Algorithms for generating cover-free families

by

Matthew Joseph Demczyk

Thesis submitted to the University of Ottawa
in partial fulfillment of the requirements for the degree of
Master of Computer Science

Abstract

Cover-free families (CFFs) are a combinatorial design used in many applications, including group testing and cryptography such as encryption and signature schemes. A d-cover-free family, denoted d-CFF(t,n), is a set system where the underlying set has t elements, the set system has n subsets, and no subset is contained in the union of any d other subsets. When cover-free families are used in applications, it is generally desirable to maximize the number of subsets for a given set of underlying elements. These subsets correspond to samples in group testing, allowing more efficient testing schemes.

Currently, there are no publicly available tables of cover-free families that show the best-known cover-free family for specific parameters. This creates a challenge when implementing applications using cover-free families, since there are a variety of sources and constructions for cover-free families that would be useful. Tables for other types of combinatorial designs are publicly available, such as covering arrays, and these tables provide useful information to researchers.

In this thesis, we outline a selection of constructions of cover-free families, then create and implement algorithms to create tables of best-known cover-free families from these selected constructions. Our selection of direct constructions includes constructions from Sperner systems, packing designs, Reed-Solomon codes, and linear error correcting codes meeting the Gilbert-Varshamov bound constructed using the method of conditional probabilities. Our selection of recursive constructions include the Kronecker product, the Optimized Kronecker product, an additive construction, a doubling construction for $d = 2$, and an extension-by-one construction, which create larger cover-free families from smaller ones.

Using these selected constructions, we iterate over combinations of parameters for each construction to create tables of best-known cover-free families from these constructions. Furthermore, we design and implement a recursive algorithm to construct any chosen cover-free family in our tables. Our results include tables of cover-free families for n up to 10 trillion, and d up to 25. The maximum t appearing in any of our tables is $18,744$ for $d = 25$.

Acknowledgements

I would like to extend my sincere gratitude to those who supported me most during my academic journey. My supervisor, Dr. Lucia Moura, provided exceptional mentorship, patience, and support that I could always rely on. My parents, who gave unwavering support, were instrumental to my success.

My fellow graduate students in Dr. Lucia Moura's research group, Kianoosh, Prangya, Mico, and Josh were always there to lend me a hand. Their technical expertise and reliable support were invaluable.

My course professors, Dr. Amiya Nayak, and Dr. Carlisle Adams, taught exceptional courses that inspired and motivated me.

I thank the examining committee members, Dr. Vida Dujmovic, and Dr. Pat Morin, for providing thoughtful feedback, which helped to refine the thesis.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

In this thesis, we look at cover-free families, a type of combinatorial design that generalizes antichains in partially ordered sets. Our main objective is to study various algorithms to construct cover-free families, and create tables of best-known bounds on the size of cover-free families from these constructions. In Chapters 2 and 3, we discuss constructions for cover-free families. In Chapter 4, we discuss algorithms to create tables of best-known bounds on the size of cover-free families from these constructions. Finally, in Chapter 5, we present and discuss these tables.

1.1 Cover-free families

A *set system* is a pair $(X, \mathcal{B})$ where X is a universal set, and $\mathcal{B}$ is a collection of subsets of X. Each subset in $\mathcal{B}$ is called a *block*. A set system can be represented as a 0-1 incidence matrix. A 0-1 incidence matrix for a set system has $|X|$ rows and $|\mathcal{B}|$ columns, where each column corresponds to one block, and each row corresponds to an element in the universal set. A cell is a 1 if the element corresponding to the row is present in the block corresponding to the column, otherwise it is a 0.

A set system is a d-cover-free family (d-CFF) if no block is contained in the union of d other blocks.

Definition 1.1 (d-Cover-Free Family)**.** *Let $d < t \leq n$. A set system $(X, \mathcal{B})$ with $|X| = t$ and $|\mathcal{B}| = n$ is a d-cover-free family, denoted d-CFF(t,n), if for any block $B_{i_0} \in \mathcal{B}$ and any other d blocks $B_{i_1}, \ldots, B_{i_d} \in \mathcal{B}$, the following is true*

$$\left| B_{i_0} \setminus \bigcup_{j=1}^{d} B_{i_j} \right| \geq 1 . \tag{1.1}$$

CFFs are often represented as incidence matrices. Throughout this thesis, we will also refer to the incidence matrix of a d-CFF as a d-CFF. An equivalent definition to Definition 1.1 can be given by requiring properties of the incidence matrix. Recall that the weight of a tuple is the number of nonzero components in the tuple.

Definition 1.2. *A $t \times n$ 0-1 matrix A is a d-CFF(t, n) if for any $(d+1)$-subset of columns, each weight-1 $(d+1)$-tuple is present in at least one row of the sub-matrix indexed by these columns.*

Figure 1.1 is a 3-CFF(20,25), represented as a set system. In this example, $X = \{1, 2, \ldots, 20\}$, and $|B| = 25$. It is constructed from a Reed-Solomon code (Section 2.1.3.1) with $q = 5$, $k = 2$, and $m = 4$. Figure 1.2 shows the incidence matrix for this 3-CFF(20,25).

$B_1 = \{1, 6, 11, 16\}$ $B_2 = \{2, 7, 12, 17\}$ $B_3 = \{3, 8, 12, 18\}$ $B_4 = \{4, 9, 14, 19\}$
$B_5 = \{5, 10, 15, 20\}$ $B_6 = \{1, 7, 13, 19\}$ $B_7 = \{2, 8, 14, 20\}$ $B_8 = \{3, 9, 15, 16\}$
$B_9 = \{3, 10, 11, 17\}$ $B_{10} = \{5, 6, 12, 18\}$ $B_{11} = \{1, 8, 15, 17\}$ $B_{12} = \{2, 9, 11, 18\}$
$B_{13} = \{3, 10, 12, 19\}$ $B_{14} = \{4, 6, 13, 20\}$ $B_{15} = \{5, 7, 14, 16\}$ $B_{16} = \{1, 9, 12, 20\}$
$B_{17} = \{2, 10, 14, 16\}$ $B_{18} = \{3, 6, 14, 17\}$ $B_{19} = \{4, 7, 15, 18\}$ $B_{20} = \{5, 8, 11, 19\}$
$B_{21} = \{1, 10, 14, 18\}$ $B_{22} = \{2, 6, 15, 19\}$ $B_{23} = \{3, 7, 11, 20\}$ $B_{24} = \{4, 8, 12, 16\}$
$B_{25} = \{5, 9, 13, 17\}$

Figure 1.1: 3-CFF(20,25) set system

	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}	B_{11}	B_{12}	B_{13}	B_{14}	B_{15}	B_{16}	B_{17}	B_{18}	B_{19}	B_{20}	B_{21}	B_{22}	B_{23}	B_{24}	B_{25}
1	**1**	**0**	**0**	**0**	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0
2	**0**	**1**	**0**	**0**	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0
3	**0**	**0**	**1**	**0**	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0
4	**0**	**0**	**0**	**1**	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0
5	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1
6	1	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0
7	0	1	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0
8	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1	0
9	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	1
10	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0
11	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0
12	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0
13	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1
14	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0
15	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0
16	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0
17	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1
18	0	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0
19	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1	0	0	0
20	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0

Figure 1.2: 3-CFF(20,25) incidence matrix

In this incidence matrix representation, the cover-free family property (Definition 1.1) is the same as requiring that in every $(d+1)$-subset of columns, each weight-1 0-1 tuple is present as a row (Definition 1.2). For example, consider $d = 3$ and $i_0 = 1, i_1 = 2, i_2 = 3, i_3 = 4$ in Equation 1.1. This equation says that there exists at least one element e in B_1 that is not in any of B_2, B_3, B_4. So we must find a row in the incidence matrix, corresponding to e with $(1, 0, 0, 0)$ under columns B_1, B_2, B_3, B_4; in this example $e = 1$ is one such row. The same should be true if in turn we take $i_0 = 2, 3$ or 4 and $\{i_1, i_2, i_3\} \setminus \{i_0\}$, implying

the existence of rows containing $(0,1,0,0),(0,0,1,0)$ and $(0,0,0,1)$ under these columns. These rows are marked in bold in Figure 1.2. This exemplifies what Definition 1.2 requires of any set of $(d+1)$ columns for a matrix to be a d-CFF. To complete the verification for this example, one needs to check the existence of these rows under each 4-set of columns, i.e for each of the $\binom{25}{4} = 12650$ combinations of 4 columns.

When using CFFs, we are usually interested in minimizing the number of rows for a given n and d. Let $t(d,n)$ be the minimum t such that a d-CFF(t,n) is exists, i.e.

$$t(d,n) \quad = \quad \min\{t \mid \exists d\text{-CFF}(t,n)\}. \tag{1.2}$$

Similarly, we define $n(d,t)$ as the maximum n for which a d-CFF(t,n) exists, i.e.

$$n(d,t) \quad = \quad \max\{n \mid \exists d\text{-CFF}(t,n)\}. \tag{1.3}$$

Proposition 1.3.

$$t(d,n) \quad = \quad \min\{t \mid n \leq n(d,t)\} \tag{1.4}$$
$$n(d,t) \quad = \quad \max\{n \mid t \geq t(d,n)\}. \tag{1.5}$$

Proof. Let $d < t \leq n$. If a d-CFF(t,n) exists, then by Equation 1.3, $n \leq n(d,t)$. Conversely, if $n \leq n(d,t)$, then a d-CFF(t,n) exists, since eliminating some columns from a d-CFF produces a d-CFF. Thus, there exists a d-CFF(t,n) if and only if $n \leq n(d,t)$. Therefore, combining this with Equation 1.2 gives Equation 1.4.

Parallel to this, for $t(d,n)$, if a d-CFF(t,n) exists, then by Equation 1.2, $t \geq t(d,n)$. Conversely, if $t \geq t(d,n)$, then a d-CFF(t,n) exists, since we can always add redundant rows to a CFF. Thus, there exists a d-CFF(t,n) if and only if $t \geq t(d,n)$. Therefore, combining this with Equation 1.3 gives Equation 1.5. □

The best-known bounds for $t(d,n)$ are as follows, retrieved from [17]. For $d = 1$, $t(1,n) \sim \log n$. For $d = 2$, there exist constants c_1, c_2 such that:

$$c_1 \cdot \frac{d^2}{\log d} \log n \quad \leq t(d,n) \leq \quad c_2 \cdot d^2 \log n.$$

The upper bound is given by Porat and Rothschild in [23].

1.2 Applications

Cover-free families have a wide range of applications, including security, privacy, and medical applications using combinatorial group testing. The reader if referred to [17] for a survey of CFF applications.

Some examples of applications in cryptography include batch verification of digital signatures [22,29], fault-tolerant digital signature schemes [13,15], key distribution [11,12], anti-jamming systems [8], and new public key encryption schemes [5].

We outline how CFFs are applied to combinatorial group testing in the following section.

1.2.1 Combinatorial group testing

Combinatorial group testing (CGT) is an approach used when we want to know the outcome of a test for each element of a set, without having to run the test on each individual element to determine whether the element is positive (such as a blood sample containing a virus, or an item being defective). Whether or not CGT is applicable to a type of testing depends on the nature of the test.

CGT is applicable if it is possible to combine the elements into groups to be tested together, such that the result of the combined group will be positive if at least one element is positive, and negative if no elements are positive in the combined group. This property allows for one test to determine the outcome for many elements if they are all negative.

An example of this type of test is blood testing for a virus. If a person has a virus and their blood is mixed with other peoples' blood, and then the mixed blood is tested for the virus, the test would be positive. If no one is sick, the test would be negative.

CGT is classified into two categories: adaptive combinatorial group testing, and non-adaptive combinatorial group testing. For more information on combinatorial group testing see [9,10].

Adaptive group testing allows for the outcomes of some tests to be determined before others, which allows the group testing scheme to be adapted depending on the results of some tests. In contrast, non-adaptive group testing has its test cases determined before starting, allowing the tests to be completed in parallel. Adaptive testing might use fewer tests, but it takes longer, since the tests cannot be completed in parallel.

The best algorithm for adaptive group testing uses the binary splitting approach. This is a recursive algorithm that takes a set of elements to be tested as a parameter. The algorithm works as follows: the base case is if the set only has one element. In this case, report the outcome of that test. Otherwise, test the set of elements as a group. If negative, report that all elements in the set are negative. If positive, split the set into two equally sized sets, and recurse on each set. Let n be the number of items to be tested, and d be the number of positive elements. This takes $O(d \log n)$ tests [10].

Non-adaptive group testing can be done using cover-free families, assuming we know an upper bound on the number of positive elements. The main idea is to eliminate all the elements in test results that are negative all at once. In a d-CFF, no block is contained in the union of d other blocks. This property of CFFs ensures there exist a subset of tests that do not contain any of the d positive elements, allowing the negative test results to eliminate all negative elements. It is known that this can be done in $O(d^2 \log n)$ tests (see [17]).

For example, we will look at the 3-CFF(20,25) from Section 1.1, where B_0, B_1, and B_2 are 3 people infected with a virus. The tests (rows) that contain any of the positive elements are bold in the next figure (positive tests). As implied by the definition of a CFF, every other B_i is present in a negative test (a row not bold). This means that every negative person has been tested and appears in a negative test. Using a CFF for group testing in this example allows the positive elements to be determined using only 20 tests, instead of the 25 that would be required if they were tested individually.

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}	B_{11}	B_{12}	B_{13}	B_{14}	B_{15}	B_{16}	B_{17}	B_{18}	B_{19}	B_{20}	B_{21}	B_{22}	B_{23}	B_{24}
1	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**
2	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**
3	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**
4	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0
5	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1
6	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**0**
7	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**0**
8	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**0**
9	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	1
10	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0
11	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**
12	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**
13	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**
14	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0
15	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0
16	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**
17	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**
18	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**
19	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1	0	0	0
20	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0

The decoding procedure for group testing results from CFFs is as follows. Let A be a d-CFF(t, n). The *vector of test results*, $y \in \{0,1\}^t$, obtained from applying A to a group testing problem is defined as:

$$y_i \quad = \quad \begin{cases} 1, \text{ if test } \{j \in X | A_{i,j} = 1\} \text{ is positive} \\ 0, \text{ otherwise.} \end{cases} \tag{1.6}$$

The proof for the following Proposition 1.4 is from [20].

Proposition 1.4. *Let A be a d-CFF(t, n), let X be a set of n elements with at most d positive items, and let y be the vector of test results of group testing defined in Equation 1.6. Then, the set P of positive items is*

$$P \quad = \quad X \setminus \left(\bigcup_{i|y_i=0} \{j \in X | A_{ij} = 1\} \right).$$

Proof. Let $N = \bigcup_{i|y_i=0} \{j \in X | A_{ij} = 1\}$. It is clear that the items with $y_i = 0$ are negative, since they appear in negative tests. It remains to be shown that every negative item will appear in N.

Let $j_1, \dots j_d$ be the positive elements in X, and j be a negative element. Since in a CFF any block is not contained in the union of d other blocks, there will always be a test i with $A_{ij} = 1$ and $A_{ij_1} = \dots = A_{ij_d} = 0$. This results in a negative test since the positive elements are not tested in this row. So $y_i = 0$ and $j \in N$. □

1.3 Thesis organization

In Chapter 2, we present a selection of direct and recursive constructions for CFFs. These direct constructions include constructions from Sperner systems, packing designs, and error-correcting codes. Our selection of recursive constructions include the Kronecker product, the Optimized Kronecker product, an additive construction, a doubling construction for $d = 2$, and an extension-by-one construction.

In Chapter 3 we detail a method of constructing CFFs using the probabilistic method proposed by Porat and Rothschild in 2011 [23]. This method matches the best asymptotic upper bound for $t(d, n)$, giving a d-CFF(t, n) with t in $O(d^2 \log n)$. One of the contributions of this thesis is a detailed explanation of the method in [23], including a pseudocode for their algorithm, which is not so obvious to derive from [23] without a careful analysis.

In Chapter 4, we discuss the algorithms that we designed to create tables of best-known CFFs for various parameters, and in Chapter 5 we present and discuss these tables. Chapters 4 and 5 are the main contributions of this thesis.

Finally, in Chapter 6 we present conclusions and discuss future work.

We present our CFF tables in Chapter 5 and in the Appendix. We made the tables available online at [6]. All the algorithms discussed in this thesis are available at [7] implemented in C. Our C implementation relies on the C library FLINT: Fast Library for Number Theory [26], for finite field operations.

Chapter 2

Cover-free families and constructions

For a survey on cover-free families and constructions, the reader is referred to [17]. Most constructions from [17] are presented here. In this chapter, we provide pseudocode for most of the considered construction necessary to obtain the parameters in our tables. The constructions that we do not give pseudocode for are visualized with simple diagrams instead. All code written for this thesis is available in [7]. This includes all selected constructions discussed in this chapter implemented in C. These algorithms for constructing CFFs are invoked in Algorithm 4.13 in Section 4.7.

2.1 Direct constructions

In this section, we discuss constructions for cover-free families from various other combinatorial objects.

2.1.1 Construction from Sperner systems for $d = 1$

An *antichain* is a set system $(X, \mathcal{B})$ where X is finite, and no subset in $\mathcal{B}$ contains another. The *size* of the antichain $(X, \mathcal{B})$ is $|\mathcal{B}|$. Sperner's Theorem [24] shows the largest size of an antichain with $|X| = t$. Antichains are also called *Sperner systems* or *Sperner families*; we refer to antichains as Sperner systems throughout this thesis.

Theorem 2.1 (Sperner's theorem (1928) [24])**.** *The size of a largest Sperner system with* $|X| = t$ *is* $\binom{t}{\lfloor \frac{t}{2} \rfloor}$.

A proof of Sperner's theorem can be found in [1].

Being a 1-CFFs is the same as being a Sperner system.

Proposition 2.2. *Let $(X, \mathcal{B})$ be a set system with $|X| = t$ and $|\mathcal{B}| = n$. Then $(X, \mathcal{B})$ is a Sperner system if and only if $(X, \mathcal{B})$ is a 1-CFF(t, n). Therefore,*

$$\begin{aligned} n(1,t) &= \binom{t}{\lfloor \frac{t}{2} \rfloor}, \text{ and} \\ t(1,n) &= \min\{t : \binom{t}{\lfloor \frac{t}{2} \rfloor} \geq n\}. \end{aligned}$$

Proof. Let us prove the equivalence of a set system being a 1-CFF and being a Sperner system. Any two columns in a 1-CFF's incidence matrix must each contain one row that has a 0 in the first column and a 1 in the second column, and another row that has a 1 in the first column and a 0 in the second column. As a result, any two sets corresponding to two columns in a 1-CFF do not contain each other, meaning that the 1-CFF is a Sperner system.

A Sperner system is always a 1-CFF for the same reason. Because no subset in a Sperner system can contain another, each pair of subsets must contain one element that the other does not, making the incidence matrix of a Sperner system a 1-CFF.

Given this equivalence between Sperner systems and 1-CFFs, Sperner's theorem can be rewritten as $n(1,t) = \binom{t}{\lfloor \frac{t}{2} \rfloor}$. This and equation 1.5 implies the second equation. □

When constructing a Sperner system with $|X| = t$ elements, the k-subset lexicographic successor algorithm (see [18]), given in Algorithm 2.1, can be used to generate the subsets in $\mathcal{B}$. The subset size for each set in the largest Sperner system with $|X| = t$ is $k = \lfloor \frac{t}{2} \rfloor$.

Algorithm 2.1 k-subset-lex-successor(t, k, subset)

Input: $t, k \in \mathbb{Z}^+, t > k$, subset is a k-length array of integers in $[0, t-1]$
Output: Returns true if subset has a successor, and false otherwise. If a successor exists, subset is modified to be this successor.

```
1: for i ← k − 1, . . . , 0 do
2:     if subset[i] ≠ t − k + i then
3:         subset[i]←subset[i]+1
4:         for x ← i + 1, . . . , k − 1 do
5:             subset[x]←subset[i]+x + i
6:         end for
7:         return True
8:     end if
9: end for
10: return False
```

Using this successor algorithm, we iterate over all subsets in a Sperner system and assign them to a column in the CFF, as shown in Algorithm 2.2.

Algorithm 2.2 SpernerConstruction(t)

Input: $t \in \mathbb{Z}^+$
Output: 1-CFF($t, \binom{t}{\lfloor \frac{t}{2} \rfloor}$)

CFF $\leftarrow t \times \binom{t}{\lfloor \frac{t}{2} \rfloor}$ matrix of 0s
$S \leftarrow [0, 1, \ldots, \lfloor t/2 \rfloor - 1]$
$j \leftarrow 0$
repeat
  for $i \leftarrow 0, 1, \ldots, \lfloor t/2 \rfloor - 1$ **do**
    CFF[$S[i]$] [j] $\leftarrow 1$
  end for
  $j \leftarrow j + 1$
until k-subset-lex-successor($t, \lfloor t/2 \rfloor, S$) = False
return CFF

The following in an example of a 1-CFF(4, 6) and its corresponding subsets that are used to construct it. The rows of A are the lexicographic 2-subsets of $\{0, 1, 2, 3\}$ generated by Algorithm 2.1, and B is the resulting 1-CFF after applying Algorithm 2.2.

$$A = \begin{array}{cc} a_1 & a_2 \\ \hline 0 & 1 \\ 0 & 2 \\ 0 & 3 \\ 1 & 2 \\ 1 & 3 \\ 2 & 3 \end{array}, \quad \text{and}$$

$$B = \begin{array}{cccccc} b_1 & b_2 & b_3 & b_4 & b_5 & b_6 \\ \hline 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{array}.$$

2.1.2 Construction from packing designs

A t-(v, k, λ) packing design with b blocks is a set system $(X, \mathcal{B})$, such that the following properties hold:

- $|X| = v$,
- $|\mathcal{B}| = b$,
- For all $B \in \mathcal{B}$, $|B| = k$,

- Every t-subset of points in X is contained in at most λ blocks of $\mathcal{B}$.

Packing designs with $\lambda = 1$ are used to construct CFFs. The corresponding set system is a $\lfloor \frac{k-1}{t-1} \rfloor$-CFF$(v, b)$. The main idea is that when $\lambda = 1$, in any $\lfloor \frac{k-1}{t-1} \rfloor + 1$ subsets of blocks, each block will have at least one unique element. That unique element guarantees that the cover-free property is satisfied. The following proof for Proposition 2.3 is from [28], but we add more details for each step.

Proposition 2.3. *A $t-(v, k, 1)$ packing design with $|X| = v$ and $|\mathcal{B}| = b$ is a $\lfloor \frac{k-1}{t-1} \rfloor$-CFF$(v, b)$.*

Proof. Because $\lambda = 1$, in any pair of blocks $B_x, B_y \in \mathcal{B}$, a t-set occurs at most once. So, $|B_x \cap B_y| \leq t - 1$. Let $d = \lfloor \frac{k-1}{t-1} \rfloor$. Recall $|B| = k$ for all $B \in \mathcal{B}$. Thus for any distinct $B_0, B_1, \ldots, B_d \in \mathcal{B}$, we can conclude that:

$$\begin{aligned}
\left| B_0 \setminus \bigcup_{i=1}^{d} B_i \right| &= \left| B_0 \setminus (B_0 \cap \bigcup_{i=1}^{d} B_i) \right| \\
&= |B_0 \setminus (B_0 \cap (B_1 \cup B_2 \cup \ldots \cup B_d))| \\
&= |B_0 \setminus ((B_0 \cap B_1) \cup (B_0 \cap B_2) \cup \ldots \cup (B_0 \cap B_d))| \\
&= |B_0| - |(B_0 \cap B_1) \cup (B_0 \cap B_2) \cup \ldots \cup (B_0 \cap B_d)| \\
&\geq |B_0| - |(B_0 \cap B_1)| - |(B_0 \cap B_2)| - \ldots - |(B_0 \cap B_d)| \\
&\geq k - d(t-1) \\
&\geq k - \frac{k-1}{t-1}(t-1) \\
&= 1.
\end{aligned}$$

□

Example 2.4. *A Steiner triple system is a set system where all blocks have length 3, and each 2-subset of v occurs in exactly one block [25]. Therefore, a Steiner triple system of order v, STS(v), is a $2-(v, 3, 1)$ packing design. Every STS has $t = 2$ and $k = 3$, so by Proposition 2.3 they are always 2-CFFs, because $d = 2 = \lfloor \frac{k-1}{t-1} \rfloor$ when constructing a CFF from a packing design.*

The following is an example of an STS(9) and the 2-CFF(9,12) that it can construct. These rows of A are the 12 blocks of STS(9), and B is the resulting CFF after applying Proposition 2.3. The resulting set system of the CFF is the set system of the STS.

$$A = \begin{array}{ccc} a_1 & a_2 & a_3 \\ \hline 1 & 4 & 7 \\ 1 & 2 & 6 \\ 4 & 5 & 9 \\ 7 & 8 & 3 \\ 1 & 3 & 5 \\ 4 & 6 & 8 \\ 7 & 9 & 2 \\ 2 & 5 & 8 \\ 2 & 3 & 4 \\ 5 & 6 & 7 \\ 8 & 9 & 1 \\ 3 & 6 & 9 \end{array}$$

$$B = \begin{array}{cccccccccccc} b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 & b_8 & b_9 & b_{10} & b_{11} & b_{12} \\ \hline 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{array}.$$

STSs can be used to construct infinite classes of 2-CFFs, because they are known to exist if and only if $v \equiv 1, 3 \pmod 6$, and there are simple constructions from Bose when $v \equiv 3 \pmod 6$ and Skolem when $v \equiv 1 \pmod 6$ (see [25]). However, we have observed experimentally that CFFs from STSs only appear in our tables of best-known CFFs (Chapter 5) when $t = 9$; for $t > 9$, CFFs from other constructions have larger n for the same t compared to CFFs from STSs.

2.1.3 Construction from codes

The *hamming distance* between two tuples is the number of positions in each tuple that are not equal. For example, let $v_1 = [0, 0, 0]$ and $v_2 = [1, 0, 2]$. Since v_1 and v_2 are different in 2 positions, their hamming distance is $d(v_1, v_2) = 2$.

An $(m, N, D)_q$-code C over an alphabet Q is a collection of $C \subseteq Q^m$, where $|C| = N$, $|Q| = q$, and D is its *minimum distance*, i.e. $D = \min\{d(x, y) : x, y \in C, x \neq y\}$. The elements of C are called codewords, and m is the length of a codeword. Error-correcting codes are used for transmitting a message over a noisy channel. A message with k symbols

can be encoded into a codeword with m symbols, with $m > k$. This redundancy permits error correction. A minimum distance of D allows the correction of up to $\lfloor \frac{D-1}{2} \rfloor$ errors, or detection of $D-1$ errors [21].

Let us consider codes over a finite field, i.e. $Q = \mathbb{F}_q$, where $\mathbb{F}_q$ is the field with q elements for a prime power q. A code is *linear* if each linear combinations of codewords is also a codeword. Therefore, a linear code is a subspace of $\mathbb{F}_q^m$; let k be the dimension of this subspace. An $[m, k, D]_q$ linear code is an $(m, q^k, D)_q$-code. A linear code can encode messages of length k into codewords of length m. Each message is a vector in $\mathbb{F}_q^k$. The set of messages consists of all vectors of $\mathbb{F}_q^k$. The following matrix M has in its rows each of the message vectors for a code with $k = 2$ and $q = 3$.

$$M = \begin{array}{cc} m_1 & m_2 \\ \hline 0 & 0 \\ 0 & 1 \\ 0 & 2 \\ 1 & 0 \\ 1 & 1 \\ 1 & 2 \\ 2 & 0 \\ 2 & 1 \\ 2 & 2 \end{array}.$$

Linear codes have an associated generator matrix. A linear code is the subspace spanned by the rows of its generator matrix. The following is an example of a generator matrix for a $[3, 2, 2]_3$ linear code:

$$G = \begin{array}{ccc} c_1 & c_2 & c_3 \\ \hline 1 & 1 & 0 \\ 0 & 1 & 1 \end{array}.$$

To determine all codewords from a code's generator matrix, multiply the matrix of message vectors by the generator matrix. For the examples of M and G above, this gives the $[3, 2, 2]_3$ linear code, which is a $(3, 8, 2)_3$ code, given in Figure 2.1. Alternatively, every linear combination of the vectors in the generator matrix can be computed, for the same result. This code's minimum distance of 2 can be verified by examining the hamming distance of every pair of codewords.

The *weight* of a codeword is the number of non-zero positions in the codeword.

Proposition 2.5. *The hamming distance between two codewords in a linear code equals the weight of their element-wise difference, i.e, $d(x, y)$=$w(x - y)$.*

$$M \cdot G = \begin{array}{ccc} c_1 & c_2 & c_3 \\ \hline 0 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & 2 & 0 \\ 0 & 1 & 1 \\ 1 & 2 & 1 \\ 2 & 0 & 1 \\ 0 & 2 & 2 \\ 1 & 0 & 2 \\ 2 & 1 & 2 \end{array}.$$

Figure 2.1: A $[3, 2, 2]_3$ linear code where each row is a codeword

Proof. Let x and y be codewords of a linear code. If x and y have the same letter in some position, the difference between those letters will be zero, otherwise non-zero. Thus, the codeword $x - y$ will be non-zero precisely in the positions that have different letters in the two codewords, so the weight of $x - y$ will be equal to the distance between x and y. □

Proposition 2.6 implies that knowing that the weight of every non-zero codeword in a linear code is at least D is enough to ensure that its minimum distance is at least D. This property of linear codes is used in Chapter 3.

Proposition 2.6. *The minimum weight over every non-zero codeword in a linear code is equal to the code's minimum distance.*

Proof. Let x and y be distinct codewords in a linear code C. In a linear code, every codeword z of the form $z = x - y$ is present, because every linear combination of codewords is also a code. In addition, since 0^m is always a codeword, any codeword can be written as $z = x - y$, where $x = z$ and $y = 0^m$. Let $d(x, y)$ represent the distance between two codewords x and y, and $w(x)$ represent the weight of a codeword x. The minimum distance in a code is defined as $D = \min\{d(x, y)|x, y \in C, x \neq y\}$. From Proposition 2.5, we know that $d(x, y) = w(x - y)$. Therefore, $D = \min\{w(x - y)|x, y \in C, x \neq y\} = \min\{w(z)|z \in C, z \text{ is a non-zero codeword}\}$.

□

Codes, linear or not, can be used to construct CFFs. The following proof for Proposition 2.7 is from [28].

Proposition 2.7. *If there exists an $(m, N, D)_q$-code, then there exists a $\lfloor \frac{m-1}{m-D} \rfloor$-CFF$(mq, N)$.*

Proof. First, construct a packing design from the code C by transforming each codeword $c = (c_1, c_2, \ldots, c_m) \in C$ into a block $B = \{(i, c_i) : 1 \leq i \leq m\}$. Since each pair of codewords differs in at least D positions, each pair of blocks will have at most $m - D$ elements in

common. Thus, any $(m - D + 1)$-subset of points appears in at most one block, so the blocks form a $(m - D + 1)$-$(mq, m, 1)$ packing design with N blocks. Use Proposition 2.3 to obtain a $\lfloor \frac{m-1}{m-D} \rfloor$-CFF$(mq, N)$. □

A CFF is constructed from codes with the following Algorithm 2.3. The alphabet of the code, $\mathbb{F}_q$, must be converted to the set of all integers in $[0, q-1]$ before using this algorithm. Accessing a code by an index r in this algorithm refers to retrieving the r-th codeword: $C[r]$. Accessing a codeword by an index e refers to retrieving the e-th letter of the codeword: $C[r][e]$.

Algorithm 2.3 CFFfromCode(C, m, N, D, q)

Input: $(m, N, D)_q$-code, given as a $N \times m$ matrix C.
Output: a $\lfloor \frac{m-1}{m-D} \rfloor$-CFF$(mq, N)$.

```
1: CFF ← mq × N matrix of 0s
2: for r ← 0, 1, ... N − 1 do
3:     for e ← 0, 1, ... m − 1 do
4:         CFF[eq+ C[r][e]][r] ← 1
5:     end for
6: end for
7: return CFF
```

The following is an example of a 2-CFF(9,9) constructed from the earlier $[3, 2, 2]_3$ code in Figure 2.1 using Algorithm 2.3.

	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9
$(1,0)$	1	0	0	1	0	0	1	0	0
$(1,1)$	0	1	0	0	1	0	0	1	0
$(1,2)$	0	0	1	0	0	1	0	0	1
$(2,0)$	1	0	0	0	0	1	0	1	0
$(2,1)$	0	1	0	1	0	0	0	0	1
$(2,2)$	0	0	1	0	1	0	1	0	0
$(3,0)$	1	1	1	0	0	0	0	0	0
$(3,1)$	0	0	0	1	1	1	0	0	0
$(3,2)$	0	0	0	0	0	0	1	1	1

Two examples of codes used to construct CFFs are Reed-Solomon codes in Section 2.1.3.1, and codes constructed by Porat and Rothschild [23] in Chapter 3.

2.1.3.1 Reed-Solomon code construction

A Reed-Solomon code is a type of linear error-correcting code. Reed-Solomon codes can be constructed by evaluating polynomials of degree at most $k-1$ over a finite field $\mathbb{F}_q$ with various elements of the field.

The construction for Reed-Solomon codes works by associating a unique polynomial to each codeword. Suppose $a_1, \ldots, a_q$ are elements of $\mathbb{F}_q$, and $m \leq q+1$. For each polynomial f of degree at most $k-1$, associate a codeword $c = (c_1, \ldots, c_m)$ where $c_i = f(a_i)$ if $i \leq q$ and $c_i = a$ otherwise, where a is the coefficient of x^{k-1} in f.

Since any two polynomials of degree $k-1$ have at most $k-1$ intersections, each codeword will have at most $k-1$ letters in common. Reed-Solomon codes have a minimum distance of $m-k+1$. When using Reed-Solomon codes with codeword length $q+1$, the extra position of each codeword is the coefficient of x^{k-1}, and the proof of the code's minimum distance becomes more complex. The full proof is available in [14].

When constructing Reed-Solomon codes there are three parameters to specify. The alphabet length q (a prime power), the number k, where each polynomial associated to a codeword has degree at most $k-1$ and the messages of the code have length k, and the length m of each codeword, where $m \leq q+1$ and $q \geq k-1 \geq 0$. The result is a $[m, k, m-k+1]_q$ code. This construction is given in Algorithm 2.4.

The following is an example of a Reed-Solomon code, and is a $[4, 2, 3]_3$ code. The leftmost column indicates the polynomial associated to each codeword. The next three columns correspond to the elements of $\mathbb{F}_3$ to evaluate each polynomial with, and the final column is the coefficient of x^{k-1}. In this example, $q = 3$, $k = 2$, and $m = 4$:

	0	1	2	*
$0x+0$	0	0	0	0
$0x+1$	1	1	1	0
$0x+2$	2	2	2	0
$1x+0$	0	1	2	1
$1x+1$	1	2	0	1
$1x+2$	2	0	1	1
$2x+0$	0	2	1	2
$2x+1$	1	0	2	2
$2x+2$	2	1	0	2

.

Reed-Solomon codes can be shortened. A *shortened* Reed-Solomon code is a subcode of a Reed-Solomon code that only uses the codewords whose first s letters are zero, and removes those s leading zeros from its codewords. Because the minimum weight of each codeword is unchanged, the minimum distance is preserved when shortening the code. To modify Algorithm 2.4 to construct shortened Reed-Solomon codes, we would need an additional if statement before line 6. It would check if the first s letters of the codeword would be zero before constructing a codeword, and would use a **continue** statement, skipping the codeword, if the first s letters of the codeword are not all zero. Additionally, these first s letters of a Reed-Solomon codeword are not included in the codeword for a shortened Reed-Solomon code.

Proposition 2.8. *A Reed-Solomon code $[m, k, m-k+1]_q$ when shortened by s becomes a $[m-s, k-s, m-k+1]_q$ code.*

Algorithm 2.4 ReedSolomonConstruction(q, k, m)

Input: $m \leq q+1$, $q \geq k-1 \geq 0$, and q is a prime power.
Output: A $\lfloor \frac{m-1}{k-1} \rfloor$-CFF$(mq, q^k)$.

```
RScode ← m × q^k matrix of 0s
r ← 0
for each polynomial f ∈ F_q[x] with degree ≤ k − 1 do
    c ← 0
    for each a ∈ F_q do
        RScode[r, c] ← f(a)                ▷ evaluate f(a) using Horner's method
        c ← c + 1
        if c = m then
            break
        end if
    end for
    if m = q + 1 then
        RScode[r, q] ← coefficient x^{k−1} in f
    end if
    r ← r + 1
end for
return CFFfromCode(RScode)
```

The following code is the previous $[4, 2, 3]_3$ code shortened with $s = 1$ resulting in a $[3, 1, 3]_3$ code.

	1	2	*
$0x+0$	0	0	0
$1x+0$	1	2	1
$2x+0$	2	1	2

By using Propositions 2.7 and 2.8, Reed-Solomon codes and shortened Reed-Solomon codes give infinite classes of CFFs for any d.

2.1.4 Construction from binary constant-weight codes for $d = 2$

A binary constant-weight code is an error correcting code with alphabet $\{0, 1\}$, codeword length m, minimum distance D, and constant weight w for every codeword. The number of codewords in the largest known binary constant weight code for some m, D, and w is denoted as $A(m, D, w)$. Best-known lower bounds for $A(m, D, w)$ can be found in [3].

Binary constant-weight codes can be used to construct CFFs with smaller t using Proposition 2.9 instead of Proposition 2.7. Specifically, t will be half of what it would be with Proposition 2.7. The following proof for Proposition 2.9 is from [17].

Proposition 2.9. *Let $x \geq 2$. If there exists a binary constant-weight code with codeword length m, minimum distance $2x$, constant weight $2x-1$, and $|C| = N$, then there exists a 2-CFF(m, N).*

Proof. First, construct a x-$(m, 2x-1, 1)$ packing design with N blocks by setting each column of the packing design's incidence matrix to a codeword. Because the code's minimum distance is $2x$ and each codeword has constant weight $2x-1$, the codewords can have at most $x-1$ positions in common, thus each block of the packing design has each x-subset repeated at most once. Because the codewords have weight $2x-1$, each block of the packing design has length $2x-1$. Use proposition 2.3 to obtain a $\lfloor \frac{2x-2}{x-1} \rfloor$ -CFF(m, N), which is always a 2-CFF(m, N). □

2.2 Recursive constructions

Recursive constructions are constructions that create larger CFFs from one or more smaller CFFs.

2.2.1 Kronecker product

The Kronecker Product is a binary operation over two matrices $P = A \otimes B$. If A is a $m \times n$ matrix and B is a $p \times q$ matrix, then their Kronecker Product P is a $pm \times qn$ matrix.

Definition 2.10 (Kronecker product)**.** *Let B be any matrix, and let*

$$A = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}.$$

Then their Kronecker product $P = A \otimes B$ is found by the following scalar-matrix multiplication:

$$P = \begin{bmatrix} a_{11}B & \dots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \dots & a_{mn}B \end{bmatrix}.$$

Taking the Kronecker product of two CFFs allows us to construct a larger CFF.

Proposition 2.11. *The Kronecker product of a d-CFF(t_1, n_1) and a d-CFF(t_2, n_2) is a d-CFF$(t_1 t_2, n_1 n_2)$. Therefore,*

$$t(d, n_1 n_2) \leq t(d, n_1) \times t(d, n_2).$$

The proof for Proposition 2.11 can be found in [16]. The algorithm below shows how the Kronecker product of two CFFs is computed. In Algorithm 2.5, the two CFFs' incidence matrices are represented as A and B. A is the incidence matrix of a d-CFF(t_1, n_1), and B is the incidence matrix of a d-CFF(t_2, n_2).

Algorithm 2.5 KroneckerProduct$(A, B, d, n_1, n_2, t_1, t_2)$

Input: $n_1, n_2, t_1, t_2 \in \mathbb{Z}^+$, A is a $t_1 \times n_1$ $0-1$ matrix, B is a $t_2 \times n_2$ $0-1$ matrix
Output: The Kronecker product of A and B

```
1: CFF ← t1t2 × n1n2 matrix of 0s
2: for r1 ← 0, 1, ..., s − 1 do
3:     for c1 ← 0, 1, ..., n1 − 1 do
4:         if A[r1][c1] = 1 then
5:             for r2 ← 0, 1, ..., t2 − 1 do
6:                 for c2 ← 0, 1, ..., n2 − 1 do
7:                     if B[r2][c2] = 1 then
8:                         CFF[r1t2 + r2][c1n2 + c2] ← 1
9:                     end if
10:                end for
11:            end for
12:        end if
13:    end for
14: end for
15: return CFF
```

2.2.2 Optimized Kronecker product

The Optimized Kronecker product is a method similar to the Kronecker product, but uses three CFFs instead of two. It uses an extra $(d-1)$-CFF to potentially reduce the total number of rows, while having the same number of columns as the Kronecker product.

Proposition 2.12 (Optimized Kronecker)**.** *If there exists a d-CFF(t_1, n_1), a d-CFF(t_2, n_2), and a $(d-1)$-CFF(s, n_2), then there exists a d-CFF$(st_1 + t_2, n_1 n_2)$. Therefore,*

$$t(d, n_1 n_2) \quad \leq \quad t(d-1, n_2)t(d, n_1) + t(d, n_2).$$

The proof for Proposition 2.12 can be found in [16].

Algorithm 2.6 shows how the Optimized Kronecker product of two CFFs is found. In Algorithm 2.6 the three CFFs' incidence matrices are passed as parameters A, B and C. A is the incidence matrix of a d-CFF(t_1, n_1), B is the incidence matrix of a d-CFF(t_2, n_2), and C is the incidence matrix of a $(d-1)$-CFF(s, n_2).

The following diagram shows how the construction works. First, the Kronecker product $C \otimes A$ is computed. Then, each column of B is repeated n_1 times in succession. Let B_i be the column i in B. Let $n_1 \cdot B_i$ be the matrix formed by the column B_i repeated n_1 times. The construction is depicted next:

$C \otimes A$		
$n_1 \cdot B_1$	...	$n_1 \cdot B_s$

Algorithm 2.6 OptimizedKroneckerProduct$(A, B, C, t_1, t_2, s, n_1, n_2)$

Input: $n_1, n_2, t_1, t_2 \in \mathbb{Z}^+$, A is a $t_1 \times n_1$ $0-1$ matrix for a d-CFF(t_1, n_1), B is a $t_2 \times n_2$ $0-1$ matrix for a d-CFF(t_2, n_2), C is a $s \times n_2$ $0-1$ matrix for a $(d-1)$-CFF(s, n_2).
Output: a d-CFF$(st_1 + t_2, n_1 n_2)$

```
1: CFF ← (st_1 + t_2) × (n_1 n_2) matrix of 0s
2: for r_1 ← 0, 1, ..., s − 1 do                ▷ Compute the Kronecker product of C and A
3:     for c_1 ← 0, 1, ..., n_1 − 1 do
4:         if C[r_1][c_1] = 1 then
5:             for r_2 ← 0, 1, ..., t_1 − 1 do
6:                 for c_2 ← 0, 1, ..., n_1 − 1 do
7:                     if A[r_2][c_2] = 1 then
8:                         CFF[r_1 t_1 + r_2][c_1 n_1 + c_2] ← 1
9:                     end if
10:                end for
11:            end for
12:        end if
13:    end for
14: end for
15: for r ← 0, 1, ..., t_2 − 1 do▷ Repeat each column in B n_1 times at the bottom of CFF
16:    for c ← 0, 1, ..., n_2 − 1 do
17:        if B[r][c] = 1 then
18:            for i ← 0, 1, ..., n_1 − 1  do
19:                CFF[st_2 + r][cn_1 + i] ← 1
20:            end for
21:        end if
22:    end for
23: end for
24: return CFF
```

2.2.3 Doubling construction for $d = 2$

Proposition 2.13 is a recursive construction from [19] that applies only to $d = 2$. This construction is sometimes called the doubling construction because it doubles the number of columns of a given 2-CFF.

Proposition 2.13 (Doubling construction)**.** *If there exists a 2-CFF(t, n), then there exists a 2-CFF$(t + t(1, n) + 2, 2n)$. If $t(1, n)$ is odd, then there exists a 2-CFF$(t + t(1, n) + 1, 2n)$.*

Therefore,

$$t(2,2n) \leq \begin{cases} t(2,n)+t(1,n)+2 \text{ if } n \text{ is even,} \\ t(2,n)+t(1,n)+1 \text{ if } n \text{ is odd.} \end{cases}$$

The idea with this constructions is to create a new CFF as follows. Let A be the original 2-CFF(t,n). Let B be an $s \times n$ matrix where each column is a unique $\lfloor \frac{s}{2} \rfloor$-subset of the set $\{0,\ldots,s-1\}$, $\mathbf{1}$ is a row vector of 1s, and $\mathbf{0}$ is a row vector of 0s of appropriate size. $\overline{C}$ is the binary complement of C, meaning that 1s are replaced by 0s, and 0s are replaced by 1s. Then, construct the new CFF in the following way:

$$\begin{pmatrix} A & A \\ C & \overline{C} \end{pmatrix}.$$

Where

$$C = \begin{pmatrix} B \\ \mathbf{1} \\ \mathbf{0} \end{pmatrix}.$$

In the case when s is odd, the $\mathbf{1}$ row can be omitted from C, reducing the number of rows. The full proof that the resulting matrix is a CFF can be found in [19].

The subsets in the $s \times n$ matrix B can be generated by Algorithm 2.1.

2.2.4 Additive construction

Another recursive construction works by combining two d-CFFs A and B as follows:

$$\begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix}.$$

The number of rows in the resulting CFF is the sum of the number of rows in A and B, and the number of columns in the resulting CFF is the sum of the number of columns in A and B. So, this construction is sometimes called the additive construction.

Proposition 2.14 (Additive construction)**.** *If there exists a d-CFF(t_1,n_1) and there exists a d-CFF(t_2,n_2), then there exists a d-CFF(t_1+t_2,n_1+n_2). Therefore,*

$$t(d,n_1+n_2) \leq t(d,n_1)+t(d,n_2).$$

The proof that this construction gives a d-CFF is available in [19].

2.2.5 Extension by one

A construction similar to the additive construction works as follows. Let A be a d-CFF, then the following matrix is a d-CFF:

$$\begin{pmatrix} A & 0 \\ 0 & 1 \end{pmatrix}.$$

This is not a special case of the additive construction because a 1×1 matrix is not a 1-CFF, since $d = 1 = t = n$, and Definition 1.1 requires that $d < t \leq n$.

Proposition 2.15 (Extension by one)**.** *If there exists a d-CFF(t, n), then there exists a d-CFF$(t + 1, n + 1)$. Therefore,*

$$t(d, n + 1) \quad \leq \quad t(d, n) + 1.$$

Proof. We must show that each sub-matrix that corresponds to a subset of $d + 1$ columns has each weight-1 $(d + 1)$-tuple present in a row. This can be broken down into two cases.

The first case is when the $(d + 1)$ sub-matrix only contains columns from the original CFF A. So, obviously, each weight-1 $(d + 1)$-tuple is present.

The second case is when the sub-matrix contains the newly added column. In this case, the d columns from A contain each weight-1 0-1 d-tuple as a row, because any d-CFF is also a $(d - 1)$-CFF. Each of these rows will gain another zero from the newly added column, and the newly added row of the resulting CFF contains the final weight-1 0-1 $(d + 1)-$tuple that must be present. □

2.3 Running time of selected constructions

In Table 2.1, we provide the running time of each construction implemented. The running time of each construction is linear with the size of the generated CFF, which is $t \times n$.

We implemented many constructions from [17], which are listed in Table 2.1. We did not choose to implement constructions from packing designs, besides STSs, or constructions from perfect hash families, separating hash families, packing arrays, orthogonal arrays, mutually orthogonal Latin squares (MOLs), because these each have many different constructions themselves. Additionally, we did not implement recursive constructions that require the use of these mathematical objects either. Furthermore, we did not implement constructions for Garcia and Stichtenoth codes, which is also listed as a source of CFFs from [17]. We selected our constructions because they are known to give good CFFs, and because these constructions are simpler to implement than the others listed.

To test that our programs are correct implementations of the constructions, we tested that the output matrices are indeed a d-CFF for a given d. The pseudocode for this algorithm that tests whether a matrix is a d-CFF is given in Appendix A.

Construction	**d**	**Parameters**	**Size $t \times n$**	**Running Time Complexity**
Sperner	1	t	$t \times \binom{t}{t/2}$	$O(t2^t)$
STS	2	v	$v \times \frac{v(v-1)}{6}$	$O(v^3)$
Doubling	2	2-CFF(t, n), s	$(t+t(1,n)+2) \times 2n$, if n is even $(t+t(1,n)+1) \times 2n$ if n is odd.	$O(tn)$[1]
Reed-Solomon codes	≥ 2	q, k, m	$mq \times q^k$	$O(mq^{k+1})$
Shortened Reed-Solomon codes	≥ 2	q, k, m, s	$(m-s)q \times q^{k-s}$	$O((m-s)(k-s)q^{k-s})$
Porat and Rothschild codes (Chapter 3)	≥ 2	q, k, r, m	$mq \times q^k$	$\Theta(mq^{k+1})$ [23]
Kronecker Product	≥ 2	d-CFF(t_1, n_1), d-CFF(t_2, n_2)	$t_1t_2 \times n_1n_2$	$O(t_1t_2n_1n_2)$
Optimized Kronecker Product	≥ 2	d-CFF(t_1, n_1), d-CFF(t_2, n_2), $(d-1)$-CFF(s, n_2)	$(st_1+t_2)\times(n_1n_2)$	$O((st_1 + t_2)n_1n_2)$
Additive	≥ 2	d-CFF(t_1, n_1), d-CFF(t_2, n_2)	$(t_1 + t_2) \times (n_1 + n_2)$	$O(\max(t_1n_1, t_2n_2))$
Extension-by-one	≥ 2	d-CFF(t, n)	$(t+1) \times (n+1)$	$O(tn)$

Table 2.1: Running time for algorithms implementing constructions

[1] $t(1, n) \leq t(2, n) \leq t$

Chapter 3

Probabilistic method constructions

The probabilistic method is a powerful non-constructive proof technique used in discrete mathematics to show the existence of a mathematical object. The method of conditional probabilities is a related technique that produces a deterministic algorithm to obtain a mathematical object that has been proven to exist using the probabilistic method.

In this chapter, we discuss the probabilistic method and an application used to find linear codes meeting the Gilbert-Varshamov bound. These codes are used to construct CFFs with best-known asymptotic upper bound for $t(d, n)$ in $O(d^2 \log n)$.

3.1 The probabilistic method

The idea of the probabilistic method is to define a probability space, and show that a mathematical object has a non-zero probability of existing within that probability space, thus proving the object's existence. The probabilistic method is best described with examples of its use. The following Theorem 3.1 and its proof is found in [2]. We present an expanded version of this proof to more clearly show the probabilistic method's use. Let K_n denote the complete graph on n vertices, that is the graph with n vertices where any two vertices are connected by an edge. A 2-colouring of K_n is an assignment of colours from a 2-set of colours to each edge of K_n. A complete subgraph K_a of K_n, $a \leq n$, is *monochromatic* if all the edges in K_a have the same colour.

Theorem 3.1. *Let $n \geq a \geq 2$. Then, there is a 2-colouring of the edges of K_n with at most*

$$\binom{n}{a} 2^{1-\binom{a}{2}}$$

monochromatic K_a.

Proof. We use a probability space of all possible edge colourings of the complete graph K_n, where each edge e has probability $P(e_{blue}) = \frac{1}{2}$ and $P(e_{red}) = \frac{1}{2}$, where e_{red} and e_{blue} are the events that edge e is coloured red or blue.

Let I_a be an indicator variable that is 1 if a complete subgraph K_a of K_n is monochromatic, and is 0 otherwise.

Each K_a has $\binom{a}{2}$ edges. There are two cases when K_a is monochromatic: being when all edges are red, or all edges are blue. Therefore,

$$\begin{aligned} Pr(I_a = 1) &= 2(\frac{1}{2})^{\binom{a}{2}} \\ &= \frac{1}{2^{\binom{a}{2}-1}} \\ &= 2^{1-\binom{a}{2}}. \end{aligned}$$

Next, find the expectation of I_a, $E[I_a]$.

$$\begin{aligned} E[I_a] &= 1 \cdot Pr(I_a = 1) + 0 \cdot Pr(I_a = 0) \\ E[I_a] &= 2^{1-\binom{a}{2}}. \end{aligned}$$

Let X be the number of monochromatic K_a. $E[X]$ can be found using linearity of expectation. There are $\binom{n}{a}$ K_a subgraphs in K_n.

$$\begin{aligned} E[X] &= E\left[\sum_{K_a \in K_n} I_a\right] \\ &= \sum_{K_a \in K_n} E[I_a] \\ &= \sum_{K_a \in K_n} 2^{1-\binom{a}{2}} \\ &= \binom{n}{a} 2^{1-\binom{a}{2}}. \end{aligned}$$

Because $E[X]$ is the average number of monochromatic K_a over all possible edge colourings, there must be at least one edge colouring that is at or below $E[X]$. Therefore there exists a 2-colouring of the edges of K_n with at most $\binom{n}{a}2^{1-\binom{a}{2}}$ monochromatic K_a. □

3.1.1 The method of conditional probabilities

The method of conditional probabilities is used to find a deterministic algorithm to construct a mathematical object that has been proven to exist with the probabilistic method. The main idea is to make local choices to construct a mathematical object, such that the local choices optimize the probability that the object exists. Since the object has been proven to exist, this procedure will always find the object. For example, we will look at this case of the previous Theorem 3.1 with $a = 4$. This example is also from [2].

Theorem 3.2. *There is a 2-colouring of the edges of K_n with at most*

$$\binom{n}{4}2^{-5}$$

monochromatic K_4.

In this example, the local choices are the colours for each edge. So, we start with an uncoloured complete graph K_n. Then, colour each edge red or blue one at a time, depending on which colour will make it more likely that the number of monochromatic K_4 is minimized. Repeat until the graph is fully coloured to obtain a complete K_n with at most $\binom{n}{4}2^{-5}$ monochromatic K_4.

To determine which colour will make it more likely that the number of monochromatic K_4 is minimized, we define a weight function $W(K)$. This function is the probability that some K, where K is a copy of K_4 in K_n, will be monochromatic given K's current colouring, assuming that the rest of the uncoloured edges are chosen randomly with equal probability of being red or blue. If there are two edges with different colours in K, then $W(K) = 0$. Otherwise, if r is the number of coloured edges in K, then $W(K) = \frac{1}{2^{\binom{4}{2}-r}}$.

From here we can give the details the algorithm in Algorithm 3.1. Let Colours(K) be a function that returns a set containing the colours of each edge in a graph K to be used in Algorithm 3.1. We iterate over each edge, find the weight total for the two cases where the edge is red or blue by calculating the conditional probability that each K_4 that contains the edge is monochromatic, and set the colour to the one that has a lower total weight. At the end of this process, we obtain a 2-colouring of K_n with at most $\binom{n}{4}2^{-5}$ monochromatic K_4. This algorithm is detailed here to demonstrate how to use the method of conditional probabilities to "transform" a probabilistic proof of existence of an object, like the one in Theorem 3.2, into an algorithm that produces the object. In Section 3.3, we will describe a more complex algorithm, using the method of conditional probabilities, that will be used to construct CFFs in Section 3.4.

3.2 Binomial distribution

The binomial distribution is used in the following Section 3.3 to construct linear error correcting codes using the probabilistic method.

A *Bernoulli trial* is a probability event where the outcome is either a success or a failure. Each Bernoulli trial is an independent event, where the outcome of each event does not impact later Bernoulli trials.

The *binomial distribution*, $B(k; n, p)$, is a probability distribution that is used to find the probability of exactly k successes in a series of n Bernoulli trials, where each trial has probability p of success, and probability $1 - p$ of failure. This can be calculated as

$$B(k; n, p) \quad = \quad \binom{n}{k} p^k (1-p)^{n-k}.$$

Algorithm 3.1 FindColouring(n)

Input: n: number of vertices in K_n.
Output: A 2-colouring of K_n with at most $\binom{n}{4}2^{-5}$ monochromatic K_4.

```
1: colour ← an empty array of length (n choose 2)
2: for each edge e in K_n do
3:     Weight_red ← 0
4:     Weight_blue ← 0
5:     for each K ⊇ e, where K is a copy of K_4 do
6:         if |Colours(K)| = 1 then
7:             r ← (number of coloured edges in K) +1
8:             if Colours(K)={blue} then
9:                 Weight_blue ← Weight_blue + (1/2)^(6−r)
10:            else
11:                Weight_red ← Weight_red + (1/2)^(6−r)
12:            end if
13:        end if
14:    end for
15:    if Weight_red ≤ Weight_blue then
16:        colour[e] ← red
17:    else
18:        colour[e] ← blue
19:    end if
20: end for
21: return colour
```

If a random variable x is distributed binomially, is it denoted as $x \sim B(k; n, p)$. To show that a random variable is binomially distributed without specifying the number of successes, it is denoted as $x \sim B(n, p)$ [27].

3.3 Porat and Rothschild linear code construction

The result of Porat and Rothschild's paper [23] regarding the probabilistic method shows the existence of linear codes meeting the Gilbert-Varshamov (GV) bound. They also give an efficient deterministic algorithm using the method of conditional probabilities to find these codes. In this section, we discuss the proof for this result. We also provide detailed pseudocode for the algorithm. Deriving this pseudocode from the explanations in [23] was a non-trivial task; we believe that this pseudocode can help others to better understand the algorithm proposed by Porat and Rothschild.

The GV bound gives a lower bound on the rate of codes that can exist. The rate of a code is defined as its message length divided by its codeword length, $R = \frac{k}{m}$. Let $q \geq 2$,

$0 \leq \delta < 1 - \frac{1}{q}$, and $0 < \epsilon \leq H_q(\delta)$, where $H_q(\delta)$ is the q-ary entropy function

$$H_q(p) \quad = \quad p \log_q \frac{q-1}{p} + (1-p) \log_q \frac{1}{1-p}.$$

The Gilbert-Varshamov bound states that there exists a code with rate $R \geq 1 - H_q(\delta)$.

The main theorem to show the existence of linear codes meeting the GV bound that can be constructed in $\Theta(mq^k)$ is given next.

Theorem 3.3 ([23]). *Let q be a prime power, k and m be positive integers, and $\delta \in \left[0, 1 - \frac{1}{q}\right)$. If $k \leq (1 - H_q(\delta))m$, then it is possible to construct an $[m, k, \lfloor \delta m \rfloor]_q$ linear code in time $\Theta(mq^k)$.*

We prove Theorem 3.3 in two parts:

- Theorem 3.8 proves the existence of an $[m, k, \lfloor \delta m \rfloor]_q$ linear code.
- Section 3.3.2.1 shows that one can construct it in time $\Theta(mq^k)$.

This result is obtained using the probabilistic method. The key idea is to define a bad event and a related goal function. By Proposition 2.6, in order to have a minimum distance of at least $\lfloor \delta m \rfloor$, it is enough to ensure the weight of each codeword is at least $\lfloor \delta m \rfloor$. A bad event $\mathcal{B}_\delta(x)$ is defined as the bad event where a codeword x has a weight less than $\lfloor \delta m \rfloor$, the code's minimum distance. $\mathcal{B}_\delta(x)$ is also used as an indicator variable, where it is 0 if there is no bad event, and 1 otherwise.

The goal function is defined as the sum of all bad events for each codeword, excluding the zero codeword. Let $\mathbf{0} = (0, 0, \ldots, 0)$ be the zero codeword. Then,

$$goal(\mathcal{G}) \quad = \quad \sum_{y \in \mathbf{F}_q^k \setminus \mathbf{0}} \mathcal{B}_\delta(\mathcal{G}y),$$

where $\mathcal{G}$ is the generator matrix, and y is a message vector. Thus $\mathcal{G}y$ indicates vector-matrix multiplication between one message vector and the generator matrix, resulting in a codeword. If no bad events occur, then $goal(\mathcal{G}) = 0$ and the code has the desired minimum distance of $\lfloor \delta m \rfloor$.

To prove Theorem 3.3 (part 1), i.e. Theorem 3.8, it will be shown that if $\mathcal{G}$ is selected randomly and uniformly, then $E[goal(\mathcal{G})] < 1$, implying that at least one $\mathcal{G}$ must exist with $goal(\mathcal{G}) = 0$. To do this, the Chernoff bound is used.

Theorem 3.4 (Chernoff bound). *Assume random variables $X_1, \ldots, X_m$ are independent and identically distributed (i.i.d), $X_i \in \mathbb{R}$, and $0 \leq X_i \leq 1$. Let $\mu = E[X_i]$ and $\epsilon > 0$.*

Then,

$$\begin{aligned} Pr(\frac{1}{m}\sum X_i \geq \mu+\epsilon) &\leq \left(\left(\frac{\mu}{\mu+\epsilon}\right)^{\mu+\epsilon}\left(\frac{1-\mu}{1-\mu-\epsilon}\right)^{1-\mu-\epsilon}\right)^m \\ &= e^{-D(\mu+\epsilon||\mu)m}. \end{aligned}$$

Where $D(x||y) = xln\frac{x}{y} + (1-x)ln\frac{1-x}{1-y}$.

The proof for Theorem 3.8 uses Lemmas 3.5, 3.6 and 3.7, then follows closely after. Lemma 3.5 is a claim stated in [23] for which we added a detailed proof.

Lemma 3.5. *Let y be a nonzero vector in $\mathbb{F}_q^k$. Let $\mathcal{G}$ be a randomly chosen generator matrix. Then the codeword $x = \mathcal{G}y$ is a random vector in $\mathbb{F}_q^m$, with each component having probability $\frac{1}{q}$ being equal to c, for each $c \in \mathbb{F}_q$.*

Proof. Let $j_1, j_2, \ldots j_\ell$ be the indices of y that are nonzero components. Then,

$$[\mathcal{G}y]_i = \sum_{j=1}^{k} \mathcal{G}[i,j]\cdot y_j = \sum_{e=1}^{\ell} \mathcal{G}[i,j_e]\cdot y_{j_e}$$

for all $1 \leq i \leq m$. We show that $[\mathcal{G}y]_i$ is a random number in $\mathbb{F}_q$, by induction on ℓ. The base case is when $\ell = 1$.

$$[\mathcal{G}y]_i = \mathcal{G}[i,j_1]\cdot y_{j_1}, \quad \text{where } y_{j_1} = a \text{ is a fixed element of } \mathbb{F}_q \setminus \{0\}.$$

Since $\mathcal{G}[i,j_1]$ is random and, $f\colon \mathbb{F}_q \to \mathbb{F}_q$, given as $f(x) = ax$ is a bijection, then $[\mathcal{G}y]_i$ is a random element of $\mathbb{F}_q$.

The inductive step is when $\ell > 1$. By induction, we know that $Z_i = \sum_{e=1}^{\ell-1} \mathcal{G}[i,j_e]\cdot y_{j_e}$ is a random number. We want to show that $[\mathcal{G}y]_i = \sum_{e=1}^{\ell} \mathcal{G}[i,j_e]\cdot y_{j_e}$ is a random number. We have

$$[\mathcal{G}y]_i = Z_i + G[i,j_\ell]\cdot y_\ell,$$

where Z_i and $G[i,j_\ell]$ are random numbers, and $y_\ell \in \mathbb{F}_q \setminus \{0\}$. From the base case we know $Z_i' = \mathcal{G}[i,j_\ell]y_\ell$ is a random number of $\mathbb{F}_q$. Thus,

$$[\mathcal{G}y]_i = Z_i + Z_i', \quad \text{where } Z_i \text{ and } Z_i' \text{ are random numbers in } \mathbb{F}_q.$$

Let $b \in \mathbb{F}_q$, we claim there are q distinct pairs $(Z_i, Z_i') \in \mathbb{F}_q$ such that $Z_i + Z_i' = b$ (for each $Z_i \in \mathbb{F}_q$, Z_i' is determined). Therefore for each $b \in \mathbb{F}_q$, $Pr([\mathcal{G}y]_i = b) = \frac{q}{q^2} = \frac{1}{q}$. □

In Lemma 3.6, we add the hypothesis that $\delta \in \left[0, 1-\frac{1}{q}\right)$, which was not included in [23]. This condition is necessary to apply the Chernoff bound, and we do not see how this is implied by other conditions. In the worst case, we are adding a redundant condition. This required adding this condition to both Lemma 3.6 and Theorem 3.3.

Lemma 3.6 ([23] Lemma 5.1)**.** *Let $\delta \in \left[0, 1 - \frac{1}{q}\right)$. Let y be a nonzero vector in $\mathbb{F}_q^k$. Let $\mathcal{G}$ be a randomly chosen generator matrix. Then, $Pr(\mathcal{B}_\delta(\mathcal{G}y)) \leq q^{-m(1-H_q(\delta))}$.*

Proof. By Lemma 3.5, each component of codeword $\mathcal{G}y$ has probability $\frac{1}{q}$ of being zero. Thus, $w(\mathcal{G}y) \sim B(m, 1 - \frac{1}{q})$, where B is the binomial distribution.

Let X_i be a random variable such that $X_i = 0$ if the letter $[\mathcal{G}y]_i = 0$, and $X_i = 1$ otherwise, for all $i \in [1, m]$. This allows us to represent the weight of a codeword $\mathcal{G}y$ as $w(\mathcal{G}y) = m - \sum_{i=1}^{m} X_i$. All X_i are i.i.d, because the letters of $\mathcal{G}$ are chosen randomly, and the outcome of one event does not impact the outcome of other event. In addition,

$$\begin{aligned} E[X_i] &= 1 \cdot Pr(X_i = 1) + 0 \cdot Pr(X_i = 0) \\ &= 1 \cdot \frac{1}{q} + 0 \cdot \left(1 - \frac{1}{q}\right) \\ &= \frac{1}{q}. \end{aligned}$$

Now use Theorem 3.4, the Chernoff bound, with $\mu = \frac{1}{q}$, and $\epsilon = \left(1 - \frac{1}{q}\right) - \delta > 0$, to bound the probability of a bad event. Thus $\mu + \epsilon = 1 - \delta$, and we have:

$$\begin{aligned} Pr(\mathcal{B}_\delta(\mathcal{G}y)) &= Pr(w(\mathcal{G}y) \leq \delta m) \\ &= Pr\left(m - \sum_{i=1}^{m} X_i \leq \delta m\right) \\ &= Pr\left(-\sum_{i=1}^{m} X_i \leq \delta m - m\right) \\ &= Pr\left(-\sum_{i=1}^{m} X_i \leq (\delta - 1)m\right) \\ &= Pr\left(\frac{1}{m}\sum_{i=1}^{m} X_i \geq (1 - \delta)\right) \\ &\leq e^{-D\left(1-\delta||\frac{1}{q}\right)m} \quad \text{(Using Chernoff bound[1])}. \end{aligned}$$

Next, we show that $D(1 - \delta||\frac{1}{q}) = \ln q(1 - H_q(\delta))$, using the definition of $H_q(\delta)$ and of $D(x||y)$.

[1]Recall that $\mu = \frac{1}{q}, \epsilon = \left(1 - \frac{1}{q}\right) - \delta > 0$, and $\mu + \epsilon = 1 - \delta$

$$\begin{aligned}
D(1-\delta||\frac{1}{q}) &= (1-\delta)\ln((1-\delta)q) + \delta\ln\frac{\delta}{1-\frac{1}{q}} \\
&= \ln q((1-\delta)\log_q((1-\delta)q) + \delta(\log_q\frac{\delta q}{q-1})) \\
&= \ln q((1-\delta)(\log_q(1-\delta)+1) + \delta(\log_q\frac{\delta}{q-1}+1)) \\
&= \ln q(1+(1-\delta)\log_q(1-\delta) + \delta\log_q(\frac{\delta}{q-1})) \\
&= \ln q(1-(1-\delta)\log_q(\frac{1}{1-\delta}) - \delta\log_q(\frac{q-1}{\delta})) \\
&= \ln q(1-\delta\log_q(\frac{q-1}{\delta}) - (1-\delta)\log_q(\frac{1}{1-\delta})) \\
&= \ln q(1-H_q(\delta)) \quad \text{(Entropy function definition).}
\end{aligned}$$

Finally use $D(1-\delta||\frac{1}{q}) = \ln q(1-H_q(\delta))$ to bound the probability of a bad event:

$$\begin{aligned}
Pr(\mathcal{B}_\delta(x)) &\leq e^{-D(1-\delta||\frac{1}{q})m} \\
&\leq e^{-\ln q(1-H_q(\delta))m} \\
&\leq q^{-m(1-H_q(\delta))}.
\end{aligned}$$

□

Lemma 3.7 ([23] Lemma 5.2)**.** *Let $\delta \in \left[0, 1-\frac{1}{q}\right)$. Let $\mathcal{G}$ be a randomly chosen generator matrix, and assume $k \leq m(1-H_q(\delta))$. Then, $E[goal(\mathcal{G})] < 1$.*

Proof. First, apply linearity of expectation to find the expectation of the goal function in terms of the probability of $\mathcal{B}_\delta(x)$:

$$\begin{aligned}
goal(\mathcal{G}) &= \sum_{y\in\mathbf{F}_q^k\setminus\{\mathbf{0}\}} \mathcal{B}_\delta(\mathcal{G}y) \quad \text{(by definition)} \\
E\left[goal(\mathcal{G})\right] &= E\left[\sum_{y\in\mathbf{F}_q^k\setminus\{\mathbf{0}\}} \mathcal{B}_\delta(\mathcal{G}y)\right] \\
&= \sum_{y\in\mathbf{F}_q^k\setminus\{\mathbf{0}\}} E[\mathcal{B}_\delta(\mathcal{G}y)] \quad \text{(linearity of expectation)} \\
&= \sum_{y\in\mathbf{F}_q^k\setminus\{\mathbf{0}\}} (1\cdot Pr(\mathcal{B}_\delta(\mathcal{G}y)=1)) + (0\cdot Pr(\mathcal{B}_\delta(\mathcal{G}y)=0)) \quad (\mathcal{B}_\delta(\mathcal{G}y)\in\{0,1\}) \\
&= \sum_{y\in\mathbf{F}_q^k\setminus\{\mathbf{0}\}} Pr(\mathcal{B}_\delta(\mathcal{G}y)) \quad (\mathcal{B}_\delta(\mathcal{G}y) \text{ is both an event and indicator variable}).
\end{aligned}$$

Finally, we show $E\left[goal(\mathcal{G})\right]$ using Lemma 3.6, and the assumption that $k \leq m(1-H_q(\delta))$. Lemma 3.6 implies that $Pr(\mathcal{B}_\delta(\mathcal{G}y)) \leq q^{-m(1-H_q(\delta))}$ for all $y \in \mathbb{F}_q^k \setminus \{\mathbf{0}\}$.

$$\begin{aligned} E[goal(\mathcal{G})] = \sum_{y \in \mathbf{F}_q^k \setminus \{\mathbf{0}\}} Pr(\mathcal{B}_\delta(\mathcal{G}y)) &\leq (q^k - 1)q^{-m(1-H_q(\delta))} \\ &\leq q^{k-m(1-H_q(\delta))} - q^{-m(1-H_q(\delta))} \\ &< q^{k-m(1-H_q(\delta))} \\ &\leq q^{k-k} \quad (\text{Since } k \leq m(1-H_q(\delta))) \\ &= 1. \end{aligned}$$

Therefore, $E[goal(\mathcal{G})] < 1$. □

From here, we can prove part 1 of Theorem 3.3, related to the existence of codes meeting the Gilbert-Varshamov bound.

Theorem 3.8. *Let q be a prime power, k and m be positive integers, and $\delta \in \left[0, 1 - \frac{1}{q}\right)$. If $k \leq (1 - H_q(\delta))m$, then there exists a $[m, k, \lfloor \delta m \rfloor]_q$ linear code.*

Proof. Because $E[goal(\mathcal{G})]$ is the average number of the sum of bad events over all possible generator matrices, there must be at least one generator matrix that is at or below $E[goal(\mathcal{G})]$. Lemma 3.7 shows $E[goal(\mathcal{G})] < 1$. Since the number of bad events must be an integer at the end of the algorithm, there must exist at least one generator matrix with 0 bad events at the end of the algorithm if this expectation is less than 1. Therefore there exists a generator matrix with no bad events, showing the existence of these codes. □

From this, we have Algorithm 3.2.

Algorithm 3.2 ConstructGeneratorMatrixRandomly(m, k, q, δ)

Input: $m, k, q \in \mathbb{Z}^+$, q is a prime power, $\delta \in \left[0, 1 - \frac{1}{q}\right)$, $k \leq (1 - H_q(\delta))m$.

Output: A generator matrix for a $[m, k, \lfloor \delta m \rfloor]_q$ code, with a non-zero probability of the code being valid.

1: Set the letters of $\mathcal{G}$ randomly, uniformly from $\mathbb{F}_q$, and independently of one another.
2: **return** $\mathcal{G}$

3.3.1 Derandomization

Theorem 3.8 shows the existence of linear codes meeting the Gilbert-Varshamov bound, but it does not show how to construct the codes. From Theorem 3.8, all we know is that if each entry is randomly chosen in a generator matrix for a code that satisfies $k \leq (1 - H_q(\delta))m$, there is a non-zero probability that the code is valid.

Algorithm 3.3 ConstructGeneratorMatrix(m, k, q, δ)

Input: $k, q, m \in \mathbb{Z}$, $\delta \in \left[0, 1 - \frac{1}{q}\right)$, q is a prime power, $k \leq (1 - H_q(\delta))$.

Output: Generator matrix for a $[m, k, \lfloor \delta m \rfloor]_q$ code.

1: **for** i in $[1, m]$ **do**
2: **for** j in $[1, k]$ **do**
3: Set $\mathcal{G}[i, j]$ to the letter that minimizes goal$(\mathcal{G})$ assuming the entries of $\mathcal{G}$ that are not chosen distribute uniformly on $\mathbb{F}_q$ and independently of one another.
4: **end for**
5: **end for**
6: **return** $\mathcal{G}$

Porat and Rothschild derandomized Algorithm 3.2 using the method of conditional probabilities to give a deterministic algorithm (Algorithm 3.3) that constructs these codes in $\Theta(mq^k)$ time. We will explain why this deterministic algorithm will always find a valid code.

First, it is necessary to define notation to indicate which position in $\mathcal{G}$ the algorithm is about to fix.

Definition 3.9. *Define step(i, j) as the step where Algorithm 3.3 is about to fix position (i, j) in the generator matrix (right before line 3 of Algorithm 3.3). Let step$(i, j) + 1$ indicate the next step after step(i, j). Thus,*

$$step(i,j)+1 \quad = \quad \begin{cases} step(i, j+1) & \text{if } j < m, \\ step(i+1, j+1) & \text{if } j = m. \end{cases}$$

Additionally, it is necessary to define notation to indicate the state of the matrix in some step(i, j).

Definition 3.10. *Let $ST_{(i,j)}$ be the state of the matrix $\mathcal{G}$ when the algorithm is in step(i, j).*

Lemma 3.11 ([23] Lemma 5.3)**.** *If the letters of $\mathcal{G}$ are fixed one at a time such that $E[goal(\mathcal{G})]$ is minimized at each step, then $goal(\mathcal{G}) = 0$*

Proof. Assume the algorithm is in some step, $step(i, j)$. Then,

$$Pr(\mathcal{B}_\delta(\mathcal{G}y)|ST_{(i,j)}) \quad = \quad \frac{1}{q} \sum_{v \in \mathbb{F}_q} Pr(\mathcal{B}_\delta(\mathcal{G}y)|ST_{(i,j)}, G[i,j] = v).$$

As a result,

$$\begin{aligned}
E[goal(\mathcal{G})|ST_{(i,j)}] &= E\left[\sum_{y\in\mathbf{F}_q^k\setminus\{\mathbf{0}\}}\left(\mathcal{B}_\delta(\mathcal{G}y)|ST_{(i,j)}\right)\right] \\
&= \sum_{y\in\mathbf{F}_q^k\setminus\{\mathbf{0}\}} E\left[\mathcal{B}_\delta(\mathcal{G}y)|ST_{(i,j)}\right] \quad \text{(Linearity of expectation)} \\
&= \sum_{y\in\mathbf{F}_q^k\setminus\{\mathbf{0}\}} Pr(\mathcal{B}_\delta(\mathcal{G}y)|ST_{(i,j)}) \quad \text{(Expectation using indicator variable)} \\
&= \frac{1}{q}\sum_{v\in\mathbb{F}_q}\sum_{y\in\mathbf{F}_q^k\setminus\{\mathbf{0}\}} Pr(\mathcal{B}_\delta(\mathcal{G}y)|ST_{(i,j)}, G[i,j]=v) \\
&\geq \min_{v\in\mathbb{F}_q}\sum_{y\in\mathbf{F}_q^k\setminus\{\mathbf{0}\}} Pr(\mathcal{B}_\delta(\mathcal{G}y)|ST_{(i,j)}, G[i,j]=v) \quad \text{(Average} \geq \text{minimum)} \\
&= \sum_{y\in\mathbf{F}_q^k\setminus\{\mathbf{0}\}} Pr(\mathcal{B}_\delta(\mathcal{G}y)|ST_{(i,j)+1}) \quad \text{(Choice at } ST_{(i,j)+1} \text{ is the minimum)} \\
&= E\left[\sum_{y\in\mathbf{F}_q^k\setminus\{\mathbf{0}\}}\left(\mathcal{B}_\delta(\mathcal{G}y)|ST_{(i,j)+1}\right)\right] \quad (\mathcal{B}_\delta(\mathcal{G}y)\in\{0,1\}) \\
&= E\left[goal(\mathcal{G})|ST_{(i,j)+1}\right] \quad \text{(By definition of goal).}
\end{aligned}$$

The algorithm runs steps in the order step(1, 1), step(1,2), ..., step(1,k), ..., step(m,1), ..., step(m, k). Thus, because Lemma 3.7 states $E[goal(\mathcal{G})] < 1$ at the beginning of the algorithm, and we have shown that $E[goal(\mathcal{G})|ST_{(i,j)}] \geq E[goal(\mathcal{G})|ST_{(i,j)+1}]$, then, the number of bad events at the end of Algorithm 3.3 must be equal to zero, since $1 > E[goal(\mathcal{G})] \geq E[goal(\mathcal{G})|ST_{(1,1)}] \geq E[goal(\mathcal{G})|ST_{(1,2)}] \geq \ldots \geq E[goal(\mathcal{G})|ST_{(1,k)}] \geq \ldots \geq E[goal(\mathcal{G})|ST_{(m,1)}] \geq \ldots \geq E[goal(\mathcal{G})|ST_{(m,k)}] \geq \ldots \geq E[goal(\mathcal{G})|ST_{(m+1,1)}]$, and $E[goal(\mathcal{G})|ST_{(m+1,1)}]$ is an integer smaller than 1. □

The final piece before we can write the algorithm is to determine how to calculate the probability of a bad event at some *step*(i, j).

Lemma 3.12 ([23] Lemma 5.4)**.** *Let c be the number of positions of a codeword $\mathcal{G}y$ that have been fixed to nonzero. Let Algorithm 3.3 currently be in step$(i+1, j)$. Then, $w(\mathcal{G}y) - c \sim B(m-i, 1-\frac{1}{q})$ where B is the binomial distribution.*

Proof. As shown in Lemma 3.5, Gy is random with $w(\mathcal{G}y) \sim B(m, 1-\frac{1}{q})$. It follows that if i elements have been fixed, and c of those elements have been fixed to nonzero, then there are $m-i$ letters left whose weight will still be distributed binomially, since the events are independent. □

Thus, if we choose the letters of the generator matrix one at a time such that $w(\mathcal{G}y) - c \sim B(m-i, 1-\frac{1}{q})$ is minimized, no bad events will occur. This is how the algorithm functions.

3.3.2 Algorithms achieving desired complexity

Porat and Rothschild [23] argue in Section B why Algorithm 3.3 can be implemented to run in time polynomial on q^k and m. Then in Section C they show how the complexity can be improved to run in time $\Theta(mq^k)$. Recall that mq^k is the number of letters in the code, since it has q^k codewords of length m; so this complexity is best possible. However, the explanation of this time complexity in [23] is very technical, and no pseudocode is explicitly given. In this section, we give pseudocode containing detailed description of the algorithm. The algorithm first iterates over each position in the generator matrix. Then, each position is set to the letter that minimizes the probability of a bad event, assuming all remaining entries will be chosen randomly and uniformly. Finally, return the generator matrix.

One key idea for attaining the complexity of the algorithms is to order the message vectors in lexicographical order with their components in reverse order. Figure 3.1 shows this order when listing all message vectors $y \in \mathbb{F}_q^k$ for $q = 3$ and $k = 3$. Using this order when computing the codeword in $\mathcal{G}y$ in the algorithm makes the computation of the expectation of a bad event much more efficient. Indeed, we first claim that to select the value of $\mathcal{G}[i,j]$ in step 3 of Algorithm 3.3, we can use the following equation:

$$\min_{v\in\mathbb{F}_q} \sum_{y=0} Pr(\mathcal{B}_\delta(\mathcal{G}y)|ST_{(i,j)}, \mathcal{G}[i,j]=v) \quad = \quad \min_{v\in\mathbb{F}_q} \sum_{\ell=q^{j-1}}^{q^j-1} Pr(\mathcal{B}_\delta(\mathcal{G}y_\ell)|ST_{(i,j)}, \mathcal{G}[i,j]=v) \quad (3.1)$$

The right hand side of Equation 3.1 requires only looking at message vectors y_ℓ for $q^{j-1} \le \ell \le q^j$ instead of q^k vectors.

Let us explain why Equation 3.1 is true by using an example. Figure 3.2 shows the position that is being chosen and the positions with a question mark in green are positions still being determined.

Figure 3.2 also shows the matrix where the codewords are represented as columns $\mathcal{G}y_\ell$, based on y_ℓ given in the lexicographic order described in Figure 3.1. Here the positions marked in yellow have been fixed, the positions marked in red depend on the choice of position $\mathcal{G}[2,2]$ in the generator matrix and the positions marked in green do not depend on $\mathcal{G}[2,2]$. Since $j = 2$, Equation 3.1 says that we only need to compute the probability of a bad event for codewords $\mathcal{G}y_\ell$ for $q^{j-1} = 3 \le \ell \le q^j = 9$. This means calculating the values of positions marked with [*] in the code, based on each choice of value $v \in \mathbb{F}_q$ that $\mathcal{G}[2,2]$ can be set to.

We give two different variations of the algorithms in this section. Algorithm 3.4 uses a brute-force approach to find the letters of the generator matrix that minimize the probability of a bad event, by trying to set each position to every element of $\mathbb{F}_q$. Algorithm 3.5 avoids this loop by directly calculating the letter of the generator matrix that minimizes the probability of a bad event.

Next, we argue that fixing $\mathcal{G}[2,2]$ to some value fixes some entries in some codewords, and leaves others uniformly distributed. In Figure 3.2, the yellow entries [ ] show fixed

0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
0	0	0	1	1	1	2	2	2	0	0	0	1	1	1	2	2	2	0	0	0	1	1	1	2	2	2
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2

Figure 3.1: Lexicographic 3-tuples $y \in \mathbb{F}_3$ with their components in reverse order

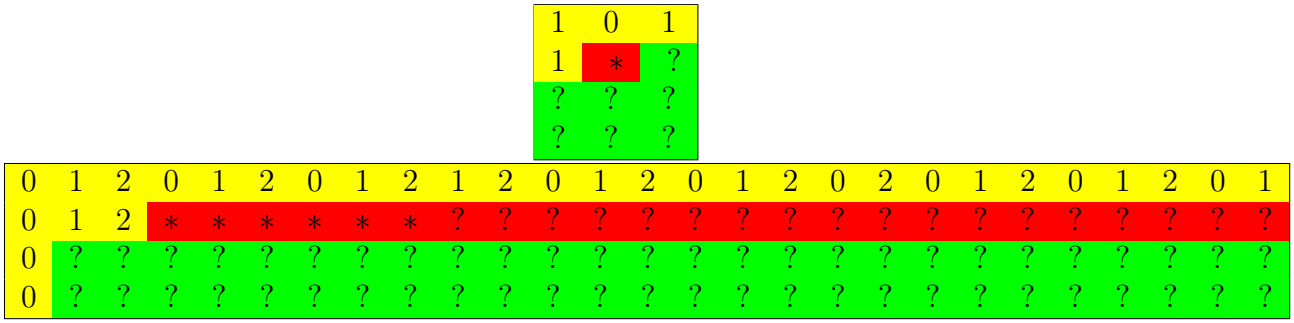


Figure 3.2: A partially complete generator matrix and $[4, 3, 2]_3$ code

letters in $\mathcal{G}$ and in some codewords. All other letters in the codewords still distribute uniformly, thought not independently, on $\mathbb{F}_q$. This is due to Lemma 3.12. Here we explain why using the example in Figures 3.1 and 3.2.

It is easy to see that the choice of $[*]$ in $\mathcal{G}$ fixes letters $[\mathcal{G}y_3]_2, [\mathcal{G}y_4]_2, \ldots [\mathcal{G}y_8]_2$ marked $[*]$ in the codewords in Figure 3.2. Thinking a bit more, it is easy to see that the $[*]$ in $\mathcal{G}$ do not have an effect on entries marked $[?]$, as it is not part of their computations during matrix multiplication.

Let us think of the effect of fixing $[*]$ in $\mathcal{G}$ in future entries calculation for $[?]$, and convince ourselves that the probability of $[?]$ to be either $0, 1$, or 2 is still uniform, even after we fixed $[*]$ in $\mathcal{G}$. Let us pick y_{12} as an example and check $[\mathcal{G}y_{12}]_2$.

$$[\mathcal{G}y_{12}]_2 \quad = \quad [1 \; * \; ?] \cdot \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \quad = \quad [?]$$

Let us assume that before $\mathcal{G}[2, 2]$ was fixed, the distribution of $[\mathcal{G}y_{12}]_2 = [?]$ over $\mathbb{F}_3$ was uniform. Now we verify that its distribution is still uniform. Let us look at the calculation

$$[\mathcal{G}y_{12}]_2 = \; 0 \times [1] + 1 \times [*] + 1 \times [?] \quad = 1 \times [*] + 1 \times [?]$$

before and after $[*]$ is fixed.

Before $[*]$ is fixed, $([*], [?])$ has uniform probability of being one of $(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)$.

After $[*]$ is fixed to any of the values (in this example $[*]$ will be chosen as 1), $[?]$ has uniform probability distributed over $\{0,1,2\}$.

Therefore $\mathcal{G}y_{12}$ does not need to be considered in the probability of causing a bad event, over different choices of $\mathcal{G}[2,2] = [*]$, because these choices do not affect $\mathcal{G}y_{12}$. Therefore Equation 3.1, holds in general:

$$\min_{v\in\mathbb{F}_q}\sum_{y=0} Pr(\mathcal{B}_\delta(\mathcal{G}y)|ST_{(i,j)},\mathcal{G}[i,j]=v) \;=\; \min_{v\in\mathbb{F}_q}\sum_{\ell=q^{j-1}}^{q^j-1} Pr(\mathcal{B}_\delta(\mathcal{G}y_\ell)|ST_{(i,j)},\mathcal{G}[i,j]=v).$$

The right-hand side of the above equation is used to calculate v_{min} in lines 9-26 of Algorithm 3.4 and assign it to $\mathcal{G}[i,j]$ in line 27. This is much more efficient than trying to compute the values using the left-hand side.

We maintain an array A to keep track of how many letters have been fixed to zero in each codeword. The number of positions fixed to zero in a codeword increases the probability of a bad event. This array A allows us to calculate the probability of bad events without having to repeatedly iterate over the elements of a codewords to determine how many positions of a codeword have been fixed to zero.

We maintain an array R of size $m \times q^k$ to store the code. This allows us to reuse matrix multiplication results to maintain constant time to calculate a letter of the code. For example, we will find a letter of the code that corresponds to the message $y_1[1,0,0]$ before we find a letter of the code that corresponds to the message $y_4[1,1,0]$. This difference of one letter being non-zero allows us to re-use the result of the previous calculation. If a message corresponds to index ℓ, $q^{j-1} \leq \ell < q^j$, then $y_\ell = (a_1,\ldots,q_{j-1},q_j \neq 0,0,\ldots,0)$. Then the message at index $(\ell \mod q^{j-1})$ is $y_{\ell \mod q^{j-1}} = (a_1,\ldots,a_{j-1},0,\ldots,0)$. Therefore, $R[\ell,i] = [\mathcal{G}y_\ell]_i = \sum_{s=1}^{j}(\mathcal{G}[i,s]a_s) = \sum_{s=1}^{j-1}(\mathcal{G}[i,s]a_s) + \mathcal{G}[i,j]a_j = \left[\mathcal{G}y_{[\ell \mod q^{j-1}]}\right]_i + \mathcal{G}[i,j]a_j = R[\ell \mod q^{j-1},i] + \mathcal{G}[i,j]a_j$. This justifies the calculation in lines 14 and 29 of Algorithm 3.4 and lines 10 and 23 of Algorithm 3.5.

Putting all these ideas together, we can present the first implementation of Algorithm 3.3, which is given in Algorithm 3.4.

Algorithm 3.4 ConstructGeneratorMatrixBruteForce(m, k, q, δ)

Input: $m, k, q \in \mathbb{Z}, q \in \left[2\left(\frac{1}{1-\delta}\right), 4\left(\frac{1}{1-\delta}\right)\right), q$ is a prime power, $\delta \in \left[0, 1 - \frac{1}{q}\right)$, such that $k \leq (1 - H_q(\delta))m$.

Output: Generator matrix for a $[m, k, \lfloor \delta m \rfloor]_q$ linear code.

$D \leftarrow \lfloor \delta m \rfloor$
$\mathcal{G} \leftarrow$ an empty $m \times k$ matrix, 1-indexed
$A \leftarrow q^k$ array of 0s
$R \leftarrow q^k \times m$ 2d array of 0s
$y \leftarrow$ all lexicographic k-tuples of $\mathbb{F}_q$, with the tuples' components in reversed order
for i in $[1, m]$ **do**
    for j in $[1, k]$ **do**
        $E_{best} \leftarrow \infty$
        for v in $\mathbb{F}_q$ **do**
            $\mathcal{G}[i, j] \leftarrow v$
            $E \leftarrow 0$
            for ℓ in $[q^{j-1}, q^j - 1]$ **do**
                $c \leftarrow i - 1 - A[\ell]$
                $R[\ell][i] \leftarrow R[\ell \mod q^{j-1}][i] + \mathcal{G}[i, j] \times y[j, \ell]$
                if $R[\ell][i] \neq 0$ **then**
                    $c \leftarrow c + 1$
                end if
                for x in $[0, D - c)$ **do**
                    $E \leftarrow E + \binom{m-i}{x}(1 - \frac{1}{q})^x(\frac{1}{q})^{m-i-x}$
                end for
            end for
            if $E < E_{best}$ **then**
                $E_{best} \leftarrow E$
                $v_{best} \leftarrow v$
            end if
        end for
        $\mathcal{G}[i, j] \leftarrow v_{best}$
        for ℓ in $[q^{j-1}, q^j - 1]$ **do**
            $R[\ell][i] \leftarrow R[\ell \mod q^{j-1}][i] + \mathcal{G}[i, j] \times y[j, \ell]$
            if $R[\ell][i] = 0$ **then**
                $A[\ell] \leftarrow A[\ell] + 1$
            end if
        end for
    end for
end for
return $\mathcal{G}$

Algorithm 3.4 is a brute force approach where the expectation of a bad event is explicitly calculated for each letter. The algorithm takes $O(m^2 q^{k+1})$ time, since it must find

the probability of a bad event for each letter of $\mathbb{F}_q$ for each position of each codeword. In Section 3.3.2.1, we do a more detailed analysis.

Porat and Rothschild improved the running time of Algorithm 3.4 by noting that there is a more efficient way to determine the letter that minimizes the probability of a bad event than calculating each expectation. We do not actually need to know what the expectation of a bad event is, we only need to know which letter minimizes this expectation.

Since a bad event is more probable when a letter in the generator matrix fixes a codeword letter to zero, we need to find a letter v that will make codeword $\mathcal{G}y_{\ell_i=0}$ in order to estimate the increase in the probability of a bad event coming from the choice of assigning letter v to $\mathcal{G}[i,j]$.

$$\begin{aligned} Pr(\mathcal{B}_\delta(\mathcal{G}y)|ST_{(i,j),\mathcal{G}y[i]=0}) - Pr(\mathcal{B}_\delta(\mathcal{G}y)|ST_{(i,j),\mathcal{G}y[i]\neq 0}) &= \\ \binom{m-i}{\lfloor \delta m \rfloor - c} \left(1-\frac{1}{q}\right)^{\lfloor \delta m \rfloor - c} \left(\frac{1}{q}\right)^{(m-i)-(\lfloor \delta m \rfloor - c)} &. \end{aligned}$$

Thus, for each relevant y we will add this value to an array $W[v]$ for each $v \in \mathbb{F}_q$, where v is the letter of the generator matrix that would force a position in the code to zero. We will then select the letter v to assign to $\mathcal{G}[i,j]$ based on which letter v has the lowest value in the array W.

In order to determine which letter in the generator matrix would cause a zero in some position in a codeword, the following equation is solved for v:

$$v = -y[j,\ell]^{-1} \sum_{t=0}^{j-1} \mathcal{G}[i,t] y_\ell[t].$$

This is solving for the letter of the generator matrix that would cause a zero in position j in codeword ℓ.

Taking this optimization under consideration we arrive at Algorithm 3.5. Lines 8-26 of Algorithm 3.4 are substituted by lines 8-21 of Algorithm 3.5. The detailed analysis of both algorithms is given in Section 3.3.2.1.

Algorithm 3.5 ConstructGeneratorMatrixOptimized(m, k, q, δ)

Input: $m, k, q \in \mathbb{Z}, q \in \left[2\left(\frac{1}{1-\delta}\right), 4\left(\frac{1}{1-\delta}\right)\right), q$ is a prime power, $\delta \in \left[0, 1 - \frac{1}{q}\right)$, such that $k \leq (1 - H_q(\delta))m$.

Output: Generator matrix for a $[m, k, \lfloor \delta m \rfloor]_q$ linear code.

1: $D \leftarrow \lfloor \delta m \rfloor$
2: $\mathcal{G} \leftarrow$ an empty $m \times k$ matrix, 1-indexed
3: $A \leftarrow q^k$ array of 0s
4: $R \leftarrow q^k \times m$ 2d array of 0s
5: $y \leftarrow$ all lexicographic k-tuples of $\mathbb{F}_q$, with their indices in reversed order
6: **for** i in $[1, m]$ **do**
7: **for** j in $[1, k]$ **do**
8: $W \leftarrow |\mathbb{F}_q|$ array of 0s
9: **for** ℓ in $[q^{j-1}, q^j - 1]$ **do**
10: $v \leftarrow R[\ell \mod q^{j-1}][i] \times -y[j, \ell]^{-1}$
11: $c \leftarrow i - A[\ell]$
12: $W[v] \leftarrow W[v] - \binom{m-i}{D-c}(1 - \frac{1}{q})^{D-c}(\frac{1}{q})^{(m-i)-(D-c)}$
13: **end for**
14: $W_{best} \leftarrow \infty$
15: **for** v in $\mathbb{F}_q$ **do**
16: **if** $W[v] < W_{best}$ **then**
17: $W_{best} \leftarrow W[v]$
18: $v_{best} \leftarrow v$
19: **end if**
20: **end for**
21: $\mathcal{G}[i, j] \leftarrow v_{best}$
22: **for** ℓ in $[q^{j-1}, q^j - 1]$ **do**
23: $R[\ell][i] \leftarrow R[\ell \mod q^{j-1}][i] + \mathcal{G}[i, j] \times y[j, \ell]$
24: **if** $R[\ell][i] = 0$ **then**
25: $A[\ell] \leftarrow A[l] + 1$
26: **end if**
27: **end for**
28: **end for**
29: **end for**
30: **return** $\mathcal{G}$

3.3.2.1 Running time analysis

In this section, we discuss the running time of Algorithms 3.4 and 3.5. We first give an analysis of the brute force variation, Algorithm 3.4.

The running time of Algorithm 3.4 is $O(m^2 q^{k+1})$. We have a factor of m from the loop on line 6, and another factor $\leq m$ from the for loop on line 18 since $D - c \leq \delta m - 0 \leq m$. Line 9 gives a factor of q, and we will show that the combination of the for loops on lines 7 and 12 give a factor of q^k. In line 7, we iterate over each j from $1, \ldots, k$. Then, in line 12,

we iterate over each ℓ from $q^{j-1}, \ldots, q^j - 1$. So, we can express the total number of times these two loops iterate as:

$$\sum_{j=1}^{k}(q^j - 1 - q^{j-1} + 1) \quad = \sum_{j=1}^{k} q^j - \sum_{j=1}^{k} q^{j-1} \quad = q^k - 1 < q^k. \tag{3.2}$$

Equation 3.2 is also true for the pair of loops on lines 7 and 28, but this factor of q^k does not affect the running time complexity, since it is dominated in a Big-O representation by the larger factor of q^{k+1}.

Porat and Rothschild note that if some precomputations are done, we can complete all operations on line 19 in constant time. They claim it is sufficient to store floating point numbers using $O(1)$ space when calculating E. If we store $\mathrm{O}(\log n)$ of the most significant bits, and compare among these to find the minimum E, any floating point loss of precision will cause the goal function to fluctuate by the amount that was lost due to precision when choosing a minimum, but if the difference between some entries is small enough that loss of precision would cause a different E than would have been chosen if there was no loss of precision, then it is too small to increase the goal function to be larger than one and it does not affect whether the algorithm finds a correct generator matrix. This argument is very technical and we refer the reader to the original paper for more details [23].

Finally, the following values must be preprocessed to compute line 19 in constant time:

- Store $a!$ for all $a \in [1, m]$ to calculate binomials in constant time,
- Store $(1 - \frac{1}{q})^a$ for all $a \in [1, m]$,
- Store $(\frac{1}{q})^a$ for all $a \in [1, m]$.

Therefore, Algorithm 3.4 takes $O(m^2 q^{k+1})$ time.

We now claim that Algorithm 3.5 takes $\Theta(mq^k)$ running time. Since the algorithm gives a $[m, k, \lfloor \delta m \rfloor]_q$ linear code, this variation runs linearly with the size of the code. This running time is due to the three for-loops in lines 6, 7, and 9. Lines 6 gives the factor of m, and there are three groups of loops in this outermost loop. The two pair of loops on lines 7 and 9, and lines 7 and 22, both give the same factor of q^k as shown in Equation 3.2. The for-loop on line 15 does $\Theta(q)$ operations, but this is dominated by the for-loops on lines 9 and 22 in Big-Theta, so it does not affect the running time complexity. The operations on line 12 of Algorithm 3.5 can be completed in constant time, just like the operations in line 19 in Algorithm 3.4.

Therefore, Theorem 3.3 is proven, since we concluded part 2, that under the hypothesis of Theorem 3.3, it is posisble to construct a $[m, k, \lfloor \delta m \rfloor]_q$ in time $\Theta(mq^k)$ using Algorithm 3.5.

3.4 CFFs from Porat and Rothschild's codes

We first show how to build a CFF from a Porat and Rothschild code.

Lemma 3.13 (Adapted from [23], Lemma 3.1)**.** *An $[m, \log_q n, \delta m]_q$ code can be used to construct a $(\lfloor \frac{1}{1-\delta} \rfloor - 1)$-CFF$(mq, n)$.*

Proof. Using Proposition 2.7, we can build a d-CFF$(mq, n = q^k)$ where $k = \log_q n$, and $d = \lfloor \frac{m-1}{m-\delta m} \rfloor$. It remains to be shown that $d = \lfloor \frac{1}{1-\delta} \rfloor - 1$. We have

$$\begin{aligned}
d &= \left\lfloor \frac{m-1}{m-\delta m} \right\rfloor \\
&= \left\lfloor \frac{m(1-1/m)}{m(1-\delta)} \right\rfloor \\
&= \left\lfloor \frac{1-1/m}{1-\delta} \right\rfloor \\
&= \left\lfloor \frac{1}{1-\delta} - \frac{1}{m}\left(\frac{1}{1-\delta}\right) \right\rfloor \\
&= \left\lfloor \frac{1}{1-\delta} - \frac{1}{m(1-\delta)} \right\rfloor \\
&= \left\lfloor \frac{1}{1-\delta} - 1 \right\rfloor \qquad (\text{Since } \frac{1}{m(1-\delta)} < 1) \\
&= \left\lfloor \frac{1}{1-\delta} \right\rfloor - 1.
\end{aligned}$$

□

Now, we show the main theorem in [23].

Theorem 3.14 ([23], Theorem 1)**.** *Let n and d be positive integers. It is possible to construct a d-CFF(t, n) with $t = \Theta(\min\{d^2 \ln n, n\})$ in time $\Theta(dn \ln n)$.*

Sketch of the Proof. We first show, given n and d, how to pick parameters δ and q to obtain a d-CFF(t, n) with t in $\Theta(\min\{d^2 \ln n, n\})$.

If $(d+1)^2 \ln n \geq n$, simply return the identity matrix. So we can assume $(d+1)^2 \ln n < n$. Set $\delta = \frac{d}{d+1}$, $q \in [2(d+1), 4(d+1))$ be a prime power, $k = \log_q n$ and $m = \lceil \frac{k}{1-H_q(\delta)} \rceil$. Note that $\frac{1}{1-\delta} = d + 1 < q$, which implies $\delta < 1 - \frac{1}{q}$.

Porat and Rothschild use a technical argument to prove $m = \Theta(d \ln n)$, which we omit here and can be found in [23], Theorem 1. Since $k \leq (1 - H_q(\delta))m$, and $\delta < 1 - \frac{1}{q}$, use Theorem 3.3 to build a $[m, \log_q n, \lfloor \delta m \rfloor]_q$ code. From this code, use Lemma 3.13 to build a $(\lfloor \frac{1}{1-\delta} \rfloor - 1)$-CFF$(mq, n)$ in time $\Theta(mq^k)$.

Now we verify the parameters in the previous statement matches the parameters in the statement of this theorem. First,

$$\begin{aligned}\left\lfloor \frac{1}{1-\delta} - 1 \right\rfloor &= \left\lfloor \frac{1}{1-\frac{d}{d+1}} - 1 \right\rfloor \\ &= \left\lfloor \frac{1}{\frac{d+1-d}{d+1}} - 1 \right\rfloor \\ &= d+1-1 \\ &= d.\end{aligned}$$

Since $q \in [2(d+1), 4(d+1))$, then $q = \Theta(d)$, and since $m \in \Theta(d \ln n)$, we have $t = mq = \Theta(d^2 \ln n)$. Finally, since $m \in \Theta(d \ln n)$ and $q^k = n$, the time to build the CFF is $\Theta(mq^k) = \Theta(dn \ln n)$. In conclusion, a d-CFF(t, n) with $t = \Theta(d^2 \ln n)$ can be built in time $\Theta(dn \ln n)$. □

Algorithm 3.6 details the algorithm described in the proof of Theorem 3.14.

Algorithm 3.6 PRconstructCFF(n, d)

Input: $n, d \in \mathbb{Z}^+$, $n > d$.
Output: A d-CFF(t, n) with $t = \Theta(\min\{d^2 \ln n, n\})$.

1: **if** $(d+1)^2 \ln n \geq n$ **then**
2: **return** the identity matrix I_n
3: **end if**
4: Pick $q \in [2(d+1), 4(d+1))$ a prime power
5: $\delta \leftarrow \frac{d}{d+1}$
6: $k \leftarrow \log_q n$
7: $m \leftarrow \lceil \frac{k}{1-H_q(\delta)} \rceil$
8: $C \leftarrow$ ConstructGeneratorMatrixOptimized(m, k, q, δ)
9: **return** CFFfromCode(C)

Chapter 4

Building a Table of Best Known Cover-Free Families

In this chapter, we discuss algorithms for determining best-known upper bounds on $t(d, n)$, or equivalently, best-known lowers bounds $n(d, t)$ from the constructions discussed in Chapters 2 and 3. We construct tables of CFFs' best bounds in Section 4.2, then define a recursive algorithm that uses this table to explicitly construct a CFF in Section 4.7.

For each entry in the table for a d-CFF, there exists a d-CFF(t_1,n_1). Therefore, $t(d, n_1) \leq t_1$ and $n(d, t_1) \geq n_1$. Furthermore,

$$\begin{aligned} t_1 &= \min\{t|\text{there exists a } d\text{-CFF}(t,n_1) \text{ using considered constructions}\} \quad \text{and,} \\ n_1 &= \max\{n|\text{there exists a } d\text{-CFF}(t_1,n) \text{ using considered constructions}\}. \end{aligned}$$

The table for a given d is stored in table[d], which contains an array (rows) indexed by t_1 that stores n_1. In other words, for t_1 and n_1 defined above, table[d].rows[t_1].n=n_1.

4.1 Algorithm overview

The main idea of the algorithm used to construct the tables is as follows. We first fill each table with parameters for trivial CFFs, which are CFFs with $t = n$, an identity matrix. Then, we use combinations of parameters of constructions to search for better CFFs.

Because some of our selected constructions give infinite classes of CFFs, it is necessary to put an upper bound on the allowed t, n, and d in our tables. Let t_{max}, n_{max}, and d_{max} be these upper bounds.

For each d from $1 \ldots d_{max}$ CFF parameters are stored in a separate table[d]. The tables are filled in ascending order of d, because the Optimized Kronecker product (Proposition 2.12) and Doubling construction (Proposition 2.13) require a $(d-1)$-CFF in the construction of a d-CFF. The main steps of the algorithm are as follows:

- For each table[d], in ascending order of d from $1 \ldots d_{max}$:

- table[d] is filled with parameters of trivial CFFs, identity matrices;
- For each direct construction, update table[d] with the parameters used to construct every CFF that can be constructed using the direct construction within the bounds of t_{max} and n_{max};
- Update table[d] with one iteration of recursive constructions, and repeat iterations of recursive constructions until an iteration has not caused any changes to the table.

We include detailed pseudocode of this algorithm in Algorithm 4.10. In Sections 4.4 and 4.5 we discuss algorithms to fill the tables using direct and recursive constructions. These various algorithms are then each called in Algorithm 4.10 in Section 4.6 to get our final tables.

4.2 Implementation details

In this section, we detail the data structures used in creating the CFF tables. Our collection of tables contains an array of tables indexed by d, and each table is an array of rows indexed by t, which stores the largest n such that a d-CFF(t, n) exists using one of our considered constructions. Since we group CFFs by d into an array of tables, we have d_{max} different tables, where each table has at most t_{max} rows.

Each array of rows for a table is stored in an attribute called `rows` for each table. This array has length $t_{max} + 1$ so it can be accessed directly by t, and the unused index zero is always NULL. A table has two other attributes in addition to its array of rows: its array's size, `numCFFs`, and a boolean attribute, `hasBeenChanged`. This boolean is used when creating a table in Algorithm 4.10 to stop applying recursive constructions, whenever in the previous iteration, applying all recursive constructions resulted in no updates of the table.

```
1 typedef struct
2 {
3     bool hasBeenChanged;
4     int numCFFs;
5     CFF_Table_Row* rows; //an array of length numCFFs + 1
6 } CFF_Table;
```

Figure 4.1: Definition of the `CFF_Table` struct in C programming language.

Each position in a table stores the information about one CFF and the position is indexed by t, the number of rows of the CFF. Each position in the table has 4 fields: the number of columns of the CFF, `n`, an array storing its construction parameters, `constructionParameters`, a pointer to a structure of function pointers that can construct the CFF or return its name in `functions`, and a pointer to a cover-free family in `cff`.

This last field is used to avoid constructing the same CFF multiple times in a recursive algorithm in Section 4.7.

In our pseudocode, we refer to the `functions` attribute as "constructionName" and store a string instead for simplicity.

```
typedef void (*TableShortSourceName)(char*);
typedef void (*TableLongSourceName)(short*, char*);
typedef void (*ConstructionFunction)(int, int);

typedef struct
{
    TableShortSourceName shortSrcFormatter;
    TableLongSourceName longSrcFormatter;
    ConstructionFunction constructionFunction;
} CFF_Construction_And_Name_Functions;

typedef struct
{
    unsigned long long n;
    short constructionParameters[4];
    CFF_Construction_And_Name_Functions* functions;
    CoverFreeFamily* cff;
} CFF_Table_Row;
```

Figure 4.2: Definition of the `CFF_Table_Row` and related structs in C programming language.

There is only one `CFF_Construction_And_Name_Functions` instantiated for each different type of construction. The functions pointed to by the variables `shortSrcFormatter` and `longSrcFormatter` are used to get the name of the construction. The other member, `constructionFunction`, is a function pointer to the function that will construct a CFF.

Finally, we create a global array called "table", which is an array of pointers to each `CFF_Table` struct. This array has length $d_{max} + 1$ so that it can be directly indexed by the d for the table we want to retrieve, similar to the array in a `CFF_Table_Row` when retrieving a row by t. This lets us access the table using table[d].`rows`[t].

4.3 Updating a table

Algorithm 4.1 is used to update table[d]. It is called each time new parameters (t, n) that can be used to construct a d-CFF are found. This algorithm first checks if t is larger than the current maximum allowed in the table, which is stored in numCFFs. Next, it checks if n, is less than our bound of n_{max}, since if this is false then there is no need to update the table since this CFF would not change the table. Most importantly, we check if the newly found CFF parameters are an improvement on our table. If all of these things are true

(line 1), then the table will be updated with the newly found CFF parameters (usually in line 6). Finally, if n is greater than n_{max}, the extra columns will be discarded (line 3), and we record the fact that this is the last valid position in this table (line 4).

Algorithm 4.1 updateTable(d, t, n, constructionName, constructionParameters)

Input: $d, t, n \in \mathbb{Z}^+$. constructionName is a string, and constructionParameters is an array of integers.

Output: table[d] is updated if table[d].rows[t].n $< n$, t does not exceed table[d].numCFFs +1, and n does not exceed n_{max}.

```
1: if (t ≤ table[d].numCFFs) & (table[d].rows[t].n < n_max) & (table[d].rows[t].n < n) then
2:     if n > n_max then
3:         table[d].rows[t].n ← n_max
4:         table[d].numCFFs ← t + 1
5:     else
6:         table[d].rows[t].n ← n
7:     end if
8:     table[d].hasBeenChanged ← True
9:     table[d].rows[t].constructionName ← constructionName
10:    table[d].rows[t].constructionParameters ← constructionParameters
11: end if
```

We limit a table by both t_{max} and n_{max} using Algorithm 4.1, because table.numCFFs will be initially set to t_{max}. If a CFF is found with $t > t_{max}$ it is discarded. If $t \leq t_{max}$ and $n > n_{max}$, we can discard the extra columns to get a d-CFF(t, n_{max}). This process is detailed in Algorithm 4.1.

4.4 Direct constructions

In this section, we discuss filling the tables using direct constructions presented in Section 2.1.

4.4.1 Previously best-known 2-CFFs for small values

Li, van Rees, and Wei [19] give the following table of best known 2-CFFs. The values for $t = 3, \ldots, 11$ are proven to be the best possible bound on $t(2, n)$, and are in bold. These 2-CFFs are all constructed using binary constant-weight codes. Refer to Section 2.1.4 for this CFF construction. Idalino and Moura [17] updated the bound on $n(2, 20)$ by using a binary constant-weight code that had not been found at the time that [19] was published.

t	n	Source
$\mathbf{3 \leq y \leq 8}$	$\mathbf{y}$	**Identity matrix**
9	**12**	$\mathbf{A(9,4,3)}$
10	**13**	$\mathbf{A(10,4,3)}$
11	**17**	$\mathbf{A(11,4,3)}$
12	20	$A(12,4,3)$
13	26	$A(13,4,3)$
14	28	$A(14,4,3)$
15	42	$A(15,6,5)$
16	48	$A(16,6,5)$
17	68	$A(17,6,5)$
18	69	$A(18,6,5)$
19	76	$A(19,6,5)$
20	90	$A(20,6,5)$
21	120	$A(21,8,7)$
22	176	$A(22,8,7)$
23	253	$A(23,8,7)$

We update the 2-CFF table with these CFFs in algorithm "AddBestKnownCFFs()", which is called in Algorithm 4.10 before applying other constructions.

4.4.2 Sperner systems

Sperner systems are known to always be optimal 1-CFFs because of Sperner's theorem [24]. This construction has a single parameter, which is the t of the CFF. This construction was presented in Section 2.1.1. The algorithm works by iterating over every t in [1, t_{max}] and calculating the resulting $n = \binom{t}{\lfloor \frac{t}{2} \rfloor}$.

Algorithm 4.2 ApplySpernerConstruction()

Input: None.

Output: The table is updated with all CFFs from Sperner systems.

1: **for** t in $[1, t_{max}]$ **do**
2: $n \leftarrow \binom{t}{\lfloor \frac{t}{2} \rfloor}$
3: updateTable(1, t, n, "Sperner", NULL)
4: **end for**

4.4.3 Steiner Triple Systems

Steiner triple systems give 2-CFFs for all possible orders. The only parameters for constructing an STS is its order v, where $v \equiv 1, 3 \bmod 6$. This algorithm only updates the 2-CFF table, since STSs can only be used to construct 2-CFFs, as shown in Example 2.4. This algorithm loops over every possible STS that can be constructed with v up to t_{max}, and updates the 2-CFF table.

Algorithm 4.3 ApplySTSConstruction()

Input: None.

Output: The table is updated with all CFFs from Steiner triple systems.

1: **for** k in $[3, \lfloor \frac{t_{max}-1}{6} \rfloor]$ **do**
2: $v \leftarrow 6k+1$
3: updateTable(2, v, $\frac{v(v-1)}{6}$, "STS", NULL)
4: $v \leftarrow 6k+3$
5: updateTable(2, v, $\frac{v(v-1)}{6}$, "STS", NULL)
6: **end for**

4.4.4 Reed-Solomon linear codes

Reed-Solomon codes have three parameters: the alphabet length q, the message vector length k, and the codeword length m. Reed-Solomon codes can also be shortened with a parameter s, so after finding a Reed-Solomon code, all shortened variations are checked. This is given in Algorithm 4.4.

First, we iterate over all possible prime powers that can be used for the alphabet of the Reed-Solomon code. The largest prime power that is considered is $\sqrt{t_{max}}$. This is because after applying the construction for CFFs from codes (Proposition 2.7) to a Reed-Solomon code $[m, k, m-k+1]_q$, the resulting CFF is a $\lfloor \frac{m-1}{m-D} \rfloor$-CFF$(mq, q^k)$. In a Reed-Solomon code, we have the requirements that $m \leq q+1$ and $q \geq k-1 \geq 0$. Since we have $t = mq$ and $m \leq (q+1)$, we can conclude that $t \leq q(q+1) \leq q^2+q \leq t_{max}$. Because $q^2+q \leq t_{max}$, we have $q \leq \sqrt{t_{max}}$.

For each of these q, we iterate over all k in $[2,q+1]$. This is because we would never use $k = 1$, since this would result in a d-CFF(mq,q), which is the same or worse than a trivial CFF. Additionally, $q \geq k-1$ is a requirement of Reed-Solomon codes, so we only use $2 \leq k \leq q+1$. From here we calculate the minimum m for this q, k, and d, using $d = \lfloor \frac{m-1}{m-D} \rfloor$, resulting in $m = d(k-1)+1$, since $D = m-k+1$ in a Reed-Solomon code. At this point we update the table with the parameters to construct a CFF with this Reed-Solomon code, and then update the table with each shortened variation of this code (Proposition 2.8).

4.4.5 Porat and Rothschild linear codes

There are three parameters for the linear code construction from Porat and Rothschild [23] that we need to consider first. The alphabet length q, dimension k, and the d of the CFF. The resulting CFF is a d-CFF(mq, q^k) after applying Proposition 2.7 to construct a CFF from the code. This means we can choose d before determining other parameters, allowing us to fill one table for each d at a time.

Finally, we need to determine the codeword length m. Since m determines the number of rows in the CFF, it should be minimized. Let $\delta = \frac{d}{d+1}$. Determining m from the parameters q, k, d can be done as follows. Using the requirement that $k \leq (1 - H_q(\delta))m$ from the hypothesis of Theorem 3.3, we get:

Algorithm 4.4 ApplyReed-SolomonConstruction(d)

Input: $2 \leq d \leq d_{max}$
Output: The table is updated with all CFFs from Reed-Solomon codes.

```
for each prime power q in [2, ⌈√t_max⌉] do
    for k ← 2 ... q + 1 do
        m ← d(k − 1) + 1
        if m > q + 1 then
            break
        end if
        updateTable(d, mq, q^k, "RS code", [q, k, m])
        for s in [1,m − 1] do
            updateTable(d, (m − s)q, q^{k−s}, "Short RS code", [q, k, m, s])
        end for
    end for
end for
```

$$\begin{aligned} & k \leq (1 - H_q(\delta))m, \\ \Leftrightarrow \quad & m \geq \frac{k}{(1 - H_q(\delta))}. \end{aligned}$$

Since we are minimizing m, we take $m = \lceil \frac{k}{1-H_q(\delta)} \rceil$.

Algorithm 4.5 first calculates $\delta = \frac{d}{d+1}$ from the parameter d. Then, we iterate over each possible q, which is all prime powers in $[2(d+1), 4(d+1))$. Then, for each possible q, we iterate over every k until the resulting CFF would have $t > t_{max}$, calculate m as shown above for each k, and update the table with this d-CFF(mq, q^k).

Algorithm 4.5 ApplyPoratAndRothschildConstruction(d)

Input: $2 \leq d \leq d_{max}$
Output: The table is updated with all CFFs from the Porat and Rothschild construction.

```
δ ← d/(d+1)
for each prime power q in [2(d + 1), 4(d + 1)) do
    k ← 2
    m ← ⌈k/(1−H_q(δ))⌉
    while mq ≤ t_max do
        updateTable(d, mq, q^k, "PR code", NULL)
        k ← k + 1
        m ← ⌈k/(1−H_q(δ))⌉
    end while
end for
```

4.5 One iteration of recursive constructions

After the tables are filled using various direct constructions, we apply recursive constructions. These are constructions that use other CFFs to make new CFFs. In order to apply these recursive constructions to each table, we do one iteration at a time until no new better CFFs are found.

When building the tables we need do so sequentially from $1, \ldots, d_{max}$ because the optimized Kronecker product construction will need to use the table for $d-1$ to find the best $(d-1)$-CFF to use in the construction of a d-CFF.

One iteration involves trying to update the table with each combinations of CFF parameters currently in the table. The idea is to update the table with the result of recursive constructions with every combination of CFF parameters currently in the tables, and try to add each resulting CFF to the table. We stop once an iteration has not caused changes in the tables.

First, we define a binary search algorithm that is used later in creating the tables, and in Algorithm 4.11. It will return the t of the CFF with n as the parameter passed to the algorithm. If no CFF in the table exists with this exact n, the t of the CFF with the smallest n larger than the specified n is returned.

Algorithm 4.6 BinarySearchOverCFFTable(d, n)

Input: $1 \leq d \leq d_{max}$
Output: The minimum index i such that table[d].rows[t].n $\geq n$ if it exists, otherwise, -1

```
1: low ← 1
2: high ← table.numCFFs
3: if table[d].rows[high].n < n then
4:     return -1
5: end if
6: while low < high do
7:     mid ← ⌊(high+low)/2⌋
8:     if n = table[d].rows[mid].n then
9:         return mid
10:    end if
11:    if n < table[d].rows[mid].n then
12:        high ← mid −1
13:    else
14:        low ← mid +1
15:    end if
16: end while
17: return low
```

4.5.1 Doubling construction

To complete one iteration of the doubling construction (Proposition 2.13), we update the table by seeing the results of applying this construction to each row of the 2-CFF table. The construction takes a single 2-CFF(t, n) as a parameter, and it must find the value $s = t(1, n)$. It constructs a 2-CFF($t + s + 2 - (s \ \% \ 2)$, $2n$). Parameter s is found using the binary search algorithm described in Algorithm 4.6 on the table for $d = 1$.

Algorithm 4.7 works by iterating over every 2-CFF currently in the table, and updates the table with the result of applying the doubling construction to each 2-CFF.

Algorithm 4.7 ApplyDoublingConstruction()

Input: None
Output: The 2-CFF table is updated with all CFFs from the doubling construction.

```
1: for t in [3, t_max] do
2:     n ← table[2].rows[t]
3:     s ← binarySearchTable(1, n)
4:     if s ≠ −1 then
5:         updateTable(2, t + s + 2 − (s%2), 2n, "Doubling", [t, s])
6:     end if
7: end for
```

4.5.2 Pair constructions

The other recursive constructions are the Kronecker Product (Proposition 2.11), Optimized Kronecker (Proposition 2.12), and the Additive construction (Proposition 2.14). To apply these constructions for one iteration, we call updateTable (Algorithm 4.1) for each pair of CFFs currently present in the tables with the resulting CFF obtained from applying each recursive construction to a pair of CFFs. This is done in Algorithm 4.8. For the Optimized Kronecker, we need to use three CFFs instead of two, and find s for a $(d-1)$-CFF(s, n_2). This is done using a binary search over the smaller table. If no $(d-1)$-CFF with n_2 is in our table, the binary search will find s for the $(d-1)$-CFF(s, n) that has the smallest n, $n > n_2$, and the extra columns are ignored in the Optimized Kronecker construction.

4.5.3 Extension by one

Algorithm 4.9 updates the table with the extension by one constructions (Proposition 2.15). This works by appending a row of all zeros to the bottom, a column of all zeros to the right, and setting the new bottom right-most cell to 1. We iterate over each d-CFF and apply the construction once in increasing order of t, so improvements propagate.

Algorithm 4.8 ApplyPairConstructions(d)

Input: $2 \leq d \leq d_{max}$
Output: The table is updated with all CFFs obtained from the Kronecker, Optimized Kronecker, and Additive constructions.

```
1: for t_1 in [d+1, t_max] do
2:     for t_2 in [t_1, t_max] do
3:         n_1 ← table[d].rows[t_1].n
4:         n_2 ← table[d].rows[t_2].n
5:         updateTable(d, t_1 t_2, n_1 n_2, "Kronecker", [t_1, t_2])
6:         updateTable(d, t_1 + t_2, n_1 + n_2, "Additive", [t_1, t_2])
7:         s ← binarySearchTable(d − 1, n_1)
8:         if s ≠ −1 then
9:             updateTable(d, s t_1 + t_2, n_1 n_2, "Optimized Kronecker", [t_1, t_2, s])
10:        end if
11:        s ← binarySearchTable(d − 1, n_2)
12:        if s ≠ −1 then
13:            updateTable(d, s t_2 + t_1, n_1 n_2, "Optimized Kronecker", [t_2, t_1, s])
14:        end if
15:    end for
16: end for
```

Algorithm 4.9 ApplyExtensionByOne(d)

Input: $2 \leq d \leq d_{max}$
Output: The table is updated with all CFFs from the extension by one construction.

```
1: for t in [d+1, t_max] do
2:     n ← table[d].rows[t].n
3:     updateTable(d, t + 1, n + 1, "Extension by one", NULL)
4: end for
```

4.6 Filling the tables using these algorithms

Algorithm 4.10 is the algorithm that will create the tables of best CFF parameters. This algorithm works as follows. First, it calls once the algorithms that update the tables with the parameters of direct constructions. Then, it generates the parameters using recursive constructions until the table does not change.

Algorithm 4.10 first initializes the tables with the trivial d-CFF(t,t) parameters in lines 2-11.

The algorithm creates three types of tables: The 1-CFF table, 2-CFF table, and the d-CFF tables for $d \geq 3$. These tables need to be created separately because we have different constructions for $d = 1, d = 2$, and $d \geq 3$.

First, the 1-CFF table is created from the parameters for the Sperner construction (Section 2.1.1) in line 12.

For the 2-CFF table, we enter the parameters for the direct constructions from STSs (Section 2.4), Reed-Solomon codes (Section 2.1.3.1), linear codes from Porat and Rothschild (Section 2.1.3.1), and best known 2-CFFs parameters from binary constant-weight codes (Section 2.1.4), in lines 13-16. Then we generate the parameters for the recursive constructions for 2-CFFs, which are the doubling construction (Proposition 2.13), extension by one (Proposition 2.15), additive construction (Proposition 2.14), Kronecker product (Proposition 2.11), and the optimized Kronecker product (Section 2.12). This is done repeatedly in a loop in lines 17-22 until no updates are observed.

After creating the 1-CFF and 2-CFF tables, we then create the tables for $d \geq 3$. Our selection of direct constructions for d-CFFs with $d \geq 3$ are the Reed-Solomon code construction, and the code construction from Porat and Rothschild in lines 24-25. After we fill a d-CFF table with the parameters of these direct constructions, we apply the additive, Kronecker, and optimized Kronecker recursive constructions until an iteration does not cause any changes to the table (lines 26-30).

It is easy to verify that Algorithm 4.10 produces a table with the best CFFs that can be obtained by using the constructions in Table 2.1 and the construction of 2-CFFs from binary constant-weight codes. The justification is as follows. Since Sperner's construction gives the best 1-CFFs, after line 12, we have table[1].rows[t].$n = n(1, t)$. For each $d \geq 2$, we can argue that at the end of the corresponding loop, table[d].rows[t].n has the largest value of n achievable with the given constructions. The main argument is that table[d].rows[t].n only depends on having table[$d-1$].rows[t'].n completed with the best values for any $1 \leq t' \leq t_{max}$ and on having table[d].rows[t''].n completed with best values for $1 \leq t'' < t$ (here "best value" refers to the largest n achievable with the given constructions). An inductive argument on d and then t can therefore be used to prove our claim.

Algorithm 4.10 makeTables($d_{max}, t_{max}, n_{max}$)

Input: $d_{max}, t_{max}, n_{max} \in \mathbb{Z}^+$.
Output: An array of `CFF_Table` with d_{max} tables. This is a global array.

```
1: table ← an array of d_max tables, with indices starting at 1
2: for d in [1, d_max] do                ▷ Initialize tables. Fill each table with ID matrices
3:     table[d].numCFFs ← t_max
4:     table[d].hasBeenChanged ← True
5:     for t ← d + 1 to t_max do
6:         table[d].rows[t].n ← t
7:         table[d].rows[t].constructionName ← "ID Matrix"
8:         table[d].rows[t].constructionParameters ← NULL
9:         table[d].rows[t].CFF ← NULL
10:    end for
11: end for
12: ApplySpernerConstruction(1)                                  ▷ Make 1-CFF table
13: AddBestKnownCFFs(2)          ▷ Fill 2-CFF table with binary constant-weight codes
14: ApplySTSConstruction(2)
15: ApplyReedSolomonConstruction(2)
16: ApplyPRConstruction(2)
17: while table[2].hasBeenChanged do
18:     table[2].hasBeenChanged ← False
19:     ApplyPairConstructions(2)
20:     ApplyDoublingConstruction(2)
21:     ApplyExtensionByOne(2)
22: end while
23: for d in [3, d_max] do                    ▷ Start filling d-CFF tables for d ≥ 3
24:     ApplyReedSolomonConstruction(d)
25:     ApplyPRConstruction(d)
26:     while table[d].hasBeenChanged do
27:         table[d].hasBeenChanged ← False
28:         ApplyPairConstructions(d)
29:         ApplyExtensionByOne(d)
30:     end while
31: end for
```

4.7 Constructing a cover-free family from the table of parameters

The idea for the algorithms constructByT(d, t) (Algorithm 4.11) and constructByN(d, n) (Algorithm 4.12) is to first construct the tables using Algorithm 4.10, then call a recursive algorithm defined in Algorithm 4.13 that uses the tables to construct a CFF. We define Algorithm 4.11 to construct a CFF when given a d and t, and Algorithm 4.12 to construct

a CFF when given a d and n.

These algorithms have access to the global table, created in Algorithm 4.10. Algorithm 4.10 is called in the pseudocode of Algorithms 4.11 and 4.12 for clarity, but instead we could create the tables outside of these algorithms, then re-use the table many times if we want to construct multiple CFFs.

Algorithm 4.11 ConstructByT(d, t)

Input: $d, t, \in \mathbb{Z}^+, d < t$.
Output: The CFF corresponding to our bound on $n(d, t)$.

1: makeTables$(d_{max}, t_{max}, n_{max})$
2: CFF $\leftarrow$ getByT(d, t)
3: **return** CFF

Algorithm 4.12 ConstructByN(d, n)

Input: $d, n, \in \mathbb{Z}^+, d < n$.
Output: The CFF corresponding to our bound on $t(d, n)$.

1: makeTables$(d_{max}, t_{max}, n_{max})$
2: $t \leftarrow$ binarySearchTable$(d,\ n)$
3: CFF $\leftarrow$ getByT(d, t)
4: **return** CFF

Algorithms 4.11 and 4.12 use the recursive Algorithm 4.13. The base cases for the recursion are when the CFF in table d at row t uses a direct construction. The recursive cases are when the CFF is constructed with a recursive construction. Algorithm 4.13 avoids reconstructing the same CFF multiple times by storing a pointer to a CFF in each row in each table. If this pointer is NULL then the CFF will be constructed, otherwise the CFF is retrieved from the pointer. This behaviour is managed by the first if statement on line 1 of Algorithm 4.10. Then, the switch case statement handles the various cases for each construction. In our actual implementation we do not use a switch, since C does not support switches on strings. Instead, each case of the switch is actually the function pointed to by "ConstructionFunction" in a `CFF_Construction_And_Name_Functions` struct (Figure 4.2).

Algorithm 4.13 omits CFFs from binary constant-weight codes, detailed in Section 4.4.1, since we have not constructed a database of them. In our tables, presented later in Chapter 5, we present two variations of the 2-CFF table, one table that includes binary constant-weight codes, and another table without them.

In Table 4.1, we provide references to the various constructions used in Algorithm 4.13.

Direct constructions	
Sperner systems	Algorithm 2.2
STS	Example 2.4
Reed-Solomon	Algorithm 2.4
Shortened Reed-Solomon	Proposition 2.8
Porat and Rothschild	Algorithm 3.6
Recursive constructions	
Kronecker product	Algorithm 2.5
Optimized Kronecker product	Algorithm 2.6
Doubling for $d = 2$	Proposition 2.13
Additive	Proposition 2.14
Extension-by-one	Proposition 2.15

Table 4.1: Constructions used in Algorithm 4.13

Algorithm 4.13 getByT(d, t)

Input: $d, t, \in \mathbb{Z}^+, d < t$.
Output: The CFF corresponding to our bound on $n(d, t)$.

```
1: if table[d].rows[t].CFF = NULL then
2:     switch table[d].rows[t].constructionName do
3:         case "ID Matrix"
4:             CFF ← IDconstruction(t)
5:         case "Sperner"
6:             CFF ← SpernerConstruction(t)
7:         case "STS"
8:             CFF ← STSconstruction(t)
9:         case "RS Code"
10:            CFF ← ReedSolomonConstruction(
11:                table[d].rows[t].constructionParameters)
12:        case "Short RS Code"
13:            CFF ← ShortenedReedSolomonConstruction(
14:                table[d].rows[t].constructionParameters)
15:        case "PR Code"
16:            CFF ← PRconstructCFF(table[d].rows[t].n, d)
17:        case "Doubling"
18:            smaller2CFF ← getByT(d, table[d].rows[t].constructionParameters[0])
19:            smaller1CFF ← getByT(d − 1, table[d].rows[t].constructionParameters[1])
20:            CFF ← DoublingConstruction(smaller2CFF, smaller1CFF)
21:        case "Extension By One"
22:            smallerCFF ← getByT(d, t − 1)
23:            CFF ← ExtensionByOne(smallerCFF)
24:        case "Additive Construction"
25:            left ← getByT(d, table[d].rows[t].constructionParameters[0])
26:            right ← getByT(d, table[d].rows[t].constructionParameters[1])
27:            CFF ← AdditiveConstruction(left, right)
28:        case "Kronecker Product"
29:            left ← getByT(d, table[d].rows[t].constructionParameters[0])
30:            right ← getByT(d, table[d].rows[t].constructionParameters[1])
31:            CFF ← KroneckerProduct(left, right)
32:        case "Optimized Kronecker Product"
33:            a ← getByT(d, table[d].rows[t].constructionParameters[0])
34:            b ← getByT(d, table[d].rows[t].constructionParameters[1])
35:            c ← getByT(d − 1, table[d].rows[t].constructionParameters[2])
36:            CFF ← OptimizedKroneckerProductConstruction(a, b, c)
37:    table[d].rows[t].CFF ← CFF
38: else
39:    CFF ← table[d].rows[t].CFF
40: end if
41: return CFF
```

Chapter 5

Tables of Cover-Free Families from Given Constructions

In this chapter, we show the results of our algorithms to find best-known CFFs using our method outlined in Chapter 4, discuss observations from these tables, and discuss a comparison between CFFs constructed from Reed-Solomon codes and CFFs constructed using Porat and Rothschild's code construction. Our tables are available online in [6].

5.1 Tables

These tables show the CFFs that minimize $t(d, n)$ and maximize $n(d, t)$ for the constructions shown in Chapter 2 up to a maximum d of 25, and a maximum n of $10,000,000,000,000$ (10 trillion). We set $t_{max} = 20,000$, but we found that the tables were bounded by n_{max} before t_{max} was reached.

To reduce the number of rows in the tables we exclude every row that is from the extension-by-one construction, or would be possible to create with the extension-by-one construction. So, any gaps in t should be understood as consecutive increases by one in both n and t. The first example of this is on the 2-CFF table, between $t = 9$ and $t = 11$. Since the increase in n between one row from $t = 9$ to $t = 10$ is only one, this row with $t = 10$ and $n = 13$ is excluded from the table.

We use a label to refer to each construction in the tables, to save space.

Construction	Table Name	Resulting CFF
Identity Matrix	$\mathrm{ID}(t)$	d-CFF(t, t)
Sperner System	$\mathrm{Sp}(t)$	1-CFF$(t, \binom{t}{\lfloor \frac{t}{2} \rfloor})$
Steiner Triple System	$\mathrm{STS}(v)$	2-CFF$(v, \frac{v(v-1)}{6})$
Binary constant-weight code	$\mathrm{A}(m, D, w)$	2-CFF$(m, A(m, D, w))$
Doubling construction	$\mathrm{Dbl}(t, s)$	2-CFF$(t + s + 2 - (s\%2), 2n)$
Reed-Solomon code	$\mathrm{RS}(q, k, m)$	d-CFF(mq, q^k)
Shortened Reed-Solomon code	$\mathrm{SRS}(q, k, m, s)$	d-CFF$((m - s)q, q^{k-s})$
Porat and Rothschild code	$\mathrm{PR}(q, k, m)$	d-CFF(mq, q^k)
Kronecker product	$\mathrm{Kr}(t_1, t_2)$	d-CFF(t_1t_2, n_1n_2)
Optimized Kronecker product	$\mathrm{OKr}(t_1, t_2, s)$	d-CFF$(st_1 + t_2, n_1n_2)$
Additive construction	$\mathrm{Add}(t_1,t_2)$	d-CFF$(t_1 + t_2, n_1 + n_2)$

Table 5.1: Labels of construction names

We present tables for $d = 1, 2, 3, 4, 10, 20, 25$ in this chapter. The other tables are available in the appendix.

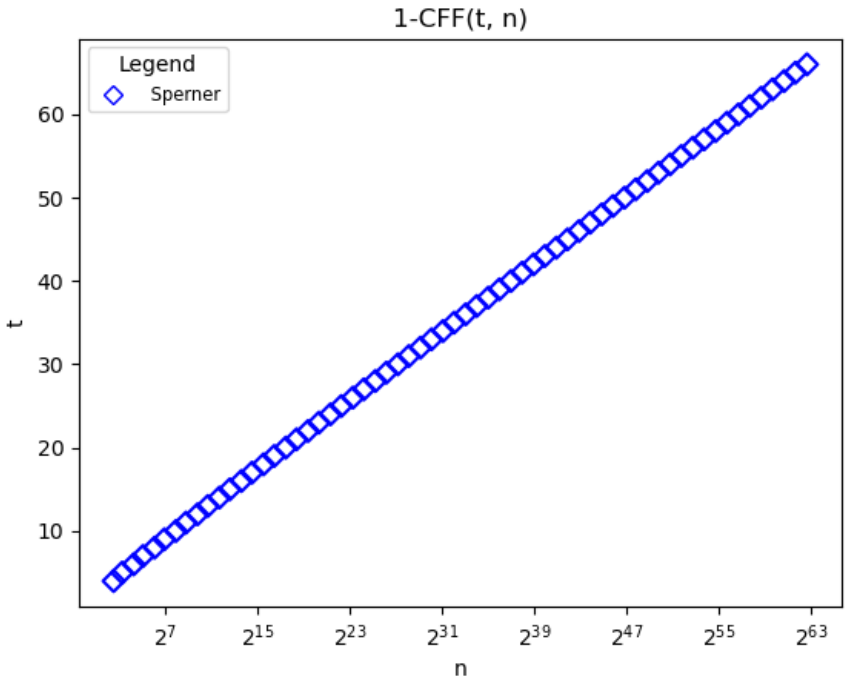
The total running time for creating all tables was roughly 75 seconds using a personal computer with an Intel i7-6700k processor and 16 GB of memory. The maximum memory usage of the program when creating the tables was 192 MB.

The number of iterations needed to apply recursive constructions to the CFF tables (see loops in lines 17 and 26 of Algorithm 4.10) is listed in Table 5.2.

d	Number of iterations
1	0
2	3
≥ 3	2

Table 5.2: Number of iterations of recursive constructions

5.1.1 1-CFF table



t	n	Source
4	6	Sp(4)
5	10	Sp(5)
6	20	Sp(6)
7	35	Sp(7)
8	70	Sp(8)
9	126	Sp(9)
10	252	Sp(10)
11	462	Sp(11)
12	924	Sp(12)
13	1716	Sp(13)
14	3432	Sp(14)
15	6435	Sp(15)
16	12870	Sp(16)
17	24310	Sp(17)
18	48620	Sp(18)
19	92378	Sp(19)
20	184756	Sp(20)
21	352716	Sp(21)
22	705432	Sp(22)
23	1352078	Sp(23)
24	2704156	Sp(24)
25	5200300	Sp(25)

t	n	Source
26	10400600	Sp(26)
27	20058300	Sp(27)
28	40116600	Sp(28)
29	77558760	Sp(29)
30	155117520	Sp(30)
31	300540195	Sp(31)
32	601080390	Sp(32)
33	1166803110	Sp(33)
34	2333606220	Sp(34)
35	4537567650	Sp(35)
36	9075135300	Sp(36)
37	17672631900	Sp(37)
38	35345263800	Sp(38)
39	68923264410	Sp(39)
40	137846528820	Sp(40)
41	269128937220	Sp(41)
42	538257874440	Sp(42)
43	1052049481860	Sp(43)
44	2104098963720	Sp(44)
45	4116715363800	Sp(45)
46	8233430727600	Sp(46)
47	10000000000000	Sp(47)

5.1.2 2-CFF table

We present two variations of the 2-CFF tables, because we have not constructed a database of binary constant-weight codes. So, we cannot construct CFFs from them using Algorithm 4.13, but we still give a variation of the 2-CFF table with them considered.

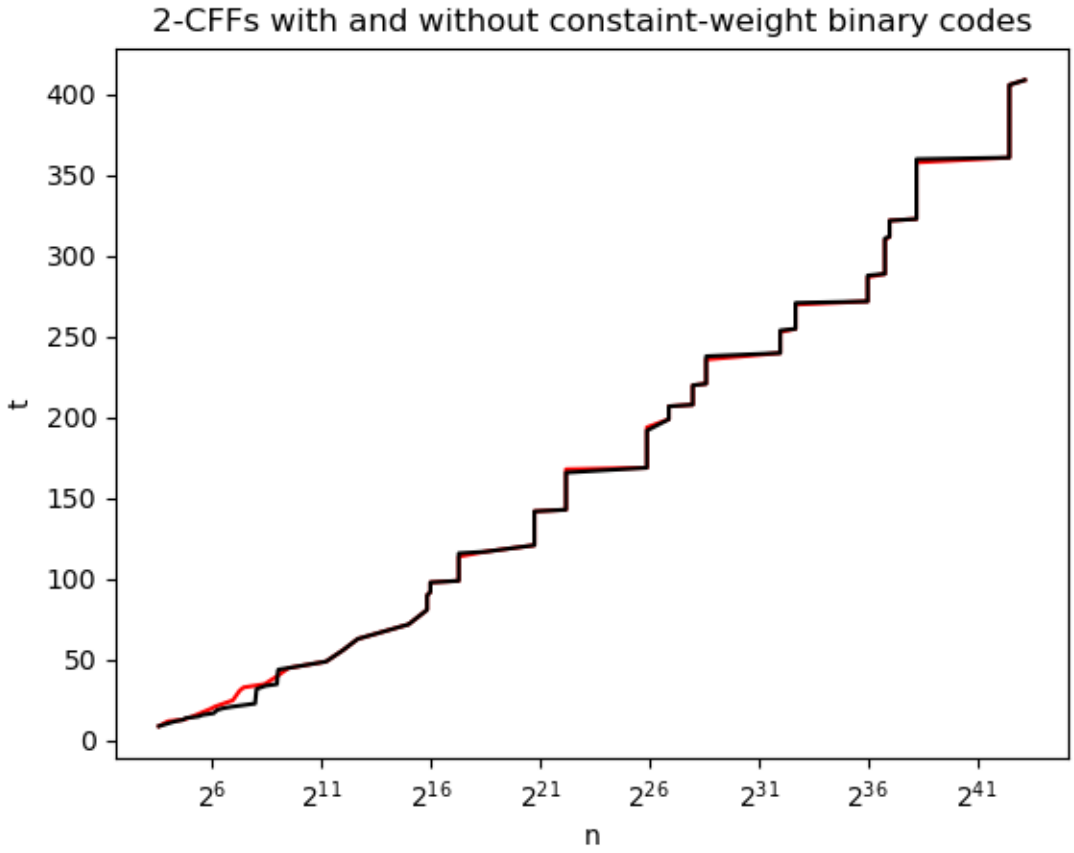
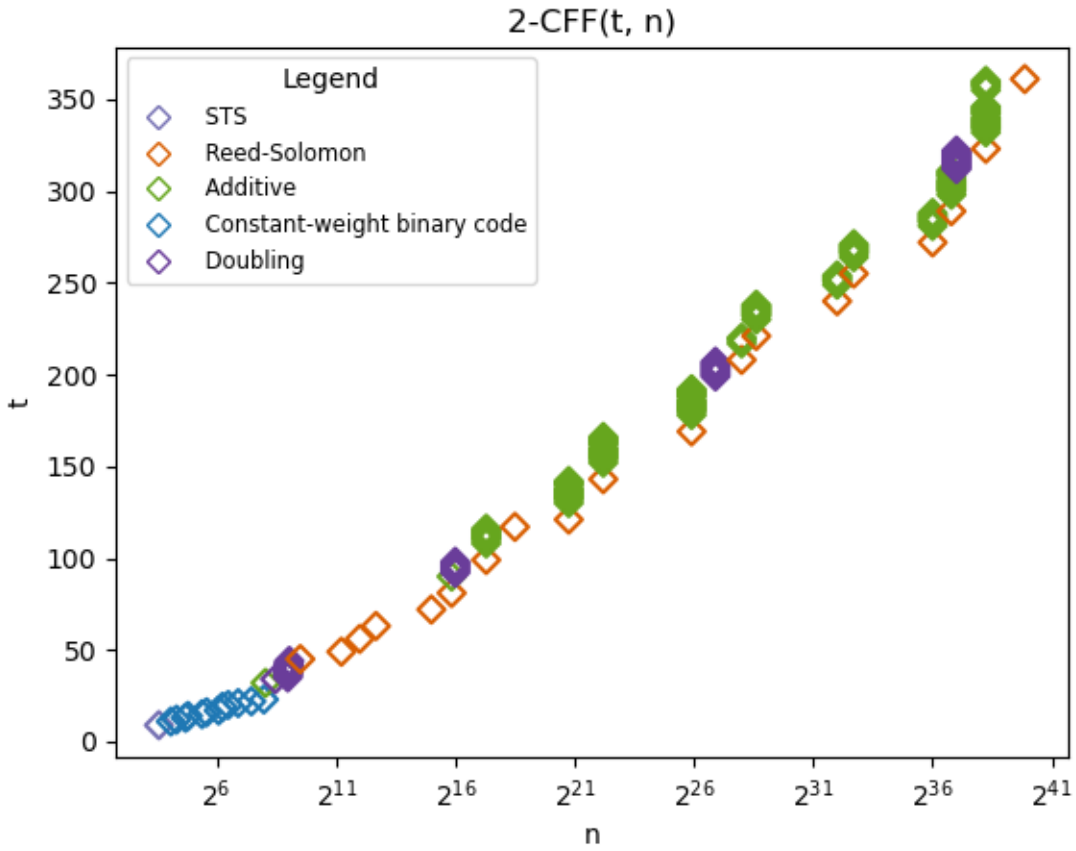


Figure 5.1: 2-CFF Table comparison

Figure 5.1 shows the growth of our bound on $t(2, n)$ with or without binary constant-weight codes. The black line includes CFFs from binary constant-weight codes, and the red line does not include them.

5.1.2.1 2-CFF table with binary codes



t	n	Source
9	12	STS(9)
11	17	A(11, 4, 3)
12	20	A(12, 4, 3)
13	26	A(13, 4, 3)
14	28	A(14, 4, 3)
15	42	A(15, 6, 5)
16	48	A(16, 6, 5)
17	68	A(17, 6, 5)
19	76	A(19, 6, 5)
20	90	A(20, 6, 5)
21	120	A(21, 8, 7)
22	176	A(22, 8, 7)
23	253	A(23, 8, 7)
32	265	Add(9,23)
34	352	Dbl(22,10)
35	506	Dbl(23,11)
36	508	Dbl(24,11)
37	510	Dbl(25,11)
38	512	Dbl(26,11)
39	514	Dbl(27,11)
40	516	Dbl(28,11)
41	518	Dbl(29,11)
42	520	Dbl(30,11)
43	522	Dbl(31,11)
44	530	Dbl(32,11)
45	729	RS(3^2,3,5)
49	2401	RS(7^1,4,7)
56	4096	RS(2^3,4,7)
63	6561	RS(3^2,4,7)
72	32768	RS(2^3,5,9)
81	59049	RS(3^2,5,9)
90	59061	Add(9,81)
92	65536	Dbl(72,18)
93	65538	Dbl(73,18)
94	65540	Dbl(74,18)
95	65542	Dbl(75,18)
96	65544	Dbl(76,18)
97	65546	Dbl(77,18)
98	65548	Dbl(78,18)
99	161051	RS(11^1,5,9)
108	161063	Add(9,99)
110	161068	Add(11,99)
111	161071	Add(12,99)
112	161077	Add(13,99)
113	161079	Add(14,99)

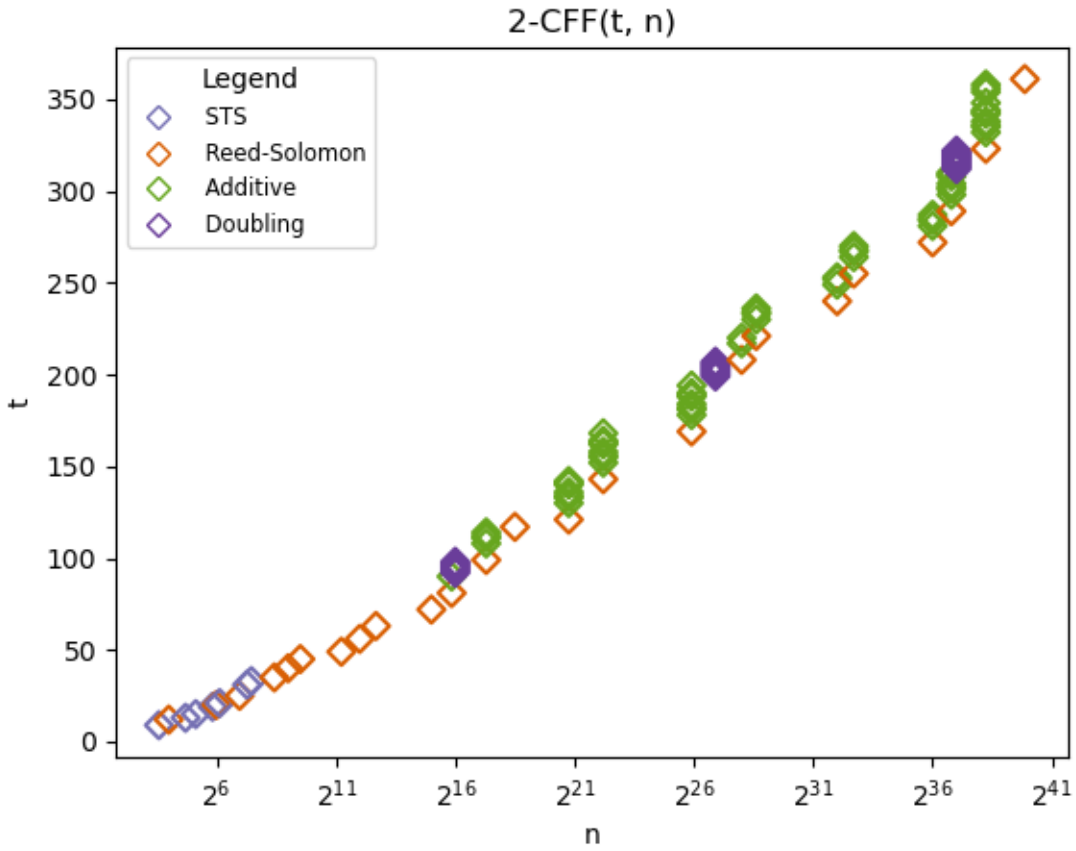
t	n	Source
114	161093	Add(15,99)
115	161099	Add(16,99)
116	161119	Add(17,99)
117	371293	RS(13^1,5,9)
121	1771561	RS(11^1,6,11)
130	1771573	Add(9,121)
132	1771578	Add(11,121)
133	1771581	Add(12,121)
134	1771587	Add(13,121)
135	1771589	Add(14,121)
136	1771603	Add(15,121)
137	1771609	Add(16,121)
138	1771629	Add(17,121)
140	1771637	Add(19,121)
141	1771651	Add(20,121)
142	1771681	Add(21,121)
143	4826809	RS(13^1,6,11)
152	4826821	Add(9,143)
154	4826826	Add(11,143)
155	4826829	Add(12,143)
156	4826835	Add(13,143)
157	4826837	Add(14,143)
158	4826851	Add(15,143)
159	4826857	Add(16,143)
160	4826877	Add(17,143)
162	4826885	Add(19,143)
163	4826899	Add(20,143)
164	4826929	Add(21,143)
165	4826985	Add(22,143)
166	4827062	Add(23,143)
169	62748517	RS(13^1,7,13)
178	62748529	Add(9,169)
180	62748534	Add(11,169)
181	62748537	Add(12,169)
182	62748543	Add(13,169)
183	62748545	Add(14,169)
184	62748559	Add(15,169)
185	62748565	Add(16,169)
186	62748585	Add(17,169)
188	62748593	Add(19,169)
189	62748607	Add(20,169)
190	62748637	Add(21,169)
191	62748693	Add(22,169)
192	62748770	Add(23,169)
199	125497034	Dbl(169,29)

t	n	Source
200	125497036	Dbl(170,29)
201	125497038	Dbl(171,29)
202	125497040	Dbl(172,29)
203	125497042	Dbl(173,29)
204	125497044	Dbl(174,29)
205	125497046	Dbl(175,29)
206	125497048	Dbl(176,29)
207	125497050	Dbl(177,29)
208	268435456	RS(2^4,7,13)
217	268435468	Add(9,208)
219	268435473	Add(11,208)
220	268435476	Add(12,208)
221	410338673	RS(17^1,7,13)
230	410338685	Add(9,221)
232	410338690	Add(11,221)
233	410338693	Add(12,221)
234	410338699	Add(13,221)
235	410338701	Add(14,221)
236	410338715	Add(15,221)
237	410338721	Add(16,221)
238	410338741	Add(17,221)
240	4294967296	RS(2^4,8,15)
249	4294967308	Add(9,240)
251	4294967313	Add(11,240)
252	4294967316	Add(12,240)
253	4294967322	Add(13,240)
254	4294967324	Add(14,240)
255	6975757441	RS(17^1,8,15)
264	6975757453	Add(9,255)
266	6975757458	Add(11,255)
267	6975757461	Add(12,255)
268	6975757467	Add(13,255)
269	6975757469	Add(14,255)
270	6975757483	Add(15,255)
271	6975757489	Add(16,255)
272	68719476736	RS(2^4,9,17)
281	68719476748	Add(9,272)
283	68719476753	Add(11,272)
284	68719476756	Add(12,272)
285	68719476762	Add(13,272)
286	68719476764	Add(14,272)
287	68719476778	Add(15,272)
288	68719476784	Add(16,272)
289	118587876497	RS(17^1,9,17)
298	118587876509	Add(9,289)

t	n	Source
300	118587876514	Add(11,289)
301	118587876517	Add(12,289)
302	118587876523	Add(13,289)
303	118587876525	Add(14,289)
304	118587876539	Add(15,289)
305	118587876545	Add(16,289)
306	118587876565	Add(17,289)
308	118587876573	Add(19,289)
309	118587876587	Add(20,289)
310	118587876617	Add(21,289)
311	118587876673	Add(22,289)
312	137438953472	Dbl(272,39)
313	137438953474	Dbl(273,39)
314	137438953476	Dbl(274,39)
315	137438953478	Dbl(275,39)
316	137438953480	Dbl(276,39)
317	137438953482	Dbl(277,39)
318	137438953484	Dbl(278,39)
319	137438953486	Dbl(279,39)
320	137438953488	Dbl(280,39)
321	137438953496	Dbl(281,39)
322	137438953498	Dbl(282,39)
323	322687697779	RS(19^1,9,17)
332	322687697791	Add(9,323)
334	322687697796	Add(11,323)
335	322687697799	Add(12,323)
336	322687697805	Add(13,323)
337	322687697807	Add(14,323)
338	322687697821	Add(15,323)
339	322687697827	Add(16,323)
340	322687697847	Add(17,323)
342	322687697855	Add(19,323)
343	322687697869	Add(20,323)
344	322687697899	Add(21,323)
345	322687697955	Add(22,323)
346	322687698032	Add(23,323)
355	322687698044	Add(23,332)
357	322687698131	Add(34,323)
358	322687698285	Add(35,323)
359	322687698287	Add(36,323)
360	322687698289	Add(37,323)
361	6131066257801	RS(19^1,10,19)
370	6131066257813	Add(9,361)
372	6131066257818	Add(11,361)
373	6131066257821	Add(12,361)

t	n	Source
374	6131066257827	Add(13,361)
375	6131066257829	Add(14,361)
376	6131066257843	Add(15,361)
377	6131066257849	Add(16,361)
378	6131066257869	Add(17,361)
380	6131066257877	Add(19,361)
381	6131066257891	Add(20,361)
382	6131066257921	Add(21,361)
383	6131066257977	Add(22,361)
384	6131066258054	Add(23,361)
393	6131066258066	Add(23,370)
395	6131066258153	Add(34,361)
396	6131066258307	Add(35,361)
397	6131066258309	Add(36,361)
398	6131066258311	Add(37,361)
399	6131066258313	Add(38,361)
400	6131066258315	Add(39,361)
401	6131066258317	Add(40,361)
402	6131066258319	Add(41,361)
403	6131066258321	Add(42,361)
404	6131066258323	Add(43,361)
405	6131066258331	Add(44,361)
406	6131066258530	Add(45,361)
409	10000000000000	Dbl(361,46)

5.1.2.2 2-CFF table without binary constant-weight codes

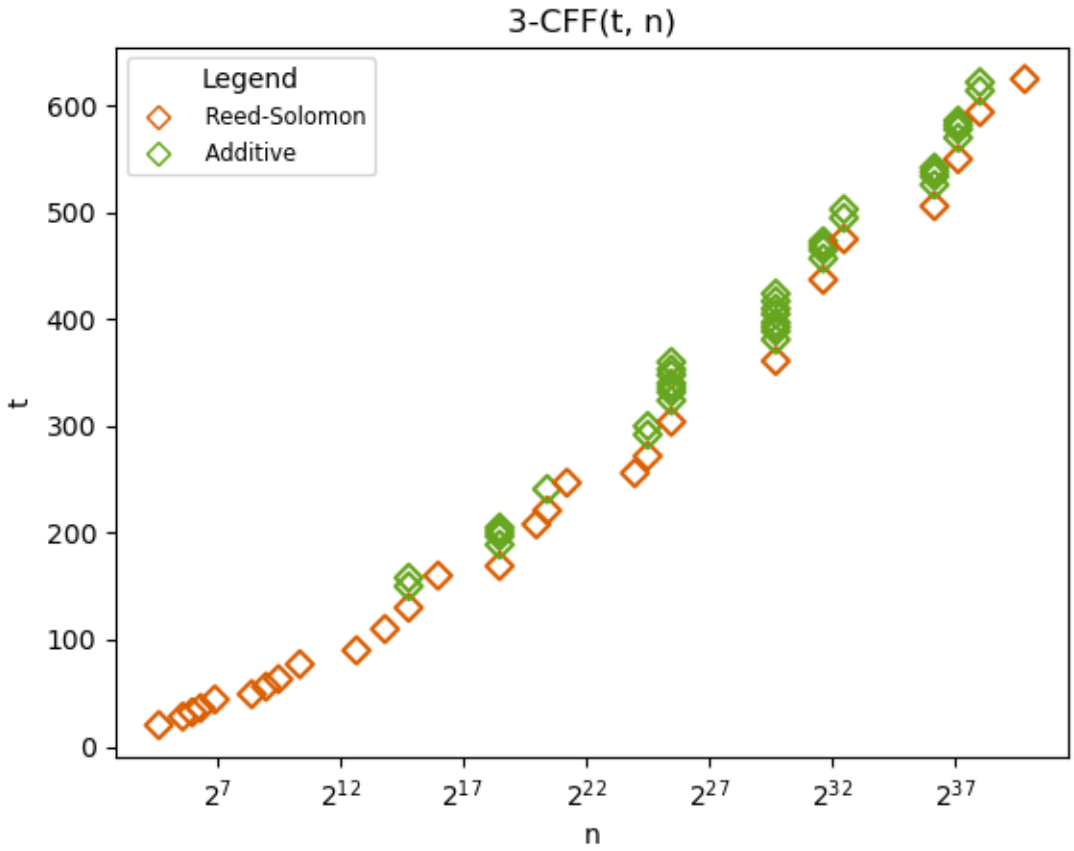


t	n	Source
9	12	STS(9)
12	16	RS(2^2,2,3)
13	26	STS(13)
15	35	STS(15)
19	57	STS(19)
20	64	RS(2^2,3,5)
21	70	STS(21)
25	125	RS(5^1,3,5)
31	155	STS(31)
33	176	STS(33)
35	343	RS(7^1,3,5)
40	512	RS(2^3,3,5)
45	729	RS(3^2,3,5)
49	2401	RS(7^1,4,7)
56	4096	RS(2^3,4,7)
63	6561	RS(3^2,4,7)
72	32768	RS(2^3,5,9)
81	59049	RS(3^2,5,9)
90	59061	Add(9,81)
92	65536	Dbl(72,18)
93	65538	Dbl(73,18)
94	65540	Dbl(74,18)
95	65542	Dbl(75,18)
96	65544	Dbl(76,18)
97	65546	Dbl(77,18)
98	65548	Dbl(78,18)
99	161051	RS(11^1,5,9)
108	161063	Add(9,99)
111	161067	Add(12,99)
112	161077	Add(13,99)
114	161086	Add(15,99)
117	371293	RS(13^1,5,9)
121	1771561	RS(11^1,6,11)
130	1771573	Add(9,121)
133	1771577	Add(12,121)
134	1771587	Add(13,121)
136	1771596	Add(15,121)
140	1771618	Add(19,121)
141	1771625	Add(20,121)
142	1771631	Add(21,121)
143	4826809	RS(13^1,6,11)
152	4826821	Add(9,143)
155	4826825	Add(12,143)
156	4826835	Add(13,143)
158	4826844	Add(15,143)

t	n	Source
162	4826866	Add(19,143)
163	4826873	Add(20,143)
164	4826879	Add(21,143)
168	4826934	Add(25,143)
169	62748517	RS(13^1,7,13)
178	62748529	Add(9,169)
181	62748533	Add(12,169)
182	62748543	Add(13,169)
184	62748552	Add(15,169)
188	62748574	Add(19,169)
189	62748581	Add(20,169)
190	62748587	Add(21,169)
194	62748642	Add(25,169)
199	125497034	Dbl(169,29)
200	125497036	Dbl(170,29)
201	125497038	Dbl(171,29)
202	125497040	Dbl(172,29)
203	125497042	Dbl(173,29)
204	125497044	Dbl(174,29)
205	125497046	Dbl(175,29)
206	125497048	Dbl(176,29)
207	125497050	Dbl(177,29)
208	268435456	RS(2^4,7,13)
217	268435468	Add(9,208)
220	268435472	Add(12,208)
221	410338673	RS(17^1,7,13)
230	410338685	Add(9,221)
233	410338689	Add(12,221)
234	410338699	Add(13,221)
236	410338708	Add(15,221)
240	4294967296	RS(2^4,8,15)
249	4294967308	Add(9,240)
252	4294967312	Add(12,240)
253	4294967322	Add(13,240)
255	6975757441	RS(17^1,8,15)
264	6975757453	Add(9,255)
267	6975757457	Add(12,255)
268	6975757467	Add(13,255)
270	6975757476	Add(15,255)
272	68719476736	RS(2^4,9,17)
281	68719476748	Add(9,272)
284	68719476752	Add(12,272)
285	68719476762	Add(13,272)
287	68719476771	Add(15,272)
289	118587876497	RS(17^1,9,17)

t	n	Source
298	118587876509	Add(9,289)
301	118587876513	Add(12,289)
302	118587876523	Add(13,289)
304	118587876532	Add(15,289)
308	118587876554	Add(19,289)
309	118587876561	Add(20,289)
310	118587876567	Add(21,289)
312	137438953472	Dbl(272,39)
313	137438953474	Dbl(273,39)
314	137438953476	Dbl(274,39)
315	137438953478	Dbl(275,39)
316	137438953480	Dbl(276,39)
317	137438953482	Dbl(277,39)
318	137438953484	Dbl(278,39)
319	137438953486	Dbl(279,39)
320	137438953488	Dbl(280,39)
321	137438953496	Dbl(281,39)
322	137438953498	Dbl(282,39)
323	322687697779	RS(19^1,9,17)
332	322687697791	Add(9,323)
335	322687697795	Add(12,323)
336	322687697805	Add(13,323)
338	322687697814	Add(15,323)
342	322687697836	Add(19,323)
343	322687697843	Add(20,323)
344	322687697849	Add(21,323)
348	322687697904	Add(25,323)
354	322687697934	Add(31,323)
356	322687697955	Add(33,323)
358	322687698122	Add(35,323)
361	6131066257801	RS(19^1,10,19)
370	6131066257813	Add(9,361)
373	6131066257817	Add(12,361)
374	6131066257827	Add(13,361)
376	6131066257836	Add(15,361)
380	6131066257858	Add(19,361)
381	6131066257865	Add(20,361)
382	6131066257871	Add(21,361)
386	6131066257926	Add(25,361)
392	6131066257956	Add(31,361)
394	6131066257977	Add(33,361)
396	6131066258144	Add(35,361)
401	6131066258313	Add(40,361)
406	6131066258530	Add(45,361)
409	10000000000000	Dbl(361,46)

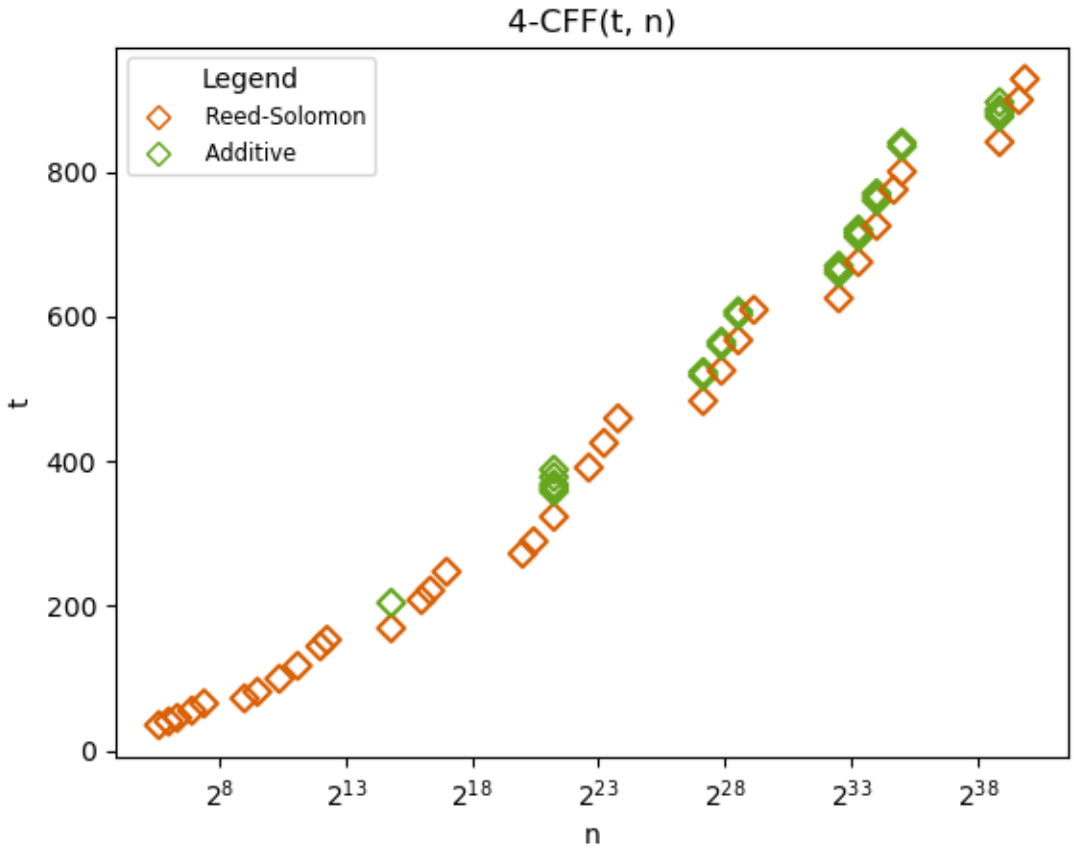
5.1.3 3-CFF table



t	n	Source
20	25	RS(5^1,2,4)
28	49	RS(7^1,2,4)
32	64	RS(2^3,2,4)
36	81	RS(3^2,2,4)
44	121	RS(11^1,2,4)
49	343	RS(7^1,3,7)
56	512	RS(2^3,3,7)
63	729	RS(3^2,3,7)
77	1331	RS(11^1,3,7)
90	6561	RS(3^2,4,10)
110	14641	RS(11^1,4,10)
130	28561	RS(13^1,4,10)
150	28586	Add(20,130)
158	28610	Add(28,130)
160	65536	RS(2^4,4,10)
169	371293	RS(13^1,5,13)
189	371318	Add(20,169)
197	371342	Add(28,169)
201	371357	Add(32,169)
205	371374	Add(36,169)
208	1048576	RS(2^4,5,13)
221	1419857	RS(17^1,5,13)
241	1419882	Add(20,221)
247	2476099	RS(19^1,5,13)
256	16777216	RS(2^4,6,16)
272	24137569	RS(17^1,6,16)
292	24137594	Add(20,272)
300	24137618	Add(28,272)
304	47045881	RS(19^1,6,16)
324	47045906	Add(20,304)
332	47045930	Add(28,304)
336	47045945	Add(32,304)
340	47045962	Add(36,304)
348	47046002	Add(44,304)
353	47046224	Add(49,304)
360	47046393	Add(56,304)
361	893871739	RS(19^1,7,19)
381	893871764	Add(20,361)
389	893871788	Add(28,361)
393	893871803	Add(32,361)
397	893871820	Add(36,361)
405	893871860	Add(44,361)
410	893872082	Add(49,361)
417	893872251	Add(56,361)
424	893872468	Add(63,361)

t	n	Source
437	3404825447	RS(23^1,7,19)
457	3404825472	Add(20,437)
465	3404825496	Add(28,437)
469	3404825511	Add(32,437)
473	3404825528	Add(36,437)
475	6103515625	RS(5^2,7,19)
495	6103515650	Add(20,475)
503	6103515674	Add(28,475)
506	78310985281	RS(23^1,8,22)
526	78310985306	Add(20,506)
534	78310985330	Add(28,506)
538	78310985345	Add(32,506)
542	78310985362	Add(36,506)
550	152587890625	RS(5^2,8,22)
570	152587890650	Add(20,550)
578	152587890674	Add(28,550)
582	152587890689	Add(32,550)
586	152587890706	Add(36,550)
594	282429536481	RS(3^3,8,22)
614	282429536506	Add(20,594)
622	282429536530	Add(28,594)
625	3814697265625	RS(5^2,9,25)
645	3814697265650	Add(20,625)
653	3814697265674	Add(28,625)
657	3814697265689	Add(32,625)
661	3814697265706	Add(36,625)
669	3814697265746	Add(44,625)
674	3814697265968	Add(49,625)
675	7625597484987	RS(3^3,9,25)
695	7625597485012	Add(20,675)
703	7625597485036	Add(28,675)
707	7625597485051	Add(32,675)
711	7625597485068	Add(36,675)
719	7625597485108	Add(44,675)
724	7625597485330	Add(49,675)
725	10000000000000	RS(29^1,9,25)

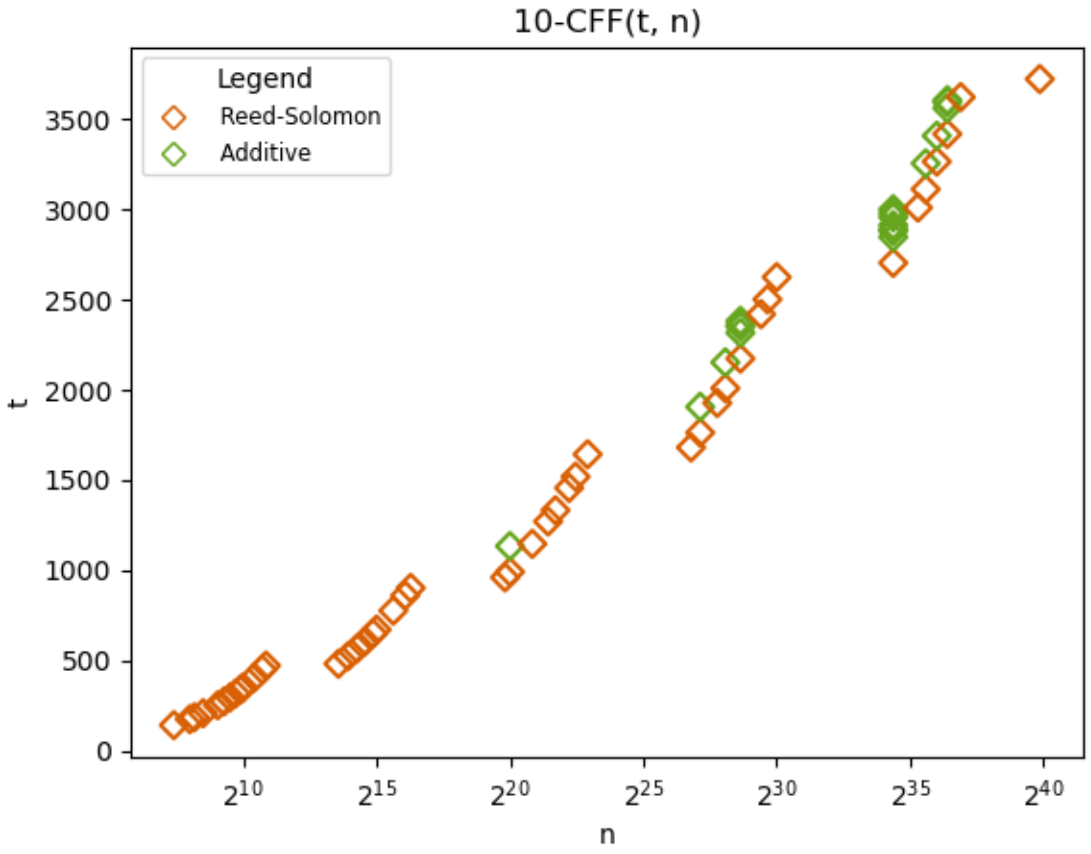
5.1.4 4-CFF table



t	n	Source
35	49	RS(7^1,2,5)
40	64	RS(2^3,2,5)
45	81	RS(3^2,2,5)
55	121	RS(11^1,2,5)
65	169	RS(13^1,2,5)
72	512	RS(2^3,3,9)
81	729	RS(3^2,3,9)
99	1331	RS(11^1,3,9)
117	2197	RS(13^1,3,9)
144	4096	RS(2^4,3,9)
153	4913	RS(17^1,3,9)
169	28561	RS(13^1,4,13)
204	28610	Add(35,169)
208	65536	RS(2^4,4,13)
221	83521	RS(17^1,4,13)
247	130321	RS(19^1,4,13)
272	1048576	RS(2^4,5,17)
289	1419857	RS(17^1,5,17)
323	2476099	RS(19^1,5,17)
358	2476148	Add(35,323)
363	2476163	Add(40,323)
368	2476180	Add(45,323)
378	2476220	Add(55,323)
388	2476268	Add(65,323)
391	6436343	RS(23^1,5,17)
425	9765625	RS(5^2,5,17)
459	14348907	RS(3^3,5,17)
483	148035889	RS(23^1,6,21)
518	148035938	Add(35,483)
523	148035953	Add(40,483)
525	244140625	RS(5^2,6,21)
560	244140674	Add(35,525)
565	244140689	Add(40,525)
567	387420489	RS(3^3,6,21)
602	387420538	Add(35,567)
607	387420553	Add(40,567)
609	594823321	RS(29^1,6,21)
625	6103515625	RS(5^2,7,25)
660	6103515674	Add(35,625)
665	6103515689	Add(40,625)
670	6103515706	Add(45,625)
675	10460353203	RS(3^3,7,25)
710	10460353252	Add(35,675)
715	10460353267	Add(40,675)
720	10460353284	Add(45,675)

t	n	Source
725	17249876309	RS(29^1,7,25)
760	17249876358	Add(35,725)
765	17249876373	Add(40,725)
770	17249876390	Add(45,725)
775	27512614111	RS(31^1,7,25)
800	34359738368	RS(2^5,7,25)
835	34359738417	Add(35,800)
840	34359738432	Add(40,800)
841	500246412961	RS(29^1,8,29)
876	500246413010	Add(35,841)
881	500246413025	Add(40,841)
886	500246413042	Add(45,841)
896	500246413082	Add(55,841)
899	852891037441	RS(31^1,8,29)
928	1099511627776	RS(2^5,8,29)
963	1099511627825	Add(35,928)
968	1099511627840	Add(40,928)
973	1099511627857	Add(45,928)
983	1099511627897	Add(55,928)
993	1099511627945	Add(65,928)
1000	1099511628288	Add(72,928)
1009	1099511628505	Add(81,928)
1027	1099511629107	Add(99,928)
1045	1099511629973	Add(117,928)
1056	10000000000000	RS(2^5,9,33)

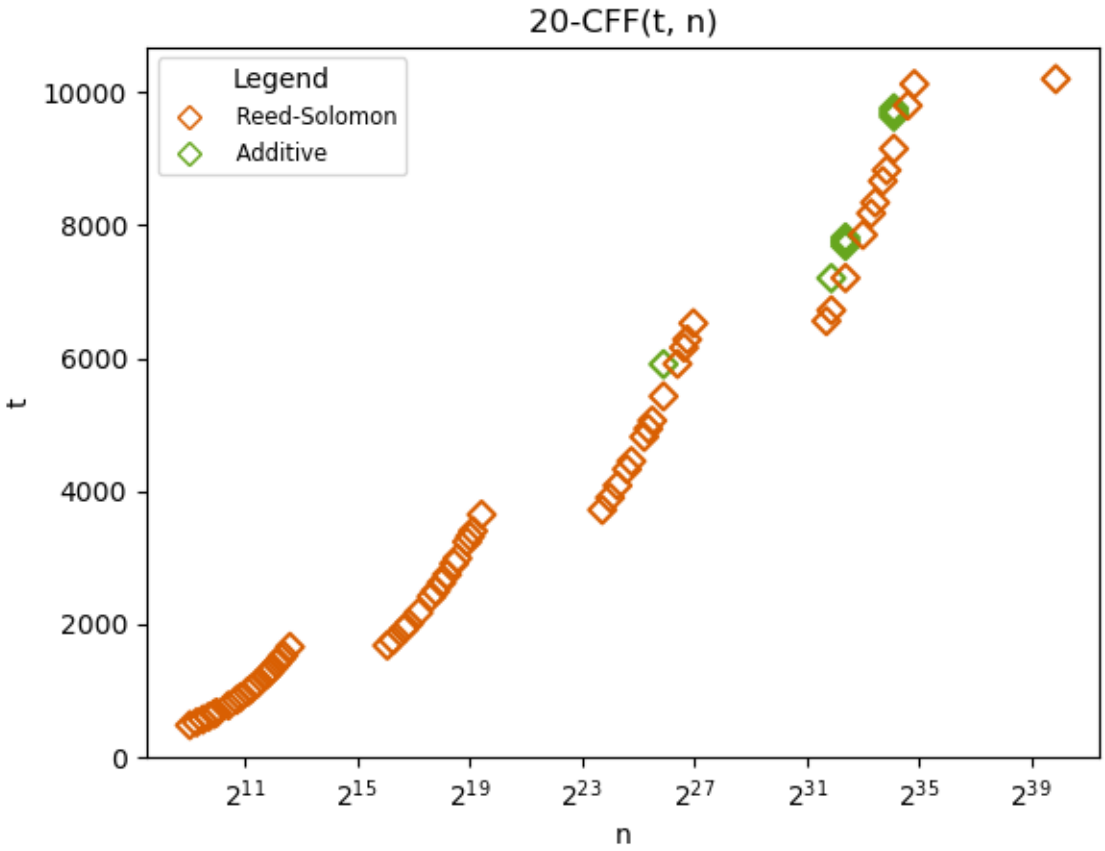
5.1.5 10-CFF table



t	n	Source
143	169	RS(13^1,2,11)
176	256	RS(2^4,2,11)
187	289	RS(17^1,2,11)
209	361	RS(19^1,2,11)
253	529	RS(23^1,2,11)
275	625	RS(5^2,2,11)
297	729	RS(3^3,2,11)
319	841	RS(29^1,2,11)
341	961	RS(31^1,2,11)
352	1024	RS(2^5,2,11)
407	1369	RS(37^1,2,11)
451	1681	RS(41^1,2,11)
473	1849	RS(43^1,2,11)
483	12167	RS(23^1,3,21)
525	15625	RS(5^2,3,21)
567	19683	RS(3^3,3,21)
609	24389	RS(29^1,3,21)
651	29791	RS(31^1,3,21)
672	32768	RS(2^5,3,21)
777	50653	RS(37^1,3,21)
861	68921	RS(41^1,3,21)
903	79507	RS(43^1,3,21)
961	923521	RS(31^1,4,31)
992	1048576	RS(2^5,4,31)
1135	1048745	Add(143,992)
1147	1874161	RS(37^1,4,31)
1271	2825761	RS(41^1,4,31)
1333	3418801	RS(43^1,4,31)
1457	4879681	RS(47^1,4,31)
1519	5764801	RS(7^2,4,31)
1643	7890481	RS(53^1,4,31)
1681	115856201	RS(41^1,5,41)
1763	147008443	RS(43^1,5,41)
1906	147008612	Add(143,1763)
1927	229345007	RS(47^1,5,41)
2009	282475249	RS(7^2,5,41)
2152	282475418	Add(143,2009)
2173	418195493	RS(53^1,5,41)
2316	418195662	Add(143,2173)
2349	418195749	Add(176,2173)
2360	418195782	Add(187,2173)
2382	418195854	Add(209,2173)
2419	714924299	RS(59^1,5,41)
2501	844596301	RS(61^1,5,41)
2624	1073741824	RS(2^6,5,41)

t	n	Source
2703	22164361129	RS(53^1,6,51)
2846	22164361298	Add(143,2703)
2879	22164361385	Add(176,2703)
2890	22164361418	Add(187,2703)
2912	22164361490	Add(209,2703)
2956	22164361658	Add(253,2703)
2978	22164361754	Add(275,2703)
3000	22164361858	Add(297,2703)
3009	42180533641	RS(59^1,6,51)
3111	51520374361	RS(61^1,6,51)
3254	51520374530	Add(143,3111)
3264	68719476736	RS(2^6,6,51)
3407	68719476905	Add(143,3264)
3417	90458382169	RS(67^1,6,51)
3560	90458382338	Add(143,3417)
3593	90458382425	Add(176,3417)
3604	90458382458	Add(187,3417)
3621	128100283921	RS(71^1,6,51)
3721	3142742836021	RS(61^1,7,61)
3864	3142742836190	Add(143,3721)
3897	3142742836277	Add(176,3721)
3904	4398046511104	RS(2^6,7,61)
4047	4398046511273	Add(143,3904)
4080	4398046511360	Add(176,3904)
4087	6060711605323	RS(67^1,7,61)
4230	6060711605492	Add(143,4087)
4263	6060711605579	Add(176,4087)
4274	6060711605612	Add(187,4087)
4296	6060711605684	Add(209,4087)
4331	9095120158391	RS(71^1,7,61)
4453	10000000000000	RS(73^1,7,61)

5.1.6 20-CFF table

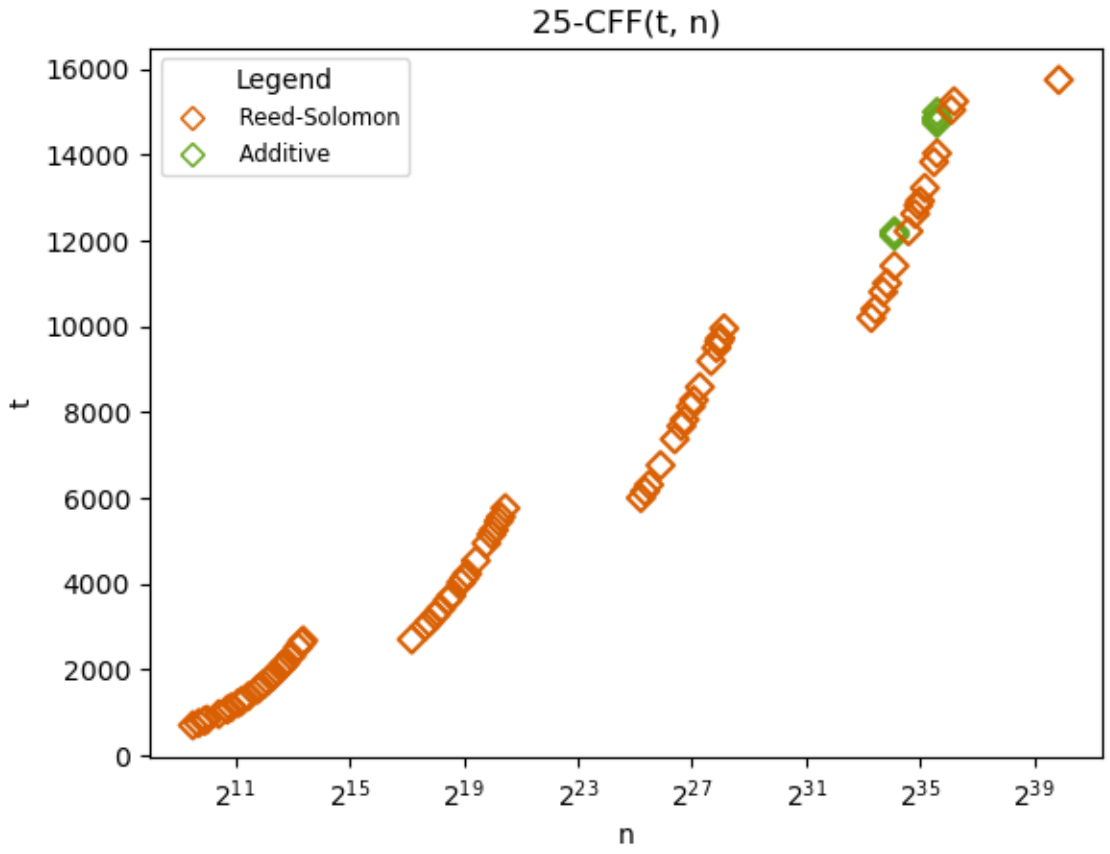


t	n	Source
483	529	RS(23^1,2,21)
525	625	RS(5^2,2,21)
567	729	RS(3^3,2,21)
609	841	RS(29^1,2,21)
651	961	RS(31^1,2,21)
672	1024	RS(2^5,2,21)
777	1369	RS(37^1,2,21)
861	1681	RS(41^1,2,21)
903	1849	RS(43^1,2,21)
987	2209	RS(47^1,2,21)
1029	2401	RS(7^2,2,21)
1113	2809	RS(53^1,2,21)
1239	3481	RS(59^1,2,21)
1281	3721	RS(61^1,2,21)
1344	4096	RS(2^6,2,21)
1407	4489	RS(67^1,2,21)
1491	5041	RS(71^1,2,21)
1533	5329	RS(73^1,2,21)
1659	6241	RS(79^1,2,21)
1681	68921	RS(41^1,3,41)
1763	79507	RS(43^1,3,41)
1927	103823	RS(47^1,3,41)
2009	117649	RS(7^2,3,41)
2173	148877	RS(53^1,3,41)
2419	205379	RS(59^1,3,41)
2501	226981	RS(61^1,3,41)
2624	262144	RS(2^6,3,41)
2747	300763	RS(67^1,3,41)
2911	357911	RS(71^1,3,41)
2993	389017	RS(73^1,3,41)
3239	493039	RS(79^1,3,41)
3321	531441	RS(3^4,3,41)
3403	571787	RS(83^1,3,41)
3649	704969	RS(89^1,3,41)
3721	13845841	RS(61^1,4,61)
3904	16777216	RS(2^6,4,61)
4087	20151121	RS(67^1,4,61)
4331	25411681	RS(71^1,4,61)
4453	28398241	RS(73^1,4,61)
4819	38950081	RS(79^1,4,61)
4941	43046721	RS(3^4,4,61)
5063	47458321	RS(83^1,4,61)
5429	62742241	RS(89^1,4,61)
5912	62742770	Add(483,5429)
5917	88529281	RS(97^1,4,61)

t	n	Source
6161	104060401	RS(101^1,4,61)
6283	112550881	RS(103^1,4,61)
6527	131079601	RS(107^1,4,61)
6561	3486784401	RS(3^4,5,81)
6723	3939040643	RS(83^1,5,81)
7206	3939041172	Add(483,6723)
7209	5584059449	RS(89^1,5,81)
7692	5584059978	Add(483,7209)
7734	5584060074	Add(525,7209)
7776	5584060178	Add(567,7209)
7818	5584060290	Add(609,7209)
7857	8587340257	RS(97^1,5,81)
8181	10510100501	RS(101^1,5,81)
8343	11592740743	RS(103^1,5,81)
8667	14025517307	RS(107^1,5,81)
8829	15386239549	RS(109^1,5,81)
9153	18424351793	RS(113^1,5,81)
9636	18424352322	Add(483,9153)
9678	18424352418	Add(525,9153)
9720	18424352522	Add(567,9153)
9762	18424352634	Add(609,9153)
9801	25937424601	RS(11^2,5,81)
10125	30517578125	RS(5^3,5,81)
10201	1061520150601	RS(101^1,6,101)
10403	1194052296529	RS(103^1,6,101)
10807	1500730351849	RS(107^1,6,101)
11009	1677100110841	RS(109^1,6,101)
11413	2081951752609	RS(113^1,6,101)
11896	2081951753138	Add(483,11413)
11938	2081951753234	Add(525,11413)
11980	2081951753338	Add(567,11413)
12022	2081951753450	Add(609,11413)
12064	2081951753570	Add(651,11413)
12085	2081951753633	Add(672,11413)
12190	2081951753978	Add(777,11413)
12221	3138428376721	RS(11^2,6,101)
12625	3814697265625	RS(5^3,6,101)
12827	4195872914689	RS(127^1,6,101)
12928	4398046511104	RS(2^7,6,101)
13231	5053913144281	RS(131^1,6,101)
13714	5053913144810	Add(483,13231)
13756	5053913144906	Add(525,13231)
13798	5053913145010	Add(567,13231)
13837	6611856250609	RS(137^1,6,101)
14039	7212549413161	RS(139^1,6,101)

t	n	Source
14522	7212549413690	Add(483,14039)
14564	7212549413786	Add(525,14039)
14606	7212549413890	Add(567,14039)
14641	10000000000000	RS(11^2,7,121)

5.1.7 25-CFF table



t	n	Source
702	729	RS(3^3,2,26)
754	841	RS(29^1,2,26)
806	961	RS(31^1,2,26)
832	1024	RS(2^5,2,26)
962	1369	RS(37^1,2,26)
1066	1681	RS(41^1,2,26)
1118	1849	RS(43^1,2,26)
1222	2209	RS(47^1,2,26)
1274	2401	RS(7^2,2,26)
1378	2809	RS(53^1,2,26)
1534	3481	RS(59^1,2,26)
1586	3721	RS(61^1,2,26)
1664	4096	RS(2^6,2,26)
1742	4489	RS(67^1,2,26)
1846	5041	RS(71^1,2,26)
1898	5329	RS(73^1,2,26)
2054	6241	RS(79^1,2,26)
2106	6561	RS(3^4,2,26)
2158	6889	RS(83^1,2,26)
2314	7921	RS(89^1,2,26)
2522	9409	RS(97^1,2,26)
2626	10201	RS(101^1,2,26)
2678	10609	RS(103^1,2,26)
2703	148877	RS(53^1,3,51)
3009	205379	RS(59^1,3,51)
3111	226981	RS(61^1,3,51)
3264	262144	RS(2^6,3,51)
3417	300763	RS(67^1,3,51)
3621	357911	RS(71^1,3,51)
3723	389017	RS(73^1,3,51)
4029	493039	RS(79^1,3,51)
4131	531441	RS(3^4,3,51)
4233	571787	RS(83^1,3,51)
4539	704969	RS(89^1,3,51)
4947	912673	RS(97^1,3,51)
5151	1030301	RS(101^1,3,51)
5253	1092727	RS(103^1,3,51)
5457	1225043	RS(107^1,3,51)
5559	1295029	RS(109^1,3,51)
5763	1442897	RS(113^1,3,51)
6004	38950081	RS(79^1,4,76)
6156	43046721	RS(3^4,4,76)
6308	47458321	RS(83^1,4,76)
6764	62742241	RS(89^1,4,76)
7372	88529281	RS(97^1,4,76)

t	n	Source
7676	104060401	RS(101^1,4,76)
7828	112550881	RS(103^1,4,76)
8132	131079601	RS(107^1,4,76)
8284	141158161	RS(109^1,4,76)
8588	163047361	RS(113^1,4,76)
9196	214358881	RS(11^2,4,76)
9500	244140625	RS(5^3,4,76)
9652	260144641	RS(127^1,4,76)
9728	268435456	RS(2^7,4,76)
9956	294499921	RS(131^1,4,76)
10201	10510100501	RS(101^1,5,101)
10403	11592740743	RS(103^1,5,101)
10807	14025517307	RS(107^1,5,101)
11009	15386239549	RS(109^1,5,101)
11413	18424351793	RS(113^1,5,101)
12115	18424352522	Add(702,11413)
12167	18424352634	Add(754,11413)
12219	18424352754	Add(806,11413)
12221	25937424601	RS(11^2,5,101)
12625	30517578125	RS(5^3,5,101)
12827	33038369407	RS(127^1,5,101)
12928	34359738368	RS(2^7,5,101)
13231	38579489651	RS(131^1,5,101)
13837	48261724457	RS(137^1,5,101)
14039	51888844699	RS(139^1,5,101)
14741	51888845428	Add(702,14039)
14793	51888845540	Add(754,14039)
14845	51888845660	Add(806,14039)
14871	51888845723	Add(832,14039)
15001	51888846068	Add(962,14039)
15049	73439775749	RS(149^1,5,101)
15251	78502725751	RS(151^1,5,101)
15750	3814697265625	RS(5^3,6,126)
16002	4195872914689	RS(127^1,6,126)
16128	4398046511104	RS(2^7,6,126)
16506	5053913144281	RS(131^1,6,126)
17208	5053913145010	Add(702,16506)
17260	5053913145122	Add(754,16506)
17262	6611856250609	RS(137^1,6,126)
17514	7212549413161	RS(139^1,6,126)
18216	7212549413890	Add(702,17514)
18268	7212549414002	Add(754,17514)
18320	7212549414122	Add(806,17514)
18346	7212549414185	Add(832,17514)
18476	7212549414530	Add(962,17514)

t	n	Source
18580	7212549414842	Add(1066,17514)
18632	7212549415010	Add(1118,17514)
18736	7212549415370	Add(1222,17514)
18774	10000000000000	RS(149^1,6,126)

5.2 Observations

In our tables, we observe that Reed-Solomon codes are very good compared to other constructions. We were surprised to find that the CFFs from Porat and Rothschild's linear code construction are never better than CFFs Reed-Solomon codes for the values of t in our tables. The construction from Porat and Rothschild meets the Gilbert-Varshamov bound and matches the best-known asymptotic bound of $O(d^2 \log n)$, whereas Reed-Solomon codes do not. We have found that when creating much larger tables, using double precision floating point numbers to allow very large n, at the cost of loss of precision, CFFs from Porat and Rothschild start to become better than Reed-Solomon codes around $n = 10^{117}$ for $d = 2$. In Section 5.3 we discuss a more thorough comparison.

The recursive Optimized Kronecker product and Kronecker product constructions do not appear in any of the tables with n up to ten trillion. However, we observe that the additive construction, doubling construction, and the extension-by-one construction are very valuable to fill in gaps between Reed-Solomon codes.

In the case of $d = 2$, we observe that having a small amount of very good small CFFs is very valuable, because these small CFFs from binary constant-weight codes are used repeatedly in recursive construction, giving us new bounds for many more cases than just the direct construction from binary constant-weight codes. We do not have similar small good CFFs for $d > 2$, but they would provide the same benefit as they have for $d = 2$.

For $d > 2$, the d-CFF tables with n up to 10 trillion only includes the Reed-Solomon, additive, and extension by one constructions. The Kronecker product and optimized Kronecker product constructions do not seem to give good CFFs. Porat and Rothschild's construction is only effective for $n > 2^{370}$ as shown in Section 5.3.

5.3 Comparing Reed-Solomon to Porat and Rothschild

The CFFs constructed using Porat and Rothschild's linear error correcting codes gives the theoretical best upper bound on $t(d, n)$. This means that eventually, Porat and Rothschild's codes would start appearing in our tables if we went to large enough values of t and n. In this section, we create tables with a slightly different strategy, with the goal of only comparing CFFs from Reed-Solomon codes and Porat and Rothschild's codes.

We use the same two algorithms from Chapter 4, Algorithms 4.4 and 4.5, but instead of calling updateTable (Algorithm 4.1), we append each found CFF to an array. Then, on that array of CFFs we remove the CFFs that are dominated by another CFF. That

is if a d-CFF(t_1, n_1) and a d-CFF(t_2, n_2) are found with $n_1 < n_2$ and $t_1 \geq t_2$, then the d-CFF(t_1, n_1) can be removed. This is done using Algorithm 5.1:

Algorithm 5.1 FlagDominatedCFFs(CFFarray)

Input: An array of CFF parameters
Output: CFF parameters that are dominated are removed.

```
1: sort CFFarray by n ascending, then by t descending if n is equal
2: lastBestT ← CFFarray[CFFarray.length −1].t
3: for i ← CFFarray.length−2 ... 0 do
4:     if CFFarray[i].t ≥ lastBestT then
5:         Flag CFFarray[i] as dominated
6:     else
7:         lastBestT ← CFFarray[i].t
8:     end if
9: end for
10: Remove the CFFs that were flagged as dominated
11: return CFFarray
```

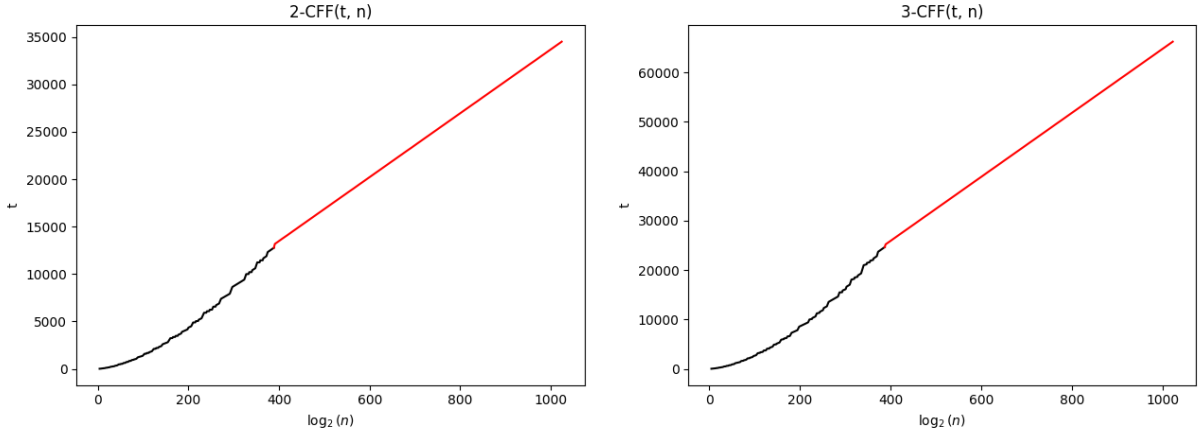
Since the algorithm first sorts the array by n ascending, then by t descending if the n is equal between some CFFs, we only need to iterate over the array in reverse order and keep the CFFs that have a lower t than the previous t that was kept. For example, we will look at this case with a collection of CFFs in Table 5.3, where some are dominated. The CFFs are already sorted as needed.

t	n	Dominated?	Source
10	5	yes	RS$(5^1, 1, 2)$
119	7	yes	PR$(7^1, 1, 37)$
7	7	no	ID(7)
16	16	yes	RS$(2^2, 2, 4)$
12	16	no	RS$(2^2, 2, 3)$

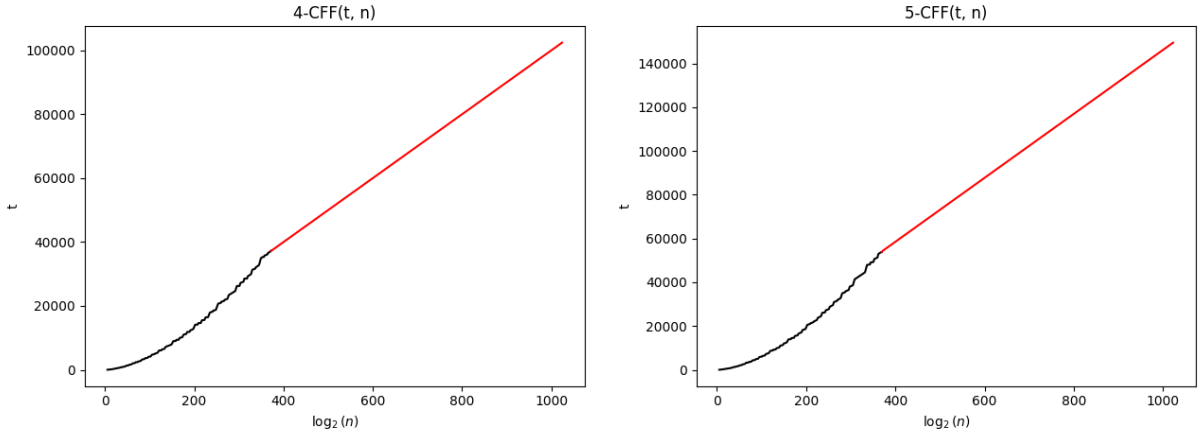
Table 5.3: An example of CFFs dominating others

In this example with d=2, we can see that a 2-CFF$(16, 16)$ is dominated by a 2-CFF$(12, 16)$. Additionally, the 2-CFF$(10, 5)$ and 2-CFF$(119, 7)$ are dominated by the 2-CFF$(7, 7)$.

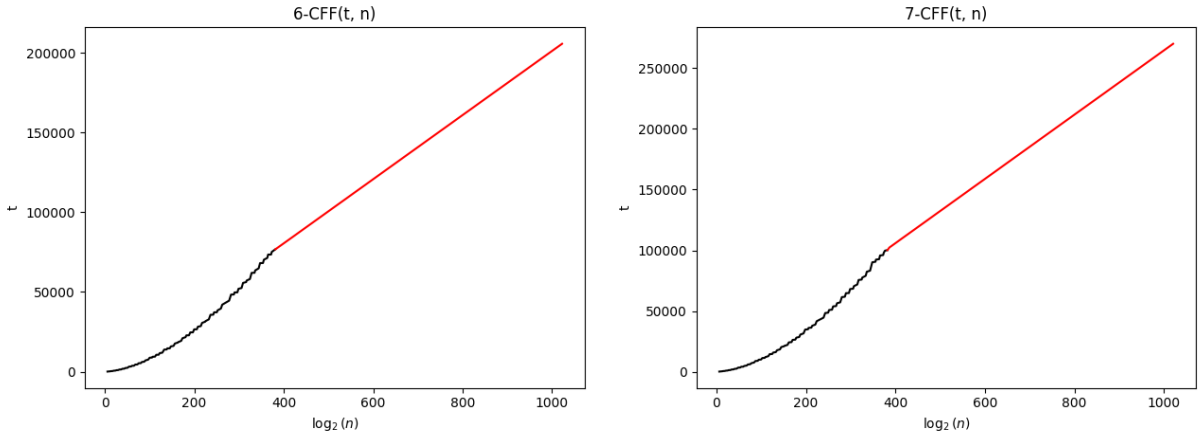
The following graphs show when CFFs from Porat and Rothschild begin dominating CFFs from Reed-Solomon codes. The red segment of the lines indicates when the CFFs that are being kept by Algorithm 5.1 are constructed with codes from Porat and Rothschild's construction, and the black segment of the lines are CFFs constructed from Reed-Solomon codes.


2-CFF(t, n)
t
log$_2$ (n)

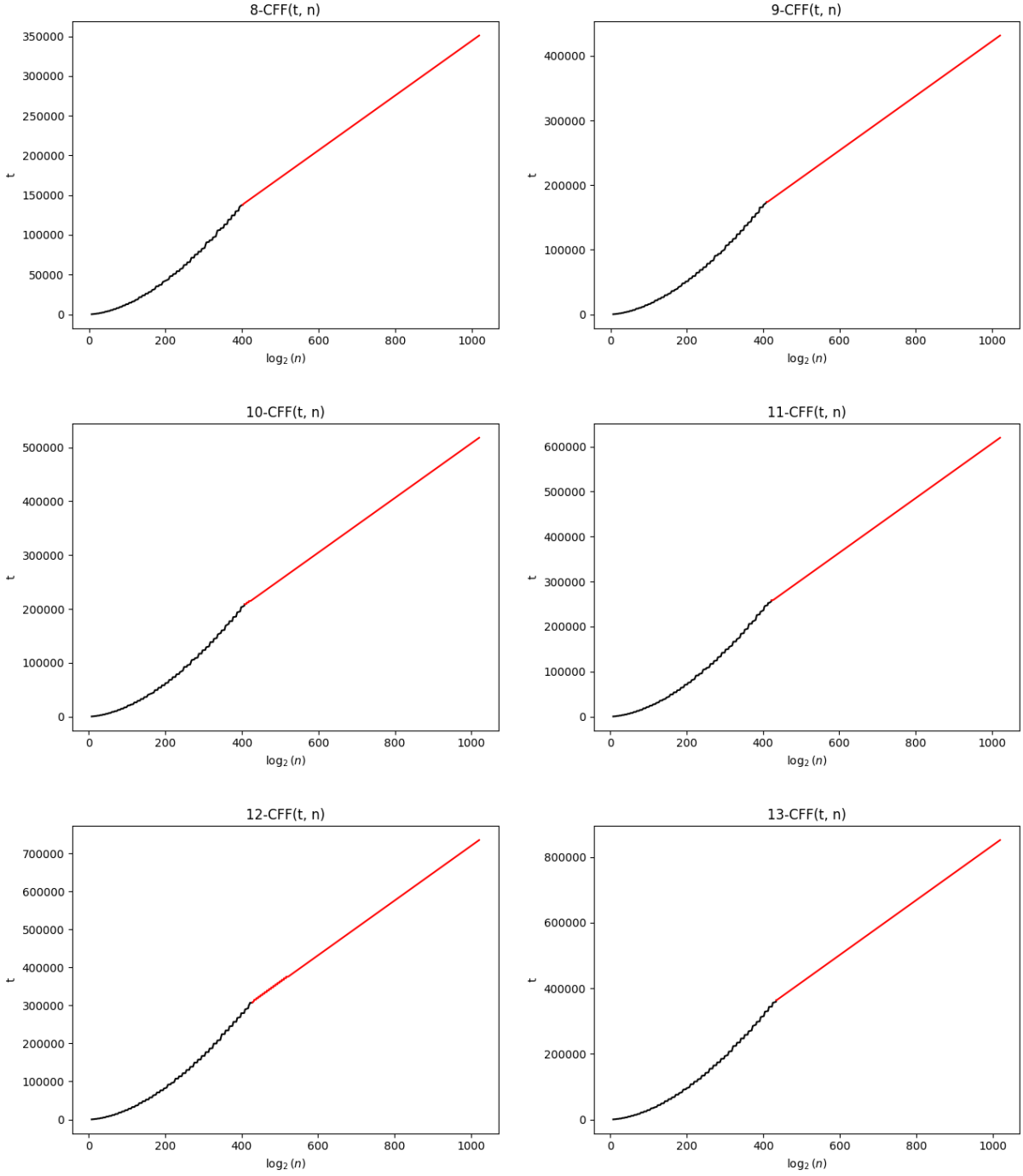
3-CFF(t, n)
t
log$_2$ (n)

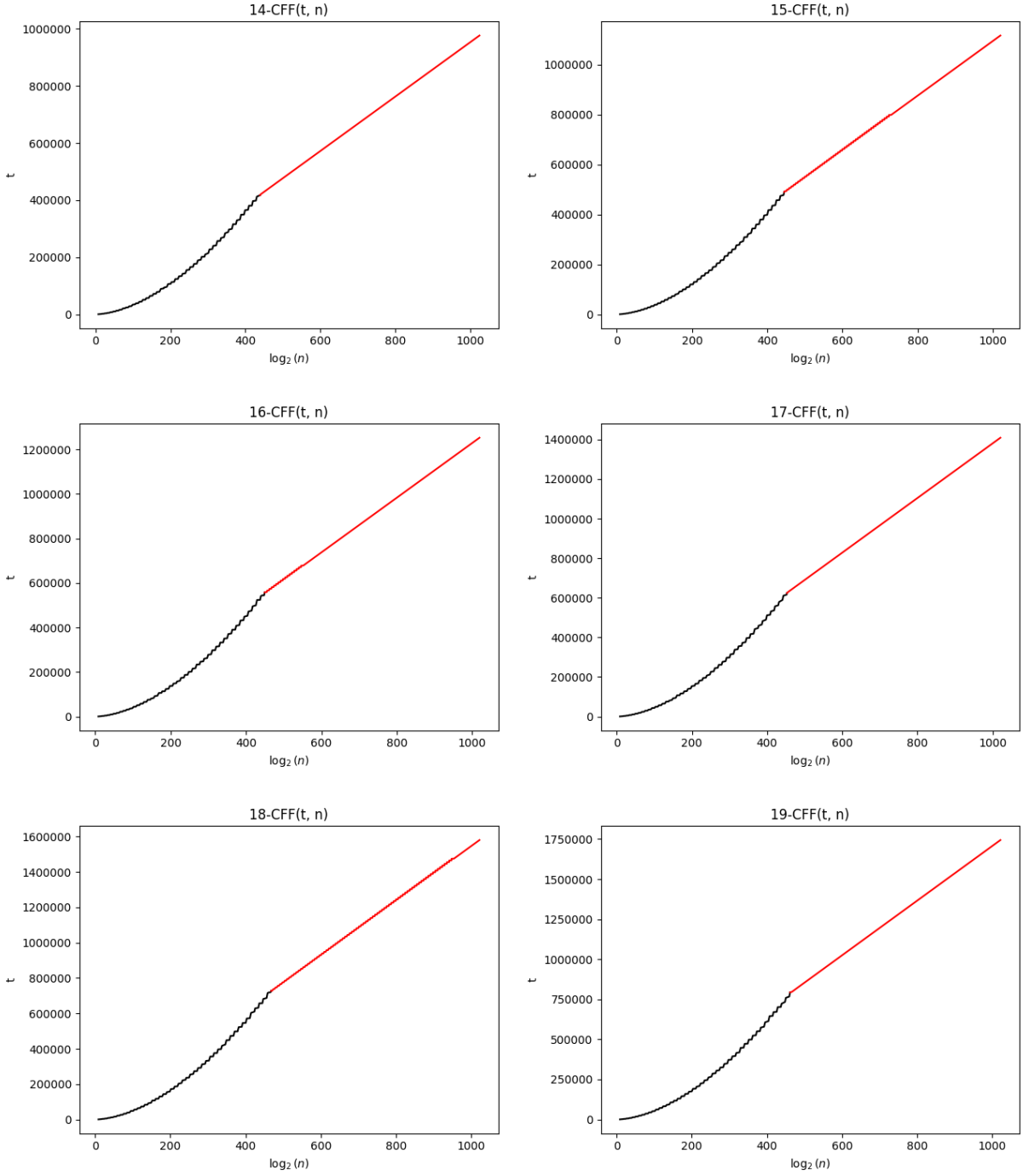

4-CFF(t, n)
t
log$_2$ (n)

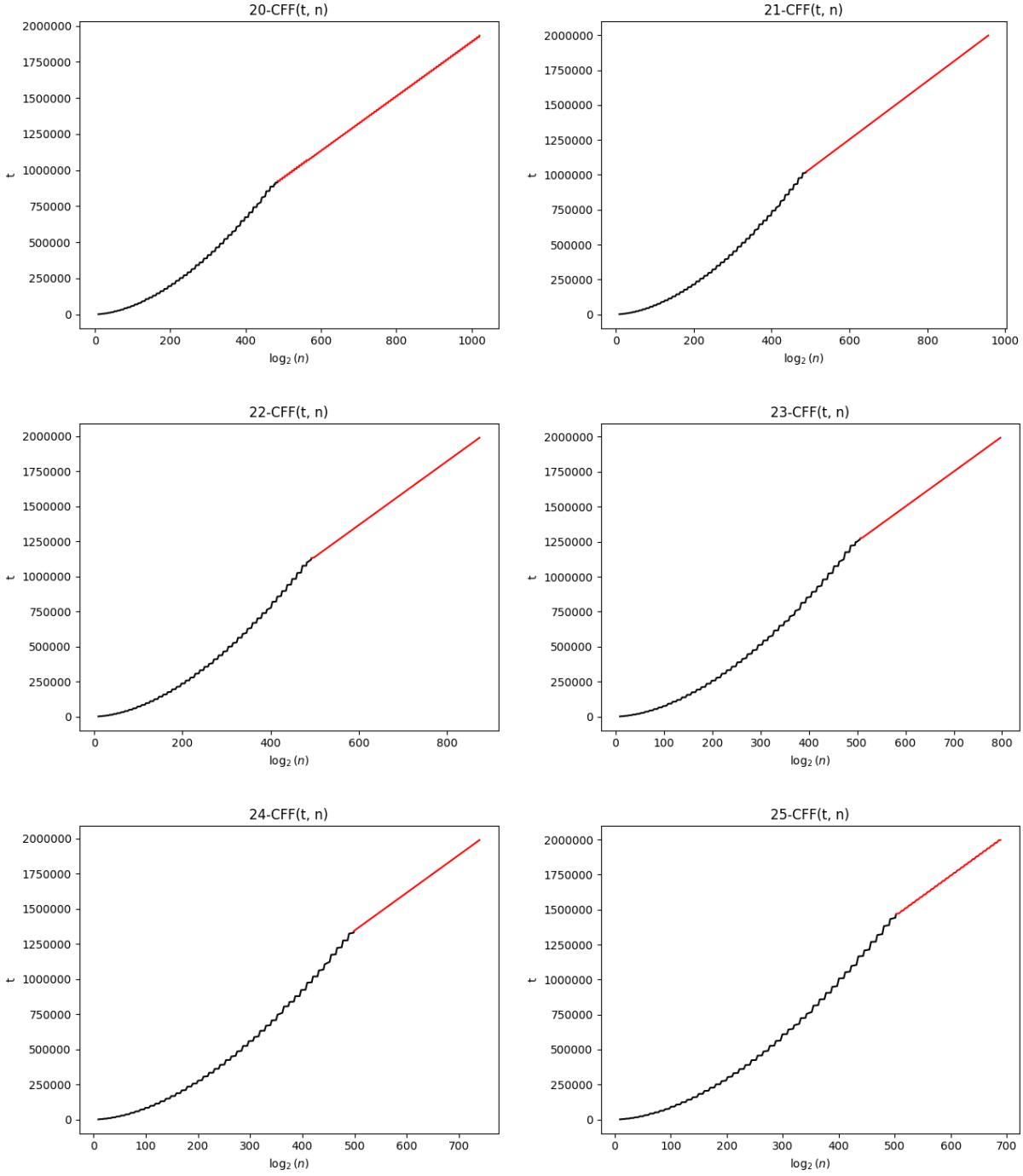
5-CFF(t, n)
t
log$_2$ (n)


6-CFF(t, n)
t
log$_2$ (n)

7-CFF(t, n)
t
log$_2$ (n)


8-CFF(t, n)
9-CFF(t, n)
10-CFF(t, n)
11-CFF(t, n)
12-CFF(t, n)
13-CFF(t, n)
t
log2 (n)


14-CFF(t, n)
15-CFF(t, n)
16-CFF(t, n)
17-CFF(t, n)
18-CFF(t, n)
19-CFF(t, n)
t
log$_2$ (n)



When comparing the first t and n that the line changes to red, meaning that only Porat and Rothschild construction is being used, we have the graphs in Figures 5.2 and 5.3.

Overall, while Porat and Rothschild's construction has the best asymptotic bound, this construction is only better in practice once n is very large; for $2 \leq d \leq 25$, such an n_0 only occurs around $2^{370} \leq n_0 \leq 2^{500}$ (Figure 5.2).

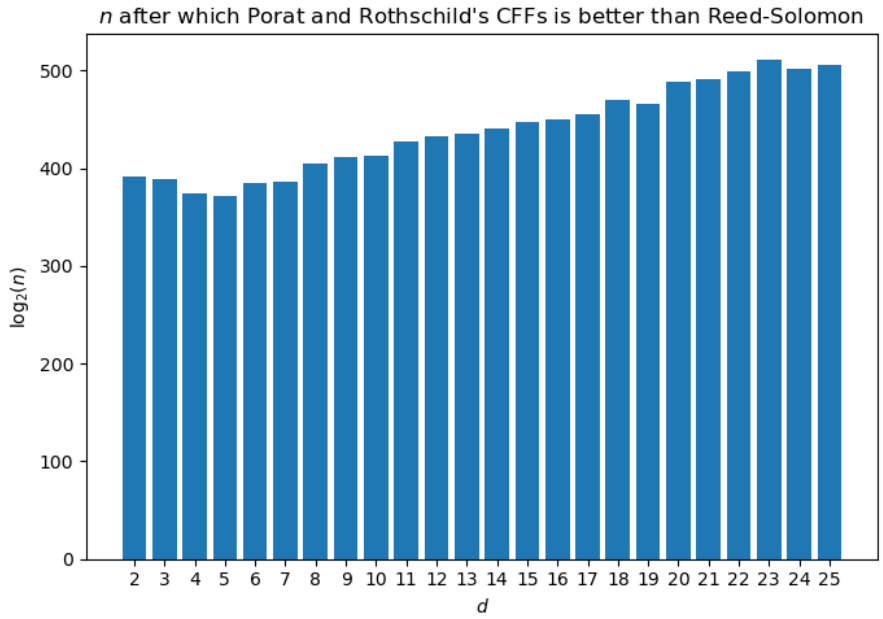


Figure 5.2: n after which Porat and Rothschild's CFFs is better than Reed-Solomon

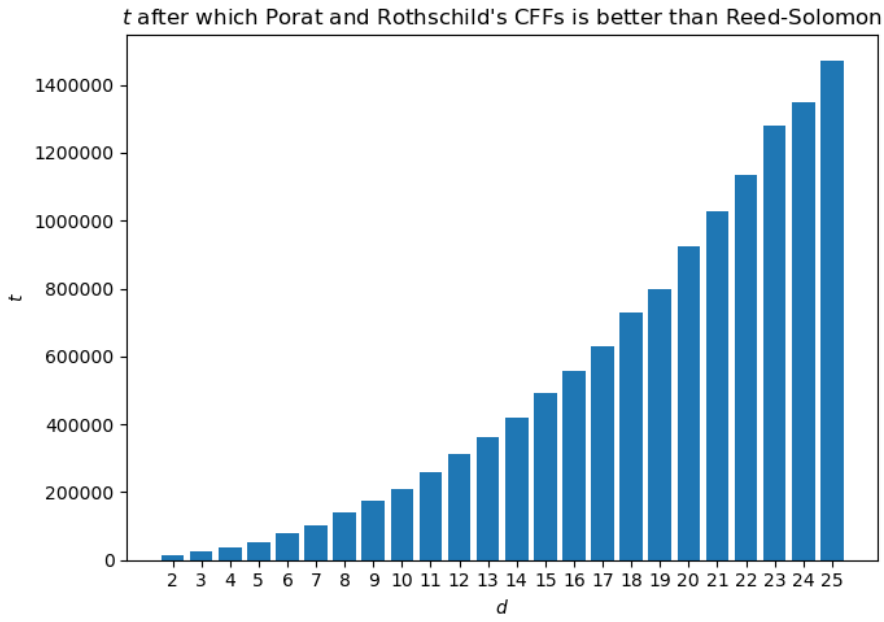


Figure 5.3: t after which Porat and Rothschild's CFFs is better than Reed-Solomon

Remark 5.1. *Porat and Rothschild codes are only used in our tables when the hypothesis of Theorem 3.8 are satisfied, that is, taking* $m = \left\lfloor \frac{k}{1-H_q(\delta)} \right\rfloor$ *in line 8 of Algorithm 4.5. This ensures* $E[goal(\mathcal{G})] < 1$*, which implies the algorithm will be successful with no bad events occurring. However, it is possible initially to have* $E[goal(\mathcal{G})] \geq 1$ *and still end up with a code with zero bad events. For example, taking* $k = 2$*,* $q = 3$*,* $m = 3$ *can be used in Algorithm 3.5 to construct a* 3×2 *generator matrix in* $\mathbb{F}_3$ *with* $\delta = \frac{2}{3}$ *producing a 2-CFF(9, 9). At the beginning of the algorithm* $E[goal(\mathcal{G})] = \frac{70}{27}$*, but it reduces until at the last step when* $E[goal(\mathcal{G})|ST_{(3,2)+1}] = 0$*. Therefore, it is possible that CFFs can be generated using Porat and Rothschild's algorithm when* $k < (1 - H_q(\delta))m$*, but these codes have not been tested and are not reported in our tables.*

Chapter 6

Conclusion

In this thesis, we studied constructions of cover-free families, and created tables of best known CFFs from these constructions. Our goals were to explore constructions of cover-free families, and find lower bounds for $n(d, t)$ and upper bounds for $t(d, n)$.

In Chapter 2, we studied direct and recursive constructions of cover-free families. These direct constructions include constructions from Sperner systems, packing designs such as Steiner triple systems, and error correcting codes including Reed-Solomon codes and binary constant-weight codes. Our selection of recursive constructions include the Kronecker product, the Optimized Kronecker product, a double construction for $d = 2$, an additive construction, and an extension-by-one construction. We implemented each construction that was discussed in Chapter 2.

In Chapter 3, we discussed the probabilistic method and an application of this used to construct linear error correcting codes meeting the Gilbert-Varshamov bound. We provided detailed proofs and pseudocode for the probabilistic method's use in [23] that outlines a CFF construction with best-known asymptotic upper bound for $t(d, n)$ in $O(d^2 \log n)$.

In Chapter 4, we presented algorithms to create tables of best-known bounds on $t(d, n)$ and $n(d, t)$ using our selected constructions by iterating over all possible combinations of parameters. We designed and implemented an algorithm to create best-known tables of CFFs using these constructions. Additionally, we created and implemented algorithms to construct a CFF given a pair (d, n) or (d, t) that allows the construction of any CFF in our tables.

Finally in Chapter 5, we presented and discussed these tables, showing that our best-known bounds on $t(d, n)$ and $n(d, t)$ from our selected constructions do not use the Kronecker product or Optimized Kronecker product constructions. Next, we compared the construction from Reed-Solomon codes to the construction from Porat and Rothschild's codes. From this analysis, we learned that Porat and Rothschild's construction only provides best-known bounds for our selected constructions for $t(d, n)$ for $2^{370} \leq n \leq 2^{500}$ with $2 \leq d \leq 25$.

6.1 Future work

Implementing either exhaustive or heuristic search to find good small CFFs would likely improve our tables significantly. As seen in the $d = 2$ table, it is very valuable to have some amount of good small CFFs, so that these small CFFs can be used in recursive constructions to fill more values in the table. This would be particularly valuable for tables with $d > 2$, since we have fewer constructions for $d > 2$ compared to $d = 2$.

Additionally, new constructions for CFFs could be added to our tables, such as different types of error correcting codes that would be used to construct CFFs. Wei notes in [28] that some algebraic codes can be a good source of codes to construct CFFs with.

Furthermore, we could use combinatorial designs from various sources to construct CFFs. For example, [4] gives tables of best-known packing designs. We could use this data to updates our CFF tables, and find better bounds on $n(d, t)$ and $t(d, n)$.

Generator matrices constructed from Porat and Rothschild sometimes contain repeated rows. So, another area of future work is to create many of the generator matrices using the Porat and Rothschild construction. It is possible to keep only one of the repeated rows to reduce the codeword size. Since codeword size only affects the t of a CFF when using the construction from codes, removing repeated rows would give a CFF with fewer rows. By doing this, we may find that CFFs from Porat and Rothschild's codes construction start becoming better than Reed-Solomon codes sooner than our experiments.

Additionally, we can try running Porat and Rothschild's algorithm with parameters that do not meet the hypothesis of Theorem 3.3. For example, the code and generator matrix in Figure 3.2 has $m < \left\lceil \frac{k}{1-H_q(\delta)} \right\rceil$, but it still gives a valid code. See Remark 5.1.

Another area of future work could be to find a construction for CFFs directly using the probabilistic method. Porat and Rothschild's construction makes a code, then uses this to construct a CFF. It may be possible to use a similar idea of a goal function, where we set each cell in a CFF's incidence matrix either to a zero or one, depending on which would make it more or less likely that more weight-1 0-1 tuples of size $d + 1$ are present in the CFF after fixing a position to zero or one. We found it might be helpful to have a constant number of ones in each row of the CFF to make writing a proof more simple, but we did not find one ourselves. An alternative approach could be possible where each decision of the probabilistic algorithm is which row to place each weight-1 0-1 $(d+1)$-tuple that must be present in a CFF, instead of setting each cell to 0 or 1 individually.

References

[1] Martin Aigner and Gnter M. Ziegler. *Proofs from THE BOOK*. Springer, Berlin, Heidelberg, sixth edition. edition, 2018 - 2018.

[2] Noga Alon and Joel H. Spencer. *The probabilistic method*. Wiley series in discrete mathematics and optimization. John Wiley I& Sons, Inc., Hoboken, New Jersey, fourth edition. edition, 2016.

[3] Andries E. Brouwer. Bounds for binary constant weight codes, 2025. `https://aeb.win.tue.nl/codes/Andw.html`. Accessed: 2025-02-23.

[4] Charles J. Colbourn and Jeffrey H. Dinitz. *Handbook of combinatorial designs*. Discrete mathematics and its applications. CRC/Taylor & Francis, Boca Raton, Fla, 2nd ed. edition, 2007.

[5] Ronald Cramer, Goichiro Hanaoka, Dennis Hofheinz, Hideki Imai, Eike Kiltz, Rafael Pass, Abhi Shelat, and Vinod Vaikuntanathan. Bounded cca2-secure encryption. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 4833, pages 502–518, 2007.

[6] Matthew Demczyk. CFF tables: Best parameters for d-cover-free families, 2025. `https://matthewdemczyk.github.io/CFFtables/index.html`.

[7] Matthew Demczyk. CFFtablesCode: C implementation of discussed algorithms. `https://github.com/matthewdemczyk/CFFtablesCode`, 2025. GitHub repository.

[8] Yvo Desmedt, Rei Safavi-Naini, Huaxiong Wang, Lynn Batten, Chris Charnes, and Josef Pieprzyk. Broadcast anti-jamming systems. *Computer networks (Amsterdam, Netherlands : 1999)*, 35(2):223–236, 2001.

[9] Dingzhu Du and Frank Hwang. *Combinatorial group testing and its applications*. Series on applied mathematics ; v. 3. World Scientific, Singapore ;, 1993.

[10] Dingzhu Du and Frank Hwang. *Pooling designs and nonadaptive group testing important tools for DNA sequencing*. Series on applied mathematics ; v. 18. World Scientific, New Jersey, 2006.

[11] Paolo D'Arco, Douglas R. Stinson, and Marc Joye. Fault tolerant and distributed broadcast encryption. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 2612 of *Lecture Notes in Computer Science*, pages 263–280. Springer Berlin / Heidelberg, Germany, 2003.

[12] Eli Gafni, Jessica Staddon, Yiqun Lisa Yin, and Michael Wiener. Efficient methods for integrating traceability and broadcast encryption. In *Advances in Cryptology - CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 372–387. Springer Berlin / Heidelberg, Germany, 1999.

[13] Gunnar Hartung, Björn Kaidel, Alexander Koch, Jessica Koch, Andy Rupp, Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang. Fault-tolerant aggregate signatures. In *Lecture Notes in Computer Science*, volume 9614, pages 331–356, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[14] A. S. Hedayat, N. J. A. Sloane, and John. Stufken. *Orthogonal Arrays : Theory and Applications*. Springer Series in Statistics. Springer New York, New York, NY, 1999.

[15] Thaís Bardini Idalino and Lucia Moura. Nested cover-free families for unbounded fault-tolerant aggregate signatures. *Theoretical computer science*, 854:116–130, 2021.

[16] Thaís Bardini Idalino and Lucia Moura. Nested cover-free families for unbounded fault-tolerant aggregate signatures. *Theoretical computer science*, 854:116–130, 2021.

[17] Thaís Bardini Idalino and Lucia Moura. A survey of cover-free families: Constructions, applications and generalizations. In *New Advances in Designs, Codes and Cryptography*, volume 86 of *Fields Institute Communications*, pages 195–239. Springer International Publishing AG, Switzerland, 2024.

[18] Donald L. Kreher and Douglas R. Stinson. *Combinatorial algorithms : generation, enumeration, and search*. CRC Press series on discrete mathematics and its applications. CRC Press, Boca Raton, 1st. edition, 2020 - 1999.

[19] P. C. Li, G. H. J. van Rees, and R. Wei. Constructions of 2-cover-free families and related separating hash families. *Journal of combinatorial designs*, 14(6):423–440, 2006.

[20] Dongxia Luo. Modification-tolerant signature schemes using combinatorial group testing: Theory, algorithms, and implementation. Master's thesis, University of Ottawa, Ottawa, Ontario, Canada, 2024.

[21] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error correcting codes*. North-Holland mathematical library ; v. 016. North-Holland Pub. Co., Amsterdam, second printing. edition, 1977.

[22] Jarosław Pastuszak, Josef Pieprzyk, Jennifer Seberry, Eiji Okamoto, Bimal Kumar Roy, Bimal Roy, and Eiji Okamoto. Codes identifying bad signatures in batches. In

Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), volume 1977 of *Lecture Notes in Computer Science*, pages 143–154. Springer Berlin / Heidelberg, Germany, 2000.

[23] E. Porat and A. Rothschild. Explicit nonadaptive combinatorial group testing schemes. *IEEE transactions on information theory*, 57(12):7982–7989, 2011.

[24] E. Sperner. Ein satz über untermengen einer endlichen menge. *Mathematische Zeitschrift, 27*, page 544–548, 1928.

[25] Douglas R. Stinson. *Combinatorial designs : constructions and analysis*. Springer, New York, 2004.

[26] The FLINT team. *FLINT: Fast Library for Number Theory*, 2025. Version 3.3.1, `https://flintlib.org`.

[27] Ronald E. Walpole. *Essentials of probability & statistics for engineers & scientists*. Pearson, Boston, 2013.

[28] Ruizhong Wei. On cover-free families. Technical report, Lakehead University, 2006. arxiv: `https://arxiv.org/abs/2303.17524`.

[29] Gregory M. Zaverucha, Douglas R. Stinson, and Kaoru Kurosawa. Group testing and batch verification. In *Information Theoretic Security*, volume 5973 of *Lecture Notes in Computer Science*, pages 140–157. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

APPENDICES

Appendix A

CFF Verification Algorithm

Algorithm A.1 is a verification algorithm that tests if a 0-1 incidence matrix A is a cover-free family or not. This was used to verify possible mistakes when implementing constructions that have been proven to yield CFFs. The k-subset lexicographic successor algorithm from Algorithm 2.1 is used to iterate over every $(d+1)$-subset of columns.

The running time of Algorithm A.1 is $O(td\binom{n}{d})$, which is $O(tdn^d)$. It runs in polynomial time in t and n, if d is a constant. This is an improvement over the naïve approach because of the break statements on lines 11 and 20.

Algorithm A.1 VerifyCFF(A, d, t, n)

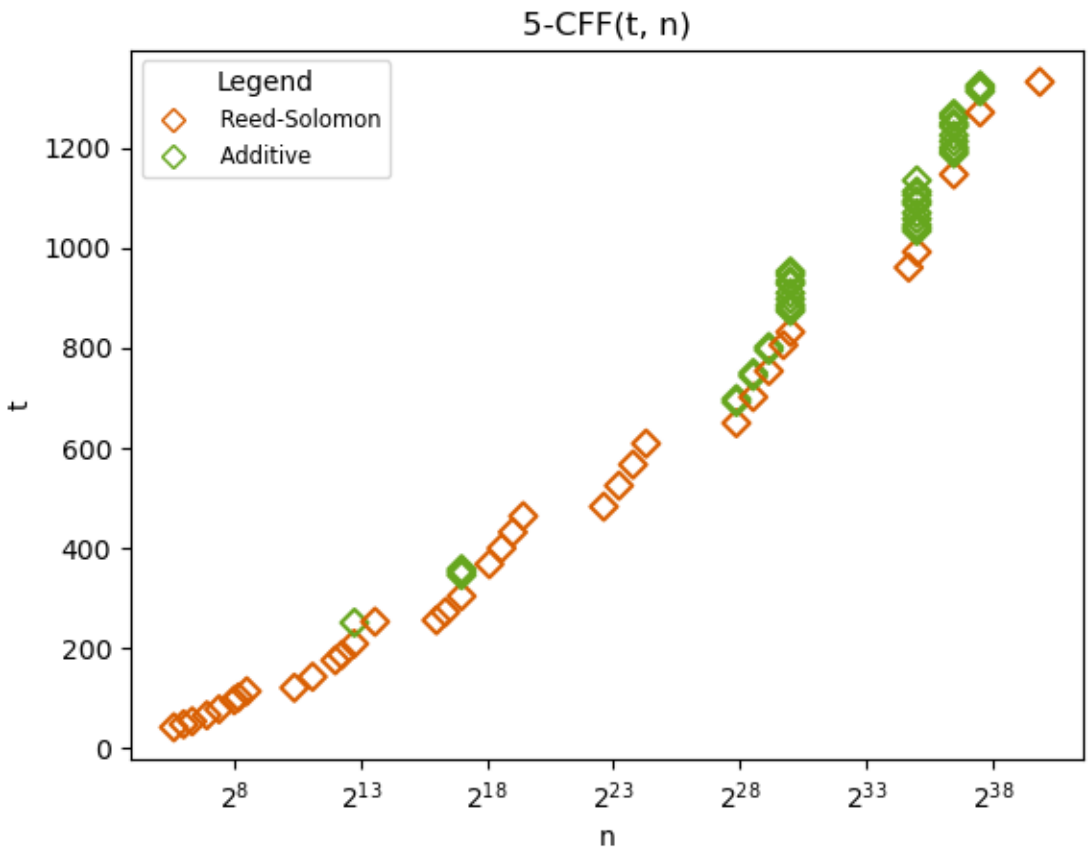
Input: $d, t, n \in \mathbb{Z}, 0 < d < t \leq n$, A is a $t \times n$ 0-1 matrix.
Output: True if A is the incidence matrix of a d-CFF(t, n), False otherwise.

```
1: T ← [0, 1, ..., d]
2: repeat                                   ▷ Iterate over every (d + 1)-subset of columns in T
3:     v ← an array of length d + 1 filled with −1
4:     f ← 0    ▷ f counts the number of weight-1 rows found for this subset of columns
5:     for r ← 0, 1, ..., t − 1 do
6:         s ← 0
7:         for i ← 0, 1, ..., d do
8:             if A[r][T[i]] = 1 then
9:                 s ← s + 1 ▷ s counts the number of ones per row of the submatrix with columns indexed by T at row r
10:                if s > 1 then        ▷ Exit early if the weight of the row is greater than 1
11:                    break
12:                end if
13:                e ← i                ▷ e stores the column of the 1 in this weight-1 row
14:            end if
15:        end for
16:        if s = 1 and v[e] = −1 then ▷ If row r has the first weight-1 row with the 1 in column T[i]
17:            v[e] ← r
18:            f ← f + 1
19:            if f = d + 1 then
20:                break
21:            end if
22:        end if
23:    end for
24:    if f ≠ d + 1 then        ▷ Return false if any of the weight-1 rows were not present
25:        return False
26:    end if
27: until k-subset-lex-successor(n, d + 1, T) = False
28: return True
```

Appendix B

CFF tables

B.1 5-CFF table

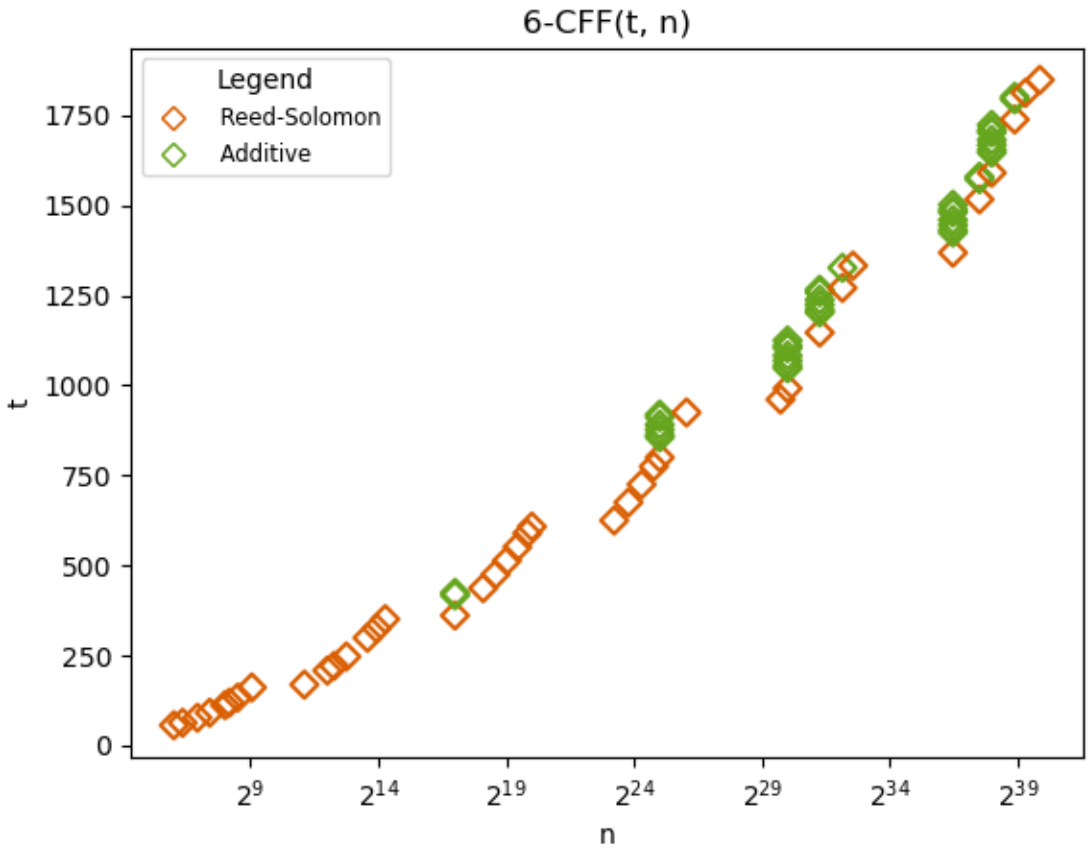


t	n	Source
42	49	RS(7^1,2,6)
48	64	RS(2^3,2,6)
54	81	RS(3^2,2,6)
66	121	RS(11^1,2,6)
78	169	RS(13^1,2,6)
96	256	RS(2^4,2,6)
102	289	RS(17^1,2,6)
114	361	RS(19^1,2,6)
121	1331	RS(11^1,3,11)
143	2197	RS(13^1,3,11)
176	4096	RS(2^4,3,11)
187	4913	RS(17^1,3,11)
209	6859	RS(19^1,3,11)
251	6908	Add(42,209)
253	12167	RS(23^1,3,11)
256	65536	RS(2^4,4,16)
272	83521	RS(17^1,4,16)
304	130321	RS(19^1,4,16)
346	130370	Add(42,304)
352	130385	Add(48,304)
358	130402	Add(54,304)
368	279841	RS(23^1,4,16)
400	390625	RS(5^2,4,16)
432	531441	RS(3^3,4,16)
464	707281	RS(29^1,4,16)
483	6436343	RS(23^1,5,21)
525	9765625	RS(5^2,5,21)
567	14348907	RS(3^3,5,21)
609	20511149	RS(29^1,5,21)
650	244140625	RS(5^2,6,26)
692	244140674	Add(42,650)
698	244140689	Add(48,650)
702	387420489	RS(3^3,6,26)
744	387420538	Add(42,702)
750	387420553	Add(48,702)
754	594823321	RS(29^1,6,26)
796	594823370	Add(42,754)
802	594823385	Add(48,754)
806	887503681	RS(31^1,6,26)
832	1073741824	RS(2^5,6,26)
874	1073741873	Add(42,832)
880	1073741888	Add(48,832)
886	1073741905	Add(54,832)
898	1073741945	Add(66,832)
910	1073741993	Add(78,832)

t	n	Source
928	1073742080	Add(96,832)
934	1073742113	Add(102,832)
946	1073742185	Add(114,832)
953	1073743155	Add(121,832)
961	27512614111	RS(31^1,7,31)
992	34359738368	RS(2^5,7,31)
1034	34359738417	Add(42,992)
1040	34359738432	Add(48,992)
1046	34359738449	Add(54,992)
1058	34359738489	Add(66,992)
1070	34359738537	Add(78,992)
1088	34359738624	Add(96,992)
1094	34359738657	Add(102,992)
1106	34359738729	Add(114,992)
1113	34359739699	Add(121,992)
1135	34359740565	Add(143,992)
1147	94931877133	RS(37^1,7,31)
1189	94931877182	Add(42,1147)
1195	94931877197	Add(48,1147)
1201	94931877214	Add(54,1147)
1213	94931877254	Add(66,1147)
1225	94931877302	Add(78,1147)
1243	94931877389	Add(96,1147)
1249	94931877422	Add(102,1147)
1261	94931877494	Add(114,1147)
1268	94931878464	Add(121,1147)
1271	194754273881	RS(41^1,7,31)
1313	194754273930	Add(42,1271)
1319	194754273945	Add(48,1271)
1325	194754273962	Add(54,1271)
1332	3512479453921	RS(37^1,8,36)
1374	3512479453970	Add(42,1332)
1380	3512479453985	Add(48,1332)
1386	3512479454002	Add(54,1332)
1398	3512479454042	Add(66,1332)
1410	3512479454090	Add(78,1332)
1428	3512479454177	Add(96,1332)
1434	3512479454210	Add(102,1332)
1446	3512479454282	Add(114,1332)
1453	3512479455252	Add(121,1332)
1475	3512479456118	Add(143,1332)
1476	7984925229121	RS(41^1,8,36)
1518	7984925229170	Add(42,1476)
1524	7984925229185	Add(48,1476)
1530	7984925229202	Add(54,1476)

t	n	Source
1542	7984925229242	Add(66,1476)
1548	10000000000000	RS(43^1,8,36)

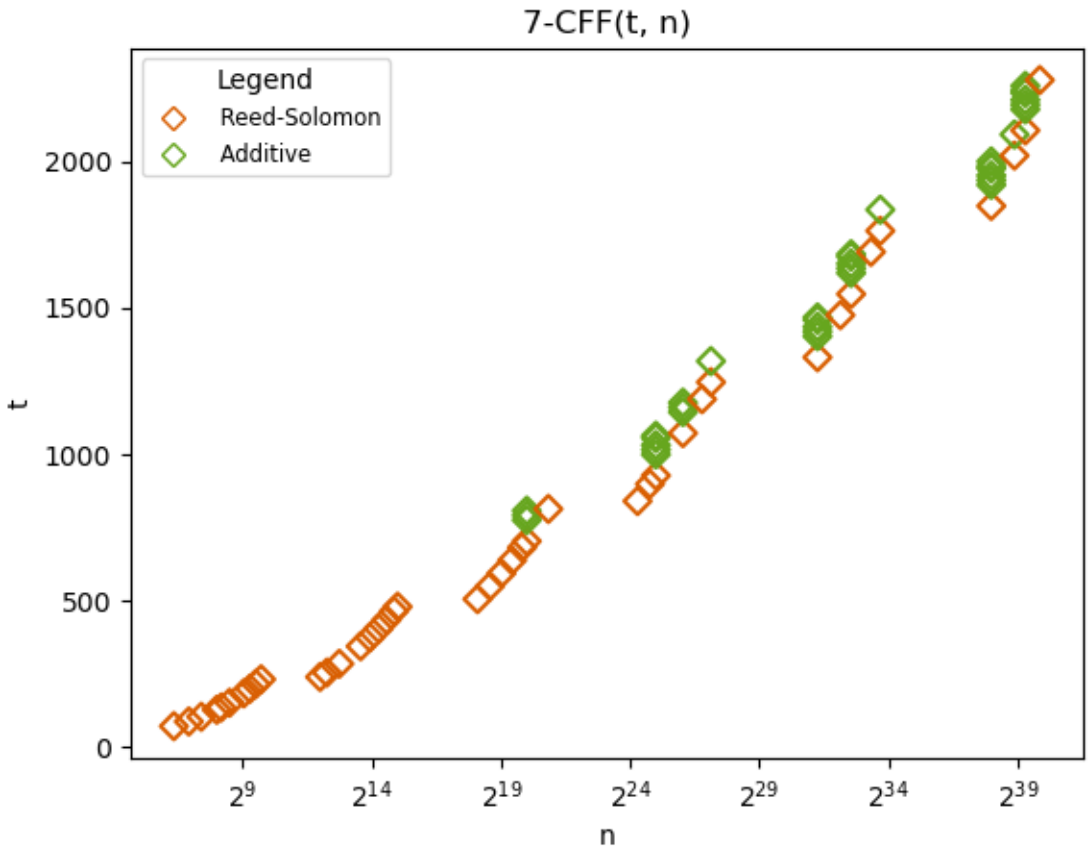
B.2 6-CFF table



t	n	Source
56	64	RS(2^3,2,7)
63	81	RS(3^2,2,7)
77	121	RS(11^1,2,7)
91	169	RS(13^1,2,7)
112	256	RS(2^4,2,7)
119	289	RS(17^1,2,7)
133	361	RS(19^1,2,7)
161	529	RS(23^1,2,7)
169	2197	RS(13^1,3,13)
208	4096	RS(2^4,3,13)
221	4913	RS(17^1,3,13)
247	6859	RS(19^1,3,13)
299	12167	RS(23^1,3,13)
325	15625	RS(5^2,3,13)
351	19683	RS(3^3,3,13)
361	130321	RS(19^1,4,19)
417	130385	Add(56,361)
424	130402	Add(63,361)
437	279841	RS(23^1,4,19)
475	390625	RS(5^2,4,19)
513	531441	RS(3^3,4,19)
551	707281	RS(29^1,4,19)
589	923521	RS(31^1,4,19)
608	1048576	RS(2^5,4,19)
625	9765625	RS(5^2,5,25)
675	14348907	RS(3^3,5,25)
725	20511149	RS(29^1,5,25)
775	28629151	RS(31^1,5,25)
800	33554432	RS(2^5,5,25)
856	33554496	Add(56,800)
863	33554513	Add(63,800)
877	33554553	Add(77,800)
891	33554601	Add(91,800)
912	33554688	Add(112,800)
919	33554721	Add(119,800)
925	69343957	RS(37^1,5,25)
961	887503681	RS(31^1,6,31)
992	1073741824	RS(2^5,6,31)
1048	1073741888	Add(56,992)
1055	1073741905	Add(63,992)
1069	1073741945	Add(77,992)
1083	1073741993	Add(91,992)
1104	1073742080	Add(112,992)
1111	1073742113	Add(119,992)
1125	1073742185	Add(133,992)

t	n	Source
1147	2565726409	RS(37^1,6,31)
1203	2565726473	Add(56,1147)
1210	2565726490	Add(63,1147)
1224	2565726530	Add(77,1147)
1238	2565726578	Add(91,1147)
1259	2565726665	Add(112,1147)
1266	2565726698	Add(119,1147)
1271	4750104241	RS(41^1,6,31)
1327	4750104305	Add(56,1271)
1333	6321363049	RS(43^1,6,31)
1369	94931877133	RS(37^1,7,37)
1425	94931877197	Add(56,1369)
1432	94931877214	Add(63,1369)
1446	94931877254	Add(77,1369)
1460	94931877302	Add(91,1369)
1481	94931877389	Add(112,1369)
1488	94931877422	Add(119,1369)
1502	94931877494	Add(133,1369)
1517	194754273881	RS(41^1,7,37)
1573	194754273945	Add(56,1517)
1580	194754273962	Add(63,1517)
1591	271818611107	RS(43^1,7,37)
1647	271818611171	Add(56,1591)
1654	271818611188	Add(63,1591)
1668	271818611228	Add(77,1591)
1682	271818611276	Add(91,1591)
1703	271818611363	Add(112,1591)
1710	271818611396	Add(119,1591)
1724	271818611468	Add(133,1591)
1739	506623120463	RS(47^1,7,37)
1795	506623120527	Add(56,1739)
1802	506623120544	Add(63,1739)
1813	678223072849	RS(7^2,7,37)
1849	10000000000000	RS(43^1,8,43)

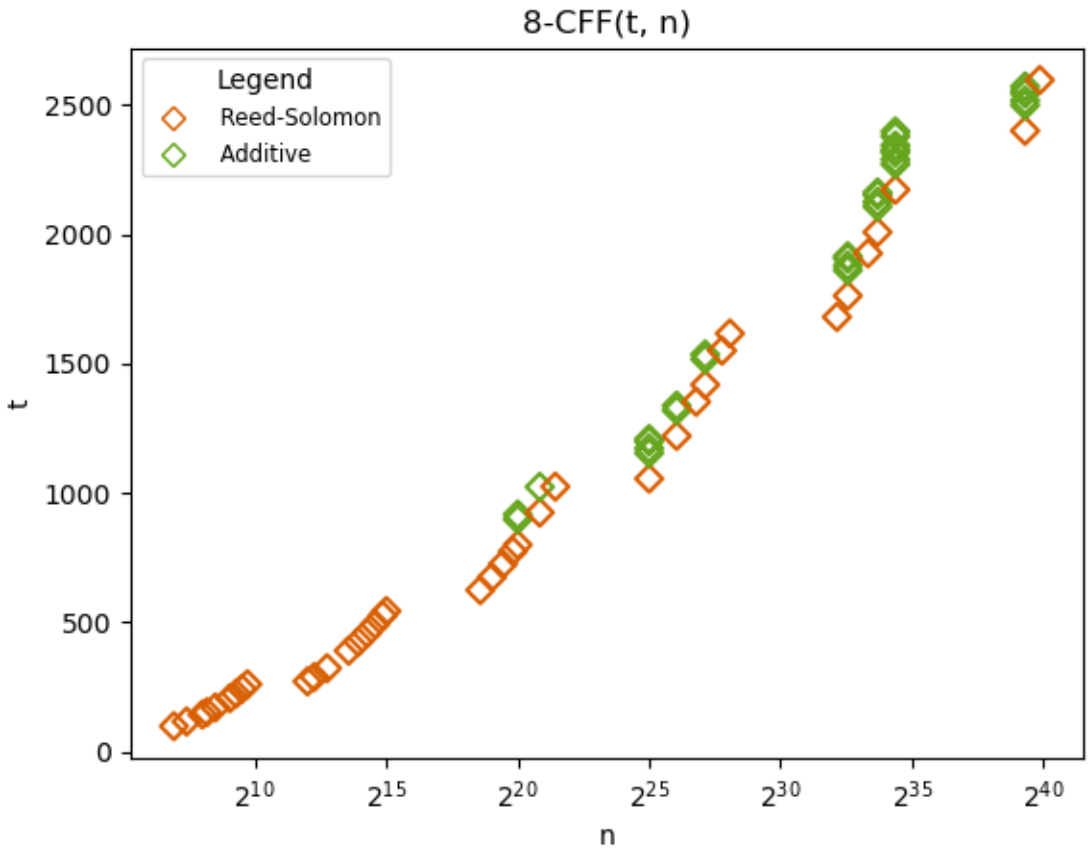
B.3 7-CFF table



t	n	Source
72	81	RS(3^2,2,8)
88	121	RS(11^1,2,8)
104	169	RS(13^1,2,8)
128	256	RS(2^4,2,8)
136	289	RS(17^1,2,8)
152	361	RS(19^1,2,8)
184	529	RS(23^1,2,8)
200	625	RS(5^2,2,8)
216	729	RS(3^3,2,8)
232	841	RS(29^1,2,8)
240	4096	RS(2^4,3,15)
255	4913	RS(17^1,3,15)
285	6859	RS(19^1,3,15)
345	12167	RS(23^1,3,15)
375	15625	RS(5^2,3,15)
405	19683	RS(3^3,3,15)
435	24389	RS(29^1,3,15)
465	29791	RS(31^1,3,15)
480	32768	RS(2^5,3,15)
506	279841	RS(23^1,4,22)
550	390625	RS(5^2,4,22)
594	531441	RS(3^3,4,22)
638	707281	RS(29^1,4,22)
682	923521	RS(31^1,4,22)
704	1048576	RS(2^5,4,22)
776	1048657	Add(72,704)
792	1048697	Add(88,704)
808	1048745	Add(104,704)
814	1874161	RS(37^1,4,22)
841	20511149	RS(29^1,5,29)
899	28629151	RS(31^1,5,29)
928	33554432	RS(2^5,5,29)
1000	33554513	Add(72,928)
1016	33554553	Add(88,928)
1032	33554601	Add(104,928)
1056	33554688	Add(128,928)
1064	33554721	Add(136,928)
1073	69343957	RS(37^1,5,29)
1145	69344038	Add(72,1073)
1161	69344078	Add(88,1073)
1177	69344126	Add(104,1073)
1189	115856201	RS(41^1,5,29)
1247	147008443	RS(43^1,5,29)
1319	147008524	Add(72,1247)
1332	2565726409	RS(37^1,6,36)

t	n	Source
1404	2565726490	Add(72,1332)
1420	2565726530	Add(88,1332)
1436	2565726578	Add(104,1332)
1460	2565726665	Add(128,1332)
1468	2565726698	Add(136,1332)
1476	4750104241	RS(41^1,6,36)
1548	6321363049	RS(43^1,6,36)
1620	6321363130	Add(72,1548)
1636	6321363170	Add(88,1548)
1652	6321363218	Add(104,1548)
1676	6321363305	Add(128,1548)
1684	6321363338	Add(136,1548)
1692	10779215329	RS(47^1,6,36)
1764	13841287201	RS(7^2,6,36)
1836	13841287282	Add(72,1764)
1849	271818611107	RS(43^1,7,43)
1921	271818611188	Add(72,1849)
1937	271818611228	Add(88,1849)
1953	271818611276	Add(104,1849)
1977	271818611363	Add(128,1849)
1985	271818611396	Add(136,1849)
2001	271818611468	Add(152,1849)
2021	506623120463	RS(47^1,7,43)
2093	506623120544	Add(72,2021)
2107	678223072849	RS(7^2,7,43)
2179	678223072930	Add(72,2107)
2195	678223072970	Add(88,2107)
2211	678223073018	Add(104,2107)
2235	678223073105	Add(128,2107)
2243	678223073138	Add(136,2107)
2259	678223073210	Add(152,2107)
2279	1174711139837	RS(53^1,7,43)
2351	1174711139918	Add(72,2279)
2367	1174711139958	Add(88,2279)
2383	1174711140006	Add(104,2279)
2407	1174711140093	Add(128,2279)
2415	1174711140126	Add(136,2279)
2431	1174711140198	Add(152,2279)
2450	10000000000000	RS(7^2,8,50)

B.4 8-CFF table

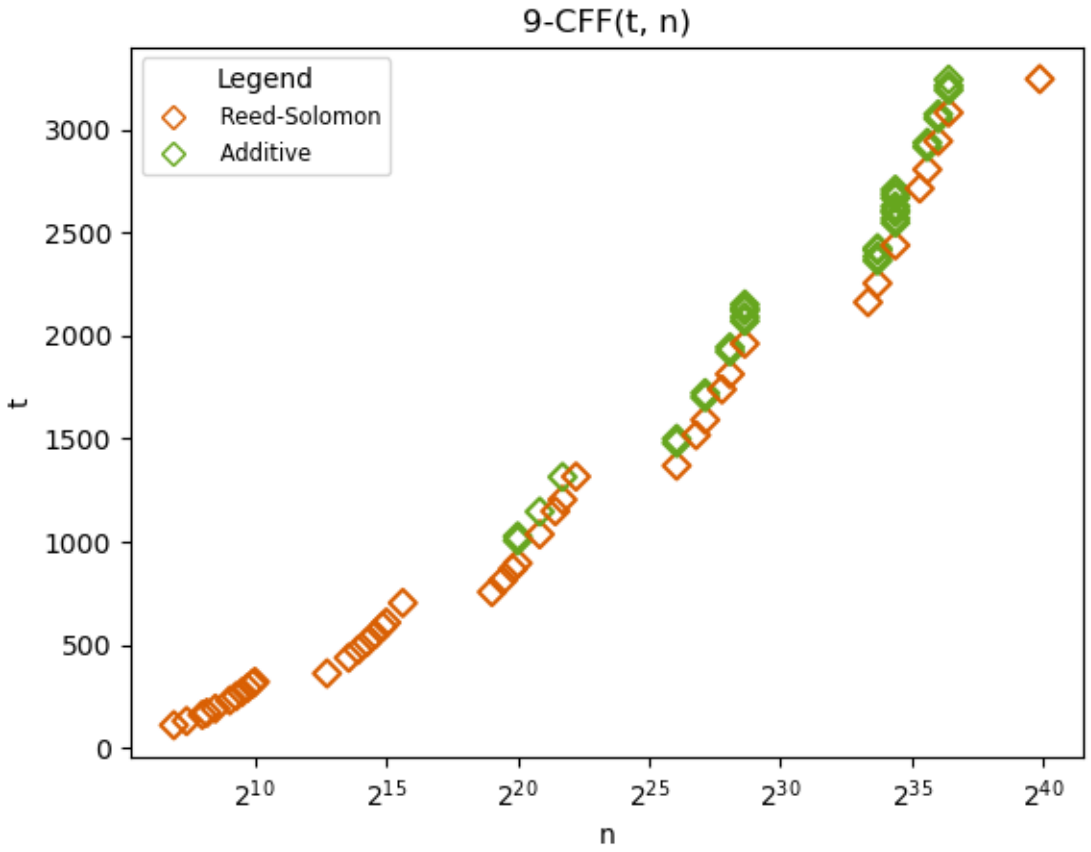


t	n	Source
99	121	RS(11^1,2,9)
117	169	RS(13^1,2,9)
144	256	RS(2^4,2,9)
153	289	RS(17^1,2,9)
171	361	RS(19^1,2,9)
207	529	RS(23^1,2,9)
225	625	RS(5^2,2,9)
243	729	RS(3^3,2,9)
261	841	RS(29^1,2,9)
272	4096	RS(2^4,3,17)
289	4913	RS(17^1,3,17)
323	6859	RS(19^1,3,17)
391	12167	RS(23^1,3,17)
425	15625	RS(5^2,3,17)
459	19683	RS(3^3,3,17)
493	24389	RS(29^1,3,17)
527	29791	RS(31^1,3,17)
544	32768	RS(2^5,3,17)
625	390625	RS(5^2,4,25)
675	531441	RS(3^3,4,25)
725	707281	RS(29^1,4,25)
775	923521	RS(31^1,4,25)
800	1048576	RS(2^5,4,25)
899	1048697	Add(99,800)
917	1048745	Add(117,800)
925	1874161	RS(37^1,4,25)
1024	1874282	Add(99,925)
1025	2825761	RS(41^1,4,25)
1056	33554432	RS(2^5,5,33)
1155	33554553	Add(99,1056)
1173	33554601	Add(117,1056)
1200	33554688	Add(144,1056)
1209	33554721	Add(153,1056)
1221	69343957	RS(37^1,5,33)
1320	69344078	Add(99,1221)
1338	69344126	Add(117,1221)
1353	115856201	RS(41^1,5,33)
1419	147008443	RS(43^1,5,33)
1518	147008564	Add(99,1419)
1536	147008612	Add(117,1419)
1551	229345007	RS(47^1,5,33)
1617	282475249	RS(7^2,5,33)
1681	4750104241	RS(41^1,6,41)
1763	6321363049	RS(43^1,6,41)
1862	6321363170	Add(99,1763)

t	n	Source
1880	6321363218	Add(117,1763)
1907	6321363305	Add(144,1763)
1916	6321363338	Add(153,1763)
1927	10779215329	RS(47^1,6,41)
2009	13841287201	RS(7^2,6,41)
2108	13841287322	Add(99,2009)
2126	13841287370	Add(117,2009)
2153	13841287457	Add(144,2009)
2162	13841287490	Add(153,2009)
2173	22164361129	RS(53^1,6,41)
2272	22164361250	Add(99,2173)
2290	22164361298	Add(117,2173)
2317	22164361385	Add(144,2173)
2326	22164361418	Add(153,2173)
2344	22164361490	Add(171,2173)
2380	22164361658	Add(207,2173)
2398	22164361754	Add(225,2173)
2401	678223072849	RS(7^2,7,49)
2500	678223072970	Add(99,2401)
2518	678223073018	Add(117,2401)
2545	678223073105	Add(144,2401)
2554	678223073138	Add(153,2401)
2572	678223073210	Add(171,2401)
2597	1174711139837	RS(53^1,7,49)
2696	1174711139958	Add(99,2597)
2714	1174711140006	Add(117,2597)
2741	1174711140093	Add(144,2597)
2750	1174711140126	Add(153,2597)
2768	1174711140198	Add(171,2597)
2804	1174711140366	Add(207,2597)
2822	1174711140462	Add(225,2597)
2840	1174711140566	Add(243,2597)
2858	1174711140678	Add(261,2597)
2869	1174711143933	Add(272,2597)
2886	1174711144750	Add(289,2597)
2891	2488651484819	RS(59^1,7,49)
2989	3142742836021	RS(61^1,7,49)
3088	3142742836142	Add(99,2989)
3106	3142742836190	Add(117,2989)
3133	3142742836277	Add(144,2989)
3136	4398046511104	RS(2^6,7,49)
3235	4398046511225	Add(99,3136)
3253	4398046511273	Add(117,3136)
3280	4398046511360	Add(144,3136)
3283	6060711605323	RS(67^1,7,49)

t	n	Source
3363	10000000000000	RS(59^1,8,57)

B.5 9-CFF table

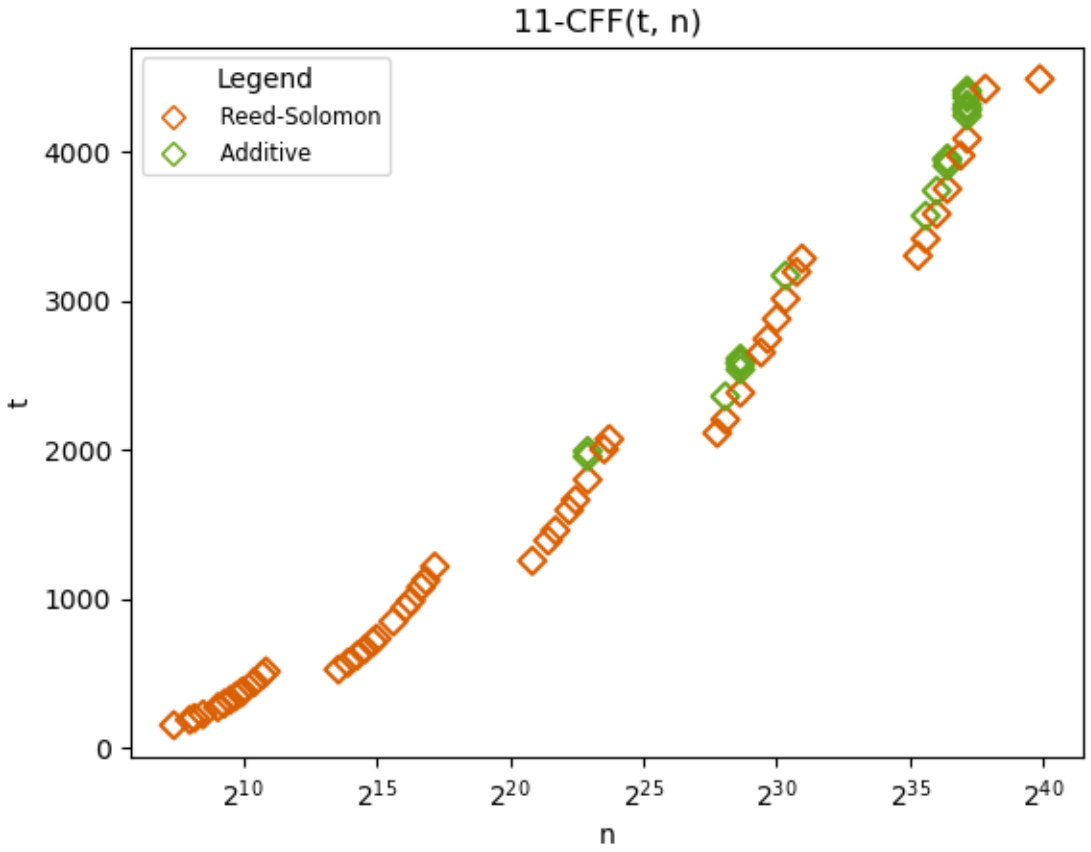


t	n	Source
110	121	RS(11^1,2,10)
130	169	RS(13^1,2,10)
160	256	RS(2^4,2,10)
170	289	RS(17^1,2,10)
190	361	RS(19^1,2,10)
230	529	RS(23^1,2,10)
250	625	RS(5^2,2,10)
270	729	RS(3^3,2,10)
290	841	RS(29^1,2,10)
310	961	RS(31^1,2,10)
320	1024	RS(2^5,2,10)
361	6859	RS(19^1,3,19)
437	12167	RS(23^1,3,19)
475	15625	RS(5^2,3,19)
513	19683	RS(3^3,3,19)
551	24389	RS(29^1,3,19)
589	29791	RS(31^1,3,19)
608	32768	RS(2^5,3,19)
703	50653	RS(37^1,3,19)
756	531441	RS(3^3,4,28)
812	707281	RS(29^1,4,28)
868	923521	RS(31^1,4,28)
896	1048576	RS(2^5,4,28)
1006	1048697	Add(110,896)
1026	1048745	Add(130,896)
1036	1874161	RS(37^1,4,28)
1146	1874282	Add(110,1036)
1148	2825761	RS(41^1,4,28)
1204	3418801	RS(43^1,4,28)
1314	3418922	Add(110,1204)
1316	4879681	RS(47^1,4,28)
1369	69343957	RS(37^1,5,37)
1479	69344078	Add(110,1369)
1499	69344126	Add(130,1369)
1517	115856201	RS(41^1,5,37)
1591	147008443	RS(43^1,5,37)
1701	147008564	Add(110,1591)
1721	147008612	Add(130,1591)
1739	229345007	RS(47^1,5,37)
1813	282475249	RS(7^2,5,37)
1923	282475370	Add(110,1813)
1943	282475418	Add(130,1813)
1961	418195493	RS(53^1,5,37)
2071	418195614	Add(110,1961)
2091	418195662	Add(130,1961)

t	n	Source
2121	418195749	Add(160,1961)
2131	418195782	Add(170,1961)
2151	418195854	Add(190,1961)
2162	10779215329	RS(47^1,6,46)
2254	13841287201	RS(7^2,6,46)
2364	13841287322	Add(110,2254)
2384	13841287370	Add(130,2254)
2414	13841287457	Add(160,2254)
2424	13841287490	Add(170,2254)
2438	22164361129	RS(53^1,6,46)
2548	22164361250	Add(110,2438)
2568	22164361298	Add(130,2438)
2598	22164361385	Add(160,2438)
2608	22164361418	Add(170,2438)
2628	22164361490	Add(190,2438)
2668	22164361658	Add(230,2438)
2688	22164361754	Add(250,2438)
2708	22164361858	Add(270,2438)
2714	42180533641	RS(59^1,6,46)
2806	51520374361	RS(61^1,6,46)
2916	51520374482	Add(110,2806)
2936	51520374530	Add(130,2806)
2944	68719476736	RS(2^6,6,46)
3054	68719476857	Add(110,2944)
3074	68719476905	Add(130,2944)
3082	90458382169	RS(67^1,6,46)
3192	90458382290	Add(110,3082)
3212	90458382338	Add(130,3082)
3242	90458382425	Add(160,3082)
3245	2488651484819	RS(59^1,7,55)
3355	3142742836021	RS(61^1,7,55)
3465	3142742836142	Add(110,3355)
3485	3142742836190	Add(130,3355)
3515	3142742836277	Add(160,3355)
3520	4398046511104	RS(2^6,7,55)
3630	4398046511225	Add(110,3520)
3650	4398046511273	Add(130,3520)
3680	4398046511360	Add(160,3520)
3685	6060711605323	RS(67^1,7,55)
3795	6060711605444	Add(110,3685)
3815	6060711605492	Add(130,3685)
3845	6060711605579	Add(160,3685)
3855	6060711605612	Add(170,3685)
3875	6060711605684	Add(190,3685)
3905	9095120158391	RS(71^1,7,55)

t	n	Source
4015	10000000000000	RS(73^1,7,55)

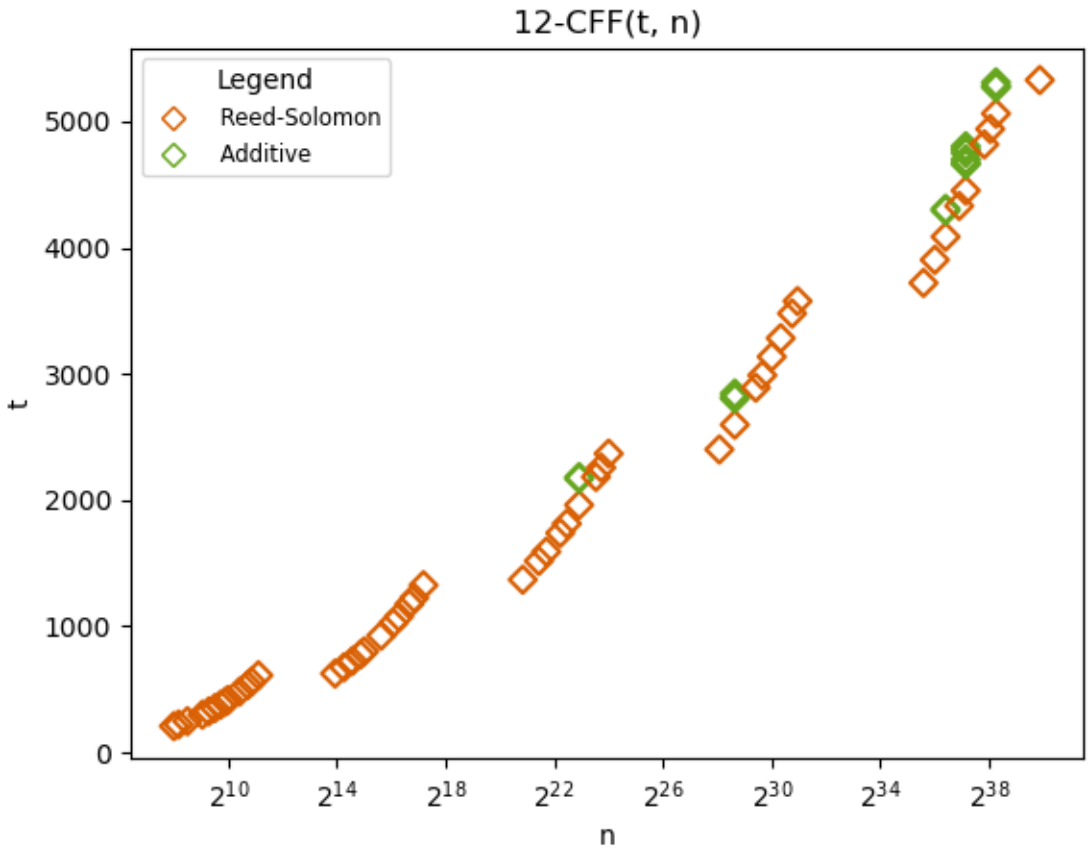
B.6 11-CFF table



t	n	Source
156	169	RS(13^1,2,12)
192	256	RS(2^4,2,12)
204	289	RS(17^1,2,12)
228	361	RS(19^1,2,12)
276	529	RS(23^1,2,12)
300	625	RS(5^2,2,12)
324	729	RS(3^3,2,12)
348	841	RS(29^1,2,12)
372	961	RS(31^1,2,12)
384	1024	RS(2^5,2,12)
444	1369	RS(37^1,2,12)
492	1681	RS(41^1,2,12)
516	1849	RS(43^1,2,12)
529	12167	RS(23^1,3,23)
575	15625	RS(5^2,3,23)
621	19683	RS(3^3,3,23)
667	24389	RS(29^1,3,23)
713	29791	RS(31^1,3,23)
736	32768	RS(2^5,3,23)
851	50653	RS(37^1,3,23)
943	68921	RS(41^1,3,23)
989	79507	RS(43^1,3,23)
1081	103823	RS(47^1,3,23)
1127	117649	RS(7^2,3,23)
1219	148877	RS(53^1,3,23)
1258	1874161	RS(37^1,4,34)
1394	2825761	RS(41^1,4,34)
1462	3418801	RS(43^1,4,34)
1598	4879681	RS(47^1,4,34)
1666	5764801	RS(7^2,4,34)
1802	7890481	RS(53^1,4,34)
1958	7890650	Add(156,1802)
1994	7890737	Add(192,1802)
2006	12117361	RS(59^1,4,34)
2074	13845841	RS(61^1,4,34)
2115	229345007	RS(47^1,5,45)
2205	282475249	RS(7^2,5,45)
2361	282475418	Add(156,2205)
2385	418195493	RS(53^1,5,45)
2541	418195662	Add(156,2385)
2577	418195749	Add(192,2385)
2589	418195782	Add(204,2385)
2613	418195854	Add(228,2385)
2655	714924299	RS(59^1,5,45)
2745	844596301	RS(61^1,5,45)

t	n	Source
2880	1073741824	RS(2^6,5,45)
3015	1350125107	RS(67^1,5,45)
3171	1350125276	Add(156,3015)
3195	1804229351	RS(71^1,5,45)
3285	2073071593	RS(73^1,5,45)
3304	42180533641	RS(59^1,6,56)
3416	51520374361	RS(61^1,6,56)
3572	51520374530	Add(156,3416)
3584	68719476736	RS(2^6,6,56)
3740	68719476905	Add(156,3584)
3752	90458382169	RS(67^1,6,56)
3908	90458382338	Add(156,3752)
3944	90458382425	Add(192,3752)
3956	90458382458	Add(204,3752)
3976	128100283921	RS(71^1,6,56)
4088	151334226289	RS(73^1,6,56)
4244	151334226458	Add(156,4088)
4280	151334226545	Add(192,4088)
4292	151334226578	Add(204,4088)
4316	151334226650	Add(228,4088)
4364	151334226818	Add(276,4088)
4388	151334226914	Add(300,4088)
4412	151334227018	Add(324,4088)
4424	243087455521	RS(79^1,6,56)
4489	6060711605323	RS(67^1,7,67)
4645	6060711605492	Add(156,4489)
4681	6060711605579	Add(192,4489)
4693	6060711605612	Add(204,4489)
4717	6060711605684	Add(228,4489)
4757	9095120158391	RS(71^1,7,67)
4891	10000000000000	RS(73^1,7,67)

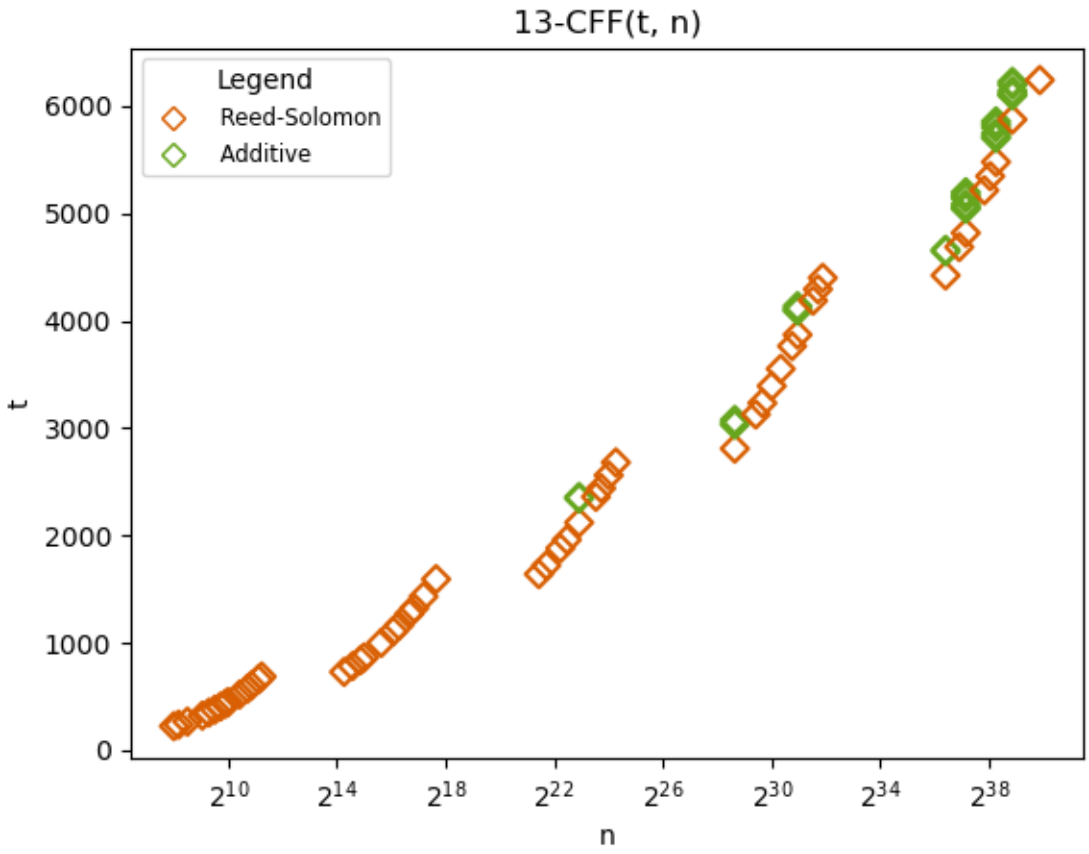
B.7 12-CFF table



t	n	Source
208	256	RS(2^4,2,13)
221	289	RS(17^1,2,13)
247	361	RS(19^1,2,13)
299	529	RS(23^1,2,13)
325	625	RS(5^2,2,13)
351	729	RS(3^3,2,13)
377	841	RS(29^1,2,13)
403	961	RS(31^1,2,13)
416	1024	RS(2^5,2,13)
481	1369	RS(37^1,2,13)
533	1681	RS(41^1,2,13)
559	1849	RS(43^1,2,13)
611	2209	RS(47^1,2,13)
625	15625	RS(5^2,3,25)
675	19683	RS(3^3,3,25)
725	24389	RS(29^1,3,25)
775	29791	RS(31^1,3,25)
800	32768	RS(2^5,3,25)
925	50653	RS(37^1,3,25)
1025	68921	RS(41^1,3,25)
1075	79507	RS(43^1,3,25)
1175	103823	RS(47^1,3,25)
1225	117649	RS(7^2,3,25)
1325	148877	RS(53^1,3,25)
1369	1874161	RS(37^1,4,37)
1517	2825761	RS(41^1,4,37)
1591	3418801	RS(43^1,4,37)
1739	4879681	RS(47^1,4,37)
1813	5764801	RS(7^2,4,37)
1961	7890481	RS(53^1,4,37)
2169	7890737	Add(208,1961)
2182	7890770	Add(221,1961)
2183	12117361	RS(59^1,4,37)
2257	13845841	RS(61^1,4,37)
2368	16777216	RS(2^6,4,37)
2401	282475249	RS(7^2,5,49)
2597	418195493	RS(53^1,5,49)
2805	418195749	Add(208,2597)
2818	418195782	Add(221,2597)
2844	418195854	Add(247,2597)
2891	714924299	RS(59^1,5,49)
2989	844596301	RS(61^1,5,49)
3136	1073741824	RS(2^6,5,49)
3283	1350125107	RS(67^1,5,49)
3479	1804229351	RS(71^1,5,49)

t	n	Source
3577	2073071593	RS(73^1,5,49)
3721	51520374361	RS(61^1,6,61)
3904	68719476736	RS(2^6,6,61)
4087	90458382169	RS(67^1,6,61)
4295	90458382425	Add(208,4087)
4308	90458382458	Add(221,4087)
4331	128100283921	RS(71^1,6,61)
4453	151334226289	RS(73^1,6,61)
4661	151334226545	Add(208,4453)
4674	151334226578	Add(221,4453)
4700	151334226650	Add(247,4453)
4752	151334226818	Add(299,4453)
4778	151334226914	Add(325,4453)
4804	151334227018	Add(351,4453)
4819	243087455521	RS(79^1,6,61)
4941	282429536481	RS(3^4,6,61)
5063	326940373369	RS(83^1,6,61)
5271	326940373625	Add(208,5063)
5284	326940373658	Add(221,5063)
5310	326940373730	Add(247,5063)
5329	10000000000000	RS(73^1,7,73)

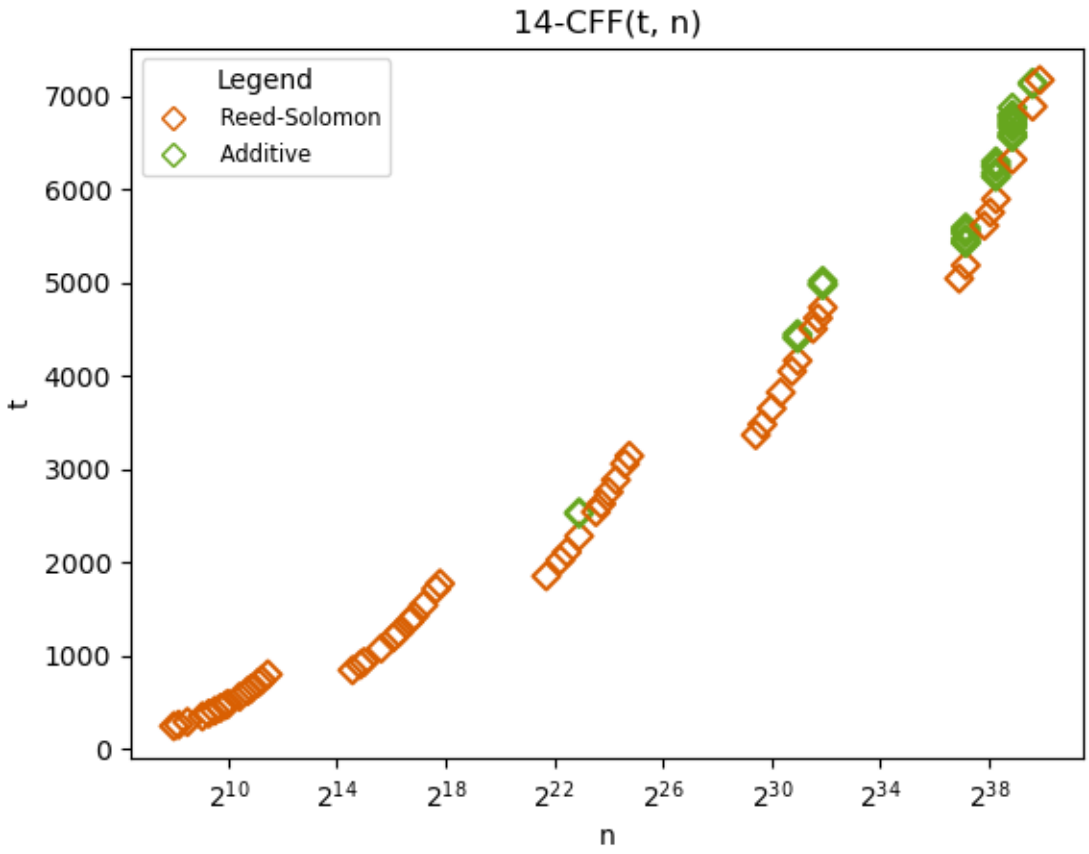
B.8 13-CFF table



t	n	Source
224	256	RS(2^4,2,14)
238	289	RS(17^1,2,14)
266	361	RS(19^1,2,14)
322	529	RS(23^1,2,14)
350	625	RS(5^2,2,14)
378	729	RS(3^3,2,14)
406	841	RS(29^1,2,14)
434	961	RS(31^1,2,14)
448	1024	RS(2^5,2,14)
518	1369	RS(37^1,2,14)
574	1681	RS(41^1,2,14)
602	1849	RS(43^1,2,14)
658	2209	RS(47^1,2,14)
686	2401	RS(7^2,2,14)
729	19683	RS(3^3,3,27)
783	24389	RS(29^1,3,27)
837	29791	RS(31^1,3,27)
864	32768	RS(2^5,3,27)
999	50653	RS(37^1,3,27)
1107	68921	RS(41^1,3,27)
1161	79507	RS(43^1,3,27)
1269	103823	RS(47^1,3,27)
1323	117649	RS(7^2,3,27)
1431	148877	RS(53^1,3,27)
1593	205379	RS(59^1,3,27)
1640	2825761	RS(41^1,4,40)
1720	3418801	RS(43^1,4,40)
1880	4879681	RS(47^1,4,40)
1960	5764801	RS(7^2,4,40)
2120	7890481	RS(53^1,4,40)
2344	7890737	Add(224,2120)
2358	7890770	Add(238,2120)
2360	12117361	RS(59^1,4,40)
2440	13845841	RS(61^1,4,40)
2560	16777216	RS(2^6,4,40)
2680	20151121	RS(67^1,4,40)
2809	418195493	RS(53^1,5,53)
3033	418195749	Add(224,2809)
3047	418195782	Add(238,2809)
3075	418195854	Add(266,2809)
3127	714924299	RS(59^1,5,53)
3233	844596301	RS(61^1,5,53)
3392	1073741824	RS(2^6,5,53)
3551	1350125107	RS(67^1,5,53)
3763	1804229351	RS(71^1,5,53)

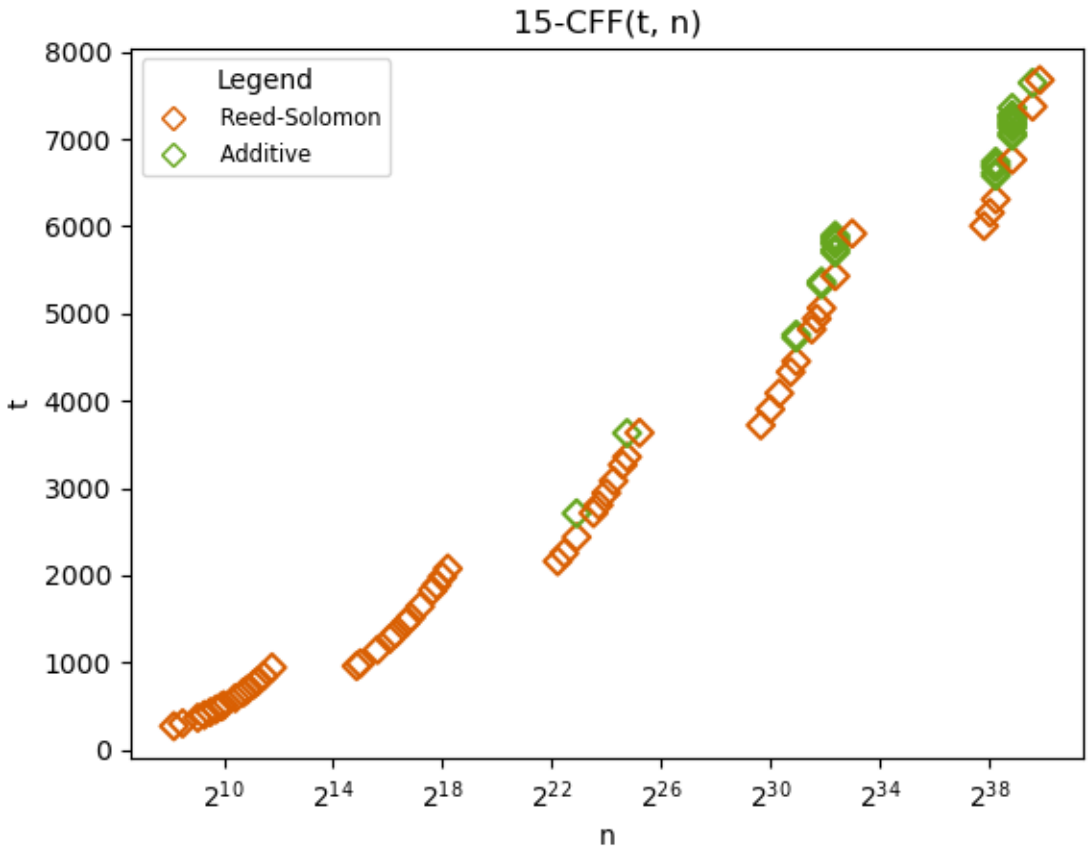
t	n	Source
3869	2073071593	RS(73^1,5,53)
4093	2073071849	Add(224,3869)
4107	2073071882	Add(238,3869)
4135	2073071954	Add(266,3869)
4187	3077056399	RS(79^1,5,53)
4293	3486784401	RS(3^4,5,53)
4399	3939040643	RS(83^1,5,53)
4422	90458382169	RS(67^1,6,66)
4646	90458382425	Add(224,4422)
4660	90458382458	Add(238,4422)
4686	128100283921	RS(71^1,6,66)
4818	151334226289	RS(73^1,6,66)
5042	151334226545	Add(224,4818)
5056	151334226578	Add(238,4818)
5084	151334226650	Add(266,4818)
5140	151334226818	Add(322,4818)
5168	151334226914	Add(350,4818)
5196	151334227018	Add(378,4818)
5214	243087455521	RS(79^1,6,66)
5346	282429536481	RS(3^4,6,66)
5478	326940373369	RS(83^1,6,66)
5702	326940373625	Add(224,5478)
5716	326940373658	Add(238,5478)
5744	326940373730	Add(266,5478)
5800	326940373898	Add(322,5478)
5828	326940373994	Add(350,5478)
5856	326940374098	Add(378,5478)
5874	496981290961	RS(89^1,6,66)
6098	496981291217	Add(224,5874)
6112	496981291250	Add(238,5874)
6140	496981291322	Add(266,5874)
6196	496981291490	Add(322,5874)
6224	496981291586	Add(350,5874)
6241	10000000000000	RS(79^1,7,79)

B.9 14-CFF table



t	n	Source
240	256	RS(2^4,2,15)
255	289	RS(17^1,2,15)
285	361	RS(19^1,2,15)
345	529	RS(23^1,2,15)
375	625	RS(5^2,2,15)
405	729	RS(3^3,2,15)
435	841	RS(29^1,2,15)
465	961	RS(31^1,2,15)
480	1024	RS(2^5,2,15)
555	1369	RS(37^1,2,15)
615	1681	RS(41^1,2,15)
645	1849	RS(43^1,2,15)
705	2209	RS(47^1,2,15)
735	2401	RS(7^2,2,15)
795	2809	RS(53^1,2,15)
841	24389	RS(29^1,3,29)
899	29791	RS(31^1,3,29)
928	32768	RS(2^5,3,29)
1073	50653	RS(37^1,3,29)
1189	68921	RS(41^1,3,29)
1247	79507	RS(43^1,3,29)
1363	103823	RS(47^1,3,29)
1421	117649	RS(7^2,3,29)
1537	148877	RS(53^1,3,29)
1711	205379	RS(59^1,3,29)
1769	226981	RS(61^1,3,29)
1849	3418801	RS(43^1,4,43)
2021	4879681	RS(47^1,4,43)
2107	5764801	RS(7^2,4,43)
2279	7890481	RS(53^1,4,43)
2519	7890737	Add(240,2279)
2534	7890770	Add(255,2279)
2537	12117361	RS(59^1,4,43)
2623	13845841	RS(61^1,4,43)
2752	16777216	RS(2^6,4,43)
2881	20151121	RS(67^1,4,43)
3053	25411681	RS(71^1,4,43)
3139	28398241	RS(73^1,4,43)
3363	714924299	RS(59^1,5,57)
3477	844596301	RS(61^1,5,57)
3648	1073741824	RS(2^6,5,57)
3819	1350125107	RS(67^1,5,57)
4047	1804229351	RS(71^1,5,57)
4161	2073071593	RS(73^1,5,57)
4401	2073071849	Add(240,4161)
4416	2073071882	Add(255,4161)
4446	2073071954	Add(285,4161)
4503	3077056399	RS(79^1,5,57)
4617	3486784401	RS(3^4,5,57)
4731	3939040643	RS(83^1,5,57)
4971	3939040899	Add(240,4731)
4986	3939040932	Add(255,4731)
5016	3939041004	Add(285,4731)
5041	128100283921	RS(71^1,6,71)
5183	151334226289	RS(73^1,6,71)
5423	151334226545	Add(240,5183)
5438	151334226578	Add(255,5183)
5468	151334226650	Add(285,5183)
5528	151334226818	Add(345,5183)
5558	151334226914	Add(375,5183)
5588	151334227018	Add(405,5183)
5609	243087455521	RS(79^1,6,71)
5751	282429536481	RS(3^4,6,71)
5893	326940373369	RS(83^1,6,71)
6133	326940373625	Add(240,5893)
6148	326940373658	Add(255,5893)
6178	326940373730	Add(285,5893)
6238	326940373898	Add(345,5893)
6268	326940373994	Add(375,5893)
6298	326940374098	Add(405,5893)
6319	496981290961	RS(89^1,6,71)
6559	496981291217	Add(240,6319)
6574	496981291250	Add(255,6319)
6604	496981291322	Add(285,6319)
6664	496981291490	Add(345,6319)
6694	496981291586	Add(375,6319)
6724	496981291690	Add(405,6319)
6754	496981291802	Add(435,6319)
6784	496981291922	Add(465,6319)
6799	496981291985	Add(480,6319)
6874	496981292330	Add(555,6319)
6887	832972004929	RS(97^1,6,71)
7127	832972005185	Add(240,6887)
7142	832972005218	Add(255,6887)
7171	1061520150601	RS(101^1,6,71)
7313	1194052296529	RS(103^1,6,71)
7553	1194052296785	Add(240,7313)
7565	10000000000000	RS(89^1,7,85)

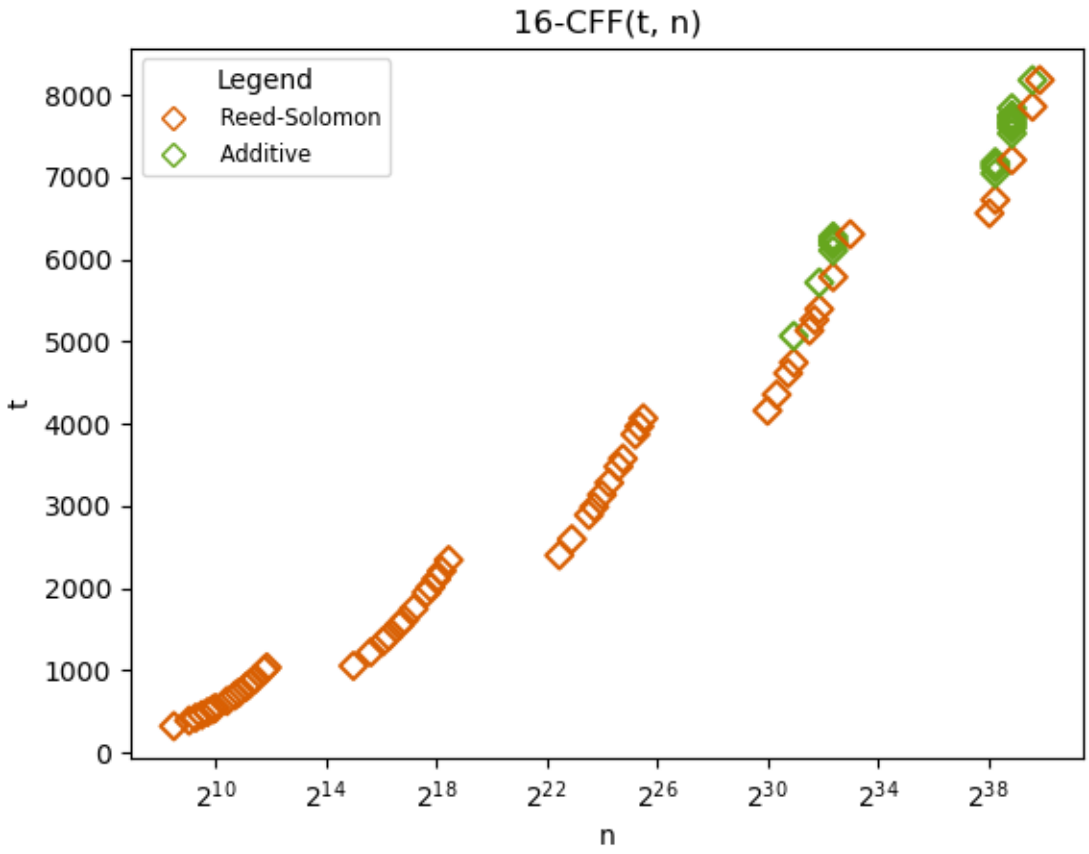
B.10 15-CFF table



t	n	Source
272	289	RS(17^1,2,16)
304	361	RS(19^1,2,16)
368	529	RS(23^1,2,16)
400	625	RS(5^2,2,16)
432	729	RS(3^3,2,16)
464	841	RS(29^1,2,16)
496	961	RS(31^1,2,16)
512	1024	RS(2^5,2,16)
592	1369	RS(37^1,2,16)
656	1681	RS(41^1,2,16)
688	1849	RS(43^1,2,16)
752	2209	RS(47^1,2,16)
784	2401	RS(7^2,2,16)
848	2809	RS(53^1,2,16)
944	3481	RS(59^1,2,16)
961	29791	RS(31^1,3,31)
992	32768	RS(2^5,3,31)
1147	50653	RS(37^1,3,31)
1271	68921	RS(41^1,3,31)
1333	79507	RS(43^1,3,31)
1457	103823	RS(47^1,3,31)
1519	117649	RS(7^2,3,31)
1643	148877	RS(53^1,3,31)
1829	205379	RS(59^1,3,31)
1891	226981	RS(61^1,3,31)
1984	262144	RS(2^6,3,31)
2077	300763	RS(67^1,3,31)
2162	4879681	RS(47^1,4,46)
2254	5764801	RS(7^2,4,46)
2438	7890481	RS(53^1,4,46)
2710	7890770	Add(272,2438)
2714	12117361	RS(59^1,4,46)
2806	13845841	RS(61^1,4,46)
2944	16777216	RS(2^6,4,46)
3082	20151121	RS(67^1,4,46)
3266	25411681	RS(71^1,4,46)
3358	28398241	RS(73^1,4,46)
3630	28398530	Add(272,3358)
3634	38950081	RS(79^1,4,46)
3721	844596301	RS(61^1,5,61)
3904	1073741824	RS(2^6,5,61)
4087	1350125107	RS(67^1,5,61)
4331	1804229351	RS(71^1,5,61)
4453	2073071593	RS(73^1,5,61)
4725	2073071882	Add(272,4453)

t	n	Source
4757	2073071954	Add(304,4453)
4819	3077056399	RS(79^1,5,61)
4941	3486784401	RS(3^4,5,61)
5063	3939040643	RS(83^1,5,61)
5335	3939040932	Add(272,5063)
5367	3939041004	Add(304,5063)
5429	5584059449	RS(89^1,5,61)
5701	5584059738	Add(272,5429)
5733	5584059810	Add(304,5429)
5797	5584059978	Add(368,5429)
5829	5584060074	Add(400,5429)
5861	5584060178	Add(432,5429)
5893	5584060290	Add(464,5429)
5917	8587340257	RS(97^1,5,61)
6004	243087455521	RS(79^1,6,76)
6156	282429536481	RS(3^4,6,76)
6308	326940373369	RS(83^1,6,76)
6580	326940373658	Add(272,6308)
6612	326940373730	Add(304,6308)
6676	326940373898	Add(368,6308)
6708	326940373994	Add(400,6308)
6740	326940374098	Add(432,6308)
6764	496981290961	RS(89^1,6,76)
7036	496981291250	Add(272,6764)
7068	496981291322	Add(304,6764)
7132	496981291490	Add(368,6764)
7164	496981291586	Add(400,6764)
7196	496981291690	Add(432,6764)
7228	496981291802	Add(464,6764)
7260	496981291922	Add(496,6764)
7276	496981291985	Add(512,6764)
7356	496981292330	Add(592,6764)
7372	832972004929	RS(97^1,6,76)
7644	832972005218	Add(272,7372)
7676	1061520150601	RS(101^1,6,76)
7828	1194052296529	RS(103^1,6,76)
8100	1194052296818	Add(272,7828)
8132	1500730351849	RS(107^1,6,76)
8284	1677100110841	RS(109^1,6,76)
8556	1677100111130	Add(272,8284)
8588	2081951752609	RS(113^1,6,76)
8827	10000000000000	RS(97^1,7,91)

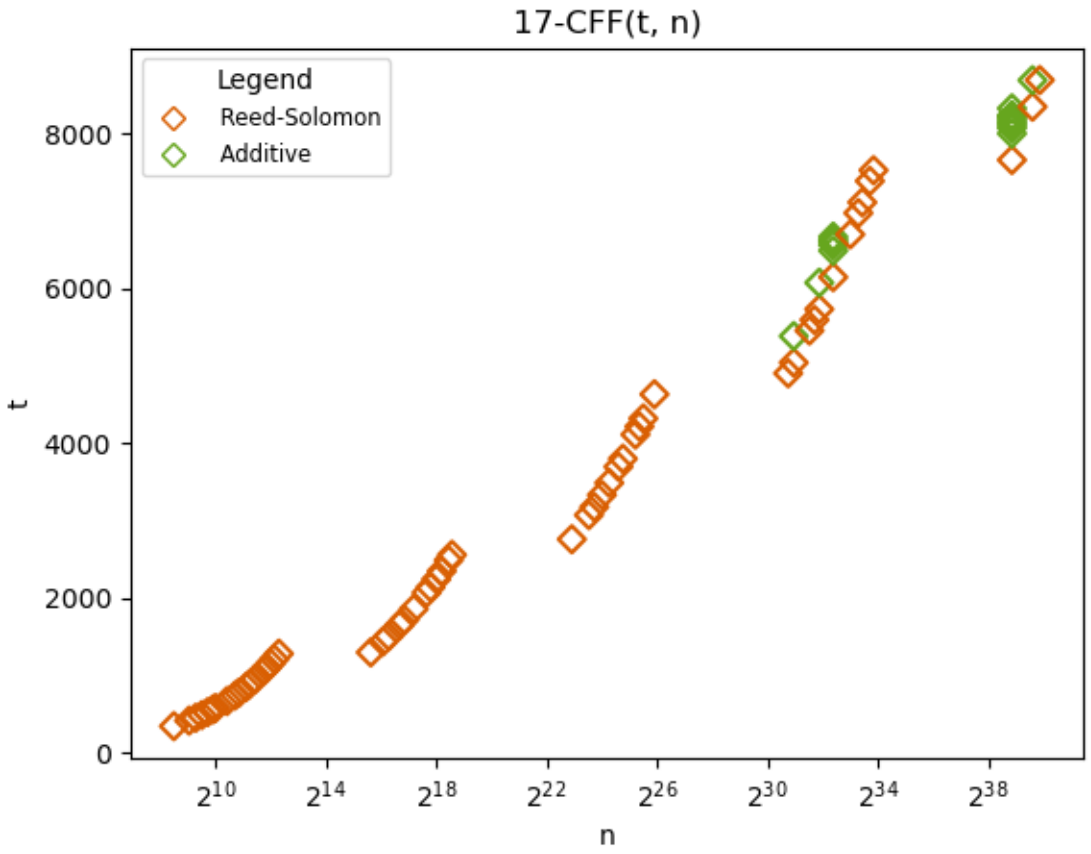
B.11 16-CFF table



t	n	Source
323	361	RS(19^1,2,17)
391	529	RS(23^1,2,17)
425	625	RS(5^2,2,17)
459	729	RS(3^3,2,17)
493	841	RS(29^1,2,17)
527	961	RS(31^1,2,17)
544	1024	RS(2^5,2,17)
629	1369	RS(37^1,2,17)
697	1681	RS(41^1,2,17)
731	1849	RS(43^1,2,17)
799	2209	RS(47^1,2,17)
833	2401	RS(7^2,2,17)
901	2809	RS(53^1,2,17)
1003	3481	RS(59^1,2,17)
1037	3721	RS(61^1,2,17)
1056	32768	RS(2^5,3,33)
1221	50653	RS(37^1,3,33)
1353	68921	RS(41^1,3,33)
1419	79507	RS(43^1,3,33)
1551	103823	RS(47^1,3,33)
1617	117649	RS(7^2,3,33)
1749	148877	RS(53^1,3,33)
1947	205379	RS(59^1,3,33)
2013	226981	RS(61^1,3,33)
2112	262144	RS(2^6,3,33)
2211	300763	RS(67^1,3,33)
2343	357911	RS(71^1,3,33)
2401	5764801	RS(7^2,4,49)
2597	7890481	RS(53^1,4,49)
2891	12117361	RS(59^1,4,49)
2989	13845841	RS(61^1,4,49)
3136	16777216	RS(2^6,4,49)
3283	20151121	RS(67^1,4,49)
3479	25411681	RS(71^1,4,49)
3577	28398241	RS(73^1,4,49)
3871	38950081	RS(79^1,4,49)
3969	43046721	RS(3^4,4,49)
4067	47458321	RS(83^1,4,49)
4160	1073741824	RS(2^6,5,65)
4355	1350125107	RS(67^1,5,65)
4615	1804229351	RS(71^1,5,65)
4745	2073071593	RS(73^1,5,65)
5068	2073071954	Add(323,4745)
5135	3077056399	RS(79^1,5,65)
5265	3486784401	RS(3^4,5,65)

t	n	Source
5395	3939040643	RS(83^1,5,65)
5718	3939041004	Add(323,5395)
5785	5584059449	RS(89^1,5,65)
6108	5584059810	Add(323,5785)
6176	5584059978	Add(391,5785)
6210	5584060074	Add(425,5785)
6244	5584060178	Add(459,5785)
6278	5584060290	Add(493,5785)
6305	8587340257	RS(97^1,5,65)
6561	282429536481	RS(3^4,6,81)
6723	326940373369	RS(83^1,6,81)
7046	326940373730	Add(323,6723)
7114	326940373898	Add(391,6723)
7148	326940373994	Add(425,6723)
7182	326940374098	Add(459,6723)
7209	496981290961	RS(89^1,6,81)
7532	496981291322	Add(323,7209)
7600	496981291490	Add(391,7209)
7634	496981291586	Add(425,7209)
7668	496981291690	Add(459,7209)
7702	496981291802	Add(493,7209)
7736	496981291922	Add(527,7209)
7753	496981291985	Add(544,7209)
7838	496981292330	Add(629,7209)
7857	832972004929	RS(97^1,6,81)
8180	832972005290	Add(323,7857)
8181	1061520150601	RS(101^1,6,81)
8343	1194052296529	RS(103^1,6,81)
8666	1194052296890	Add(323,8343)
8667	1500730351849	RS(107^1,6,81)
8829	1677100110841	RS(109^1,6,81)
9152	1677100111202	Add(323,8829)
9153	2081951752609	RS(113^1,6,81)
9409	10000000000000	RS(97^1,7,97)

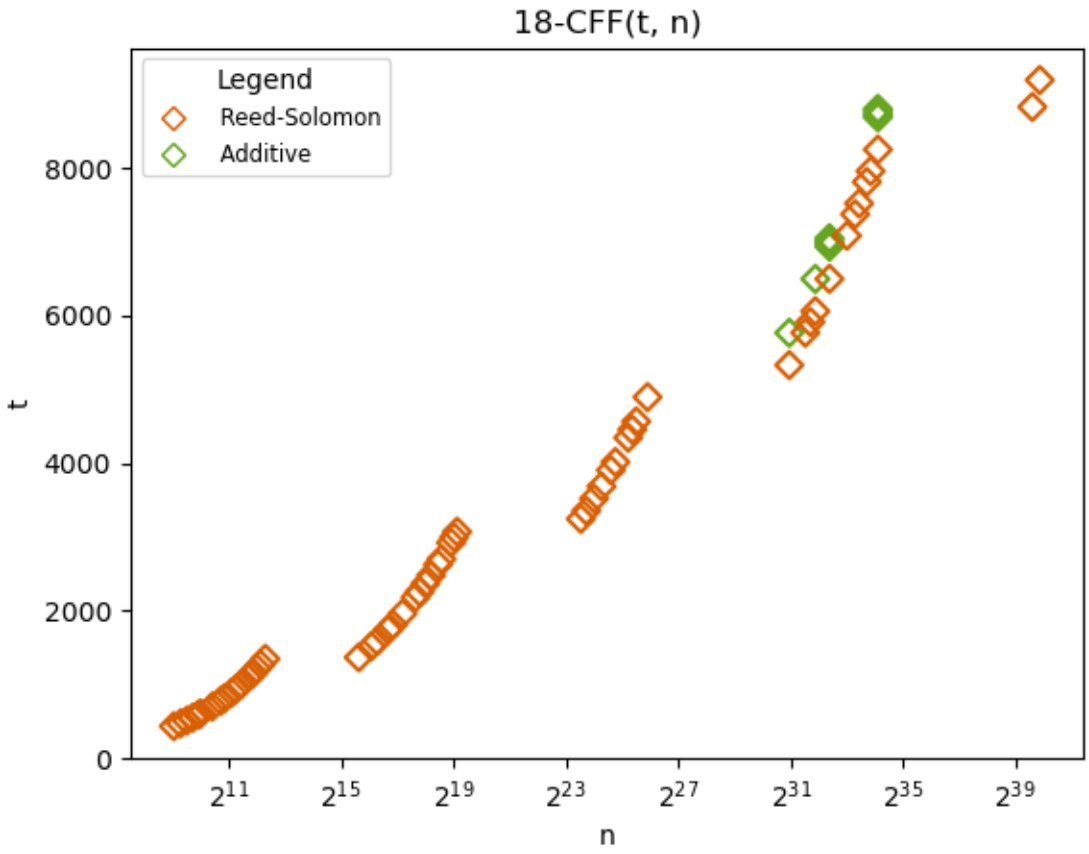
B.12 17-CFF table



t	n	Source
342	361	RS(19^1,2,18)
414	529	RS(23^1,2,18)
450	625	RS(5^2,2,18)
486	729	RS(3^3,2,18)
522	841	RS(29^1,2,18)
558	961	RS(31^1,2,18)
576	1024	RS(2^5,2,18)
666	1369	RS(37^1,2,18)
738	1681	RS(41^1,2,18)
774	1849	RS(43^1,2,18)
846	2209	RS(47^1,2,18)
882	2401	RS(7^2,2,18)
954	2809	RS(53^1,2,18)
1062	3481	RS(59^1,2,18)
1098	3721	RS(61^1,2,18)
1152	4096	RS(2^6,2,18)
1206	4489	RS(67^1,2,18)
1278	5041	RS(71^1,2,18)
1295	50653	RS(37^1,3,35)
1435	68921	RS(41^1,3,35)
1505	79507	RS(43^1,3,35)
1645	103823	RS(47^1,3,35)
1715	117649	RS(7^2,3,35)
1855	148877	RS(53^1,3,35)
2065	205379	RS(59^1,3,35)
2135	226981	RS(61^1,3,35)
2240	262144	RS(2^6,3,35)
2345	300763	RS(67^1,3,35)
2485	357911	RS(71^1,3,35)
2555	389017	RS(73^1,3,35)
2756	7890481	RS(53^1,4,52)
3068	12117361	RS(59^1,4,52)
3172	13845841	RS(61^1,4,52)
3328	16777216	RS(2^6,4,52)
3484	20151121	RS(67^1,4,52)
3692	25411681	RS(71^1,4,52)
3796	28398241	RS(73^1,4,52)
4108	38950081	RS(79^1,4,52)
4212	43046721	RS(3^4,4,52)
4316	47458321	RS(83^1,4,52)
4628	62742241	RS(89^1,4,52)
4899	1804229351	RS(71^1,5,69)
5037	2073071593	RS(73^1,5,69)
5379	2073071954	Add(342,5037)
5451	3077056399	RS(79^1,5,69)

t	n	Source
5589	3486784401	RS(3^4,5,69)
5727	3939040643	RS(83^1,5,69)
6069	3939041004	Add(342,5727)
6141	5584059449	RS(89^1,5,69)
6483	5584059810	Add(342,6141)
6555	5584059978	Add(414,6141)
6591	5584060074	Add(450,6141)
6627	5584060178	Add(486,6141)
6663	5584060290	Add(522,6141)
6693	8587340257	RS(97^1,5,69)
6969	10510100501	RS(101^1,5,69)
7107	11592740743	RS(103^1,5,69)
7383	14025517307	RS(107^1,5,69)
7521	15386239549	RS(109^1,5,69)
7654	496981290961	RS(89^1,6,86)
7996	496981291322	Add(342,7654)
8068	496981291490	Add(414,7654)
8104	496981291586	Add(450,7654)
8140	496981291690	Add(486,7654)
8176	496981291802	Add(522,7654)
8212	496981291922	Add(558,7654)
8230	496981291985	Add(576,7654)
8320	496981292330	Add(666,7654)
8342	832972004929	RS(97^1,6,86)
8684	832972005290	Add(342,8342)
8686	1061520150601	RS(101^1,6,86)
8858	1194052296529	RS(103^1,6,86)
9200	1194052296890	Add(342,8858)
9202	1500730351849	RS(107^1,6,86)
9374	1677100110841	RS(109^1,6,86)
9716	1677100111202	Add(342,9374)
9718	2081951752609	RS(113^1,6,86)
10060	2081951752970	Add(342,9718)
10132	2081951753138	Add(414,9718)
10168	2081951753234	Add(450,9718)
10204	2081951753338	Add(486,9718)
10240	2081951753450	Add(522,9718)
10276	2081951753570	Add(558,9718)
10294	2081951753633	Add(576,9718)
10384	2081951753978	Add(666,9718)
10406	3138428376721	RS(11^2,6,86)
10609	10000000000000	RS(103^1,7,103)

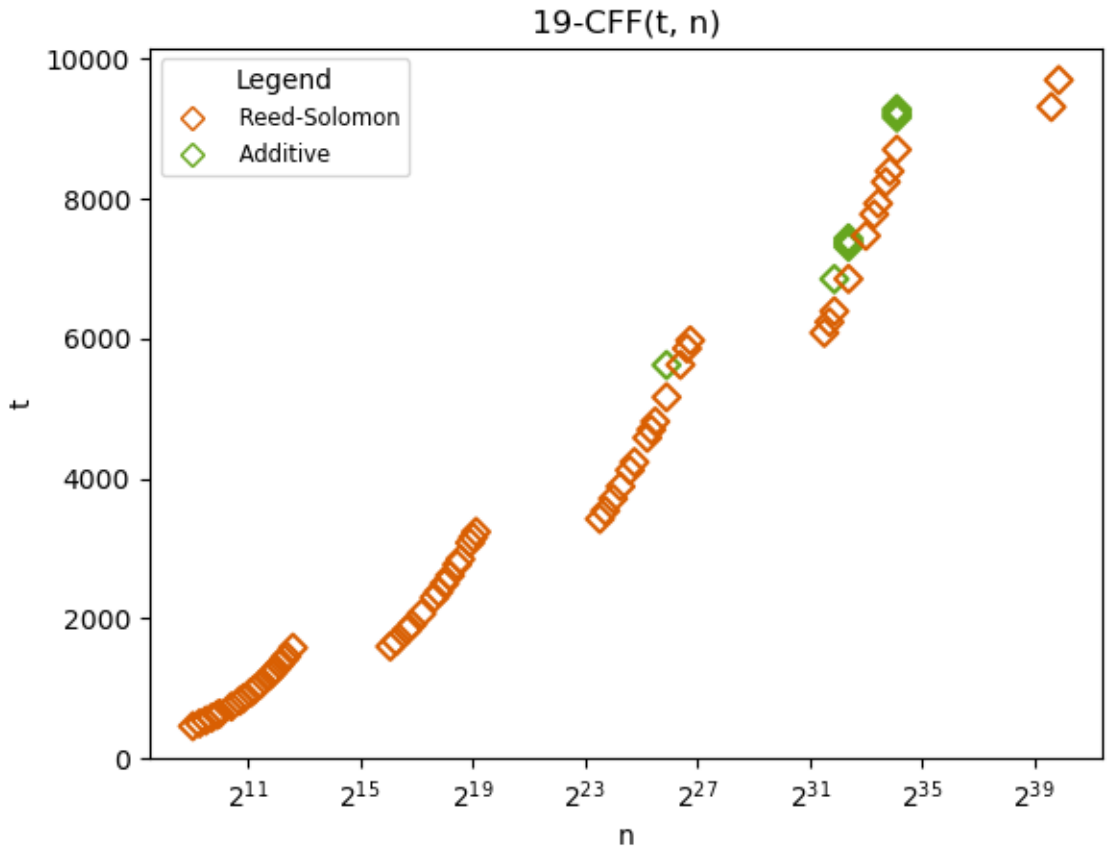
B.13 18-CFF table



t	n	Source
437	529	RS(23^1,2,19)
475	625	RS(5^2,2,19)
513	729	RS(3^3,2,19)
551	841	RS(29^1,2,19)
589	961	RS(31^1,2,19)
608	1024	RS(2^5,2,19)
703	1369	RS(37^1,2,19)
779	1681	RS(41^1,2,19)
817	1849	RS(43^1,2,19)
893	2209	RS(47^1,2,19)
931	2401	RS(7^2,2,19)
1007	2809	RS(53^1,2,19)
1121	3481	RS(59^1,2,19)
1159	3721	RS(61^1,2,19)
1216	4096	RS(2^6,2,19)
1273	4489	RS(67^1,2,19)
1349	5041	RS(71^1,2,19)
1369	50653	RS(37^1,3,37)
1517	68921	RS(41^1,3,37)
1591	79507	RS(43^1,3,37)
1739	103823	RS(47^1,3,37)
1813	117649	RS(7^2,3,37)
1961	148877	RS(53^1,3,37)
2183	205379	RS(59^1,3,37)
2257	226981	RS(61^1,3,37)
2368	262144	RS(2^6,3,37)
2479	300763	RS(67^1,3,37)
2627	357911	RS(71^1,3,37)
2701	389017	RS(73^1,3,37)
2923	493039	RS(79^1,3,37)
2997	531441	RS(3^4,3,37)
3071	571787	RS(83^1,3,37)
3245	12117361	RS(59^1,4,55)
3355	13845841	RS(61^1,4,55)
3520	16777216	RS(2^6,4,55)
3685	20151121	RS(67^1,4,55)
3905	25411681	RS(71^1,4,55)
4015	28398241	RS(73^1,4,55)
4345	38950081	RS(79^1,4,55)
4455	43046721	RS(3^4,4,55)
4565	47458321	RS(83^1,4,55)
4895	62742241	RS(89^1,4,55)
5329	2073071593	RS(73^1,5,73)
5766	2073072122	Add(437,5329)
5767	3077056399	RS(79^1,5,73)

t	n	Source
5913	3486784401	RS(3^4,5,73)
6059	3939040643	RS(83^1,5,73)
6496	3939041172	Add(437,6059)
6497	5584059449	RS(89^1,5,73)
6934	5584059978	Add(437,6497)
6972	5584060074	Add(475,6497)
7010	5584060178	Add(513,6497)
7048	5584060290	Add(551,6497)
7081	8587340257	RS(97^1,5,73)
7373	10510100501	RS(101^1,5,73)
7519	11592740743	RS(103^1,5,73)
7811	14025517307	RS(107^1,5,73)
7957	15386239549	RS(109^1,5,73)
8249	18424351793	RS(113^1,5,73)
8686	18424352322	Add(437,8249)
8724	18424352418	Add(475,8249)
8762	18424352522	Add(513,8249)
8800	18424352634	Add(551,8249)
8827	832972004929	RS(97^1,6,91)
9191	1061520150601	RS(101^1,6,91)
9373	1194052296529	RS(103^1,6,91)
9737	1500730351849	RS(107^1,6,91)
9919	1677100110841	RS(109^1,6,91)
10283	2081951752609	RS(113^1,6,91)
10720	2081951753138	Add(437,10283)
10758	2081951753234	Add(475,10283)
10796	2081951753338	Add(513,10283)
10834	2081951753450	Add(551,10283)
10872	2081951753570	Add(589,10283)
10891	2081951753633	Add(608,10283)
10986	2081951753978	Add(703,10283)
11011	3138428376721	RS(11^2,6,91)
11375	3814697265625	RS(5^3,6,91)
11557	4195872914689	RS(127^1,6,91)
11648	4398046511104	RS(2^7,6,91)
11881	10000000000000	RS(109^1,7,109)

B.14 19-CFF table

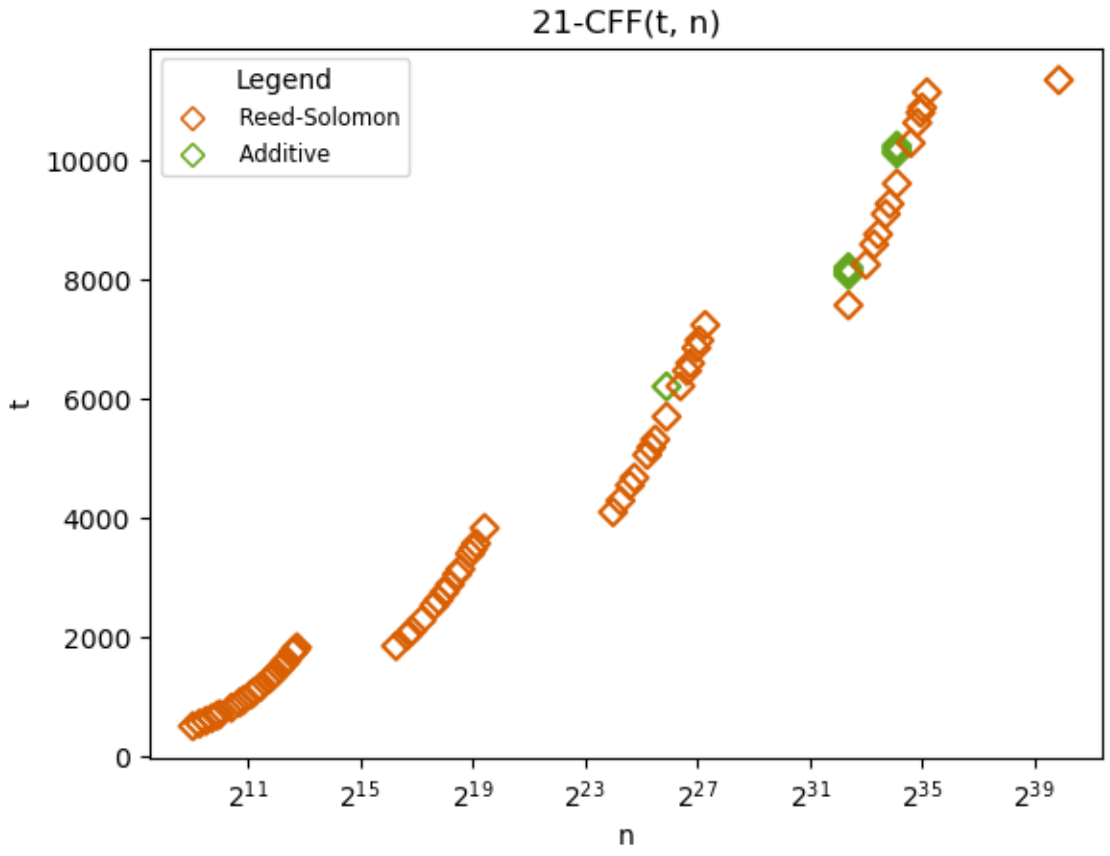


t	n	Source
460	529	RS(23^1,2,20)
500	625	RS(5^2,2,20)
540	729	RS(3^3,2,20)
580	841	RS(29^1,2,20)
620	961	RS(31^1,2,20)
640	1024	RS(2^5,2,20)
740	1369	RS(37^1,2,20)
820	1681	RS(41^1,2,20)
860	1849	RS(43^1,2,20)
940	2209	RS(47^1,2,20)
980	2401	RS(7^2,2,20)
1060	2809	RS(53^1,2,20)
1180	3481	RS(59^1,2,20)
1220	3721	RS(61^1,2,20)
1280	4096	RS(2^6,2,20)
1340	4489	RS(67^1,2,20)
1420	5041	RS(71^1,2,20)
1460	5329	RS(73^1,2,20)
1580	6241	RS(79^1,2,20)
1599	68921	RS(41^1,3,39)
1677	79507	RS(43^1,3,39)
1833	103823	RS(47^1,3,39)
1911	117649	RS(7^2,3,39)
2067	148877	RS(53^1,3,39)
2301	205379	RS(59^1,3,39)
2379	226981	RS(61^1,3,39)
2496	262144	RS(2^6,3,39)
2613	300763	RS(67^1,3,39)
2769	357911	RS(71^1,3,39)
2847	389017	RS(73^1,3,39)
3081	493039	RS(79^1,3,39)
3159	531441	RS(3^4,3,39)
3237	571787	RS(83^1,3,39)
3422	12117361	RS(59^1,4,58)
3538	13845841	RS(61^1,4,58)
3712	16777216	RS(2^6,4,58)
3886	20151121	RS(67^1,4,58)
4118	25411681	RS(71^1,4,58)
4234	28398241	RS(73^1,4,58)
4582	38950081	RS(79^1,4,58)
4698	43046721	RS(3^4,4,58)
4814	47458321	RS(83^1,4,58)
5162	62742241	RS(89^1,4,58)
5622	62742770	Add(460,5162)
5626	88529281	RS(97^1,4,58)

t	n	Source
5858	104060401	RS(101^1,4,58)
5974	112550881	RS(103^1,4,58)
6083	3077056399	RS(79^1,5,77)
6237	3486784401	RS(3^4,5,77)
6391	3939040643	RS(83^1,5,77)
6851	3939041172	Add(460,6391)
6853	5584059449	RS(89^1,5,77)
7313	5584059978	Add(460,6853)
7353	5584060074	Add(500,6853)
7393	5584060178	Add(540,6853)
7433	5584060290	Add(580,6853)
7469	8587340257	RS(97^1,5,77)
7777	10510100501	RS(101^1,5,77)
7931	11592740743	RS(103^1,5,77)
8239	14025517307	RS(107^1,5,77)
8393	15386239549	RS(109^1,5,77)
8701	18424351793	RS(113^1,5,77)
9161	18424352322	Add(460,8701)
9201	18424352418	Add(500,8701)
9241	18424352522	Add(540,8701)
9281	18424352634	Add(580,8701)
9312	832972004929	RS(97^1,6,96)
9696	1061520150601	RS(101^1,6,96)
9888	1194052296529	RS(103^1,6,96)
10272	1500730351849	RS(107^1,6,96)
10464	1677100110841	RS(109^1,6,96)
10848	2081951752609	RS(113^1,6,96)
11308	2081951753138	Add(460,10848)
11348	2081951753234	Add(500,10848)
11388	2081951753338	Add(540,10848)
11428	2081951753450	Add(580,10848)
11468	2081951753570	Add(620,10848)
11488	2081951753633	Add(640,10848)
11588	2081951753978	Add(740,10848)
11616	3138428376721	RS(11^2,6,96)
12000	3814697265625	RS(5^3,6,96)
12192	4195872914689	RS(127^1,6,96)
12288	4398046511104	RS(2^7,6,96)
12576	5053913144281	RS(131^1,6,96)
13036	5053913144810	Add(460,12576)
13076	5053913144906	Add(500,12576)
13116	5053913145010	Add(540,12576)
13152	6611856250609	RS(137^1,6,96)
13344	7212549413161	RS(139^1,6,96)
13804	7212549413690	Add(460,13344)

t	n	Source
13844	7212549413786	Add(500,13344)
13884	7212549413890	Add(540,13344)
13915	10000000000000	RS(11^2,7,115)

B.15 21-CFF table

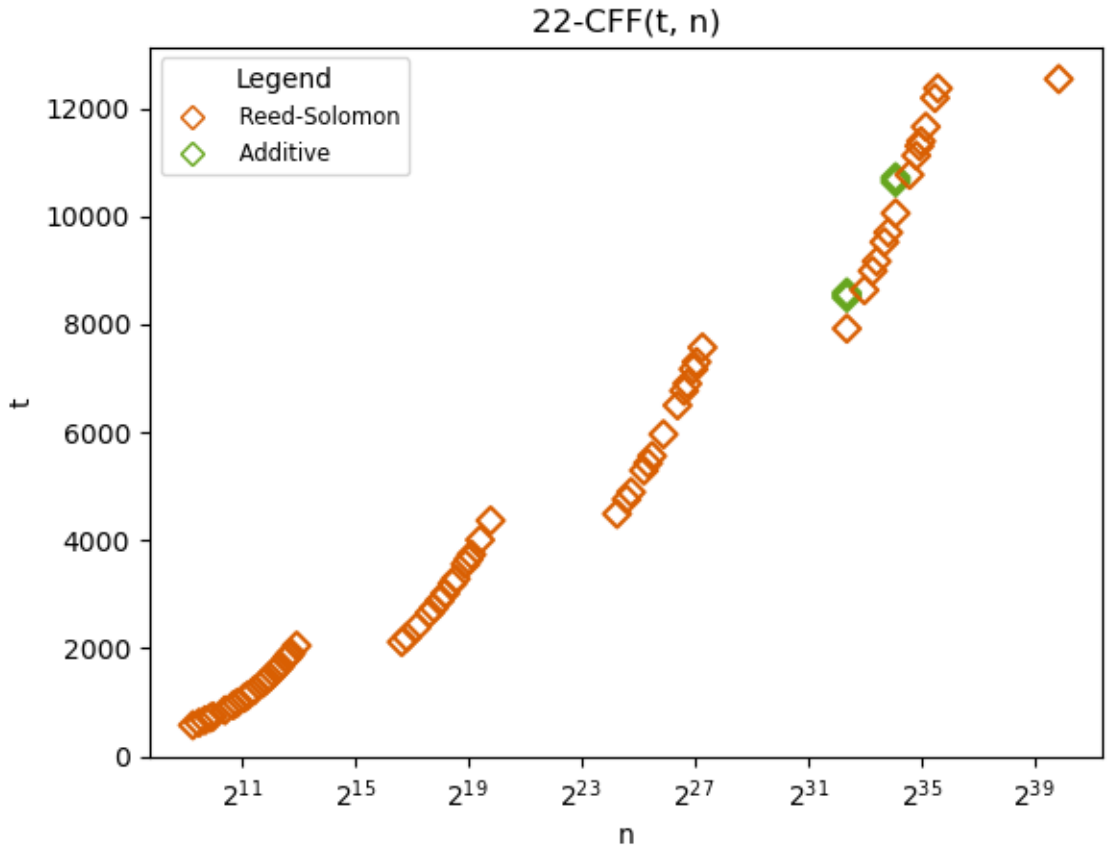


t	n	Source
506	529	RS(23^1,2,22)
550	625	RS(5^2,2,22)
594	729	RS(3^3,2,22)
638	841	RS(29^1,2,22)
682	961	RS(31^1,2,22)
704	1024	RS(2^5,2,22)
814	1369	RS(37^1,2,22)
902	1681	RS(41^1,2,22)
946	1849	RS(43^1,2,22)
1034	2209	RS(47^1,2,22)
1078	2401	RS(7^2,2,22)
1166	2809	RS(53^1,2,22)
1298	3481	RS(59^1,2,22)
1342	3721	RS(61^1,2,22)
1408	4096	RS(2^6,2,22)
1474	4489	RS(67^1,2,22)
1562	5041	RS(71^1,2,22)
1606	5329	RS(73^1,2,22)
1738	6241	RS(79^1,2,22)
1782	6561	RS(3^4,2,22)
1826	6889	RS(83^1,2,22)
1849	79507	RS(43^1,3,43)
2021	103823	RS(47^1,3,43)
2107	117649	RS(7^2,3,43)
2279	148877	RS(53^1,3,43)
2537	205379	RS(59^1,3,43)
2623	226981	RS(61^1,3,43)
2752	262144	RS(2^6,3,43)
2881	300763	RS(67^1,3,43)
3053	357911	RS(71^1,3,43)
3139	389017	RS(73^1,3,43)
3397	493039	RS(79^1,3,43)
3483	531441	RS(3^4,3,43)
3569	571787	RS(83^1,3,43)
3827	704969	RS(89^1,3,43)
4096	16777216	RS(2^6,4,64)
4288	20151121	RS(67^1,4,64)
4544	25411681	RS(71^1,4,64)
4672	28398241	RS(73^1,4,64)
5056	38950081	RS(79^1,4,64)
5184	43046721	RS(3^4,4,64)
5312	47458321	RS(83^1,4,64)
5696	62742241	RS(89^1,4,64)
6202	62742770	Add(506,5696)
6208	88529281	RS(97^1,4,64)

t	n	Source
6464	104060401	RS(101^1,4,64)
6592	112550881	RS(103^1,4,64)
6848	131079601	RS(107^1,4,64)
6976	141158161	RS(109^1,4,64)
7232	163047361	RS(113^1,4,64)
7565	5584059449	RS(89^1,5,85)
8071	5584059978	Add(506,7565)
8115	5584060074	Add(550,7565)
8159	5584060178	Add(594,7565)
8203	5584060290	Add(638,7565)
8245	8587340257	RS(97^1,5,85)
8585	10510100501	RS(101^1,5,85)
8755	11592740743	RS(103^1,5,85)
9095	14025517307	RS(107^1,5,85)
9265	15386239549	RS(109^1,5,85)
9605	18424351793	RS(113^1,5,85)
10111	18424352322	Add(506,9605)
10155	18424352418	Add(550,9605)
10199	18424352522	Add(594,9605)
10243	18424352634	Add(638,9605)
10285	25937424601	RS(11^2,5,85)
10625	30517578125	RS(5^3,5,85)
10795	33038369407	RS(127^1,5,85)
10880	34359738368	RS(2^7,5,85)
11135	38579489651	RS(131^1,5,85)
11342	1500730351849	RS(107^1,6,106)
11554	1677100110841	RS(109^1,6,106)
11978	2081951752609	RS(113^1,6,106)
12484	2081951753138	Add(506,11978)
12528	2081951753234	Add(550,11978)
12572	2081951753338	Add(594,11978)
12616	2081951753450	Add(638,11978)
12660	2081951753570	Add(682,11978)
12682	2081951753633	Add(704,11978)
12792	2081951753978	Add(814,11978)
12826	3138428376721	RS(11^2,6,106)
13250	3814697265625	RS(5^3,6,106)
13462	4195872914689	RS(127^1,6,106)
13568	4398046511104	RS(2^7,6,106)
13886	5053913144281	RS(131^1,6,106)
14392	5053913144810	Add(506,13886)
14436	5053913144906	Add(550,13886)
14480	5053913145010	Add(594,13886)
14522	6611856250609	RS(137^1,6,106)
14734	7212549413161	RS(139^1,6,106)

t	n	Source
15240	7212549413690	Add(506,14734)
15284	7212549413786	Add(550,14734)
15328	7212549413890	Add(594,14734)
15372	7212549414002	Add(638,14734)
15416	7212549414122	Add(682,14734)
15438	7212549414185	Add(704,14734)
15548	7212549414530	Add(814,14734)
15636	7212549414842	Add(902,14734)
15680	7212549415010	Add(946,14734)
15768	7212549415370	Add(1034,14734)
15794	10000000000000	RS(149^1,6,106)

B.16 22-CFF table

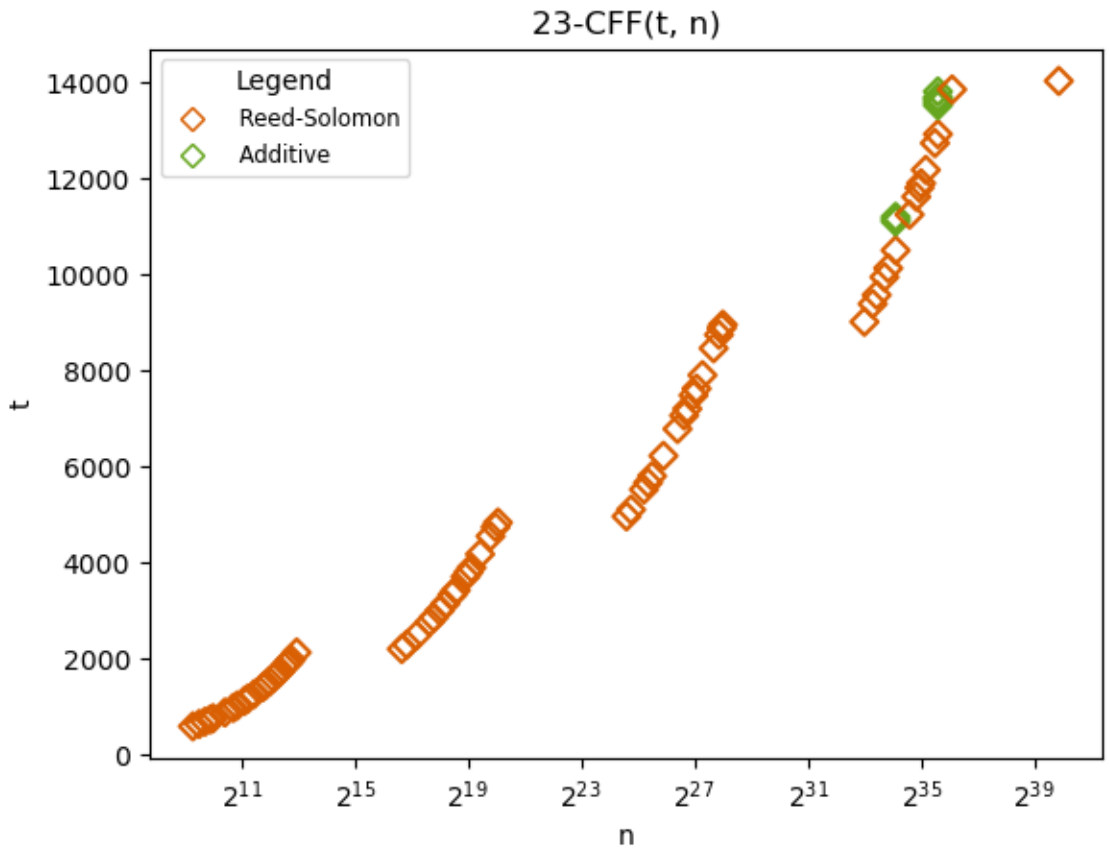


t	n	Source
575	625	RS(5^2,2,23)
621	729	RS(3^3,2,23)
667	841	RS(29^1,2,23)
713	961	RS(31^1,2,23)
736	1024	RS(2^5,2,23)
851	1369	RS(37^1,2,23)
943	1681	RS(41^1,2,23)
989	1849	RS(43^1,2,23)
1081	2209	RS(47^1,2,23)
1127	2401	RS(7^2,2,23)
1219	2809	RS(53^1,2,23)
1357	3481	RS(59^1,2,23)
1403	3721	RS(61^1,2,23)
1472	4096	RS(2^6,2,23)
1541	4489	RS(67^1,2,23)
1633	5041	RS(71^1,2,23)
1679	5329	RS(73^1,2,23)
1817	6241	RS(79^1,2,23)
1863	6561	RS(3^4,2,23)
1909	6889	RS(83^1,2,23)
2047	7921	RS(89^1,2,23)
2115	103823	RS(47^1,3,45)
2205	117649	RS(7^2,3,45)
2385	148877	RS(53^1,3,45)
2655	205379	RS(59^1,3,45)
2745	226981	RS(61^1,3,45)
2880	262144	RS(2^6,3,45)
3015	300763	RS(67^1,3,45)
3195	357911	RS(71^1,3,45)
3285	389017	RS(73^1,3,45)
3555	493039	RS(79^1,3,45)
3645	531441	RS(3^4,3,45)
3735	571787	RS(83^1,3,45)
4005	704969	RS(89^1,3,45)
4365	912673	RS(97^1,3,45)
4489	20151121	RS(67^1,4,67)
4757	25411681	RS(71^1,4,67)
4891	28398241	RS(73^1,4,67)
5293	38950081	RS(79^1,4,67)
5427	43046721	RS(3^4,4,67)
5561	47458321	RS(83^1,4,67)
5963	62742241	RS(89^1,4,67)
6499	88529281	RS(97^1,4,67)
6767	104060401	RS(101^1,4,67)
6901	112550881	RS(103^1,4,67)

t	n	Source
7169	131079601	RS(107^1,4,67)
7303	141158161	RS(109^1,4,67)
7571	163047361	RS(113^1,4,67)
7921	5584059449	RS(89^1,5,89)
8496	5584060074	Add(575,7921)
8542	5584060178	Add(621,7921)
8588	5584060290	Add(667,7921)
8633	8587340257	RS(97^1,5,89)
8989	10510100501	RS(101^1,5,89)
9167	11592740743	RS(103^1,5,89)
9523	14025517307	RS(107^1,5,89)
9701	15386239549	RS(109^1,5,89)
10057	18424351793	RS(113^1,5,89)
10632	18424352418	Add(575,10057)
10678	18424352522	Add(621,10057)
10724	18424352634	Add(667,10057)
10769	25937424601	RS(11^2,5,89)
11125	30517578125	RS(5^3,5,89)
11303	33038369407	RS(127^1,5,89)
11392	34359738368	RS(2^7,5,89)
11659	38579489651	RS(131^1,5,89)
12193	48261724457	RS(137^1,5,89)
12371	51888844699	RS(139^1,5,89)
12543	2081951752609	RS(113^1,6,111)
13118	2081951753234	Add(575,12543)
13164	2081951753338	Add(621,12543)
13210	2081951753450	Add(667,12543)
13256	2081951753570	Add(713,12543)
13279	2081951753633	Add(736,12543)
13394	2081951753978	Add(851,12543)
13431	3138428376721	RS(11^2,6,111)
13875	3814697265625	RS(5^3,6,111)
14097	4195872914689	RS(127^1,6,111)
14208	4398046511104	RS(2^7,6,111)
14541	5053913144281	RS(131^1,6,111)
15116	5053913144906	Add(575,14541)
15162	5053913145010	Add(621,14541)
15207	6611856250609	RS(137^1,6,111)
15429	7212549413161	RS(139^1,6,111)
16004	7212549413786	Add(575,15429)
16050	7212549413890	Add(621,15429)
16096	7212549414002	Add(667,15429)
16142	7212549414122	Add(713,15429)
16165	7212549414185	Add(736,15429)
16280	7212549414530	Add(851,15429)

t	n	Source
16372	7212549414842	Add(943,15429)
16418	7212549415010	Add(989,15429)
16510	7212549415370	Add(1081,15429)
16539	10000000000000	RS(149^1,6,111)

B.17 23-CFF table

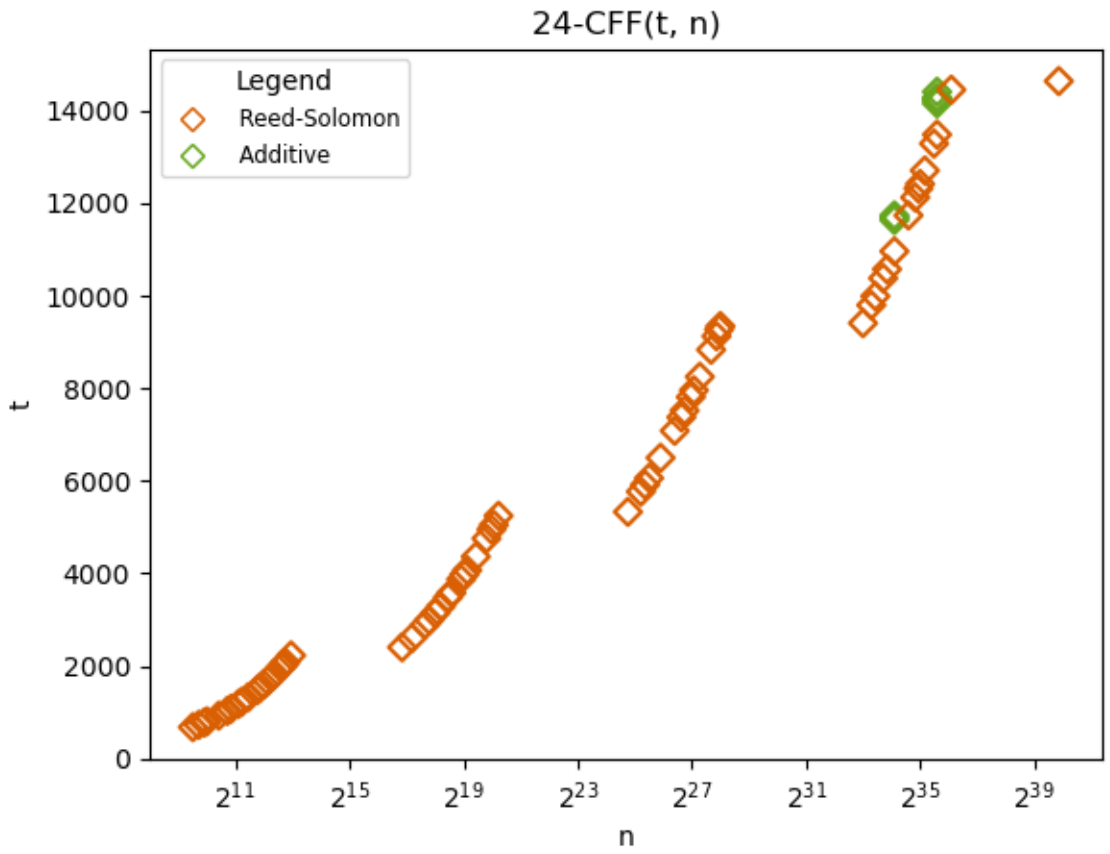


t	n	Source
600	625	RS(5^2,2,24)
648	729	RS(3^3,2,24)
696	841	RS(29^1,2,24)
744	961	RS(31^1,2,24)
768	1024	RS(2^5,2,24)
888	1369	RS(37^1,2,24)
984	1681	RS(41^1,2,24)
1032	1849	RS(43^1,2,24)
1128	2209	RS(47^1,2,24)
1176	2401	RS(7^2,2,24)
1272	2809	RS(53^1,2,24)
1416	3481	RS(59^1,2,24)
1464	3721	RS(61^1,2,24)
1536	4096	RS(2^6,2,24)
1608	4489	RS(67^1,2,24)
1704	5041	RS(71^1,2,24)
1752	5329	RS(73^1,2,24)
1896	6241	RS(79^1,2,24)
1944	6561	RS(3^4,2,24)
1992	6889	RS(83^1,2,24)
2136	7921	RS(89^1,2,24)
2209	103823	RS(47^1,3,47)
2303	117649	RS(7^2,3,47)
2491	148877	RS(53^1,3,47)
2773	205379	RS(59^1,3,47)
2867	226981	RS(61^1,3,47)
3008	262144	RS(2^6,3,47)
3149	300763	RS(67^1,3,47)
3337	357911	RS(71^1,3,47)
3431	389017	RS(73^1,3,47)
3713	493039	RS(79^1,3,47)
3807	531441	RS(3^4,3,47)
3901	571787	RS(83^1,3,47)
4183	704969	RS(89^1,3,47)
4559	912673	RS(97^1,3,47)
4747	1030301	RS(101^1,3,47)
4841	1092727	RS(103^1,3,47)
4970	25411681	RS(71^1,4,70)
5110	28398241	RS(73^1,4,70)
5530	38950081	RS(79^1,4,70)
5670	43046721	RS(3^4,4,70)
5810	47458321	RS(83^1,4,70)
6230	62742241	RS(89^1,4,70)
6790	88529281	RS(97^1,4,70)
7070	104060401	RS(101^1,4,70)

t	n	Source
7210	112550881	RS(103^1,4,70)
7490	131079601	RS(107^1,4,70)
7630	141158161	RS(109^1,4,70)
7910	163047361	RS(113^1,4,70)
8470	214358881	RS(11^2,4,70)
8750	244140625	RS(5^3,4,70)
8890	260144641	RS(127^1,4,70)
8960	268435456	RS(2^7,4,70)
9021	8587340257	RS(97^1,5,93)
9393	10510100501	RS(101^1,5,93)
9579	11592740743	RS(103^1,5,93)
9951	14025517307	RS(107^1,5,93)
10137	15386239549	RS(109^1,5,93)
10509	18424351793	RS(113^1,5,93)
11109	18424352418	Add(600,10509)
11157	18424352522	Add(648,10509)
11205	18424352634	Add(696,10509)
11253	25937424601	RS(11^2,5,93)
11625	30517578125	RS(5^3,5,93)
11811	33038369407	RS(127^1,5,93)
11904	34359738368	RS(2^7,5,93)
12183	38579489651	RS(131^1,5,93)
12741	48261724457	RS(137^1,5,93)
12927	51888844699	RS(139^1,5,93)
13527	51888845324	Add(600,12927)
13575	51888845428	Add(648,12927)
13623	51888845540	Add(696,12927)
13671	51888845660	Add(744,12927)
13695	51888845723	Add(768,12927)
13815	51888846068	Add(888,12927)
13857	73439775749	RS(149^1,5,93)
14036	3138428376721	RS(11^2,6,116)
14500	3814697265625	RS(5^3,6,116)
14732	4195872914689	RS(127^1,6,116)
14848	4398046511104	RS(2^7,6,116)
15196	5053913144281	RS(131^1,6,116)
15796	5053913144906	Add(600,15196)
15844	5053913145010	Add(648,15196)
15892	6611856250609	RS(137^1,6,116)
16124	7212549413161	RS(139^1,6,116)
16724	7212549413786	Add(600,16124)
16772	7212549413890	Add(648,16124)
16820	7212549414002	Add(696,16124)
16868	7212549414122	Add(744,16124)
16892	7212549414185	Add(768,16124)

t	n	Source
17012	7212549414530	Add(888,16124)
17108	7212549414842	Add(984,16124)
17156	7212549415010	Add(1032,16124)
17252	7212549415370	Add(1128,16124)
17284	10000000000000	RS(149^1,6,116)

B.18 24-CFF table



t	n	Source
675	729	RS(3^3,2,25)
725	841	RS(29^1,2,25)
775	961	RS(31^1,2,25)
800	1024	RS(2^5,2,25)
925	1369	RS(37^1,2,25)
1025	1681	RS(41^1,2,25)
1075	1849	RS(43^1,2,25)
1175	2209	RS(47^1,2,25)
1225	2401	RS(7^2,2,25)
1325	2809	RS(53^1,2,25)
1475	3481	RS(59^1,2,25)
1525	3721	RS(61^1,2,25)
1600	4096	RS(2^6,2,25)
1675	4489	RS(67^1,2,25)
1775	5041	RS(71^1,2,25)
1825	5329	RS(73^1,2,25)
1975	6241	RS(79^1,2,25)
2025	6561	RS(3^4,2,25)
2075	6889	RS(83^1,2,25)
2225	7921	RS(89^1,2,25)
2401	117649	RS(7^2,3,49)
2597	148877	RS(53^1,3,49)
2891	205379	RS(59^1,3,49)
2989	226981	RS(61^1,3,49)
3136	262144	RS(2^6,3,49)
3283	300763	RS(67^1,3,49)
3479	357911	RS(71^1,3,49)
3577	389017	RS(73^1,3,49)
3871	493039	RS(79^1,3,49)
3969	531441	RS(3^4,3,49)
4067	571787	RS(83^1,3,49)
4361	704969	RS(89^1,3,49)
4753	912673	RS(97^1,3,49)
4949	1030301	RS(101^1,3,49)
5047	1092727	RS(103^1,3,49)
5243	1225043	RS(107^1,3,49)
5329	28398241	RS(73^1,4,73)
5767	38950081	RS(79^1,4,73)
5913	43046721	RS(3^4,4,73)
6059	47458321	RS(83^1,4,73)
6497	62742241	RS(89^1,4,73)
7081	88529281	RS(97^1,4,73)
7373	104060401	RS(101^1,4,73)
7519	112550881	RS(103^1,4,73)
7811	131079601	RS(107^1,4,73)

t	n	Source
7957	141158161	RS(109^1,4,73)
8249	163047361	RS(113^1,4,73)
8833	214358881	RS(11^2,4,73)
9125	244140625	RS(5^3,4,73)
9271	260144641	RS(127^1,4,73)
9344	268435456	RS(2^7,4,73)
9409	8587340257	RS(97^1,5,97)
9797	10510100501	RS(101^1,5,97)
9991	11592740743	RS(103^1,5,97)
10379	14025517307	RS(107^1,5,97)
10573	15386239549	RS(109^1,5,97)
10961	18424351793	RS(113^1,5,97)
11636	18424352522	Add(675,10961)
11686	18424352634	Add(725,10961)
11736	18424352754	Add(775,10961)
11737	25937424601	RS(11^2,5,97)
12125	30517578125	RS(5^3,5,97)
12319	33038369407	RS(127^1,5,97)
12416	34359738368	RS(2^7,5,97)
12707	38579489651	RS(131^1,5,97)
13289	48261724457	RS(137^1,5,97)
13483	51888844699	RS(139^1,5,97)
14158	51888845428	Add(675,13483)
14208	51888845540	Add(725,13483)
14258	51888845660	Add(775,13483)
14283	51888845723	Add(800,13483)
14408	51888846068	Add(925,13483)
14453	73439775749	RS(149^1,5,97)
14641	3138428376721	RS(11^2,6,121)
15125	3814697265625	RS(5^3,6,121)
15367	4195872914689	RS(127^1,6,121)
15488	4398046511104	RS(2^7,6,121)
15851	5053913144281	RS(131^1,6,121)
16526	5053913145010	Add(675,15851)
16576	5053913145122	Add(725,15851)
16577	6611856250609	RS(137^1,6,121)
16819	7212549413161	RS(139^1,6,121)
17494	7212549413890	Add(675,16819)
17544	7212549414002	Add(725,16819)
17594	7212549414122	Add(775,16819)
17619	7212549414185	Add(800,16819)
17744	7212549414530	Add(925,16819)
17844	7212549414842	Add(1025,16819)
17894	7212549415010	Add(1075,16819)
17994	7212549415370	Add(1175,16819)

t	n	Source
18029	10000000000000	RS(149^1,6,121)