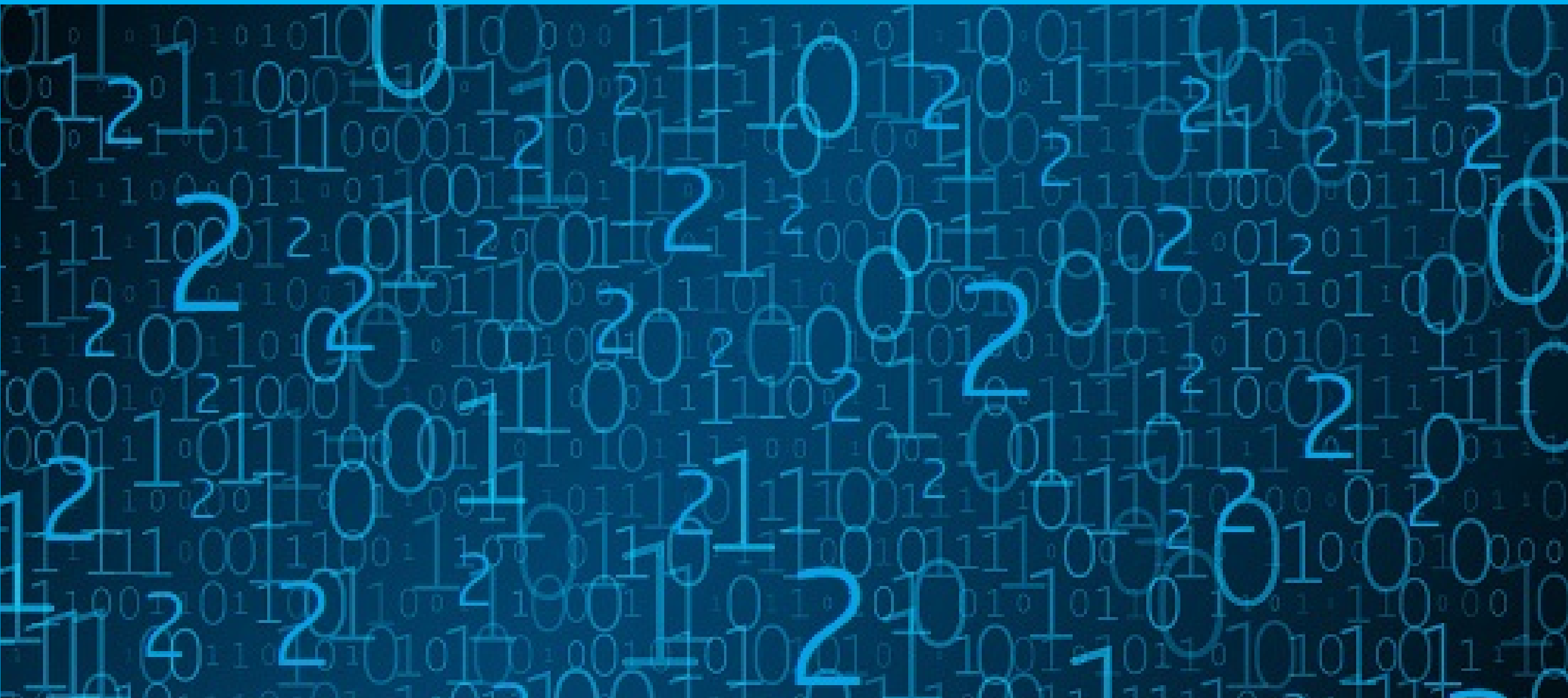
The Trifference Problem

Bounds, Small Values and Explicit Constructions

M. van Donkelaar



The Trifference Problem

Bounds, Small Values and Explicit Constructions

by

M. van Donkelaar

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Thursday July 14, 2022 at 15:00 AM.

Student number:	5132525	
Project duration:	April 25, 2022 – July 8, 2022	
Thesis committee:	Prof. dr. D. Gijswijt,	TU Delft, supervisor
	dr. A. Bishnoi,	TU Delft, supervisor
	dr. J. Komjáthy,	TU Delft

TU Delft

Preface

Dear reader,

This report marks the end of my bachelor of Applied Mathematics at the TU Delft. I have always been fascinated by puzzles involving mathematics and numbers, which is how I ended up with the trifference problem as my thesis subject. Writing this paper has made me realize that the academic world is even more extensive and thorough than I imagined, which has made me more enthusiastic and agitated to do more research in the future.

Most of all, I want to thank my supervisors Anurag Bishnoi and Dion Gijswijt for their guidance and our weekly meetings. They helped me a lot with new insights in the problem, and all the obstacles that come with writing my first thesis.

I also want to thank Júlia Komjáthy for taking the time and effort to be in my graduation committee. Lastly I want to thank my dad René, and Hugo for proofreading my script and their support.

M. van Donkelaar
Delft, July 2022

Abstract

A *code C* is defined to be a set of S *words*, where a word is a sequence of n entries. We call S the *size* and n the *length* of the code. The entries of the code can have k different values, $\{0,..,(k-1)\}$.
Define a *perfect k-hash code* (PHC) as a code with the property that any collection of v words in the code is different at at least one index.
PHC's are useful mathematical objects within different fields of theoretical computer science and coding theory. This thesis will focus on one typical kind of PHC, a *trifferent code*.
Such a trifferent code is defined as a PHC where $k = v = 3$. This means that any collection of three words in the code has to differ at some index. We now define $T(n)$ to be the largest possible size of a trifferent code given that it has length n. The question that arises is, what is the value of $T(n)$?
It turns out that determining the exact value is complicated, unless n is really small. Therefore, we try to understand the asymptotic nature of the function $T(n)$. This is what we call *the trifference problem.* This paper will cover and prove the best known asymptotic upper and lower bound on $T(n)$. Moreover, it will explain and include a Python code that can be used to show and prove all values of $T(n)$ for $n \in \{1,..,6\}$. Lastly, two different ways to explicitly construct trifferent codes for any value of n will be given and compared.

Contents

1 Introduction

A *code C* is defined to be a set of S *words*, where a word is a sequence of n entries. We call S the *size* and n the *length* of the code. The entries of the code can have k different values, $\{0,..,(k-1)\}$. We will write such a code as an $S \times n$ matrix, where the words of the code are written as the columns of the matrix.
We can now define a *perfect k-hash code* [1] PHC$(S; n, k, v)$ as a code with the property that any collection of v words in the code there is an index at which these words have pairwise distinct values.
Perfect hash codes have existed for quite some time and are used in multiple fields. They first arose within data-management. Melhorn showed some early results of this [11]. On top of that they are also involved in some circuit complexity problems [12] and the design of deterministic analogues of probabilistic algorithms [1]. Moreover, they are used more and more within coding theory and cryptography [14].
A *trifferent code* is a PHC where $k = v = 3$. Therefore we can look at it as if it is a problem over the finite field $\mathbb{F}_3$. This means that for any three codewords a, b, c in a trifferent code, there is an index i such that $\{a_i, b_i, c_i\} = \mathbb{F}_3$. An example of such a code is the so called 'Tetra-Code', which can be seen below. This code has 9 words of length 4.

$$\begin{bmatrix} 0 & 0 & 2 & 2 & 2 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 2 & 1 & 1 & 0 & 1 & 2 \\ 0 & 1 & 1 & 0 & 2 & 1 & 1 & 0 & 1 \\ 0 & 1 & 2 & 1 & 0 & 1 & 2 & 1 & 0 \end{bmatrix}$$

It can be checked that any three columns in this code are pairwise disjoint in some row. For example: The first, second and fourth column are $\{0,1,2\}$ respectively on the second row.
We now define $T(n)$ to be the largest possible size of a trifferent code given that it has length n. The question that arises is, what is the value of $T(n)$? It turns out that determining the exact value is complicated unless n is very small. The only known exact values of $T(n)$ are those of $n \in \{1,...,6\}$, where the value for $T(6)$ has just recently been computed by Fiore, Gnutti and Polak [2]. These known values for $n = (1,2,3,4,5,6)$ are $T(n) = (3,4,6,9,10,13)$. For all $n \in \{1,\dots,6\}$ an example of a code with length $T(n)$ can be found in Appendix A. Since it is so hard to find these exact values, we are more interested in trying to understand the asymptotic nature of the function $T(n)$. This is what we call *the trifference problem*.
Throughout the years there have been numerous attempts to find upper and lower bounds on the asymptotic values of $T(n)$. While some were successful, the upper bound and the lower bound still differ by quite a large factor. Moreover, the upper bound that has been proved by J. Körner [7] in 1973 still has not been improved on by any exponential factor yet. This upper bound will be shown and proved in Chapter 2.
After that, the best known asymptotic lower bound will be given and proved in Chapter 3.
In Chapter 4 a Python code that can be used to show the values of $T(n)$ for $1 \le n \le 6$ will be given and explained.
Chapters 5 and 6 contain two ways to explicitly construct trifference codes for any value of n.
Lastly, in Chapter 7 the known values, together with these two explicit constructions will be used to create an overview of all known lower bounds on $T(n)$ for small values of n.

[1] Also known as perfect k-hash families, or PHF$'s$

2 Upper bound

2.1. Introduction

The best known general upper bound on the trifference problem has been shown in 1973 by Körner [7]. The bound he gave is not specifically for the trifference problem, but for general perfect k-hash codes.[1] The proof is very short and follows from a simple counting argument. Throughout the years there have been numerous attempts to improve on his bound. Some of these were successful for PHC's with $k > 3$ [3][4]. Yet, almost 50 years later, for the case $k = 3$ his bound remains unbeaten.

2.2. Körner's upper bound

Theorem 2.1 (Körner)**.** *For all n > 0*

$$T(n) \leq 2\left(\frac{3}{2}\right)^n.$$

Remark. *This means that for any n, the value of* $T(n+1)$ *can be at most 1.5 times larger than the value of* $T(n)$*. If we look at the known values for* $T(n)$ *in the introduction we see that for* $1 \leq n \leq 4$ *this is a tight bound, but for* $n > 4$ *it is already not tight anymore.*

Proof. Let $A \subseteq \{0,1,2\}^n$ be a trifferent code of maximum size $T(n)$. Now consider the set:

$$I_1 = \{x \in A : x_1 = 0 \text{ or } x_1 = 1\}$$

It is easy to see that $|I_1| \leq T(n-1)$. This can be shown by a contradiction proof.
Assume that $|I_1| > T(n-1)$, then the code generated by all words in I_1 without their first index generates a trifferent code of length $n-1$ and size $|I_1| > T(n-1)$. But $T(n-1)$ is defined to be the largest code of length $n-1$ so a contradiction follows.
In the same manner define:

$$I_2 = \{x \in A : x_1 = 0 \text{ or } x_1 = 2\}$$

$$I_3 = \{x \in A : x_1 = 1 \text{ or } x_1 = 2\}$$

By the same reasoning as above we have that $|I_2| \leq T(n-1)$ and $|I_3| \leq T(n-1)$ as well.
Now since any vector $y \in A$ starts with some number in $\{0,1,2\}$, each y is included in exactly two of the three sets $\{I_1, I_2, I_3\}$. Therefore we have that $I_1 + I_2 + I_3 = 2A$. Using this it follows that:

$$2T(n) = 2|A| \leq 3T(n-1).$$

Or, also using that $T(n)$ is always an integer:

$$T(n) \leq \left\lfloor \frac{3}{2} T(n-1) \right\rfloor \tag{2.1}$$

[1]The general bound is of the form $|\text{PHC}(n,k)| \leq (k-1)\cdot\left(\frac{k}{k-1}\right)^n$

Since this is a recursive formula, use that $T(1) = 3$ to find the general formula to be equal to:

$$T(n) \leq 2 \cdot \left(\frac{3}{2}\right)^n. \tag{2.2}$$

□

2.3. Improving the constant

History has shown that it is very hard to improve on the exponential bound of $\left(\frac{3}{2}\right)^n$. However, the constant of 2 in Theorem 2.1 can be improved for higher values of n, using that some values of $T(n)$ are known. Since it is known that $T(5) = 10$, $T(6) = 13$, these values can be used to construct the following bounds:

Theorem 2.2.

$$T(n) \leq \frac{10}{\left(\frac{3}{2}\right)^5} \cdot \left(\frac{3}{2}\right)^n < 1.33 \cdot \left(\frac{3}{2}\right)^n \textit{for } n \geq 5$$
$$T(n) \leq \frac{13}{\left(\frac{3}{2}\right)^6} \cdot \left(\frac{3}{2}\right)^n < 1.15 \cdot \left(\frac{3}{2}\right)^n \textit{for } n \geq 6$$

There are no known values for $n \geq 7$ yet, so better bounds can not be constructed in this way. However, the floor function in (2.1) can be used to construct slightly lower upper bounds for $n \geq 7$. This can be done by starting with the value $T(6) = 13$, then recursively multiplying by $\frac{3}{2}$ and rounding down to an integer. The constants that arise from this are slightly better than 1.15. See figure 2.1.

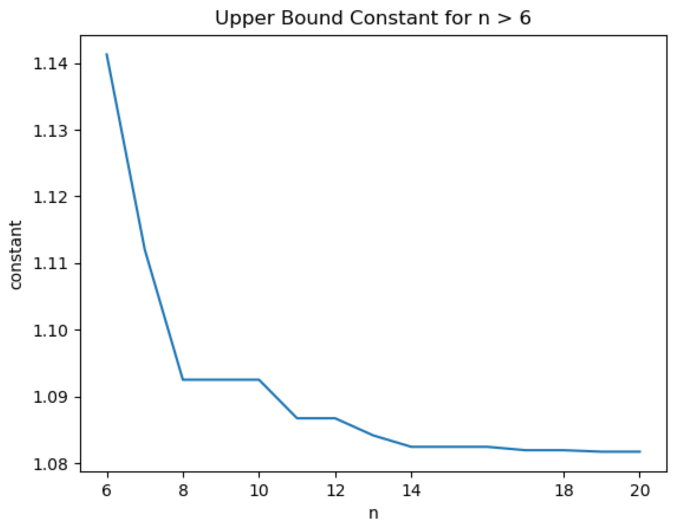


Figure 2.1: Plot of the Upper Bound Constant for $n \geq 6$

It can be seen that the constant decreases to value of approximately 1.08 after $n = 14$. The exact value for $n = 14$ is 1.0825. As can be expected by looking at Figure 2.1, using this method will not decrease the constant a lot further. The value for $n = 100$ is equal to 1.0815, which is only 0.01 less. We can therefore conclude that

$$T(n) < 1.0825 \cdot \left(\frac{3}{2}\right)^n \text{ for } n \geq 14 \tag{2.3}$$

is a reasonable upper bound for larger values of n.

3 Lower bound

3.1. Introduction

The proof of the upper bound in the previous section is relatively easy, but the best known lower bound requires a bit more work. This lower bound is only an asymptotic bound, and does not give any specific values for small n. The construction of the bound consists of two parts. It uses an upper bound on a similar problem proven by Fredman and Komlos in 1984 [3], where they defined the problem of perfect k-hashing in a slightly different way. The bound that they found is a general bound for all values of k, and they use a probabilistic approach to construct it. Their result has been used later on by Körner and Marton [8] to improve on the bound specifically for the value $k = 3$.

3.2. Komlos and Fredman

In the introduction we defined 'the trifference problem' in the following way: Given a word length n, what is the maximum size $T(n)$ of a trifference code with this length? Komlos and Fredman define the problem in a different way. They ask: If I want to construct a trifferent code of size T, what is the minimum required length n the code needs to be? Komlos' and Fredman's method is for general perfect hash codes, so they use k and v in their definition as well. They formally define the problem in the following way:
Let $U = \{1,2,\dots,T\}$ and let P be a partition of U into k (possibly empty) blocks. We call a subset S of U *separated* by P if every block of P contains at most one element of S. Note that this is only possible if S has at most k elements. A family F of partitions $P_1, P_2, ..., P_n$ is called a (k, v)-system if every subset of S of v elements of U is separated by at least one partition in F. This means that all possible $N := \binom{T}{k}$ subsets with k elements of U have to be partitioned by some P_i in F. The question that arises now is what the minimum size of any (k, v)-system is given k and v.

Definition 3.1. *The minimum size of a (k, v)-system given k, v and T is defined as $Y(k, v, T)$.*

As stated in the introduction, we will use the upper bound on $Y(k, v, T)$ later on to construct a lower bound on our version of the trifference problem.
For the probabilistic approach we first need to define the number of sets that one given partition P_i separates. This is what we call the *volume q* of the partition. Since we want to minimize the number of partitions needed, we want to look for partitions that have a volume that is as large as possible. We therefore define G to be the family of all partitions that have a maximum volume Q. The value of Q will be discussed later in this section. Now assume that we have a sequence of m random partitions in G. (So m partitions with volume Q.)

Lemma 3.2. *The probability that a sequence of m random partitions in G do not form a (k, v)-system is at most:*

$$N \cdot \left(1 - \frac{Q}{N}\right)^m \tag{3.1}$$

.

Proof. This can be seen in the following way: $\frac{Q}{N}$ is the fraction of all subsets that a single partition can separate. $1 - \frac{Q}{N}$ is therefore the fraction of subsets that it can't separate. Since all partitions $P \in G$ are random partitions of U, they all have an equal probability of occurring and are independent of each other. Therefore, the probability that any random subset is not separated by m partitions is $\left(1-\frac{Q}{N}\right)^m$. Multiplying this by the number of subsets N gives an upper bound on the probability that there is at least one subset which is not separated. This is the probability that the partitions are not a (k,v)-system, as we wanted.

□

We want the probability that the m partitions do not form a (k,v) system to be less than 1, because this implies that there is a chance that the random partitions do form a (k,v)-system. That will imply that such a system exists.
Thus, if we find m such that the expression in 3.1 is smaller than 1, then $Y(k,v,T) < m$. Writing 3.1 to be smaller than 1 gives:

$$\begin{aligned} N\cdot\left(1-\frac{Q}{N}\right)^m &< 1 \\ \left(1-\frac{Q}{N}\right)^m &< \frac{1}{N} \\ m\cdot\log\left(1-\frac{Q}{N}\right)) &< \log\frac{1}{N} \\ m &< \frac{\log\frac{1}{N}}{\log\left(1-\frac{Q}{N}\right)} \\ m &< -\frac{\log N}{\log\left(1-\frac{Q}{N}\right)} \end{aligned}$$

From this we can conclude that

$$Y(k,v,T) \le -\frac{\log N}{\log(1-\frac{Q}{N})} \tag{3.2}$$

Remark. *Note that* $\log\left(1-\frac{Q}{N}\right) < 0$*, so that the right hand side of 3.2 has a positive value.*

The next question that arises is: what is the vlaue of Q? Assume that P has maximal volume Q. Name the subsets of P to be $p_1, p_2, ..., p_k$. It is clear that P has to be distributed as uniformly as possible to attain a maximum volume. This means that $p_1, p_2, ...$ all have a size of approximately $\frac{T}{k}$. This means that any collection of v of these partition subsets will be able to separate approximately $\left(\frac{T}{k}\right)^v$ subsets of size k. Since there are $\binom{k}{v}$ ways to pick these v partition subsets the maximum value Q can be approximated by:

$$Q \approx \left(\frac{T}{k}\right)^v \cdot \binom{k}{v} \tag{3.3}$$

We can now fill in the value of Q in (3.3) into the right hand side of (3.2) to obtain:

$$\begin{aligned} &\frac{\log\binom{T}{v}}{\log(1-\frac{(\frac{T}{k})^v\binom{k}{v}}{\binom{T}{v}})} \\ &= \frac{\log\binom{T}{v}}{\log\left(1-\frac{T^v}{k^v}\cdot\frac{k!}{(k-v)!\cdot v!}\cdot\frac{(T-v)!\cdot v!}{T!}\right)} \\ &= \frac{\log\binom{T}{v}}{\log\left(1-\frac{k^{\underline{v}}}{k^v}\cdot\frac{T^v}{T^{\underline{v}}}\right)} \end{aligned}$$

Where $k^{\underline{v}} := \frac{k!}{(k-v)!}$.
Both the nominator and the denominator of this expression can be bounded to obtain a more general final expression.

Numerator
For the top hand side, use that $\binom{T}{v} \leq T^{v} \leq \left(\frac{T}{v}\right)^{v}$ to obtain:

$$\begin{aligned}\log\binom{T}{v} &\leq \log\left(\frac{T}{v}\right)^{v} \\ &= v \cdot \log\frac{T}{v} \\ &\leq v \cdot \log T\end{aligned}$$

Denominator
For given $v \geq 1$, with $x \geq v$ define the function $f(x) := \frac{x^{\underline{v}}}{x^{v}}$ and $\frac{1}{f(x)} := \frac{x^{v}}{x^{\underline{v}}}$. In this way the denominator becomes $\log\left(1 - \frac{1}{f(T)} \cdot f(k)\right)$. Note that $0 \leq f(x) \leq 1$ for all $x \geq v$ and that it is an increasing function over x. Conversely, $\frac{1}{f(x)} \geq 1$ for all $x \geq v$ is a decreasing function over x.
Since $T \geq v$ we have that $1 \leq \frac{1}{f(T)}$, which implies that $f(k) \leq f(k) \cdot f^{-1}(T)$. Using that $f(x)$ is increasing and that $T \geq k$ we find that $0 \leq f(k) \cdot f^{-1}(T) \leq 1$.
It follows that $1 - f(k) \geq 1 - f(k) \cdot f^{-1}(T)$.
Taking logarithms (which both result in negative values) gives then that: $\log\left(1 - f(k)\right) \geq \log\left(1 - f(k) \cdot f^{-1}(T)\right)$.
Multiplying by minus one gives:

$$-\log\left(1 - f(k)\right) \leq -\log\left(1 - f(k) \cdot f^{-1}(T)\right) \tag{3.4}$$

which can be written as

$$-\log\left(1 - \frac{k^{\underline{v}}}{k^{v}}\right) \leq -\log\left(1 - \frac{k^{\underline{v}}}{k^{v}} \cdot \frac{T^{v}}{T^{\underline{v}}}\right) \tag{3.5}$$

.

Final expression
We can now combine the expressions for the numerator and the denominator to obtain the following theorem:

Theorem 3.3. $Y(k, v, T) \leq \left(\frac{v \cdot \log(T)}{-\log\left(1 - \frac{k^{\underline{v}}}{k^{v}}\right)}\right) = \left(-v \frac{\log(T)}{\log\left(1 - \frac{k^{\underline{v}}}{k^{v}}\right)}\right)$

Which is the inequality that Komlos and Fredman present in their paper.

3.3. Körner and Marton

Körner and Marton reformulated the problem of Fredman and Komlos.
Instead of looking for the minimum length $Y(k, v, T)$ required to create a code of size T, they defined $S(k, v, n)$ to be the maximum size of a code given that it has length n. This corresponds with the way we define the trifference problem as well, where $v = k = 3$. In the bound of Theorem 3.3 write $Y(k, v, T) = n$, $S(k, v, n) = T$ and use that $-\log x = \log\frac{1}{x}$ to obtain:

$$n \leq v \frac{\log S(k, v, n)}{-\log\left(1 - \frac{k^{\underline{v}}}{k^{v}}\right)}$$

$$\frac{1}{v} \log\left(\frac{1}{1 - \frac{k^{\underline{v}}}{k^{v}}}\right) \leq \frac{1}{n} \log S(k, v, n). \tag{3.6}$$

Using this inequality we can prove the following theorem:

Theorem 3.4.

$$\frac{1}{n} \log S(3, 3, n) \gtrsim \frac{1}{4} \log\frac{9}{5}$$

Proof. To prove this inequality we have to use the known fact that $S(3, 3, 4) = 9$, also known as the 'tetra-code'. An example of such a code is given in the introduction.

Denote $B = \{0, 1, \ldots, 8\}$, and take $A \subseteq B^n$ to be 3-separated. This means any subset of three elements in A is different at at least one index. We can plug in $k = 9$, $v = 3$ in the bound above to find that:

$$\frac{1}{n}\log|A| \gtrsim \frac{1}{2}\log\left(\frac{1}{1-\frac{9^{\underline{3}}}{9^3}}\right) = \frac{1}{2}\log\left(\frac{1}{1-\frac{56}{81}}\right) = \frac{1}{2}\log\frac{81}{25} = \frac{1}{2}\log\left(\frac{9}{5}\right)^2 = \log\frac{9}{5}. \tag{3.7}$$

Now assign any number within B to one of the 9 words in the tetra-code. Replacing all numbers in A with these elements will produce a code C that is four times as long, so has length $4n$, and with equal size, so $|A| = |C|$. Since the 'tetra-code' has values in $\{0, 1, 2\}$ so does C. From the fact that any three elements in A are different at some index, and all elements in the tetra-code that belong to these numbers also differ at some index. This implies that any three elements in our code C differ at some index, which means that C is a trifferent code. It follows that:

$$\frac{1}{n}\log S(3,3,4n) \gtrsim \log\frac{9}{5}.$$

Now assume $s = 4n$. Substituting this in (3.3) gives

$$\frac{4}{s}\log S(3,3,s) \gtrsim \log\frac{9}{5}.$$
$$\frac{1}{s}\log S(3,3,s) \gtrsim \frac{1}{4}\log\frac{9}{5}.$$

We see now that theorem 3.4 holds for s if s is a multiple of 4. Since we are working with an asymptotic bound, however, this is sufficient to assume that the inequality holds for all n. This concludes the proof. □

3.4. Can we improve?

It might seem quite random that Korner and Marton decide to substitute the 'Tetra-code' of $S(3,3,4)$ in equation 3.6. Why wouldn't they choose $S(3,3,6)$ or $S(3,3,3)$ for example, since these values are also known. The answer is that $n = 4$ simply happens to produce the best bound. A code with a relatively large size works best in producing a result in the way Körner and Marton did and $S(3,3,4) = 9$ performed better than other values. In the same way as equation (3.7) other bounds can be calculated produced by different values of $S(3,3,4)$. The question that arises is: What values are required to improve on the current bound? These can be found by solving the following inequality:

$$\frac{1}{n}\frac{1}{2}\log\left(\frac{1}{1-\frac{S^{\underline{3}}}{S^3}}\right) \geq \frac{1}{4}\log\frac{9}{5} \tag{3.8}$$

$$-\frac{1}{n}\log\left(1-\frac{(S-1)(S-2)}{S^2}\right) \geq \frac{1}{2}\log\frac{9}{5} \tag{3.9}$$

Where $S = S(3,3,n)$. This inequality can be easily solved using mathematics software like Maple. Some required values can be found in table 3.1 below.

value of n	minimum required value $S(3,3,n)$	Known upper bound for $S(3,3,n)$
1	4	**3**
2	5	**4**
3	7	**6**
4	9	**9**
5	13	**10**
6	17	**13**
7	23	19
8	31	29
9	42	43
10	57	64

Table 3.1: Required values of $S(3,3,n)$ with their known upper bounds

Here the upper bounds for $t \leq 6$ are just the known values of $S(3,3,n)$. The other bounds are constructed using formula (2.1). It can be seen that for $n = 7,8$ it is impossible to improve the bound. For $n \geq 9$ it might be possible but it seems very unlikely since the required values are still very close to the upper bound. It might be interesting to analyse the required value of $T(n)$ larger values of n and determine whether it is possible to create a better bound in the same manner Körner and Marton did, but because of time management we leave this to future work.

4
Finding Exact values for small n

4.1. Introduction

As stated in the introduction, the exact values of $T(n)$ are only known for $n \leq 6$. These exact values for $n = 5, 6$ have recently been shown by Fiore, Gnutti and Polak [2]. They showed this by using an algorithm that finds all possible trifferent codes of size N and length n. Using this algorithm they showed that there exist codes of length 6 and size 13 (see Appendix A), and that there exists no trifferent code of length 6 and size 14, which proves that $T(6) = 13$.
But why can't their algorithm go any higher than $T(6)$? The answer is that there is simply not enough computing power.
Assume that your algorithm wants to check trifference on all possible sets of length 6 and size 14. Checking all sets of three words on trifference within a code of size 14 takes $\binom{14}{3} = 364$ calculations. There are $3^6 = 729$ possible words of length 6, from with 14 have to be chosen and checked. In total, this would result in

$$\binom{729}{14} \cdot \binom{14}{3} \approx 4 \cdot 10^{31} \text{ calculations.}$$

An average computer takes about a 4 trillions years to do this many calculations. This is why it is so hard to prove exact values of $T(n)$, even for relatively small n. Luckily, the algorithm of [2] uses some properties of trifferent codes to drastically reduce the amount of computations necessary, which is how they managed to show the value for $T(6)$.
This chapter will cover these properties and how they have been implemented in a Python code that can be used to verify $T(6) = 13$. Moreover, an improvement on the code will be given that can make it run even faster. A Python code that uses the same properties as Fiore et al [2] can be found in Appendix B.2. A Python code with these properties and an improvement can be found in Appendix 4.3.

4.2. The algorithm

The algorithm consist of 3 main points: ordering the words, using equivalence and determining the first entries.

4.2.1. Ordering the words

The first key point in the algorithm is that it lists the set of all possible words $W := \{0, 1, 2\}^n$ in lexicographic order: $[0, ..., 0, 0], [0, ..., 0, 1], [0, ..., 0, 2], [0, ..., 1, 0], ..., [2, ..., 2, 2]$. Giving each word w an *index* I_w from 0 to $3^n - 1$. For any trifferent set of j words $\{w_1, \ldots, w_j\}$ we require that $I_{w_1} \leq \ldots \leq I_{w_j}$.
We start our algorithm with the 3 words with the smallest indices (which are trifferent). Then we iterate over all other words by index in ascending order. If a word can be added to the current trifferent set, we add it and restart the algorithm with the new set of 4 words, and look only for words after the newly add word. We can do this because we know that all words before this word do not form a new trifferent code. As soon as we arrive at the last index $3^n - 1$ and this word can't be added either, we know that there are no new words that can be added. Therefore we remove the last word from the current trifferent code. In this way we iterate over all words in a very fast way while knowing that will find any trifferent code in it (where the words have

ascending indices). The algorithm iterates over all possibilities in a recursive way, which can be seen in the pseudo code below.

```
WordSet = [[0,...,0],...,[2,...,2]] #all possible words of length n in lexicographical
    order

def recursive(index, trifset, n, M):
    countList = []  #countlist counts the amount of words that can be added to the code
     for each i, while remaining trifferent
    if len(trifset) == M:
        print(trifset)
    for k in range(index, 3^n): #all words with larger index
     if trifset, WordSet[k] are trifferent:
        countlist.append(recursive(k+1, trifset + WordSet[k], n, M) #Add the word and
    look for more
    if trifset, WordSet[k] not trifferent for all k:
    #If no more words can be added
        return 0
    else:
    #if at least one word can be added
        return max(countlist)+1

recursive(0, [[0,...,0]], n, M)
```

Listing 4.1: Pseudo code for finding codes of size N and length n

It can be seen that the 'recursive' function iterates over all words with a larger index than k, and recalls itself when it has found one. The 'countlist' variable counts the number of words can be added extra to the current code, for all possible values of k. Therefore, if the recursive function does not find any new words, it returns 0. If it does find words that can be added, taking the maximum over all these words and adding 1 returns maximum amount of words for the previous iteration. In this way the countlist can keep track of the largest trifferent set that was found.

4.2.2. Equivalence

The second property the algorithm uses the following. If we have a code C which we know is trifferent, then we have three ways to construct codes from C which are also trifferent. We do this by applying certain permutations to C.
The first way to do this is by permuting the order of the columns of the matrix of C. However, we already fixed the order of the words by ascending index, so this permutation is not relevant.
The second way is to permute the rows of the matrix of C. It is easy to see that if three words are trifferent in some row, then putting this row somewhere else won't change the fact that those columns are trifferent.
The third way to create a new trifferent code is to apply permutations to the symbols $\{0,1,2\}$ within any row of the matrix. This is because if any three columns form F_3 in some row, they will still form F_3 after applying any permutation to the entries within that row. Since these permutations can be used on any trifferent matrix, we can also combine them. This gives the following definition.

Definition 4.1. *If a code C^* can be obtained from C by subsequently applying permutations to the column order, row order and the symbols $\{1,2,3\}$ in each row, then C^* and C are equivalent.*

Remark. *We do need to include column order, because after permuting the row order of C the columns of C^* might not have ascending indices anymore.*

We will use these properties of equivalence to prove the following theorem:

Theorem 4.2. *For any code C there exists an equivalent trifferent code C' that contains columns v_0 and v_1, where v_0 is the zero vector and v_1 is of the form $(0,...,0,1,...,1)$. Moreover, there is no nonzero word in C' that has a smaller index than v_1.*

Proof. Constructing the v_0 is easy. Take any column in the code and apply permutations to the values of $\{0,1,2\}$ in such a way that the column becomes a zero vector. This is equal to subtracting this vector from all vectors in the code (in $\mathbb{F}_3$).
To construct the vector v_1 take the vector in the remaining code that consists of the maximum number of zeros. Now take any row where this word has a 2, and permute the ones and twos in these rows (this won't

affect the 0-vector). We now have a word consisting of only zeros and ones. We can switch the rows of the matrix in such a way that the zeros are at the top of the word and the ones are at the bottom to obtain a vector of the desired form. Since we chose this vector as the one with the most zeros in it, there can not be any nonzero words with a smaller index than this vector. □

Therefore we can start our algorithm with the vectors v_0 v_1, and let it ignore all words that come before this vector. If we iterate over all $n-1$ possible vectors that v_1 can be, we are sure that a trifference code will be found if it exists.

4.2.3. Determining the first entries

The last property the algorithm uses is the following: Let s_0, s_1, s_2 the number of words in the code C that have 0, 1 and 2 respectively at their first coordinate. It is evident that $s_i + s_j \leq T(n-1)$ for $i \neq j$. This is because if $s_i + s_j > T(n-1)$, a code of size larger than $T(n-1)$ with length $n-1$ could be formed by using all words that start with i and j and removing their first coordinate. This results in the fact that there are only a few possible compositions of first coordinates for different values of n. These can be found by solving $s_0 + s_1 + s_2 = M, s_i >= 0$ and $s_i + s_j \leq T(n-1)$. The values are the following

- $n = 4$, $N = 9$: [3,3,3]
- $n = 5, N = 10$: [8,1,1], [7,2,1], [6,3,1], [6,2,2], [5,4,1], [5,3,2], [4,4,2], [4,3,3]
- $n = 5, N = 11$: [7,2,2], [6,3,2], [5,4,2], [5,3,3], [4,4,3]
- $n = 6, N = 13$: [7,3,3], [6,4,3], [5,5,3], [5,4,4]
- $n = 6, N = 13$: [6,4,4], [5,5,4]

Knowing for all words in the code what the first entry is drastically reduces the amount of computations necessary.
The complete code which can be found in Appendix B. The values for n and N can be chosen for $n = 4,5,6$ and N anything that is relevant for that specific n. Running the code for $n = 5, N = 10$ returns 689 codes of length 5 and size 10. Running the code for $n = 5, N = 11$ returns no codes. This confirms that $T(5) = 10$.
Running the code for $n = 6, N = 13$ will take approximately 10 hours [1] and will return exactly 1046 different trifference codes of length 6 and size 13. This is exactly the same number of codes found by Stefano et al. [2]. It will return no codes if $N = 14$. This proves that $T(6) = 13$ as desired.

4.3. Improving the code

To improve on the approach of Stefano et al [2] we can extract some more properties from equivalent trifferent codes. Keep in mind that we already proved that for any code there exists an equivalent code with the 0 vector v_0 and some vector v_1 of the form $(0,...,0,1,...,1)$. To construct these vectors we first took any arbitrary vector within the code and subtracted this vector from all others. Afterwards we picked the nonzero vector containing the most zeros. In this vector we permuted the ones and twos in any row that contained a two and switched the order of the rows to achieve v_1.
The first property that we can add which is not used in the code above is the fact that the vector v_1 can be picked in such a way that it has the least zeros of all nonzero words. We already showed this in the proof of Theorem 4.2. We can therefore remove all words that contain more zeros than our vector v_1 from the set of words we still want to test. Especially when we take $v_1 = (0,1,...,1)$ this greatly reduces the calculation time.
The second property that we will add to our code is the following. We define v_2 as the vector that makes up the next column of our code, such that there is no vector with an index in between v_1 and v_2. There are a few specifications that we can add to v_2 to make our code run faster.
First of all, since v_2 has to form a trifferent set with v_0 and v_1 we know that v_2 contains a 2 in some row where $v_1 = 1$.

[1] All calculations in this project were done on a laptop with a Intel Core i7-8750H CPU processor

Second, we can make another specification for v_2. Within permutations of symbols, we have 'pinned' the value of 0 on any row by picking our vector v_0. For all rows where $v_1 = 1$ we have also pinned the value of 1 and therefore the value of 2. For all rows where $v_1 = 0$, however, the values of 1 and 2 can still be permuted, since they are zero in both v_0 and v_1. We can therefore permute all values of v_2 within the rows where $v_1 = 0$, such that all values where $v_2 = 2$ are switched to 1. Basically this tells us that there is some equivalent code where $v_2 = 0 \vee v_2 = 1$ in the rows where $v_1 = 0$.
Third, we can still separately permute the order of the rows where $v_1 = 0$ and the order of the rows where $v_1 = 1$, since permuting either of them won't affect v_0 and v_1. This allows us to put all values of v_2 in ascending order, separately for the values where $v_1 = 0$ and where $v_1 = 1$.
It is easily checked that there exists a word v_2 that has this form and the smallest index after v_0 and v_1. That is because applying these permutations to a vector will never increase the index of that vector. Say we picked a vector to be v_2 and permuted it in the above described way. If after these permutations there is still some vector v_2^* that has a smaller index than v_2 we can apply the permutations to v_2^* to make v_2^* the word with the smallest index. We can repeat this process until we find a vector v_2 that does have the smallest index after v_0 and v_1.
Combining all this we can greatly reduce the number of vectors v_2 that need to be tested, given v_1. A code that combines all these specifications for v_2 given v_1, as well as uses the property that all words containing more zeros than v_1 can be omitted, can be found in Appendix B.3. Running this code for $n = 6$ and $N = 13$ takes approximately 1 hour, which is about 10 times faster than the code described in Section 4.2. It finds 108 codes for $n = 6$ and $N = 13$, and finds 0 for $n = 6$ and $N = 14$. Again proving that $T(6) = 13$.
The next question is, can we use this new code to calculate $T(7)$? It was tried to run the improved code for $n = 7$ and $N = 17$ for first entry density $[6,6,5]$. However, it took approximately a week to calculate just the possibilities of $v_1 = [0,0,0,0,0,0,1]$, $v_1 = [0,0,0,0,0,1,1]$ and $v_1 = [0,0,0,0,1,1,1]$.[2] These are only half the possibilities for v_1. Therefore, it would by estimation take approximately 8 weeks to calculate all possibilities for $n = 7$ and $N = 17$, since there are four different possible first entry densities for this combination. We know that $15 \leq T(7) \leq 19$, so algorithm should run two or three times to find our the exact value of $T(7)$, depending on whether it finds a code of size 17 or not. This would mean that it would take at least 16 weeks to compute the answer to $T(7)$ on a single laptop, which is a really long time. In the future, however, stronger computers and parallel computing could be used to reduce this time and find a final answer for $T(7)$.

[2]The maximum size that the Python code found for these possibilities is 15.

5 Lower bound of 2n+1

5.1. Introduction

The algorithm of [2] in the Section 4.2 returns 3 different trifferent codes for $n = 6$ (up to equivalence). These are the following:

$$\begin{bmatrix} 0&0&2&2&2&2&2&0&1&1&1&1&1 \\ 0&1&0&2&2&2&1&2&0&1&1&1&2 \\ 0&1&1&0&2&1&2&2&2&0&1&2&1 \\ 0&1&1&1&0&2&2&2&2&2&0&1&1 \\ 0&1&1&2&1&0&2&2&2&1&2&0&1 \\ 0&1&2&1&1&1&0&2&1&2&2&2&0 \end{bmatrix} \begin{bmatrix} 0&0&2&2&2&2&2&0&1&1&1&1&1 \\ 0&1&0&2&2&1&2&2&0&1&1&2&1 \\ 0&1&1&0&2&2&1&2&2&0&1&1&2 \\ 0&1&1&1&0&2&2&2&2&2&0&1&1 \\ 0&1&2&1&1&0&2&2&1&2&2&0&1 \\ 0&1&1&2&1&1&0&2&2&1&2&2&0 \end{bmatrix} \begin{bmatrix} 0&0&2&2&2&2&2&0&1&1&1&1&1 \\ 0&1&0&2&2&1&2&2&0&1&1&2&1 \\ 0&1&1&0&2&2&2&2&2&0&1&1&1 \\ 0&1&1&1&0&2&1&2&2&2&0&1&2 \\ 0&1&2&1&1&0&2&2&1&2&2&0&1 \\ 0&1&1&1&2&1&0&2&2&2&1&2&0 \end{bmatrix}$$

These matrices are constructed in a very specific way. All three start with a 0 vector as its first column. Then there is an nxn sub matrix which we can call B. Note that B has a zero diagonal and is anti symmetric (so $B^T = -B$). After that there is the matrix $-B$.
It turns out we can generalize this structure of matrices to construct trifferent codes of length $2n+1$ for any $n \geq 6$, which will be shown in this chapter.

5.2. The construction

Theorem 5.1. *For all $n > 5$ there exist a trifferent code with word length n and size $2n+1$.*

To prove this theorem, we construct a matrix that is analogous to the sub matrix of the first code in the introduction.

Definition 5.2. *For every $n \geq 6$, we can construct a 'Trif-Matrix' A in the following step by step way: First make all the diagonal entries of A 0. Then make all entries in the lower half 1 and the entries in the top half 2. After that, add an 'anti-diagonal' which starts at the last row of the second column and goes up to the second row of the last column. On this diagonal, switch any index i with -i. (In $\mathbb{F}_3$, so 1 becomes 2 and 2 becomes 1). Lastly, take the first row of the $\lceil \frac{n}{2} \rceil$-th column and the first column of the $\lceil \frac{n}{2} \rceil$-th row and switch both these indexes with their inverse as well. An example of a Trif-Matrix A can be seen below.*

$$A = \begin{bmatrix} 0 & 2 & \cdots & 2 & 1 & 2 & \cdots & 2 \\ 1 & 0 & & 2 & 2 & 2 & & 1 \\ \vdots & & \ddots & & & & \iddots & 2 \\ 1 & 1 & & 0 & 2 & 1 & & \vdots \\ 2 & 1 & & 1 & 0 & 2 & & 2 \\ 1 & 1 & & 2 & 1 & 0 & & \vdots \\ \vdots & & \iddots & & & & \ddots & 2 \\ 1 & 2 & 1 & & \cdots & & 1 & 0 \end{bmatrix}$$

Note that for A it holds that $-A = A^T$ over the field $\mathbb{F}_3$.

Lemma 5.3. *For any $n > 5$, taking the columns of a 'Trif-Matrix' A, together with the columns of -A and the 0 vector will produce a trifferent code.*

Note that proving this lemma will prove Theorem 1, since both A and $-A$ have n columns.

Proof. The proof consists of showing that any three vectors in the code are trifferent with each other. The code can be split up in three parts:

- The zero vector, denoted 0
- The columns of A, denoted by letters: e.g. a
- The columns of -A, denoted by the negative vectors of A, e.g. $-a$

This produces seven possible combinations of vectors from different groups, which are the following:

1. Three columns of A $[a, b, c]$
2. Three columns of -A $[-a, -b, -c]$
3. The 0 vector and two columns of A $[0, a, b]$
4. The 0 vector and two columns of -A $[0, -a, -b]$
5. The 0 vector, one column of A and one of -A $[0, a, -a]$ or $[0, a, -b]$
6. One column of A and two of -A $[a, -a, -b]$ or $[a, -b, -c]$
7. Two columns of A and one of -A $[a, b, a]$ or $[a, b, -c]$

By symmetry of A and $-A$ it follows that 1 implies 2, 3 implies 4 and 6 implies 7. Therefore it is sufficient to show 1,3,5 and 6.

Three columns of A ($[a, b, c]$)

Number the columns of A from 1 to n, left to right. Furthermore, define i_j to be the j'th row of the i'th column. (Note that $i_i = 0$ for any i because of the zero diagonal). Now take three columns $a, b, c \in [1, .., n]$ arbitrary. Without loss of generality, we can assume that $a < b < c$. Since all columns have exactly one zero, there are only three rows at which the three columns are possibly trifferent. These are the rows/columns a, b, c.
For almost all a, b, c the submatrix induced by the rows/columns of a, b, c is equal to:

$$\begin{bmatrix} 0,2,2 \\ 1,0,2 \\ 1,1,0 \end{bmatrix} \tag{5.1}$$

Which is clearly trifferent in row b. This is the case unless some of the values are flipped on the 'anti-diagonal'. It is easy to check that if this is the case, there will still be a row containing $0, 1, 2$.

The 0 vector and two columns of A, ($[0, a, b]$)

Assume again that $a < b$. Showing that a and b are trifferent with the 0 vector is equal to showing that $a_k = -b_k \neq 0$ for some k in $[1..n]$. It is easy to see that if $b - a > 1$, so if a and b are not adjacent, there is at least one index i where $a_i = 1, b_i = 2$. It rests to show for $a + 1 = b$, so two adjacent columns. This can be seen easily by looking at the anti diagonal, since next to any index on the diagonal the adjacent indices are different. The only place this does not work is in the middle two columns for odd matrices. This is where the 0 diagonal and the anti diagonal intersect. For even matrices this does not give any problems, but for odd matrices it does, since the intersection looks like this:

$$\begin{bmatrix} 0 & 2 & 2 & 1 \\ 1 & \underline{0} & \underline{1} & 2 \\ 1 & \underline{2} & \underline{0} & 2 \\ 2 & 1 & 1 & 0 \end{bmatrix}$$

It can now be seen that the middle two rows are not trifferent with the 0 vector. This is why the 2 in the top middle of the matrix is turned into a 1. This 1 directly ensures that the middle two columns are indeed trifferent with the 0 vector and therefore all columns of A are.
Note that, for even values of n, the 1 on the top and the 2 on the left of the matrix are not necessary for forming a trifferent code.
The fact that all columns of A are trifferent with the 0 vectors implies the same for the columns of $-A$ as well.

The 0 vector, one column of A and one column of -A ($[0, a, -a]$ **or** $[0, a, -b]$)

Number the vectors of -A from $1'$ up to n'. As can be seen in the subtitle this problem has two cases. One where the two columns are each others inverse, and one where they are not. In the first case, [0,a,-a], trifference is trivial because all columns have at least one nonzero index.
For the other case, first consider $1 < a < n, 1' < -b < n'$. Here it is evident that a and b differ in the first and/or the last row. Now take $a = 1$. Now $a_n = 1, b_n = 2$ for all $b \neq 2, n$, and $a_{n-1} = 1, b_{n-1} = 2$ for $b \in 2, n$. In the same way the cases for $a = n, b = 1', b = n'$ can be shown.

One column of A and two of -A ($[a, a', b']$ **or** $[a, b', c']$)

Again this problem consists of two parts. One where two of the columns are each others inverse, and one where they are not.
If two of the columns are each others inverse, trifference immediately follows because $a_b \neq -a_b \neq 0, b_b = 0$.
For the second part, use that it is known that a,b,c are trifferent. First consider the case where $0 = a_a \neq b_a \neq c_a$. It now immediately follows that $a_a \neq -b_a \neq -c_a$, which implies trifference. If this is not the case, it follows that $a_a \neq b_a = c_a = 2$. This is because there are no two columns that have a 1 above the diagonal in the same row. This implies that $a_b = a_c = -c_a = 1$ and that $-b_a = a_b = 1$. The 3×3 sub matrix consisting of the a'th b'th and c'th rows and columns of $a, -b, -c$ now looks like this:

$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & -c_b \\ 1 & -b_c & 0 \end{bmatrix}$$

We know that $-c_b \neq -b_c$, which implies that the three columns are trifferent at either the b'th or the c'th row of the matrix. It follows that any three vectors $[a, -b, -c]$ are trifferent and this implies that any three vectors $[a, b, -c]$ are trifferent as well. This proves Lemma 5.3, and therefore it also proves Theorem 5.1. □

5.3. General 'Trif-Matrices'

The definition of a Trif-Matrix might seem a bit arbitrary. Of course, this kind of matrix just happens to have some properties that result in a trifferent code. So what are these exact properties?
One of the first things that is important is the anti-symmetry of A. This is because if for some columns $i, j \in A$ we have $i_j = j_i$, most combinations of three rows and columns of A containing i and j will no longer be trifferent. This can be understood by looking at submatrix 5.1.
Considering only that A should be symmetric we could still take a matrix with only ones on the bottom half and twos on the top half. This construction fails as soon as we want to add the 0-vector to A. It's easy to see that any two adjacent columns within A are no longer trifferent with this 0-vector. This is why the anti-diagonal is added to A. However, the numbers that are flipped could also be chosen in another way. The only thing that needs to happen is that there is at least one flipped index within any pair of two adjacent matrices. For the case $n = 6$ for example, the following completions of a 6×6 matrix do also produce a trifferent code of length 13:

$$\begin{bmatrix} 0 & 2 & 2 & 2 & 2 & 2 \\ 1 & 0 & 2 & 2 & 1 & 2 \\ 1 & 1 & 0 & 2 & 2 & 2 \\ 1 & 1 & 1 & 0 & 2 & 1 \\ 1 & 2 & 1 & 1 & 0 & 2 \\ 1 & 1 & 1 & 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 2 & 1 & 1 & 1 & 1 \\ 1 & 0 & 2 & 1 & 1 & 1 \\ 2 & 1 & 0 & 2 & 1 & 1 \\ 2 & 2 & 1 & 0 & 2 & 1 \\ 2 & 2 & 2 & 1 & 0 & 2 \\ 2 & 2 & 2 & 2 & 1 & 0 \end{bmatrix}$$ [1]

[1]Any $n \times n$ matrix A with $n \geq 6$, constructed in the same way as the matrix on the right, together with the 0 vector and $-A$, will also form

However, not any random configuration of ones and twos will produce a trifference code. Some combinations of vectors fail for arbitrary configurations, which makes it nearly impossible to find a general rule for constructing trif-matrices.

5.4. Code

A Python code that constructs a trifference code of length $2n+1$ given any n in the way described above can be found in Appendix B.4.
The follow up question is, can we add vectors to the constructed code to build codes with a size larger than $2n+1$. In the code of B.4 the value of 'add' can be adjusted to make the code search a number of extra vectors. Doing this for small values of n results in the following extra vectors.

- $n = 6,7,8,9$: no extra vectors can be added
- $n = 10$: 2 extra vectors can be added (size = 23)
- $n = 11,12,13$: 4 extra vectors can be added (size = 27, 29, 31 respectively)
- $n \geq 14$: code takes too long to run

Since the codes that are constructed in this chapter have a very specific form it might be interesting to find a pattern within the vectors that can be added to them to be able to show that, for example $T(n) \geq 2n+5$ for $n \geq 11$. Due to a lack of time, however, we leave this to potential further research.

a trifferent code of length $2n-1$.

6 Another Explicit Construction

6.1. Introduction

The explicit construction method in the previous chapter only finds trifferent codes with a size that is linear in n. Since the lower bound of chapter 3 is asymptotic, this is not a very good way to find codes for larger values of n. The question therefore arises whether there is a better method to explicitly construct trifferent codes. The answer is yes, for specific values of n, where $n = k^2 : k \in \mathbb{N}$. This was achieved by S. Martirosan and S. Martirosan[13]. In this section their method will be described and compared with the method in the previous chapter.

6.2. The construction

Theorem 6.1. *For any integer j there exists a trifferent code with length j^2 and size 3^j.*

We will prove this theorem by explicitly constructing a trifferent code of this length and size.

First we define the following three matrices:

$$a = \begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 0 \\ 2 & 0 & 1 \end{bmatrix}, \quad c = \begin{bmatrix} 0 & 1 & 2 \\ 2 & 0 & 1 \\ 1 & 2 & 0 \end{bmatrix}$$

Note that the columns of any combination of two of these three matrices forms a trifferent code of length 3 and size 6. Denote by L_j the $3^j \times j$ matrix that consists of all vectors of length j with elements in $\{a, b, c\}$. For $j = 2$, for example we get:

$$L_2 = \begin{bmatrix} a & a & a & b & b & b & c & c & c \\ a & b & c & a & b & c & a & b & c \end{bmatrix}$$

The $3^j \times j^2$ matrix that contains the final trifferent code of length 3^j is denoted by M_j. The method to construct these codes is recursive, so we start with:

$$M_1 = \begin{bmatrix} 0 & 1 & 2 \end{bmatrix}$$

Which is a trifferent code of length 1 and size 3. Now construct the matrix

$$M_2' = \begin{bmatrix} M_1 \\ L_1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 \\ a & b & c \end{bmatrix}$$

From this matrix we can construct M_2 by tripling the rows of the first column of the matrix and substituting a, b and c by their corresponding matrix to obtain:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \\ 0 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 \\ 0 & 1 & 2 & 1 & 2 & 0 & 2 & 0 & 1 \\ 0 & 1 & 2 & 2 & 0 & 1 & 1 & 2 & 0 \end{bmatrix}$$

This is a slightly different version of the tetra code described in the introduction. We now need the following lemma:

Lemma 6.2. *If the $j^2 \times 3^j$ matrix M_j is trifferent, then the matrix:*

$$M''_j = \begin{bmatrix} M^*_{j-1} \\ L^*_{j-1} \end{bmatrix}$$

Is a trifferent code of size 3^j and length $(j-1)^2 + 3(j-1)$. Where M^*_{j-1} is obtained by tripling the columns of M_{j-1} and L^*_{j-1} is obtained by substituting a, b, c in L_{j-1} by their corresponding matrices.

Proof. Let $\overline{x} = \overline{x_m x_l}$, $\overline{y} = \overline{y_m y_l}$, $\overline{z} = \overline{z_m z_l}$ be three columns in M''_j. Here $\overline{x_m}$ is the part of $\overline{x}$ in M_{j-1} and $\overline{x_l}$ is the part in L_{j-1}Then we consider three cases:

- Let $\overline{x_m} \neq \overline{y_m} \neq \overline{z_m}$, then these three columns are trifferent because the matrix M_j is assumed to be trifferent
- Let $\overline{x_m} = \overline{y_m} = \overline{z_m}$, then any row within $\overline{x_l}, \overline{y_l}, \overline{z_l}$ is trifferent. This is because these are the columns of matrices a, b or c.
- Let any two out of the three vectors $\overline{x_m}, \overline{y_m}, \overline{z_m}$ coincide, but the third one not. Without loss of generality assume $\overline{x_m} = \overline{y_m} \neq \overline{z_m}$. Then in the matrix $\begin{bmatrix} \overline{x_l} & \overline{y_l} & \overline{z_l} \end{bmatrix}$ there is a row in which all indices are different. This follows from the fact that any combination of two matrices from a, b, c forms a trifferent code, as we stated before.

□

This proves Lemma 6.2. However, we want to find a code that has length j^2, and not a code of length $(j-1)^2 + 3(j-1)$. To do this, we have to see that the row of the form $(012012012\ldots012)^T$ occurs $j-1$ times in the matrix M''_j. This is because all matrices a, b, c start with a row $(0,1,2)^T$, and the last $(j-1)\cdot 3$ rows consist of only a, b and c matrices. We can remove all but one of these rows and still have a trifferent code, since if any three columns differ in one of these rows they differ in all of them. This means that we subtract $(j-1)-1$ rows. The total remaining rows is equal to:

$$(j-1)^2 + 3(j-1) - ((j-1)-1) = (j-1)^2 + 2(j-1) + 1 = j^2 - 2j + 1 + 2j - 2 + 1 = j^2$$

This proves Theorem 6.1.

6.3. A general bound

The method of Martirosan and Martirosan gives better values than the method described in the previous chapter, because it is not a linear bound. For example, for $n = 9$, the previous chapter calculates a size of $T(9) \geq 2\cdot 9 + 1 = 19$, while this chapter gives $T(9) \geq 3^3 = 27$. However, as stated before, this method can only construct codes for some values of n, specifically $n = k^2$ for $k \in \mathbb{N}$. However, we can use the following lemma to construct a general lower bound using the construction above.

Lemma 6.3. $T(n+1) \geq T(n) + 1$ *for $n \in \mathbb{N}$*

Proof. Define A to be a of length n and size $T(n)$. Now take some word $a \in A$ arbitrary. Take A' to be $A \setminus a$, and add a 2 at the end of all words in A'. Define a', a'' to be a with a 1 and a 0 added at the end respectively. Now $A' \cup \{a', a''\}$ form a trifferent set of length $n+1$ and size $T(n)+1$. This can be seen because both a' and a'' are already trifferent with all combinations of vectors in A', because A is a trifferent code. And a', a'' with some vector is A' are always different on the last index.
Using this lemma we can construct explicit codes for all n. These codes will have size $3^{n^*} + (n - n^*)$, where n^* is the largest square smaller than n. □

6.4. Another slight improvement

It is clear that big gaps arise in the value of $T(n)$ on the general bound of Section 6.3. For example, this bound now gives $T(15) \geq T(9) + 6 = 3^3 + 6 = 33$ and $T(16) \geq 3^4 = 81$. While it is hard to say something about the values in between $n = 9$ and $n = 16$, we can use Theorem 2.1 to say something about $T(14)$ and $T(15)$. It follows easily from this theorem that $T(15) \geq T(16)\cdot\frac{2}{3} = 54$ and $T(14) \geq T(15)\cdot\frac{2}{3} = 36$. This can be used for any values before n where $n = k^2, k \in \mathbb{N}$. However, this can only be used for the 2 preceding values, since $\left(\frac{3}{2}\right)^3 = 3.375 > 3$.

6.5. Comparison

In the following plot, a comparison can be seen between the method in this chapter and the one in Chapter 5.

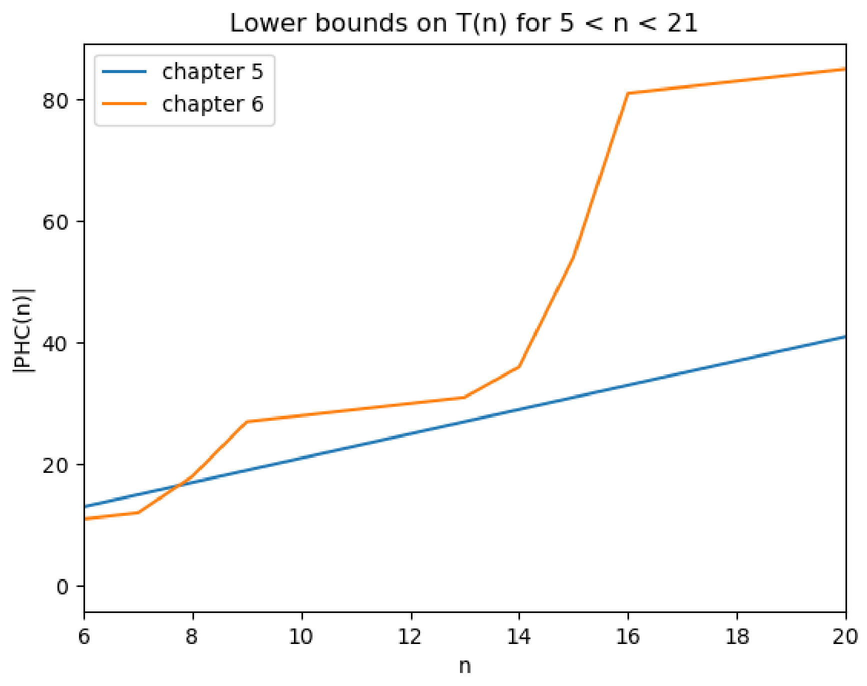


Figure 6.1: Lower bounds on $T(n)$ by the explicit constructions of Chapter 5 and 6

It is clear that the method in this chapter greatly outperforms the other method. This makes sense because the values of this method are of order $3^{\sqrt{n}}$ while the other ones are linear.
It should be remarked that there are other ways to explicitly construct trifferent codes [1], and that the method above is asymptotically not the best one.
Stinson et al [9], for example, found a way to explicitly construct trifferent codes of size 5^{2^j} and length $n = 3 \cdot 4^j$ for $j \geq 1$. Asymptotically this results in a bound of approximately $T(n) \geq 0.56 \cdot \log^2 n$. However, this explicit construction does not yield better results than the method of Martirosan et al [9] described in this chapter.
Asymptotically, the best known construction has been found by H.Wang and C.Xing [5]. They use algebraic curves to construct codes of size $3^{4 \cdot 3^{2j}}$ and length $n = 56 \cdot 8 \cdot 3^{2j}$ for $j \geq 1$. For large n this results in approximately $T(n) \geq 2^{\frac{n}{70}}$. The proof of these constructions is beyond the scope of this report.

[1]For explicit constructions of general Perfect K-Hash Codes, see [6],[10]

7 Small values

7.1. Introduction

In the past two sections we have looked at two different ways to find explicit trifferent codes. By constructing such an explicit code, we can form lower bound on the value of $T(n)$. In this section an overview will be presented of all known lower bounds on $T(n)$ obtained by constructing explicit codes for small values of n, using the methods in Chapters 5 and 6.

Remark. *The constructions by Stinson et al [9] and Chapoing et al[5], mentioned in Section 6.5, are not in this table. This is because the former is outperformed by the method of Chapter 6 for all n, and the latter only works for* $n \geq 4032$.

7.2. Construction

In Table 7.1 the overview of all lower bounds on $T(n)$ can be seen that can be found by explicitly constructed codes. The background colors of the table show how these values were achieved.

- **Red** values are the known values, found in Chapter 4.
- **Yellow** Values were obtained by the linear algorithm in Chapter 5.
- **Blue** Values were obtained by the explicit construction method in Section 6.2
- **Green** Values were obtained by using the general bound of Section 6.3
- **Pink** values were obtained by the improvement of Section 6.4.

n	T(n)	n	T(n)	n	T(n)	n	T(n)
1	**3**	2	**4**	3	**6**	4	**9**
5	**10**	6	**13**	7	**15**	8	**18**
9	**27**	10	**28**	11	**29**	12	**30**
13	**31**	14	**36**	15	**54**	16	**81**
17	**82**	18	**83**	19	**84**	20	**85**
21	**86**	22	**85**	23	**108**	24	**162**
25	**243**	26	**244**	27	**245**	28	**246**
29	**247**	30	**248**	31	**249**	32	**250**
33	**251**	34	**342**	35	**486**	36	**729**

Table 7.1: Lower bounds on $T(n)$ by explicitly constructed codes

The table will go on in this manner, simply adding 1 to $T(n)$ unless n is (close to) a square. So the construction of Chapter 6.2. It will go on like this until $n = 36288$, where the construction by Wang et al [9] will give a value of 3^{244}, which is better than the value of approximately 3^{190}, given by the method of Chapter 6.

It should be noted that there exist other explicit constructions which are not considered in this paper. It is possible that some of these constructions give better lower bounds on $T(n)$ for certain values of n than the values given above.

8 Conclusion and Discussion

This thesis contains a detailed reconstruction of the proofs for the best known asymptotic upper and lower bound on the trifference problem (Chapters 2 and 3). Also, the report contains a Python code that can be used to find all exact values of $T(n)$ for $n \leq 6$, using the properties of Stefano et al[2] . On top of that some new properties were added to make the code run even faster. (Chapter 4)
From the structures of the found trifference codes of length 6 a general way was found to explicitly construct trifference codes of length $2n+1$. (See Chapter 5). However, it was not achieved to completely generalize any code that can be build in this way. It was considered what kind of vectors can be added to these kind of codes to construct larger codes, but a generalization for this was not found as well.
After this, the result of $2n+1$ was compared to an explicit construction by Martirosyan et al [13]. The comparison demonstrated that this explicit construction outperformed the construction of $2n+1$ for almost any value of n. A reproduction of this construction can be found in Chapter 6. Two other explicit methods are mentioned and compared to as well [5][9].
In chapter 7 an overview of all known explicit lower bounds on trifferent codes for small n is given using chapters 4, 5 and 6.
Regrettably, the explicit construction of Chapter 5 turned out to be beaten by other already existing constructions. Therefore it might have been better to do a more thorough literature study on other explicit constructions beforehand. This way, perhaps more time could have been spend on finding other new insights on other areas within the problem. However, by exploring the problem with only little background information one might address the problem with more creativity and find new approaches or structures that were not found before. Even though the size the codes described in Chapter 5 is beaten by other constructions, their 'anti-symmetric' structure remains interesting and might become useful in future work.
Therefore, it could be interesting to further analyze the codes that were found in Chapter 5. Their structure could could help in a better understanding of the nature of trifference codes or perfect k-hashing codes in general. It might also be interesting to aim for a generalization of the explicit construction of Chapter 6, since the one given is this paper is very elementary. On the other hand it could be a challenge to find even more new properties of trifference codes to further optimize the code in section 4 to find exact values of $T(n)$ for $n \geq 7$. If one has enough resources, it could also be possible to calculate this value of $T(7)$ with the code in 4.3, by using parallel computing.
Apart from that, finding new asymptotic upper and/or lower bounds on $T(n)$, either by explicit constructions or other approaches, remains the biggest challenge within the trifference problem.

A

Appendix A: Explicit codes for $1 \leq n \leq 6$

$n = 1, T(n) = 3$

$$\begin{bmatrix} 0 & 1 & 2 \end{bmatrix}$$

$n = 2, T(n) = 4$

$$\begin{bmatrix} 0 & 0 & 1 & 2 \\ 0 & 1 & 2 & 2 \end{bmatrix}$$

$n = 3, T(n) = 6$

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 \\ 0 & 1 & 2 & 0 & 1 & 2 \\ 0 & 2 & 2 & 1 & 1 & 0 \end{bmatrix}$$

$n = 4, T(n) = 9$

$$\begin{bmatrix} 0 & 0 & 2 & 2 & 2 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 2 & 1 & 1 & 0 & 1 & 2 \\ 0 & 1 & 1 & 0 & 2 & 1 & 1 & 0 & 1 \\ 0 & 1 & 2 & 1 & 0 & 1 & 2 & 1 & 0 \end{bmatrix}$$

$n = 5, T(n) = 10$

$$\begin{bmatrix} 0 & 0 & 2 & 2 & 2 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 2 & 1 & 1 & 0 & 1 & 2 & 2 \\ 0 & 1 & 1 & 0 & 2 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 2 & 1 & 0 & 1 & 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \end{bmatrix}$$

$n = 6, T(n) = 13$

$$\begin{bmatrix} 0 & 0 & 2 & 2 & 2 & 2 & 2 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 2 & 2 & 2 & 1 & 2 & 0 & 1 & 1 & 1 & 2 \\ 0 & 1 & 1 & 0 & 2 & 1 & 2 & 2 & 2 & 0 & 1 & 2 & 1 \\ 0 & 1 & 1 & 1 & 0 & 2 & 2 & 2 & 2 & 2 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 & 1 & 0 & 2 & 2 & 2 & 1 & 2 & 0 & 1 \\ 0 & 1 & 2 & 1 & 1 & 1 & 0 & 2 & 1 & 2 & 2 & 2 & 0 \end{bmatrix}$$

B

Appendix B: Python Code

```python
def checker(A,B,C):              #Fastest way to check if three words are trifferent
    for i in range(len(A)):
        if(len(set([A[i],B[i],C[i]])) == 3):
            return True
    return False

def is_trifferent(A):        #Checks whether a given code is triffernt
    l = len(A)
    k = 3
    if checker(A[0],A[1],A[2]):      #If the first three elements are trifferent
        while k < l:
#We now know that the first k-1 elements are trifferent, so we only check if the kth
    element is trifferent with the others
            for i in range(k-1):
                for j in range(i+1,k):
                    #print(A[i],A[j],A[k])
                    if not checker(A[i],A[j],A[k]):
                        return False
            k+=1                        #We add to k until we have reached the length of A
        return True
    else:
        return False

def is_trifferent_2(A,B):   #Checks whether adding a word B to a trifferent code A will
     give a trifferent code as well
    for i in range(len(A)-1):
        for j in range(i+1, len(A)):
            if not checker(A[i],A[j], B):
                return False
    return True
```

Listing B.1: Some basic code that checks trifference, which is used in the other codes

```python
from Trif_Checker import is_trifferent, checker, is_trifferent_2

# n is the code length, and M is the requested size
n = 6
M = 13

def wordset_generator(n):   #generates all possible words of length n, ordered from
    0,...,0 to 2,...,2
    a = [range(0,3)]*n
    return list(tuple(map(list, itertools.product(*a))))

WordSet = wordset_generator(n)
zero = [0] * n #The zero vector
zerostop = 3**(n-1)      #The index where all words that start with a 1 begin
onestop = 2*3**(n-1)     #The index where all words that start with a 2 begin
twostop = len(WordSet)  #The last index
```

```python
firstindex = []
c = 0
for i in range(n):
    c+= i**3
    firstindex.append(c)
    #Adding all possible words of the form [0,...,0,1,...,1]

#Here the possible densities of the first number are decided, given n and M
if n == 6:
    if M == 14:
        possibleDensities= [[6,4,4], [5,5,4]];
    elif M == 13:
        possibleDensities = [[7,3,3], [6,4,3], [5,5,3], [5,4,4]];
elif n == 5:
    if M == 10:
        possibleDensities = [[8,1,1], [7,2,1], [6,3,1], [6,2,2], [5,4,1], [5,3,2], [4,4,2], [4,3,3]];
    elif M == 11:
        possibleDensities = [[7,2,2], [6,3,2], [5,4,2], [5,3,3], [4,4,3]];
elif n == 4:
    if M == 9:
        possibleDensities = [[3,3,3]];
    else:
        possibleDensities = []
elif n == 7:
    if M = 16:
        possibleDensities = [[10,3,3], [9,4,3],[8,4,4],[8,5,3],[7,6,3],[7,5,4],[6,5,5]]
    elif M= 17:
        possibleDensities = [[9,4,4],[8,5,4], [7,6,4], [7,5,5] [6,6,5]
    elif M = 18:
        possibleDensities = [[8,5,5],[7,6,5],[6,6,6]]


#The function 'recursive' recursively checks whether trifferent sets of length n and size M exist
#index = index within the wordset, trifset = current set of trifferent words, WordSet = wordset, density = in possibleDensities
def recursive(index, trifset, WordSet, n, M, density):
    if len(trifset) == M:   #If len(trifset) == M) we have found a trifferent code of lenght M!
        print(trifset)
    if len(trifset) >= density[1]+density[0]: #If this statement is true, the next word starts with a 2
        countlist = []  #The countlist keeps track of all possible lenghts of trifferent codes using the current one
        cont = False    #Continue; if this statement remains False there are no further words possible to add to the code
        for k in range(max(index, one), twostop):   #Check all words with greater index that start with 2
            if is_trifferent_2(trifset, WordSet[k]):    #is_trifferent_2 checks trifference
                cont = True
                countlist.append(recursive(k, trifset+ [WordSet[k]], WordSet, n,M, density)) #Recursively add the value of the amount of words that can be added
        if cont == False:   #If there are no words to be added the value is 0
            return 0
        else:               #If there were words that could be added the maximum amount of words is equal to the maximal over all k + the word itself
            return max(countlist) + 1
    elif len(trifset) >= density[0]:    #If this statement is true, the next word starts with a 1
        countlist = []
        cont = False
        for k in range(max(index, zerostop), onestop):   #Check all words with greater index that start with 1
            if is_trifferent_2(trifset, WordSet[k]):
                cont = True
                countlist.append(recursive(k, trifset+ [WordSet[k]], WordSet, n,M, density))
        if cont == False:
            return 0
        else:
            return max(countlist) + 1
    else:#The words starts with a 0
        countlist = []
        cont = False
        for k in range(index, zerostop):                   #Check all words with greater index that start with 0
            if is_trifferent_2(trifset, WordSet[k]):
                cont = True
                countlist.append(recursive(k, trifset+ [WordSet[k]], WordSet, n,M, density))
        if cont == False:
            return 0
        else:
            return max(countlist) + 1

#Run the algorithm for all possible densities and all values in firstindex
for density in possibleDensities:
    for i in firstindex:
        print("density =", density, "firstindex =", firstindex, "maximum size=", recursive(9, [[0,0,0,0,0,0],firstindex], wordset_generator(n), n,M, density) + 2)
        #Add 2 because we start with 2 words in the basis
```

Listing B.2: Python code for finding exact values of $T(n)$ for $n = 4, 5, 6$

```python
from Trif_Checker import is_trifferent, checker, is_trifferent_2
from Equivalent_Codes_Generator import Eq_codes_5, Eq_codes_4, Eq_codes_6
import itertools

n = 6
M = 13

def wordset_generator(n):   #generates wordset of length n
    a = [range(0,3)]*n
    return list(tuple(map(list, itertools.product(*a))))

WordSet = wordset_generator(n)
zero = [0] * n
zerostop = 3**(n-1)
onestop = 2*3**(n-1)
twostop = len(WordSet)
firstindex = []
c = 0
for i in range(n-1):        #generate indices of vectors of the form (0,...,0,1,...,1)
    c+= 3**i
    firstindex.append(c)

def zero_counter(word):     #counts the number of zeros within a word
    counter = 0
    for i in word:
        if i == 0:
            counter += 1
    return counter

def check_descending(word):     #checks whether the values in a word a in descending order
    for i in range(len(word)-1):
        if word[i] < word[i+1]:
            return False
    return True

def WordSet_Adjust(WordSet, firstindex):    #Removes the words with too many zeros, from the total wordset
    zeros = zero_counter(WordSet[firstindex])
    NewWordSet = []
    for word in WordSet:
        if zero_counter(word) <= zeros:
            NewWordSet.append(word)
    onestart = 0                        #onestart and twostart keep track from after which value the words begin with a 1/2
    twostart = 0
    for word in NewWordSet:
        if word[0] == 0:
            onestart += 1
            twostart += 1
        if word[0] == 1:
            twostart += 1
    return (NewWordSet, onestart, twostart)

def secondindex(firstindex, n, WordSet):    #Generates a list of all vectors for v_2 with indices, given v_1
    zeros = zero_counter(WordSet[firstindex])#number of zeros in v_1
    ones = n - zeros                        #number of ones in v_1
    secondindex  = []                       #list that contains all v_2
    countlist = []                          #list that contains indices for v_2 within the toatl WordSet
    count = 0
    for word in WordSet:
        count += 1
        if word[zeros] == 2:                #We want a two at the first index where the ones start, since they are in descending order
            if zeros == 1:                  #Two separate statements for if there is only one zero, since max() does not work then
                if word[0] == 0:
                    if check_descending(word[1:zeros]):
                        if check_descending(word[zeros:]):
                            secondindex.append(word)
                            countlist.append(count-1)
            elif max(word[1:zeros]) < 2:
                if word[0] == 0:
                    if check_descending(word[1:zeros]):
                        if check_descending(word[zeros:]):
                            secondindex.append(word)
                            countlist.append(count-1)
    return (secondindex, countlist)


if n == 6:          #Possible densities given n and N
    if M == 14:
        possibleDensities= [[6,4,4], [5,5,4]];
    elif M == 13:
        possibleDensities = [[7,3,3], [6,4,3], [5,5,3], [5,4,4]];
elif n == 5:
    if M == 10:
        possibleDensities = [[8,1,1], [7,2,1], [6,3,1], [6,2,2], [5,4,1], [5,3,2], [4,4,2], [4,3,3]];
    elif M == 11:
        possibleDensities = [[7,2,2], [6,3,2], [5,4,2], [5,3,3], [4,4,3]];
elif n == 4:
    if M == 9:
        possibleDensities = [[3,3,3]];
    else:
        possibleDensities = []
elif n == 7:
    if M == 17:
        possibleDensities = [[7,5,5],[6,6,5]]

def recursive(index, trifset, WordSet, n, M, density, onebegin, twobegin):      #function that generates all possible trifferent sets, same as previous code
    if len(trifset) == M:
        print(trifset)

    if len(trifset) >= density[1]+density[0]:
        countlist = []
        flip = False
        for k in range(max(index, twobegin), len(WordSet)):
            if is_trifferent_2(trifset, WordSet[k]):
                flip = True
                countlist.append(recursive(k, trifset+ [WordSet[k]], WordSet, n,M, density, onebegin, twobegin))
        if flip == False:
            return 0
        else:
            return max(countlist) + 1
    elif len(trifset) >= density[0]:
        countlist = []
        flip = False
        for k in range(max(index, onebegin), twobegin):
            if is_trifferent_2(trifset, WordSet[k]):
                flip = True
                countlist.append(recursive(k, trifset+ [WordSet[k]], WordSet, n,M,
    density, onebegin, twobegin))
        if flip == False:
            return 0
        else:
            return max(countlist) + 1
    else:
        countlist = []
        flip = False
        for k in range(index, onebegin):
            if is_trifferent_2(trifset, WordSet[k]):
                flip = True
                countlist.append(recursive(k, trifset+ [WordSet[k]], WordSet, n,M,
    density, onebegin, twobegin))
        if flip == False:
            return 0
        else:
            return max(countlist) + 1

for density in possibleDensities:
    for i in firstindex:
        NewWordSet = WordSet_Adjust(WordSet,i)[0]
        onebegin = WordSet_Adjust(WordSet,i)[1]         #This is the index from which
    words start with a 1, to use in densities
        twobegin = WordSet_Adjust(WordSet,i)[2]         #''     '' ''
     ''                2
        A = secondindex(0,n,NewWordSet)
        for j in range(len(A[0])):
            print(density, WordSet[i], A[0][j], recursive(A[1][j], [zero,WordSet[i], A
    [0][j]], NewWordSet, n,M, density, onebegin, twobegin) + 3)
```

Listing B.3: Improved python code from section 4.3, to show $T(6) = 13$

```
from Trif_Checker import is_trifferent, checker, is_trifferent_2
import itertools

n =   10  #imput the length here
add = 2   #imput amount of words to be added here

def wordset_generator(n):   #generates wordset of length n
    a = [range(0,3)]*n
    return list(tuple(map(list, itertools.product(*a))))

def Agenerator(n):      #This generates a 'trif'-matrix A given n
    A = []
    for i in range(n):
        newvector = [2]*i
        newvector.append(0)
        newvector_2 = [1]*(n-i-1)
        vector = newvector + newvector_2
        A.append(vector)
        index = n-i
        if index < n:
            if vector[index] == 2:
                vector[index] = 1
            elif vector[index] == 1:
                vector[index] = 2
    half = int((n-1)/2)
    A[half][0] = 1
    A[0][half] =2
    return A
```

```python


def codegenerator(A):   #This generates a trifference code given A is a 'trif' matrix
    wordlength = len(A[0])
    newA = A.copy()
    for j in range(len(A)):
        newword = A[j].copy()
        for i in range(wordlength):
            if A[j][i] == 1:
                newword [i]= 2
            elif A[j][i] == 2:
                newword[i] =1
        newA.append(newword)
    newA.append([0]*wordlength) #Add the 0 vector
    return newA

wordset = wordset_generator(n)
Matrix = Agenerator(n)
Code = codegenerator(Matrix)#This generates a trifference code of length n and size 2n+1
trifset = [Code.copy()]


for i in range(add):    #This code checks whether it is possible to add more vectors to the code of length 2n+1
    newtrifset = []
    newwordset = []
    length = n + i
    for code in trifset:
        index = wordset.index(code[-1]) #Take the index of the last added word, which is 0 for the begincode
        for word in wordset[index::]:
            if is_trifferent_2(code, word):
                newcode = code.copy()
                newcode.append(word)
                print(newcode[-1], len(newcode))
                newtrifset.append(newcode)
                if word not in newwordset:
                    newwordset.append(word)
    trifset = newtrifset
    wordset = newwordset
```

Listing B.4: Python code that generates a trifference code of length $2n+1$, given, any n, and checks whether it is possible to add more vectors to the code.

Bibliography

[1] N. Alon and M. Naor. Derandomization, witnesses for boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, 16:434–449, 1996.

[2] S. D. Fiore, A. Gnutti, and S. Polak. The maximum cardinality of trifferent codes with lengths 5 and 6. *Examples and Counterexamples*, 2, 2022. doi: 10.1016/j.exco.2022.100051.

[3] M. Fredman and J. Komlós. On the size of separating systems and families of perfect hash functions. *SIAM Journal on Algebraic Discrete Methods*, 5(1):61–68, 1984. doi: 10.1137/0605009.

[4] Venkatesan Guruswami and Andrii Riazanov. Beating fredman-komlós for perfect k-hashing. *Journal of Combinatorial Theory, Series A*, 188, 2022. doi: 10.48550/arXiv.1805.04151.

[5] Xing H. Wang, Chaoping. Explicit constructions of perfect hash families from algebraic curves over finite fields. *Journal of Combinatorial Theory, Series A*, 93:112–124, 2001.

[6] R. A. Walker II and C. J. Colbourn. Perfect hash families: Constructions and existence. *Journal of Mathematical Cryptology*, 1:125 – 150, 2007. doi: 10.1515/JMC.2007.008.

[7] J. Körner. Coding of an information source having ambiguous alphabet and the entropy of graphs. *6th Prague conference on information theory*, pages 411–425, 1973.

[8] J. Körner and K. Marton. New bounds for perfect hashing via information theory. *European Journal of Combinatorics*, 9:523–530, 1988.

[9] D.R. Stinson M. Atici, S.S. Magliveras and W.-D. Wei. Some recursive constructions for perfect hash families. *Journal of Combinatorial Designs*, 4(5):353–363, 1996. doi: 10.1006.jcta.2000.306.

[10] S. Martirosyan and T. van Trung. Explicit constructions for perfect hash families. *Designs, Codes and Cryptography*, 46:97–112, 2008. doi: 10.1007/s10623-007-9138-6.

[11] K. Melhorn. Data structures and algorithms: 1. searching and sorting. *Springer*, 84:90, 1984.

[12] I. Newman and A. Wigderson. Lower bounds on formula size of boolean functions using hypergraph entropy. *SIAM Journal on discrete mathematics*, 8(4):536–542, 1995.

[13] S. Martirosyan S. Martirosyan. New upper bound on the cardinality of a k-separated set or perfect hash family and a near optimal construction for it. *Mathematical issues of cybernetics and computer technology*, 21:104–115, 2000.

[14] Y. Desmedt S. R. Blackburn, M. Burmester and P. R. Wild. Efficient multiplicative sharing schemes. *Advances in Cryptology- EUROCRYPT'96*, pages 107–118, 1996.