

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, \dots, L_k , each of length m , find $x_1 \in L_1, \dots, x_k \in L_k$ such that $x_1 + \dots + x_k = 0$, 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.

1 Introduction

1.1 Background. The *k-sum problem* is the following. We are given k lists L_1, \dots, 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 $x_1 \in L_1, \dots, x_k \in L_k$ such that

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

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 [10] presents a number of applications in cryptography, while

a recent paper of Coron and Joux [6] shows how to use the k -sum problem to find codewords in a certain context. Other applications include finding shortest lattice vectors [1, 7], solving subset sum problems [8], 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.,

$$(1.1) \quad m \geq 2^{n/2^q}.$$

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

A simple algorithm for solving this problem is based on the birthday paradigm, and works as follows. Compute a list S_1 of sums $x_1 + \dots + x_{2^{q-1}}$, and a list S_2 of sums $x_{2^{q-1}+1} + \dots + x_{2^q}$, 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 $\tilde{O}(2^{n/2})$.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 [10] showed that if

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

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, \dots, 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

*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.

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

²In this paper, the notation \tilde{O} hides factors that are logarithmic in the running time—i.e., polynomial in $n, \log m$ and q .

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

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

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 simple birthday algorithm.³

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 view is taken in Wagner's paper [10], 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 constants 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 [6], 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 [10]. 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 $x_1 + \dots + x_k = 0$ 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 classical birthday 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 birthday 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)⁴. 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 $\mathbb{E}[|L'_1|] = 2^{-n/7} |L_1| |L_2| = 2^{3n/7}$. We then eliminate the remaining $\ell_2 = 6n/7$ bits in the second round, giving us an expected number of solutions equal to

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

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

³Throughout the paper, expectations are taken over the random input lists. The algorithms are deterministic.

⁴Throughout the paper, \log denotes base-2 logarithm

is still significantly better than the $\tilde{O}(2^{n/2})$ birthday 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 naively aims for *two* solutions in expectation then the failure probability will be at most slightly larger than $1/2$.

1.3 Related work. The basic idea of the k -tree algorithm was apparently rediscovered several times. In 1991, Camion and Patarin constructed a k -tree scheme for breaking knapsack-based hash functions [4]. In 2000, Blum, Kalai and Wasserman [3] devised a similar algorithm to prove a conjecture in learning theory. In 2002, Wagner published a paper dedicated entirely to the k -tree algorithm, including some extensions and several applications.

In the same year, Chose, Joux and Mitton [5] 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 [6] 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.

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 [8] 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. Like 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 [9] 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 grow 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 [10] is a special case.

Given an instance of the k -sum problem as described in the Introduction, the (extended) k -tree algo-

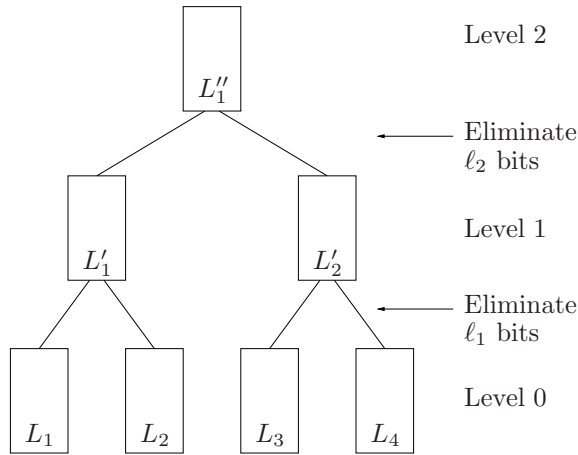


Figure 1: The k -tree algorithm for $k = 4$.

gorithm 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 $L''_1, \dots, L''_{2^{q-2}}$ are created from $L'_1, \dots, L'_{2^{q-1}}$, eliminating a further sequence of ℓ_2 bits and thus causing the vectors in the lists $L''_1, \dots, L''_{2^{q-2}}$ 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

$$(2.3) \quad \sum_{i=1}^q \ell_i \geq n.$$

The k -tree 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, \dots, 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 ℓ_1, \dots, ℓ_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

$$(2.4) \quad \mathbb{E}[M_q] \geq 2^c.$$

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, in order to obtain non-trivial bounds on the probability that the algorithm finds a solution, the value of c must be chosen slightly larger (e.g., $c = 1$). This entails a slight tightening of condition (1.1), which becomes

$$(2.5) \quad m \geq 2^{(n+c)/2^q}.$$

In the remainder of the paper we shall always assume that (2.5) holds. 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. Finally, we will assume

$$(2.6) \quad m \leq 2^{(n+c)/(q+1)},$$

since Wagner's algorithm applies for all larger m .

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 $\mathbb{E}[M_j]$ 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 $\mathbb{E}[M_j]$ is not too large for any $1 \leq j \leq q - 1$; however, we also need to ensure that the constraints (2.3) and (2.4) both hold.

As a simple example, assuming m is a power of two, Wagner's original k -tree algorithm [10] chooses $\ell_j = \log m$ for $j = 1, \dots, q - 1$ and $\ell_q = 2 \log m$, leading to $\mathbb{E}[M_j] = m$ for $j = 1, \dots, q - 1$ and $\mathbb{E}[M_q] = 1$, i.e., all lists (except the last) have the same expected length as the initial lists. In this case, condition (2.3) translates to $(q+1) \log m \geq n$, 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 a solution (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

$$(3.7) \quad b_j = 2b_{j-1} - \ell_j,$$

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 $2^{b_j} = E[M_j]$. 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

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

which proves (3.7). 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

$$(3.8) \quad \max_{0 \leq i \leq q-1} 2^{b_i}.$$

(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 (3.8). 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 (3.7) 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:

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

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 naive birthday 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 (2.5), (2.6) and $c < 2 \log m$, the linear program defined above is feasible and has an optimal solution of the following form:*

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

where p is the least integer such that

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

and

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

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 $\ell_1 = \dots = \ell_{q-1} = 0$, $\ell_q = n$. From (3.7), it follows then that $b_i = 2^i b_0$ for $i < q$, and $b_q = 2^q b_0 - n$. Set $u = 2^{q-1} b_0 = \max_{i \leq q-1} b_i$.

We now verify that this solution is feasible. Clearly, all the ℓ_i are non-negative and $\sum_{i=0}^q \ell_i \geq n$. The condition $b_q \geq c$ translates to $2^q b_0 \geq n + c$, which, recalling that $b_0 = \log m$, is equivalent to condition (2.5) 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 (3.7)) this will establish the theorem.

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

$$\ell_j \mapsto \ell_j - \varepsilon, \quad \ell_{j+1} \mapsto \ell_{j+1} + 2\varepsilon, \quad b_j \mapsto b_j + \varepsilon$$

yields another feasible solution with the same value of u . (This can easily be checked using the recursive definition (3.7).) Note that this transformation increases the sum of the ℓ_i by ε , so the constraint $\sum_{i=1}^q \ell_i \geq n$ becomes slack.

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

$$\ell_q \mapsto \ell_q + \varepsilon, \quad b_q \mapsto b_q - \varepsilon$$

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 $j \in \{1, \dots, q-1\}$, 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 (3.7) 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 (2.6). So now let $1 \leq j \leq q-1$ be an index such that $b_j = u$, and consider the transformation

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

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 (3.7) we would have $b_{j+1} = 2b_j = 2u$. If $j+1 < q$ this gives a contradiction because $b_{j+1} \leq u$. And if $j+1 = q$ then $c = b_{j+1} = 2u \geq 2b_0 = 2 \log m$, 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 $1 \leq j \leq q-1$. 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 (3.7) we have $b_{j+1} = 2u$ which contradicts the constraint $b_{j+1} \leq u$.

Equation (3.7) can now be used to reconstruct the values of $\ell_p, \dots, \ell_q, b_1, \dots, b_{p-1}$ by direct computation.

It remains to determine p and u . From $b_{p-1} \leq u$ and $0 \leq \ell_p = 2^p b_0 - u$ we get $2^{p-1} b_0 \leq u \leq 2^p b_0$. Since, as we have seen above, the constraint $\sum_i \ell_i \geq n$ 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 $p \in \{1, \dots, q-1\}$ 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 (2.5), 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 ℓ_1, \dots, ℓ_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 $\tilde{u}, \tilde{\ell}_1, \dots, \tilde{\ell}_q, \tilde{b}_1, \dots, \tilde{b}_q$ for the putative integer solution obtained by applying the above rounding to the LP solution $u, \ell_1, \dots, \ell_q, b_1, \dots, b_q$. Then $\sum_{i=1}^q \tilde{\ell}_i = (q-p)u + 2^p b_0 - c$, which is increasing with u since $q-p \geq 0$; hence $\sum_{i=1}^q \tilde{\ell}_i \geq n$, as required. To see that $\tilde{\ell}_j \geq 0$, note that clearly $\tilde{\ell}_j \geq \ell_j$ for all j except $j = p$. But we also have $\tilde{\ell}_p \geq 0$ because $z - u \geq 0$ implies $z - \lceil u \rceil \geq 0$ for any integer z .

Finally, it can be checked by direct computation that the \tilde{b}_i and $\tilde{\ell}_i$ still satisfy (3.7).

Remark: The constraint $\sum_i \ell_i \geq n$ 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 birthday 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.

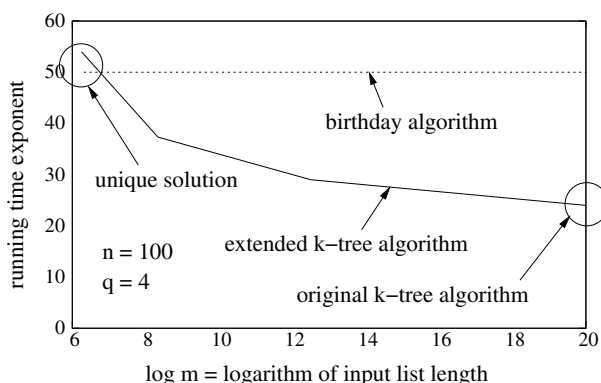


Figure 2: Comparison of the extended k -tree algorithm with the birthday algorithm

At the left border, for $m = 2^{6.25}$, our algorithm is nothing but a (somewhat complicated) variant of the birthday 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 birthday algorithm and the original k -tree algorithm as special cases, but does substantially better than the birthday 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 birthday 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 done 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 birthday 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 slightly larger than 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 [10], 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 $L_1 = (x_1^1, \dots, x_m^1), \dots, L_{2^q} = (x_1^{2^q}, \dots, x_m^{2^q})$. Let $\mathcal{S} = \{1, \dots, m\}^{2^q}$, and let $a = (a_1, \dots, a_{2^q}) \in \mathcal{S}$. Then the vector of indices a corresponds to a solution found by the algorithm if $x_{a_1}^1 + \dots + x_{a_{2^q}}^{2^q} = 0$, 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 $2^{q+1} - 1$ such constraints to be satisfied.

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

$$\begin{aligned} \Pr(N = 0) &\leq \frac{\text{Var}(N)}{\mathbb{E}[N]^2} \leq \frac{|\mathcal{S}|\mu + \sum_{a,b \in \mathcal{S}, a \neq b} \text{Cov}(I_a, I_b)}{|\mathcal{S}|^2 \mu^2} \\ (4.9) \quad &\leq \mathbb{E}[N]^{-1} + \frac{\mathbb{E}_{ab}[\text{Cov}(I_a, I_b) \mid a \neq b]}{\mu^2}, \end{aligned}$$

where $\mathbb{E}_{ab}[\cdot]$ 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 $\mathbb{E}[N]^{-1}$. 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 (4.9).

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 i th leaf (from the left) is associated with the i th 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 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, see Figure 3.

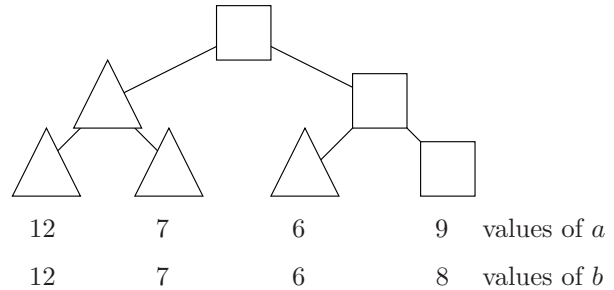


Figure 3: (a, b) -incidence tree for $q = 2$

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

$$I_a = \prod_x J_a(x),$$

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

For fixed internal nodes x and y , the random variables $J_a(x)$ and $J_b(y)$ are equal if $x = y$ and x is a triangle in the (a, b) -incidence tree. Otherwise $J_a(x)$ and $J_b(y)$ are independent. In particular, the $J_a(x)$ (where x runs over the internal nodes) are mutually independent. So for a node x at level $j \geq 1$, we have

$$\mathbb{E}[J_a(x)J_b(x)] = \begin{cases} \mathbb{E}[J_a(x)] = 2^{-\ell_j} & \text{if } x \text{ is } \triangle; \\ \mathbb{E}[J_a(x)]\mathbb{E}[J_b(x)] = 2^{-2\ell_j} & \text{if } x \text{ is } \square. \end{cases}$$

Writing $F_{ab} := \prod_{x \text{ square}} J_b(x)$, we have

$$\mathbb{E}[F_{ab}] = \mathbb{E}\left[\prod_{x \text{ square}} J_b(x)\right] = 2^{-\sum_{x \text{ square}} \ell_j},$$

where j is the level of the node x . Furthermore,

$$\mathbb{E}[I_a I_b] = \mathbb{E}[I_a F_{ab}] = \mathbb{E}[I_a] \mathbb{E}[F_{ab}] = \mu \mathbb{E}[F_{ab}].$$

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

$$(4.10) \quad \text{Cov}(I_a, I_b) = \mathbb{E}[I_a I_b] - \mu^2 = \mu (\mathbb{E}[F_{ab}] - \mu).$$

4.3 Computing the average covariance.

We now derive an exact recursive formula for $\mathbb{E}_{ab}[\text{Cov}(I_a, I_b) \mid a \neq b]$. To this end, we study the behavior of the random variable F_{ab} when a and b are

random. For a node x at level $j \geq 1$, define

$$S_j := \prod_{\substack{y \text{ a square} \\ \text{descendant of } x}} 2^{-\ell_i},$$

where i is the level of the node y , and y runs over all square internal nodes in the subtree whose root is x . With this notation, note that $E[F_{ab}] = S_q$. For $j \geq 1$, we write $E_{ab}^\square[S_j]$ 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

$$(4.11) \quad E_{ab}^\square[S_j] = 2^{-\ell_j} E_{ab}^\square[S_{j-1}] (E_{ab}^\square[S_{j-1}] (1 - \alpha_j) + \alpha_j),$$

where

$$\begin{aligned} \alpha_j &= \Pr(\text{a level } j \text{ node } x \text{ has a } \triangle \text{ child} \mid x \text{ is a } \square) \\ &= \frac{2 \cdot m^{-2^{j-1}}}{1 + m^{-2^{j-1}}}. \end{aligned}$$

Now, equation (4.11) can be used recursively to compute $E_{ab}^\square[S_q]$. From this we get $E_{ab}[\text{Cov}(I_a, I_b) \mid a \neq b]$ via the relation $E_{ab}[\text{Cov}(I_a, I_b) \mid a \neq b] = \mu(E_{ab}^\square[S_q] - \mu)$, which follows from (4.10) and the facts that $E[F_{ab}] = S_q$ and $E_{ab}^\square[S_q] = E_{ab}[S_q \mid a \neq b]$.

Putting everything together, we get that the error bound (4.9) of section 4.1 can be written as

$$(4.12) \quad \Pr(N = 0) \leq E[N]^{-1} + \mu^{-1} E_{ab}^\square[S_q] - 1.$$

Note that the quantity $\mu^{-1} E_{ab}^\square[S_q] - 1$ captures the contribution due to dependencies between the indicator random variables I_a .

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 $E_{ab}[\text{Cov}(I_a, I_b) \mid a \neq b]$, 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. The technology of the previous section provides a method for numerically bounding the failure probability of the extended k -tree algorithm for any particular set of parameter values (ℓ_1, \dots, ℓ_q) . We now give an analytic upper bound on this failure probability for the optimal choice of the ℓ_i that is useful in many applications.

THEOREM 4.1. *If ℓ_1, \dots, ℓ_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 $2^c = E[N]$ is the expected number of solutions.

Proof. In light of inequality (4.12), it is enough to show that the quantity $\mu^{-1} E_{ab}^\square[S_q]$ is bounded above by $\exp(qk/m)$.

Define $s_j := 2^{-\ell_j} s_{j-1}^2 (1 + \frac{\alpha_j}{s_{j-1}})$ with $s_0 = 1$, and $\mu_j := 2^{-\ell_j} \mu_{j-1}^2$ with $\mu_0 = 1$. Then, by inspecting the recursions, we get $\mu_q = 2^{-\sum_{i=1}^q \ell_i 2^{q-i}} = \mu$, $\mu_j \leq s_j$, and $s_j \geq E_{ab}^\square[S_j]$. (Note that s_j is in fact a rather close approximation to $E_{ab}^\square[S_j]$.)

Unwinding the formula for s_q , we get

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

Thus we have

$$(4.13) \quad \mu^{-1} E_{ab}^\square[S_q] - 1 \leq \prod_{j=0}^{q-1} \left(1 + \frac{\alpha_{j+1}}{s_j}\right)^{2^{q-j-1}} - 1.$$

Our goal is to find an ε which is an upper bound on the right hand side of (4.13). Using the fact that $\ln(1+x) \leq x$ and $\mu_j \leq s_j$, it suffices to choose ε satisfying

$$\sum_{j=0}^{q-1} 2^{q-j-1} \frac{\alpha_{j+1}}{\mu_j} \leq \ln(1 + \varepsilon).$$

The sum on the left hand side has q terms, so we can approximate it by the largest summand times q . We have $\alpha_j \leq 2m^{-2^{j-1}} = 2^{1-2^{j-1}b_0}$ and $\mu_j = 2^{-\sum_{i=1}^j \ell_i 2^{j-i}}$. Plugging in these values, and taking the logarithm, we get the condition

$$\begin{aligned} \max_{j=0, \dots, q-1} \left(q - j - 1 + (1 - 2^j b_0) + \sum_{i=1}^j \ell_i 2^{j-i} \right) \\ \leq \log(q^{-1} \ln(1 + \varepsilon)). \end{aligned}$$

So, if ε satisfies

$$(4.14) \quad q - j - 2^j b_0 + \sum_{i=1}^j \ell_i 2^{j-i} \leq \log(q^{-1} \ln(1 + \varepsilon))$$

for all $j = 0, \dots, q-1$, then it is an upper bound on the right hand side of (4.13).

For $j = 0$, condition (4.14) becomes $q 2^{q-b_0} \leq \ln(1 + \varepsilon)$, which, recalling that $b_0 = \log m$ and $q = \log k$, can be rewritten as

$$(4.15) \quad \exp(qk/m) - 1 \leq \varepsilon.$$

The left-hand side of (4.15) is the value of ε required by the statement of the theorem. To finish the proof, we must show that the conditions on ε given by (4.14) are weaker for $j = 1, \dots, q - 1$, if the ℓ_j are chosen as in Theorem 3.1; the same reasoning applies if the ℓ_j are chosen as in Claim 3.2.

First, consider the case $j < p$. Then, since $\ell_1 = \dots = \ell_j = 0$, the left hand side of (4.14) is decreasing with increasing j ; therefore, if (4.15) holds then (4.14) holds for all such j .

Now we consider the case $p \leq j < q$. Inserting the optimal values for the ℓ_i given in Theorem 3.1, we get that the left hand side of (4.14) is equal to $q - j - u$, and since $u \geq b_0$ this is again weaker than (4.15).

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 the ideal value of $1/2$ and only slightly larger than the value 0.5000017 computed at the end of the previous subsection.

Remark: The failure probability given by Theorem 4.1 differs at first sight qualitatively from those in [8, 9] for the special case of Wagner's algorithm in that it does not decay to zero with the list size. However, our bound applies to the optimal algorithm for given n , m , q and c , while in [8, 9] the list length m is padded by a factor 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 , padding the list length by a factor of α allows us to run α independent trials of our algorithm, which also causes the failure probability to decrease exponentially with α .

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] 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.
- [5] Philippe Chose, Antoine Joux and Michel Mitton. *Fast Correlation Attacks: An Algorithmic Point of View*. Proceedings of EUROCRYPT 2002, LNCS 2332, pages 209–221, Springer-Verlag, 2002.
- [6] 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>
- [7] 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.
- [8] Vadim Lyubashevsky. *On random high density subset sums*. APPROX-RANDOM, LNCS 3624, pages 378–389, Springer-Verlag, 2005.
- [9] Andrew Shallue. *An improved multi-set algorithm for the dense subset sum problem*. ANTS VIII, Algorithmic Number Theory Symposium, LNCS 5011, pages 416–429, Springer-Verlag, 2008.
- [10] David Wagner. *A Generalized Birthday Problem*. Proceedings of CRYPTO 2002, LNCS 2442, pages 288–303, Springer-Verlag, 2002.