as if G is not, then we can efficiently get G into such a form in time $O(mn\log(q))$). Now, let this non-zero entry be denoted by $a \in \mathbb{Z}_q$, and let c be the smallest non-negative integer such that $c \cdot a = 0$ over $\mathbb{Z}_q$. Let G' denote the resulting matrix when we replace the first column c_1 of G with $c \cdot c_1$. In particular, this forces the entire first row of G' to be 0.

Then, we will show that the number of distinct codewords in the span of G is exactly c times greater than the number of distinct codewords in the span of G'.

To see this, we will show that there is an equivalence class in the codewords of G, such that there are exactly c codewords in the span of G that map to each codeword in the span of G'. In particular, for a codeword $z \in G'$, we denote the corresponding codewords in the span of G by $z + \alpha \cdot c_1$, where $\alpha \in \{0, 1, \ldots c-1\}$. Clearly, this yields a map from elements in the image of G' to *every* codeword in the span of G. Indeed, for any codeword in the span of G, it can be written as

$$y = \sum_{i=1}^{n} \alpha_i c_i = (\alpha_1 \mod c) \cdot c_1 + ((\alpha_1 - (\alpha_i \mod c))/c) \cdot c \cdot c_1 + \sum_{i=2}^{n} \alpha_i c_i.$$

It remains to show that this map is unique. Indeed, suppose that for a codewords y in the image of G we can write it as $y = z_1 + \beta_1 c_1 = z_2 + \beta_2 c_1$. Now, observe that it must be the case that $\beta_1 = \beta_2$ because $(z_1)_1 = 0$ and $(z_2)_1 = 0$. This follows because the first row of the matrix G' is all zeros, so any codeword generated by G' must be zero in its first coordinate. Further, because c is the smallest value such that $c \cdot (c_1)_1 = 0$, for any values $0 \leq \beta < c$, the values $\beta \cdot (c_1)_1$ must be distinct. Thus, we see that $y = z_1 + \beta_1 c_1 = z_2 + \beta_1 c_1$, and so it follows that $z_1 = z_2$.

Thus, the natural algorithm for computing the number of distinct codewords is as follows: given the generating matrix G, contract on the first row to get a generating matrix G', and keep track of the value c that was used to scale the first column. Then, inductively, the number of distinct codewords in the span of G is c times the number of distinct codewords in the span of G'.

The running time for this algorithm is $O(mn^2 \log^2(q))$, as there can be at most $n\log(q)$ rows we contract on, each of which takes time $O(mn\log(q))$. □

Next, we will show that storing disjoint maximum spanning subsets suffices for recovering the set S guaranteed by Corollary 4.5. First however, let us introduce some notation regarding this set S of bad rows in G.

Recall that we are given a parameter d, and we wish to remove a set of $\leq n\log(q)d$ bad rows, such that the codeword counting bound will hold with parameter d after removing these rows. As mentioned, S is really the support of the union of several subspaces which are *particularly* dense. That is, if we let C denote the span of G, and we analyze the following expression

$$\max_{C'\subseteq C}\frac{\log|C'|}{|\text{Supp}(C')|},$$

if the expression is $\geq 1/d$, then the subspace is too dense for our counting bound to hold. When this occurs, we let C'_1 denote the corresponding maximizing subspace, we let S_1 denote the support of C'_1, and we let k_1 denote the log of the number of distinct codewords removed by taking away the rows in S_1. Now, after removing this subspace C'_1 (i.e., removing the support of C'_1 from C), it is still possible that the counting bound is violated. Thus, we continue to analyze the expression

$$\max_{C'\subseteq C|_{\bar{S_1}}}\frac{\log|C'|}{|\text{Supp}(C')|},$$

and if the expression is $\geq 1/d$, then again we let C'_2 denote the maximizing subspace, S_2 denote the support of C'_2, and we let $k_2=\log|C_{\bar{S_1}}|-\log|C_{\bar{S_1}\cap\bar{S_2}}|$. We continue doing this iteratively until finally

$$\max_{C'\subseteq C|_{\bar{S_1}\cap\cdots\cap\bar{S_\ell}}}\frac{\log|C'|}{|\text{Supp}(C')|}<1/d.$$

We let the sequence of subspaces that are recovered be denoted by $C'_1,\ldots C'_\ell$, let $S_i=\text{Supp}(C'_i)$, and let $k_i=\log|C_{\bar{S_1}\cap\ldots S_{i-1}^-}|-\log|C_{\bar{S_1}\cap\ldots S_{i-1}^-\cap\bar{S_i}}|$. Under these definitions, note that $S=S_1\cup\cdots\cup S_\ell, k=\sum_{i=1}^{\ell}k_i$, and therefore $|S|\leq kd$.

Now, let $T_1,\ldots T_{2d(\log(n)+\log(q))}$ denote *disjoint* maximum spanning subsets of a generating matrix $G\in\mathbb{Z}_q^{m\times n}$. Note that computing such a set of disjoint maximum spanning subsets can be done efficiently, by iterating through the rows of G to first create T_1, and then removing the rows from T_1 from G, re-iterating through G and creating a maximum spanning subset T_2, etc.

That is, we can use the following algorithm:

Algorithm 4: ConstructSpanningSubsets(H,t)

```
for i ∈ [t] do
    Let T_i = BuildMaxSpanningSubset(H|_{T̄_1∩...T_{i-1}^-}).
end
return T_i : i ∈ [t].
```

Clearly, the spanning subsets are disjoint, and the run-time is bounded by $O(t\cdot mn^2\log^2(q))$.

Thus, it remains to show the following claim:

Claim 4.9. *For a generating matrix $G\in\mathbb{Z}_q^{m\times n}$, any choice of d and the set S of bad rows (i.e. those guaranteed by Corollary 4.5) for G and parameter d, for any disjoint maximum spanning subsets $T_1,\ldots T_{2d\log(q)(\log(n)+\log(q))}$, we have that $S\subseteq T_1\cup T_2\cdots\cup T_{2d\log(q)(\log(n)+\log(q))}$.*

Proof. First, we will adopt the notation for S_i,k_i,C'_i,T_i for $i=1,\ldots\ell$ that was created in the preceding paragraph.

Let us consider a single maximum spanning subset T_1 of G along with the set S. We let $k = \sum_{i=1}^{\ell} k_i$ denote the sum of the logs of the number of distinct codewords that were removed. In particular then, we know that after removing the supports of $C'_1, \dots C'_\ell$ from C (i.e., removing the set S), then $\log(|C_{\bar{S}}|) = \log(|C|) - k$. So, in order to get a *maximum spanning subset* (where the log of the number of distinct codewords is $\log(|C|)$), it must be the case that T_1 includes at least $k/\log(q)$ rows from S, as each row we include increases the number of codewords by a factor of at most q (and the log by at most $\log(q)$), and there is a factor of 2^k distinct codewords that must be captured by these rows. However, naively repeating this argument does not suffice, as it is possible that as we recover rows in the T_i's, the remaining number of distinct codewords to be recovered from $C|_S$ decreases (i.e., perhaps $\log(|C_{\bar{T}}|) - \log(|C_{\bar{T}\cap\bar{S}}|) << k$). Instead, we argue in a manner reminiscent of [BK96]. After the first $2d\log(q)$ disjoint maximum spanning subsets are recovered (and denote the set of rows they recover by T), there are two cases:

1. One case is that $\log(|C_{\bar{T}}|) - \log(|C_{\bar{T}\cap\bar{S}}|) \geq k/2$. However, this yields a contradiction, as we know that the set S is of size at most kd. The above implies that for $2d\log(q)$ iterations, each maximum spanning subset must have recovered at least $k/(2\log(q))$ (distinct) rows from S. This is because any maximum spanning subset *must* contain at least $k/(2\log(q))$ rows from the support of these subspaces (otherwise they are not *maximally* spanning) in order to have the maximum possible number of distinct codewords. But, then we must have recovered at least kd rows from S total, which would have exhausted the entire set.

2. The other case is that $\log(|C_{\bar{T}}|) - \log(|C_{\bar{T}\cap\bar{S}}|) < k/2$. That is, among the first $2d\log(q)$ maximum spanning subsets that are recovered, these contained sufficiently many rows from S such that the difference between the log of the remaining number of distinct codewords in $C_{\bar{T}}$ versus in $C_{\bar{T}\cap\bar{S}}$ decreased by more than half. We want to argue that this means there are at most $kd/2$ remaining rows that must be recovered. The key claim we will use is the following:

   Claim 4.10. *If* $\log(|C_{\bar{S}_1\cap\dots\cap S_{i-1}^-}|) - \log(|C_{\bar{S}_1\cap\dots\cap\bar{S}_i}|) = k_i$ *with corresponding support* S_i*, and after removing a set* T $\log(|C_{\bar{S}_1\cap\dots\cap S_{i-1}^-\cap\bar{T}}|) - \log(|C_{\bar{S}_1\cap\dots\cap\bar{S}_i\cap\bar{T}}|) < k_i - s_i$*, then* $|S_i/T| \leq (k_i - s_i)d$.

   Proof. First, note that after removing $S_1, \dots S_{i-1}$, C'_i was the maximizer for the expression

$$\max_{C'\subseteq C|_{\bar{S}_1\cap\dots S_{i-1}^-}} \frac{\log(|C'|)}{|\text{Supp}(C')|},$$

   and in particular, C'_i achieved some value $1/d_i \geq 1/d$ for this expression, and had $\log(|C_{\bar{S}_1\cap\dots\cap S_{i-1}^-}|) - \log(|C_{\bar{S}_1\cap\dots\cap\bar{S}_i}|) = k_i$. Now, we claim that if we removed some rows T from the support of C'_i such that now $\log(|C_{\bar{S}_1\cap\dots\cap S_{i-1}^-\cap\bar{T}}|) - \log(|C_{\bar{S}_1\cap\dots\cap\bar{S}_i\cap\bar{T}}|)$ is $\leq k_i - s_i$, then T must have recovered at least $s_i \cdot d_i$ rows from the support of C'_i. This follows because instead of removing the entirety of S_i, we instead only remove $T \cap S_i$, yet still find a reduction in log of the number of distinct codewords by k_i. In particular then, the subspace defined on $T \cap S_i$ contains $\geq s_i$ of the log of the number of distinct codewords contributed by $C|_{S_i}$.

   So, if T removed fewer than $s_i \cdot d_i$ rows, yet still decreased the log of the number of distinct codewords by s_i, this means there must have been a *more optimal* subspace C''_i, defined on a support of size $< s_i d_i$ with at least 2^{s_i} distinct codewords, therefore contradicting the optimality of C'_i. Thus, we get the stated claim. □

Now, note that originally, the spaces $C'_1, \dots C'_\ell$ had logs of the number of distinct codewords they contributed totaling $k_1, \dots k_\ell$. We are in the case where after removing the rows from T, $\log(|C_{\bar{T}}|) - \log(|C_{\bar{T}\cap\bar{S}}|) < k/2$. If we let s_i denote $(\log(|C_{\bar{S_1}\cap\dots\cap\bar{S_{i-1}^-}}|) - \log(|C_{\bar{S_1}\cap\dots\cap\bar{S}_i}|)) - (\log(|C_{\bar{S_1}\cap\dots\cap\bar{S_{i-1}^-}\cap\bar{T}}|) - \log(|C_{\bar{S_1}\cap\dots\cap\bar{S}_i\cap\bar{T}}|))$ then it must be the case that $|S_i/T| \leq (k_i - s_i)d$. In particular,

$$|S/T| = \sum_{i=1}^{\ell} |S_i/T| \leq \sum_{i=1}^{\ell} (k_i - s_i)d = kd - d\sum_{i=1}^{\ell} s_i \leq kd/2,$$

because $\sum_{i=1}^{\ell} s_i \geq k/2$ (half of the log of the number of distinct codewords contributed by these spaces has been recovered). Thus, the number of remaining rows that have to be recovered is bounded by $kd/2$.

In particular, we now repeat this $\log(n) + \log(q)$ times. In the ith iteration, we are guaranteed that one of the following happens:

1. The sum of the log of the number of distinct codewords in the spaces in the subspaces $C'_1, \dots C'_\ell$ is at least $k/2^i$. If this is the case, then we will recover all of the rows in the support. This is because by induction, after the $i-1$st iteration, there will be at most $(kd/2^{i-1})$ rows remaining in S to be recovered. If at the end of the ith iteration, the sum of the logs of the number of distinct codewords of $C'_1, \dots C'_\ell$ is at least $k/2^i$, then this means we will have recovered $2kd/2^i = kd/2^{i-1}$ new rows from S in the ith iteration, which will be the entirety of the remaining rows.

2. The remaining the sum of the logs of the number of distinct codewords in the spaces $C'_1, \dots C'_\ell$ is $< k/2^i$. In this case, by the same argument as above, the remaining number of rows that must be recovered from S is bounded by $kd/2^i$.

After $\log(n) + \log(q)$ iterations, the remaining number of distinct codewords contributed by rows (not already recovered) in S goes to 0, and therefore we must have recovered all of S.

This yields the desired claim. □

Proof of Theorem 4.6. The set S exists because of Corollary 4.5. The efficient algorithm for recovering the set T of the given size follows from Claim 4.9. □

5 Sparsifying Codes Over $\mathbb{Z}_q$

In this section we use the decomposition theorem of the previous section to derive a proof of Theorem 1.7 for the special case of the group $\mathbb{Z}_q$ and a dependence on $\log^2(q)$. Specifically, given an arbitrary (possibly weighted) generating matrix $G \in \mathbb{Z}_q^{m\times n}$, our goal is to return a sparsifier for this generating matrix. Note that our codeword counting bounds as stated hold only for *unweighted* codes. Thus, our first stepn is generalizing the weight-class decomposition technique from [KPS24] to the non-field setting.

5.1 Weighted Decomposition

We give an algorithm to show that given a weighted code, decomposes the code cleanly into weight classes. To this end, we suggest the following procedure upon being given a code of length m and at most q^n distinct codewords in Algorithm 5.

Next, we prove some facts about this algorithm.

Algorithm 5: WeightClassDecomposition$(\mathcal{C}, \epsilon, \alpha)$

Let E_i be all coordinates of $\mathcal{C}'$ that have weight between $[\alpha^{i-1}, \alpha^i]$.
Let $\mathcal{D}_{\text{odd}} = E_1 \cup E_3 \cup E_5 \cup \ldots$, and let $\mathcal{D}_{\text{even}} = E_2 \cup E_4 \cup E_6 \cup \ldots$.
return $\mathcal{D}_{odd}, \mathcal{D}_{even}$.

Lemma 5.1. *Consider a code $\mathcal{C}$ with at most q^n distinct codewords and length m. Let*

$$\mathcal{D}_{odd}, \mathcal{D}_{even} = \text{WeightClassDecomposition}(\mathcal{C}, \epsilon, (m/\epsilon)^3).$$

To get a $(1 \pm \epsilon)$-sparsifier for $\mathcal{C}$, it suffices to get a $(1 \pm \epsilon)$ sparsifier to each of $\mathcal{D}_{odd}, \mathcal{D}_{even}$.

Proof. First, note that the creation of $\mathcal{D}_{\text{odd}}, \mathcal{D}_{\text{even}}$ forms a *vertical* decomposition of the code $\mathcal{C}'$. Thus, by Claim 3.1, if we have a $(1 \pm \epsilon)$ sparsifier for each of $\mathcal{D}_{\text{odd}}, \mathcal{D}_{\text{even}}$, we have a $(1 \pm \epsilon)$ sparsifier to $\mathcal{C}$. □

Because of the previous claim, it is now our goal to create sparsifiers for $\mathcal{D}_{\text{odd}}, \mathcal{D}_{\text{even}}$. Without loss of generality, we will focus our attention only on $\mathcal{D}_{\text{even}}$, as the procedure for $\mathcal{D}_{\text{odd}}$ is exactly the same (and the proofs will be the same as well). At a high level, we will take advantage of the fact that

$$\mathcal{D}_{\text{even}} = E_2 \cup E_4 \cup \ldots,$$

where each E_i contains coordinates (i.e. rows of the generating matrix) with weights in the range $[\alpha^{i-1}, \alpha^i]$ where we have now set $\alpha = (m/\epsilon)^3$. Because the code $\mathcal{C}$ is of length m we know that the length of $\mathcal{D}_{\text{even}}$ is also $\leq m$. Thus, whenever a codeword $c \in \mathcal{D}_{\text{even}}$ is non-zero in a coordinate in E_i, we can effectively ignore all coordinates of lighter weights $E_{i-2}, E_{i-4}, \ldots$. This is because any coordinate in $E_{\leq i-2}$ has weight which is at most a $\frac{\epsilon^3}{m^3}$ fraction of any single coordinate in E_i. Because there are at most m coordinates in $\mathcal{D}_{\text{even}}$, it follows that the total possible weight of all rows in $E_{\leq i-2}$ is still at most a $O(\epsilon/m)$ fraction of the weight of a single coordinate in E_i. Thus, we will argue that when we are creating a sparsifier for codewords that are non-zero in a row in E_i, we will be able to effectively ignore all rows corresponding to $E_{\leq i-2}$. Thus, our decomposition is quite simple: we first restrict our attention to E_i and create a $(1 \pm \epsilon)$ sparsifier for these rows. Then, we transform the remaining code such that only codewords which are all zeros on E_i remain. We present this transformation below:

Algorithm 6: SingleSpanDecomposition$(\mathcal{D}_{\text{even}}, \alpha, i)$

Let E_i be all rows of $\mathcal{D}_{\text{even}}$ with weights between α^{i-1} and α^i.
Let G be a generating matrix for $\mathcal{D}_{\text{even}}$.
Store $G|_{E_i}$.
Let $G' = G$.
while $G'|_{E_i}$ *is not all zero* **do**
  Find the first non-zero coordinate of $G'|_{E_i}$, call this j.
  Set $G' = \text{Contract}(G', j)$.
end
return $G|_{E_i}$, $G'|_{\bar{E}_i}$

Claim 5.2. *If the span of G originally had $2^{n'}$ distinct codewords, and the span of $G|_{E_i}$ has $2^{n''}$ distinct codewords, then after Algorithm 6, the span of $G'|_{\bar{E}_i}$ has $2^{n'-n''}$ distinct codewords.*

Proof. After running the above algorithm, G' is entirely 0 on the rows corresponding to E_i, hence it follows that after running the algorithm, G' and $G'|_{\bar{E}_i}$ have the same number of distinct codewords. Now, we will argue that the span of G' has at most $2^{n'-n''}$ distinct codewords. Indeed, for any codeword c in the span of G', c is also in the span of the original G. However, for this same c, in the original G we could add any of the $2^{n'}$ distinct vectors which are non-zero on the rows corresponding to E_i. Thus, the span of G must have at least $2^{n'}$ times as many distinct codewords as G'. This concludes the claim. □

Claim 5.3. *For any codeword c in the span of G, if c is zero in the coordinates corresponding to E_i, then c is still in the span of G' after the contractions of Algorithm 6.*

Proof. This follows from Claim 4.1. If a codeword is 0 in a coordinate which we contract on, then it remains in the span. Hence, if we denote by c a codeword which is zero in all of the coordinates of E_i, then c is still in the span after contracting on the coordinates in E_i. □

Claim 5.4. *In order to get a $(1 \pm \epsilon)$ approximation to $\mathcal{D}_{even}$, it suffices to combine a $(1 \pm \epsilon/2)$ approximation to $G|_{E_i}$ and a $(1 \pm \epsilon)$ approximation to $G'|_{\bar{E}_i}$.*

Proof. For any codeword $c \in \mathcal{D}_{\text{even}}$ which is non-zero on the coordinates E_i, it suffices to get a $(1 \pm \epsilon/2)$ approximation to their weight on $G|_{E_i}$, as this makes up at least a $(1 \pm \epsilon/m)$ fraction of the overall weight of the codeword.

For any codeword $c \in \mathcal{D}_{\text{even}}$ which is zero on rows E_i, then c is still in the span of G', and in particular, its weight when generated by G is exactly the same as its weight in $G'_{\bar{E}_i}$ (as it is zero in the coordinates corresponding to E_i, we can ignore these coordinates). Hence, it suffices to get a $(1 \pm \epsilon)$ approximation to the weight of c on $G'_{\bar{E}_i}$. Taking the union of these two sparsifiers will then yield a sparsifier for every codeword in the span of $\mathcal{D}_{\text{even}}$. □

This then yields the decomposition we will use to construct sparsifiers:

Algorithm 7: SpanDecomposition($\mathcal{D}_{\text{even}}, \alpha$)

1 Let $\mathcal{D}'_{\text{even}} = \mathcal{D}_{\text{even}} = E_2 \cup E_4 \cup E_6 \ldots$.
2 Let $S = \{\}$.
3 **while** $\mathcal{D}'_{even}$ *is not empty* **do**
4 Let i be the largest integer such that E_i is non-empty in $\mathcal{D}'_{\text{even}}$.
5 Let $G|_{E_i}, G'|_{\bar{E}_i} = \text{SingleSpanDecomposition}(\mathcal{D}'_{\text{even}}, \alpha, i)$.
6 Let $\mathcal{D}'_{\text{even}}$ be the span of $G'|_{\bar{E}_i}$, and let $H_i = G|_{E_i}$.
7 Add i to S.
8 **end**
9 **return** S, H_i *for every* $i \in S$

Claim 5.5. *Let S, H_i be as returned by Algorithm 7. Then, $\sum_{i \in S} \log(|Span(H_i)|) = \log(|Span(\mathcal{D}_{even})|)$.*

Proof. This follows because from line 5 of Algorithm 7. In each iteration, we store $G|_{E_i}$, and iterate on $G'|_{E_i}$. From Claim 5.2, we know that

$$\text{(number of distinct codewords in } G|_{E_i}) \cdot \text{(number of distinct codewords in } G'|_{\bar{E}_i})$$
$$= \text{(number of distinct codewords in } G),$$

thus taking the log of both sides, we can see that the sum of the logs of the number of distinct codewords is preserved. □

Lemma 5.6. *Suppose we have a code of the form $\mathcal{D}_{even}$ created by Algorithm 5. Then, for $S, (H_i)_{i \in S} = \text{SpanDecomposition}(\mathcal{D}_{even})$, it suffices to get a $(1 \pm \epsilon/2)$ sparsifier for each of the H_i in order to get a $(1 \pm \epsilon)$ sparsifier for $\mathcal{D}_{even}$.*

Proof. This follows by inductively applying Claim 5.4. Let our inductive hypothesis be that getting a $(1 \pm \epsilon/2)$ sparsifier to each of codes returned of Algorithm 7 suffices to get a $(1 \pm \epsilon)$ sparsifier to the code overall. We will induct on the number of recursive levels that Algorithm 7 undergoes (i.e., the number of distinct codes returned by the algorithm). In the base case, we assume that there is only one level of recursion, and that Algorithm 7 simply returns a single code. Clearly then, getting a $(1 \pm \epsilon/2)$ sparsifier to this code suffices to sparsify the code overall.

Now, we prove the claim inductively. Assume the algorithm returns ℓ codes. After the first iteration, we decompose $\mathcal{D}_{\text{even}}$ into $H_i = G|_{E_i}$ and $G'|_{\bar{E}_i}$. By Claim 5.4, it suffices to get a $(1 \pm \epsilon/2)$ sparsifier to H_i, while maintaining a $(1 \pm \epsilon)$ sparsifier to $G'|_{\bar{E}_i}$. By invoking our inductive claim, it then suffices to get a $(1 \pm \epsilon/2)$ sparsifier for the $\ell - 1$ codes returned by the algorithm on $G'|_{\bar{E}_i}$. Thus, we have proved our claim. □

5.2 Dealing with Bounded Weights

From the previous section, we know that for a code of length m, we can decompose the code into disjoint sections, where each section has weights bounded in the range $[\alpha^i, \alpha^{i+1}]$. In this section, we will show how we can sparsify these codes with weights in a bounded range.

Let us consider any H_i that is returned by Algorithm 7, when called with parameter α. By construction, H_i will contain weights only in the range $[\alpha^{i-1}, \alpha^i]$ and will have at most $O(m)$ coordinates. In this subsection, we will show how we can turn H_i into an unweighted code with at most $O(m \cdot \alpha/\epsilon)$ coordinates by essentially repeating each coordinate j about $w(j)$ times (where $w(j)$ is the weight of the corresponding coordinate in H_i). First, note however, that we can simply pull out a factor of α^{i-1}, and treat the remaining graph as having weights in the range of $[1, \alpha]$. Because multiplicative approximation does not change under multiplication by a constant, this can be done without loss of generality. Formally, consider the following algorithm:

Algorithm 8: MakeUnweighted$(\mathcal{C}, \alpha, i, \epsilon)$

1 Divide all edge weights in $\mathcal{C}$ by α^{i-1}.
2 Make a new unweighted code $\mathcal{C}'$ by duplicating every coordinate j of $\mathcal{C}$ $\lfloor 10w(j)/\epsilon \rfloor$ times.
3 **return** $\mathcal{C}', \alpha^{i-1} \cdot \frac{\epsilon}{10}$

Lemma 5.7. *Consider a code $\mathcal{C}$ with weights bounded in the range $[\alpha^{i-1}, \alpha^i]$. To get a $(1 \pm \epsilon)$ sparsifier for $\mathcal{C}$ it suffices to return a $(1 \pm \epsilon/10)$ sparsifier for $\mathcal{C}' =$ MakeUnweighted$(\mathcal{C}, \alpha, i, \epsilon)$ weighted by $\alpha^{i-1} \cdot \frac{\epsilon}{10}$.*

Proof. It suffices to show that $\mathcal{C}'$ is $(1 \pm \epsilon/10)$ sparsifier for $\mathcal{C}$, as our current claim will then follow by Claim 3.2 (composing approximations). Now, to show that $\mathcal{C}'$ is $(1 \pm \epsilon/10)$ sparsifier for $\mathcal{C}$, we will use Claim 3.1 (vertical decomposition of a code), and show that in fact the weight contributed by every coordinate in $\mathcal{C}$ is approximately preserved by the copies of the coordinate introduced in $\mathcal{C}'$.

Without loss of generality, let us assume that $i = 1$, as otherwise pulling out the factor of α^{i-1} in the weights clearly preserves the weights of the codewords. Indeed, for every coordinate j in $\mathcal{C}$, let $w(j)$ be the corresponding weight on this coordinate, and consider the corresponding $\lfloor 10w(j)/\epsilon \rfloor$

coordinates in $\mathcal{C}'$. We will show that the contribution from these coordinates in $\mathcal{C}'$, when weighted by $\epsilon/10$, is a $(1 \pm \epsilon/10)$ approximation to the contribution from j.

So, consider an arbitrary coordinate j. Then,

$$\frac{10w}{\epsilon} - 1 \leq \lfloor 10w(j)/\epsilon \rfloor \leq \frac{10w}{\epsilon}.$$

When we normalize by $\frac{\epsilon}{10}$, we get that the combined weight of the new coordinates w' satisfies

$$w(j) - \epsilon/10 \leq w'(j) \leq w(j).$$

Because $w \geq 1$, it follows that this yields a $(1 \pm \epsilon/10)$ sparsifier, and we can conclude our statement. □

Claim 5.8. *Suppose a code $\mathcal{C}$ of length m has weight ratio bounded by α, and minimum weight α^{i-1}. Then, MakeUnweighted$(\mathcal{C}, \alpha, i, \epsilon)$ yields a new unweighted code of length $O(m\alpha/\epsilon)$.*

Proof. Each coordinate is repeated at most $O(\alpha/\epsilon)$ times. □

5.3 Sparsifiers for Codes of Polynomial Length

In this section, we introduce an efficient algorithm for sparsifying codes. We will take advantage of the decomposition proved in Corollary 4.5 in conjunction with the following claim:

Claim 5.9. *Suppose $\mathcal{C}$ is a code with at most q^n distinct codewords over $\mathbb{Z}_q$, and let $b \geq 1$ be an integer such that for any integer $\alpha \geq 1$, the number of codewords of weight $\leq \alpha b$ is at most $\binom{n\log(q)}{\alpha} \cdot q^{\alpha+1} \leq (qn)^{2\alpha}$. Suppose further that the minimum distance of the code $\mathcal{C}$ is b. Then, sampling the ith coordinate of $\mathcal{C}$ at rate $p_i \geq \frac{\log(n)\log(q)\eta}{b\epsilon^2}$ with weights $1/p_i$ yields a $(1 \pm \epsilon)$ sparsifier with probability $1 - 2^{-(0.19\eta - 110)\log n} \cdot n^{-101}$.*

Proof. Consider any codeword c of weight $[\alpha b/2, \alpha b]$ in $\mathcal{C}$. We know that there are at most $(qn)^\alpha$ codewords that have weight in this range. The probability that our sampling procedure fails to preserve the weight of c up to a $(1 \pm \epsilon)$ fraction can be bounded by Claim 3.3. Indeed,

$$\Pr[\text{fail to preserve weight of } c] \leq 2e^{-0.38 \cdot \epsilon^2 \cdot \frac{\alpha b}{2} \cdot \frac{\eta \log(n)\log(q)}{\epsilon^2 b}} = 2e^{-0.19\alpha\eta\log(n)\log(q)}.$$

Now, let us take a union bound over the at most $(qn)^{2\alpha}$ codewords of weight between $[\alpha b/2, \alpha]$. Indeed,

$$\begin{aligned}
\Pr[\text{fail to preserve any } c \text{ of weight } [\alpha b/2, \alpha b]] &\leq 2^{2\alpha\log(qn)} \cdot 2e^{-0.19\alpha\eta\log(n)\log(q)} \\
&\leq 2^{\alpha\cdot(-0.19\eta+2)\log(n)\log(q)} \\
&\leq 2^{\alpha\cdot(-0.19\eta+2)\log(n)} \\
&\leq 2^{-(0.19\eta-110)\alpha\log n} \cdot 2^{-108\alpha\log n} \\
&\leq 2^{-(0.19\eta-110)\log n} \cdot n^{-108\alpha},
\end{aligned}$$

where we have chosen η to be sufficiently large. Now, by integrating over $\alpha \geq 1$, we can bound the failure probability for any integer choice of α by $2^{-(0.19\eta-110)\log n} \cdot n^{-101}$. □

Next, we consider Algorithm 9:

Algorithm 9: CodeDecomposition($\mathcal{C}, d$)

Let T be $\cup_i T_i$ for T_i the sets of coordinates returned by ConstructSpanningSubsets($\mathcal{C}, 2d(\log(n) + \log(q))$).
Let $\mathcal{C}'$ be the code $\mathcal{C}$ after removing the set of coordinates T.
return $T, \mathcal{C}'$

Intuitively, the set T returned by Algorithm 9 contains all of the "bad" rows which were causing the violation of the codeword counting bound. We know that if we removed *only* the true set of bad rows, denoted by S, then we could afford to simply sample the rest of the code at rate roughly $1/d$ while preserving the weights of all codewords. Thus, it remains to show that when we remove T (a superset of S) that this property still holds. More specifically, we will consider the following algorithm:

Algorithm 10: CodeSparsify($\mathcal{C} \subseteq \mathbb{Z}_q^m, n, \epsilon, \eta$)

Let m be the length of $\mathcal{C}$.
if $m \leq 100 \cdot n \cdot \eta \log^2(n) \log^2(q)/\epsilon^2$ **then**
  return $\mathcal{C}$
end
Let $d = \frac{m\epsilon^2}{2\eta \cdot n \log^2(n) \log^2(q)}$.
Let $T, \mathcal{C}' = \text{CodeDecomposition}(\mathcal{C}, \sqrt{d} \cdot \eta \cdot \log(n) \log(q)/\epsilon^2)$. Let $\mathcal{C}_1 = \mathcal{C}|_T$. Let $\mathcal{C}_2$ be the result of sampling every coordinate of $\mathcal{C}'$ at rate $1/\sqrt{d}$.
return CodeSparsify($\mathcal{C}_1, n, \epsilon, \eta$) $\cup \sqrt{d} \cdot$ CodeSparsify($\mathcal{C}_2, n, \epsilon, \eta$)

Lemma 5.10. *In Algorithm 10, starting with a code $\mathcal{C}$ of size $2dn \log^2(n) \log^2(q)/\epsilon^2$, after i levels of recursion, with probability $1 - 2^i \cdot 2^{-\eta n}$, the code being sparsified at level i, $\mathcal{C}^{(i)}$ has at most*

$$2(1 + 1/2 \log\log(n))^i \cdot d^{1/2^i} \cdot \eta \cdot n \log^2(n) \log^2(q)/\epsilon^2$$

surviving coordinates.

Proof. Let us prove the claim inductively. For the base case, note that in the 0th level of recursion the number of surviving coordinates in $\mathcal{C}^{(0)} = \mathcal{C}$ is $d \cdot 2n \log^2(n) \log^2(q)/\epsilon^2$, so the claim is satisfied trivially.

Now, suppose the claim holds inductively. Let $\mathcal{C}^{(i)}$ denote a code that we encounter in the ith level of recursion, and suppose that it has at most

$$2(1 + 1/2 \log\log(n))^i \cdot d^{1/2^i} \cdot \eta \cdot n \log^2(n) \log^2(q)/\epsilon^2$$

coordinates. Denote this number of coordinates by ℓ. Now, if this number is smaller than $100n\eta \log^2(n) \log^2(q)/\epsilon^2$, we will simply return this code, and there will be no more levels of recursion, so our claim holds vacuously. Instead, suppose that this number is larger than $100n\eta \log^2(n) \log^2(q)/\epsilon^2$, and let $d' = \frac{\ell\epsilon^2}{2\eta n \log^2(n) \log^2(q)} \leq (1 + 1/2 \log\log(n))^i \cdot d^{1/2^i}$.

Then, we decompose $\mathcal{C}^{(i)}$ into two codes, $\mathcal{C}_1$ and $\mathcal{C}_2$. $\mathcal{C}_1$ is the restriction of $\mathcal{C}$ to the set of disjoint maximum spanning subsets. By construction, we know that T is constructed by calling ConstructSpanningSubsets with parameter $\sqrt{d'}\eta \log(n) \log(q)/\epsilon^2$, and therefore

$$|T| \leq 2\sqrt{d'} n\eta \log^2(n) \log^2(q)/\epsilon^2 \leq 2(1 + 1/2 \log\log(n))^i n\eta d^{1/2^{i+1}} \log^2(n) \log^2(q)/\epsilon^2,$$

satisfying the inductive claim.

For $\mathcal{C}_2$, we define random variables $X_1 \dots X_\ell$ for each coordinate in the support of $\mathcal{C}_2$. X_i will take value 1 if we sample coordinate i, and it will take 0 otherwise. Let $X = \sum_{i=1}^{\ell} X_i$, and let $\mu = \mathbb{E}[X]$. Note that

$$\frac{\mu^2}{\ell} = \left(\frac{\ell}{\sqrt{d'}}\right)^2 / \ell = \frac{\ell}{d'} \geq \eta \cdot n \cdot \log^2(n) \log^2(q)/\epsilon^2.$$

Now, using Chernoff,

$$\Pr[X \geq (1 + 1/2 \log\log(n))\mu] \leq e^{\frac{-2}{4 \log^2 \log(n)} \cdot \eta \cdot n \cdot \log(n) \log(q)/\epsilon^2} \leq 2^{-\eta n},$$

as we desire. Since $\mu = \ell/\sqrt{d'} \leq (1 + 1/2 \log\log(n))^i \cdot d^{1/2^{i+1}} \cdot \eta \cdot n \log^2(n) \log^2(q)/\epsilon^2$, we conclude our result.

Now, to get our probability bound, we also operate inductively. Suppose that up to recursive level $i-1$, all sub-codes have been successfully sparsified to their desired size. At the ith level of recursion, there are at most 2^{i-1} codes which are being probabilistically sparsified. Each of these does not exceed its expected size by more than the prescribed amount with probability at most $2^{-\eta n}$. Hence, the probability all codes will be successfully sparsified up to and including the ith level of recursion is at least $1 - 2^{i-1}2^{-\eta n} - 2^{i-1}2^{-\eta n} = 1 - 2^i 2^{-\eta n}$. □

Lemma 5.11. *For any iteration of Algorithm 10 called on a code $\mathcal{C}$, $\mathcal{C}_1 \cup \sqrt{d} \cdot \mathcal{C}_2$ is a $(1 \pm \epsilon)$ sparsifier to $\mathcal{C}$ with probability at least $1 - 2^{-(0.19\eta - 110)\log n} \cdot n^{-101}$.*

Proof. First, let us note that the set T returned from Algorithm 9 is a superset of the bad set S specified in Corollary 4.5 (this follows from Claim 4.9). Thus, we can equivalently view the procedure as producing three codes: $\mathcal{C}|_S, \mathcal{C}|_{T/S}$ and $\mathcal{C}_{\bar{T}} = \mathcal{C}'$. For our analysis, we will view this procedure in a slightly different light: we will imagine that first the algorithm removes exactly the bad set of rows S, yielding $\mathcal{C}|_S$ and $\mathcal{C}_{\bar{S}}$. Now, for this second code, $\mathcal{C}_{\bar{S}}$, we know the code-word counting bound will hold, and in particular, random sampling procedure will preserve codeword weights with high probability. However, our procedure is *not* uniformly sampling the coordinates in $\mathcal{C}_{\bar{S}}$, because some of these coordinates are in T/S, and thus are preserved exactly (i.e. with probability 1). For this, we will take advantage of the fact that preserving coordinates with probability 1 is *strictly better* than sampling at any rate < 1. Thus, we will still be able to argue that the ultimate result $\mathcal{C}_1 \cup \sqrt{d} \cdot \mathcal{C}_2$ is a $(1 \pm \epsilon)$ sparsifier to $\mathcal{C}$ with high probability.

As mentioned above, we start by noting that $\mathcal{C}', \mathcal{C}|_S, \mathcal{C}_{T/S}$ form a *vertical* decomposition of $\mathcal{C}$. $\mathcal{C}|_S$ is preserved exactly, so we do not need to argue concentration of the codewords on these coordinates. Hence, it suffices to show that $\sqrt{d} \cdot \mathcal{C}_1 \cup \mathcal{C}_{T/S}$ is a $(1 \pm \epsilon)$-sparsifier to $\mathcal{C}' \cup \mathcal{C}_{T/S}$.

To see that $\sqrt{d} \cdot \mathcal{C}_1 \cup \mathcal{C}_{T/S}$ is a $(1 \pm \epsilon)$-sparsifier to $\mathcal{C}' \cup \mathcal{C}_{T/S}$, first note that every codeword in $\mathcal{C}' \cup \mathcal{C}_{T/S}$ is of weight at least $\sqrt{d} \cdot \eta \cdot \log(n) \log(q)/\epsilon^2$. This is because if there were a codeword of weight smaller than this, there would exist a subcode of $\mathcal{C}' \cup \mathcal{C}_{T/S}$ with 2 distinct codewords, and support bounded by $\sqrt{d} \cdot \eta \cdot \log(n) \log(q)/\epsilon^2$. But, because we have removed the set S of bad rows, we know that there can be no such sub-code remaining in $\mathcal{C}' \cup \mathcal{C}_{T/S}$. Thus, every codeword in $\mathcal{C}' \cup \mathcal{C}_{T/S}$ is of weight at least $\sqrt{d} \cdot \eta \cdot \log(n) \log(q)/\epsilon^2$.

Now, we can invoke Claim 5.9 with $b = \sqrt{d}\eta \log(n) \log(q)/\epsilon^2$. Note that the hypothesis of Claim 5.9 is satisfied by virtue of our code decomposition. Indeed, we removed coordinates of the code such that in the resulting $\mathcal{C}' \cup \mathcal{C}_{T/S}$, for any $\alpha \geq 1$, there are at most $(qn)^{2\alpha}$ codewords of weight $\leq \alpha\sqrt{d}\eta \log(n) \log(q)/\epsilon^2$. Using the concentration bound of Claim 5.9 yields that with probability

at least $1 - 2^{-(0.19\eta - 110)\log n} \cdot n^{-101}$, when samplin every coordinate at rate $\geq 1/\sqrt{d}$ the resulting sparsifier for $\mathcal{C}' \cup \mathcal{C}_{T/S}$ is a $(1 \pm \epsilon)$ sparsifier, as we desire. Note that we are using the fact that every coordinate is sampled with probability $\geq 1/\sqrt{d}$ (in particular, those in $T - S$ are sampled with probability 1). □

Corollary 5.12. *If Algorithm 10 achieves maximum recursion depth ℓ when called on a code $\mathcal{C}$, and $\eta > 600$, then the result of the algorithm is a $(1 \pm \epsilon)^\ell$ sparsifier to $\mathcal{C}$ with probability $\geq 1 - (2^\ell - 1) \cdot 2^{-(0.19\eta - 110)\log n} \cdot n^{-101}$*

Proof. We prove the claim inductively. Clearly, if the maximum recursion depth reached by the algorithm is 0, then we have simply returned the code itself. This is by definition a $(1 \pm \epsilon)^0$ sparsifier to itself.

Now, suppose the claim holds for maximum recursion depth $i - 1$. We will show it holds for maximum recursion depth i. Let the code we are sparsifying be $\mathcal{C}$. We break this into $\mathcal{C}_1$, $\mathcal{C}'$, and sparsify these. By our inductive claim, with probability $1 - (2^{i-1} - 1) \cdot 2^{-(0.19\eta - 110)\log n} \cdot n^{-101}$ each of the sparsifiers for $\mathcal{C}_1, \mathcal{C}'$ are $(1 \pm \epsilon)^{i-1}$ sparsifiers. Now, by Lemma 5.11 and our value of η, $\mathcal{C}_1, \mathcal{C}'$ themselves together form a $(1 \pm \epsilon)$ sparsifier for $\mathcal{C}$ with probability $1 - 2^{-(0.19\eta - 110)\log n} \cdot n^{-101}$. So, by using Claim 3.2, we can conclude that with probability $1 - (2^i - 1) \cdot 2^{-(0.19\eta - 110)\log n} \cdot n^{-101}$, the result of sparsifying $\mathcal{C}_1, \mathcal{C}'$ forms a $(1 \pm \epsilon)^i$ approximation to $\mathcal{C}$, as we desire. □

We can then state the main theorem from this section:

Theorem 5.13. *For a code $\mathcal{C}$ over $\mathbb{Z}_q$ with at most q^n distinct codewords, and length m, Algorithm 10 creates a $(1 \pm \epsilon)$ sparsifier for $\mathcal{C}$ with probability $1 - \log(m) \cdot 2^{-(0.19\eta - 110)\log n} \cdot n^{-100}$ with at most*

$$O(n\eta \log(n) \log^2(q) \log^2(m) (\log\log(m))^2 / \epsilon^2)$$

coordinates.

Proof. For a code of with q^n distinct codewords, and length m, this means that our value of d as specified in the first call to Algorithm 10 is at most m as well. As a result, after only $\log\log m$ iterations, $d = m^{1/2^{\log\log m}} = m^{1/\log m} = O(1)$. So, by Corollary 5.12, because the maximum recursion depth is only $\log\log m$, it follows that with probability at least $1 - (2^{\log\log m} - 1) \cdot 2^{-(0.19\eta - 110)\log n} \cdot n^{-101}$, the returned result from Algorithm 10 is a $(1 \pm \epsilon)^{\log\log m}$ sparsifier for $\mathcal{C}$.

Now, by Lemma 5.10, with probability $\geq 1 - 2^{\log\log m} \cdot 2^{-\eta n} \geq 1 - \log(m) \cdot 2^{-(0.19\eta - 110)\log n} \cdot 2^{-n}$, every code at recursive depth $\log\log m$ has at most

$$(1 + 1/2\log\log(n))^{\log\log m} \cdot m^{1/\log m} \cdot \eta \cdot n \log(n) \log(q)/\epsilon^2 = O(n\eta \log(n) \log^2(q) \cdot e^{\frac{\log\log m}{\log\log n}}/\epsilon^2)$$

coordinates. Because the ultimate result from calling our sparsification procedure is the *union* of all of the leaves of the recursive tree, the returned result has size at most

$$\log(m) \cdot e^{\frac{\log\log m}{\log\log n}} \cdot O(n\eta \log(n) \log^2(q)/\epsilon^2) = O(n\eta \log(n) \log^2(q) \log^2(m)/\epsilon^2),$$

with probability at least $1 - \log(m) \cdot 2^{-(0.19\eta - 110)\log n} \cdot n^{-101}$.

Finally, note that we can replace ϵ with a value $\epsilon' = \epsilon/2\log\log m$. Thus, the resulting sparsifier will be a $(1 \pm \epsilon')^{\log\log m} \leq (1 \pm \epsilon)$ sparsifier, with the same high probability.

Taking the union bound of our errors, we can conclude that with probability $1 - \log(m) \cdot 2^{-(0.19\eta - 110)\log n} \cdot n^{-100}$, Algorithm 10 returns a $(1 \pm \epsilon)$ sparsifier for $\mathcal{C}$ that has at most

$$O(n\eta \log(n) \log^2(q) \log^2(m) (\log\log(m))^2 / \epsilon^2)$$

coordinates. □

However, as we will address in the next subsection, this result is not perfect:

1. For large enough m, there is no guarantee that this probability is ≥ 0 unless η depends on m.
2. For large enough m, $\log^2(m)$ may even be larger than n.

5.4 Final Algorithm

Finally, we state our final algorithm in Algorithm 11, which will create a $(1 \pm \epsilon)$ sparsifier for any code $\mathcal{C} \subseteq \mathbb{Z}_q^m$ with $\leq q^n$ distinct codewords preserving only $\widetilde{O}(n \log^2(q)/\epsilon^2)$ coordinates. Roughly speaking, we start with a weighted code of arbitrary length, use the weight class decomposition technique, sparsify the decomposed pieces of the code, and then repeat this procedure now that the code will have a polynomial length. Ultimately, this will lead to the near-linear size complexity that we desire. We write a single iteration of this procedure below:

Algorithm 11: FinalCodeSparsify($\mathcal{C}, \epsilon$)

1 Let n be $\log_q(|\mathcal{C}|)$.
2 Let m be the length of the code.
3 Let $\alpha = (m/\epsilon)^3$, and $\mathcal{D}_{\text{odd}}, \mathcal{D}_{\text{even}}$ =WeightClassDecomposition($\mathcal{C}, \epsilon, \alpha$).
4 Let $S_{\text{even}}, \{H_{\text{even},i}\}$ =SpanDecomposition($\mathcal{D}_{\text{even}}, \alpha$).
5 Let $S_{\text{odd}}, \{H_{\text{odd},i}\}$ =SpanDecomposition($\mathcal{D}_{\text{odd}}, \alpha$).
6 **for** $i \in S_{even}$ **do**
7 Let $\widehat{H}_{\text{even},i}, w_{\text{even},i}$ = MakeUnweighted($H_{\text{even},i}, \alpha, i, \epsilon/8$).
8 Let $\widetilde{H}_{\text{even},i}$ =CodeSparsify($\widehat{H}_{\text{even},i}, \log_q(|\text{Span}(\widehat{H}_{\text{even},i})|), \epsilon/80, 100(\log(m/\epsilon)\log\log(q))^2$).
9 **end**
10 **for** $i \in S_{odd}$ **do**
11 Let $\widehat{H}_{\text{odd},i}, w_{\text{odd},i}$ = MakeUnweighted($H_{\text{odd},i}, \alpha, i, \epsilon/8$).
12 Let $\widetilde{H}_{\text{odd},i}$ =CodeSparsify($\widehat{H}_{\text{odd},i}, \log_q(|\text{Span}(\widehat{H}_{\text{odd},i})|), \epsilon/80, 100(\log(m/\epsilon)\log\log(q))^2$).
13 **end**
14 **return** $\bigcup_{i \in S_{even}} \left(w_{even,i} \cdot \widetilde{H}_{even,i}\right) \cup \bigcup_{i \in S_{odd}} \left(w_{odd,i} \cdot \widetilde{H}_{odd,i}\right)$

First, we analyze the space complexity. WLOG we will prove statements only with respect to $\mathcal{D}_{\text{even}}$, as the proofs will be identical for $\mathcal{D}_{\text{odd}}$.

Claim 5.14. *Suppose we are calling Algorithm 11 on a code $\mathcal{C}$ with q^n distinct codewords. Let $n_{even,i} = \log_q(Span((\widehat{H}_{even,i}))|)$ from each call to the for loop in line 5.*

For each call $\widetilde{H}_{even,i}$ =CodeSparsify($\widehat{H}_{even,i}, \log_q(|\text{Span}(\widehat{H}_{even,i})|), \epsilon/10, 100(\log(n/\epsilon)\log\log(q))^2$) in Algorithm 11, the resulting sparsifier has

$$O\left(n_{even,i} \log(n_{even,i}) \log^4(m/\epsilon) \log^2(q) (\log\log(m/\epsilon)\log\log(q))^2/\epsilon^2\right)$$

coordinates with probability at least $1 - \log(m/\epsilon) \cdot 2^{-\Omega(\log^2(m/\epsilon)(\log\log(q))^2)}$.

Proof. We use several facts. First, we use Theorem 5.13. Note that the m in the statement of Theorem 5.13 is actually a $\text{poly}(m/\epsilon)$ because $\alpha = m^3/\epsilon^3$, and we started with a weighted code of length $O(m)$. So, it follows that after using Algorithm 8, the support size is bounded by $O(m^4/\epsilon^3)$. We've also added the fact that η is no longer a constant, and instead carries $O((\log(m/\epsilon)\log\log(q))^2)$, and carried this through to the probability bound. □

Lemma 5.15. *In total, the combined number of coordinates over $i \in S_{even}$ of all of the $\widetilde{H}_{even,i}$ is at most $\widetilde{O}(n \log^4(m) \log^2(q)/\epsilon^2)$ with probability at least $1 - \log(m \log(q)/\epsilon) \cdot 2^{-\Omega(\log^2(m/\epsilon)(\log\log(q))^2)}$.*

Proof. First, we use Claim 5.5 to see that

$$\sum_{i \in S_{\text{even}}} \log_q(|\text{Span}(\widehat{H}_{\text{even},i})|) \leq n.$$

Thus, in total, the combined length (total number of coordinates preserved) of all the $\widetilde{H}_{\text{even},i}$ is

$$\begin{aligned}
&\sum_{i \in S_{\text{even}}} \text{number of coordinates in } \widehat{H}_{\text{even},i} \\
&\leq \sum_{i \in S_{\text{even}}} O\left(n_{\text{even},i} \log(n_{\text{even},i}) \log^4(m/\epsilon) \log^2(q) (\log\log(m/\epsilon) \log\log(q))^2/\epsilon^2\right) \\
&\leq \sum_{i \in S_{\text{even}}} (n_{\text{even},i}) \cdot \widetilde{O}\left(\log^4(m) \log^2(q)/\epsilon^2\right) \\
&= n \cdot \widetilde{O}\left(\log^4(m) \log^2(q)/\epsilon^2\right) \\
&= \widetilde{O}(n \log^4(m) \log^2(q)/\epsilon^2).
\end{aligned}$$

To see the probability bound, we simply take the union bound over all at most n distinct $\widetilde{H}_{\text{even},i}$, and invoke Claim 5.14. □

Now, we will prove that we also get a $(1 \pm \epsilon)$ sparsifier for $\mathcal{D}_{\text{even}}$ when we run Algorithm 11.

Lemma 5.16. *After combining the $\widehat{H}_{even,i}$ from Lines 5-8 in Algorithm 11, the result is a $(1 \pm \epsilon/4)$-sparsifier for $\mathcal{D}_{even}$ with probability at least $1 - \log(m \log(q)/\epsilon) \cdot 2^{-\Omega(\log^2(m/\epsilon)(\log\log(q))^2)}$.*

Proof. We use Lemma 5.6, which states that to sparsify $\mathcal{D}_{\text{even}}$ to a factor $(1 \pm \epsilon/4)$, it suffices to sparsify each of the $H_{\text{even},i}$ to a factor $(1 \pm \epsilon/8)$, and then combine the results.

Then, we use Lemma 5.7, which states that to sparsify any $H_{\text{even},i}$ to a factor $(1 \pm \epsilon/8)$, it suffices to sparsify $\widehat{H}_{\text{even},i}$ to a factor $(1 \pm \epsilon/80)$, where again, $\widehat{H}_{\text{even},i}$ is the result of calling Algorithm 8. Then, we must multiply $\widehat{H}_{\text{even},i}$ by a factor $\alpha^{i-1} \cdot \epsilon/10$.

Finally, the resulting code $\widehat{H}_{\text{even},i}$ is now an unweighted code, whose length is bounded by $O(m^4/\epsilon^3)$, with at most $q^n_{\text{even},i}$ distinct codewords. The accuracy of the sparsifier then follows from Theorem 5.13 called with parameter $\epsilon/80$.

The failure probability follows from noting that we take the union bound over at most $n \log(q)$ $H_{\text{even},i}$. By Theorem 5.13, our choice of η, and the bound on the length of the support being $O(m^4/\epsilon^3)$, the probability bound follows. □

For Theorem 5.13, the failure probability is characterized in terms of the number of distinct codewords of the code that is being sparsified. However, when we call Algorithm 10 as a sub-routine in Algorithm 11, we have no guarantee that the number of distinct codewords is $\omega(q)$. Indeed, it is certainly possible that the decomposition in H_i creates n different codes, each with q distinct codewords in their span. Then, choosing η to only be a constant, as stated in Theorem 5.13, the failure probability could be constant, and taking the union bound over n choices, we might not get anything meaningful. To amend this, instead of treating η as a constant in Algorithm 10, we set $\eta = 100(\log(m/\epsilon) \log\log(q))^2$, where now m is the length of the original code $\mathcal{C}$, *not* in the current code that is being sparsified H_i. With this modification, we can then attain our desired probability bounds.

Theorem 5.17. *For any code $\mathcal{C}$ with q^n distinct codewords and length m over $\mathbb{Z}_q$, Algorithm 11 returns a $(1 \pm \epsilon)$ sparsifier to $\mathcal{C}$ with $\widetilde{O}(n \log^4(m) \log^2(q)/\epsilon^2)$ coordinates with probability $\geq 1 - 2^{-\Omega((\log(m/\epsilon) \log\log(q))^2)}$.*

Proof. First, we use Lemma 5.1. This Lemma states that in order to get a $(1 \pm \epsilon)$ sparsifier to a code $\mathcal{C}$, it suffices to get a $(1 \pm \epsilon/4)$ sparsifier to each of $\mathcal{D}_{\text{even}}, \mathcal{D}_{\text{odd}}$, and then combine the results.

Then, we invoke Lemma 5.16 to conclude that with probability $\geq 1 - 2^{-\Omega((\log(m/\epsilon) \log\log(q))^2)}$, Algorithm 11 will produce $(1 \pm \epsilon/4)$ sparsifiers for $\mathcal{D}_{\text{even}}, \mathcal{D}_{\text{odd}}$.

Further, to argue the sparsity of the algorithm, we use Lemma 5.15. This states that with probability $\geq 1 - 2^{-\Omega((\log(m/\epsilon) \log\log(q))^2)}$, Algorithm 11 will produce code sparsifiers of size $\widetilde{O}(n \log^4(m) \log^2(q)/\epsilon^2)$ for $\mathcal{D}_{\text{even}}, \mathcal{D}_{\text{odd}}$.

Thus, in total, the failure probability is at most $2^{-\Omega((\log(m/\epsilon) \log\log(q))^2)}$, the total size of the returned code sparsifier is at most $\widetilde{O}(n \log^4(m) \log^2(q)/\epsilon^2)$, and the returned code is indeed a $(1 \pm \epsilon)$ sparsifier for $\mathcal{C}$, as we desire.

Note that the returned sparsifier may have some duplicate coordinates because of Algorithm 8. Even when counting duplicates of the same coordinate separately, the size of the sparsifier will be at most $\widetilde{O}(n \log^4(m) \log^2(q)/\epsilon^2)$. We can remove duplicates of coordinates by adding their weights to a single copy of the coordinate. □

Claim 5.18. *Running Algorithm 11 on a code of length m with parameter ϵ takes time poly$(mn \log(q)/\epsilon)$.*

Proof. Let us consider the constituent algorithms that are invoked during the execution of Algorithm 11. First, we consider weight class decomposition. This groups rows together by weight (which takes time $\widetilde{O}(\log(m))$). Next, we invoke SpanDecomposition, which then contracts on the rows in the largest weight class. Note that in the worst case, we perform $O(n \log(q))$ contractions, as each contraction reduces the number of codewords by a factor of ≥ 2. Further, each contraction takes time $O(mn \log(q))$ as the total number of rows is bounded by m, and there are at most $n \log(q)$ columns in the generating matrix. Thus, the total runtime of this step is $\widetilde{O}(mn^2 \log^2(q))$.

The next step is to invoke the algorithm MakeUnweighted. Because the value of α is m^3/ϵ^3, this takes time at most $O(m^4/\epsilon^3)$ to create the new code with this many rows.

Finally, we invoke CodeSparsify on a code of length $\leq m^4/\epsilon^3$ and with at most $q^{n_{\text{even},i}}$ distinct codewords. Note that there are polylog(m) nodes in the recursive tree that is built by CodeSparsify. Each such node requires removing the set T which is a set of $\leq \widetilde{O}(\sqrt{m^4/\epsilon^3}) = \widetilde{O}(m^2/\epsilon^{1.5})$ maximum spanning subsets. Constructing each such subset (by Claim 4.8) takes time at most $O((m^4\epsilon^3)^2 n \log(q))$. After this step, the subsequent random sampling is efficiently doable. Thus, the total runtime is bounded by poly$(mn \log(q)/\epsilon)$, as we desire.

□

Note that creating codes of linear-size now simply requires invoking Algorithm 11 two times (each with parameter $\epsilon/2$). Indeed, because the length of the code to begin with is $\leq q^n$, this means that after the first invocation, the resulting $(1 \pm \epsilon/2)$ sparsifier $\mathcal{C}'$ that we get maintains $\leq \widetilde{O}(n^5 \log^6(q)/\epsilon^2)$ coordinates. In the second invocation, we get a $(1 \pm \epsilon/2)$-sparsifier $\mathcal{C}''$ for $\mathcal{C}'$, whose length is bounded by $\widetilde{O}(n \log^4(n^5 \log^6(q)/\epsilon^2) \log^2(q)/\epsilon^2) = \widetilde{O}(n \log^2(q)/\epsilon^2)$, as we desire.

Formally, this algorithm can be written as:

Algorithm 12: Sparsify$(\mathcal{C}, \epsilon)$

```
C' = FinalCodeSparsify(C, ε/2).
return FinalCodeSparsify(C', ε/2)
```

Theorem 5.19. *Algorithm 12 returns a $(1\pm\epsilon)$-sparsifier to $\mathcal{C}$ of size $\widetilde{O}(n\log^2(q)/\epsilon^2)$ with probability $1-2^{-\Omega((\log(n/\epsilon)\log\log(q))^2)}$ in time poly$(mn\log(q)/\epsilon)$.*

Proof. Indeed, because the length of the code to begin with is $\leq q^n$, this means that after the first invocation, the resulting $(1\pm\epsilon/2)$ sparsifier $\mathcal{C}'$ that we get maintains $\leq \widetilde{O}(n^5\log^6(q)/\epsilon^2)$ coordinates. In the second invocation, we get a $(1\pm\epsilon/2)$-sparsifier $\mathcal{C}''$ for $\mathcal{C}'$, whose length is bounded by $\widetilde{O}(n\log^4(n^5\log^6(q)/\epsilon^2)\log^2(q)/\epsilon^2) = \widetilde{O}(n\log^2(q)/\epsilon^2)$, as we desire. To see the probability bounds, note that $m \geq n$, and thus both processes invocations of FinalCodeSparsify succeed with probability $1-2^{-\Omega((\log(n/\epsilon)\log\log(q))^2)}$.

Finally, to see that the algorithm is efficient, we simply invoke Claim 5.18 for each time we fun the algorithm. Thus, we get our desired bound. □

6 Sparsifying Affine Predicates over Abelian Groups

In this section, we will generalize the result from the previous section. Specifically, whereas the previous section showed that one can sparsify CSP systems where each constraint is of the form $\sum_{i=1}^r a_i b_i \neq a_0 \mod q$, for $a_i \in \mathbb{Z}_q, b_i \in \{0,1\}$, we will show that in fact, for any Abelian group A, we can sparsify predicates of the form $\sum_{i=1}^r a_i b_i \neq a_0 \mod q$, for $a_i \in A, b_i \in \{0,1\}$.

Concretely, we first define affine constraints.

Definition 6.1. *We say a constraint (or predicate) $P : \{0,1\}^r \to \{0,1\}$ is **affine** if it can be written as*

$$P(b_1,\dots b_r) = \mathbf{1}[\sum_{i=1}^r a_i b_i \neq a_0],$$

where $a_i \in A, b_i \in \{0,1\}$, and the addition is done over an Abelian group A.

Going forward, for an Abelian group A, we will let q be the number of elements in A. We will make use of the following fact:

Fact 6.1. [Pin10] Any finite Abelian group A is isomorphic to a direct product of the form

$$\mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_u},$$

where each q_j is a prime power. We will let $q = \prod_{i=1}^u q_i$.

Going forward, we will only work in accordance with this isomorphism. That is, instead of considering constraints of the form

$$\mathbf{1}[\sum_{i=1}^r a_i b_i \neq a_0],$$

where $a_i \in A, b_i \in \{0,1\}$, we will simply consider constraints of the same form only with $a_i \in \mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_u}$. We will view these elements as tuples $(d_1,\dots d_u)$, where $d_j \in \mathbb{Z}_{q_j}$.

Now, we can introduce the analog of the contraction algorithm in the Abelian case. We will let $G \in (\mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_u})^{m\times n}$ be a generating matrix, which generates a code $\mathcal{C} \subseteq (\mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_u})^m$. In particular, each entry in the matrix G will be from $(\mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_u})$. Being the generating matrix for $\mathcal{C}$ implies that $\mathcal{C} = \{Gx : x \in \mathbb{Z}^n\} = \{Gx : x \in \mathbb{Z}_q^n\}$ (i.e., all linear combinations of columns of G), where $q = q_1 \cdot \cdots \cdot q_u$.

For a codeword $c \in \mathcal{C}$, we will still say that $\mathrm{wt}(c) = |\{j \in [n] : c_j \neq 0\}|$, where $c_j = 0$ if the corresponding coordinate is the tuple $(0,0,0,\dots 0) \in \mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_u}$. As before, we say

$$\mathrm{Density}(\mathcal{C}) = \frac{\log_2(|\mathcal{C}|)}{|\mathrm{Supp}(\mathcal{C})|}.$$

Going forward, we will refer to the jth row of the generating matrix G. This is a row vector in $(\mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_u})^n$. Roughly speaking, our new contraction algorithm, when contracting on a specific coordinate j, will perform the contraction algorithm on each index of the tuples in row j. We will use the notation G_j to refer to the jth row of the generating matrix G, and we will use the notation $G_j^{(p)}$ to refer to the component of G_j corresponding to $\mathbb{Z}_{q_p}$. This will be a row vector in $\mathbb{Z}_{q_p}^n$. Note that we will be consistent in referring to the coordinate of a codeword c as the entry in $[m]$, while the the index of the jth coordinate will refer to a specific entry in the tuples of that coordinate.

Algorithm 13: ContractAbelian(G, j)

```
1 Let G_j be the jth row of the generating matrix G.
2 for p ∈ [u] do
3     if G_j^(p) is non-zero then
4         Run the Euclidean GCD algorithm on G_j^(p), using column operations to get the
          GCD of the G_j^(p) into the first entry of G_j^(p). Call this first column v.
5         Add multiples of v to cancel out every other non-zero entry in G_j^(p).
6         Replace column v with c · v, where c = min{ℓ ∈ [q_p] : v_j · ℓ = 0}.
7     end
8 end
9 return G
```

Claim 6.2. *Consider a codeword $c \in \mathcal{C}$ where $\mathcal{C}$ is the span of $G \in (\mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_u})^{m \times n}$. If we run ContractAbelian(G, j) on a coordinate j such that $c_j = 0$, then c will still be in the span of G after the contraction.*

Proof. First, we show that after line 4 in the Algorithm 13, c is still in the span. Indeed, the Euclidean GCD algorithm only ever makes column operations of the form $v_1 + c \cdot v_2 \to v_1$. This means that every step of the algorithm is invertible, so in particular, after every step of the Euclidean GCD algorithm, the span of the matrix G will not have changed.

Now, in line 5 of Algorithm 13, this again does not change the span of G. Indeed, we can simply undo this step by adding back multiples of v to undo the subtraction. Since the span of G hasn't changed after this step, c must still be in the span.

Finally, in line 6, the span of G actually changes. However, since c was zero in its jth coordinate, it must have been 0 in the pth index of the jth coordinate as well. This means it could only contain an integer multiple of c times the column v (again where c is defined as $c = \min\{\ell \in [q] : v_j \cdot \ell = 0\}$), as otherwise it would be non-zero in this index. Hence c will still be in the span of the code after this step, as it is only codewords that are non-zero in the pth index of coordinate j that are being removed. □

Claim 6.3. *After calling* ContractAbelian(G, j) *on a non-zero coordinate j, the number of distinct codewords in the span of G decreases by at least a factor of 2.*

Proof. Consider the matrix G after line 5 of Algorithm 13. By the previous proof, we know that the span of G has not changed by line 2. Now, after scaling up v by c, we completely zero out the pth index of row j of G. Previously, for any codeword in the code which was zero in coordinate j, we could correspondingly add a single multiple of column v to it to create a new codeword

that was non-zero in the pth index of its jth coordinate. Thus, the number of codewords which were non-zero in coordinate j was at least as large as the number which were zero. After scaling up column v, codewords which were non-zero in this coordinate are no longer in the span, so the number of distinct codewords has gone down by at least a factor of 2. □

Algorithm 14: Repeated Contractions(G, α, q)

```
Input generating matrix G ∈ Z_q^{m×n}.
while more than q^{α+1} distinct codewords in the span of G do
    Choose a random non-zero row G_j of G.
    Set G = ContractAbelian(G, j).
end
```

Claim 6.4. *Let $\mathcal{C}$ be the span of $G \in (\mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_u})^{m \times n}$, and let d be an integer. Suppose every subcode $\mathcal{C}' \subseteq \mathcal{C}$ satisfies $\text{Density}(\mathcal{C}') < \frac{1}{d}$. Then for any $\alpha \in \mathbb{Z}$, the number of distinct codewords of weight $\leq \alpha d$ is at most $\binom{n \log(q)}{\alpha} \cdot q^{\alpha+1}$, where $q = \prod_{i=1}^{u} q_i$.*

Proof. Consider an arbitrary codeword c in the span of G of weight $\leq \alpha d$. Let us consider what happens when we run Algorithm 14. In the worst case, we remove only a factor of 2 of the codewords in each iteration. Thus, suppose that this is indeed the case and that the code starts with q^n distinct codewords. Now, after ℓ iterations, this means in the worst case there are $2^{n \log(q) - \ell}$ distinct codewords remaining. Note that after running these contractions, the resulting generating matrix defines a subcode $\mathcal{C}'$ of our original space. Then, by our assumption on the density of every subcode, the number of non-zero rows in the generating matrix must be at least $(n \log(q) - \ell)d$. Thus, the probability that c remains in the span of the generating matrix after the next contraction is at least

$$\Pr[c \text{ survives iteration}] \geq 1 - \frac{\alpha d}{(n \log(q) - \ell)d} = 1 - \frac{\alpha}{n \log(q) - \ell}.$$

Now, we can consider the probability that c survives all iterations. Indeed,

$$\begin{aligned} \Pr[c \text{ survives all iterations}] &\geq \frac{n \log(q) - \alpha}{n \log(q)} \cdot \frac{n \log(q) - \alpha - 1}{n \log(q) - 1} \cdot \cdots \cdot \frac{\alpha + 1 - \alpha}{\alpha + 1} \\ &= \binom{n \log(q)}{\alpha}^{-1}. \end{aligned}$$

At this point, the number of remaining codewords is at most $q^{\alpha+1}$. Hence, the total number of codewords of weight $\leq \alpha d$ is at most $\binom{n \log(q)}{\alpha} \cdot q^{\alpha+1}$. □

Corollary 6.5. *For any linear code $\mathcal{C}$ over $(\mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_u})^m$, with $\leq q^n$ distinct codewords, and for any integer $d \geq 1$, there exists a set S of at most $n \log(q) \cdot d$ rows, such that upon removing these rows, for any integer $\alpha \geq 1$, the resulting code has at most $\binom{n \log(q)}{\alpha} \cdot q^{\alpha+1}$ distinct codewords of weight $\leq \alpha d$.*

Proof. Suppose for the code $\mathcal{C}$ that there is a subcode of density $\geq 1/d$. Then, this means there exists a set $n'd$ rows (coordinates), with at least $2^{n'}$ distinct codewords that are completely contained on these rows. Now, by removing these rows, this yields a new code where the number of distinct codewords has decreased by a factor of $2^{n'}$. Thus, if removing these rows yields a new code $\mathcal{C}'$, this new code has at most $q^n / 2^{n'}$ distinct codewords. Now, if there is no subcode of density $\geq 1/d$

in $\mathcal{C}'$, we are by the preceding claim (i.e. we will satisfy the counting bound). Otherwise, we can again apply this decomposition, removing another $n''d$ rows to yield a new code with at most $2^{n\log(q)-n'-n''}$ distinct codewords. In total, as long as we remove rows in accordance with condition 1, we can only ever remove $n\log(q)d$ rows total, before there are no more distinct codewords remaining, at which point the counting bound must hold.

Thus, there exists a set of at most $n\log(q)\cdot d$ rows, such that upon their removal, we satisfy the aforementioned counting bound. □

Theorem 6.6. *For any code $\mathcal{C} \subseteq (\mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_u})^m$ with q^n distinct codewords, there is an efficient algorithm for creating $(1 \pm \epsilon)$ sparsifiers to $\mathcal{C}$ with $\widetilde{O}(n\log^2(q)/\epsilon^2)$ weighted coordinates.*

Proof. Note that we can simply replace Algorithm 1 with Algorithm 13 in every instance in which it appears in Algorithm 11. Because this contraction algorithm satisfies all the same properties as the original, the theorems follow immediately. Note that we set $q = q_1 \cdot q_2 \cdot \cdots \cdot q_u$, and hence do not get any improvement by working over $\mathbb{Z}_{p_1} \times \mathbb{Z}_{p_2}$ as opposed to $\mathbb{Z}_{p_1 \cdot p_2}$. For completeness, we provide a more extensive proof in the appendix (see Appendix A). □

With this, we are now able to prove the efficient sparsification for any CSP using an affine predicate P over an Abelian group.

Theorem 6.7. *Let P be an affine predicate over an Abelian group A. Then, for any CSP instance C with predicate P on a universe of n vaiables, we can efficiently $(1\pm\epsilon)$-sparsify C to $\widetilde{O}(n\log^2(|A|)/\epsilon^2)$ constraints.*

Proof. Indeed, let the predicate P be given. Then, we can write P as a constraint of the form

$$P(b_1, \dots b_r) = \mathbf{1}[\sum_{i=1}^{r} a_i b_i \neq a_0],$$

(invoking Fact 6.1) where $a_i \in \mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_u}$ and $b_i \in \{0,1\}$. Now, we will show that we can create a generating matrix G such that for any assignment $x \in \{0,1\}^n$, there is a corresponding codeword in the code which is non-zero in coordinate j if and only if constraint j is satisfied. Indeed, let us initialize a generating matrix $G \in (\mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_u})^{m \times (n+1)}$ where m is the number of constraints in C, and for each variable $x_1, \dots x_n$ in the universe, there is a single corresponding column among the first n columns. Now, suppose that the jth constraint of C is of the form

$$\mathbf{1}[\sum_{i=1}^{r} a_i x_{j_i} \neq a_0].$$

Then, in the jth row of the generating matrix, let us place the value a_i in the j_ith column. Finally, in the last column ($n+1$st column), let us place $-a_0$.

With this generating matrix, it follows that for any assignment $x \in \{0,1\}^n$, we can define a corresponding message $x' = [x_1, \dots x_n, 1]$ such that $(Gx')_j$ is non-zero if and only if $C_j(x)$ (the jth constraint) is satisfies (equal to 1) on this assignment. Indeed, $C_j(x) = 1$ if and only if

$$\sum_{i=1}^{r} a_i x_{j_i} \neq a_0,$$

or equivalently if

$$\sum_{i=1}^{r} a_i x_{j_i} - a_0 \neq 0.$$

Correspondingly,

$$(Gx')_j = \sum_{i=1}^{r} a_i x_{j_i} - a_0,$$

and the weight of the codeword Gx' is

$$\mathrm{wt}(Gx') = \left| \{ j \in [m] : \sum_{i=1}^{r} a_i x_{j_i} - a_0 \neq 0 \} \right|,$$

which exactly equals the value of the CSP on assignment x. Thus because there is an exact equivalence between the weight contributed by the coordinates in the codeword corresponding to x' and the weight contributed by the constraints with assignment x, returning a $(1 \pm \epsilon)$ sparsifier for the code generated by G also creates a sparsifier for the CSP C. Because this former task can be done efficiently (Theorem 6.6), this yields our desired result. □

6.1 Infinite Abelian Groups

The previous section has the disadvantage of the sparsifier size having a dependence on the size of the Abelian group. In this section we show that this is not necessary, and instead we can replace this with a polynomial dependence on the arity of the predicates. First, we make the following observation regarding affine Abelian predicates:

Claim 6.8. *Let $P : \{0,1\}^r \to \{0,1\}$ be an affine Abelian predicate such that $P(0^r) = 0$. Then, the space $P^{-1}(0)$ is closed under integer linear combinations over $\mathbb{Z}$.*

Proof. Let $y_1, \dots y_\ell \in P^{-1}(0)$, and $\alpha_1, \dots \alpha_\ell$ be such that $\sum_{j=1}^{\ell} \alpha_j y_j \in \{0,1\}^r$ (when addition is done over $\mathbb{Z}$). We claim then that $P(\sum_{j=1}^{\ell} \alpha_j y_j) = 0$ also.

This follows simply from P being an affine Abelian predicate. It must be the case that $P(b_1, \dots b_r) = \mathbf{1}[\sum_i a_i b_i \neq 0]$, for some $a_i \in A$, where A is an Abelian group. Then,

$$P(\sum_{j=1}^{\ell} \alpha_j y_j) = \mathbf{1}[\sum_i a_i (\sum_{j=1}^{\ell} \alpha_j y_j)_i \neq 0] = \mathbf{1}[\sum_{j=1}^{\ell} \alpha_j \sum_i a_i (y_j)_i \neq 0].$$

But, because each $y_j \in P^{-1}(0)$, it must be the case that $\sum_i a_i (y_j)_i = 0$. Thus, the entire sum must be 0, we can conclude that $P(\sum_{j=1}^{\ell} \alpha_j y_j) = 0$, as we desire. □

Now, we will show that any predicate whose unsatisfying assignments form a closed space under integer linear combinations will be representable over a finite Abelian group. An immediate consequence of this is that affine Abelian predicates over infinite Abelian groups are sparsifiable to finite size. We explain this more in depth below.

Definition 6.2. *For a matrix $B \in \mathbb{Z}^{d \times \ell}$, the lattice generated by B (often denoted by $\mathcal{L}(B)$), is the set $\{Bx : x \in \mathbb{Z}^\ell\}$.*

From the preceding claim, we know that for an affine Abelian predicate P, the set of unsatisfying assignments is closed as a lattice. Thus, our goal will be to show that for any lattice, as long as the coefficients of the generating matrix are bounded, then we can characterize exactly which elements are in the lattice with a finite set of constraints over a finite alphabet. It follows then that we will be able to express this as an affine predicate over a finite Abelian group.

Lemma 6.9. *Suppose B is a $d \times \ell$ matrix with entries in $\mathbb{Z}$, such that the magnitude of the largest sub-determinant is bounded by M, and rank$(B) = k$. Then, every element of the lattice generated by the columns of B is given exactly by the solutions to $d - k$ linear equations and k modular equations. All coefficients of the linear equations are bounded in magnitude by M, and all modular equations are written modulo a single $M' \leq M$.*

Proof. First, if $\text{rank}(B) = k < d$, this means that there exist k linearly independent rows such that the remaining $d - k$ rows are linear combinations of these rows. Let us remove these rows for now, and focus on $B' \in \mathbb{Z}^{k \times \ell}$ where now the matrix has full row-rank.

It follows that for this matrix B' we can create a new matrix $\hat{B}$ such that the lattice generated by B' (denoted by $\mathcal{L}(B') = \mathcal{L}(\hat{B})$) and $\hat{B}$ is in Hermite Normal Form (HNF) [Mic14b]. In this form, $\hat{B}$ is lower triangular with the diagonal entries of $\hat{B}$ satisfying

$$\det(\hat{B}) = \prod_{i=1}^{k} \hat{B}_{i,i} \leq \max_{k \times k \text{ subrectangle A}} \det(B'_A) \leq M.$$

Further, all columns beyond the kth column will be all zeros, so we can remove these from the matrix (because the lattice generated by the first k columns will be the same as the lattice generated by *all* columns).

We can now define the *dual* lattice to $\mathcal{L}(B') = \mathcal{L}(\hat{B})$). For a lattice $\Lambda \subseteq \mathbb{Z}^k$, we say that

$$\text{dual}(\Lambda) = \{x \in \mathbb{Q}^k : \forall y \in \Lambda, \langle x, y \rangle \in Z\}.$$

Here it is known that the dual is an exact characterization of the lattice Λ. I.e., any vector in Λ will have integer-valued inner product with *any* vector in the dual, while for any vector not in Λ, there exists a vector in the dual such that the inner-product is not integer valued [Mic14a].

Now, for our matrix $\hat{B}$, it is known that one can express the dual lattice to $\mathcal{L}(\hat{B})$ with a generating matrix $\hat{D} = \hat{B}(\hat{B}^T\hat{B})^{-1}$ [Mic14a]. As a result, it must be the case that $\mathcal{L}(\hat{D}) \subseteq \mathbb{Z}^k / \det(\hat{B})$. If a vector x of length k is not in $\mathcal{L}(\hat{B})$, it must be the case that there exists a column y of $\hat{D}$ such that $\langle x, y \rangle \notin \mathbb{Z}$. Otherwise, if the inner-product with every column is in $\mathbb{Z}$, it follows that for any vector in the dual, the inner product would also be in $\mathbb{Z}$, as we can express any vector in the dual as an integer linear combination of columns in $\hat{D}$. Thus, it follows that membership of a vector x in $\mathcal{L}(\hat{B})$ can be tested exactly by the k equations $\forall i \in [k] : \langle x, \hat{D}_i \rangle \in \mathbb{Z}$, where $\hat{D}_i$ is the ith column of $\hat{D}$.

Now, because every entry of $\hat{D}$ has denominator dividing $\det(\hat{B})$, it follows that we can scale up the entire equation by $\det(\hat{B})$. Thus, an equivalent way to test if $x \in \mathcal{L}(\hat{B})$ is by checking if $\forall i \in [k] : \langle x, \det(\hat{B}) \cdot \hat{D}_i \rangle = 0 \mod \det(\hat{B})$. Now, all the coefficients of these equation are integers, and we are testing whether the sum is 0 modulo an integer. Thus, we can test membership of any k-dimensional integer vector in $\mathcal{L}(\hat{B})$ with k modular equations over $\det \hat{B} \leq M$.

The above argument gives a precise way to characterize when the restriction of a d dimensional vector to a set of coordinates corresponding with linearly independent rows in B is contained in the lattice generated by these same rows of B. It remains to show that we can also characterize when the dependent coordinates (i.e. coordinates corresponding to the rows that are linearly dependent on these rows) are contained in the lattice. Roughly speaking, the difficulty here arises from the fact that we are operating with a non-full dimensional lattice. I.e., there exist directions that one can continue to travel in $\mathbb{Z}^n$ without ever seeing another lattice point. In this case, we do not expect to be able to represent membership in the lattice with a modular linear equation, as these modular linear equations rely on periodicity of the lattice.

Instead, here we rely on the fact that for any of the rows of B that are linearly dependent, we know that there is a way to express it as a linear combination of the set of linearly independent rows. WLOG, we will assume the first k rows $r_1, \dots r_k$ are linearly independent, and we are interested in finding c_i such that $\sum_{i=1}^{k} c_i r_i = r_{k+1}$. Now, consider any subset A of k linearly independent columns amongst these k rows. We denote the corresponding restriction of the rows to these columns by $r_i^{(A)} \in \{0,1\}^k$. It follows that if we want a linear combination of these rows such that $\sum_{i=1}^{k} c_i r_i^{(A)} = r_{k+1}^{(A)}$, we can express this a constraint of the form $M^{(A)} c = (r_{k+1}^{(A)})^T$, where we view the ith column of M as being the (transpose of) $r_i^{(A)}$ and c as being the vector of values $c_1, \dots c_k$. Using Cramer's rule, we can calculate that $c_i = \det(M_i^{(A)}) / \det(M^{(A)})$, where $M_i^{(A)}$ is defined to be the matrix $M^{(A)}$ with the ith column replaced by $(r_{k+1}^{(A)})^T$. In particular, this means that we can express $r_{k+1}^{(A)} = \sum_{i=1}^{k} \det(M_i^{(A)}) / \det(M^{(A)}) \cdot r_i^{(A)}$, and because A corresponds to a set of linearly independent columns, it must also be the case that $r_{k+1} = \sum_{i=1}^{k} \det(M_i^{(A)}) / \det(M^{(A)}) \cdot r_i$. We can re-write this as an integer linear equation by expressing $r_{k+1} \cdot \det(M^{(A)}) = \sum_{i=1}^{k} \det(M_i^{(A)}) \cdot r_i$. This means that for any valid vector $x \in \mathbb{Z}^d$ expressable as a linear combination of the columns of B, it must be the case that $x_{k+1} \cdot \det(M^{(A)}) = \sum_{i=1}^{k} \det(M_i^{(A)}) \cdot x_i$, as the $k+1$st coordinate is always a specific linear combination of the first k coordinates.

We can repeat the above argument for each of the $d-k$ linearly independent rows. Let j denote the index of a row linearly dependent on the first k rows. It follows that $x_j \cdot \det(M^{(A)}) = \sum_{i=1}^{k} \det(M_i^{(A),j}) \cdot x_i$, where now $M_i^{(A),j}$ is the $d \times d$ matrix $M^{(A)}$ where the ith column has been replaced with $(r_j^{(A)})^T$. Note that every coefficient that appears in these equations above is of the form $\det(C)$ where C is a $d \times d$ submatrix of B. It follows that each of these coefficients is bounded in magnitude by M, where M is again defined to be the maximum magnitude of the determinant of any square sub-matrix.

To conclude, we argue that for any vector $x \in \mathcal{L}(B)$, x satisfies all of the above linear equations and modular equations. The first part of the proof showed that amongst the set of linearly independent rows S, the dual of the lattice exactly captures when x_S is in the lattice generated by B_S. That is, if x_S is generated by B_S, then x_S satisfies the above modular linear equations, while if x_S is not in the span of B_S, then x_S does not satisfy the modular linear equations. Now, if x_S does not satisfy the modular linear equations, this is already a witness to the fact that x is not in the $\mathcal{L}(B)$. But, if x_S is in the span of B_S, then if x is in the span of B, it must also be the case that the coordinates of $x_{\bar{S}}$ satisfy the exact same linear dependence on the x_S that $B_{\bar{S}}$ has on B_S. This is captured by our second set of linear equations. □

Theorem 6.10. *Let $P : \{0,1\}^r \to \{0,1\}$ be a predicate of arity r. Let $S = P^{-1}(0) \subseteq \{0,1\}^r$ denote the unsatisfying assignments of P. If S is closed under integer valued linear combinations, then CSPs with predicate P on n variables are efficiently-sparsifiable to size $\widetilde{O}(n \cdot r^4/\epsilon^2)$.*

Proof. Let us create a matrix $B \in \{0,1\}^{r \times |S|}$ where the ith column of B is the ith element of S. Let k be the rank of B. It follows that for any assignment $x \in \{0,1\}^r$, we can exactly express the membership of x in $\mathcal{L}(B)$ with k modular linear equations, and $r-k$ linear equations. I.e., x is in $\mathcal{L}(B)$ if and only if all of these equations are satisfied. Note that the d modular linear equations are all over modulus $M \leq \max_{k \in [r], k \times k \text{ subrectangle } A} \det(B'_A) \leq (r)^r$. Likewise, the integer linear equations also all have coefficients $\leq (r)^r$ (by Lemma 6.9). It follows that because $x \in \{0,1\}^r$, we can choose a prime p such that $p \geq 2 \cdot r \cdot (r)^r$. Now, for any of the integer linear equations of the form $c_1 x_1 + \dots c_k x_k - c_{k+1} x_{k+1}$, it will be the case that for $x \in \{0,1\}^r$,

$$c_1 x_1 + \dots c_k x_k - c_{k+1} x_{k+1} = 0 \iff c_1 x_1 + \dots c_k x_k - c_{k+1} x_{k+1} = 0 \mod p.$$

This is because the expression on the left can never be as large as p or $-p$ since we chose p to be sufficiently large.

Thus, we can express $x \in \{0,1\}^r$ as being in the lattice $\mathcal{L}(B)$ if and only if all r modular equations are 0. This is then the OR of r modular equations, which can be expressed over the Abelian group $A = \mathbb{Z}_M \times \mathbb{Z}_m \cdots \times Z_m \times Z_p \cdots \times \mathbb{Z}_p$, where there are k copies of Z_m, and r copies of $\mathbb{Z}_p$. It follows that $|A| \leq (2 \cdot r \cdot (r)^r)^r = (2r)^r \cdot (r)^{2r^2}$. Thus, for any CSP on predicate P of the above form on n variables, we can efficiently $(1 \pm \epsilon)$ sparsify P to size $\widetilde{O}(n \cdot r^4/\epsilon^2)$ via Theorem 6.7. □

Theorem 1.7. *If $P : \{0,1\}^r \to \{0,1\}$ is an affine Abelian predicate over an Abelian group A, then CSP(P) is $(\epsilon, \widetilde{O}(n \cdot \min(r^4, \log^2(|A|))/\epsilon^2)$-efficiently-sparsifiable for every $\epsilon > 0$.*[5]

Proof. This follows because we can assume WLOG that $P(0^r) = 0$. Then, it must be the case that $P^{-1}(0)$ is closed under integer linear combinations (by Claim 6.8), and we can invoke the preceding theorem. □

Note that here we are dealing with boolean predicates (i.e., operating on $\{0,1\}^r$). We can extend this lattice characterization to larger alphabets, and do so in Appendix C.

7 Impossibility of Sparsifying Affine CSPs Over Non-Abelian Groups

In this section, we will complement the result of the previous section and show that in fact, if an affine constraint is written as

$$P(b_1, \ldots b_r) \sum_{i=1}^{r} a_i b_i \neq a_0$$

for $a_i \in H$, $b_i \in \{0,1\}$ and H being a non-Abelian group, then there exist CSP instances with predicate P that require sparsifiers of quadratic size. To show this, we will make use of Theorem 3.4 from the work of Khanna, Putterman, and Sudan [KPS24]. This theorem shows that it suffices to show that a predicate has a projection to an AND of arity 2 in order to conclude that there exist CSP instances with this predicate that require sparsifiers of quadratic size.

Theorem 1.8. *For every non-Abelian group G, there exists a predicate $P : \{0,1\}^4 \to \{0,1\}$ and an $\epsilon_0 > 0$ such that P is an affine predicate over G and CSP(P) is not $(\epsilon_0, o(n^2))$-sparsifiable.*

Proof. Indeed, let H be a non-Abelian group, and let a, b be two elements which do not commute. That is, let us assume that $a + b \neq b + a$. Then, consider the following predicate P of arity 4:

$$P(x_1, x_2, x_3, x_4) = \mathbf{1}[ax_1 + bx_2 - ax_3 - bx_4 \neq 0].$$

Note that $P(0,0,0,0) = 0$, $P(1,0,1,0) = \mathbf{1}[a - a \neq 0] = 0$, $P(0,1,0,1) = \mathbf{1}[b - b \neq 0] = 0$, but

$$P(1,1,1,1) = \mathbf{1}[a + b - a - b \neq 0] = 1,$$

as if $a+b-a-b = 0$, then a, b will commute with one another, which violates our original assumption. This means that $0000, 1010, 0101$ are unsatisfying assignments, while 1111 is a satisfying assignment. In particular, we can consider the following restriction

$$\pi(x_1) = c, \pi(x_2) = d, \pi(x_3) = c, \pi(x_4) = d.$$

[5] We note that A does not have to be a finite group in this theorem, though in all the applications to CSPs we only use finite Abelian groups.

It follows then that $P(\pi(x_1), \pi(x_2), \pi(x_3), \pi(x_4)) = \text{AND}(c, d)$. Thus, there exists a projection of P to an AND of arity 2, so we can invoke Theorem 3.4 and conclude that there exist CSP instances on predicate P which require sparsifiers with $\Omega(n^2)$ surviving constraints. □

In this sense, the preceding result we proved about sparsifying affine constraints over Abelian groups is in fact the strongest possible result we could hope to hold over affine constraints, as even the most simple non-Abelian groups will have instances which are not sparsifiable.

8 Sparsifying Symmetric CSPs

In this section, we will characterize the sparsifiability of symmetric CSPs. Namely, for a predicate $P : \{0,1\}^r \to \{0,1\}$ we say that P is symmetric if for any permutation $\pi : [r] \to [r]$, $P(x) = P(\pi \circ x)$. This is equivalent to saying that $P(x)$ is uniquely determined by the number of non-zero entries in x. First, we will introduce the definition of a symmetric predicate being periodic.

Definition 8.1. *A symmetric predicate $P : \{0,1\}^r \to \{0,1\}$ is **periodic** if for any $x, y \in \{0,1\}^r$ such that $P(x) = P(y) = 0$ and $|y| > |x|$, then for any $z \in \{0,1\}^r$ such that $|z| = 2|y| - |x|$ or $|z| = 2|x| - |y|$, $P(z) = 0$. In this context $|x|$ refers to the number of non-zero entries in x.*

With this, we are then able to state the main theorem of this section.

Theorem 8.1. *For a symmetric predicate $P : \{0,1\}^r \to \{0,1\}$, all CSPs with predicate P are sparsifiable to nearly-linear size if and only if P is periodic.*

Now, we are ready to prove the following:

Lemma 8.2. *Suppose a symmetric predicate $P : \{0,1\}^r \to \{0,1\}$ is not periodic. Then there is a projection $\pi : \{x_1, \dots x_r\} \to \{0, 1, a, \neg a, b, \neg b\}$ such that $P(\pi(x_1), \dots \pi(x_r)) = AND(a, b)$.*

Proof. Suppose that a symmetric CSP does not have this periodic property. This implies that there exists a, b, c such that for $|x| = a$, $|x| = b$, $P(x) = 0$, but for $|x| = 2a - b$, $P(x) = 1$. WLOG we will assume $b > a$ but we can equivalently swap a with b in the following construction and assume the opposite. We will show that this contains an affine projection to AND, and therefore requires sparsifiers of quadratic size (see Theorem 3.4). Indeed, consider the following bit strings:

1. The bit string y_1 with $r - a$ 0's followed by a 1's.
2. The bit string y_2 with $r - b$ 0's followed by b 1's.
3. The bit string y_3 with $r - 2b + a$ 0's, followed by $b - a$ 1's followed by $b - a$ 0's, followed by a 1's.
4. The bit string y_4 with $r - 2b + a$ 0's, followed by all 1's.

By the above conditions, $P(y_1) = P(y_2) = P(y_3) = 0$, while $P(y_4) = 1$. Now, we consider the projection $\pi : \{x_1, \dots x_r\} \to \{0, 1, x, \neg x, y, \neg y\}$ such that $\pi(x_1) = \pi(x_2) = \cdots = \pi(x_{r-2b+a}) = 0$, $\pi(x_{r-2b+a+1}) = \cdots = \pi(x_{r-b}) = x$, $\pi(x_{r-b+1}) = \cdots = \pi(x_{r-a}) = y$, and all remaining are sent to 1. Once can verify that under this projection, $P_\pi = \text{AND}$. □

We now show that symmetric, perioidic predicates can be written as a simple affine equation.

Lemma 8.3. *Let $P : \{0,1\}^r \to \{0,1\}$ be a symmetric, periodic predicate. Then, there exist $c, \ell \in [r+1]$ such that*

$$P(b_1, \dots b_r) = \mathbf{1}[\sum_{i=1}^{r} b_i \neq c \mod \ell].$$

Proof. Clearly, if there exists ℓ such that $P(x_1, \dots x_r) = \mathbf{1}[\sum_i x_i \neq c \mod \ell]$, then the zero levels of P are evenly spaced out, and clearly satisfy the condition of being periodic.

Now, let $P : \{0,1\}^r \to \{0,1\}$ be a symmetric, periodic predicate, and let $x, y \in \{0,1\}^r$ be such that $P(x) = P(y) = 0$ and $||x| - |y||$ is as small as possible (without being 0). Let $||x| - |y|| = \ell$. Now, note that the predicate P can be written as

$$P(b_1, \dots b_r) = \mathbf{1}[\sum_{i=1}^{r} b_i \neq |x| \mod \ell].$$

This follows because by the definition of periodicity, every string z such that $|z| = |x| - \ell$ or $|z| = |y| + \ell$ must satisfy $P(z) = 0$. Inductively then, every string z' which satisfies $|z'| = |x| - 2\ell$, $|z'| = |y| + 2\ell$ must also satisfy $P(z') = 0$, and so on. Further, note that by the minimality of ℓ, no strings x, y such that $P(x) = P(y) = 0, |x| \neq |y|$ can be closer than hamming distance ℓ. Thus, $P(b_1, \dots b_r)$ can be written as $\mathbf{1}[\sum_{i=1}^{r} b_i \neq |x| \mod \ell]$.

Finally, if there is only one level of the predicate which is 0, we can choose $\ell = r + 1$. □

Next, we will show that for any symmetric, periodic predicate, we can indeed sparsify CSPs using this predicate to nearly-linear size.

Lemma 8.4. *Let $P : \{0,1\}^r \to \{0,1\}$ be a symmetric, periodic predicate. Let C be a CSP using predicate P on a universe of n variables. Then, we can efficiently create a $(1 \pm \epsilon)$ sparsifier for C with only $\widetilde{O}(n \log^2(r)/\epsilon^2)$ surviving weighted constraints.*

Proof. So, let such a predicate P be given. Per Lemma 8.3 there exists $c, \ell \in [r+1]$ such that

$$P(b_1, \dots b_r) = \mathbf{1}[\sum_{i=1}^{r} b_i \neq c \mod \ell].$$

Now, we will show how to create a code over Z_ℓ which exactly captures the CSP built on predicate P. First, we create a generating matrix $G \in \mathbb{Z}_\ell^{m \times n+1}$, where m is the number of constraints in the CSP, and n is the size of the universe of variables. We associate each of the first n columns to each of the n variables. Next, let C_j refer to the jth constraint of the CSP. Suppose that C_j acts on variables $x_{j_1}, \dots x_{j_r}$. Then, in the corresponding jth row of the generating matrix G, we place a 1 in the columns corresponding to $x_{j_1}, \dots x_{j_r}$, and place $-c \mod \ell$ in the final $n+1$st column. Now, consider an assignment to the variables $x_1, \dots x_n$ in the original CSP C. Note that the jth constraint C_j is satisfied if and only if

$$\sum_{i=1}^{r} x_{j_i} \neq c \mod \ell.$$

Correspondingly, consider the code generated by the generating matrix G. For the message $x' = (x_1, \dots x_n, 1)$, note that the jth coordinate in the codeword Gx' is

$$(Gx')_j = \sum_{i=1}^{r} x_{j_i} - c \mod \ell.$$

This means that the weight of the codeword Gx' is exactly

$$\mathrm{wt}(Gx') = |\{j \in [m] : (Gx')_j \neq 0\}| = |\{j \in [m] : \sum_{i=1}^{r} x_{j_i} - c \neq 0 \mod \ell\}|$$

$$= |\{j \in [m] : \sum_{i=1}^{r} x_{j_i} \neq c \mod \ell\}| = |\{j \in [m] : C_j(x) = 1\}|.$$

Thus, any subset of the coordinates of G which creates $(1 \pm \epsilon)$ sparsifier for the codewords in the code generated by G must also yield a subset of the constraints of C which yields a $(1 \pm \epsilon)$ sparsifier for the entire CSP. Because we can efficiently compute such sparsifiers for G with only $\widetilde{O}(n \log^2(\ell)/\epsilon^2)$ surviving coordinates (by Theorem 5.19), we can thus efficiently find sparsifiers for C with only $\widetilde{O}(n \log^2(\ell)/\epsilon^2)$ surviving constraints. Using that $\ell \leq r + 1 \leq n + 1$, we get our desired result. □

Finally, we can now conclude the main result of this section:

Theorem 1.3. *Let $P : \{0,1\}^r \to \{0,1\}$ be a symmetric predicate. Then if P is periodic, CSP(P) is $(\epsilon, \widetilde{O}(n/\epsilon^2))$-efficiently sparsifiable for every $\epsilon \in (0,1)$. On the other hand, if P is not periodic then for every $0 < \epsilon < 1$, CSP(P) is not $(\epsilon, o(n^2))$-sparsifiable.*

Proof. If P is not periodic, then by Lemma 8.2 we know that P has a projection to AND. Then, by invoking the work of Khanna, Putterman, and Sudan [KPS24], we know there exist CSP instances with predicate P that require sparsifiers of size $\Omega_r(n^2)$.

Otherwise, if P is periodic, then by Lemma 8.4 we can efficiently $(1 \pm \epsilon)$ sparsify any CSP instance on P to only $\widetilde{O}(n \log^2(r)/\epsilon^2) = \widetilde{O}(n/\epsilon^2)$ constraints. □

9 Non-trivial Sparsification for Almost All Predicates

In this section, we show that almost all predicates can be non-trivially sketched. Indeed, given a predicate $P : \{0,1\}^r \to \{0,1\}$, we show that as long as P has 0 or at least 2 satisfying assignments, then any CSP instance with predicate P on a universe of n admits an $(\epsilon, \widetilde{O}(n^{r-1}/\epsilon^2))$-sparsifier. At a high level, we do this by extending our method for sparsifying systems of affine equations not being zero to sparsifying systems of polynomials not being equal to zero. Although this leads to a blowup in the size of our sparsifiers, it is distinctly more powerful, and allows us to model more general predicates. We recall Theorem 1.9:

Theorem 1.9. *If $P : \{0,1\}^r \to \{0,1\}$ is a degree ℓ polynomial over an Abelian group A then CSP(P) is $(\epsilon, \widetilde{O}(n^\ell \min(r^{4\ell}, \log^2(|A|))/\epsilon^2)$-efficiently-sparsifiable for every $\epsilon > 0$.*

Alternatively stated, the above implies the following:

Corollary 9.1. *Let $P : \{0,1\}^r \to \{0,1\}$ be a predicate such that there exists a polynomial Q with coefficients in an Abelian group A of degree ℓ satisfying $\forall x \in \{0,1\}^r : \mathbf{1}[Q(x) \neq 0] = P(x)$. Then, all CSPs with predicate P on a universe of n variables efficiently admit $(\epsilon, \widetilde{O}(n^\ell \min(\log^2(|A|), r^{4\ell})/\epsilon^2))$-sparsifiers.*

Proof. We will prove this via a reduction to code sparsification. We will prove the following lemma along the way:

Lemma 9.2. *For any polynomials $P_1, \dots P_m$ of degree $\leq \ell$ over A, there exists generating matrix $G \in A^{m \times O(n^\ell)}$ such that for any $x \in \{0,1\}^n$, there exists a corresponding $x' \in \{0,1\}^{O(n^\ell)}$ such that $\forall j \in [m]$, $(Gx')_j = P_j(x')$.*

Proof. Indeed, let us consider the jth polynomial P_j. We can write

$$P_j(x) = \sum_{i_1+i_2+\cdots+i_n \leq \ell} a_{j,i_1,\dots i_n} \prod_{u=1}^{n} x_u^{i_u}.$$

Now, in our generating matrix, we associate each column of the generating matrix with a single term of each polynomial. In the jth row (corresponding to the jth polynomial), in the column corresponding to the term $\prod_{u=1}^n x_u^{i_u}$, we place $a_{j,i_1,\dots i_n}$. Note that there can be at most $O(n^\ell)$ terms of degree $\leq \ell$, so it follows the size of the generating matrix is at $m \times O(n^\ell)$.

Now consider any assignment $x \in \{0,1\}^n$. We will show that there is a corresponding assignment $x' \in \{0,1\}^{O(n^\ell)}$ such that Gx' is non-zero in coordinate j if and only if $P_j(x)$ was non-zero. From there, it follows that sparsifying the code generated by G will yield a sparsifier for the system of polynomials.

So, let such an assignment $x \in \{0,1\}^n$ be given. Let $x' \in \{0,1\}^{O(\ell)}$ be such that the pth entry of x' is the evaluation of the term corresponding to the pth column of G. That is, $x'_p = \prod_{u=1}^n x_u^{i_u}|_x$, where $\prod_{u=1}^n x_u^{i_u}$ is the term corresponding to the pth column of G.

Then, it follows that $(Gx')_j = \sum_{i_1+i_2+\cdots+i_n \leq \ell} a_{j,i_1,\dots i_n} \prod_{u=1}^n x_u^{i_u} = P_j(x)$. □

Now, by Theorem 6.7, we can efficiently sparsify the code generated by G to only $\widetilde{O}(n^\ell \min(\log^2(|A|), r^{4\ell})/\epsilon^2)$ remaining coordinates, so it also follows that we can sparsify our set of polynomials to $\widetilde{O}(n^\ell \min(\log^2(|A|), r^{4\ell})/\epsilon^2)$ remaining weighted polynomials such that for any assignment $x \in \{0,1\}^n$, the number of non-zero polynomials is preserved to a $(1 \pm \epsilon)$ fraction. □

We will also use the following trick for any predicate over any finite group, and in this context, $\mathbb{Z}_2$.

Claim 9.3. *Let $P_1, P_2, \dots P_s$ be such that each P_i is an affine predicate over $\mathbb{Z}_2$. Then, one can write the predicate $P = P_1 \vee P_2 \vee \cdots \vee P_s$ as single affine predicate over $(Z_2)^s$, such that $P(x) \neq 0$ if and only if there exists some i such that $P_i(x) \neq 0$.*

Proof. Indeed, each P_i is affine so we can write $P_i(x) = \sum_{j=1}^r a_{i,j}x_j - b_i$, where the addition is over $\mathbb{Z}_2$. Now, let the tuple $A^{(j)} = (a_{1,j}, \dots a_{s,j})$ and let $B = (b_1, \dots b_s)$. It follows that we can write

$$P(x) = \sum_{j=1}^{r} x_j \cdot A^{(j)} - B.$$

It follows that the ith entry of the tuple returned by $P(x)$ will be exactly $P_i(x) = \sum_{j=1}^r a_{i,j}x_j - b_i$. Thus, $P(x)$ is zero if and only if each of the constituent $P_i(x)$'s also evaluates to zero. Otherwise, if at least one of the $P_i(x)$ is non-zero, $P(x)$ will also be non-zero. □

Claim 9.4. *Let $P : \{0,1\}^r \to \{0,1\}$ be a predicate with two satisfying assignments. Then, there exists a polynomial $L_{a,b} : \{0,1\}^r \to \{0,1\}$ over $\mathbb{Z}_2$ of degree $r-1$ such that for any $y \in \{0,1\}^r$, $L_{a,b}(y) = 0$ if and only if $P(y) = 0$.*

Proof. Let the two satisfying assignments be $a = a_1, \dots a_r$ and $b = b_1, \dots b_r$. WLOG let us assume that the first t_1 bits of a and b are 1, the second t_2 bits of a and b are 0, the next t_3 bits are 1 for a and 0 for b, while the last t_4 bits of a, b are 0 for a and 1 for b. Consider then the following polynomial:

$$L_{a,b}(y) = \prod_{i=1}^{t_1} y_i \prod_{i=t_1+1}^{t_1+t_2} (1 - y_1) \prod_{i=t_1+t_2+2}^{t_1+t_2+t_3} (y_{t_1+t_2+1} + y_i - 1) \prod_{i=t_1+t_2+t_3}^{t_1+t_2+t_3+t_4} (y_{t_1+t_2+1} - y_i).$$

Note that the degree of L is $r-1$, and the only y's for which the expression evaluates to something non-zero are a and b (and in this case it either evaluates to 1 or -1). Over $\mathbb{Z}_2$, note that this is still the case, as $-1 = 1 \neq 0$ over $\mathbb{Z}_2$. □

Theorem 1.5. *Let $P : \{0,1\}^r \to \{0,1\}$. If $|P^{-1}(1)| \neq 1$ then CSP(P) is $(\epsilon, \widetilde{O}_r(n^{r-1}/\epsilon^2))$ efficiently-sparsifiable for every $\epsilon > 0$. Otherwise, for every $0 < \epsilon < 1$, CSP(P) is not $(\epsilon, o(n^r))$ sparsifiable.*

Proof. Let such a CSP instance C be given. Consider the jth constraint in this CSP, and call the corresponding predicate for the jth constraint $P_j : \{0,1\}^r \to \{0,1\}$. Note that if P_j is the constant 0 predicate, we can simply remove it from C without changing the value, so we will assume that P_j has at least 2 satisfying assignments. Let us write the satisfying assignments to P_j as $a_1, \dots a_s$. It follows that we can define the polynomials $L_{a_1,a_2}, L_{a_1,a_3}, \dots L_{a_1,a_s}$ such that $L_{a_1,a_i}(y) \neq 0$ if and only if $y = a_i$ or $y = a_1$ in accordance with Lemma 9.2. Now, by Lemma 9.2, each of these polynomials can be written as an affine equation over $\mathbb{Z}_2$ over a universe of variables of size $O(n^{r-1})$. So, let $\hat{L}_{a_1,a_i}$ refer to the polynomial L_{a_1,a_i} when instead viewed as a linear equation over $\mathbb{F}_2$ in a variable set of size $O(n^{r-1})$.

Now, it follows that $P_j = L_{a_1,a_2} \vee L_{a_1,a_3} \vee \cdots \vee L_{a_1,a_s}$ and $s \leq 2^r$. Over the universe of variables of size $O(n^{r-1})$, it follows that we can write each polynomial as a linear equation by Lemma 9.2, and hence we can write $P_j = \hat{L}_{a_1,a_2} \vee \hat{L}_{a_1,a_3} \vee \cdots \vee \hat{L}_{a_1,a_s}$. Because $s \leq 2^r$, this means we can write each P_j as a single affine constraint over $(\mathbb{Z}_2)^{2^r}$, such that $P_j(x) \neq 0$ if and only if one of the $\hat{L}_{a_1,a_i}(x) \neq 0$.

Now, we can write the generating matrix G for this linear space over $O(n^{r-1})$ variables on $(\mathbb{Z}_2)^{2^r}$. It follows that for each original assignment $x \in \{0,1\}^n$, there exists a corresponding assignment $x' \in \{0,1\}^{O(n^{r-1})}$ such that for each $j \in [m]$,

$$(Gx')_j \neq 0 \iff \hat{L}_{a_1,a_2}(x') \vee \hat{L}_{a_1,a_3}(x') \vee \cdots \vee \hat{L}_{a_1,a_s}(x') \neq 0 \iff P_j(x) \neq 0.$$

Hence, for any $x \in \{0,1\}^n$, there is a corresponding $x' \in \{0,1\}^{O(n^{r-1})}$ such that for ever $j \in [m]$, $(Gx')_j \neq 0 \iff P_j(x) \neq 0$. Therefore if a set of weighted indices is a sparsifier for the code generated by G, this same set of indices must be a sparsifier for the CSP over predicates P_j.

Finally, we conclude by noting that we proved the existence of $(\epsilon, \widetilde{O}(n \log^2(q)/\epsilon^2))$ sparsifiers for affine Abelian predicates over any Abelian group of size q, in a universe of variables of size n. By applying this to our generating matrix G, this translates to a $(1 \pm \epsilon)$-sparsifier for G of size $\widetilde{O}(n^{r-1} \cdot 4^r/\epsilon^2) = \widetilde{O}_r(n^{r-1}/\epsilon^2)$. □

Remark 9.5. *Note that in many senses the aforementioned result is the best possible. Indeed, for predicates on r variables which have 1 satisfying assignment, these predicates will have a projection to an AND of arity r, and thus require sparsifiers of size $\Omega(n^r)$ in the worst case.*

Likewise, there exist predicates of arity r with $2^{r-1} + 1$ satisfying assignments which require sparsifiers of size $\Omega(n^{r-1})$ in the worst case. Consider for instance the predicate $P : \{0,1\}^r \to$

$\{0,1\}$, *such that* $P(1,...) = 1$, *and* $P(0, x_1, \ldots x_{r-1}) = AND(x_1, \ldots x_{r-1})$. *This predicate has a projection to an AND of arity* $r-1$, *and thus requires sparsifiers of size* $\Omega(n^{r-1})$. *We are able to match this sparsifier size even when we are only told that the predicate has* two *satisfying assignments, and further, our result holds for* any *collection of predicates, provided each one has* 0, *or at least* 2 *satisfying assignments.*

10 Classifying Predicates of Arity 3

In this section, we prove Theorem 1.6. Recall this theorem:

Theorem 1.6. *For a predicate* $P : \{0,1\}^3 \to \{0,1\}$ *let* c *be the largest integer such that* P *has a projection to* AND_c. *Then, CSP*(P) *is* $(\epsilon, \widetilde{\Theta}(n^c))$ *efficiently-sparsifiable for every* $\epsilon \in (0,1)$, *and moreover, for every* $\epsilon \in (0,1)$, *CSP*(P) *is not* $(\epsilon, o(n^c))$ *sparsifiable.*

Proof. Let any such predicate P be given. First, the work of [KPS24] shows that if P does not have a projection to AND_2, then $\text{CSP}(P)$ can be written as an affine predicate (and in particular, can thus be efficiently sparsified to size $\widetilde{O}(n/\epsilon^2)$). Further, if P has a projection to AND_2, the beset possible size of any sparsifier is $\Omega(n^2)$. So, it remains only to distinguish between when $\text{CSP}(P)$ is sparsifiable to near-quadratic size vs. cubic size. Now, suppose that P has only 1 satisfying assignment. Then P (up to negation) is AND_3, and requires sparsifiers of size $\Omega(n^3)$ (and trivially admits such sparsifiers). Finally, let us consider all other predicates, i.e., P with projections to AND_2, but not only 1 satisfying assignment. These predicates require sparsifiers of size $\Omega(n^2)$ (by [KPS24]), yet by Theorem 1.5, efficiently admit sparsifiers of size $\widetilde{O}(n^2/\epsilon^2)$. This completes the proof. □

11 Applications Beyond CSPs

In this section, we discuss applications of our framework beyond just CSPs. In particular, we discuss efficient $\mathbb{F}_2$ Cayley-graph sparsification, hedge-graph sparsification, and sparsifying general hypergraphs with $\{0,1\}$-valued cardinality-based splitting functions.

11.1 Efficient Cayley-graph Sparsification over $\mathbb{F}_2$

First, we recall the definition of a Cayley graph and the notion of a Cayley graph sparsifier.

Definition 11.1. *A Cayley graph* G *is a graph with algebraic structure; its vertex set is defined to be a group, and the edges correspond to a set of generators* S, *along with weight* $(w_i)_{i \in S}$. *For every element in* $s \in S$, *and for every vertex* v, *there is an edge from* v *to* $v + s$ *of weight* w_s.

Definition 11.2. *Given a Cayley graph* G *with generating set* S *over* $\mathbb{F}_2^n$, *we say that* $\widetilde{G}$ *with a re-weighted generating set* $\widetilde{S} \subseteq \widetilde{S}$ *is a* $(1 \pm \epsilon)$ *Cayley-graph sparsifier of* G *if*

$$(1-\epsilon)L_G \preceq L_{\widetilde{G}} \preceq (1+\epsilon)L_G,$$

where L_G *here is used to denote the Laplacian of the graph* G.

[KPS24] provided the first proof of the existence of $(1 \pm \epsilon)$ Cayley-graph sparsifiers over $\mathbb{F}_2^n$ where the resulting generating set retains only $\widetilde{O}(n/\epsilon^2)$ generators. At the core of their result is the following theorem:

Fact 11.1. [KPS24] Given a Cayley graph G with generating set S over $\mathbb{F}_2^n$, let H be the matrix in $\mathbb{F}_2^{|S|\times n}$ where one places the generators as rows in the matrix (with their corresponding weights). If $\widetilde{H}$ is a $(1\pm\epsilon)$ code-sparsifier of H, then the Cayley-graph $\widetilde{G}$ with weighted generators coming from the rows of $\widetilde{H}$ is a $(1\pm\epsilon)$ Cayley-graph sparsifier of G.

Using this, we can derive the following theorem:

Theorem 11.2. *Given a Cayley graph G with generating set S over $\mathbb{F}_2^n$ and a parameter $\epsilon\in(0,1)$, there is a polynomial time (in $|S|, n, 1/\epsilon$), randomized algorithm which produces a $(1\pm\epsilon)$ Cayley-graph sparsifier $\widetilde{G}$ with generating set $\widetilde{S}\subseteq S$ such that $|\widetilde{S}|=\widetilde{O}(n/\epsilon^2)$.*

Proof. Given the graph G we create the generating matrix H as per Fact 11.1. Next, we invoke Theorem 5.19 to efficiently sparsify the generating matrix H. This yields a $(1\pm\epsilon)$ code-sparsifier $\widetilde{H}$ of H (in randomized polynomial time) such that $\widetilde{H}$ preserves only $\widetilde{O}(n/\epsilon^2)$ coordinates of H with probability $1-1/\mathrm{poly}(n)$. This yields the claim. □

11.2 Cayley-graph Sparsifiers over $\mathbb{Z}_q^n$

In this section, we show how to sparsify more general cayley-graphs over $\mathbb{Z}_q^n$ (where q is an arbitrary composite number).

For this, we first recall the following characterization of the eigenvectors of cayley-graphs over $\mathbb{Z}_q^n$.

Definition 11.3. *Let Γ be the group over $\mathbb{Z}_q^n$. Then we say that Γ has q^n* characters*, one for each vector $r\in\mathbb{Z}_q^n$. We denote each character by $\chi_r:\mathbb{Z}_q^n\to\mathbb{C}$. For a vector $x\in\mathbb{Z}_q^n$, we have that*

$$\chi_r(x)=e^{\frac{2\pi i}{q}\cdot\langle r,x\rangle}.$$

Further, for each character, we can define a vector $x_r\in\mathbb{C}^\Gamma$, where $(x_r)_a=\chi_r(a)$.

Then, if we let $G=\mathrm{Cay}(\Gamma,S)$, we have that $(x_r)_a$ is an eigenvalue of the adjacency matrix of G with eigenvalue

$$\sum_{s\in S}\chi_r(s).$$

In general, Cayley graphs may have weights w_s associated with each generator s. Then, the corresponding eigenvalues are simply the weighted sum.

In particular, we can simplify the expression of the eigenvalues when we consider *cyclically closed* Cayley graphs:

Definition 11.4. *For a generator $s\in\mathbb{Z}_q^n$, we say the cycle induced by s is $s,2s,3s,\dots(q-1)s$. We denote these cycles by $\mathrm{Cyc}(s)$, and we give each element in the cycle the same weight as the generator s.*

We say a Cayley graph $G=\mathrm{Cay}(\Gamma,S)$ is cyclically closed if there exists a set S' such that

$$\bigcup_{s\in S'}\mathrm{Cyc}(s)=S,$$

where we use the convention that union operator adds the weights of generators. I.e., if an element p appears in $\mathrm{Cyc}(s_1),\mathrm{Cyc}(s_2)$, then the weight of p is $w_{s_1}+w_{s_2}$. Notationally, we say that $S=\mathrm{Cyc}(S')$.

Now, observe that for a generator $s \in \mathbb{Z}_q^n$, and a character χ_r, we have that

$$\chi_r(s) + \chi_r(2s) + \cdots + \chi_r((q-1)s) = \sum_{j=1}^{q} \chi_r(js) - 1.$$

In particular, when $\langle r, s\rangle = 0$, then we see that this expression evaluates to $q-1$. Otherwise, when $\langle r, s\rangle \neq 0$, it evaluates to -1, as we are summing together (all but one of) the powers of a root of unity. With this, we get the following characterization:

Claim 11.3. *Let $G = \mathrm{Cay}(\mathbb{Z}_q^n, \mathrm{Cyc}(S'))$. Then, the eigenvalue of L_G corresponding to χ_r is*

$$q \cdot \mathrm{wt}(Hr),$$

where $H \in \mathbb{Z}_q^{n \times |S'|}$ is the generating matrix of a code over $\mathbb{Z}_q$ where there is a single row for each element $s \in S'$.

Proof. Recall that the eigenvalue of the adjacency matrix corresponding to χ_r is equal to

$$\sum_{s \in \mathrm{Cyc}(S')} \chi_r(s) = \sum_{s \in S'} \left(\sum_{j=1}^{q} \chi_r(js) - 1 \right) = \sum_{s \in S'} q \cdot \mathbf{1}[\langle r, s\rangle = 0] - 1.$$

For the Laplacian, we simply subtract the corresponding eigenvalue of the adjacency matrix from the degree. For each $s \in S'$, because the Cayley graph is cyclically closed, the corresponding degree is $q-1$. Thus, the corresponding eigenvalue of L_G is

$$\sum_{s \in S'} (q - 1 - (q \cdot \mathbf{1}[\langle r, s\rangle = 0] - 1)) = \sum_{s \in S'} w_s \cdot q \cdot \mathbf{1}[\langle r, s\rangle \neq 0].$$

In particular, if we let $H \in \mathbb{Z}_q^{n \times |S'|}$ denote the generating matrix of a code over $\mathbb{Z}_q$ where there is a single row for each $s \in S'$, then we can observe that the eigenvalue of L_G corresponding to $r \in \mathbb{Z}_q^n$ is exactly $q \cdot \mathrm{wt}(Hr)$, where we use the weighted notion of Hamming weight. □

Claim 11.4. *Let $G = \mathrm{Cay}(\mathbb{Z}_q^n, \mathrm{Cyc}(S'))$, and let H be its corresponding generating matrix over $\mathbb{Z}_q$. If $\widetilde{H}$ is a $(1 \pm \epsilon)$ code-sparsifier of H with the rows of $\widetilde{H}$ being denoted by $\widetilde{S}'$, then $\widetilde{G} = \mathrm{Cay}(\mathbb{Z}_q^n, \mathrm{Cyc}(\widetilde{S}'))$ is a $(1 \pm \epsilon)$ spectral Cayley-graph sparsifier of G.*

In the above claim, we are using the convention that the weight assigned to the coordinate corresponding to a generator $s \in \widetilde{S}'$ is the same as the weight assigned the same generator s in the Cayley graph $\widetilde{H}$.

Proof. First, recall that the eigenvectors of the Laplacian of an abelian Cayley graph are completely determined by the underlying group. Thus, $\widetilde{G}, G$ have the same eigenvectors. Thus, it remains only to show that their eigenvalues are within a factor of $(1 \pm \epsilon)$. For this, recall that by Claim 11.3, the eigenvalues of G are exactly $q \cdot \mathrm{wt}(Hr)$ for each $r \in \mathbb{Z}_q^n$. Likewise, for the graph $\widetilde{G}$, the eigenvalues are exactly $q \cdot \mathrm{wt}(\widetilde{H}r)$, where in both instances the Hamming weight we use is the *weighted* notion of Hamming weight. In particular, because $\widetilde{H}$ is a $(1 \pm \epsilon)$ code-sparsifier of H, we have that

$$q \cdot \mathrm{wt}(\widetilde{H}r) \in (1 \pm \epsilon) q \cdot \mathrm{wt}(Hr).$$

Hence, $\widetilde{G}$ is a $(1 \pm \epsilon)$ spectral-sparsifier of G. □

With this, we are ready to conclude our main theorem:

Theorem 11.5. *Let $G = \mathrm{Cay}(\mathbb{Z}_q^n, \mathrm{Cyc}(S))$, and let $\epsilon \in (0,1)$. Then, there is a randomized, polynomial time algorithm (in $\log(q), \epsilon, n, |S|$) which returns (with high probability) a re-weighted sub-Cayley graph $\widetilde{G} = \mathrm{Cay}(\mathbb{Z}_q^n, \mathrm{Cyc}(\widetilde{S}))$, with $|\widetilde{S}| = \widetilde{O}(n\mathrm{polylog}(q)/\epsilon^2)$ such that $\widetilde{G}$ is a $(1 \pm \epsilon)$ spectral sparsifier of G.*

Proof. By Claim 11.4, it suffices to simply compute a $(1 \pm \epsilon)$ code sparsifier of H, where H is the generating matrix induced by S. By invoking Theorem 5.19, this can be done in the stated time, yielding the above theorem. □

11.3 Efficient Hedge-graph Sparsification

In this section, we detail the procedure of creating efficient $(1 \pm \epsilon)$ hedge-graph cut sparsifiers. To start, we recap the definition of a hedge-graph.

Definition 11.5. *A hedge-graph $G = (V, E)$ is defined by a set of vertices V and a set of hedges E. Each hedge E is itself a collection of edges $\in \binom{V}{2}$. For a subset of vertices $S \subseteq V$, we say that a hedge e is cut by S if there exists an edge $f \in e$ such that f is cut by S. We define cut-sizes globally by*

$$cut_G(S) = \sum_{e \in E} \mathbf{1}[e \text{ is cut by } S].$$

As remarked in prior work (see [GKP17]) hedge-graph cuts behave very differently from graph (or hypergraph) cuts. In particular, there exist hedge-graphs with respect to which the cut function is not submodular. In this same work [GKP17] it was remarked that uniform random sampling at a rate roughly equal to the reciprocal of the minimum hedge-cut *does not* preserve all the cuts in the sparsifier. Thus, an analysis mimicking [BK96] for creating cut-sparsifiers of hedge-graphs. Nevertheless, we show here that the code sparsification framework is sufficiently general such that one can derive efficient sparsifications of many hedge-graphs. We formalize this below:

Definition 11.6. *Given a hedge-graph G, we say that a re-weighted sub-hedge-graph $\widetilde{G}$ is a $(1 \pm \epsilon)$ cut-sparsifier of G if $\forall S \subseteq V$:*

$$(1-\epsilon) cut_G(S) \leq cut_{\widetilde{G}}(S) \leq (1+\epsilon) cut_G(S).$$

We will also use the following observation in our discussion of hedge-graph sparsifiers:

Fact 11.6. Each hedge $e \in E$ induces a set of connected components, which we denote $\mathcal{P}_e$, that partitions the vertex set V. The hedge e is cut by a set $S \subseteq V$ *if and only if* one of the constituent connected components $C_i \in \mathcal{P}_e$ is cut by the set S. Formally, e is cut if and only if $\exists C_i \in \mathcal{P}_\urcorner : C_i \cap S \neq \emptyset \wedge C_i \cap \bar{S} \neq \emptyset$.

Going forward, we also use R_e to denote the number of connected components in $\mathcal{P}_e$ which are of size ≥ 2. With this, we are able to state our main theorem:

Theorem 11.7. *Let G be a hedge-graph on n vertices, let $\epsilon \in (0,1)$, and let $R = \max_e R_e$. Then, there is a randomized polynomial time algorithm for computing a $(1 \pm \epsilon)$ cut-sparsifier $\widetilde{G}$ of G (with high probability), such that $\widetilde{G}$ preserves only $\widetilde{O}(nR^2/\epsilon^2)$ re-weighted hedges.*

Remark 11.8. *Note that R is separate from the size of the hedge. In particular, a hyperedge is a hedge where $R = 1$.*

Proof. First, let us choose a prime $p \in [n, 2n]$. The proof proceeds by creating a code over $\mathbb{F}_{p^R}$. This means that the generating matrix H we create is $\in \mathbb{F}_{p^R}^{m \times n}$. Now, for any hedge $e \in E$, let us use $C_1, \ldots C_R$ to denote its connected components of size ≥ 2. Our goal will be to write the cut-function of G as the OR of a cut-condition of each of the individual components. Then, observe that we are exactly implementing the logic of a hedge-cut; namely, a hedge is cut if and only if some constituent component of the hedge is cut.

Next, for the field $\mathbb{F}_{p^R}$, recall that we create this field by extending the field $\mathbb{F}_p$. We let $\alpha_1, \ldots \alpha_{R-1}$ denote the roots we use to extend the field. An equivalent way to view each element x in $\mathbb{F}_{p^R}$ is as a linear combination of the α_i's:

$$x = b_0 + b_1\alpha_1 + \cdots + b_{R-1}\alpha_{R-1},$$

where $b_i \in \mathbb{F}_p$.

In particular, an element x is non-zero *if and only if* $\exists b_i$ in the above representation such that $b_i \neq 0$. Thus gives us a natural way to express the cut function of each hedge, as the α_i's can essentially simulate an OR of each individual component being cut. So, before concluding, our final ingredient is an expression which evaluates to 0 if and only if a certain component is not cut. Indeed let C_i denote the component, and let v^* denote the largest label of vertex in C_i. We can write:

$$\mathbf{1}[C_i \text{ is cut}] = \mathbf{1}[(\sum_{v \in C_i} x_v) - |C_i| \cdot x_{v^*} \neq 0] = \mathbf{1}[f_{C_i}(x) \neq 0] = .$$

One can verify that for $x = \mathbf{1}_S \in \{0,1\}^n$, the above expression exactly captures whether the cut S splits the component C_i. To conclude then, for the hedge e, we simply include a row in the generating matrix of the form:

$$\sum_{i=0}^{\leq R-1} \alpha_i \cdot f_{C_{i-1}}(x).$$

By the above logic,

$$\sum_{i=0}^{\leq R-1} \alpha_i \cdot f_{C_{i-1}}(\mathbf{1}_S) = \mathbf{1}[e \text{ is cut}].$$

In particular after creating the generating matrix H in this manner, we can invoke Theorem 5.19 to create a $(1 \pm \epsilon)$ code-sparsifier of H with only $\widetilde{O}(nR^2/\epsilon^2)$ re-weighted rows remaining. This sparsifier must preserve codeword weights for any vector $x \in \{0,1\}^n$, and hence the same selection of re-weighted hedges would constitute a $(1 \pm \epsilon)$ cut-sparsifier of the hedge-graph G. This yields the claim. □

11.4 Efficient Cardinality-based Splitting Function Sparsification

First, let us recall the notion of generalized hypergraph sparsification:

Definition 11.7. *For a hypergraph $G = (V, E)$, a general hypergraph associates a splitting function $g_e : 2^e \to \mathbb{R}^+$ to each hyperedge $e \in E$. For a set $S \subseteq V$, we define*

$$cut_G(S) = \sum_{e \in E} g_e(S \cap e).$$

We say a re-weighted sub-hypergraph of G is a $(1 \pm \epsilon)$ cut-sparsifier of $\widetilde{G}$ if for all $S \subseteq V$:

$$(1 - \epsilon)cut_G(S) \leq \sum_{e \in \widetilde{e}} \widetilde{w}_e \cdot g_e(S \cap e) \leq (1 + \epsilon)cut_G(S).$$

In the general hypergraph sparsification literature, one particularly important class of splitting functions are the so-called "cardinality-based splitting function".

Definition 11.8. *For a hyperedge $e \in E$, we say that $g_e : 2^e \to \mathbb{R}^+$ is a cardinality-based splitting function if there exists $f_e : \mathbb{Z} \to \mathbb{R}^+$ such that $\forall S \subseteq V$, $g_e(S \cap e) = f(|S \cap e|)$ (i.e., there is a function f which depends only on the cardinality of the input set which dictates the value of the splitting function).*

Beyond this, we call a splitting function a $\{0,1\}$-valued cardinality-based splitting function if g_e is cardinality-based and maps to the range $\{0,1\}$.

In this regime, we derive the following theorem:

Theorem 11.9. *Let G be a general hypergraph, and for every $e \in E$, suppose that g_e is the same $\{0,1\}$-valued, cardinality-based splitting function. Then, G is efficiently-sparsifiable to $\widetilde{O}(n/\epsilon^2)$ hyperedges if and only if f_e is periodic.*

Here, we are using f_e in the same way as defined in Definition 11.8.

Proof. Fix a hyperedge $e \in E$, and recall that for $S \subseteq V$, we have $g_e(S) = f_e(|e \cap S|) \in \{0,1\}$. In particular, because every hyperedge has the same splitting function, we can simply use $f = f_e$ for every $e \in E$. Note that this also necessarily means that every hyperedge is of the same arity, which we denote by r.

The key observation is that this general hypergraph is now equivalent to a CSP. Indeed, for any hyperedge $e \in E$, there is a corresponding constraint $f : \{0,1\}^r \to \{0,1\}$ operating on the variables $(y_1, \dots y_r)$ in e. For any cut $S \subseteq V$, and any assignment to the variables $x = \mathbf{1}_S$, we have that $g_e(S) = 1$ if and only if $f((\mathbf{1}_S)_{e_1}, \dots (\mathbf{1}_S)_{e_r}) = 1$.

Now, we can simply invoke Theorem 1.3 to conclude the stated theorem. The efficiency follows from the same theorem. □

12 Acknowledgements

We would like to thank Swastik Kopparty for supplying us with the proof of Claim E.2 and Srikanth Srinivasan for introducing us to the notion of polynomials over groups as used in Theorem 1.9.

References

[AK95] Charles J Alpert and Andrew B Kahng. Recent directions in netlist partitioning: a survey. *Integration*, 19(1):1–81, 1995.

[BBR94] David A. Mix Barrington, Richard Beigel, and Steven Rudich. Representing Boolean functions as polynomials modulo composite numbers. *Comput. Complex.*, 4:367–382, 1994.

[BDKS16] Grey Ballard, Alex Druinsky, Nicholas Knight, and Oded Schwartz. Hypergraph partitioning for sparse matrix-matrix multiplication. *ACM Trans. Parallel Comput.*, 3(3):18:1–18:34, 2016.

[BG24] Joshua Brakensiek and Venkatesan Guruswami. Redundancy is all you need. *Private communication*, 2024.

[BK96] András A. Benczúr and David R. Karger. Approximating s-t minimum cuts in $\tilde{O}(n^2)$ time. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 47–55. ACM, 1996.

[BSS09] Joshua D. Batson, Daniel A. Spielman, and Nikhil Srivastava. Twice-Ramanujan sparsifiers. In Michael Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 255–262. ACM, 2009.

[BZ20] Silvia Butti and Stanislav Zivný. Sparsification of binary CSPs. *SIAM J. Discret. Math.*, 34(1):825–842, 2020.

[CKN20] Yu Chen, Sanjeev Khanna, and Ansh Nagda. Near-linear size hypergraph cut sparsifiers. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 61–72. IEEE, 2020.

[DMS03] Ilya Dumer, Daniele Micciancio, and Madhu Sudan. Hardness of approximating the minimum distance of a linear code. *IEEE Transactions on Information Theory*, 49(1):22–37, January 2003. Preliminary version in FOCS 1999.

[Efr12] Klim Efremenko. 3-query locally decodable codes of subexponential length. *SIAM J. Comput.*, 41(6):1694–1703, 2012.

[FGK+24] Fedor V Fomin, Petr A Golovach, Tuukka Korhonen, Daniel Lokshtanov, and Saket Saurabh. Fixed-parameter tractability of hedge cut. *arXiv preprint arXiv:2410.17641*, 2024.

[FHHP11] Wai Shing Fung, Ramesh Hariharan, Nicholas J.A. Harvey, and Debmalya Panigrahi. A general framework for graph sparsification. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, STOC '11, page 71–80, New York, NY, USA, 2011. Association for Computing Machinery.

[FK17] Arnold Filtser and Robert Krauthgamer. Sparsification of two-variable valued constraint satisfaction problems. *SIAM J. Discret. Math.*, 31(2):1263–1276, 2017.

[GK10] Ryan Gomes and Andreas Krause. Budgeted nonparametric learning from data streams. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 391–398. Omnipress, 2010.

[GKP17] Mohsen Ghaffari, David R. Karger, and Debmalya Panigrahi. Random contractions and sampling for hypergraph and hedge connectivity. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1101–1114. SIAM, 2017.

[Gro97] Vince Grolmusz. On set systems with restricted intersections modulo a composite number. In Tao Jiang and D. T. Lee, editors, *Computing and Combinatorics, Third Annual International Conference, COCOON '97, Shanghai, China, August 20-22, 1997, Proceedings*, volume 1276 of *Lecture Notes in Computer Science*, pages 82–90. Springer, 1997.

[JLLS23] Arun Jambulapati, James R. Lee, Yang P. Liu, and Aaron Sidford. Sparsifying sums of norms. *CoRR*, abs/2305.09049, 2023.

[JLM+23] Lars Jaffke, Paloma T. Lima, Tomás Masarík, Marcin Pilipczuk, and Uéverton S. Souza. A tight quasi-polynomial bound for global label min-cut. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 290–303. SIAM, 2023.

[JLS23] Arun Jambulapati, Yang P. Liu, and Aaron Sidford. Chaining, group leverage score overestimates, and fast spectral hypergraph sparsification. In Barna Saha and Rocco A. Servedio, editors, *Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, Orlando, FL, USA, June 20-23, 2023*, pages 196–206. ACM, 2023.

[Kar93] David R. Karger. Global min-cuts in rnc, and other ramifications of a simple min-cut algorithm. In Vijaya Ramachandran, editor, *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, pages 21–30. ACM/SIAM, 1993.

[KK15] Dmitry Kogan and Robert Krauthgamer. Sketching cuts in graphs and hypergraphs. In Tim Roughgarden, editor, *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science, ITCS 2015, Rehovot, Israel, January 11-13, 2015*, pages 367–376. ACM, 2015.

[KK23] Yotam Kenneth and Robert Krauthgamer. Cut sparsification and succinct representation of submodular hypergraphs. *CoRR*, abs/2307.09110, 2023.

[KPS24] Sanjeev Khanna, Aaron Putterman, and Madhu Sudan. Code sparsification and its applications. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 5145–5168. SIAM, 2024.

[Law73] Eugene L. Lawler. Cutsets and partitions of hypergraphs. *Networks*, 3(3):275–285, 1973.

[LB11] Hui Lin and Jeff A. Bilmes. A class of submodular functions for document summarization. In Dekang Lin, Yuji Matsumoto, and Rada Mihalcea, editors, *The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Conference, 19-24 June, 2011, Portland, Oregon, USA*, pages 510–520. The Association for Computer Linguistics, 2011.

[Lee23] James R. Lee. Spectral hypergraph sparsification via chaining. In Barna Saha and Rocco A. Servedio, editors, *Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, Orlando, FL, USA, June 20-23, 2023*, pages 207–218. ACM, 2023.

[LM17] Pan Li and Olgica Milenkovic. Inhomogeneous hypergraph clustering with applications. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 2308–2318, 2017.

[LM18] Pan Li and Olgica Milenkovic. Submodular hypergraphs: p-Laplacians, cheeger inequalities and spectral clustering. In Jennifer G. Dy and Andreas Krause, editors,

Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018, volume 80 of *Proceedings of Machine Learning Research*, pages 3020–3029. PMLR, 2018.

[LVS+21] Meng Liu, Nate Veldt, Haoyu Song, Pan Li, and David F. Gleich. Strongly local hypergraph diffusions for clustering and semi-supervised learning. In Jure Leskovec, Marko Grobelnik, Marc Najork, Jie Tang, and Leila Zia, editors, *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, pages 2092–2103. ACM / IW3C2, 2021.

[Mic14a] Daniele Miccancio. The dual lattice. pages 1–10, 2014. Reading material for *CSE 206A: Lattice Algorithms and Applications*. https://cseweb.ucsd.edu/classes/sp14/cse206A-a/lec3.pdf.

[Mic14b] Daniele Miccancio. Point lattices. pages 1–13, 2014. Reading material for *CSE 206A: Lattice Algorithms and Applications*. https://cseweb.ucsd.edu/classes/sp14/cse206A-a/lec1.pdf.

[Pin10] Charles C. Pinter. *A book of abstract algebra*. Dover books on mathematics. Dover Publications, Mineola, N.Y., dover ed. edition, 2010.

[ST11] Daniel A. Spielman and Shang-Hua Teng. Spectral sparsification of graphs. *SIAM J. Comput.*, 40(4):981–1025, 2011.

[TIWB14] Sebastian Tschiatschek, Rishabh K. Iyer, Haochen Wei, and Jeff A. Bilmes. Learning mixtures of submodular functions for image collection summarization. In Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 1413–1421, 2014.

[Var97] Alexander Vardy. The intractability of computing the minimum distance of a code. *IEEE Trans. Inf. Theory*, 43(6):1757–1766, 1997.

[VBK20] Nate Veldt, Austin R. Benson, and Jon M. Kleinberg. Minimizing localized ratio cut objectives in hypergraphs. In Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash, editors, *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pages 1708–1718. ACM, 2020.

[VBK21] Nate Veldt, Austin R. Benson, and Jon M. Kleinberg. Approximate decomposable submodular function minimization for cardinality-based components. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 3744–3756, 2021.

[VBK22] Nate Veldt, Austin R. Benson, and Jon M. Kleinberg. Hypergraph cuts with general splitting functions. *SIAM Rev.*, 64(3):650–685, 2022.

[Wei] Eric W. Weisstein. Euclidean algorithm. From *MathWorld* – A Wolfram Web Resource. https://mathworld.wolfram.com/EuclideanAlgorithm.html.

[YNY+19] Naganand Yadati, Madhav Nimishakavi, Prateek Yadav, Vikram Nitin, Anand Louis, and Partha P. Talukdar. Hypergcn: A new method for training graph convolutional networks on hypergraphs. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 1509–1520, 2019.

[ZHS06] Dengyong Zhou, Jiayuan Huang, and Bernhard Schölkopf. Learning with hypergraphs: Clustering, classification, and embedding. In Bernhard Schölkopf, John C. Platt, and Thomas Hofmann, editors, *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006*, pages 1601–1608. MIT Press, 2006.

[ZLS22] Yu Zhu, Boning Li, and Santiago Segarra. Hypergraph 1-spectral clustering with general submodular weights. In *56th Asilomar Conference on Signals, Systems, and Computers, ACSSC 2022, Pacific Grove, CA, USA, October 31 - Nov. 2, 2022*, pages 935–939. IEEE, 2022.

A Detailed Proof of Sparsifiers for Abelian Codes

A.1 Efficient Spanning Subsets for Abelian Codes

Here, we re-produce the algorithms used to create spanning subsets for codes over $\mathbb{Z}_q$ in the new setting of $(\mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_u})$. For a code $\mathcal{C} \subseteq (\mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_u})^m$, the following algorithms take as input a generating matrix $H \in (\mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_u})^{m \times n}$, and construct maximum spanning subsets. Note that these maximum spanning subsets are defined in the same manner as before. Going forward, we will let $q = \prod_{i=1}^{u} q_i$.

Algorithm 15: BuildMaxSpanningSubsetAbelian(H)

Initialize $T = \emptyset$.
Let $k = 0$.
for $i \in [m]$ **do**
  If the number of distinct codewords in the span of $H|_{T \cup \{i\}}$ is $\geq k$, then set $T = T \cup \{i\}$, and $k \leftarrow$ the number of distinct codewords in the span of $H|_{T \cup \{i\}}$.
end
return T

Algorithm 16: ConstructSpanningSubsetsAbelian(H, t)

for $i \in [t]$ **do**
  Let $T_i = \text{BuildMaxSpanningSubsetAbelian}(H|_{\bar{T_1} \cap \ldots T_{i-1}^{-}})$.
end
return $T_i : i \in [t]$.

Note that, as before, the correctness and efficient implementation of these algorithms follows from exactly the same reasoning as with codes over $\mathbb{Z}_q$, as these proofs rely on the contraction algorithm which has a direct analog. In particular, we are able to conclude the following:

Claim A.1. *For a generating matrix $G \in (\mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_u})^{m \times n}$, any choice of d and the set S of bad rows (i.e. those guaranteed by Corollary 6.5) for G and parameter d, for any disjoint maximum spanning subsets $T_1, \ldots T_{2d(\log(n)+\log(q))}$, we have that $S \subseteq T_1 \cup T_2 \cdots \cup T_{2d(\log(n)+\log(q))}$.*

A.2 Weighted Decomposition

Here, we introduce an analog of the weight decomposition step used for codes over $\mathbb{Z}_q$ which will instead be defined for codes over $(\mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \ldots \mathbb{Z}_{q_u})$. Consider the following procedure which operates on a code of length m and at most q^n distinct codewords in Algorithm 17.

Algorithm 17: WeightClassDecomposition$(\mathcal{C}, \epsilon, \alpha)$

Let E_i be all coordinates of $\mathcal{C}'$ that have weight between $[\alpha^{i-1}, \alpha^i]$.
Let $\mathcal{D}_{\text{odd}} = E_1 \cup E_3 \cup E_5 \cup \ldots$, and let $\mathcal{D}_{\text{even}} = E_2 \cup E_4 \cup E_6 \cup \ldots$.
return $\mathcal{D}_{odd}, \mathcal{D}_{even}$.

Next, we prove some facts about this algorithm.

Lemma A.2. *Consider a code $\mathcal{C}$ with at most q^n distinct codewords and length n. Let*

$$\mathcal{D}_{odd}, \mathcal{D}_{even} = \text{WeightClassDecomposition}(\mathcal{C}, \epsilon, n).$$

To get a $(1 \pm \epsilon)$-sparsifier for $\mathcal{C}$, it suffices to get a $(1 \pm \epsilon)$ sparsifier to each of $\mathcal{D}_{odd}, \mathcal{D}_{even}$.

Proof. The creation of $\mathcal{D}_{\text{odd}}, \mathcal{D}_{\text{even}}$ forms a *vertical* decomposition of the code $\mathcal{C}'$. Thus, by Claim 3.1, if we have a $(1 \pm \epsilon)$ sparsifier for each of $\mathcal{D}_{\text{odd}}, \mathcal{D}_{\text{even}}$, we have a $(1 \pm \epsilon)$ sparsifier to $\mathcal{C}$. □

Because of the previous claim, it is now our goal to create sparsifiers for $\mathcal{D}_{\text{odd}}, \mathcal{D}_{\text{even}}$. Without loss of generality, we will focus our attention only on $\mathcal{D}_{\text{even}}$, as the procedure for $\mathcal{D}_{\text{odd}}$ is exactly the same (and the proofs will be the same as well). At a high level, we will take advantage of the fact that

$$\mathcal{D}_{\text{even}} = E_2 \cup E_4 \cup \ldots,$$

where each E_i contains edges of weights $[\alpha^{i-1}, \alpha^i]$, for $\alpha = \frac{m^3}{\epsilon^3}$, where m is the length of the code. Because of this, whenever a codeword $c \in \mathcal{C}'$ is non-zero in a coordinate in E_i, we can effectively ignore all coordinates of lighter weights $E_{i-2}, E_{i-4}, \ldots$. This is because any coordinate in $E_{\leq i-2}$ has weight at most a $\frac{\epsilon^3}{m^3}$ fraction of any single coordinate in E_i. Because there are at most $O(m)$ coordinates in $\mathcal{C}'$, it follows that the total possible weight of all rows in $E_{\leq i-2}$ is still at most a $O(\epsilon/m)$ fraction of the weight of a single row in E_i. Thus, we will argue that when we are creating a sparsifier for codewords that are non-zero in a row in E_i, we will be able to effectively ignore all rows corresponding to $E_{\leq i-2}$. Thus, our decomposition is quite simple: we first restrict our attention to E_i and create a $(1 \pm \epsilon)$ sparsifier for these rows. Then, we transform the remaining code such that only codewords which are all zeros on E_i remain. We present this transformation below:

Claim A.3. *If the span of G originally had $2^{n'}$ distinct codewords, and the span of $G|_{E_i}$ has $2^{n''}$ distinct codewords, then after Algorithm 18, the span of $G'|_{\bar{E}_i}$ has $2^{n'-n''}$ distinct codewords.*

Proof. After running the above algorithm, G' is entirely 0 on the rows corresponding to E_i, hence it follows that after running the algorithm, G' and $G'|_{\bar{E}_i}$ have the same number of distinct codewords.

Algorithm 18: SingleSpanDecomposition($\mathcal{D}_{\text{even}}, \alpha, i$)

Let E_i be all rows of $\mathcal{D}_{\text{even}}$ with weights between α^{i-1} and α^i.
Let G be a generating matrix for $\mathcal{D}_{\text{even}}$.
Store $G|_{E_i}$.
Let $G' = G$.
while $G'|_{E_i}$ *is not all zero* **do**
    Find the first non-zero coordinate of $G'|_{E_i}$, call this j.
    Set $G' = \text{ContractAbelian}(G', j)$.
end
return $G|_{E_i}$, $G'|_{\bar{E}_i}$

Now, we will argue that the span of G' has at most $2^{n'-n''}$ distinct codewords. Indeed, for any codeword c in the span of G', c is also in the span of the original G. However, for this same c, in the original G we could add any of the $2^{n'}$ distinct vectors which are non-zero on the rows corresponding to E_i. Thus, the span of G must have at least $2^{n'}$ times as many distinct codewords as G'. This concludes the claim. □

Claim A.4. *For any codeword c in the span of G, if c is zero in the coordinates corresponding to E_i, then c is still in the span of G' after the contractions of Algorithm 18.*

Proof. This follows from Claim 6.2. If a codeword is 0 in a coordinate which we contract on, then it remains in the span. Hence, if we denote by c a codeword which is zero in all of the coordinates of E_i, then c is still in the span after contracting on the coordinates in E_i. □

Claim A.5. *In order to get a $(1 \pm \epsilon)$ approximation to $\mathcal{D}_{even}$, it suffices to combine a $(1 \pm \epsilon/2)$ approximation to $G|_{E_i}$ and a $(1 \pm \epsilon)$ approximation to $G'|_{\bar{E}_i}$.*

Proof. For any codeword $c \in \mathcal{D}_{\text{even}}$ which is non-zero on rows E_i, it suffices to get a $(1 \pm \epsilon/2)$ approximation to their weight on $G|_{E_i}$, as this makes up at least a $(1 \pm \epsilon/n)$ fraction of the overall weight of the codeword.

For any codeword $c \in \mathcal{D}_{\text{even}}$ which is zero on rows E_i, then c is still in the span of G', and in particular, its weight when generated by G is exactly the same as its weight in $G'_{\bar{E}_i}$ (as it is zero in the coordinates corresponding to E_i, we can ignore these coordinates). Hence, it suffices to get a $(1 \pm \epsilon)$ approximation to the weight of c on $G'_{\bar{E}_i}$. Taking the union of these two sparsifiers will then yield a sparsifier for every codeword in the span of $\mathcal{D}_{\text{even}}$. □

Claim A.6. *Let S, H_i be as returned by Algorithm 19. Then, $\sum_{i \in S} \log(|Span(H_i)|) = \log(|Span(\mathcal{D}_{even})|)$.*

Proof. This follows because from line 5 of Algorithm 19. In each iteration, we store $G|_{E_i}$, and iterate on $G'|_{E_i}$. From Claim A.3, we know that

$$(\text{number of distinct codewords in } G|_{E_i}) \cdot (\text{number of distinct codewords in } G'|_{\bar{E}_i})$$

$$= (\text{number of distinct codewords in } G),$$

thus taking the log of both sides, we can see that the sum of the logs of the number of distinct codewords is preserved. □

Algorithm 19: SpanDecomposition($\mathcal{D}_{\text{even}}, \alpha$)

Let $\mathcal{D}'_{\text{even}} = \mathcal{D}_{\text{even}} = E_2 \cup E_4 \cup E_6 \dots$.
Let $S = \{\}$.
while $\mathcal{D}'_{even}$ *is not empty* **do**
    Let i be the largest integer such that E_i is non-empty in $\mathcal{D}'_{\text{even}}$.
    Let $G|_{E_i}, G'|_{\bar{E}_i} = \text{SingleSpanDecomposition}(\mathcal{D}'_{\text{even}}, \alpha, i)$.
    Let $\mathcal{D}'_{\text{even}}$ be the span of $G'|_{\bar{E}_i}$, and let $H_i = G|_{E_i}$.
    Add i to S.
end
return S, H_i *for every* $i \in S$

Lemma A.7. *Suppose we have a code of the form $\mathcal{D}_{even}$ created by Algorithm 17. Then, if we run Algorithm 19 on $\mathcal{D}_{even}$, to get $S, (H_i)_{i \in S}$, it suffices to get a $(1 \pm \epsilon/2)$ sparsifier for each of the H_i in order to get a $(1 \pm \epsilon)$ sparsifier for $\mathcal{D}_{even}$.*

Proof. This follows by inductively applying Claim A.5. Let our inductive hypothesis be that getting a $(1 \pm \epsilon/2)$ sparsifier to each of codes returned of Algorithm 19 suffices to get a $(1 \pm \epsilon)$ sparsifier to the code overall. We will induct on the number of recursive levels that Algorithm 19 undergoes (i.e., the number of distinct codes returned by the algorithm). In the base case, we assume that there is only one level of recursion, and that Algorithm 19 simply returns a single code. Clearly then, getting a $(1 \pm \epsilon/2)$ sparsifier to this code suffices to sparsify the code overall.

Now, we prove the claim inductively. Assume the algorithm returns ℓ codes. After the first iteration, we decompose $\mathcal{D}_{\text{even}}$ into $H_i = G|_{E_i}$ and $G'|_{\bar{E}_i}$. By Claim A.5, it suffices to get a $(1 \pm \epsilon/2)$ sparsifier to H_i, while maintaining a $(1 \pm \epsilon)$ sparsifier to $G'|_{\bar{E}_i}$. By invoking our inductive claim, it then suffices to get a $(1 \pm \epsilon/2)$ sparsifier for the $\ell - 1$ codes returned by the algorithm on $G'|_{\bar{E}_i}$. Thus, we have proved our claim. □

A.3 Dealing with Bounded Weights

Let us consider any H_i that is returned by Algorithm 19, when called with $\alpha = m^3/\epsilon^3$. By construction, H_i will contain weights only in the range $[\alpha^{i-1}, \alpha^i]$ and will have at most $O(m)$ coordinates. In this subsection, we will show how we can turn H_i into an unweighted code with at most $\text{poly}(m/\epsilon)$ coordinates. First, note however, that we can simply pull out a factor of α^{i-1}, and treat the remaining graph as having weights in the range of $[1, \alpha]$. Because multiplicative approximation does not change under multiplication by a constant, this is valid. Formally, consider the following algorithm:

Algorithm 20: MakeUnweighted($\mathcal{C}, \alpha, i, \epsilon$)

Divide all edge weights in $\mathcal{C}$ by α^{i-1}.
Make a new unweighted code $\mathcal{C}'$ by duplicating every coordinate j of $\mathcal{C}$ $\lfloor 10w(j)/\epsilon \rfloor$ times.
return $\mathcal{C}', \alpha^{i-1} \cdot \frac{\epsilon}{10}$

Lemma A.8. *Consider a code $\mathcal{C}$ with weights bounded in the range $[\alpha^{i-1}, \alpha^i]$. To get a $(1 \pm \epsilon)$ sparsifier for $\mathcal{C}$ it suffices to return a $(1 \pm \epsilon/10)$ sparsifier for $\mathcal{C}' = \text{MakeUnweighted}(\mathcal{C}, \alpha, i, \epsilon)$ weighted by $\alpha^{i-1} \cdot \frac{\epsilon}{10}$.*

Proof. It suffices to show that $\mathcal{C}'$ is $(1 \pm \epsilon/10)$ sparsifier for $\mathcal{C}$, as our current claim will then follow by Claim 3.2 (composing approximations). Now, to show that $\mathcal{C}'$ is $(1 \pm \epsilon/10)$ sparsifier for $\mathcal{C}$, we will use Claim 3.1 (vertical decomposition of a code), and show that in fact the weight contributed by every coordinate in $\mathcal{C}$ is approximately preserved by the copies of the coordinate introduced in $\mathcal{C}'$.

Without loss of generality, let us assume that $i = 1$, as otherwise pulling out the factor of α^{i-1} in the weights clearly preserves the weights of the codewords. Indeed, for every coordinate j in $\mathcal{C}$, let $w(j)$ be the corresponding weight on this coordinate, and consider the corresponding $\lfloor 10w(r)/\epsilon \rfloor$ coordinates in $\mathcal{C}'$. We will show that the contribution from these coordinates in $\mathcal{C}'$, when weighted by $\epsilon/10$, is a $(1 \pm \epsilon/10)$ approximation to the contribution from j.

So, consider an arbitrary coordinate j, and let its weight be $w(j)$. Then,

$$\frac{10w}{\epsilon} - 1 \leq \lfloor 10w(j)/\epsilon \rfloor \leq \frac{10w}{\epsilon}.$$

When we normalize by $\frac{\epsilon}{10}$, we get that the combined weight of the new coordinates w' satisfies

$$w - \epsilon/10 \leq w' \leq w.$$

Because $w \geq 1$, it follows that this yields a $(1 \pm \epsilon/10)$ sparsifier, and we can conclude our statement. □

Claim A.9. *Suppose a code $\mathcal{C}$ of length m has weight ratio bounded by α, and minimum weight α^{i-1}. Then, MakeUnweighted$(\mathcal{C}, \alpha, i, \epsilon)$ yields a new unweighted code of length $O(m\alpha/\epsilon)$.*

Proof. Each coordinate is repeated at most $O(\alpha/\epsilon)$ times. □

A.4 Sparsifiers for Codes of Polynomial Length

In this section, we introduce an efficient algorithm for sparsifying codes. We will take advantage of the decomposition proved in Corollary 4.5 in conjunction with the following claim:

Claim A.10. *Suppose $\mathcal{C}$ is a code with at most q^n distinct codewords over $(\mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \ldots \mathbb{Z}_{q_u})$ (where $q = q_1 \cdot \ldots q_u$), and let $b \geq 1$ be an integer such that for any integer $\alpha \geq 1$, the number of codewords of weight $\leq \alpha b$ is at most $\binom{n \log(q)}{\alpha} \cdot q^{\alpha+1} \leq (qn)^{2\alpha}$. Suppose further that the minimum distance of the code $\mathcal{C}$ is b. Then, sampling the ith coordinate of $\mathcal{C}$ at rate $p_i \geq \frac{\log(n)\log(q)\eta}{b\epsilon^2}$ with weights $1/p_i$ yields a $(1 \pm \epsilon)$ sparsifier with probability $1 - 2^{-(0.19\eta - 110)\log n} \cdot n^{-101}$.*

Proof. Consider any codeword c of weight $[\alpha b/2, \alpha b]$ in $\mathcal{C}$. We know that there are at most $(qn)^\alpha$ codewords that have weight in this range. The probability that our sampling procedure fails to preserve the weight of c up to a $(1 \pm \epsilon)$ fraction can be bounded by Claim 3.3. Indeed,

$$\Pr[\text{fail to preserve weight of } c] \leq 2e^{-0.38 \cdot \epsilon^2 \cdot \frac{\alpha b}{2} \cdot \frac{\eta \log(n)\log(q)}{\epsilon^2 b}} = 2e^{-0.19\alpha\eta \log(n)\log(q)}.$$

Now, let us take a union bound over the at most $(qn)^{2\alpha}$ codewords of weight between $[\alpha b/2, \alpha]$. Indeed,

$$\begin{aligned}
\Pr[\text{fail to preserve any } c \text{ of weight } [\alpha b/2, \alpha b]] &\leq 2^{2\alpha \log(qn)} \cdot 2e^{-0.19\alpha\eta \log(n)\log(q)} \\
&\leq 2^{\alpha \cdot (-0.19\eta + 2)\log(n)\log(q)} \\
&\leq 2^{\alpha \cdot (-0.19\eta + 2)\log(n)} \\
&\leq 2^{-(0.19\eta - 110)\alpha \log n} \cdot 2^{-108\alpha \log n} \\
&\leq 2^{-(0.19\eta - 110)\log n} \cdot n^{-108\alpha},
\end{aligned}$$

where we have chosen η to be sufficiently large. Now, by integrating over $\alpha \geq 1$, we can bound the failure probability for any integer choice of α by $2^{-(0.19\eta-110)\log n} \cdot n^{-101}$. □

Next, we consider Algorithm 21:

Algorithm 21: CodeDecomposition$(\mathcal{C}, d)$

Let T be $\cup_i T_i$ for T_i the sets of coordinates returned by ConstructSpanningSubsetsAbelian$(\mathcal{C}, 2d(\log(n) + \log(q)))$.
Let $\mathcal{C}'$ be the code $\mathcal{C}$ after removing the set of coordinates T.
return $T, \mathcal{C}'$

Intuitively, the set T returned by Algorithm 21 contains all of the "bad" rows which were causing the violation of the codeword counting bound. We know that if we removed *only* the true set of bad rows, denoted by S, then we could afford to simply sample the rest of the code at rate roughly $1/d$ while preserving the weights of all codewords. Thus, it remains to show that when we remove T (a superset of S) that this property still holds. More specifically, we will consider the following algorithm:

Algorithm 22: CodeSparsify$(\mathcal{C}, n, \epsilon, \eta)$

Let m be the length of $\mathcal{C}$.
if $m \leq 100 \cdot n \cdot \eta \log^2(n) \log^2(q)/\epsilon^2$ **then**
  return $\mathcal{C}$
end
Let $d = \frac{m\epsilon^2}{2\eta \cdot n \log^2(n) \log^2(q)}$.
Let $T, \mathcal{C}' =$ CodeDecomposition$(\mathcal{C}, \sqrt{d} \cdot \eta \cdot \log(n) \log(q)/\epsilon^2)$. Let $\mathcal{C}_1 = \mathcal{C}|_T$. Let $\mathcal{C}_2$ be the result of sampling every coordinate of $\mathcal{C}'$ at rate $1/\sqrt{d}$.
return CodeSparsify$(\mathcal{C}_1, n, \epsilon, \eta) \cup \sqrt{d} \cdot$ CodeSparsify$(\mathcal{C}_2, n, \epsilon, \eta)$

Lemma A.11. *In Algorithm 22, starting with a code $\mathcal{C}$ of size $2dn \log^2(n) \log^2(q)/\epsilon^2$, after i levels of recursion, with probability $1 - 2^i \cdot 2^{-\eta n}$, the code being sparsified at level i, $\mathcal{C}^{(i)}$ has at most*

$$2(1 + 1/2 \log\log(n))^i \cdot d^{1/2^i} \cdot \eta \cdot n \log^2(n) \log^2(q)/\epsilon^2$$

surviving coordinates.

Proof. Let us prove the claim inductively. For the base case, note that in the 0th level of recursion the number of surviving coordinates in $\mathcal{C}^{(0)} = \mathcal{C}$ is $d \cdot 2n \log^2(n) \log^2(q)/\epsilon^2$, so the claim is satisfied trivially.

Now, suppose the claim holds inductively. Let $\mathcal{C}^{(i)}$ denote a code that we encounter in the ith level of recursion, and suppose that it has at most

$$2(1 + 1/2 \log\log(n))^i \cdot d^{1/2^i} \cdot \eta \cdot n \log^2(n) \log^2(q)/\epsilon^2$$

coordinates. Denote this number of coordinates by ℓ. Now, if this number is smaller than $100n\eta \log^2(n) \log^2(q)/\epsilon^2$, we will simply return this code, and there will be no more levels of recursion, so our claim holds vacuously. Instead, suppose that this number is larger than $100n\eta \log^2(n) \log^2(q)/\epsilon^2$, and let $d' = \frac{\ell\epsilon^2}{2\eta n \log^2(n) \log^2(q)} \leq (1 + 1/2 \log\log(n))^i \cdot d^{1/2^i}$.

Then, we decompose $\mathcal{C}^{(i)}$ into two codes, $\mathcal{C}_1$ and $\mathcal{C}_2$. $\mathcal{C}_1$ is the restriction of $\mathcal{C}$ to the set of disjoint maximum spanning subsets. By construction, we know that T is constructed by calling ConstructSpanningSubsetsAbelian with parameter $\sqrt{d'}\eta\log(n)\log(q)/\epsilon^2$, and therefore

$$|T| \leq 2\sqrt{d'}n\eta\log^2(n)\log^2(q)/\epsilon^2 \leq 2(1+1/2\log\log(n))^i n\eta d^{1/2^{i+1}}\log^2(n)\log^2(q)/\epsilon^2,$$

satisfying the inductive claim.

For $\mathcal{C}_2$, we define random variables $X_1 \dots X_\ell$ for each coordinate in the support of $\mathcal{C}_2$. X_i will take value 1 if we sample coordinate i, and it will take 0 otherwise. Let $X = \sum_{i=1}^{\ell} X_i$, and let $\mu = \mathbb{E}[X]$. Note that

$$\frac{\mu^2}{\ell} = \left(\frac{\ell}{\sqrt{d'}}\right)^2/\ell = \frac{\ell}{d'} \geq \eta\cdot n\cdot\log^2(n)\log^2(q)/\epsilon^2.$$

Now, using Chernoff,

$$\Pr[X \geq (1+1/2\log\log(n))\mu] \leq e^{\frac{-2}{4\log^2\log(n)}\cdot\eta\cdot n\cdot\log(n)\log(q)/\epsilon^2} \leq 2^{-\eta n},$$

as we desire. Since $\mu = \ell/\sqrt{d'} \leq (1+1/2\log\log(n))^i\cdot d^{1/2^{i+1}}\cdot\eta\cdot n\log^2(n)\log^2(q)/\epsilon^2$, we conclude our result.

Now, to get our probability bound, we also operate inductively. Suppose that up to recursive level $i-1$, all sub-codes have been successfully sparsified to their desired size. At the ith level of recursion, there are at most 2^{i-1} codes which are being probabilistically sparsified. Each of these does not exceed its expected size by more than the prescribed amount with probability at most $2^{-\eta n}$. Hence, the probability all codes will be successfully sparsified up to and including the ith level of recursion is at least $1-2^{i-1}2^{-\eta n}-2^{i-1}2^{-\eta n} = 1-2^i 2^{-\eta n}$. □

Lemma A.12. *For any iteration of Algorithm 22 called on a code $\mathcal{C}$, $\mathcal{C}_1 \cup \sqrt{d}\cdot\mathcal{C}_2$ is a $(1\pm\epsilon)$ sparsifier to $\mathcal{C}$ with probability at least $1-2^{-(0.19\eta-110)\log n}\cdot n^{-101}$.*

Proof. First, let us note that the set T returned from Algorithm 21 is a superset of the bad set S of rows guaranteed by Corollary 6.5 (this follows from Claim A.1). Thus, we can equivalently view the procedure as producing three codes: $\mathcal{C}|_S, \mathcal{C}|_{T/S}$ and $\mathcal{C}_{\bar{T}} = \mathcal{C}'$. For our analysis, we will view this procedure in a slightly different light: we will imagine that first the algorithm removes exactly the bad set of rows S, yielding $\mathcal{C}|_S$ and $\mathcal{C}_{\bar{S}}$. Now, for this second code, $\mathcal{C}_{\bar{S}}$, we know the code-word counting bound will hold, and in particular, random sampling procedure will preserve codeword weights with high probability. However, our procedure is *not* uniformly sampling the coordinates in $\mathcal{C}_{\bar{S}}$, because some of these coordinates are in T/S, and thus are preserved exactly (i.e. with probability 1). For this, we will take advantage of the fact that preserving coordinates with probability 1 is *strictly better* than sampling at any rate < 1. Thus, we will still be able to argue that the ultimate result $\mathcal{C}_1 \cup \sqrt{d}\cdot\mathcal{C}_2$ is a $(1\pm\epsilon)$ sparsifier to $\mathcal{C}$ with high probability.

As mentioned above, we start by noting that $\mathcal{C}', \mathcal{C}|_S, \mathcal{C}_{T/S}$ form a *vertical* decomposition of $\mathcal{C}$. $\mathcal{C}|_S$ is preserved exactly, so we do not need to argue concentration of the codewords on these coordinates. Hence, it suffices to show that $\sqrt{d}\cdot\mathcal{C}_1 \cup \mathcal{C}_{T/S}$ is a $(1\pm\epsilon)$-sparsifier to $\mathcal{C}' \cup \mathcal{C}_{T/S}$.

To see that $\sqrt{d}\cdot\mathcal{C}_1 \cup \mathcal{C}_{T/S}$ is a $(1\pm\epsilon)$-sparsifier to $\mathcal{C}' \cup \mathcal{C}_{T/S}$, first note that every codeword in $\mathcal{C}' \cup \mathcal{C}_{T/S}$ is of weight at least $\sqrt{d}\cdot\eta\cdot\log(n)\log(q)/\epsilon^2$. This is because if there were a codeword of weight smaller than this, there would exist a subcode of $\mathcal{C}' \cup \mathcal{C}_{T/S}$ with 2 distinct codewords, and support bounded by $\sqrt{d}\cdot\eta\cdot\log(n)\log(q)/\epsilon^2$. But, because we have removed the set S of bad rows, we know that there can be no such sub-code remaining in $\mathcal{C}' \cup \mathcal{C}_{T/S}$. Thus, every codeword in $\mathcal{C}' \cup \mathcal{C}_{T/S}$ is of weight at least $\sqrt{d}\cdot\eta\cdot\log(n)\log(q)/\epsilon^2$.

Now, we can invoke Claim A.10 with $b = \sqrt{d}\eta \log(n)\log(q)/\epsilon^2$. Note that the hypothesis of Claim A.10 is satisfied by virtue of our code decomposition. Indeed, we removed coordinates of the code such that in the resulting $\mathcal{C}' \cup \mathcal{C}_{T/S}$, for any $\alpha \geq 1$, there are at most $(qn)^{2\alpha}$ codewords of weight $\leq \alpha\sqrt{d}\eta \log(n)\log(q)/\epsilon^2$. Using the concentration bound of Claim A.10 yields that with probability at least $1 - 2^{-(0.19\eta - 110)\log n} \cdot n^{-101}$, when samplin every coordinate at rate $\geq 1/\sqrt{d}$ the resulting sparsifier for $\mathcal{C}' \cup \mathcal{C}_{T/S}$ is a $(1 \pm \epsilon)$ sparsifier, as we desire. Note that we are using the fact that every coordinate is sampled with probability $\geq 1/\sqrt{d}$ (in particular, those in $T - S$ are sampled with probability 1).

□

Corollary A.13. *If Algorithm 22 achieves maximum recursion depth ℓ when called on a code $\mathcal{C}$, and $\eta > 600$, then the result of the algorithm is a $(1 \pm \epsilon)^\ell$ sparsifier to $\mathcal{C}$ with probability $\geq 1 - (2^\ell - 1) \cdot 2^{-(0.19\eta - 110)\log n} \cdot n^{-101}$*

Proof. We prove the claim inductively. Clearly, if the maximum recursion depth reached by the algorithm is 0, then we have simply returned the code itself. This is by definition a $(1 \pm \epsilon)^0$ sparsifier to itself.

Now, suppose the claim holds for maximum recursion depth $i - 1$. We will show it holds for maximum recursion depth i. Let the code we are sparsifying be $\mathcal{C}$. We break this into $\mathcal{C}_1$, $\mathcal{C}'$, and sparsify these. By our inductive claim, with probability $1 - (2^{i-1} - 1) \cdot 2^{-(0.19\eta - 110)\log n} \cdot n^{-101}$ each of the sparsifiers for $\mathcal{C}_1, \mathcal{C}'$ are $(1 \pm \epsilon)^{i-1}$ sparsifiers. Now, by Lemma A.12 and our value of η, $\mathcal{C}_1, \mathcal{C}'$ themselves together form a $(1 \pm \epsilon)$ sparsifier for $\mathcal{C}$ with probability $1 - 2^{-(0.19\eta - 110)\log n} \cdot n^{-101}$. So, by using Claim 3.2, we can conclude that with probability $1 - (2^i - 1) \cdot 2^{-(0.19\eta - 110)\log n} \cdot n^{-101}$, the result of sparsifying $\mathcal{C}_1, \mathcal{C}'$ forms a $(1 \pm \epsilon)^i$ approximation to $\mathcal{C}$, as we desire. □

We can then state the main theorem from this section:

Theorem A.14. *For a code $\mathcal{C}$ over $(\mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \ldots \mathbb{Z}_{q_u})$ with at most q^n distinct codewords, and length m, Algorithm 22 creates a $(1 \pm \epsilon)$ sparsifier for $\mathcal{C}$ with probability $1 - \log(m) \cdot 2^{-(0.19\eta - 110)\log n} \cdot n^{-100}$ with at most*

$$O(n\eta \log(n) \log^2(q) \log^2(m) (\log\log(m))^2/\epsilon^2)$$

coordinates.

Proof. For a code of with q^n distinct codewords, and length m, this means that our value of d as specified in the first call to Algorithm 22 is at most m as well. As a result, after only $\log\log m$ iterations, $d = m^{1/2^{\log\log m}} = m^{1/\log m} = O(1)$. So, by Corollary A.13, because the maximum recursion depth is only $\log\log m$, it follows that with probability at least $1 - (2^{\log\log m} - 1) \cdot 2^{-(0.19\eta - 110)\log n} \cdot n^{-101}$, the returned result from Algorithm 22 is a $(1 \pm \epsilon)^{\log\log m}$ sparsifier for $\mathcal{C}$.

Now, by Lemma A.11, with probability $\geq 1 - 2^{\log\log m} \cdot 2^{-\eta n} \geq 1 - \log(m) \cdot 2^{-(0.19\eta - 110)\log n} \cdot 2^{-n}$, every code at recursive depth $\log\log m$ has at most

$$(1 + 1/2\log\log(n))^{\log\log m} \cdot m^{1/\log m} \cdot \eta \cdot n \log(n)\log(q)/\epsilon^2 = O(n\eta \log(n)\log^2(q) \cdot e^{\frac{\log\log m}{\log\log n}}/\epsilon^2)$$

coordinates. Because the ultimate result from calling our sparsification procedure is the *union* of all of the leaves of the recursive tree, the returned result has size at most

$$\log(m) \cdot e^{\frac{\log\log m}{\log\log n}} \cdot O(n\eta \log(n)\log^2(q)/\epsilon^2) = O(n\eta \log(n)\log^2(q)\log^2(m)/\epsilon^2),$$

with probability at least $1 - \log(m) \cdot 2^{-(0.19\eta - 110)\log n} \cdot n^{-101}$.

Finally, note that we can replace ϵ with a value $\epsilon' = \epsilon/2\log\log m$. Thus, the resulting sparsifier will be a $(1\pm\epsilon')^{\log\log m} \leq (1\pm\epsilon)$ sparsifier, with the same high probability.

Taking the union bound of our errors, we can conclude that with probability $1 - \log(m)\cdot 2^{-(0.19\eta-110)\log n}\cdot n^{-100}$, Algorithm 22 returns a $(1\pm\epsilon)$ sparsifier for $\mathcal{C}$ that has at most

$$O(n\eta\log(n)\log^2(q)\log^2(m)(\log\log(m))^2/\epsilon^2)$$

coordinates. □

However, as we will address in the next subsection, this result is not perfect:

1. For large enough m, there is no guarantee that this probability is ≥ 0 unless η depends on m.
2. For large enough m, $\log^2(m)$ may even be larger than n.

A.5 Final Algorithm

Finally, we state our final algorithm in Algorithm 23, which will create a $(1\pm\epsilon)$ sparsifier for any code $\mathcal{C}\subseteq(\mathbb{Z}_{q_1}\times\mathbb{Z}_{q_2}\ldots\mathbb{Z}_{q_u})^m$ with $\leq q^n$ distinct codewords preserving only $\widetilde{O}(n\log^2(q)/\epsilon^2)$ coordinates. Roughly speaking, we start with a weighted code of arbitrary length, use the weight class decomposition technique, sparsify the decomposed pieces of the code, and then repeat this procedure now that the code will have a polynomial length. Ultimately, this will lead to the near-linear size complexity that we desire. We write a single iteration of this procedure below:

Algorithm 23: FinalCodeSparsify($\mathcal{C},\epsilon$)

1 Let n be $\log_q(|\mathcal{C}|)$.
2 Let m be the length of the code.
3 Let $\alpha=(m/\epsilon)^3$, and $\mathcal{D}_{\text{odd}},\mathcal{D}_{\text{even}}$ =WeightClassDecomposition($\mathcal{C},\epsilon,\alpha$).
4 Let $S_{\text{even}},\{H_{\text{even},i}\}$ =SpanDecomposition($\mathcal{D}_{\text{even}},\alpha$).
5 Let $S_{\text{odd}},\{H_{\text{odd},i}\}$ =SpanDecomposition($\mathcal{D}_{\text{odd}},\alpha$).
6 **for** $i\in S_{even}$ **do**
7 Let $\widehat{H}_{\text{even},i},w_{\text{even},i}$ = MakeUnweighted($H_{\text{even},i},\alpha,i,\epsilon/8$).
8 Let $\widetilde{H}_{\text{even},i}$ =CodeSparsify($\widehat{H}_{\text{even},i},\log_q(|\text{Span}(\widehat{H}_{\text{even},i})|),\epsilon/80,100(\log(m/\epsilon)\log\log(q))^2$).
9 **end**
10 **for** $i\in S_{odd}$ **do**
11 Let $\widehat{H}_{\text{odd},i},w_{\text{odd},i}$ = MakeUnweighted($H_{\text{odd},i},\alpha,i,\epsilon/8$).
12 Let $\widetilde{H}_{\text{odd},i}$ =CodeSparsify($\widehat{H}_{\text{odd},i},\log_q(|\text{Span}(\widehat{H}_{\text{odd},i})|),\epsilon/80,100(\log(m/\epsilon)\log\log(q))^2$).
13 **end**
14 **return** $\bigcup_{i\in S_{even}}\left(w_{even,i}\cdot\widetilde{H}_{even,i}\right)\cup\bigcup_{i\in S_{odd}}\left(w_{odd,i}\cdot\widetilde{H}_{odd,i}\right)$

First, we analyze the space complexity. WLOG we will prove statements only with respect to $\mathcal{D}_{\text{even}}$, as the proofs will be identical for $\mathcal{D}_{\text{odd}}$.

Claim A.15. *Suppose we are calling Algorithm 23 on a code $\mathcal{C}$ with q^n distinct codewords. Let $n_{even,i}=\log_q(Span((\widehat{H}_{even,i}))|)$ from each call to the for loop in line 5.*

For each call $\widetilde{H}_{even,i}$ =CodeSparsify($\widehat{H}_{even,i},\log_q(|\text{Span}(\widehat{H}_{even,i})|),\epsilon/10,100(\log(n/\epsilon)\log\log(q))^2$) in Algorithm 23, the resulting sparsifier has

$$O\left(n_{even,i}\log(n_{even,i})\log^4(m/\epsilon)\log^2(q)(\log\log(m/\epsilon)\log\log(q))^2/\epsilon^2\right)$$

coordinates with probability at least $1 - \log(m/\epsilon) \cdot 2^{-\Omega(\log^2(m/\epsilon)(\log\log(q))^2)}$.

Proof. We use several facts. First, we use Theorem A.14. Note that the m in the statement of Theorem A.14 is actually a $\text{poly}(m/\epsilon)$ because $\alpha = m^3/\epsilon^3$, and we started with a weighted code of length $O(m)$. So, it follows that after using Algorithm 20, the support size is bounded by $O(m^4/\epsilon^3)$. We've also added the fact that η is no longer a constant, and instead carries $O((\log(m/\epsilon)\log\log(q))^2)$, and carried this through to the probability bound. □

Lemma A.16. *In total, the combined number of coordinates over $i \in S_{even}$ of all of the $\widetilde{H}_{even,i}$ is at most $\widetilde{O}(n\log^4(m)\log^2(q)/\epsilon^2)$ with probability at least $1 - \log(m\log(q)/\epsilon) \cdot 2^{-\Omega(\log^2(m/\epsilon)(\log\log(q))^2)}$.*

Proof. First, we use Claim A.6 to see that

$$\sum_{i \in S_{\text{even}}} \log_q(|\text{Span}(\widehat{H}_{\text{even},i})|) \leq n.$$

Thus, in total, the combined length (total number of coordinates preserved) of all the $\widetilde{H}_{\text{even},i}$ is

$$\begin{aligned}
&\sum_{i \in S_{\text{even}}} \text{number of coordinates in } \widehat{H}_{\text{even},i} \\
&\leq \sum_{i \in S_{\text{even}}} O\left(n_{\text{even},i}\log(n_{\text{even},i})\log^4(m/\epsilon)\log^2(q)(\log\log(m/\epsilon)\log\log(q))^2/\epsilon^2\right) \\
&\leq \sum_{i \in S_{\text{even}}} (n_{\text{even},i}) \cdot \widetilde{O}\left(\log^4(m)\log^2(q)/\epsilon^2\right) \\
&= n \cdot \widetilde{O}\left(\log^4(m)\log^2(q)/\epsilon^2\right) \\
&= \widetilde{O}(n\log^4(m)\log^2(q)/\epsilon^2).
\end{aligned}$$

To see the probability bound, we simply take the union bound over all at most n distinct $\widetilde{H}_{\text{even},i}$, and invoke Claim A.15. □

Now, we will prove that we also get a $(1 \pm \epsilon)$ sparsifier for $\mathcal{D}_{\text{even}}$ when we run Algorithm 23.

Lemma A.17. *After combining the $\widehat{H}_{even,i}$ from Lines 5-8 in Algorithm 23, the result is a $(1\pm\epsilon/4)$-sparsifier for $\mathcal{D}_{even}$ with probability at least $1 - \log(m\log(q)/\epsilon) \cdot 2^{-\Omega(\log^2(m/\epsilon)(\log\log(q))^2)}$.*

Proof. We use Lemma A.7, which states that to sparsify $\mathcal{D}_{\text{even}}$ to a factor $(1 \pm \epsilon/4)$, it suffices to sparsify each of the $H_{\text{even},i}$ to a factor $(1 \pm \epsilon/8)$, and then combine the results.

Then, we use Lemma A.8, which states that to sparsify any $H_{\text{even},i}$ to a factor $(1\pm\epsilon/8)$, it suffices to sparsify $\widehat{H}_{\text{even},i}$ to a factor $(1 \pm \epsilon/80)$, where again, $\widehat{H}_{\text{even},i}$ is the result of calling Algorithm 20. Then, we must multiply $\widehat{H}_{\text{even},i}$ by a factor $\alpha^{i-1} \cdot \epsilon/10$.

Finally, the resulting code $\widehat{H}_{\text{even},i}$ is now an unweighted code, whose length is bounded by $O(m^4/\epsilon^3)$, with at most $q^n_{\text{even},i}$ distinct codewords. The accuracy of the sparsifier then follows from Theorem A.14 called with parameter $\epsilon/80$.

The failure probability follows from noting that we take the union bound over at most $n\log(q)$ $H_{\text{even},i}$. By Theorem A.14, our choice of η, and the bound on the length of the support being $O(m^4/\epsilon^3)$, the probability bound follows. □

For Theorem A.14, the failure probability is characterized in terms of the number of distinct codewords of the code that is being sparsified. However, when we call Algorithm 22 as a sub-routine in Algorithm 23, we have no guarantee that the number of distinct codewords is $\omega(q)$. Indeed, it is certainly possible that the decomposition in H_i creates n different codes, each with q distinct codewords in their span. Then, choosing η to only be a constant, as stated in Theorem A.14, the failure probability could be constant, and taking the union bound over n choices, we might not get anything meaningful. To amend this, instead of treating η as a constant in Algorithm 22, we set $\eta = 100(\log(m/\epsilon)\log\log(q))^2$, where now m is the length of the original code $\mathcal{C}$, *not* in the current code that is being sparsified H_i. With this modification, we can then attain our desired probability bounds.

Theorem A.18. *For any code $\mathcal{C}$ with q^n distinct codewords and length m over $(\mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \dots \mathbb{Z}_{q_u})$, Algorithm 23 returns a $(1 \pm \epsilon)$ sparsifier to $\mathcal{C}$ with $\widetilde{O}(n \log^4(m) \log^2(q)/\epsilon^2)$ coordinates with probability $\geq 1 - 2^{-\Omega((\log(m/\epsilon)\log\log(q))^2)}$.*

Proof. First, we use Lemma A.2. This Lemma states that in order to get a $(1 \pm \epsilon)$ sparsifier to a code $\mathcal{C}$, it suffices to get a $(1 \pm \epsilon/4)$ sparsifier to each of $\mathcal{D}_{\text{even}}, \mathcal{D}_{\text{odd}}$, and then combine the results.

Then, we invoke Lemma A.17 to conclude that with probability $\geq 1 - 2^{-\Omega((\log(m/\epsilon)\log\log(q))^2)}$, Algorithm 11 will produce $(1 \pm \epsilon/4)$ sparsifiers for $\mathcal{D}_{\text{even}}, \mathcal{D}_{\text{odd}}$.

Further, to argue the sparsity of the algorithm, we use Lemma A.16. This states that with probability $\geq 1 - 2^{-\Omega((\log(m/\epsilon)\log\log(q))^2)}$, Algorithm 23 will produce code sparsifiers of size $\widetilde{O}(n \log^4(m) \log^2(q)/\epsilon^2)$ for $\mathcal{D}_{\text{even}}, \mathcal{D}_{\text{odd}}$.

Thus, in total, the failure probability is at most $2^{-\Omega((\log(m/\epsilon)\log\log(q))^2)}$, the total size of the returned code sparsifier is at most $\widetilde{O}(n \log^4(m) \log^2(q)/\epsilon^2)$, and the returned code is indeed a $(1 \pm \epsilon)$ sparsifier for $\mathcal{C}$, as we desire.

Note that the returned sparsifier may have some duplicate coordinates because of Algorithm 20. Even when counting duplicates of the same coordinate separately, the size of the sparsifier will be at most $\widetilde{O}(n \log^4(m) \log^2(q)/\epsilon^2)$. We can remove duplicates of coordinates by adding their weights to a single copy of the coordinate. □

Claim A.19. *Running Algorithm 23 on a code of length m with parameter ϵ takes time poly$(mn \log(q)/\epsilon)$.*

Proof. Let us consider the constituent algorithms that are invoked during the execution of Algorithm 23. First, we consider weight class decomposition. This groups rows together by weight (which takes time $\widetilde{O}(\log(m))$). Next, we invoke SpanDecomposition, which then contracts on the rows in the largest weight class. Note that in the worst case, we perform $O(n \log(q))$ contractions, as each contraction reduces the number of codewords by a factor of ≥ 2. Further, each contraction takes time $O(mn \log(q))$ as the total number of rows is bounded by m, and there are at most $n \log(q)$ columns in the generating matrix. Thus, the total runtime of this step is $\widetilde{O}(mn^2 \log^2(q))$.

The next step is to invoke the algorithm MakeUnweighted. Because the value of α is m^3/ϵ^3, this takes time at most $O(m^4/\epsilon^3)$ to create the new code with this many rows.

Finally, we invoke CodeSparsify on a code of length $\leq m^4/\epsilon^3$ and with at most $q^{n_{\text{even},i}}$ distinct codewords. Note that there are polylog(m) nodes in the recursive tree that is built by CodeSparsify. Each such node requires removing the set T which is a set of $\leq \widetilde{O}(\sqrt{m^4/\epsilon^3}) = \widetilde{O}(m^2/\epsilon^{1.5})$ maximum spanning subsets. Constructing each such subset (by Claim 4.8) takes time at most $O((m^4\epsilon^3)^2 n \log(q))$. After this step, the subsequent random sampling is efficiently doable. Thus, the total runtime is bounded by poly$(mn \log(q)/\epsilon)$, as we desire.

□

Note that creating codes of linear-size now simply requires invoking Algorithm 23 two times (each with parameter $\epsilon/2$). Indeed, because the length of the code to begin with is $\leq q^n$, this means that after the first invocation, the resulting $(1 \pm \epsilon/2)$ sparsifier $\mathcal{C}'$ that we get maintains $\leq \widetilde{O}(n^5 \log^6(q)/\epsilon^2)$ coordinates. In the second invocation, we get a $(1 \pm \epsilon/2)$-sparsifier $\mathcal{C}''$ for $\mathcal{C}'$, whose length is bounded by $\widetilde{O}(n \log^4(n^5 \log^6(q)/\epsilon^2) \log^2(q)/\epsilon^2) = \widetilde{O}(n \log^2(q)/\epsilon^2)$, as we desire.

Formally, this algorithm can be written as:

Algorithm 24: Sparsify$(\mathcal{C}, \epsilon)$

$\mathcal{C}' =$FinalCodeSparsify$(\mathcal{C}, \epsilon/2)$.
return *FinalCodeSparsify*$(\mathcal{C}', \epsilon/2)$

Theorem A.20. *Algorithm 22 returns a $(1 \pm \epsilon)$-sparsifier to $\mathcal{C}$ of size $\widetilde{O}(n \log^2(q)/\epsilon^2)$ with probability $1 - 2^{-\Omega((\log(n/\epsilon) \log\log(q))^2)}$ in time poly$(mn \log(q)/\epsilon)$.*

Proof. Indeed, because the length of the code to begin with is $\leq q^n$, this means that after the first invocation, the resulting $(1 \pm \epsilon/2)$ sparsifier $\mathcal{C}'$ that we get maintains $\leq \widetilde{O}(n^5 \log^6(q)/\epsilon^2)$ coordinates. In the second invocation, we get a $(1 \pm \epsilon/2)$-sparsifier $\mathcal{C}''$ for $\mathcal{C}'$, whose length is bounded by $\widetilde{O}(n \log^4(n^5 \log^6(q)/\epsilon^2) \log^2(q)/\epsilon^2) = \widetilde{O}(n \log^2(q)/\epsilon^2)$, as we desire. To see the probability bounds, note that $m \geq n$, and thus both processes invocations of FinalCodeSparsify succeed with probability $1 - 2^{-\Omega((\log(n/\epsilon) \log\log(q))^2)}$.

Finally, to see that the algorithm is efficient, we simply invoke Claim A.19 for each time we fun the algorithm. Thus, we get our desired bound. □

B Sparsifying Binary Predicates over General Alphabets

In this section, we show how to rederive the result of [BZ20] using our framework. To do this, we first introduce a more general definition of an affine predicate.

Definition B.1. *For a predicate $P : \Sigma^r \to \{0, 1\}$, we say that P is an affine predicate if there exists a group A, $a_i, b \in A$, and $T_1, \ldots T_r : \Sigma \to \mathbb{Z}$ such that $\forall x \in \Sigma^r$*

$$P(x_1, \ldots x_r) = 1 \iff \sum_{i=1}^{r} T_i(x_i) a_i \neq b.$$

If the group A is Abelian, then we say that P is an affine Abelian *predicate.*

Note that the purpose of the functions T_i is to embed the alphabet into the integers. These functions *do not* have to be linear in order to be sparsifiable. As an immediate consequence of Theorem 1.7, we can sparsify general alphabet, affine predicate P over Abelian groups.

Theorem B.1. *For any alphabet Σ, an affine Abelian predicate P over Abelian group A, any instance $\Phi \in CSP(P)$ on n variables can be $(\epsilon, \widetilde{O}(n \log^2(|\Sigma|)/\epsilon^2))$-sparsified.*

Proof sketch. By definition, there exists $a_i, b \in A$, $T_1, \ldots T_r : \Sigma \to \mathbb{Z}$ such that $\forall x \in \Sigma^r$

$$P(x_1, \ldots x_r) = 1 \iff \sum_{i=1}^{r} T_i(x_i) a_i \neq b.$$

Now, let us create a generating matrix G in $A^{m\times(n+1)}$ in the canonical way. Indeed, for each of the $[m]$ constraints, let the variables that the predicate is operating on be $x_j^{(1)}, \dots x_j^{(r)}$. In the jth row of the generating matrix, for the column corresponding to $x_j^{(i)}$, place the coefficient a_i. Finally, in the final column (the $n+1$st column), place the value b. It follows that for any linear combination of the columns $x \in \mathbb{Z}^n \circ 1$, the resulting codeword Gx will have weight equal to the weight of the satisfied constraints on assignment x. Then, by sparsifying the code defined by G, this yields a sparsifier for our CSP instance Φ. We conclude then by invoking Theorem 1.7.

Note that even though the functions T_i may not be linear, they still correspond to linear combinations of the columns of this generating matrix. Our sparsifier works for *any* linear combination of the columns of the generating matrix, even if the coefficients $T_i(x_i)$ are calculated in a non-linear way. □

With this, we are able to explain our proof strategy. Indeed, given a general binary predicate $P : \Sigma^2 \to \{0,1\}$, we view the predicate as a matrix in $\{0,1\}^{|\Sigma|\times|\Sigma|}$. For this matrix, the work of [BZ20] showed that the predicate P is sparsifiable if and only if there is no rectangle with corners forming an AND: i.e., a rectangle with corners having 3 0's and a single 1. When this is the case, [BZ20] showed that this yields an AND of arity 2, and hence requires sparsifiers of quadratic size. On the other hand, when none of these rectangles are of this form, we can indeed sparsify the predicate.

Lemma B.2. *Suppose a predicate $P : \Sigma^2 \to \{0,1\}$ does not have a projection to AND. Then, any instance $\Phi \in CSP(P)$ on n variables admits an $(\epsilon, \widetilde{O}_{|\Sigma|}(n/\epsilon^2))$ sparsifier.*

Proof. As pointed out by Butti and Živný [BZ20], we note that if P does not have a projection to AND then there must exist t disjoint sets $S_1, \dots, S_t \subseteq \Sigma$ and another family of t disjoint sets $U_1, \dots, U_t \subseteq \Sigma$ such that $P(a,b) = 1$ if and only if $(a,b) \notin \cup_{i=1}^t S_i \times U_i$. We claim that this implies P is an affine predicate.

We work over the group $A = \mathbb{Z}_{t+2}$. Next, we define the functions T_1, T_2 in our affine, Abelian predicate. We let $T_1, T_2 : \Sigma \to \mathbb{Z}$ be defined as $T_1(\sigma) = i$ if $\sigma \in S_i$, and 0 otherwise. Likewise, we define $T_2(\sigma) = i$ if $\sigma \in U_i$, and $t+1$ otherwise. Further we set $a_1 = 1$, $a_2 = -1$ and $b = 0$. This leads to the affine Abelian predicate P' given by:

$$P'(x_1, x_2) = \mathbf{1}[T_1(x_1) - T_2(x_2) \neq 0 \mod t+2].$$

One can verify that $\forall (x_1, x_2) \in \Sigma^2, P'(x_1, x_2) = P(x_1, x_2)$. Thus, by Theorem B.1, we conclude the existence of an $(\epsilon, \widetilde{O}(n \log^2(|\Sigma|)/\epsilon^2))$ sparsifier.

□

C Sparsifying Affine Predicates over Larger Alphabets

C.1 More General Notions

In a more general way, we can say:

Definition C.1. *For a predicate $P : \Sigma^r \to \{0,1\}$, we say that P is a general affine predicate if there exists a constant c, a group A, $a_i, b \in A^c$, and $T_1, \dots T_r : \Sigma \to \mathbb{Z}^c$ such that $\forall x \in \Sigma^r$*

$$P(x_1, \dots x_r) = 1 \iff \sum_{i=1}^{r} \langle T_i(x_i), a_i \rangle \neq b.$$

If the group A is Abelian, then we say that P is a general affine Abelian *predicate.*

Note that by the equivalence with the generating matrix, this will still be sparsifiable, as we can simply add c columns for each variable.

C.2 Infinite Abelian Groups

Note that we can extend our proof technique to sparsify affine abelian predicates that are defined over larger alphabets too.

Suppose that a predicate is affine only over an infinite Abelian group. I.e. $P(x) = \mathbf{1}[\sum_{i=1}^{r} a_i x_i \neq b]$, for $a_i, b \in A$, and A being an infinite group. First, we note that the relevant elements of A will be finitely generated. I.e., the elements we care about $a_i, b \in H$, where H is a subgroup of A generated by a_i, b. This is because we only care about the subgroup generated by $a_1, \dots a_r, b$, and can effectively ignore everything else. We rewrite the predicate $P(x) = \mathbf{1}[\sum_{i=1}^{r} a_i x_i - b \neq 0]$. Note that WLOG we can consider the case when $b = 0$, as otherwise we can simply make the predicate of arity $r + 1$, and treat b as the coefficient to another variable.

Now, because the subgroup H we care about is finitely generated, we can invoke the fundamental theorem of Abelian groups. This tells us that H is isomorphic to a group of the form

$$\mathbb{Z}^{k_0} \times \mathbb{Z}_{p_1}^{k_1} \times \cdots \times \mathbb{Z}_{p_\ell}^{k_\ell}.$$

Let the new size $w = \sum_{j=0}^{\ell} k_0$. This means that we can write P as a predicate over a tuple of length w, where the ith entry in the tuple will be a constraint over one of the cyclic groups in the product. I.e., we can write

$$P(x) = \mathbf{1}[\sum_{i=1}^{r} (a_1^{(i)}, \dots a_w^{(i)}) x_i].$$

Now, note that this predicate is essentially taking the OR over each entry in the tuple. That is,

$$P(x) = \mathbf{1}[\sum_{i=1}^{r} a_1^{(i)} x_i \neq 0] \vee \cdots \vee \mathbf{1}[\sum_{i=1}^{r} a_1^{(w)} x_i \neq 0].$$

Without loss of generality, we can then assume that $w \leq 2^{2^r}$. This is because the number of possible predicates on r variables is at most 2^{2^r}, so it follows that if we are taking an OR, there can be at most 2^{2^r} distinct predicates before we are forced to have repeating predicates. If any predicates are repeating, they are not contributing to the OR.

Now, we know that $P(x)$ can be written as the OR of at most 2^{2^r} predicates, where each predicate $P^{(i)}$ is of the form

$$P^{(i)} = \mathbf{1}[\sum_{i=1}^{r} a_1^{(i)} x_i \neq 0],$$

and algebra is done over *some* cyclic group.

It suffices for us to argue that we can represent each of these predicates $P^{(i)}$ over a smaller cyclic group $\mathbb{Z}_{p^*}$, where p^* depends only on r. Indeed, to see this, consider any $P^{(i)}$, where the algebra is done over a finite cyclic group. We know that there is some set of assignments $S \subseteq \{0,1\}^r$ such that $P^{(i)}$ is 1 for the assignments in S, and 0 for the assignments outside S. We also know that there exists a constant p such that $P^{(i)}$ is expressible as a linear equation of $\mathbb{Z}_p$. Now, let p' be the smallest value of p such that $P^{(i)}$ is expressible as a linear equation over $\mathbb{Z}_{p'}$. Then p' must be bounded by a function of r. Next, consider the case when the algebra for $P^{(i)}$ is done over $\mathbb{Z}$. This means

$$P^{(i)} = \mathbf{1}[\sum_{i=1}^{r} a_i x_i \neq 0].$$

In particular, because we know $P^{(i)}$ is expressible over $\mathbb{Z}$, there must exist some choice of $a_1, \ldots a_r$ such that $\max(|a_1|, \ldots |a_r|) = a$ is as small as possible. Again, for this choice, a must be bounded by some function of r (denote this $f(r)$). Now, it follows that we can simply choose p' to be the smallest prime larger than $2r \cdot f(r)$. For this choice of p', doing $\{0,1\}$ weighted arithmetic with coefficients $a_1, \ldots a_r$ will be the same as doing arithmetic of $\mathbb{Z}_{p'}$. Further p' will be bounded by a function of r. Thus, we have that P is expressible as the OR of at most 2^{2^r} affine functions, each of which is doing arithmetic over a cyclic group bounded in size as a function of r. In particular, for r constant, this leads to no overhead in the size of our sparsifiers.

C.3 Lattice Perspective

Lemma C.1. *Suppose B is a $d \times \ell$ matrix with entries in $\mathbb{Z}$, such that the magnitude of the largest sub-determinant is bounded by M, and rank$(B) = k$. Then, every element of the lattice generated by the columns of B is given exactly by the solutions to $d - k$ linear equations and k modular equations. All coefficients of the linear equations are bounded in magnitude by M, and all modular equations are written modulo a single $M' \leq M$.*

Proof. First, if $\text{rank}(B) = k < d$, this means that there exist k linearly independent rows such that the remaining $d - k$ rows are linear combinations of these rows. Let us remove these rows for now, and focus on $B' \in \mathbb{Z}^{k \times \ell}$ where now the matrix has full row-rank.

It follows that for this matrix B' we can create a new matrix $\hat{B}$ such that the lattice generated by B' (denoted by $\mathcal{L}(B') = \mathcal{L}(\hat{B})$) and B' is in Hermite Normal Form (HNF). In this form, $\hat{B}$ is lower triangular with the diagonal entries of $\hat{B}$ satisfying

$$\det(\hat{B}) = \prod_{i=1}^{k} \hat{B}_{i,i} \leq \max_{k \times k \text{ subrectangle A}} \det(B'_A).$$

Further, all columns beyond the kth column will be all zeros, so we can remove these from the matrix.

We can now define the *dual* lattice to $\mathcal{L}(B') = \mathcal{L}(\hat{B})$). For a lattice $\Lambda \subseteq \mathbb{Z}^k$, we say that

$$\text{dual}(\Lambda) = \{x \in \mathbb{Q}^k : \forall y \in \Lambda, \langle x, y \rangle \in Z\}.$$

Here it is known that the dual is an exact characterization of the lattice Λ. I.e., any vector in Λ will have integer-valued inner product with *any* vector in the dual, while for any vector not in Λ, there exists a vector in the dual such that the inner-product is not integer valued.

Now, for our matrix $\hat{B}$, it is known that one can express the dual lattice to $\hat{B}$ as $\hat{D} = \hat{B}(\hat{B}^T\hat{B})^{-1}$. As a result, it must be the case that $\hat{D} \subseteq \mathbb{Z}^k / \det(\hat{B})$. If a vector x of length k is not in $\mathcal{L}(\hat{B})$, it must be the case that there exists a column y of $\hat{D}$ such that $\langle x, y \rangle \notin \mathbb{Z}$. Otherwise, if the inner-product with every column is in $\mathbb{Z}$, it follows that for any vector in the dual, the inner product would also be in $\mathbb{Z}$, as we can express any vector in the dual as an integer linear combination of columns in $\hat{D}$. Thus, it follows that membership of a vector x in $\mathcal{L}(\hat{B})$ can be tested exactly by the k equations $\forall i \in [k] : \langle x, \hat{D}_i \rangle \in \mathbb{Z}$. Now, because every entry of $\hat{D}$ has denominator dividing $\det(\hat{B})$, it follows that we can scale up the entire equation by $\det(\hat{B})$. Thus, an equivalent way to test if $x \in \mathcal{L}(\hat{B})$ is by checking if $\forall i \in [k] : \langle x, \det(\hat{B}) \cdot \hat{D}_i \rangle = 0 \mod \det(\hat{B})$. Now, all the coefficients of these equation are integers, and we are testing whether the sum is 0 modulo an integer. Thus, we can test membership of any k-dimensional integer vector in $\mathcal{L}(\hat{B})$ with k modular equations over $\det \hat{B} \leq M$.

The above argument gives a precise way to characterize when the restriction of a d dimensional vector to a set of coordinates corresponding with linearly independent rows in B is contained in the

lattice generated by these same rows of B. It remains to show that we can also characterize when the dependent coordinates (i.e. coordinates corresponding to the rows that are linearly dependent on these rows) are contained in the lattice. Roughly speaking, the difficulty here arises from the fact that we are operating with a non-full dimensional lattice. I.e., there exist directions that one can continue to travel in $\mathbb{Z}^n$ without ever seeing another lattice point. In this case, we do not expect to be able to represent membership in the lattice with a modular linear equation, as these modular linear equations rely on periodicity of the lattice.

Instead, here we rely on the fact that for any of the rows of B that are linearly dependent, we know that there is a way to express it as a linear combination of the set of linearly independent rows. WLOG, we will assume the first k rows $r_1, \dots r_k$ are linearly independent, and we are interested in finding c_i such that $\sum_{i=1}^{k} c_i r_i = r_{k+1}$. Now, consider any subset A of k linearly independent columns amongst these k rows. We denote the corresponding restriction of the rows to these columns by $r_i^{(A)} \in \{0,1\}^k$. It follows that if we want a linear combination of these rows such that $\sum_{i=1}^{k} c_i r_i^{(A)} = r_{k+1}^{(A)}$, we can express this a constraint of the form $M^{(A)} c = (r_{k+1}^{(A)})^T$, where we view the ith column of M as being the (transpose of) $r_i^{(A)}$ and c as being the vector of values $c_1, \dots c_k$. Using Cramer's rule, we can calculate that $c_i = \det(M_i^{(A)}) / \det(M^{(A)})$, where $M_i^{(A)}$ is defined to be the matrix $M^{(A)}$ with the ith column replaced by $(r_{k+1}^{(A)})^T$. In particular, this means that we can express $r_{k+1}^{(A)} = \sum_{i=1}^{k} \det(M_i^{(A)}) / \det(M^{(A)}) \cdot r_i^{(A)}$, and because A corresponds to a set of linearly independent columns, it must also be the case that $r_{k+1} = \sum_{i=1}^{k} \det(M_i^{(A)}) / \det(M^{(A)}) \cdot r_i$. We can re-write this as an integer linear equation by expressing $r_{k+1} \cdot \det(M^{(A)}) = \sum_{i=1}^{k} \det(M_i^{(A)}) \cdot r_i$. This means that for any valid vector $x \in \mathbb{Z}^d$ expressable as a linear combination of the columns of B, it must be the case that $x_{k+1} \cdot \det(M^{(A)}) = \sum_{i=1}^{k} \det(M_i^{(A)}) \cdot x_i /$

We can repeat the above argument for each of the $d - k$ linearly independent rows. Let j denote the index of a row linearly dependent on the first k rows. It follows that $x_j \cdot \det(M^{(A)}) = \sum_{i=1}^{k} \det(M_i^{(A),j}) \cdot x_i$, where now $M_i^{(A),j}$ is the $d \times d$ matrix $M^{(A)}$ where the ith column has been replaced with $(r_j^{(A)})^T$. Note that every coefficient that appears in these equations above is of the form $\det(C)$ where C is a $d \times d$ submatrix of B. It follows that each of these coefficients is bounded in magnitude by M, where M is again defined to be the maximum magnitude of the determinant of any square sub-matrix.

To conclude, we argue that for any vector $x \in \mathcal{L}(B)$, x satisfies all of the above linear equations and modular equations. The first part of the proof showed that amongst the set of linearly independent rows S, the dual of the lattice exactly captures when x_S is in the lattice generated by B_S. That is, if x_S is generated by B_S, then x_S satisfies the above modular linear equations, while if x_S is not in the span of B_S, then x_S does not satisfy the modular linear equations. Now, if x_S does not satisfy the modular linear equations, this is already a witness to the fact that x is not in the $\mathcal{L}(B)$. But, if x_S is in the span of B_S, then if x is in the span of B, it must also be the case that the coordinates of $x_{\bar{S}}$ satisfy the exact same linear dependence on the x_S that $B_{\bar{S}}$ has on B_S. This is captured by our second set of linear equations. □

Theorem C.2. *Let $P : \Sigma^r \to \{0,1\}$ be a predicate over an arbitrary alphabet of arity r. Let $S = P^{-1}(0) \subseteq \Sigma^r$ denote the unsatisfying assignments of P, and let $\hat{S} \subseteq \{0,1\}^{|\Sigma|r}$ denote the lifted version of S where we map $\sigma \in \Sigma$ to a vector $v \in \{0,1\}^{|\Sigma|}$ such that $v_\sigma = 1$, and is 0 otherwise. If $\hat{S}$ is closed under integer valued linear combinations, then CSPs with predicate P on n variables are sparsifiable to size $\widetilde{O}(n \cdot |\Sigma|^4 \cdot r^4 / \epsilon^2)$.*

Proof. Let us create a matrix $B \in \{0,1\}^{|\Sigma|r \times |\hat{S}|}$ where the ith column of B is the ith element of $\hat{S}$.

Let k be the rank of B. It follows that for any assignment $x \in \{0,1\}^{|\Sigma|r}$, we can exactly express the membership of x in $\mathcal{L}(B)$ with d modular linear equations, and $|\Sigma|r - d$ linear equations. I.e., x is in $\mathcal{L}(B)$ if and only if all of these equations are satisfied. Note that the d modular linear equations are all over modulus $M \leq \max_{k \in [|\Sigma|r], k \times k \text{ subrectangle } A} \det(B'_A) \leq (|\Sigma|r)^{|\Sigma|r}$. Likewise, the integer linear equations also all have coefficients $\leq (|\Sigma|r)^{|\Sigma|r}$. It follows that because $x \in \{0,1\}^{|\Sigma|r}$, we can choose a prime p such that $p \geq 2 \cdot |\Sigma|r \cdot (|\Sigma|r)^{|\Sigma|r}$. Now, for any of the integer linear equations of the form $c_1x_1 + \ldots c_kx_k - c_{k+1}x_{k+1}$, it will be the case that for $x \in\in \{0,1\}^{|\Sigma|r}$,

$$c_1x_1 + \ldots c_kx_k - c_{k+1}x_{k+1} = 0 \iff c_1x_1 + \ldots c_kx_k - c_{k+1}x_{k+1} = 0 \mod p.$$

This is because the expression on the left can never be as large as p or $-p$ since we chose p to be sufficiently large.

Thus, we can express $x \in \{0,1\}^{|\Sigma|r}$ as being in the lattice $\mathcal{L}(B)$ if and only if all $|\Sigma|r$ modular equations are 0. This is then the OR of $|\Sigma|r$ modular equations, which can be expressed over the Abelian group $A = \mathbb{Z}_M \times \mathbb{Z}_m \cdots \times Z_m \times Z_p \cdots \times \mathbb{Z}_p$, where there are k copies of Z_m, and $|\Sigma|r$ copies of $\mathbb{Z}_p$. It follows that $|A| \leq \left(2 \cdot |\Sigma|r \cdot (|\Sigma|r)^{|\Sigma|r}\right)^{|\Sigma|r} = (2|\Sigma|r)^{|\Sigma|r} \cdot (|\Sigma|r)^{|\Sigma|^2r^2}$. Thus, for any CSP on predicate P of the above form on n variables, we can $1 \pm \epsilon$ sparsify P to size $\widetilde{O}(n \cdot |\Sigma|^4 \cdot r^4/\epsilon^2)$. $\square$

D Non-Affine Predicates with no Projections to AND

In this section, we present a predicate which provably is not affine, and yet still does not have any projection to an AND of arity 2. This shows that there is necessarily a separation between the techniques we have for creating sparsifiers versus showing lower-bounds. We will do this by specifically constructing a predicate which is not affine (in particular, there will exist a linear combination of unsatisfying assignments which yields a *satisfying* assignment).

Indeed, consider the following predicate $P : \{0,1\}^9 \to \{0,1\}$:

1. $P(000000000) = 0$.
2. $P(111111000) = 0$.
3. $P(111000111) = 0$.
4. $P(110001001) = 0$.
5. $P(101010010) = 0$.
6. For all other assignments x, $P(x) = 1$.

In particular, note that the unsatisfying assignments of P are *not* closed under integer linear combinations. We can consider

$$1 \cdot (000000000) + 1 \cdot (111111000) + 1 \cdot (111000111) - 1 \cdot (110001001) - 1 \cdot (101010010) = (011100100).$$

Thus, we immediately get the following claim:

Claim D.1. *P is not expressible as an affine predicate over* any *Abelian group.*

Proof. Any predicate P which is expressible as an affine predicate over an Abelian group must have its unsatisfying assignments closed under integer linear combinations. Indeed, let $y_1, \ldots y_\ell \in$

$P^{-1}(0)$, and $\alpha_1, \dots \alpha_\ell$ be such that $\sum_{j=1}^{\ell} \alpha_j y_j \in \{0,1\}^r$ (when addition is done over $\mathbb{Z}$). We claim then that $P(\sum_{j=1}^{\ell} \alpha_j y_j) = 0$ also.

This follows simply from P being an affine Abelian predicate. It must be the case that $P(b_1, \dots b_r) = \mathbf{1}[\sum_i a_i b_i \neq 0]$, for some $a_i \in A$, where A is an Abelian group. Then,

$$P(\sum_{j=1}^{\ell} \alpha_j y_j) = \mathbf{1}[\sum_i a_i (\sum_{j=1}^{\ell} \alpha_j y_j)_i \neq 0] = \mathbf{1}[\sum_{j=1}^{\ell} \alpha_j \sum_i a_i (y_j)_i \neq 0].$$

But, because each $y_j \in P^{-1}(0)$, it must be the case that $\sum_i a_i (y_j)_i = 0$. Thus, the entire sum must be 0, so we can conclude that $P(\sum_{j=1}^{\ell} \alpha_j y_j) = 0$. □

It remains to prove that the above predicate has no projection to AND_2.

Claim D.2. *P has no projection to AND_2.*

Proof sketch. Note that if there is a projection to AND this implies there is some restriction of the variables $y_1, \dots y_9$ which yields 3 unsatisfying assignments and 1 satisfying assignment. One can verify that for any set of 3 unsatisfying assignments y_1, y_2, y_3 for P, they can be captured exactly with an affine predicate (i.e., one can construct an affine predicate $\hat{P}$ which is unsatisfied if and only if the inputs are z_1, z_2, or z_3). But, any predicate which is affine does not have a projection to AND_2. Thus, there is no projection of $\hat{P}$ which yields an AND_2, and consequently no restriction of the variables $y_1, \dots y_9$ for which there are ≥ 3 surviving unsatisfying assignments. □

E Symmetric Predicates with No AND_3 and No Degree 2 Polynomial

Consider the predicate $P : \{0,1\}^r \to \{0,1\}$ (for instance, with $r = 20$) such that $P(x) = 0$ if $|x| = 0 \mod 6$ or $|x| = 1 \mod 6$, and otherwise, $P(x) = 1$. Clearly, P is a symmetric predicate, as P depends only on the number of 1's in x. Next, we will show that P does not have a projection to AND_3.

Claim E.1. *P does not have a projection to AND_3.*

Proof. Recall that we define a projection as a fixing π of the variables $x_1, \dots x_r$ such that each x_i maps to $0, 1, Y_1, Y_2, Y_3$ (or the negation). In particular, in order to get an AND of arity 3, it must be the case that $P(\pi(x_1), \dots \pi(x_r)) = \text{AND}(Y_1, Y_2, Y_3)$. Let us consider any such restriction π, and suppose that it sends α_0 of the x_i's to 0, α_1 of the x_i's to 1, α_{Y_i} to Y_i, and α_{1-Y_i} to $1 - Y_i$.

Also, recall that because P is symmetric, we can equivalently create the predicate $P_0 : [r] \to \{0,1\}$, such that $P(x) = P_0(|x|)$. In terms of the Y_i's then, we have that

$$P(\pi(x_1), \dots \pi(x_r)) = P_0(\alpha_0 \cdot 0 + \alpha_1 \cdot 1 + \alpha_{Y_1} \cdot Y_1 + \alpha_{1-Y_1}(1-Y_1) + \alpha_{Y_2} \cdot Y_2 + \alpha_{1-Y_2}(1-Y_2) + \alpha_{Y_3} \cdot Y_3 + \alpha_{1-Y_3}(1-Y_3))$$

$$= P_0(\alpha_1 + \alpha_{1-Y_1} + \alpha_{1-Y_2} + \alpha_{1-Y_3} + (\alpha_{Y_1} - \alpha_{1-Y_1}) \cdot Y_1 + (\alpha_{Y_2} - \alpha_{1-Y_2}) \cdot Y_2 + (\alpha_{Y_3} - \alpha_{1-Y_3}) \cdot Y_3)$$

$$= P_0(a + bY_1 + cY_2 + dY_3),$$

for some choice of constants a, b, c, d. WLOG let us assume that $a = 0$ or 1, since the predicate is periodic.

Now, for $Y_1 = Y_2 = Y_3 = 0$, we know that the expression must evaluate to 0, as $\text{AND}(0,0,0) = 0$. Thus, $P_0(a) = 0$. Further, when exactly 1 or exactly 2 of Y_1, Y_2, Y_3 are 1, the expression must also

be 0. Hence, $P_0(a+b) = P_0(a+c) = P_0(a+d) = P_0(a+b+c) = P_0(a+c+d) = P_0(a+b+d) = 0$, yet $P_0(a+b+c+d) = 1$.

We consider cases, based on whether $a = 0$ or 1:

1. $a = 0$. Then $P_0(b) = P_0(c) = P_0(d) = P_0(b+c) = P_0(c+d) = P_0(b+d) = 0$. In particular, all of b, c, d must be $0, 1 \mod 6$. In fact, at most one of them can be $1 \mod 6$, as if 2 are $1 \mod 6$, then their sum would be $2 \mod 6$, and P_0 would evaluate to 1. But, if at most one of them is $1 \mod 6$, then their sum is also either $0, 1 \mod 6$, and hence $P_0(b+c+d) = 0$, so it doesn't simulate AND_3.

2. $a = 1$. Then $P_0(1+b) = P_0(1+c) = P_0(1+d) = P_0(1+b+c) = P_0(1+c+d) = P_0(1+b+d) = 0$. In particular, all of b, c, d must be $-1, 0 \mod 6$. In fact, at most one of them can be $-1 \mod 6$, as if 2 are $-1 \mod 6$ (say b, c), then $1 + b + c = -1 \mod 6$, and P_0 would evaluate to 1. But, if at most one of them is $-1 \mod 6$, then their sum is also either $0, 1 \mod 6$, and hence $P_0(1+b+c+d) = 0$, so it doesn't simulate AND_3.

This concludes the proof. □

Simultaneously, there is no canonical way to express P as a polynomial. Indeed, what we would like to do is write P as the product of two linear functions, one which is 0 if and only if $x = 0 \mod 6$, and the other which is 0 if and only if $x = 1 \mod 6$. That is, we would like to simply sparsify the predicate

$$P'(x) = \mathbf{1}[(|x|) \cdot (|x| - 1) \neq 0 \mod 6].$$

However, this predicate unfortunately *does not* capture the behavior of P. For instance, when $|x| = 4$, we get that $P'(x) = 0$, whereas $P(x) = 1$.

We present this more formally below, due to Swastik Kopparty:

Claim E.2. *There is no symmetric degree 2 polynomial f such that for $|x| = 0, 1 \mod 6$ $f(x) = 0$, and otherwise $f(x) = 1$.*

Proof. Indeed, because f is symmetric, f depends only on the hamming weight of x. Because f is degree 2, f takes the form $a \cdot \binom{|x|}{2} + b \cdot |x| + c$ (over some group). When $|x| = 0$, we know f must be 0, which means that $c = 0$. Likewise, when $|x| = 1$, f must also be 0, which means that $b = 0$. So, f can only be a polynomial of the form $f(x) = a \cdot \binom{|x|}{2}$. We also have that for any x, then for any $y : |y| = |x| + 6$, $f(y) = f(x)$. Plugging this in yields that $f(y) = a \cdot \binom{|x|+6}{2} = a \cdot (\binom{|x|}{2} + 6|x| + 15) = a \cdot \binom{|x|}{2} = f(x)$. Thus, it must be the case that over any group we consider, $a \cdot (6|x| + 15) = 0$ (for all $|x|$), which immediately implies that $3a = 0$. By cases then, if we assume $2a = 0$ it must be that $a = 0$, and the polynomial is identically 0, which does not work. Otherwise, the output of f only has a period of 3, as it depends only on whether $\binom{|x|}{2} \mod 3 = 0, 1, 2$, and thus fails to capture our predicate. □

To conclude, the predicate has a period of 6 and therefore seemingly requires a period of 6 (i.e., over $\mathbb{Z}_6$) in whichever polynomial we use to represent it. At the same time, because the period is composite, the polynomial over $\mathbb{Z}_6$ has extra zeros, meaning we do not accurately capture the behavior of the predicate P.