The extended k-tree algorithm*

Lorenz Minder† Alistair Sinclair‡

Abstract

Consider the following problem: Given $k = 2^q$ random lists of n-bit vectors, $L_1, \ldots, L_k$, each of length m, find $x_1 \in L_1, \ldots, x_k \in L_k$ such that $x_1 + \cdots + x_k = 0$, where $+$ is the XOR operation. This problem has applications in a number of areas, including cryptanalysis, coding theory, finding shortest lattice vectors, and learning theory. The so-called *k-tree algorithm*, due to Wagner, solves this problem in $\tilde{O}(2^{q+n/(q+1)})$ expected time provided the length m of the lists is large enough, specifically if $m \geq 2^{n/(q+1)}$.

In many applications, however, it is necessary to work with lists of smaller length, where the above algorithm breaks down. In this paper we generalize the algorithm to work for significantly smaller values of the list length m, all the way down to the threshold value for which a solution exists with reasonable probability. Our algorithm exhibits a tradeoff between the value of m and the running time. We also provide the first rigorous bounds on the failure probability of both our algorithm and that of Wagner.

As a third contribution, we give an extension of this algorithm to the case where the vectors are not binary, but defined over an arbitrary finite field $\mathbb{F}_r$, and a solution to $\lambda_1 x_1 + \cdots + \lambda_k x_k = 0$ with $\lambda_i \in \mathbb{F}_r^*$ and $x_i \in L_i$ is sought.

Keywords: k-sum problem; time-space tradeoff; birthday problem; collision search; finding low-weight codewords; correlation attack; sparse polynomials.

1 Introduction

1.1 Background

The *k-sum problem* is the following. We are given k lists $L_1, \ldots, L_k$ of n-bit vectors, each of length m and chosen independently and uniformly at random, and we want to find one vector from each list such that the XOR of these k vectors is equal to zero, i.e., find $x_1 \in L_1, \ldots, x_k \in L_k$ such that

$$x_1 + x_2 + \cdots + x_k = 0.$$

For simplicity, we will take $k = 2^q$ to be a power of two.

This problem, which can be viewed as a k-dimensional variant of the classical birthday problem, arises in various domains. For example, Wagner [13] presents a number of applications in cryptography, while a recent paper of Coron and Joux [7] shows how to use the k-sum problem to find

*A preliminary version of this paper appeared as [11]

†Computer Science Division, University of California, Berkeley CA 94720-1776, U.S.A. Email: lorenz@eecs.berkeley.edu. Supported by grant PBEL2–120932 from the Swiss National Science Foundation, and by NSF grants 0528488 and 0635153.

‡Computer Science Division, University of California, Berkeley CA 94720-1776, U.S.A. Email: sinclair@cs.berkeley.edu. Supported in part by NSF grant 0635153 and by a UC Berkeley Chancellor's Professorship.

codewords in a certain context. Other applications include finding shortest lattice vectors [1, 9], solving subset sum problems [10], and statistical learning [3].

The k-sum problem is of course only interesting when a solution does indeed exist with reasonable probability. A necessary condition for this is $m^{2^q} \geq 2^n$, i.e.,

$$m \geq 2^{n/2^q}. \tag{1.1}$$

(This condition ensures that the expected number of solutions is at least 1.) Hence we will always assume that (1.1) holds.

A naïve algorithm for solving this problem works as follows. Compute a list S_1 of sums $x_1 + \cdots + x_{2^{q-1}}$, and a list S_2 of sums $x_{2^{q-1}+1} + \cdots + x_{2^q}$, where $x_i \in L_i$. (The summands x_i can be chosen in any way, provided only that no two sums are identical.) Then any vector appearing in both S_1 and S_2 yields a solution; such a vector can be found in time essentially linear in the lengths of S_1 and S_2. In order to keep the success probability reasonably large, we must ensure that a collision is likely to exist in S_1 and S_2. The birthday paradigm tells us that it suffices to take $|S_1|, |S_2| = \Theta(2^{n/2})$, resulting in an algorithm with running time $\tilde{O}(2^{n/2})$.[†]

In the case where condition (1.1) holds with equality, this is also the best known algorithm. But it turns out that (for $q > 1$) we can do much better if a stronger condition holds. Wagner [13] showed that if

$$m \geq 2^{n/(q+1)} \tag{1.2}$$

then the problem can be solved in expected time $\tilde{O}(2^{q+n/(q+1)})$. The algorithm that achieves this is called the "k-tree algorithm."

To illustrate the main idea behind this algorithm, consider the case $k = 4$. Let $L_1, \ldots, L_4$ be four lists of length $m = 2^{n/3}$ each. (Here we have chosen m so that (1.2) holds with equality.) We proceed in two rounds. In the first round, we compute a list L'_1 that contains all sums $x_1 + x_2$ with $x_1 \in L_1$ and $x_2 \in L_2$ such that the first $n/3$ bits of the sum are zero. Similarly, we compute a list L'_2 of all sums of vectors in L_3 and L_4 such that the first $n/3$ bits are zero. Then the expected length of L'_1 (and analogously of L'_2) is

$$2^{-n/3} \cdot |L_1| \cdot |L_2| = 2^{n/3}.$$

In the second round, we find a pair $x'_1 \in L'_1$ and $x'_2 \in L'_2$ such that $x'_1 + x'_2 = 0$. Since any sum of elements in L'_1 and L'_2 will be zero on the first $n/3$ bits, the probability that a random sum $x'_1 + x'_2$ equals zero is $2^{-2n/3}$. Therefore, the expected number of matching sums is

$$2^{-2n/3}\mathrm{E}[|L'_1|]\mathrm{E}[|L'_2|] = 2^{-2n/3}2^{n/3}2^{n/3} = 1,$$

so we expect the algorithm to find a solution. The lists L'_1 and L'_2 can both be computed in time $\tilde{O}(2^{n/3})$, as can the final set of matches. Hence this algorithm has an expected running time of $\tilde{O}(2^{n/3})$, which is significantly smaller than the $\tilde{O}(2^{n/2})$ time required by the naïve algorithm.[‡]

A major limitation of the k-tree algorithm is that it breaks down when (1.2) fails to hold. For applications where it is possible to either increase the length of the lists, m, or increase the number of lists, k, this is not a problem, since we can then always arrange for (1.2) to hold. This point of

[†]In this paper, the notation $\tilde{O}$ hides factors that are logarithmic in the running time—i.e., polynomial in $n, \log m$ and q.

[‡]Throughout the paper, expectations are taken over the random input lists. The algorithms are deterministic.

view is taken in Wagner's paper [13], where it is assumed in particular that the list length m can be made as large as desired.

However, in many applications, q, m and n are given values that cannot be varied at will. One example of such a setting is the cryptographic attack against code-based hash functions presented by Coron and Joux [7], where the values of n, q and m are given by the designer of the hash function and the attacker cannot change them. Another example is the problem of finding a sparse feedback polynomial for a given linear feedback shift register, as discussed in [13]. Here q is fixed, since it determines the Hamming weight of the polynomial to be found. Increasing the list length m has the effect of increasing the degree of the polynomial being sought. Now if the sparse polynomial is to be used in a correlation attack, then its degree must not exceed the amount of known running-key data, and so in practice it cannot be arbitrarily large. Consequently, the value of m should also be considered fixed in this application.

Motivated by such examples, in this paper we consider the k-sum problem with the values of q, m and n fixed (subject only to the non-triviality requirement (1.1)). Our goal is to find a solution $x_1 + \ldots + x_k = 0$ as quickly as possible in this constrained setting.

1.2 Results

We first show that the k-tree algorithm can be generalized to work for any set of parameter values satisfying the condition (1.1) for existence of a solution, i.e., for all values of m satisfying

$$2^{n/2^q} \leq m \leq 2^{n/(q+1)}.$$

(For larger values of m, the original k-tree algorithm applies.) As we will see, the price we pay for decreasing m in this range is a larger running time: the exponent of the running time decreases with $\log m$ in a continuous, convex and piecewise linear fashion. Our algorithm can be seen as interpolating between Wagner's k-tree algorithm and the naïve algorithm: at one extreme ($m = 2^{n/(q+1)}$) it becomes the k-tree algorithm, and at the other ($m = 2^{n/2^q}$) it becomes the naïve algorithm.

The idea behind our modification (which we call the "extended k-tree algorithm") is the following. We can think of the original k-tree algorithm as *eliminating* (i.e., finding vectors that sum to zero on) a fixed number $\log m$ bits in each round (except for the last round, where $2 \log m$ bits are eliminated)[§]. This choice keeps the list length constant over all rounds, thereby balancing the work done in each round. (See section 2 for a more precise description of the k-tree algorithm.) While this guarantees a minimum maximal list length, it also entails the strong requirement (1.2). In our extension, we vary the number of bits eliminated (and thus the intermediate list lengths) in each round, in such a way that ultimately more bits can be eliminated in total.

To illustrate how this can help, consider again the $k = 4$ example from earlier, and suppose now that we take a smaller value of m, say $m = 2^{2n/7}$ instead of $m = 2^{n/3}$. (Note that this value takes us outside the scope of Wagner's algorithm, but is still within the existence bound (1.1).) If we eliminate $\ell_1 = n/7$ bits in the first round (instead of $n/3$ as previously), then $\mathrm{E}[|L'_1|] = 2^{-n/7}|L_1||L_2| = 2^{3n/7}$. We then eliminate the remaining $\ell_2 = 6n/7$ bits in the second round, giving us an expected number of solutions equal to

$$2^{-6n/7}\mathrm{E}[|L'_1|]\mathrm{E}[|L'_2|] = 2^{-6n/7}2^{6n/7} = 1;$$

[§]Throughout the paper, log denotes base-2 logarithm unless otherwise stated

thus we again expect the algorithm to find a solution. This gives an algorithm with expected running time $\tilde{O}(2^{3n/7})$ for this particular set of parameters, which is still significantly better than the $\tilde{O}(2^{n/2})$ naïve algorithm.

The key step in designing our algorithm is to specify an optimal strategy for choosing the expected list lengths (or equivalently, the number of bits to be eliminated) in each round. We do this by formulating this optimization problem as an integer program, which we are then able to solve analytically. Perhaps surprisingly, the optimal strategy turns out to be to let the lists grow in the first few rounds without eliminating any bits, and then to switch to a second phase in which a fixed number of bits are eliminated in each round. The role of the first phase is apparently to simply increase the pool of vectors (by summing combinations from the original lists) until the number of vectors is large enough for the elimination phase to work successfully.

We then go on to address the failure probability of the algorithm. Note that both our algorithm as described above, and Wagner's original k-tree algorithm, are based only on an analysis of the *expected* number of solutions found, which says nothing useful about the probability that a solution is actually found. In the last section of the paper, we give the first rigorous bound on this probability. Our analysis, which uses the second moment method, applies to both Wagner's algorithm and our extension. The upshot is that, for a wide range of parameters, if one naïvely aims for a single solution in expectation, then the failure probability will be at most slightly larger than 3/4. Moreover, at the cost of a small increase in running time, the failure probability can be reduced substantially.

In the final part of the paper, we present a modification of the algorithm that can be used to solve instances where the lists contain vectors over an arbitrary finite field $\mathbb{F}_r$ rather than over $\mathbb{F}_2$. In this case the problem is generalized to that of finding a suitable linear combination (with coefficients in $\mathbb{F}_r^*$) of vectors summing to zero. Such an algorithm can be used, for example, to find low-weight codewords in non-binary linear codes, or to compute sparse multiples of polynomials in $\mathbb{F}_r[X]$.

1.3 Related work

The basic idea of the k-tree algorithm was apparently rediscovered several times. In 1991, Camion and Patarin [5] constructed a k-tree scheme for breaking knapsack-based hash functions. In 2000, Blum, Kalai and Wasserman [3] devised a similar algorithm to prove a conjecture in learning theory. In 2002, Wagner [13] published a paper dedicated entirely to the k-tree algorithm, including some extensions and several applications.

In the same year, Chose, Joux and Mitton [6] proposed an algorithm for finding low weight parity checks for a linear feedback shift register. Their algorithm is similar to the 4-tree algorithm. Unlike the other authors, Chose *et al.* propose a scheme where the number of eliminated bits varies from round to round. However, their motivation for doing so is quite different from ours, leading to very different results: unlike the k-tree algorithm, their algorithm finds *all* the solutions, and their choice of parameters is designed so as to minimize the memory use without sacrificing too much speed. Our goal, on the other hand, is to find only a single solution, and we choose the parameters so as to minimize running time (and memory use) for that purpose.

In 2004, Coron and Joux [7] used Wagner's algorithm to break a hash function based on error correcting codes. Since Wagner's condition (1.2) does not hold in their case, they tweaked the algorithm by removing one level of the tree and working on lists that were sums of pairs of vectors. This strategy is a special case of our algorithm, and hence can be viewed as an interesting application of the extended k-tree algorithm. The attack by Coron and Joux was subsequently refined by Augot,

Sendrier and Finiasz [8] to a variant that does not always eliminate the same number of bits per round.

We are not aware of any previous analysis of the failure probability of the original k-tree algorithm; however, some modified versions have been analyzed, as we now discuss.

First, Blum, Kalai and Wasserman [3] analyzed a related algorithm, which differs from Wagner's algorithm in that it searches for collisions in a single list. Another difference is that only a subset of the valid pairs is selected in the merging step. In 2005, Lyubashevsky [10] analyzed a variant of Wagner's algorithm devised to solve the integer subset-sum problem; thus the list elements are integers mod t rather than bitstrings. As in the Blum *et al.* algorithm, only a subset of the valid pairs is used when merging. In this construction, the length of the lists has to be roughly the square of the length that Wagner's algorithm prescribes. In 2008, Shallue [12] modified Lyubashevsky's algorithm so that the merging step selects a larger subset of valid pairs. As a result, in order to achieve non-trivial failure probability the lists need to be of length $O(m \log m)$ where m is the length required by Wagner's algorithm.

A key difference between all these three constructions and Wagner's original algorithm is that the list merging step does not select all valid pairs. This has two drawbacks. First, it results in an inflation of the list length (and hence running time) relative to Wagner's algorithm. Second, in these constructions the merge cannot possibly expand the list length, which is a key ingredient of our extended algorithm.

Finally, we briefly mention an alternative approach to the k-sum problem which is applicable in the regime where $k \geq n$ (which typically does not hold in the kind of applications mentioned earlier). Bellare and Micciancio [2] show that in this scenario the k-sum problem can be solved by Gaussian elimination in time $O(n^3 + kn)$.

2 The extended k-tree algorithm

In this section we present a framework for our extended k-tree algorithm; as we shall see, the original k-tree algorithm of Wagner [13] is a special case.

Given an instance of the k-sum problem as described in the Introduction, the (extended) k-tree algorithm proceeds in q rounds, where $k = 2^q$ is the number of input lists. In each round, pairs of lists are merged to form a new list, so that the number of lists is halved in each round. For example, in the first round the lists L_1 and L_2 are merged into a new list L'_1, the lists L_3 and L_4 are merged into a list L'_2, and so forth. Specifically, the list L'_i is composed of all the sums $x + y$ with $x \in L_{2i-1}$ and $y \in L_{2i}$ such that $x + y$ is zero on the first ℓ_1 bits. The integer ℓ_1 is a parameter of the algorithm that is to be selected for optimal performance. We say that the first round *eliminates* ℓ_1 bits.

The other rounds are akin to the first one. In the second round, lists $L''_1, \ldots L''_{2^{q-2}}$ are created from $L'_1, \ldots, L'_{2^{q-1}}$, eliminating a further sequence of ℓ_2 bits and thus causing the vectors in the lists $L''_1, \ldots, L''_{2^{q-2}}$ to be zero on the first $\ell_1 + \ell_2$ bits.

Iterating this procedure for q rounds, we get a single, final list containing vectors that are zero on the first $\sum_{i=1}^{q} \ell_i$ bits, each of which is a sum of the form $\sum_{i=1}^{k} x_i$ with $x_i \in L_i$. Since our goal is to find sums that are zero on all n bits, the final list will contain sums of the desired form provided that

$$\sum_{i=1}^{q} \ell_i \geq n. \tag{2.1}$$

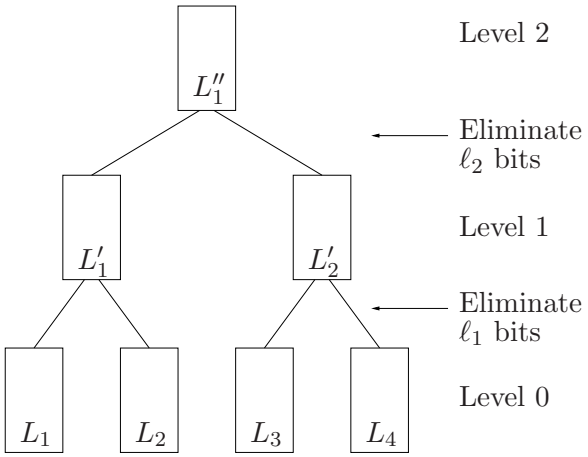

Figure 1: The k-tree algorithm for $k = 4$.

The algorithm can be visually represented as a complete binary tree of height q, with each node containing a list of vectors. Level j of the tree contains the lists after j rounds of the algorithm, with the leaves (at level 0) containing the input lists $L_1, \ldots, L_k$. Figure 1 gives a pictorial illustration of the case $k = 4$.

Note that, since the input lists are random, the lengths of the lists at all internal nodes within the tree are random variables. We will write M_j for the random variable representing the length of a list at level j. The distribution of M_j is determined by the values of m and $\ell_1, \ldots, \ell_j$.

Note that M_q is the total number of solutions found by the algorithm. We will also specify as a parameter the desired expected number of solutions found, which we write as 2^c. So our goal is to ensure that

$$\mathrm{E}[M_q] \geq 2^c. \tag{2.2}$$

Canonically one may think of the value $c = 0$, i.e., a single solution is sufficient in expectation. (This is what we did in the examples in the Introduction.) However, as we will see in section 4, the failure probability of the algorithm can be made significantly smaller by increasing the value of c slightly (e.g., by choosing $c = 1$). This entails a slight tightening of condition (1.1), which becomes

$$m \geq 2^{(n+c)/2^q}. \tag{2.3}$$

In the remainder of the paper we shall always assume that (2.3) holds. We will also assume that

$$m \leq 2^{(n+c)/(q+1)}, \tag{2.4}$$

since Wagner's algorithm applies for all larger m. Finally, for technical reasons we will also assume that $c < 2 \log m$; since typically c is a small constant (such as 0 or 1), while the list lengths are quite large, this represents no restriction in practice.

Note that the choice of the parameters ℓ_i critically impacts the behavior of the algorithm. Roughly speaking, increasing ℓ_i has the effect of reducing $\mathrm{E}[M_j]$ for every $j \geq i$, while decreasing it has the opposite effect. Since the running time is essentially proportional to the sum of the lengths of the lists at internal nodes in the tree, for optimal performance we seek a strategy for choosing

the ℓ_i such that $\mathrm{E}[M_j]$ is not too large for any $1 \le j \le q-1$; however, we also need to ensure that the constraints (2.1) and (2.2) both hold.

As a simple example, assuming m is a power of two, Wagner's original k-tree algorithm [13] chooses $\ell_j = \log m$ for $j = 1, \ldots, q-1$ and $\ell_q = 2 \log m$, leading to $\mathrm{E}[M_j] = m$ for $j = 1, \ldots, q-1$ and $\mathrm{E}[M_q] = 1$, i.e., all lists (except the last) have the same expected length as the initial lists. In this case, condition (2.1) translates to $(q+1) \log m \ge n$, which is exactly Wagner's condition (1.2) discussed earlier. If this condition holds, this choice for the ℓ_i works very well. One of the main goals of this paper is to find optimal choices for the ℓ_i when (1.2) does not hold.

Remark: The merging step at each node, as presented above, retains pairs of vectors whose sum on certain subsequences of bits is zero. As a result, the algorithm produces only solutions that satisfy these constraints. This is an arbitrary choice that was made only to simplify the presentation. In fact, the target values for the sums at each node could be chosen randomly, subject only to the requirement that the sum of all the target values at any level equals zero. This yields an algorithm that chooses a random solution, rather than one of the above special form.

3 Choosing the parameters

As we saw in section 2, our algorithm is specified by the parameters ℓ_i that determine the number of bits eliminated in each round. Our goal now is to find an optimal choice for the ℓ_i when m, q, n and c are given, i.e., to find a set of parameters that minimizes the running time while guaranteeing 2^c solutions in expectation.

In section 3.1, we will show how to reduce the problem of finding the optimal ℓ_i to an integer program. We will then give an explicit solution to this integer program in section 3.2.

3.1 The integer program

We start by giving a formula for the expected list length at each level of the tree. We write $b_0 := \log m$, and define 2^{b_j} as the expected length of the lists at level j of the tree. Then we have

$$b_j = 2b_{j-1} - \ell_j, \tag{3.1}$$

where ℓ_j is the number of bits eliminated at level j. To see this, let the random variable M_j be the number of vectors appearing in the list at some fixed node at level j, so that $2^{b_j} = \mathrm{E}[M_j]$. Writing M_{j-1}^l, M_{j-1}^r for the number of vectors in the lists at the left and right children of the node respectively, we have

$$2^{b_j} = \mathrm{E}[M_{j-1}^l M_{j-1}^r] 2^{-\ell_j} = \mathrm{E}[M_{j-1}^l]\mathrm{E}[M_{j-1}^r] 2^{-\ell_j} = 2^{2b_{j-1} - \ell_j},$$

which proves (3.1). Since the list at the root of the tree consists exactly of the solutions found by the algorithm, the expected number of solutions found is 2^{b_q}. The maximum expected list length that the algorithm has to process is

$$\max_{0 \le j \le q-1} 2^{b_j}. \tag{3.2}$$

(Note that b_q does not appear in this formula; this is because we do not need to explicitly compute the complete list of all matches, but can stop as soon as we have found a solution.) Since the expected running time of our algorithm is $\tilde{O}(2^{q+u})$, where 2^u is the maximum expected list length,

our goal will be to choose the ℓ_j so as to minimize the expression (3.2). For our formulation of the integer program it will be convenient to use both the ℓ_j and the b_j as variables. However, it can be seen from (3.1) that the ℓ_j determine the b_j and vice versa.

Suppose now that we specify that the expected number of solutions found by the algorithm should be at least 2^c. This leads to the following integer program:

$$\begin{aligned}
&\text{minimize } u \\
&\quad \text{s.t. } b_j \le u && j = 0, \ldots, q-1 \\
&\qquad \ell_j \ge 0,\ \ell_j \text{ integer} && j = 1, \ldots, q \\
&\qquad \sum_{j=1}^{q} \ell_j \ge n \\
&\qquad b_q \ge c.
\end{aligned}$$

Example: For $n = 100$, $m = 2^{16}$, and $q = 4$ (which are typical parameter values in, e.g., codeword-finding applications), and setting $c = 1$ (for an expected two solutions), the integer program dictates that we should choose $\ell_1 = 9$, $\ell_2 = 23$, $\ell_3 = 23$, $\ell_4 = 45$. This solution has an expected maximum list length of 2^{23}, resulting in roughly $2^{23+4} = 2^{27}$ expected vector operations, which is a very feasible computation.

For the same parameters, the naïve algorithm performs approximately 2^{50} operations, which is plainly unreasonable. Wagner's original algorithm is not intended to be used in this case, but if we use it anyway, eliminating 16 bits in each round to keep the list lengths constant, it will only succeed with probability at most 2^{-20}. Since a single run of Wagner's algorithm costs roughly $2^{16+4} = 2^{20}$ operations in this case, the expected running time (with repeated trials until a solution is found) would be about 2^{40}, again prohibitively large.

3.2 Solution of the integer program

We will now compute the optimum of the above integer program. We shall first consider the linear programming relaxation (without the integrality constraint), and then show that its solution can easily be rounded to a solution of the integer version.

We proceed by showing that the optimal solution of the LP has three "phases." In the first phase, for small i (i.e., low levels of the tree), the ℓ_i are all equal to zero, and b_i is doubled (so the length of the lists is squared) in each round. In the second phase, for larger values of i, b_i (and hence the length of the lists) remains fixed, meaning that a fixed number of bits ℓ_i is eliminated in each round. The third phase consists only of the final round, where the list is collapsed to the desired expected number of solutions, which is 2^c.

More precisely, we will prove the following.

Theorem 3.1 *For any set of parameters n, m, q, c satisfying conditions* (2.3), (2.4) *and $c < 2 \log m$, the linear program defined above is feasible and has an optimal solution of the following*

form:

$$\begin{array}{lll} b_i = 2^i b_0 & \ell_i = 0, & \text{for } 1 \le i < p; \\ b_p = u & \ell_p = 2^p b_0 - u; & \\ b_i = u & \ell_i = u, & \text{for } p < i < q; \\ b_q = c & \ell_q = 2u - c; & \end{array}$$

where p is the least integer such that

$$n \le (q - p + 1)2^p \log m - c,$$

and u is given by

$$u = \frac{n + c - 2^p \log m}{q - p}.$$

Note that the value $i = p$ marks the beginning of the second phase.

Proof: We will first show that the linear program is feasible. To this end, set $\ell_1 = \cdots = \ell_{q-1} = 0$, $\ell_q = n$. From (3.1), it follows then that $b_i = 2^i b_0$ for $i < q$, and $b_q = 2^q b_0 - n$. Set $u = \max_{i \le q-1} b_i = 2^{q-1} b_0$.

We now verify that this solution is feasible. Clearly, all the ℓ_i are non-negative and $\sum_{i=0}^{q} \ell_i \ge n$. The condition $b_q \ge c$ translates to $2^q b_0 \ge n + c$, which, recalling that $b_0 = \log m$, is equivalent to condition (2.3) and hence satisfied by assumption. Thus the solution is feasible.

Next we will show that any solution not of the form given in the statement of the theorem can be strictly improved. Since the LP is bounded (as can readily be checked from (3.1)) this will establish the theorem.

Consider first a feasible solution $\vec{\ell} = (\ell_1, \ldots, \ell_q)$ in which there is some index $j \in \{1, \ldots, q-1\}$ such that $\ell_j > 0$ and $b_j < u$. Then for suitably small $\varepsilon > 0$ the transformation

$$\ell_j \mapsto \ell_j - \varepsilon, \qquad \ell_{j+1} \mapsto \ell_{j+1} + 2\varepsilon, \qquad b_j \mapsto b_j + \varepsilon \tag{3.3}$$

yields another feasible solution with the same value of u. (This can easily be checked using the recursive definition (3.1).) Note that this transformation increases the sum of the ℓ_i by ε, so the constraint $\sum_{i=1}^{q} \ell_i \ge n$ becomes slack.

Similarly, in a feasible solution $\vec{\ell}$ in which $b_q > c$, the transformation

$$\ell_q \mapsto \ell_q + \varepsilon, \qquad b_q \mapsto b_q - \varepsilon \tag{3.4}$$

yields another feasible solution with the same value of u and makes the sum constraint slack.

Thus any solution that does not satisfy the conditions $\ell_j = 0$ or $b_j = u$ for all $j \in \{1, \ldots, q-1\}$, and $b_q = c$, can be transformed into a solution with the same value of the objective function u that does satisfy these conditions and where in addition $\sum_{i=1}^{q} \ell_i > n$.

We now show that such a solution can be transformed into one with a smaller value of u. Since $u = \max_j b_j$, it is enough to show that any maximal b_j can be reduced; the procedure can be repeated if necessary. We argue first that we cannot have $b_0 = \max_j b_j$. For if so, substituting the recursion (3.1) into $\sum_i \ell_i > n$, we get $\sum_{i=1}^{q} (2b_{i-1} - b_i) > n$, or equivalently $2b_0 + \sum_{i=1}^{q-1} b_i - b_q > n$,

and hence $(q+1)b_0 - c > n$; but this violates condition (2.4). So now let $1 \le j \le q-1$ be an index such that $b_j = u$, and consider the transformation

$$\ell_j \mapsto \ell_j + \varepsilon, \qquad \ell_{j+1} \mapsto \ell_{j+1} - 2\varepsilon, \qquad b_j \mapsto b_j - \varepsilon. \tag{3.5}$$

We claim that, for small enough $\varepsilon > 0$, this yields a feasible solution. To see this, we just need to check that $\ell_{j+1} > 0$. But if $\ell_{j+1} = 0$ then by (3.1) we would have $b_{j+1} = 2b_j = 2u$. If $j+1 < q$ this gives a contradiction because $b_{j+1} \le u$. And if $j+1 = q$ then $c = b_{j+1} = 2u \ge 2b_0 = 2\log m$, which violates our assumption that $c < 2\log m$.

The above argument shows that any optimal solution must satisfy $\ell_j = 0$ or $b_j = u$ for $1 \le j \le q-1$. We need to verify that the indices j for which $\ell_j = 0$ form an initial segment. To see this, simply observe that if $b_j = u$ and $\ell_{j+1} = 0$ then from (3.1) we have $b_{j+1} = 2u$ which contradicts the constraint $b_{j+1} \le u$.

Equation (3.1) can now be used to reconstruct the values of $\ell_p, \ldots, \ell_q, b_1, \ldots, b_{p-1}$ by direct computation.

It remains to determine p and u. From $b_{p-1} \le u$ and $0 \le \ell_p = 2^p b_0 - u$ we get $2^{p-1}b_0 \le u \le 2^p b_0$. Since, as we have seen above, the constraint $\sum_i \ell_i \ge n$ must be tight in an optimal solution, we also have $n = \sum_{i=1}^{q} \ell_i = (q-p)u + 2^p b_0 - c$. Substituting the above bounds on u into this equation for n gives

$$n \in [(q-p+2)2^{p-1}b_0 - c, (q-p+1)2^p b_0 - c].$$

Note that for distinct $p \in \{1, \ldots, q-1\}$ the interiors of these intervals are disjoint, and that the intervals cover $[(q+1)b_0 - c, 2^q b_0 - c]$, which is precisely the range of values of n for which the algorithm is applicable. So for given n, m, q and c satisfying (2.3) and (2.4), there is a unique choice of p (except at the endpoints, which belong to two intervals; either choice of p yields the same solution in this case). Once p is known, we can solve for u from $n = (q-p)u + 2^p b_0 - c$. □

The optimal solution to the linear program as given by Theorem 3.1 can result in fractional values for the ℓ_i; however, we need them to be integers. Fortunately, it turns out that the optimal solution of the corresponding integer program can be obtained by a simple rounding of the LP solution. This is the content of the following claim.

Claim 3.2 *Assume b_0 and c are integers. The optimal solution $\ell_1, \ldots, \ell_q$ of the integer program can be obtained by replacing u by $\lceil u \rceil$ in the LP solution of Theorem 3.1.*

Note that the value of p is not changed by this rounding operation.

Proof: Clearly, if this solution is feasible then it must be optimal since $\lceil u \rceil$ is the smallest integer exceeding u. Write $\tilde{u}, \tilde{\ell}_1, \ldots, \tilde{\ell}_q, \tilde{b}_1, \ldots, \tilde{b}_q$ for the putative integer solution obtained by applying the above rounding to the LP solution $u, \ell_1, \ldots, \ell_q, b_1, \ldots, b_q$. Then $\sum_{i=1}^{q} \ell_i = (q-p)u + 2^p b_0 - c$, which is increasing with u since $q - p \ge 0$; hence $\sum_{i=1}^{q} \tilde{\ell}_i \ge n$, as required. To see that $\tilde{\ell}_j \ge 0$, note that clearly $\tilde{\ell}_j \ge \ell_j$ for all j except $j = p$. But we also have $\tilde{\ell}_p \ge 0$ because $z - u \ge 0$ implies $z - \lceil u \rceil \ge 0$ for any integer z.

Finally, it can be checked by direct computation that the $\tilde{b}_i$ and $\tilde{\ell}_i$ still satisfy (3.1). □

Remark: The constraint $\sum_i \ell_i \ge n$ may not be tight in the given integer solution. This is not a problem however; for example, the length of the vectors can be increased to $\sum_i \ell_i$ by padding them with random bits at the end. Any solution to this new instance will also be a solution to the original one.

Note that the value of u given in Theorem 3.1 does not change if we replace n by $n+c$ and c by 0. So for simplicity we will assume $c = 0$ for the remainder of this section, i.e., we will assume that the algorithm aims for just one solution in expectation.

Corollary 3.3 *For all parameters n, m, q such that $2^{n/2^q} \leq m \leq 2^{n/(q+1)}$, the expected running time of the algorithm is $\tilde{O}(2^{q+u^*(n,m,q)})$, where $u^*(n,m,q)$ is the optimal value of u in the LP for parameters n, m and q as given in Theorem 3.1.*

Moreover $u^*(n,m,q)$ is a continuous, convex, piecewise affine and decreasing function of $\log m$.

Proof: Up to logarithmic factors, the running time is equal to the sum of all the list lengths that the algorithm processes. There are $2^{q+1} - 1 = O(2^q)$ lists, each of expected length at most $2^{\lceil u^*(n,m,q) \rceil} = O(2^{u^*})$ by Claim 3.2, resulting in an expected running time $\tilde{O}(2^{q+u^*(n,m,q)})$.

By Theorem 3.1, p is piecewise constant (as a function of n, m, q), and hence $u^*(n,m,q)$ is piecewise affine as a function of $\log m$. The other properties are easy to verify. □

To illustrate this Corollary, consider the plot in Figure 2 which compares the expected running time exponents of the naïve algorithm and the extended k-tree algorithm. We take the same example as in section 3.1, with $q = 4$, $n = 100$. The relevant range for m is then $2^{6.25} \leq m \leq 2^{20}$. At the very right, for $m = 2^{20}$, our algorithm is the same as the original k-tree algorithm, and uses roughly $2^{20+4} = 2^{24}$ vector operations. As noted before, the original k-tree algorithm does not work for $m < 2^{20}$ in this setting.

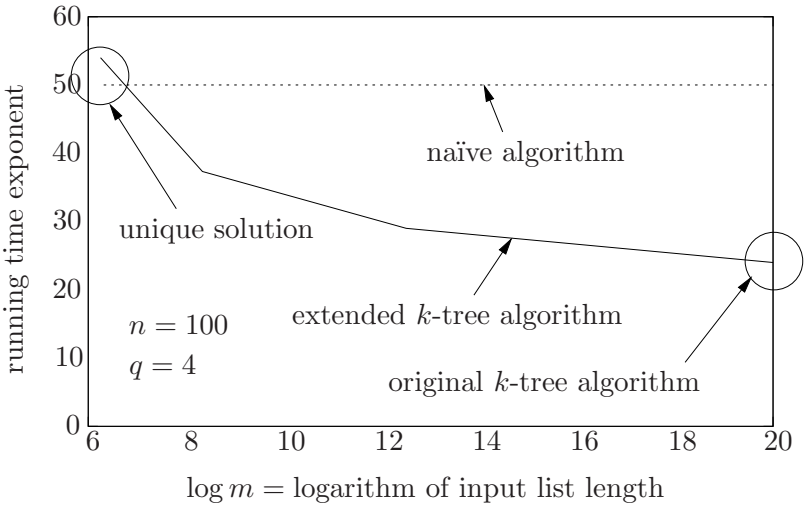

Figure 2: Comparison of the extended k-tree algorithm with the naïve algorithm

At the left border, for $m = 2^{6.25}$, our algorithm is nothing but a (somewhat complicated) variant of the naïve algorithm, and the estimated expected running time is 2^{50+4}. For $m < 2^{6.25}$ the probability that any solution exists at all decays rapidly.

Note that our algorithm contains both the naïve algorithm and the original k-tree algorithm as special cases, but does substantially better than the naïve algorithm for a wide range of values of m where the original k-tree algorithm no longer works.

Remark: In the graph we are seemingly overtaken by the naïve algorithm shortly before $m = 2^{6.25}$. This is just an artifact of our analysis. While our running time estimate is the best that can be achieved purely in terms of the maximum list length, it should be noted that the additional factor 2^q of Corollary 3.3 is crude for small m, because (as can be seen from the LP solution) in that case only very few lists will have maximal length. Since for $m = 2^{6.25}$ our algorithm is essentially the same as the naïve algorithm, it must in fact have the same complexity.

4 Analysis of the failure probability

Up to this point, we have implicitly assumed that it is enough to design the algorithm so that the expected number of solutions found is at least one, and that a single run of the algorithm would then yield a solution with good probability. The goal of this section is to justify this assumption, i.e., to show that the number of solutions per run is concentrated around its expectation in most interesting cases, and that therefore the algorithm does indeed produce an output with reasonable probability. We note that our analysis applies also to the original k-tree algorithm of Wagner [13], whose failure probability had apparently not previously been bounded.

4.1 Preliminaries

Let N be the number of solutions found by the algorithm. Thus the algorithm succeeds when $N > 0$ and fails if $N = 0$. Write the input lists as $L_1 = (x_1^1, \ldots, x_m^1), \ldots, L_{2^q} = (x_1^{2^q}, \ldots, x_m^{2^q})$. Let $\mathcal{S} = \{1, \ldots, m\}^{2^q}$, and let $a = (a_1, \ldots, a_{2^q}) \in \mathcal{S}$. Then the vector of indices a corresponds to a solution found by the algorithm if $x_{a_1}^1 + \cdots + x_{a_{2^q}}^{2^q} = 0$, and if in addition the $x_{a_i}^i$ satisfy the constraints imposed by the internal nodes of the tree. For example, we must have $x_{a_1}^1 + x_{a_2}^2 = 0$ on the first ℓ_1 bits, and so on; there are $2^q - 1$ such constraints to be satisfied.

If we write I_a as the indicator random variable of the event that a is a solution, then $N = \sum_{a \in \mathcal{S}} I_a$. Writing $\mu := \mathrm{E}[I_a]$, we get by Chebyshev's inequality that

$$\begin{aligned} \Pr(N = 0) &\leq \frac{\mathrm{Var}(N)}{\mathrm{E}[N]^2} \leq \frac{|\mathcal{S}|\mu + \sum_{a,b \in \mathcal{S}, a \neq b} \mathrm{Cov}(I_a, I_b)}{|\mathcal{S}|^2 \mu^2} \\ &\leq \mathrm{E}[N]^{-1} + \frac{\mathrm{E}_{ab}[\mathrm{Cov}(I_a, I_b) \mid a \neq b]}{\mu^2}, \end{aligned} \tag{4.1}$$

where $\mathrm{E}_{ab}[\cdot]$ denotes expectation over a and b chosen independently and uniformly at random from $\mathcal{S}$.

If I_a and I_b were independent whenever $a \neq b$, then this probability would of course be bounded by $\mathrm{E}[N]^{-1}$. However, I_a and I_b can be highly correlated if a and b have many components in common. We therefore have to bound the covariance terms in (4.1).

4.2 Incidence trees

Fix $a, b \in \mathcal{S}$. The *incidence tree* for a and b is a complete binary tree of height q with the nodes being either squares ($\square$) or triangles ($\triangle$) according to the following rules. The ith leaf (from the left) is associated with the ith components of a and b. A node is a triangle if and only if all the components of a and b in the leaves below it are equal. Note that the shape of any internal node

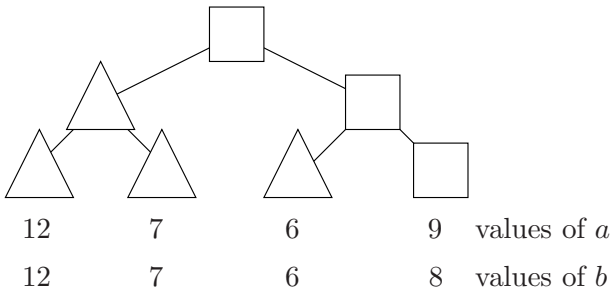

Figure 3: The incidence tree of $a = (12, 7, 6, 9)$ and $b = (12, 7, 6, 8)$.

can be deduced from the shape of its children: it is a triangle if and only if both its children are triangles. For an example of an incidence tree with $q = 2$, see Figure 3.

If the incidence tree of a and b is known, then the value of $\mathrm{Cov}(I_a, I_b)$ can be computed easily. First note that we can factor I_a as follows:

$$I_a = \prod_y J_a(y),$$

where y runs over all the internal nodes of the tree, and $J_a(y)$ is the indicator random variable of the event that the constraint implied by node y is satisfied by a. (Note that the constraint at y involves only ℓ_j bits, where j is the level of y; this constraint can be satisfied even if the constraints at some of the descendants of y are not.)

For fixed internal nodes y and z, the random variables $J_a(y)$ and $J_b(z)$ are equal if $y = z$ and y is a triangle in the (a, b)-incidence tree. Otherwise $J_a(y)$ and $J_b(z)$ are independent. In particular, the $J_a(y)$ (where y runs over the internal nodes) are mutually independent. So for any internal node y, we have

$$\mathrm{E}[J_a(y)J_b(y)] = \begin{cases} \mathrm{E}[J_a(y)] = 2^{-\ell_{\mathrm{level}(y)}} & \text{if } y \text{ is a triangle;} \\ \mathrm{E}[J_a(y)]\mathrm{E}[J_b(y)] = 2^{-2\ell_{\mathrm{level}(y)}} & \text{if } y \text{ is a square.} \end{cases}$$

Here, $\mathrm{level}(y)$ denotes the level of the node y in question. Writing $F_{ab} := \mathrm{E}\big[\prod_{y \text{ square}} J_b(y)\big] = 2^{-\sum_{y \text{ square}} \ell_{\mathrm{level}(y)}}$, we have

$$\mathrm{E}[I_a I_b] = \mathrm{E}[I_a \prod_{y \text{ square}} J_b(y)] = \mathrm{E}[I_a]\mathrm{E}[\prod_{y \text{ square}} J_b(y)] = \mu F_{ab}.$$

We can then compute the covariance $\mathrm{Cov}(I_a, I_b)$ as follows:

$$\mathrm{Cov}(I_a, I_b) = \mathrm{E}[I_a I_b] - \mu^2 = \mu\left(F_{ab} - \mu\right). \tag{4.2}$$

4.3 Computing the expected covariance

We now derive an exact recursive formula for $\mathrm{E}_{ab}[\mathrm{Cov}(I_a, I_b) \,|\, a \neq b]$. To this end, we study the behavior of the random variable F_{ab} when a and b are random. For a node y at level $j \geq 1$, define

$$S_j := \prod_{\substack{z \text{ a square} \\ \text{descendant of } y}} 2^{-\ell_{\mathrm{level}(z)}},$$

where z runs over all square internal nodes in the subtree whose root is y. With this notation, note that $F_{ab} = S_q$. For $j \geq 1$, we write $\mathrm{E}^{\square}_{ab}[S_j]$ for the expectation (over a, b) of S_j, conditional on the node with respect to which S_j is defined being a square. Then, setting $S_0 = 1$, we have

$$\mathrm{E}^{\square}_{ab}[S_j] = 2^{-\ell_j}\mathrm{E}^{\square}_{ab}[S_{j-1}]\big(\mathrm{E}^{\square}_{ab}[S_{j-1}](1-\alpha_j)+\alpha_j\big), \tag{4.3}$$

where

$$\alpha_j := \Pr(\text{a level-}j\text{ node } y \text{ has a } \triangle \text{ child} \mid y \text{ is a } \square) = \frac{2 \cdot m^{-2^{j-1}}}{1 + m^{-2^{j-1}}}. \tag{4.4}$$

Now, equation (4.3) can be used recursively to compute $\mathrm{E}^{\square}_{ab}[S_q]$. From this we can compute $\mathrm{E}_{ab}[\mathrm{Cov}(I_a, I_b) \mid a \neq b] = \mu(\mathrm{E}^{\square}_{ab}[S_q] - \mu)$, which follows from (4.2) and the facts that $F_{ab} = S_q$ and $\mathrm{E}^{\square}_{ab}[S_q] = \mathrm{E}_{ab}[S_q \mid a \neq b]$.

Putting everything together, we get that the error bound (4.1) of section 4.1 can be written as

$$\Pr(N = 0) \leq \mathrm{E}[N]^{-1} + \mu^{-1}\mathrm{E}^{\square}_{ab}[S_q] - 1, \tag{4.5}$$

where $\mathrm{E}^{\square}_{ab}[S_q]$ is the solution to the recurrence (4.3). Note that the quantity $\mu^{-1}\mathrm{E}^{\square}_{ab}[S_q] - 1$ captures the contribution due to dependencies between the indicator random variables I_a.

Remark: Inequality (4.5) provides a method for numerically bounding the failure probability of the extended k-tree algorithm for any choice of the parameter values $(\ell_1, \ldots, \ell_q)$, provided only that $\sum_i \ell_i \geq n$. Of course, the values N and $\mathrm{E}^{\square}_{ab}[S_q]$ will depend on the choice of the ℓ_i.

Example: Consider our running example with $q = 4$, $m = 2^{16}$, $n = 100$, $\ell_1 = 9$, $\ell_2 = 23$, $\ell_3 = 23$, $\ell_4 = 45$. With these settings we get two solutions per run in expectation ($c = 1$); hence we would like the failure probability to be close to $1/2$ (as would be the case if the random variables I_a were independent). Using the above recursive method to compute $\mathrm{E}_{ab}[\mathrm{Cov}(I_a, I_b) \mid a \neq b]$, we get a bound on the failure probability of 0.5000017. Thus the effect of dependencies is very small, as desired.

4.4 Bounding the failure probability

In this section we consider the optimal choice of the parameters ℓ_i, as described in section 3. For this choice of the ℓ_i, we provide the following *analytic* upper bound on the failure probability that is useful in many applications.

Theorem 4.1 *If $\ell_1, \ldots, \ell_q$ are chosen optimally as in section 3.2, then the algorithm will fail to find a solution with probability at most*

$$2^{-c} + \exp(qk/m) - 1,$$

where $2^c = \mathrm{E}[N]$ is the expected number of solutions.

Proof: In light of inequality (4.5), it is enough to show that the quantity $\mu^{-1}\mathrm{E}^{\square}_{ab}[S_q]$ is bounded above by $\exp(qk/m)$.

Define $s_j := \mathrm{E}^{\square}_{ab}[S_j]$. From (4.3), we have $s_j = 2^{-\ell_j} s_{j-1}^2\big(1 - \alpha_j + \frac{\alpha_j}{s_{j-1}}\big)$ with $s_0 = 1$. Let $\mu_j := 2^{-\ell_j}\mu_{j-1}^2$ with $\mu_0 = 1$. Note that μ_j is the probability that a fixed partial sum appears in a node at level j; in particular $\mu = \mu_q$. By inspecting the recursions, we also get $\mu = \mu_q = 2^{-\sum_{i=1}^{q} \ell_i 2^{q-i}}$, and $\mu_j \leq s_j$.

We also define $t_j := m^{-2^j}$ and remark that t_j^{-1} is equal to the number of partial sums that are candidates to appear in a node at level j; hence the expected list length at level j is equal to $t_j^{-1}\mu_j$. Note also from the definition (4.4) of α_j that $\alpha_j \le 2t_{j-1}$.

Unwinding the formula for s_q, we get

$$\begin{aligned} s_q &= 2^{-\sum_{i=1}^{q} \ell_i 2^{q-i}} \prod_{j=0}^{q-1} \left(1 - \alpha_{j+1} + \frac{\alpha_{j+1}}{s_j}\right)^{2^{q-j-1}} \\ &\le \mu \cdot \prod_{j=0}^{q-1} \left(1 + \frac{2t_j}{\mu_j}\right)^{2^{q-j-1}} \\ &= \mu \cdot \exp\left\{\sum_{j=0}^{q-1} 2^{q-j-1} \ln\left(1 + \frac{2t_j}{\mu_j}\right)\right\}. \end{aligned}$$

Now we bound the sum in the exponent. Noting that $\ln(1+x) \le x$, and that $t_j\mu_j^{-1} = 2^{-b_j}$, the summand in j can be bounded above by 2^{q-j-b_j}. Since b_j is increasing in j (for the optimal choice of ℓ_j given by Theorem 3.1 and Claim 3.2), the largest summand is the one for which $j = 0$, so estimating the sum by taking q times the largest summand we get

$$\mu^{-1}\mathrm{E}_{ab}^{\square}[S_q] = \mu^{-1}s_q \le \exp(q2^q/2^{b_0}) = \exp(qk/m),$$

which completes the proof. □

To interpret Theorem 4.1, note that the additional error probability due to dependencies is approximately qk/m, assuming this quantity is fairly small. Hence if $c = 1$, we will get an overall error probability very close to $1/2$ provided qk is much smaller than m. This condition is satisfied in particular for the various applications mentioned in the introduction. E.g., for our running example above with $n = 100$, $m = 2^{16}$, $q = 4$, $c = 1$, Theorem 4.1 bounds the failure probability by 0.50097, which is very close to $1/2$ (the best we can hope for from a second moment analysis), and only slightly larger than the value 0.5000017 computed at the end of the previous subsection.

Note also that in typical applications (when qk/m is small) the principal variable controlling the error probability is c; if c is not too large, increasing c causes the failure probability to decrease significantly.

Remark: The failure probability given by Theorem 4.1 differs at first sight qualitatively from those in [10, 12] for the special case of Wagner's algorithm in that it does not decay to zero with the list length. However, our bound applies to the optimal algorithm for given n, m, q and c, while in [10, 12] the list length m is required to be larger by a factor of α in order to achieve the bound on the failure probability (which decays exponentially with α). Obviously, since we achieve a constant failure probability for the given list length m, increasing the list length by a factor of α would allow us to run α independent trials of our algorithm, which also causes the failure probability to decrease exponentially with α.

To obtain a nontrivial failure probability with Theorem 4.1, it is necessary to aim for more than one solution in expectation ($c > 0$). We will now show that even if we aim for one solution, or indeed somewhat less, the failure probability can still be usefully bounded in many cases.

Corollary 4.2 *If $\ell_1, \ldots, \ell_q$ are chosen optimally as in section 3.2 with $-2\log m + 1 < c \leq 0$, then the algorithm will fail to find a solution with probability at most*

$$1 - 2^{c-1}\left(\frac{3}{2} - \exp(qk/m)\right).$$

Proof: Consider an instance I with parameters n, m, q. We wish to bound the failure probability of the algorithm when solving I with $c \leq 0$. From the instance I, build a new instance I' with parameters $n + c - 1, m, q$ by removing the last $1 - c$ bits from each vector in every list. (By our assumed lower bound on c, these bits are all eliminated in the last round.) The choice of the ℓ_i made by the algorithm for instance I' with $c = 1$ is the same as that for instance I with the given value of c. If the algorithm is applied to I', by Theorem 4.1 it finds a solution with probability at least

$$\frac{3}{2} - \exp(qk/m).$$

If we now consider the corresponding sum in I, we get a vector that has a zero in all positions except possibly for the last $1 - c$ of them. The values in these positions will all be zero with probability 2^{c-1}; in that case, this solution will also be found by the algorithm when it runs on instance I with the given value of c. Therefore the algorithm fails on I with probability at most $1 - 2^{c-1}\left(\frac{3}{2} - \exp(qk/m)\right)$, as claimed. □

5 Larger alphabets

The k-sum problem has a natural generalization to non-binary alphabets, where the goal is now to find a non-trivial linear combination of vectors summing to zero. Formally, we state the problem as follows.

Let r be a prime power. We are given $k = 2^q$ lists $L_1, \ldots, L_k$, each of length m, containing (independent, uniformly sampled) random vectors from $\mathbb{F}_r^n$. We wish to find $x_1 \in L_1, \ldots, x_k \in L_k$ and $\lambda_1, \ldots, \lambda_k \in \mathbb{F}_r^*$ such that

$$\lambda_1 x_1 + \cdots + \lambda_k x_k = 0.$$

The requirement that *all* the λ_i have to be nonzero is arguably somewhat artificial, since for typical applications any solution with not all the λ_i zero is satisfactory. However, this formulation has the advantage that it does indeed generalize the binary case. Moreover in both the binary and the non-binary cases, a solution allowing some of the λ_i to be zero can be sought by adding the zero-vector to every list.

The k-sum problem over non-binary alphabets can be used to solve general finite-field versions of the various problems mentioned in the introduction. For example, it can be used to find low-weight codewords in codes with comparatively few redundant symbols defined over a moderate size alphabet, such as the erasure-correcting codes analyzed in [4]. Typical parameter values that might arise in such a setting are the following, which we shall use as our running example throughout this section:

Example: We work in $\mathbb{F}_{64}$, and consider vectors of $n = 18$ symbols; such a vector is then representable with 108 bits. We wish to find a sum of $k = 8$ of them from 8 lists, such that their sum equals zero.

The obvious generalization of the k-tree algorithm consists of just putting every possible nonzero scalar multiple of every vector into the lists. This has two serious drawbacks, however. First, it inflates the list lengths (and hence both the space and time requirements) by a factor of $r-1$; and second, it destroys the independence of the vectors in a list, making analysis much harder. In the following, we develop an alternative version of the extended k-tree algorithm tailored to the non-binary case. Our version suffers at most only a factor 4 increase in space and a factor $\sqrt{r}$ increase in time compared to the binary version with the same parameter values. Moreover, we are still able to carry out a full analysis of our algorithm, including the failure probability, similar to that for the binary case given earlier. The algorithm works for a range of m analogous to that for the binary case (see (5.3) below).

5.1 The merging procedure

The starting point for our modified algorithm is a more involved merging procedure, which ensures that only a single scalar multiple of each relevant linear combination is retained in the lists at each stage of the algorithm. This is key to avoiding a blow-up of a factor $r-1$ in the list lengths. In this section we describe this merging procedure.

We are given two lists L_1, L_2 and an integer ℓ designating the number of positions to eliminate. We wish to construct a minimal merged list L, i.e., a list of vectors having the following properties:

- *Validity.* Every vector in L is of the form $\lambda_1 x_1 + \lambda_2 x_2$ with $\lambda_i \in \mathbb{F}_r^*$ and $x_i \in L_i$ and it has a prefix of ℓ zeros.
- *Completeness.* If $\lambda_1, \lambda_2 \in \mathbb{F}_r^*$, $x_1 \in L_1$ and $x_2 \in L_2$ are such that $y = \lambda_1 x_1 + \lambda_2 x_2$ has ℓ leading zeros, then there exists a $\mu \in \mathbb{F}_r^*$ such that μy appears in L.
- *Minimality.* If $y = \lambda_1 x_1 + \lambda_2 x_2$ appears in L, then no other linear combination $\mu y = (\mu\lambda_1)x_1 + (\mu\lambda_2)x_2$ with $\mu \neq 1$ appears in L. (A scalar multiple of y may of course appear in L by coincidence, i.e., if it can be written as a linear combination that is not just a scaling of $\lambda_1 x_1 + \lambda_2 x_2$.)

Such a list can be efficiently computed as follows. By multiplying the vectors in the lists with suitable constants, we can assume they are all of the *normalized* form

$$(0, \ldots, 0, 1, *, \ldots, *); \tag{5.1}$$

that is, the leftmost nonzero position, if it exists, is a 1.

Let $x_1 \in L_1$ and $x_2 \in L_2$ be two normalized vectors in their respective lists, and let ℓ be the number of positions that we wish to eliminate. We make the following observations:

1. If either x_1 or x_2 has less than ℓ leading zeros, then the only possible valid linear combinations $\lambda_1 x_1 + \lambda_2 x_2$ with ℓ leading zeros satisfy $\lambda_1 + \lambda_2 = 0$. In particular the first ℓ symbols of x_1 and x_2 are equal in this case.
2. Otherwise, both x_1 and x_2 have at least ℓ leading zeros. Then there are exactly $r-1$ distinct valid sums of x_1 and x_2 which have at least ℓ leading zeros, namely all sums of the form $x_1 + \lambda x_2$, where λ runs over $\mathbb{F}_r^*$.

This shows that by normalizing the vectors in both lists, and by sorting one list, a merging algorithm can be implemented in $\tilde{O}(|L_1| + |L_2|)$ vector operations.

The expected number of vectors in the merged list is given by the following Proposition.

Proposition 5.1 *If $m_1 = |L_1|$ and $m_2 = |L_2|$, the (unnormalized) vectors in L_i are uniformly distributed, and the list L_1 is independent of L_2, then the merged list L contains*

$$\mathrm{E}[|L|] = m_1 m_2 (r-1) r^{-\ell} \tag{5.2}$$

elements in expectation.

Note that formula (5.2) generalizes (3.1).

Proof: W. l. o. g., we can assume that all the vectors in L_1 and L_2 are normalized as in (5.1). Write p_i for the expected fraction of vectors in L_1 (or L_2) with exactly i leading zeros, and $p_{\geq \ell}$ for the expected fraction of vectors with at least ℓ leading zeros. Then

$$p_i = r^{-i}(1 - r^{-1}), \qquad p_{\geq \ell} = r^{-\ell}.$$

A vector in L_1 having $i < \ell$ leading zeros is merged with probability $r^{-(\ell-(i+1))}$ with any given vector in L_2 with i leading zeros. It cannot be merged with any vector having a different number of leading zeros. Hence the expected number of vectors in L obtained from merging such vectors is

$$m_1 m_2 p_i^2 r^{-(\ell-(i+1))}.$$

In addition, we get all the vectors obtained from combining the vectors which already have at least ℓ leading zeros. In expectation, this will result in

$$m_1 m_2 (r-1) p_{\geq \ell}^2$$

vectors. So, summing up, we get

$$\mathrm{E}[|L|] = m_1 m_2 \left[(r-1) p_{\geq \ell}^2 + \sum_{i=0}^{\ell-1} p_i^2 r^{i-(\ell-1)} \right] = m_1 m_2 (r-1) r^{-\ell},$$

as desired. □

5.2 The range of m

The following inequalities give the interesting range of m analogous to the range given in the binary case by equations (2.3) and (2.4):

$$\frac{r^{(n+c)/2^q}}{(r-1)^{1-1/2^q}} \leq m \leq \frac{r^{(n+c)/(q+1)}}{(r-1)^{1-1/(q+1)}}. \tag{5.3}$$

The lower bound for m corresponds to the requirement that a solution exists with reasonable probability and follows from the Markov inequality, while the upper bound corresponds to the list length at which a Wagner-style algorithm can be implemented and follows from equation (5.2) and the condition that the list size remains constant over the rounds.

Taking the example from the beginning of section 5 with $n = 18$, $r = 64$, and $k = 8$, we get the approximate range $309 \leq m \leq 6 \cdot 10^6$ for m in this case.

5.3 The linear program

Write ℓ_i for the number of symbols that we wish to eliminate at level i. Following our notation from section 3.1, for $i = 0, \ldots, q$ we define $b_j := \log_r \mathrm{E}[\text{list length at level } j]$, and using (5.2) we see that

$$\begin{aligned} b_0 &= \log_r m; \\ b_j &= 2b_{j-1} + \Delta_r - \ell_j \qquad \text{if } j \geq 1, \end{aligned}$$

where $\Delta_r := \log_r(r-1)$.

Assuming we wish to find r^c nonequivalent solutions in expectation, we get the following linear program to minimize the maximal list length:

$$\begin{aligned} &\text{minimize } u \\ &\quad \text{s.t. } b_j \leq u && j = 0, \ldots, q-1 \\ &\qquad \ell_j \geq 0 && j = 1, \ldots, q \\ &\qquad \sum_{j=1}^{q} \ell_j \geq n \\ &\qquad b_q \geq c. \end{aligned}$$

We will for now ignore the problem that the ℓ_j should be integer; this problem will be separately addressed in section 5.4.

Theorem 5.2 *Suppose that m is in the range given by (5.3), and that $c < 2\log_r m + \Delta_r$. Then the optimal solution of the above linear program is as follows:*

$$\begin{array}{lll} b_i = 2^i b_0 + (2^i - 1)\Delta_r & \ell_i = 0, & \text{for } 1 \leq i < p; \\ b_p = u & \ell_p = 2^p b_0 + (2^p - 1)\Delta_r - u; & \\ b_i = u & \ell_i = u + \Delta_r, & \text{for } p < i < q; \\ b_q = c & \ell_q = 2u + \Delta_r - c. & \end{array}$$

Here, p is the least integer such that

$$n \leq (q - p + 1)2^p(\log_r m + \Delta_r) - \Delta_r - c,$$

and the value of u is

$$u = \frac{n + c - 2^p \log_r m - (2^p - 1)\Delta_r}{q - p} - \Delta_r.$$

Proof: The proof is analogous to the proof of Theorem 3.1. We therefore give an abridged version, just filling in some new computational details.

Direct verification shows that if m satisfies the lower bound given by (5.3), then the solution $\ell_1 = \cdots = \ell_{q-1} = 0$, $\ell_q = n$ is feasible.

Now we need to show that any solution not having the shape given in the theorem can be improved. This follows exactly as in the proof of Theorem 3.1, by showing that any such solution can be always be improved using transformations (3.3), (3.4) and (3.5) the same way as in the proof of Theorem 3.1.

We now compute p. For $i = 1, \ldots, q$, write

$$t_i = 2^i b_0 + (2^i - 1)\Delta_r.$$

Note that with this notation we have $b_i = t_i$ for $i < p$, and $\ell_p = t_p - u$. Plugging this into the constraints $b_{p-1} \le u$ and $0 \le \ell_p$ gives

$$t_{p-1} \le u \le t_p.$$

Since the sum constraint is tight in the optimal solution, we can deduce from these bounds on u that

$$n \in [t_p + (q-p)(t_{p-1} + \Delta_r) - c, t_p + (q-p)(t_p + \Delta_r) - c].$$

The fact that $t_{p+1} = 2t_p + \Delta_r$ implies that, as p varies, these intervals for n have disjoint interiors and meet at the endpoints. Moreover, the range of n increases with p monotonically. This gives us the condition on computing p in the theorem.

Finally, the value of n follows immediately from the tight sum constraint $\sum_{j=1}^{q} \ell_j = n$. □

Example. With $c = 0$, our running example with $n = 18$, $r = 64$ and $q = 3$, an input list length of $m = 10^6$ leads to a solution with a maximal list length of $r^u \approx 2^{25}$. For these parameters, the extended k-tree algorithm thus runs much faster than the naïve algorithm, which takes about $2^{56-6} = 2^{50}$ time[¶].

Remark. We should observe that the following algorithm based on solving linear equations may also be competitive here. Build a $k \times n$ matrix A with coefficients in $\mathbb{F}_r$ constructed by setting the ith row to be a randomly selected vector from L_i, $1 \le i \le k$. Now solve the linear system $\lambda A = 0$, where $\lambda \in \mathbb{F}_r^k$. If there is a nontrivial solution with $\lambda_i \neq 0$ for every $1 \le i \le k$, then a solution to the given k-list problem can easily be deduced.

A comparison is in order. Since in some applications a solution with not all the $\lambda_i \neq 0$ is satisfactory, we will ignore this requirement in our comparison. Then a randomly constructed matrix A has probability approximately r^{k-n} of yielding a solution. Hence, approximately r^{n-k} trials are needed for this algorithm to produce a solution, leading to a running time of $\tilde{O}(r^{n-k})$.

For the example numbers above ($n = 18$, $r = 64$, $q = 3$), we thus estimate the running time of this algorithm to be 2^{60} trials, which is much worse than the extended k-tree algorithm; indeed, it is even worse than the naïve algorithm! However, in some cases (when r is large) the linear equations algorithm can outperform the k-tree algorithm. To illustrate when this may occur, note from the optimal solution above that $n + c \le (q+1)u + q$. If we set $c = 0$, it follows that the linear equations algorithm is faster (i.e., $r^{n-k} < r^u$) if

$$\frac{2^q}{q} - 1 \ge u. \tag{5.4}$$

Note that (5.4) could potentially also hold in the binary case. However, this is unlikely to occur in practical settings because the maximum expected list length there is 2^u (rather than r^u), so u is typically much larger in the binary case.

[¶]The -6 in the exponent comes from the fact that we only seek a solution up to linear scaling.

5.4 Implementation

We now face an amplified version of the rounding problem that we already had in the binary case: the linear program finds real values for ℓ_i, but the list merging step *a priori* needs them to be integer. We could proceed as in the binary case, and use the same rounding, but this results in a potential increase of the maximal list length by a factor r in the worst case.

In this section, we present a solution to this problem which results in a list length increase by a factor 4 at most, and is thus preferable in most cases. We proceed in several steps. First, in section 5.4.1 we present a way to *partially* eliminate symbols. The effect of this is that we no longer need the ℓ_i to be integer, but they can also take certain non-integer values. Even with partial elimination, the ℓ_i cannot be chosen freely: they still have to be chosen from some discrete superset of $\mathbb{N}$. Therefore, some rounding is necessary all the same. In section 5.4.2 we solve the problem of appropriately rounding the ℓ_i. Finally, in section 5.4.3, we combine these techniques and show that this rounding scheme results in a list length increase of at most factor of 4 (independently of r), when compared to the unrounded version. The running time, however, increases by a factor $O(\sqrt{r})$, as we will also see.

5.4.1 Basic partial elimination

Assume we wish to eliminate ℓ_1 symbols, where ℓ_1 is not necessarily an integer. First we eliminate $\lfloor \ell_1 \rfloor$ symbols as usual, giving us a list of (normalized) vectors of the form

$$(0, \ldots, 0, 1, *, \ldots, *, x).$$

Now, eliminating the additional γ_1-fraction of a symbol, where $\gamma_1 := \ell_1 - \lfloor \ell_1 \rfloor$, corresponds to putting another constraint on the value that x in the above vector is allowed to take. We do this as follows. Fix an arbitrary *partial elimination set* $V \subset \mathbb{F}_r$ of size $|V| = r^{1-\gamma_1}$, and throw away all the vectors for which $x \notin V$. (This assumes that $r^{1-\gamma_1} = r^{1-\ell_1+\lfloor \ell_1 \rfloor}$ is an integer, a requirement that we will arrange for in section 5.4.3.)

Assuming that the elements of the last position are distributed uniformly over $\mathbb{F}_r$ (see section 5.4.4), the new list will have an expected length predicted by (5.2). In the next elimination round, the collision probability for the position corresponding to x will be $|V|^{-1}$, instead of r^{-1} as is the case for the other nonzero positions.

Now suppose that in the next partial elimination round, we wish to eliminate ℓ_2 symbols (i.e., in total, we want $\ell_1 + \ell_2$ symbols to be zero after this round). Let t be the largest integer such that

$$t + (1 - \gamma_1) \leq \ell_2.$$

We first eliminate t positions from the left plus the last position (which contains a $(1-\gamma_1)$-fraction of a symbol), and then eliminate a γ_2-fraction of the symbol in the second-last position, where

$$\gamma_2 := \ell_2 - (t + 1 - \gamma_1).$$

This procedure can be extended in the obvious way to any number of rounds. Note that we assumed that $\ell_2 \geq 1$, and similarly for all the subsequent values of ℓ_i. Inspection of the optimal solution shows that this is the case if $u \geq 1$; and if $u < 1$, the linear equations algorithm mentioned earlier is preferable anyway (see (5.4)).

Remark. An induction proof can be used to show that Proposition 5.1 still holds when a fractional number of symbols are eliminated according to the above scheme.

5.4.2 Finer-grained rounding

Recall that in the binary case our algorithm required the values of ℓ_i to be integers. As we will see, in the $\mathbb{F}_r$ case we can relax this restriction, but we still need the ℓ_i to belong to some discrete set containing the integers.

We will now describe a rounding scheme that can be used to round the values $\sum_{i=1}^{j} \ell_i$, $1 \le j \le q$, to any discrete subset $\mathbb{N} \subset S \subset \mathbb{R}$. This scheme has the following properties:

- The increase in the maximal list length can be bounded by a factor that depends only on the maximum distance between two neighboring points in S, i.e., on the quantity $\mathrm{gap}(S)$ defined as follows:
$$\mathrm{gap}(S) := \sup\{b - a \mid a, b \in \mathbb{R}, b \ge a \text{ such that } (a,b) \cap S = \emptyset\}. \tag{5.5}$$
- There is no restriction on other parameters such as the input list length b_0.

We first need a useful characterization of feasible solutions.

Claim 5.3 *Let $\ell_1, \ldots, \ell_q$ be a feasible solution to an (n, m, q, c)-instance, and let $b_0, \ldots, b_q$ be the corresponding b-values. Let $\hat{c} \le c$. Then any solution $\hat{\ell}_1, \ldots, \hat{\ell}_q$ with b-values $\hat{b}_i$ satisfying*

$$\hat{b}_i \ge b_i \text{ for } 1 \le i \le q-1, \text{ and } \hat{c} \le \hat{b}_q \le b_q$$

and such that $\hat{\ell}_i \ge 0$ is a feasible solution to an $(n, m, q, \hat{c})$-instance.

Proof: We just need to verify that the sum constraint is satisfied by the $\hat{\ell}_i$. Since $\hat{\ell}_i = 2\hat{b}_{i-1} - \hat{b}_i + \Delta_r$, we have

$$\begin{aligned} n &\le \sum_{i=1}^{q} \ell_i \\ &= 2\sum_{i=0}^{q-1} b_i - \sum_{i=1}^{q} b_i + q\Delta_r \\ &= 2b_0 + \sum_{i=1}^{q-1} b_i - b_q + q\Delta_r \\ &\le 2b_0 + \sum_{i=1}^{q-1} \hat{b}_i - \hat{b}_q + q\Delta_r \\ &= \sum_{i=1}^{q} \hat{\ell}_i, \end{aligned}$$

finishing the proof. □

Claim 5.3 can be used to obtain the following rounding scheme. Let S be some discrete set with $\mathrm{gap}(S) < \infty$. We are given an $(n, m, q, \hat{c})$-instance, and we seek a feasible solution $\hat{\ell}_1, \ldots, \hat{\ell}_q$ with the property that $\sum_{j=1}^{i} \hat{\ell}_j \in S$ for $1 \le i \le q$.

We begin by computing an unrounded solution. To this aim, we set $c := \hat{c} + \mathrm{gap}(S)$, and use Theorem 5.2 to compute the optimal solution to an (n, m, q, c)-instance. Th constraint on c in Theorem 5.2 translates to a slightly stronger constraint on $\hat{c}$, namely

$$\hat{c} + \mathrm{gap}(S) \leq 2 \log_r m + \Delta_r; \tag{5.6}$$

we assume this from now on. Let $\ell_1, \ldots, \ell_q$ be this unrounded solution.

Now, we greedily deduce a rounded solution as follows: Pick for $\hat{\ell}_1$ the largest value in S such that $\hat{b}_1 \geq b_1$. Then find the largest $\hat{\ell}_2$ such that $\hat{\ell}_1 + \hat{\ell}_2 \in S$, and such that $\hat{b}_2 \geq b_2$. Proceed to determine $\hat{\ell}_3, \ldots, \hat{\ell}_{q-1}$ in that order. Finally, pick $\hat{\ell}_q$ such that $\sum_{i=1}^{q} \hat{\ell}_i \in S$ and $\hat{b}_q \in [\hat{c}, c]$. (This last choice is possible because of 5.6 and the fact that we choose c such that $c - \hat{c} \geq \mathrm{gap}(S)$.)

We claim that the $\hat{\ell}_i$ obtained this way are nonnegative. This is clear for $1 \leq i \leq q-1$, because choosing $\hat{\ell}_i = 0$ already yields

$$\hat{b}_i = 2\hat{b}_{i-1} + \Delta_r \geq 2b_{i-1} + \Delta_r \geq b_i.$$

In addition we have, by our choice of $\hat{b}_q$,

$$\hat{\ell}_q = 2\hat{b}_{q-1} + \Delta_r - \hat{b}_q \geq 2b_{q-1} + \Delta_r - c = \ell_q \geq 0.$$

So we can use Claim 5.3 and conclude that this solution is feasible.

5.4.3 Combining partial elimination and rounding

For the partial elimination algorithm in section 5.4.1 to work, the values ℓ_i must be chosen so that partial elimination sets of the appropriate size exist. This leads to the following choice of S for the rounding scheme of section 5.4.2:

$$S = \{t \in \mathbb{R} \mid r^{1-(t-\lfloor t \rfloor)} \in \mathbb{N}\}.$$

Let $\ell_1, \ldots, \ell_q$ be a rounded solution. The size of the partial elimination set at level j can then be computed as follows. If $t := \sum_{i=1}^{j} \ell_i$ is an integer, then no partial elimination takes place in this round. Otherwise we have a partial elimination set of size $|V_j| = r^{1-(t-\lfloor t \rfloor)} \in \mathbb{N}$. Note that $1 \leq |V_j| \leq r - 1$.

We now turn our attention to the cost (in terms of the increase in maximal list size) of this rounding. We first need to compute $\mathrm{gap}(S)$. We have $\mathbb{N} \subset S$, and $S + z = S$ for any $z \in \mathbb{Z}$. So we can restrict our attention to the points of S in the interval $[0, 1]$, which are the points $\log_r(r/i)$ for $i = 1, \ldots, r$. The distance between the neighbors $\log_r(r/i)$ and $\log_r(r/(i+1))$ is $\log_r(1 + i^{-1})$. This is largest if $i = 1$, in which case the value is $\log_r 2$. Hence, we have $\mathrm{gap}(S) = \log_r 2$.

Therefore, in the algorithm of section 5.4.2, we set $c = \hat{c} + \log_r 2$, where $\hat{c}$ is the true target value. The cost increase for finding a solution to an (n, m, q, c)-instance rather than an $(n, m, q, \hat{c})$-instance is at most a factor $r^{\log_r 2} = 2$, if we ignore rounding. Now, since $\hat{b}_i \leq b_i + \log_r 2$, the rounding itself contributes another factor $r^{\log_r 2} = 2$ to the cost, resulting in a rounded version that has a maximal list length at most 4 times that of the unrounded solution.

Remark. In section 5.4.1 we remarked that $\ell_i \geq 1$ is needed for $p < i \leq q$ in order for partial elimination to work, and that this follows from $u \geq 1$. For the rounded solution, however, $u \geq 1$ is not sufficient to guarantee $\ell_i \geq 1$; we also need $\Delta_r \geq \log_r 2$, which holds for all $r \geq 3$. Thus the present rounding scheme is not applicable to the binary case.

5.4.4 Uniformity of partially eliminated symbols

In our analysis in section 5.1, we made the assumption that the elements in the partial elimination position are uniformly distributed. Strictly speaking, this assumption is wrong: the fact that we normalize vectors has the effect that the value 1 appears slightly more often than other values.

On the other hand, if we disregard vectors that are zero in every position other than the last, the distribution *is* uniform. We will thus bound the probability that we encounter any such "bad" vector throughout the algorithm, and simply abort whenever such a bad event happens.

The only levels that are susceptible to this problem are the levels $p, p+1, \ldots, q-1$. We start by studying a list of the $(q-1)$st level. At this level, a random vector is bad with probability $r^{1-\ell_q}$, and so a list contains a bad vector with probability at most $r^{b_{q-1}+1-\ell_q}$. Plugging in the values for ℓ_q and b_{q-1} from Theorem 5.2, we get that this probability is at most

$$\frac{r^{1+c-u}}{r-1}.$$

The total failure probability at level $q-1$ is hence at most twice this value, since we have two lists to consider. For a general level $p \le q-i \le q-1$, a similar computation yields a failure probability of

$$2^i r^{b_{q-i}+i-\sum_{j=q-i+1}^{q}\ell_j}.$$

Now note that the value $\sum_{j=q-i+1}^{q}\ell_j$ does not decrease under our rounding scheme of sections 5.4.2 and 5.4.3, and the quantity $r^{b_{q-i}}$ increases by at most a factor of two. Hence, plugging in the optimal unrounded solution for ℓ_j and b_j, and taking into account the worst-case additional penalty due to rounding, we get a maximum of

$$2^{i+1}r^{c+i(1-u-\Delta_r)} = 2r^c\left(\frac{2r^{1-u}}{r-1}\right)^i.$$

Summing from $i = 1, \ldots, \infty$ gives an upper bound on the overall probability of a bad vector appearing of

$$\frac{4r^c}{r^u(1-r^{-1})-2} \le \frac{4r^c}{m(1-r^{-1})-2}, \tag{5.7}$$

where we have used the fact that $r^u \ge m$, because the expected list length is minimum at the beginning of the algorithm. This bound is useful unless the input list length m is small compared to the desired number of solutions r^c.

5.4.5 Fast partial elimination

Partial elimination reduces the number of matches by a factor $|V|/r$. So if we just do the integer elimination part first, and then throw away non-matches for the partial part, this entails

$$r/|V| = r/(r^{1-\gamma}) = r^\gamma$$

additional operations per match. (Here, γ is the fractional part of the number ℓ of symbols to be eliminated.) This is of course not a problem if γ is small, but can introduce a factor as large as r if γ is close to 1.

We now give a different strategy that is better if γ is large. Fix the set V. We need to merge vectors of the form

$$v_1 = (1, a_1, *, \ldots, *, a_2)$$

with vectors of the form

$$v_2 = (1, b_1, *, \ldots, *, b_2).$$

Specifically, for v_1 we want to quickly find all the v_2 such that

$$v_2 - v_1 = (0, b_1 - a_1, *, \ldots, *, b_2 - a_2) \sim (0, 1, *, \ldots, *, (b_1 - a_1)^{-1}(b_2 - a_2))$$

is a partial match, i.e., $(b_1 - a_1)^{-1}(b_2 - a_2) \in V$. We do this as follows. For each $c \in V$, we seek a match with in addition

$$(b_1 - a_1)^{-1}(b_2 - a_2) = c,$$

i.e.,

$$b_2 - cb_1 = a_2 - ca_1.$$

So for a fixed c, we sort the list in such a way that the lookup for a match on the integer part together with the value $b_2 - cb_1$ is fast. Since we have to do this for every c, this slows down the algorithm by a factor of $|V| = r^{1-\gamma}$.

Combining with the first strategy, we can get away with increasing the running time by a factor of

$$\min(r^{1-\gamma}, r^{\gamma}) \leq r^{1/2}.$$

With a little care in implementation, the memory use is in either case proportional to the length of the lists. Thus we have achieved the time and space bounds claimed at the beginning of section 5.

5.5 Failure probability analysis

In this section, we bound the failure probability of the extended k-tree algorithm over large alphabets. Specifically, we shall prove the following theorem:

Theorem 5.4 *If $\ell_1, \ldots, \ell_q$ are chosen according to Theorem 5.2 and section 5.4.2, then the failure probability of the algorithm is at most*

$$r^{-c} + \frac{4r^c}{m(1 - r^{-1}) - 2} + \exp(3qk/2m) - 1.$$

Here, r^{-c} is the error term arising from the expected number of solutions, the second term comes from (5.7), and the last term follows from the second moment as described in the next two subsections. The analysis based on the second moment method is similar to but rather more technically involved than that for the binary case, as presented in section 4.

In the following failure probability analysis, in particular in the proof of Lemma 5.5, we will assume that the value 0 is not contained in any partial elimination set. Since the maximal size of a partial elimination set is $r - 1$, and the sets can be chosen freely by the algorithm, we may assume w.l.o.g. that this property holds.

5.5.1 Preliminaries

Following section 4.1, we write

$$L_i = \{x_1^i, \ldots, x_m^i\}$$

for $i = 1, \ldots, k$, where the x_j^i are $\mathbb{F}_r$-vectors of length n. The components of x_j^i are random variables, sampled uniformly and independently from $\mathbb{F}_r$. We define the outputs of the algorithm as follows. Let

$$\mathcal{S} = (\mathbb{F}_r^* \times \{1, \ldots, m\})^k.$$

Then $a = ((\lambda_1, i_1), \ldots, (\lambda_k, i_k)) \in \mathcal{S}$ is a solution for the given input if and only if

$$\lambda_1 x_{i_1}^1 + \cdots + \lambda_k x_{i_k}^k = 0.$$

Two solutions $((\lambda_1, i_1), \ldots, (\lambda_k, i_k))$ and $((\lambda_1', i_1'), \ldots, (\lambda_k', i_k'))$ are *equivalent* if

$$i_1 = i_1', \ldots, i_k = i_k' \quad \text{and} \quad \lambda_1 = c\lambda_1', \ldots, \lambda_k = c\lambda_k'$$

for some $c \in \mathbb{F}_r^*$. Writing $\overline{\mathcal{S}}$ for the set of equivalence classes of $\mathcal{S}$, we have $|\overline{\mathcal{S}}| = (r-1)^{k-1}m^k$. For $a \in \overline{\mathcal{S}}$, let I_a be the indicator random variable of the event that a is a solution found by the algorithm.[‖] The number of (nonequivalent) solutions found by the algorithm is $N := \sum_{a \in \overline{\mathcal{S}}} I_a$; the algorithm fails if and only if $N = 0$. As in section 4.1, we get can bound the probability of this event as follows:

$$\Pr(N = 0) \le \mathrm{E}[N]^{-1} + \frac{\mathrm{E}_{ab}[\mathrm{Cov}(I_a, I_b) \mid a \ne b]}{\mu^2}, \tag{5.8}$$

where $\mu := \mathrm{E}[I_a]$ and $\mathrm{E}_{ab}[\cdot]$ denotes the expectation over a, b selected randomly from $\overline{\mathcal{S}}$.

5.5.2 Covariance computation

In this section, we estimate the correlation of the events that two different sums are solutions found by the algorithm, i.e., we establish an inequality analogous to (4.2). The analysis is lengthier than that of section 4.2, mainly because of complications introduced by partial elimination.

The sample space in this section is the input of the algorithm, i.e., the random lists. For a given $a = ((\lambda_1, i_1), \ldots, (\lambda_k, i_k)) \in \overline{\mathcal{S}}$, we define the event

$$\mathcal{E}_{q,1}(a) := \{\text{the sum } \lambda_1 x_{i_1}^1 + \cdots + \lambda_k x_{i_k}^k \text{ appears in the root node}\},$$

which is the event that a is a solution found by the algorithm. Note that I_a is just the indicator random variable variable of $\mathcal{E}_{q,1}(a)$.

We can define similar events for the other nodes of the tree. For $1 \le j \le q$ and $1 \le t \le 2^{q-j}$, we denote the event that the relevant partial sum corresponding to a appears in the tth level-j node by $\mathcal{E}_{j,t}(a)$. With this notation, we have

$$\Pr(\mathcal{E}_{j,t}(a)) = r^{-\sum_{i=1}^{j} 2^{j-i}\ell_i} =: \mu_j$$

[‖] We shall abuse notation by having $a = ((\lambda_1, i_1), \ldots, (\lambda_k, i_k))$ denote either an element of $\mathcal{S}$ or its equivalence class in $\overline{\mathcal{S}}$, depending on the context. This makes sense because the events we consider hold either for all members of an equivalence class or for none of them.

for $1 \le j \le q$ and $1 \le t \le 2^{q-j}$. This is the probability that a particular set of parameters appears in a level-j node, so in particular the probability that a is a solution found by the algorithm is equal to $\Pr(\mathcal{E}_{q,1}(a)) = \mu_q = \mu$.

We now generalize the notion of an (a,b)-incidence-tree defined in section 4.2. Let $a, b \in \mathcal{S}$. The (a,b)-incidence tree is the complete binary tree of height $q+1$ which has in every node either a square or a triangle corresponding to the following rule. Write $a = ((\lambda_1, i_1), \ldots, (\lambda_k, i_k))$ and $b = ((\lambda'_1, i'_1), \ldots, (\lambda'_k, i'_k))$. Let x be a node of the tree. If for every j such that the jth node on level 0 is below x, we have $i_j = i'_j$ and $\lambda_j = c\lambda'_j$ for some c independent of j, then the node x is a triangle. Otherwise it is a square.

From this definition it follows, as in the binary case, that if a node has no triangle or exactly one triangle as child, then it is a square. However, unlike in the binary case, a node which has two triangle children can be either a triangle or a square.

The following Lemma establishes a bound analogous to (4.2).

Lemma 5.5 *Fix $a, b \in \mathcal{S}$. Assume that the the partial elimination sets are chosen so that they do not contain the value* 0*. Then we have*

$$\mathrm{Cov}(I_a, I_b) \le \mu(F_{ab} - \mu), \tag{5.9}$$

where $F_{ab} = r^{-\sum_{y \text{ square}} \ell_{\mathrm{level}(y)}}$.

Proof: We proceed by induction on the rounds of the algorithm. We need to show that

$$\Pr(\mathcal{E}_{j,1}(a) \wedge \mathcal{E}_{j,1}(b)) \le \mu_j r^{-\sum_{y \text{ square}} \ell_{\mathrm{level}(y)}}, \tag{5.10}$$

where the sum only runs over the nodes in the tree rooted at the $(j,1)$-node: applied to the root node, the latter can be rewritten as

$$\mathrm{E}[I_a I_b] \le \mu_q F_{ab},$$

which in turn is just a restatement of (5.9).

Write $\mathcal{N}_{j,i}(a)$ for the event that the constraint implied by the node itself holds. Hence $\mathcal{N}_{j,i}(a)$ constrains the values of the appropriate sum at the previous partial elimination position (if any) and at the newly eliminated positions to be zero, and puts a constraint on the value at the new partial elimination position. So we have, for example,

$$\mathcal{E}_{j,1}(a) = \mathcal{E}_{j-1,1}(a) \wedge \mathcal{E}_{j-1,2}(a) \wedge \mathcal{N}_{j,1}(a);$$

but note that if a symbol was partially eliminated in round $j-1$, the three events in this conjunction are not necessarily independent.

We have, using the induction hypothesis at level $j-1$,

$$\begin{aligned}
\Pr(\mathcal{E}_{j,1}(a) \wedge \mathcal{E}_{j,1}(b)) &= \Pr(\mathcal{N}_{j,1}(a) \wedge \mathcal{N}_{j,1}(b) \mid \mathcal{E}_{j-1,1}(a) \wedge \mathcal{E}_{j-1,2}(a) \wedge \mathcal{E}_{j-1,1}(b) \wedge \mathcal{E}_{j-1,2}(b)) \\
&\quad \times \Pr(\mathcal{E}_{j-1,1}(a) \wedge \mathcal{E}_{j-1,2}(a) \wedge \mathcal{E}_{j-1,1}(b) \wedge \mathcal{E}_{j-1,2}(b)) \\
&\le \Pr(\mathcal{N}_{j,1}(a) \wedge \mathcal{N}_{j,1}(b) \mid \mathcal{E}_{j-1,1}(a) \wedge \mathcal{E}_{j-1,2}(a) \wedge \mathcal{E}_{j-1,1}(b) \wedge \mathcal{E}_{j-1,2}(b)) \\
&\quad \times r^{-\sum_{y \text{ square}} \ell_{\mathrm{level}(y)}} r^{-\sum_{y \text{ square}} \ell_{\mathrm{level}(y)}} \mu_{j-1}^2,
\end{aligned} \tag{5.11}$$

where the sums in the exponents run over the subtrees rooted at $(j-1,1)$ and $(j-1,2)$ respectively.

Now we distinguish some cases. First, if the $(j,1)$-node is a triangle, then all the nodes below it are triangles, and the inequality (5.10) is trivial in this case.

Second, we show that the case in which the $(j,1)$-node is a square but its children are triangles cannot occur if partial elimination took place in round $j-1$. To this aim, we show that $\mathcal{N}_{j,1}(a)$ and $\mathcal{N}_{j,1}(b)$ are mutually exclusive in this case. Write $a = ((\lambda_1, i_1), \ldots, (\lambda_{2^q}, i_{2^q}))$ and $b = ((\lambda'_1, i'_1), \ldots, (\lambda'_{2^q}, i'_{2^q}))$. Since nodes $(j-1,1)$ and $(j-1,2)$ are triangles, we have $i_1 = i'_1, \ldots, i_{2^j} = i'_{2^j}$, and $\lambda_1 = c\lambda'_1, \ldots, \lambda_{2^{j-1}} = c\lambda'_{2^{j-1}}$ and $\lambda_{2^{j-1}+1} = d\lambda'_{2^{j-1}+1}, \ldots, \lambda_{2^j} = d\lambda'_{2^j}$ for some $c, d \in \mathbb{F}_r^*$. Now let $z_e \in \mathbb{F}_r$, $1 \le e \le 2^j$, be the value of $x_{i_e}^e$ at the partial elimination position at round $j-1$. Since $\mathcal{E}_{j,1}(a)$ holds, we have

$$\underbrace{\lambda_1 z_1 + \cdots + \lambda_{2^{j-1}} z_{2^{j-1}}}_{=:T_1} + \underbrace{\lambda_{2^{j-1}+1} z_{2^{j-1}+1} + \cdots + \lambda_{2^j} z_{2^j}}_{=:T_2} = 0.$$

Similarly, because of $\mathcal{E}_{j,1}(b)$, we also have $cT_1 + dT_2 = 0$, giving

$$\begin{pmatrix} 1 & 1 \\ c & d \end{pmatrix} \begin{pmatrix} T_1 \\ T_2 \end{pmatrix} = 0.$$

Now, T_1 and T_2 are nonzero because those values are contained in some partial elimination set, and hence the determinant of the above matrix must be zero; thus $c = d$. However, if $c = d$ then the $(j,1)$-node is a triangle, which we excluded. Therefore $\mathcal{N}_{j,1}(a)$ and $\mathcal{N}_{j,1}(b)$ are mutually exclusive conditional on $\mathcal{E}_{j-1,1}(a)$, $\mathcal{E}_{j-1,2}(a)$, $\mathcal{E}_{j-1,1}(b)$ and $\mathcal{E}_{j-1,2}(b)$ in this case, so (5.10) again holds trivially.

Second, if no partial position was eliminated in round $j-1$, the events $\mathcal{N}_{j,1}(a)$ and $\mathcal{N}_{j,1}(b)$ are independent conditional on $\mathcal{E}_{j-1,1}(a) \wedge \mathcal{E}_{j-1,2}(a) \wedge \mathcal{E}_{j-1,1}(b) \wedge \mathcal{E}_{j-1,2}(b)$. Therefore, we have

$$\begin{aligned} &\Pr(\mathcal{N}_{j,1}(a) \wedge \mathcal{N}_{j,1}(b) \mid \mathcal{E}_{j-1,1}(a) \wedge \mathcal{E}_{j-1,2}(a) \wedge \mathcal{E}_{j-1,1}(b) \wedge \mathcal{E}_{j-1,2}(b)) \\ &\quad = \Pr(\mathcal{N}_{j,1}(a) \mid \mathcal{E}_{j-1,1}(a) \wedge \mathcal{E}_{j-1,2}(a)) \times \Pr(\mathcal{N}_{j,1}(b) \mid \mathcal{E}_{j-1,1}(b) \wedge \mathcal{E}_{j-1,2}(b)), \\ &\quad = r^{-2\ell_j}. \end{aligned} \tag{5.12}$$

Plugging (5.12) into (5.11) establishes the desired inequality (5.10) for j in this case.

Finally, if $(j,1)$ is a square and only one of its children is a triangle, equation (5.12) is still valid, again by conditional independence, and thus (5.10) also holds in this case. □

5.5.3 Estimating the expected covariance

Lemma 5.5 establishes an upper bound on the covariance $\mathrm{Cov}(I_a, I_b)$ for given $a, b \in \overline{\mathcal{S}}$. Now, in analogous fashion to section 4.3, we randomize the choice of $a, b \in \overline{\mathcal{S}}$ and compute an upper bound on $\mathrm{E}[\mathrm{Cov}(I_a, I_b) \mid a \neq b]$.

Define

$$\begin{aligned} \alpha_j &= \Pr(\text{a level-}j \text{ node } y \text{ has exactly one } \triangle \text{ child} \mid y \text{ is a } \square) \\ \beta_j &= \Pr(\text{a level-}j \text{ node } y \text{ has two } \triangle \text{ children} \mid y \text{ is a } \square) \end{aligned}$$

Then, a short computation gives

$$\alpha_j = \frac{2m^{-2^{j-1}}(r-1)^{-(2^{j-1}-1)}(1 - m^{-2^{j-1}}(r-1)^{-(2^{j-1}-1)})}{1 - m^{-2^j}(r-1)^{-(2^j-1)}}$$

and

$$\beta_j = \frac{(r-2)m^{-2^j}(r-1)^{-(2^j-1)}}{1-m^{-2^j}(r-1)^{-(2^j-1)}}.$$

The denominator is in each case close to 1, so we get a good upper bound if we discard it. Doing this, and writing $t_j := m^{-2^j}(r-1)^{-(2^j-1)}$, we get

$$\begin{aligned}\alpha_j &\le 2t_{j-1}(1-t_{j-1}) \le 2t_{j-1},\\ \beta_j &\le (r-2)t_j.\end{aligned}$$

Note that the t_j satisfy the recursion $t_j = \frac{t_{j-1}^2}{r-1}$. Also, t_j^{-1} is equal to the number of distinct candidate sums (up to equivalence) for a level-j node, and so we have

$$\mu_j \cdot t_j^{-1} = \mathrm{E}[\text{list length at level } j].$$

For $1 \le j \le q$, we define S_j in analogous fashion to section 4.3. Let y be a node at level $j \ge 1$, and set

$$S_j := \prod_{\substack{z \text{ a square}\\ \text{descendant of } y}} r^{-\ell_{\mathrm{level}(z)}}.$$

Note that $S_q = F_{ab}$, where F_{ab} is the quantity defined in Lemma 5.5. Set $s_j := \mathrm{E}_{ab}^{\square}[S_j]$. Note that $s_j \ge \mu_j$. Our goal is to upper bound s_q/μ_q. First, the s_j satisfy the following:

$$\begin{aligned}s_j &= r^{-\ell_j}[(1-\alpha_j-\beta_j)s_{j-1}^2 + \alpha_j s_{j-1} + \beta_j]\\ &\le r^{-\ell_j}s_{j-1}^2\left[1+\frac{\alpha_j}{s_{j-1}}+\frac{\beta_j}{s_{j-1}^2}\right]\\ &\le r^{-\ell_j}s_{j-1}^2\left[1+\frac{2t_{j-1}}{\mu_{j-1}}+\frac{r-2}{r-1}\left(\frac{t_{j-1}}{\mu_{j-1}}\right)^2\right]\\ &\le \mu_j\prod_{i=0}^{j-1}\left[1+\frac{2t_i}{\mu_i}+\frac{r-2}{r-1}\left(\frac{t_i}{\mu_i}\right)^2\right]^{2^{j-i-1}}.\end{aligned}$$

Now, using the approximation $\ln(1+x) \le x$ and the fact that the expected list size $t_i^{-1}\mu_i$ is at least 1 for any level $< q$, we get

$$s_q \le \mu_q \exp\left\{\sum_{i=0}^{q-1} 2^{q-i-1}\frac{t_i}{\mu_i}\left(2+\frac{(r-2)t_i}{(r-1)\mu_i}\right)\right\} \le \mu_q \exp\left\{\frac{3}{2}\sum_{i=0}^{q-1}2^{q-i}\frac{t_i}{\mu_i}\right\} \le \mu_q \exp\left\{\frac{3q2^q}{2m}\right\}. \quad (5.13)$$

To get the last inequality, we bounded the sum by the maximum times q; the largest term is the one for $i=0$, since the smallest expected list length at any level $< q$ is at level 0, where it is m.

5.5.4 Proof of Theorem 5.4

We are now ready to prove Theorem 5.4. Using first (5.9) and then (5.13), we have

$$\mathrm{E}_{ab}[\mathrm{Cov}(I_a, I_b) \mid a \ne b] \le \mu_q(\mathrm{E}_{ab}^{\square}[F_{ab}] - \mu_q) \le \mu_q^2\left(\exp\left(\frac{3qk}{2m}\right)-1\right).$$

Now, plugging this and the fact that $\mathrm{E}[N] = r^c$ into (5.8), and adding the probability of a bad vector from (5.7), we get the statement of Theorem 5.4.

6 Concluding remarks and open problems

The problem studied in this paper is sometimes referred to as the *multi-list* k-sum problem, to emphasise the fact that the summands are taken from k distinct lists. An interesting variant is the *single-list* k-sum problem, which is the problem of finding k distinct entries in one list L that sum to zero. The single-list problem has a number of natural applications; for example, the problem of finding a sparse multiple of a given polynomial reduces more naturally to the single-list k-sum problem.

It seems obvious that an algorithm similar to the extended k-tree algorithm can be used for the single-list problem, but it appears to be much more difficult to analyze the behavior of such an algorithm. The difficulties stem from the fact that the lists are no longer independent, and that the event that some sum appears in a node of the tree is related to the events that permutations of that sum appear. These effects make it hard to get a rigorous estimate even of the expected list sizes, let alone an estimate of their variance.

It is of course always possible to reduce the single-list problem with k summands to a multi-list problem with k independent lists simply by chopping up the input list into k distinct lists. However, this artificially reduces the effective list length and removes a large number of sums from consideration. It would be interesting to develop, and rigorously analyze, an algorithm tailored to the single-list case.

Another point worth mentioning is that our version of the k-tree algorithm on finite fields differs from the binary version in an interesting respect. As we observed at the end of section 2, in the binary algorithm it is possible to randomly choose the constraints on the internal nodes of the tree, and hence sample a random solution of the original problem. Our non-binary version no longer has this property, but rather finds certain special solutions only. We do not currently know whether it is possible to modify the non-binary algorithm so that it finds arbitrary solutions.

Lastly, our non-binary algorithm has a merging cost overhead of $\sqrt{r}$. This additional cost is somewhat surprising, and it would be nice to avoid it (without substantially increasing the space requirement). Such a savings could be significant in applications where the field size r is large.

References

[1] Miklós Ajtai, Ravi Kumar and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pages 601–610, 2001.

[2] Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: incrementality at reduced cost. *Proceedings of Eurocrypt '97*, LNCS 1233, pages 163–192, Springer-Verlag, 1997.

[3] Avrim Blum, Adam Tauman Kalai and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *Journal of the ACM* **50**(4), pages 506–519, 2003. (Extended abstract appeared in *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, 2000.)

[4] Andrew Brown and Amin Shokrollahi. Algebraic-geometric codes over the erasure channel. *Proceedings of the IEEE International Symposium on Information Theory*, page 77, 2004.

[5] Paul Camion and Jacques Patarin. The knapsack hash function proposed at Crypto'89 can be broken. *Proceedings of Eurocrypt '91*, LNCS 547, pages 39–53, Springer-Verlag, 1991.

[6] Philippe Chose, Antoine Joux and Michel Mitton. Fast correlation attacks: an algorithmic point of view. *Proceedings of Eurocrypt '02*, LNCS 2332, pages 209–221, Springer-Verlag, 2002.

[7] Jean-Sebastien Coron and Antoine Joux. Cryptanalysis of a provably secure cryptographic hash function. Cryptology ePrint Archive Report 2004/013, 2004. http://eprint.iacr.org/2004/013

[8] Daniel Augot, Matthieu Finiasz and Nicolas Sendrier. A family of fast syndrome based cryptographic hash functions. *Proceedings of Mycrypt 2005*, LNCS 3715, Springer-Verlag, 2005.

[9] Ravi Kumar and D. Sivakumar. On polynomial approximation to the shortest lattice vector length. *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 126–127, 2001.

[10] Vadim Lyubashevsky. On random high density subset sums. *Proceedings of APPROX-RANDOM 2005*, LNCS 3624, pages 378–389, Springer-Verlag, 2005.

[11] Lorenz Minder and Alistair Sinclair. The extended k-tree algorithm. *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 586–595, 2009.

[12] Andrew Shallue. An improved multi-set algorithm for the dense subset sum problem. *Proceedings of Algorithmic Number Theory Symposium, ANTS VIII*, LNCS 5011, pages 416–429, Springer-Verlag, 2008.

[13] David Wagner. A generalized birthday problem. *Proceedings of CRYPTO 2002*, LNCS 2442, pages 288–303, Springer-Verlag, 2002.