

# Time-Memory tradeoffs for large-weight syndrome decoding in ternary codes

Pierre Karpman<sup>1</sup> and Charlotte Lefevre<sup>2</sup>

<sup>1</sup> Université Grenoble Alpes, Grenoble, France

<sup>2</sup> Radboud University, Nijmegen, The Netherlands

pierre.karpman@univ-grenoble-alpes.fr  
charlotte.lefevre@ru.nl

**Abstract.** We propose new algorithms for solving a class of large-weight syndrome decoding problems in random ternary codes. This is the main generic problem underlying the security of the recent Wave signature scheme (Debris-Alazard *et al.*, 2019), and it has so far received limited attention. At SAC 2019 Bricout *et al.* proposed a reduction to a binary subset sum problem requiring many solutions, and used it to obtain the fastest known algorithm. However—as is often the case in the coding theory literature—its memory cost is proportional to its time cost, which makes it unattractive in most applications.

In this work we propose a range of memory-efficient algorithms for this problem, which describe a near-continuous time-memory tradeoff curve. Those are obtained by using the same reduction as Bricout *et al.* and carefully instantiating the derived subset sum problem with exhaustive-search algorithms from the literature, in particular dissection (Dinur *et al.*, 2012) and dissection in tree (Dinur, 2019). We also spend significant effort adapting those algorithms to decrease their *granularity*, thereby allowing them to be smoothly used in a syndrome decoding context when not *all* the solutions to the subset sum problem are required. For a proposed parameter set for Wave, one of our best instantiations is estimated to cost  $2^{177}$  bit operations and requiring  $2^{88.5}$  bits of storage, while we estimate this to be  $2^{152}$  and  $2^{144}$  for the best algorithm from Bricout *et al.*

## 1 Introduction

At ASIACRYPT 2019, Debris-Alazard *et al.* proposed a new (conjecturally post-quantum secure) code-based signature scheme called Wave [4]. Some of the more unusual and notable features of this scheme are that it is based on *ternary* linear codes, *i.e.* codes whose alphabet is  $\mathbb{F}_3$ , and that its security relies in part on the generic hardness of some *large-weight* syndrome decoding problems. Most of the existing cryptography and coding-theory literature does not quite address either of those aspects as it tends to focus on binary codes (where low- and large-weight problems are symmetric) and, in the few existing adaptations to  $q$ -ary codes, on low-weight problems [2,13,10,17,12].

Shortly following the introduction of Wave, Bricout *et al.* introduced new dedicated algorithms for solving the specific large-weight ternary syndrome decoding instances underlying Wave’s security [1]. Their approach consists in exploiting the fact that a large-weight syndrome may be found by: 1) finding a *full-weight* syndrome for a smaller derived sub-problem and; 2) extending this smaller solution to one for the original problem, hoping that it satisfies the weight constraint. While this overall strategy is quite typical of the family of *information-set decoding* algorithms, the fact that the first step searches for *full-weight* syndromes over  $\mathbb{F}_3$  leads to a clean reduction to a  $\{0,1\}$ -subset sum problem. Furthermore, since the success probability of the second step is typically small, one in fact needs to repeat the first one many times; the best results are then obtained when many solutions for the latter can be obtained at a low (ideally constant) amortised cost. Bricout *et al.* consider several algorithms for solving the subset sum problem and obtain their best results by using Wagner’s  $k$ -tree algorithm [18] with an adaptation of the so-called representation technique. For parameter sizes relevant to Wave’s security, their best algorithm has an asymptotic time cost of  $\mathcal{O}(2^{0.0176n})$ , where  $n$  is the length of the code. However, the memory cost of this algorithm is also  $\mathcal{O}(2^{0.0176n})$ ; while this is a common behaviour of the “fastest” algorithms from the cryptography and coding theory literature, this is an unattractive feature for “real-life” implementations as (beyond a certain point) memory is much more expensive than time in existing hardware, and certainly not on par as analyses focusing on optimising time cost alone somewhat implicitly assume.

**Our contribution.** In this paper, we perform a detailed study of time-memory tradeoffs for the large-weight ternary syndrome decoding problem, in the regime relevant to Wave’s security. We use the same reduction to  $\{0,1\}$ -subset sum as Bricout *et al.*, and the tradeoffs are obtained by acting on one parameter used in the reduction and, more importantly, by carefully instantiating the resolution of the subset sum problem with memory-efficient algorithms. For this task we rely on the dissection [6] and dissection in tree [5] frameworks. One main hurdle in efficiently applying both frameworks to the syndrome decoding setting is that they are designed to *exhaustively* solve general-birthday (or subset sum-like) problems, which they do at a low (possibly constant) amortised cost. The reduction by Bricout *et al.* only requires comparatively few solutions, and providing more than necessary inevitably leads to a sub-optimal instantiation. We thus spend a significant effort in adapting both frameworks to lower the *granularity* at which they return solutions (*i.e.* the minimum number of solution that can be returned with constant amortised cost), so that only the right amount is computed. This eventually leads to attractive time-memory tradeoffs which significantly outperform the results of Bricout *et al.* when taking the cost of memory into account. We however make no attempt at accurately modeling the cost of memory *access* which we assume to be constant and only compute for our algorithms the cost of memory *storage*. A summary of our results is shown in Table 1 in the asymptotic regime where we include the product of time and memory costs as a primitive tool of comparison between different tradeoffs.

Table 1: Asymptotic exponents (in base 2) of some algorithms for solving a ternary syndrome decoding problem for a random code of length  $n$ , dimension  $0.676n$ , and syndrome weight  $0.948366n$ .

| Time       | Memory      | Time $\times$ Memory | Tradeoff  | Algorithm                          |
|------------|-------------|----------------------|-----------|------------------------------------|
| $0.0176n$  | $0.0176n$   | $0.0352n$            | $T = M$   | $k$ -tree +<br>representations [1] |
| $0.02014n$ | $0.01007n$  | $0.03021n$           | $T = M^2$ | 4,4-dissection (Section 5)         |
| $0.02256n$ | $0.007521n$ | $0.03008n$           | $T = M^3$ | 2,11-dissection (Section 5)        |
| $0.02335n$ | $0.005838n$ | $0.02919n$           | $T = M^4$ | 3,11-dissection (Section 5)        |

**Structure of the paper.** We recall some definitions and state our problem in Section 2. We then present the framework of Bricout *et al.* in a detailed and self-contained way in Section 3, while also emphasising the role played by the granularity. Section 4 recalls some classical frameworks for the generalised birthday problem and applies them (sometimes with some tweaks) to syndrome decoding, and Section 5 does the same with the more recent dissection-in-tree framework. Finally Section 6 presents numerical results applied to the most recent parameter set for Wave.

## 2 Preliminaries

### 2.1 Notation and definitions

Except specified otherwise, we assume to be in a *ternary* setting, *i.e.* with all structures defined over  $\mathbb{F}_3$ .

Vectors (resp. matrices) names are written in a bold font and in lower (resp. upper) case, for instance  $\mathbf{x}$  (resp.  $\mathbf{M}$ ); vectors are *row* vectors. The  $i^{\text{th}}$  coordinate of a vector  $\mathbf{x}$  is written  $x_i$ , and indices start from 1. The (Hamming) weight  $\text{wt}(\mathbf{x})$  of an  $n$ -dimensional vector  $\mathbf{x}$  is the size of its support, *i.e.*  $\#\{i \in \llbracket 1, n \rrbracket \mid x_i \neq 0\}$ , where  $\llbracket 1, n \rrbracket = \{1, 2, \dots, n\}$ .

A (ternary) *linear code* of length  $n$  and dimension  $k$  is a  $k$ -dimensional linear subspace of  $\mathbb{F}_3^n$ ; any code with such parameters is said to be an  $[n, k]$  linear code. A *parity-check matrix* of an  $[n, k]$  (ternary) linear code  $\mathcal{C}$  is any full-rank matrix  $\mathbf{H} \in \mathbb{F}_3^{(n-k) \times n}$  s.t.  $\mathbf{x} \in \mathcal{C} \Leftrightarrow \mathbf{x}\mathbf{H}^T = \mathbf{0}$ , where  $\mathbf{0} \in \mathbb{F}_3^{n-k}$  is the null vector.

We use  $x := y$  (resp  $x =: y$ ) to define  $x$  as being equal to  $y$  (resp.  $y$  as being equal to  $x$ ), and  $x \leftarrow \mathcal{S}$  means that  $x$  has been drawn uniformly at random from the finite set  $\mathcal{S}$ ; except specified otherwise, this drawing is supposed to be independent from any other.

We say that an algorithm  $\mathcal{A}$  returning  $S$  distinct (and *a priori* independent) outputs in time  $\mathcal{O}(ST)$  runs in *amortised time*  $\mathcal{O}(T)$ . Also, in order to simplify notation, we often drop the “ $\mathcal{O}(\cdot)$ ” when discussing the cost of algorithms. Finally, except specified otherwise, the logarithm function  $\log$  is in base 2.

## 2.2 The large-weight ternary syndrome decoding problem

We now define the *ternary syndrome decoding problem* (or “SDP” for short), which is the main problem studied in this paper. We specialise the definition to the ternary case, *i.e.* with all underlying structures with coefficients in  $\mathbb{F}_3$ , but generalizations to other fields are straightforward.

*Problem 1 (Ternary syndrome decoding problem).* Let  $\mathbf{H} \in \mathbb{F}_3^{(n-k) \times n}$  be a parity-check matrix for an  $[n, k]$  ternary linear code,  $w \in \llbracket 1, n \rrbracket$ ,  $\mathbf{s} \in \mathbb{F}_3^{n-k}$ . The *ternary syndrome decoding problem* with inputs  $\mathbf{H}$ ,  $\mathbf{s}$ ,  $w$  asks to find  $\mathbf{e} \in \mathbb{F}_3^n$  s.t.:

1.  $\mathbf{e}\mathbf{H}^T = \mathbf{s}$ ;
2.  $\text{wt}(\mathbf{e}) = w$ .

We may refer to  $\mathbf{s}$  as the target *syndrome*, and to  $\mathbf{e}$  as an *error*.

A natural variant of this problem, which we however do not consider here, is to constraint the weight of  $\mathbf{e}$  not to a single value  $w$  but only requiring that it be included in some interval.

In all of this work we only consider instances of **Problem 1** with the following additional restrictions:

1. We consider uniformly random linear codes:

$$\mathbf{H} \leftarrow \left\{ \mathbf{M} \in \mathbb{F}_3^{(n-k) \times n} \mid \text{rank}(\mathbf{M}) = n - k \right\}.$$

2. We consider uniformly random syndromes:  $\mathbf{s} \leftarrow \mathbb{F}_3^{n-k}$ .
3. The code parameters  $n$  and  $k$  and the target weight  $w$  are proportional, with the same ratios as in the “updated” parameters for the Wave signature scheme given by Bricout *et al.* [1], *viz.*  $k = 0.676n$ ,  $w = 0.948366n$ . In the following we refer to this setting as the *Wave regime* which, since  $w \approx 0.95n$  is a particular instance of a *large-weight regime*.

*Remark 2.* The Wave regime as defined above corresponds to a setting for which no efficient (in particular no polynomial-time) algorithm for solving the problem is known, yet one expects a random instance to have a number of solution exponential in the length  $n$  of the code. We refer to [4, §3] for more details on the topic and on parameter selection for Wave in general.

## 3 A framework for solving the ternary syndrome decoding problem

At SAC 2019 [1], Bricout *et al.* formalised a high-level framework to solve (hard) instances of the ternary syndrome decoding problem. They name this framework “PGE+SS”, standing for *partial Gaussian elimination + subset sum*, and its structure closely follows the one used by similar *information set decoding* (or ISD) algorithms used in the (more usual) binary setting. Since our work fully adheres to this framework we wish to give here a self-contained description of its main ideas and analysis, and refer to [1] for more details.

### 3.1 The PGE+SS framework

Let  $\mathbf{H} \in \mathbb{F}_3^{(n-k) \times n}$ ,  $\mathbf{s}$ ,  $w$  define an instance of **Problem 1**; the PGE+SS framework fixes two additional parameters  $l \in \llbracket 0, n - k \rrbracket$  and  $p \in \llbracket 0, \min(w, k + l) \rrbracket$ . One then does the following:

1. *Partial information set selection.* Pick  $\mathbf{P} \in \mathbb{F}_3^{n \times n}$  uniformly at random among the permutation matrices that are s.t. the  $n - k - l$  first columns of  $\mathbf{HP}$  are linearly independent.
2. *Partial Gaussian elimination.* Compute the reduced row-echelon form of  $\mathbf{HP}$ , stopping after the first  $n - k - l$  rows have been processed. This returns an invertible matrix  $\mathbf{S} \in \text{GL}(n - k, 3)$  s.t.:

$$\mathbf{SHP} =: \begin{pmatrix} \mathbf{I}_{n-k-l} & \mathbf{H}_1 \\ \mathbf{0} & \mathbf{H}_2 \end{pmatrix},$$

with  $\mathbf{H}_1 \in \mathbb{F}_3^{(n-k-l) \times (k+l)}$ ,  $\mathbf{H}_2 \in \mathbb{F}_3^{l \times (k+l)}$ , and further let  $\mathbf{s}' = (\mathbf{s}'_1 \ \mathbf{s}'_2) := \mathbf{s}\mathbf{S}^T$ , with  $\mathbf{s}'_1 \in \mathbb{F}_3^{n-k-l}$ ,  $\mathbf{s}'_2 \in \mathbb{F}_3^l$ .

Remark then that if  $\mathbf{e}'$  is a solution to the syndrome decoding problem instance defined by  $\mathbf{SHP}$ ,  $\mathbf{s}'$  and  $w$  then  $\mathbf{e}'\mathbf{P}^T$  is a solution to the initial instance, as from  $\mathbf{e}'\mathbf{P}^T\mathbf{H}^T\mathbf{S}^T = \mathbf{s}'$  one has  $(\mathbf{e}'\mathbf{P}^T)\mathbf{H}^T = \mathbf{s}$ .

3. *Subset sum problem resolution.* Solve the syndrome decoding problem instance defined by  $\mathbf{H}_2$ ,  $\mathbf{s}'_2$  and weight  $p$  and return  $S$  distinct solutions  $\{\mathbf{e}'_2 \in \mathbb{F}_3^{k+l}\}$ , where  $S$  is a parameter to be determined later. For large-weight ternary syndrome decoding and well-chosen parameters  $l$  and  $p$ , this in fact reduces to a  $\{0,1\}$ -subset sum problem (see [Sections 3.2](#) and [4.1](#) and [[1](#), [§2](#)] for details).
4. *Probabilistic reconstruction.* For every solution  $\mathbf{e}'_2$  returned at step [3](#) compute the unique vector  $\mathbf{e}'_1 := \mathbf{s}'_1 - \mathbf{e}'_2 \mathbf{H}_1^T$  s.t.  $(\mathbf{e}'_1 \ \mathbf{e}'_2) \mathbf{P}^T \mathbf{H}^T \mathbf{S}^T = (\mathbf{s}'_1 \ \mathbf{s}'_2)$ , and if  $\text{wt}(\mathbf{e}'_1) = w - p$  return  $(\mathbf{e}'_1 \ \mathbf{e}'_2) \mathbf{P}^T$  as a solution to the initial problem. If none of the solutions satisfied the weight constraint the algorithm fails.

*Remark 3.* Prange's algorithm [[14](#)] corresponds to the setting  $l = 0$ . In that case the subset sum problem from step [3](#) becomes trivial since the zero-dimensional  $\mathbf{s}'_2$  imposes no constraint. Yet for the same number of returned solutions  $S$  and for most target weights  $w$  the success probability of step [4](#) is in this case typically smaller than for  $l > 0$ .

We now analyse some aspects of the PGE+SS framework, but only in the regime relevant to us, *i.e.* when the target weight  $w$  is close to  $n$  (but lower than the Gilbert-Varshamov bound). In particular we only consider the case where  $p = k + l$ , that is where the solutions for the smaller syndrome decoding sub-problem at step [3](#) are required to be full-weight. This has two consequences: 1) except for very large values of  $l$ , this maximises the probability that a solution to the sub-problem extends to a solution to the initial problem in step [4](#); 2) since there are exactly two non-zero elements in  $\mathbb{F}_3$ , this sub-problem can be solved by using an algorithm for the (quite common)  $\{0,1\}$ -subset sum problem.

### 3.2 Required number of solution for the subset-sum problem

With the above constraint on the PGE+SS parameterization, the number  $S$  of returned solution to the sub-problem required for the algorithm to succeed with constant probability becomes only a function of  $n$ ,  $l$ ,  $k$  and  $w$  (or in fact only  $n$  and  $l$  inasmuch as  $k$  and  $w$  depend on  $n$  in the Wave regime): assuming independence of the solutions, it precisely needs to be proportional to the inverse probability that  $\mathbf{e}'$  as computed in step [4](#) has the right weight; we compute this probability in [Proposition 4](#), and often denote  $S_l$  its inverse in the remainder of this paper.

**Proposition 4.** *Let  $\mathbf{H}$ ,  $\mathbf{s}$ ,  $w$  define a random instance of [Problem 1](#) in the Wave regime, and  $\mathbf{H}_1$ ,  $\mathbf{H}_2$ ,  $\mathbf{s}'_1$ ,  $\mathbf{s}'_2$  be as in [Section 3.1](#). Then assuming that the syndrome decoding sub-problem defined by  $\mathbf{H}_2$ ,  $\mathbf{s}'_2$ ,  $k + l$  has many solutions, and if  $\mathbf{e}'_2$  is picked uniformly at random among them, one has:*

$$\Pr[\text{wt}(\mathbf{e}'_1) = w - k - l] \approx \frac{\binom{n-k-l}{w-k-l} 2^{w-k-l}}{3^{n-k-l}}, \quad (1)$$

where  $\mathbf{e}'_1 \in \mathbb{F}_3^{n-k-l}$  is equal to  $\mathbf{s}'_1 - \mathbf{e}'_2 \mathbf{H}_1^T$ .

*Proof.* Let  $\mathcal{P}$  be as in [Section 3.1](#) and  $\mathcal{S}$  denote the set of solutions to the main decoding problem; we have that  $\text{wt}(\mathbf{e}'_1) = w - k - l$  iff.  $(\mathbf{e}'_1 \ \mathbf{e}'_2) \mathbf{P}^T \in \mathcal{S}$ . Thus  $\Pr[\text{wt}(\mathbf{e}'_1) = w - k - l] = \Pr[\exists \mathbf{e} \in \mathcal{S}, \mathbf{e} = (* \ \mathbf{e}'_2) \mathbf{P}^T]$ , *i.e.* the probability that there is a solution with the right structure.<sup>3</sup>

To compute this probability, we first assume that the elements of  $\mathcal{S}$  are uniformly distributed among the  $2^w \binom{n}{w}$  weight- $w$  vectors of  $\mathbb{F}_3^n$ . Also, since the Wave regime is such that  $w$  is far away from the Gilbert-Varshamov bounds we approximate the expected size of  $\mathcal{S}$  by  $\mathbb{S} := 2^w \binom{n}{w} / 3^{n-k}$ . Similarly, the expected number of solution to the sub-problem is approximated by  $\mathbb{S}_2 := 2^{k+l} / 3^l$ .

Now for  $\mathbf{e} \in \mathcal{S}$  to have the right structure, two conditions must be satisfied: 1) it must have the right support, which happens with probability  $\binom{n-k-l}{w-k-l} / \binom{n}{w}$ ; 2) it must be equal to  $\mathbf{e}'_2$  on the right part, which happens with probability  $\mathbb{S}_2^{-1}$  conditioned on having the right support (since by construction this part then constitutes a solution to the sub-problem). Finally, equating the probability with the (approximated) expectancy, we get  $\Pr[\text{wt}(\mathbf{e}'_1) = w - k - l] = \mathbb{S} \times \mathbb{S}_2^{-1} \times \binom{n-k-l}{w-k-l} / \binom{n}{w}$ .  $\square$

In practice we sometimes rely in our cost computations on the same simpler asymptotic estimate for [Eq. \(1\)](#) as [[3](#), Lemma 1.2], which we provide in [Appendix A](#) for completeness.

*Remark 5.* In the Wave regime,  $S_l < 2^{k+l} / 3^l$ , the number of solutions to the sub-problem, so by properly choosing  $S$  at step [3](#) one can ensure that the algorithm succeeds w.h.p..

<sup>3</sup> Note that since  $\mathbf{e}'_1$  is fully determined by  $\mathbf{e}'_2$  there can be at most one such solution.

### 3.3 Parameterization of the subset-sum problem

The choice for the (unique) parameter  $l$  of the PGE+SS framework has a considerable influence on the final cost of solving the problem. Some of the consequences are quite obvious: if  $l$  is small, then the decoding sub-problem is easy to solve, but the required number of solution  $S_l$  is huge; similarly, if  $l$  is large one requires much fewer solutions but solving the sub-problem becomes much harder. A slightly less naïve observation is that although at first sight one is asking in step 3 to solve a problem similar to the original (*viz.* a syndrome decoding problem), the fact that *many* solutions are required (and not just one) opens the way to specific optimisations; in particular one may aim at finding these solutions at a low (ideally constant) *amortised cost*, so that the total time cost be proportional to  $S_l$ . To reach this goal one has at its disposal a full range of powerful algorithms for the subset sum problem. Yet those algorithms are not without some limitations, and their (efficient) usage is often not straightforward. We now mention two of those limitations at a high level, and explore their consequences systematically in Sections 4 and 5.

- The algorithms we consider have an intrinsic non-trivial *granularity* at which they return solutions. This is the smallest number of solutions that an algorithm may return at its nominal (usually constant) amortised cost, see Definition 6. In our case one incurs some loss in using an algorithm if its granularity is larger than the number of required solutions  $S_l$ .
- They also all have a large memory cost, sometimes equal to their granularity.

**Definition 6 (Granularity of an algorithm).** *Let  $\mathcal{A}$  be an algorithm that returns  $S$  outputs and runs in amortised time  $\mathcal{O}(T)$ . We define its granularity as the least positive integer  $S' \leq S$  s.t. there exists a tweaked algorithm  $\mathcal{A}'$  for the same problem that returns  $S'$  outputs in amortised time  $\mathcal{O}(T)$ .*

The above can be summarised as the following rough estimation for the cost of a PGE+SS instantiation in our case: a subset sum algorithm that returns  $S$  solutions in amortised constant time and with memory cost  $M$  and granularity  $S'$  can be used to solve the decoding problem with memory cost  $M$ ,<sup>4</sup> and time cost  $\max(S_l, S')$ .

## 4 Fundamental algorithms for the generalised birthday problem

### 4.1 Subset sum as a generalised birthday problem

In this section we present and compare two families of algorithms that solve the generalised birthday problem (whose definition we recall in Problem 7, in the specific case of  $\mathbb{F}_3^n$ ): the  $k$ -tree algorithm and its variants [18] and the dissection framework [6]. Both can be seen as a way to generalise the meet-in-the-middle algorithm, which we recall in Appendix B for completeness.

*Problem 7 (Generalised birthday problem or  $r$ -list problem).* Let  $L_1, \dots, L_r$  be  $r$  lists of vectors uniformly sampled from  $\mathbb{F}_3^n$  and  $\mathbf{s} \in \mathbb{F}_3^n$  be a *target*, the *generalised birthday problem* asks to find  $(\mathbf{x}_1, \dots, \mathbf{x}_r) \in L_1 \times \dots \times L_r$  s.t.  $\sum_{i=1}^r \mathbf{x}_i = \mathbf{s}$ .

Depending on the context where this problem arises, one may be content with finding one solution, or on the contrary need to obtain many of them. This latter situation typically occurs within the PGE+SS framework, where an algorithm solving Problem 7 can be used to address the sub decoding problem in the full-weight regime.

Let us hereafter denote by  $\mathbf{H} \in \mathbb{F}_3^{l \times (k+l)}$  and  $\mathbf{s} \in \mathbb{F}_3^l$  the matrix  $\mathbf{H}_2$  and vector  $\mathbf{s}'_2$  from Section 3.1 respectively. Then finding a full-weight vector  $\mathbf{e}$  s.t.  $\mathbf{e}\mathbf{H}^T = \mathbf{s}$  can be done by: 1) building  $r$  lists  $L_i = \{\mathbf{x}\mathbf{H}^T : \mathbf{x} \in \mathcal{W}_i\}$ , where the elements of the sets  $\mathcal{W}_i$  have full weight on a set of indices  $\mathcal{I}_i$  and weight zero on its complementary and  $\mathcal{I}_1, \dots, \mathcal{I}_r$  forms a partition of  $\llbracket 0, k+l \rrbracket$ ; 2) solving a generalised-birthday problem with input  $L_1, \dots, L_r$  and  $\mathbf{s}$ .

This is a classical approach in general, and it was successfully applied to ternary syndrome decoding by Bricout *et al.*, who consider a number of variants of the  $k$ -tree algorithm [1]. We recall their results and start exploring some related time-memory tradeoffs next.

<sup>4</sup> If  $S_l > M$ , one would in practice interleave steps 3 and 4 so as to avoid storing all  $S_l$  solutions at the same time.

## 4.2 Application of the $k$ -tree algorithm to syndrome decoding

From now on assume that  $r =: 2^a$  is a power of two. Recall that the basic  $k$ -tree algorithm [18] works as follows: at the first step, subtract the target  $\mathbf{s}$  to every element of  $L_r$ , then for each pair of lists  $(L_{2i-1}, L_{2i})$ ,  $i \in \llbracket 1, 2^{a-1} \rrbracket$ , compute the merged list  $L'_i := L_{2i-1} \bowtie_{\vartheta} L_{2i} := \{\mathbf{x}_u + \mathbf{x}_v : (\mathbf{x}_u, \mathbf{x}_v) \in L_{2i-1} \times L_{2i}, \mathbf{x}_u =_{\vartheta} -\mathbf{x}_v\}$ , where  $\vartheta$  is a parameter and  $\mathbf{x} =_{\vartheta} \mathbf{y}$  means that  $\mathbf{x}$  and  $\mathbf{y}$  are equal on their last  $\vartheta$  coordinates. This process is then repeated on the lists  $L'_1, \dots, L'_{2^{a-1}}$  with the equality constraint being imposed on the  $\vartheta'$  coordinates before the last  $\vartheta$  ones, etc.; after  $a$  iterations in total, and provided that  $\vartheta + \vartheta' + \dots = l = \dim(\mathbf{s})$ , all the elements of the last list (if non empty) are solutions to the problem.

In a classical and typical parameterization of the  $k$ -tree algorithm, one takes  $\vartheta = \vartheta' = \dots$  and lists of initial size equal to the ‘‘entropy’’ of a size- $\vartheta$  constraint; in our case this is  $3^{\vartheta}$ . This ensures that on average the size of all lists (except possibly the last one) remains equal to  $3^{\vartheta}$  at every level of the tree and this also gives the memory cost of the algorithm (up to a factor  $2^a$  if the lists cannot be generated on-the-fly). Then the two typical choices for  $\vartheta$  are  $l/(a+1)$  and  $l/a$ ; in the former case the expected size of the root list is 1, while it is  $3^{\vartheta} = 3^{l/a}$  in the latter. This last parameterization is of particular interest in our context since it gives an algorithm with time and memory cost  $\mathcal{O}(2^a 3^{l/a})$  that on average returns  $3^{l/a}$  solutions. The amortised cost per solution is then  $\mathcal{O}(2^a)$ , or  $\mathcal{O}(1)$  as  $a$  is in fact often a constant, and the granularity is  $3^{l/a}$ .

As we have just described it, the  $k$ -tree algorithm only returns in the root lists solutions which are highly structured which, put another way, means that it highly decimates the number of possible solutions to be found among the initial lists. Yet if more than  $3^{l/a}$  solutions are needed, two (non-exclusive) options exist: 1) restart the algorithm from new lists (if possible); 2) jointly change the merging condition for two pairs of lists at the same level, so that one merges elements s.t.  $\mathbf{x}_u =_{\vartheta} -\mathbf{x}_v + \mathbf{t}$  and the other elements s.t.  $\mathbf{x}_u =_{\vartheta} -\mathbf{x}_v - \mathbf{t}$ ; this is easily implementable by simply adding (resp. subtracting) the right  $\vartheta$  coordinates of  $\mathbf{t}$  to one of the two lists for each pair. This second option in fact lets one now exhaustively search for all the possible solutions, something we will discuss again in Section 4.3. We illustrate this and the general process of a typical  $k$ -tree instantiation in Fig. 1 for  $a = 3$ .

*Remark 8.* We defined here the  $k$ -tree algorithm with as input a number of lists which is a power of two. It is possible to adapt the algorithm to a relaxed setting without this constraint, but there is no added gain<sup>5</sup> in doing it.

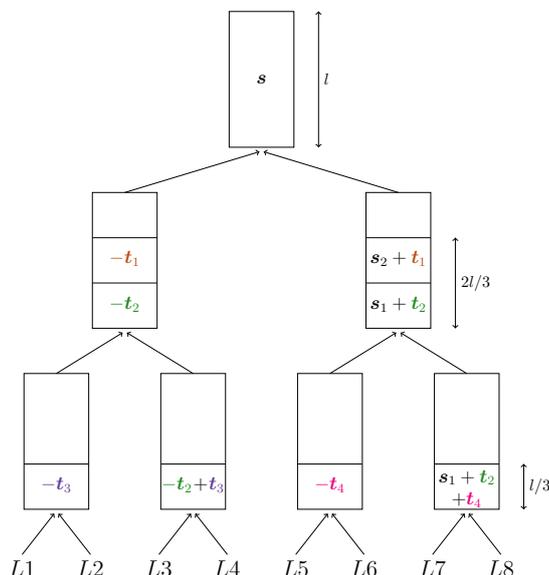


Fig. 1: Illustration of the  $k$ -tree algorithm with  $M = 3^{l/3}$ . For only one iteration of the tree, the targets  $\mathbf{t}_i$  are all set to  $\mathbf{0}$ . For more than one iteration, the targets  $\mathbf{t}_i$  must be set to non-zero values, and every distinct tuple of  $\mathbf{t}_i$ ’s provides disjoint solutions.

Bricout *et al.* use the  $k$ -tree algorithm within the PGE+SS framework to solve hard instances of the ternary syndrome decoding problem [1]. In a basic application, the only additional con-

<sup>5</sup> In the next part, the gain is formally defined in Definition 10.

straint to what has already been described above is that for a fixed  $l$  parameter, the depth of the tree must be s.t. it is possible to build lists of size  $3^{l/a}$  at its leaves. When elements of those initial lists are of the form  $\{\mathbf{x}\mathbf{H}^T : \mathbf{x} \in \mathcal{W}_i\}$ , this list population constraint can be expressed as:

$$3^{l/a} \leq 2^{(k+l)/2^a}. \quad (2)$$

This simply expresses the fact that there are  $2^a$  lists of full-weight vectors to build at the leaves of the tree and one must then split the support of the domain  $\mathbb{F}_3^{k+l}$  into that many equally-sized disjoint sets.

For a fixed  $l$  parameter, the memory (and the granularity) of this application of the  $k$ -tree algorithm is minimised by simply selecting the largest  $a$  for which this constraint is satisfied.

*Smoothed  $k$ -tree algorithm.* *Smoothing* the  $k$ -tree algorithm is a technique that allows to slightly relax constraint (2) by adding one more level to the tree than what it dictates. This corresponds to the *extended  $k$ -tree algorithm* of Minder and Sinclair [11, Theorem 3.1], and it was adapted to the ternary case under this name by Bricout *et al.* [1].

In a nutshell the idea is the following: if one cannot build initial lists that are large enough, the constraint size  $\vartheta$  from the level 1 lists to level 2 is lowered so as to increase the size of the latter; then this increased (expected) list size is preserved all the way up to the root of the tree. Schematically this translates into a sequence of constraints sizes  $\vartheta < \vartheta' = \vartheta'' = \dots$  which sum to  $l$ ; the memory cost is then equal to  $3^{\vartheta'}$ , which is more than if one had had constraints of identical sizes, *i.e.* one has to “pay” for the dissatisfaction of Eq. (2) with memory. Nonetheless, in the case of the SDP, adding one more level to the tree to apply the smoothing technique is always more beneficial, even if this is done at a less favourable time/memory ratio. We summarise the consequences of smoothing as Proposition 9, which restates [1, Prop. 4].

**Proposition 9.** *Let  $l, k, n$  be as above and  $a > 3$  be a constant. If  $3^{l/(a-1)} \leq 2^{(k+l)/2^{a-1}}$ , then one can use a smoothed  $k$ -tree algorithm with  $a$  levels to obtain  $2^m$  solutions to the generalised birthday problem in amortised constant time and memory cost  $2^m$ , where:*

$$m = \frac{1}{a-2} \left( l \log(3) - \frac{k+l}{2^{a-1}} \right).$$

*Proof.* We only prove this informally without showing optimality nor checking initial conditions, our main goal here being to illustrate the inner workings of smoothing.

Let  $\varsigma := l \log(3)$  normalise in base 2 the size of the dimension- $l$  ternary constraint that one wishes to solve and  $\tau := (k+l)/2^a$  be the logarithm of the maximum size of  $2^a$  lists of full-weight vectors partitioning the domain. We wish to find initial and subsequent constraints  $\vartheta$  and  $\vartheta' = \vartheta'' = \dots$  s.t.: 1)  $\vartheta' = 2\tau - \vartheta$ ; 2)  $\vartheta + (a-1)\vartheta' = \varsigma$ . The first condition expresses the fact that the constraint of size  $\vartheta$  ensures that the first level lists merge into lists of expected size  $2^{\vartheta'}$ ; the structure of the  $k$ -tree algorithm together with the second condition then ensure the fact that the root list contains  $2^{\vartheta'}$  solutions to the problem, and since  $\vartheta < \vartheta'$  that the algorithm runs in amortised constant time and with memory  $2^m := 2^{\vartheta'}$ .

To find the stated value of  $m$ , one simply substitutes  $2\tau - \vartheta$  for  $\vartheta'$  into  $\vartheta + (a-1)\vartheta' = \varsigma$  and solves the latter for  $\vartheta$ , *i.e.*:

$$\begin{aligned} \vartheta + (a-1)(2\tau - \vartheta) &= \varsigma \\ \Leftrightarrow (a-1)2\tau - (a-2)\vartheta &= \varsigma \\ \Leftrightarrow \vartheta &= ((a-1)2\tau - \varsigma)/(a-2) \end{aligned}$$

Using again  $\vartheta' = 2\tau - \vartheta$  one then gets:

$$\begin{aligned} \vartheta' &= [2(a-2)\tau - ((a-1)2\tau - \varsigma)]/(a-2) \\ \Leftrightarrow \vartheta' &= (\varsigma - 2\tau)/(a-2) = (l \log(3) - (k+l)/2^{a-1})/(a-2). \end{aligned}$$

□

*Using representations.* Bricout *et al.* obtained their best result in the Wave regime by applying the so-called representation technique [9] to their ternary  $k$ -tree algorithm, slightly beating their instantiations that used smoothing. We do not detail this approach since we do not consider it in our work, and refer to [1] for details. When optimised for time, this uses a tree with  $a = 7$  levels and parameter  $l = 0.060835n$  and solves the decoding problem in asymptotic time and memory  $\mathcal{O}(2^{0.0176n})$ .

**Time-memory tradeoffs from the  $k$ -tree algorithm.** Recall that within the PGE+SS framework, solving the sub decoding problem for parameter  $l$  in amortised constant time with granularity and memory cost  $S'$  allows to solve the initial problem with memory cost  $S'$  and time cost  $\max(S_l, S')$ . Since the (smoothed)  $k$ -tree algorithm may in principle be used for any  $l$  one then naturally obtains a time-memory tradeoff by varying this parameter and using the best variant of the  $k$ -tree algorithm to solve the derived sub-problem. Choosing this variant is a rather straightforward consequence of what has been presented above and we give pseudocode for this parameter selection in [Appendix E](#) for both “standard” and smoothed  $k$ -tree algorithms as [Algorithms 1](#) and [2](#) respectively. We show the resulting time-memory tradeoff curves in [Fig. 2](#), where we also include the best attack of Bricout *et al.* as a point of comparison. One may notice there the natural discontinuity exhibited by the standard  $k$ -tree algorithm and the fact that the smoothed variant is indeed always superior. The near monotonicity of the curves is consequence of the fact that the granularity of the  $k$ -tree algorithm is low and thence does not limit the performance of the algorithm, except for the relatively large  $l$  parameters used to draw the bottom right part of the graph.

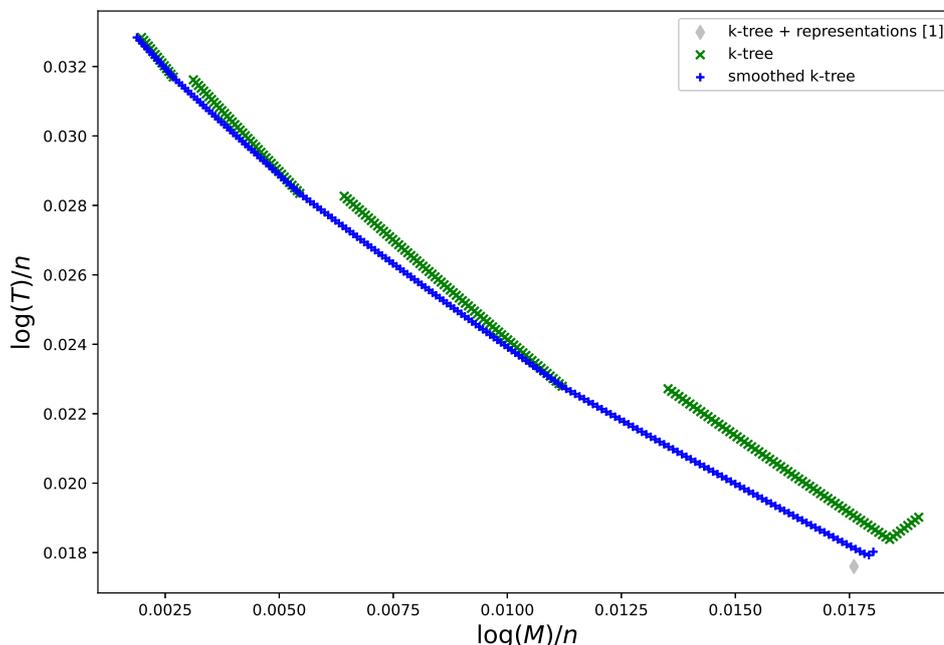


Fig. 2: Time-Memory tradeoffs from the (smoothed)  $k$ -tree algorithm.

### 4.3 Solving generalised birthday problems with dissection

The *dissection* framework was introduced by Dinur *et al.* at CRYPTO 2012 to solve “composite” problems in a memory-efficient way [6]. The main initial motivation was provided by the generic key recovery attack of iterated block ciphers, but the framework adapts easily to an  $r$ -list problem and was already used in this context by Esser *et al.* and Dinur [8,5], who also study it in some non-exhaustive regimes. Dissection generalises the meet-in-the-middle algorithm in a different way than the  $k$ -tree algorithm (with both techniques also being refinements of [15]). Its main originality is that instead of merging lists along a (typically) balanced binary tree, it uses a recursive asymmetric decomposition; the solutions of the smaller sub-problem resulting from this decomposition are stored in memory and combined with solutions for the larger problem that are generated on-the-fly. Altogether, this asymmetric decomposition and the structure of the algorithm make dissection a memory-friendly family of algorithms.

An important notion to quantitatively analyse dissection algorithms (and algorithms for the  $r$ -list problem in general) is the *gain* [6], which we state in [Definition 10](#) in our specific ternary

case. In there, and in all of the following, we let by definition the size of the  $r$  initial lists of **Problem 7** be equal to  $3^m$ , and often treat  $m$  as a parameter.

**Definition 10.** Let  $\mathcal{A}$  be an algorithm that solves **Problem 7** with  $r$  lists in  $\mathbb{F}_3$  in time  $\mathcal{O}(3^{mT})$  and memory  $\mathcal{O}(3^{mM})$  with  $m \in \mathbb{R}$ . Then its gain is defined as  $\text{gain}(\mathcal{A}) := r - (T + M)$ .

This should be understood as a gain *over* the time-memory tradeoff offered by the meet-in-the-middle algorithm, which always has gain 0. Any positive gain then gives a better tradeoff than the latter. Hereafter we use  $\text{gain}(r)$  to denote the gain of an  $r$ -dissection that solves an  $r$ -list problem (or sometimes simply  $g$ , when  $r$  is clear from the context).

We now illustrate the dissection framework with two examples and give a general description in **Appendix C**.

*Example 11 (4-dissection).* The 4-dissection is simply the exhaustive variant of the  $k$ -tree algorithm with two levels as described in **Section 4.2**, and it was in fact well-known before the general formulation of the dissection framework. Unlike instantiations with a larger number of lists, it also uses a symmetric decomposition. Starting from four lists  $L_{1,\dots,4}$  of size  $3^m$  and with a target  $\mathbf{s}$  of dimension  $l := 2m$ , one introduces an intermediate target  $\mathbf{t}$  of dimension  $m$ . Then for each value of  $\mathbf{t}$ , one applies the  $k$ -tree algorithm to  $L_1 + \mathbf{t} := \{\mathbf{x} + (0 \ \mathbf{t}) : \mathbf{x} \in L_1\}$ ,  $L_2$ ,  $L_3 - \mathbf{t}$ ,  $L_4 - \mathbf{s}$ , obtaining as a result a list of solutions with a unique structure (*viz.*  $\mathbf{x}_{1,\dots,4}$  s.t.  $\mathbf{x}_1 + \mathbf{x}_2 =_m -\mathbf{t}$ ,  $\mathbf{x}_3 + \mathbf{x}_4 =_m \mathbf{s} + \mathbf{t}$ ), and enumerating all values for  $\mathbf{t}$  yields all the solutions to be found within  $L_{1,\dots,4}$ . The memory cost and the granularity is  $3^m$  and the time cost  $3^{2m}$ , also equal to the expected number of returned solutions. The product of the time and memory cost is then  $3^{3m}$ , which is a factor  $3^m$  less than what one would get from meet-in-the-middle algorithms, hence the gain is equal to 1.

*Example 12 (7-dissection).* The 7-dissection is the first instantiation of the framework with gain 2. It groups its 7 input lists into a group of three (resp. four) lists, for which a meet-in-the-middle algorithm (resp. 4-dissection) will be used. Let again  $3^m$  be the size of the initial lists, and  $\mathbf{t}_1$  and  $\mathbf{t}_2$  be as in **Fig. 3**, which also illustrates the structure of the algorithm; to solve a 7-list problem for a target  $\mathbf{s}$  of size  $3m$  one does the following for all values of  $\mathbf{t}_1$  and  $\mathbf{t}_2$ :

1. Exhaustively search for all solutions to a 3-list problem for the  $2m$  target  $(\mathbf{s}_2 - \mathbf{t}_2 \ \mathbf{s}_1 - \mathbf{t}_1)$ , using a meet-in-the-middle algorithm with memory (resp. time) cost  $\mathcal{O}(3^m)$  (resp.  $\mathcal{O}(3^{2m})$ ), and store all solutions in a list  $L'$ .
2. Exhaustively search for all solutions to a 4-list problem for the  $2m$  target  $(\mathbf{t}_2 \ \mathbf{t}_1)$ , using 4-dissection with memory and granularity (resp. time) cost  $\mathcal{O}(3^m)$  (resp.  $\mathcal{O}(3^{2m})$ ), and for every returned solution  $\mathbf{x} = (* \ \mathbf{t}_2 \ \mathbf{t}_1)$  check on-the-fly if there is  $\mathbf{x}' \in L'$  s.t. they sum to  $\mathbf{s}$ .

The total memory cost is  $\mathcal{O}(3^m)$ , the time cost and number of returned solutions is  $\mathcal{O}(3^{4m})$ , and the granularity is given by the size of the intermediate target  $(\mathbf{t}_2 \ \mathbf{t}_1)$  for the 4-dissection and thence  $3^{2m}$ .

In general an  $r$ -dissection of gain  $g$  is built from an  $(r - g - 1)$ -dissection and a meet-in-the-middle algorithm with  $g + 1$  lists. This leads to a *magic sequence*  $(M_n)$  of gains [6], where  $M_g$  is the least  $r$  s.t. there is an  $r$ -dissection with gain  $g$ . Dinur *et al.* showed that  $M_g = \frac{(g+1)(g+2)}{2} + 1 \approx g^2/2$ , leading to the following approximation:

$$\text{gain}(r) \approx \sqrt{2r} \tag{3}$$

One may also characterise an  $r$ -dissection with gain  $g$  from the fact that it returns all the  $3^{m(r-g-1)}$  solutions to an  $r$ -list problem with target size  $(g + 1)m$  in amortised constant time and with memory cost  $\mathcal{O}(3^m)$ . Since in this case the intermediate target  $\mathbf{t}$  used in the recursion is of size  $gm$ , it also follows that in this regime the granularity of the dissection is at most  $3^{m(r-g-1)}/3^{gm} = 3^{m(r-2g-1)}$ . Remark that it is also straightforward to exhaustively solve for target sizes smaller than  $(g + 1)m$  by running many times a dissection with dummy targets of the latter size.

The above description concerns dissection with a memory cost equal to the size of the initial lists, but the framework can be easily extended to use more memory [6]. For any integer  $\mu > 1$ , one increases the number  $\mu_r$  of lists in the meet-in-the-middle step and returns a list  $L'$  of partial solutions of size  $3^{\mu m}$  while also allowing the recursive dissection to have memory cost  $3^{\mu m}$ . Denoting  $\text{gain}(r, \mu)$  the gain of such an  $r$ -dissection with memory parameter  $\mu$ , one has the relation  $\mu_r = \text{gain}(r, \mu) + \mu$ . A convenient consequence of generalising dissection in this way

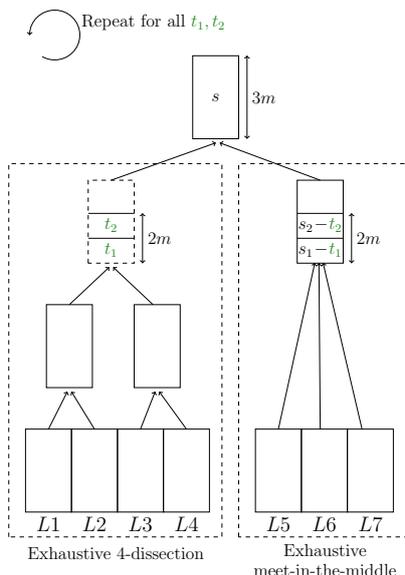


Fig. 3: 7-dissection with initial lists of size  $3^m$  and a target  $s$  of size  $3m$ . A list drawn with dashed lines is not stored in memory and processed on-the-fly.

is that in some sense an  $r$ -dissection with  $\mu = 1$  and  $m = n$  is equivalent to an  $rN$ -dissection with  $\mu = N$  and  $m = n/N$ , where  $N \geq 1$  is an arbitrary integer. In our context where we have considerable freedom in the choice for the initial number of lists, this remark simplifies the search for good parameterization of the dissection to solve the problem at hand. Indeed, considering one  $r$ -dissection with  $r$  large and allowing  $\mu$  to vary is enough to reasonably represent all tradeoffs offered by the dissection framework, as we do in Fig. 4. In the following description we however let  $\mu = 1$  unless mentioned otherwise.

#### 4.4 Application of the dissection framework to syndrome decoding

Since the dissection framework can be used to solve an  $r$ -list problem, it readily applies to the full-weight sub decoding problem encountered in the PGE+SS framework, in exactly the same way as the  $k$ -tree algorithm does. In principle this provides a range of memory-efficient tradeoffs to solve the (full) decoding problem, yet the main hurdle in a straightforward application of dissection to this context is that its granularity is quite coarse; in particular it is coarser than the one of the  $k$ -tree algorithm. In this section we slightly adapt the dissection to decrease its granularity and make it more easily applicable to the PGE+SS framework. We then compare the results with instantiations based on the (smoothed)  $k$ -tree algorithm in the next Section 4.5.

Let  $n, l, r, m$  be as above; a straightforward adaptation of Eq. (2) to the use of dissection is:

$$3^m \leq 2^{(k+l)/r}. \quad (4)$$

By design, an  $r$ -dissection with gain  $g$  returns solutions to an  $r$ -list problem with target size (at most)  $(g+1)m$  in amortised constant time. If used within the PGE+SS framework, one thus ideally requires that  $l \leq (g+1)m$ . When the size  $l$  of the sub-problem increases, and since  $g$  increases monotonically with  $r$  one may require to increase  $r$  at some point in order to remain in the same regime. Yet since  $S_l$  decreases with  $l$  while the granularity of the dissection increases with  $r$ , this eventually results in unattractive instantiations of the PGE+SS framework where more solutions to the sub-problem are returned than needed. Essentially this quick analysis hints at the fact that the dissection framework is mostly useful in the low-memory regime implied by small values of  $l$ .

**Improving the granularity of the dissection.** Recall that at a high level, the granularity of an  $r$ -dissection with gain  $g$  in the amortised constant time regime is equal to  $3^{m(r-2g-1)}$ .

One may first remark that since such a dissection recursively decomposes into an  $(r-1-g)$ -dissection and a meet-in-the-middle algorithm on  $g+1$  lists, and since the solutions returned by the former are processed on-the-fly, one may possibly reduce the granularity by asking the

former to return fewer solutions (*i.e.* not to be exhaustive in its resolution of the recursive sub-problem). However this will only decrease the granularity if the lowered cost of this non-exhaustive dissection does not become smaller than the one of the meet-in-the-middle, which otherwise would dominate the running time. Yet if this condition is not met one may replace this meet-in-the-middle algorithm by a  $(g + 1)$ -dissection to do the exact same work at a lower cost,<sup>6</sup> as already considered by Dinur [5]. We illustrate this in [Example 13](#) and generalise the process in [Proposition 14](#).

*Example 13 (11-dissection with lowered granularity).* An 11-dissection has gain 3 and is composed of a 7-dissection and a meet-in-the-middle algorithm. In the amortised constant time regime the 7-dissection has granularity at most  $3^{2m}$  but the meet-in-the-middle with 4 lists and memory  $3^m$  has time cost  $3^{3m}$ , so the granularity of the 11-dissection for a target of size  $4m$  is given by the latter and equal to  $3^{3m}$ . Even though this is already smaller than what one would obtain by asking the 7-dissection to exhaustively return the  $3^{4m}$  solutions to its sub-problem of size  $3m$ , it is possible to do better: since a 4-dissection has gain 1, using one instead of a meet-in-the-middle algorithm lets one building the list  $L'$  in time  $3^{2m}$ , thus lowering the overall granularity to  $3^{2m}$ .

**Proposition 14.** *The granularity of an  $r$ -dissection with gain  $g$ , initial lists size  $3^m$  and target size at most  $m(g + 1)$  is  $3^{m(g - gain(g+1))}$ .*

*Proof.* It is enough to prove the statement for target sizes exactly  $m(g + 1)$ , since lower sizes can then be accommodated for by considering one or more larger dummy targets.

We prove this by induction on the gain  $g$ .

The base case  $g = 1$  corresponds to a 4-, 5- or 6-dissection. We have already seen in [Example 11](#) that the granularity of the 4-dissection is  $3^m$ . The 5- and 6-dissection just add one or two additional lists to a 4-dissection and thus cannot have a lower granularity. The fine-granularity version of 5 and 6-dissections are performed using a slightly tweaked 4-dissection, let us describe it:

- Take  $x_5 \in L_5$  (resp.  $x_5 \in L_5, x_6 \in L_6$ ).
- Replace the target  $s$  by  $s - x_5$  (resp.  $s - x_5 - x_6$ ).
- Run the fine-granularity version of the 4-dissection with  $L_1, L_2, L_3, L_4$  and with the full target, and append to the solution  $x_5$  (resp.  $x_5 || x_6$ ).

Therefore, the three algorithms are able to return  $3^m$  solutions in amortised constant time. Thus for  $r \in \{4, 5, 6\}$ , the induction property holds.

We now assume that the property holds for any dissection of gain  $g - 1 \geq 1$ , and will prove it for any dissection of gain  $g$ . Consider an  $r$ -dissection of gain  $g$ ; by construction it is built from an  $(r - g - 1)$ -dissection of gain  $g - 1$  and (with our tweak) an exhaustive  $(g + 1)$ -dissection. The time cost of the  $(g + 1)$ -dissection is  $\mathcal{O}(3^{m(g - gain(g+1))})$  while it returns  $3^m$  intermediate solutions, and by induction the granularity of the  $(r - g - 1)$ -dissection is  $3^{m(g - 1 - gain(g))}$ . Since  $1 + gain(g) \geq gain(g + 1)$ , the latter dissection is more fine-grained than the former; it is then possible to ask the  $(r - g - 1)$ -dissection to return only  $3^{m(g - gain(g+1))}$  solutions with a target size of  $mg$  in amortised constant time. The remaining target size required to merge the solutions of the two sub dissections into solutions of the main one being  $m$ , one expects to find  $3^{m(g - gain(g+1))}$  of them and so the  $r$ -dissection is able to provide that many solutions in amortised constant time.  $\square$

Despite the improvement provided by [Proposition 14](#), the granularity of the dissection remains too high in our context for many values of  $l$ , as shown in [Example 15](#).

*Example 15.* Let  $l = 0.04n$ , one has  $S_l \approx 3^{0.0148n}$ . Solving the derived  $r$ -list problem using dissection in the amortised constant time regime and with minimum memory gives the constraint  $m = l/(g + 1)$  and the granularity is thus  $3^{m(g - gain(g+1))} = 3^{l(g - gain(g+1))/(g+1)}$ ; this latter quantity is lower-bounded by  $3^{0.02n}$  for any  $g$  and therefore no suitable dissection has a granularity less than  $S_l$ . This fact is illustrated in [Fig. 4](#) where no instantiation reaches the grey line representing a time cost of  $S_l$ .

<sup>6</sup> Remark that there would be no point in doing this in an exhaustive dissection since in that case the cost of the (exhaustive)  $(r - 1 - g)$ -dissection is always higher than the one of the meet-in-the-middle.

We conclude by proposing another tweak to the dissection framework to further reduce its granularity. Recall that we let  $\mu = 1$  for simplicity, but the process generalises to other values in the same way as the original dissection. Let us again consider an  $r$ -dissection of gain  $g$  with initial lists of size  $3^m$  and denote by  $3^s$  the desired number of solution. Assume that  $s < m(g - \mathit{gain}(g + 1))$ , so that the granularity guaranteed by [Proposition 14](#) is too high. The idea here is to reduce the dominating time cost of the exhaustive  $(g + 1)$ -dissection by asking for fewer solutions, which mechanically means that the number of solutions that need to be returned by the  $(r - g - 1)$ -dissection has to be increased. In some sense this consists in balancing the cost of the two sub-problems in the (typically highly asymmetric) dissection, and thus making it somewhat closer to a  $k$ -tree algorithm. A possible explanation as to why this eventually leads to better results is that when only a very small fraction of the total number of solutions is required, more symmetric algorithms (one of whose drawbacks is that they highly decimate the solution space) tend to perform better. More formally one asks for  $3^{s+c}$  solutions in the  $(r - g - 1)$ -dissection and  $3^{m-c}$  in the  $(g + 1)$ -dissection for some  $c \in \mathbb{R}$ , and the overall time cost is minimised under the equality constraint:

$$s + c = m(g - \mathit{gain}(g + 1)) - c,$$

which gives:

$$c = \frac{m(g - \mathit{gain}(g + 1)) - s}{2}$$

One must also satisfy the ‘‘granularity constraint’’ given by the  $(r - g - 1)$ -dissection, *viz.*:

$$s + c \geq m(g - 1 - \mathit{gain}(g)).$$

There are then two possibilities:

$$\begin{cases} s \geq m(g - \mathit{gain}(g + 1)) & : \mathit{gain}(g + 1) = \mathit{gain}(g) + 1 \\ s \geq m(g - 2 - \mathit{gain}(g + 1)) & : \mathit{gain}(g + 1) = \mathit{gain}(g) \end{cases}$$

As it was initially assumed that  $s < m(g - \mathit{gain}(g + 1))$ , this technique is thus only useful if  $\mathit{gain}(g + 1) = \mathit{gain}(g)$ . In that case and under the above conditions, one can check that the granularity constraint of the  $(g + 1)$ -dissection is satisfied and so the overall time cost is given by  $\mathcal{O}(3^{s+c}) = \mathcal{O}(3^{(m(g - \mathit{gain}(g + 1)) + s)/2})$ . This is simply the middle point (in the exponent) between the granularity of the original dissection and the number of required solutions. Here the solutions are not obtained in amortised constant time any more, but one does not ‘‘waste’’ any in the sense that only the desired number is returned.

In [Fig. 4](#), the time-memory tradeoffs obtained by using this modified dissection are drawn in black.

Finally one may somewhat further extend the above by using a  $u$ -dissection instead of a  $(g + 1)$ -dissection for some parameter  $u$ , further balancing the cost of the two sub dissections. This does not provide an added gain from the above but allows a finer control of the time/memory ratio.

**Results.** We illustrate the time-memory tradeoffs offered by the dissection to solve the ternary syndrome decoding problem in the Wave regime in [Fig. 4](#). For simplicity, this graph illustrates the tradeoffs obtained using only a fixed (sub-optimal) value of  $l = 0.04n$ ; the best tradeoffs, all using  $l < 0.034n$ , are shown in [Fig. 5](#) in the next [Section 4.5](#). The figure was obtained by using the parameter selection [Algorithm 3](#) in [Appendix E](#), an implementation of which being available at [https://github.com/charlotte-lefevre/TM\\_tradeoffs\\_SDP](https://github.com/charlotte-lefevre/TM_tradeoffs_SDP). All the results come from a single 400-dissection which, as remarked previously, allows to implicitly consider many dissections with fewer initial lists by simply varying  $\mu$ . We do not consider  $r$ -dissections with  $r > 400$ , since it would only improve the tradeoffs  $T \approx M^m$  with  $m > 20$ . The figure reads as follows: each line represents a different value for  $\mu$  in ascending order from left to right, and each point on a line represents a different value for  $m$ , the  $\log_3$  of initial lists size. The additional tradeoffs obtained with the last proposed tweak to improve its granularity are singled out as black crosses, and provide here the best results.

Finally a notable aspect of the results shown in [Fig. 4](#) (which also applies to [Fig. 5](#)) is that there is very little interest in increasing the memory allocated to the dissection beyond a certain point.

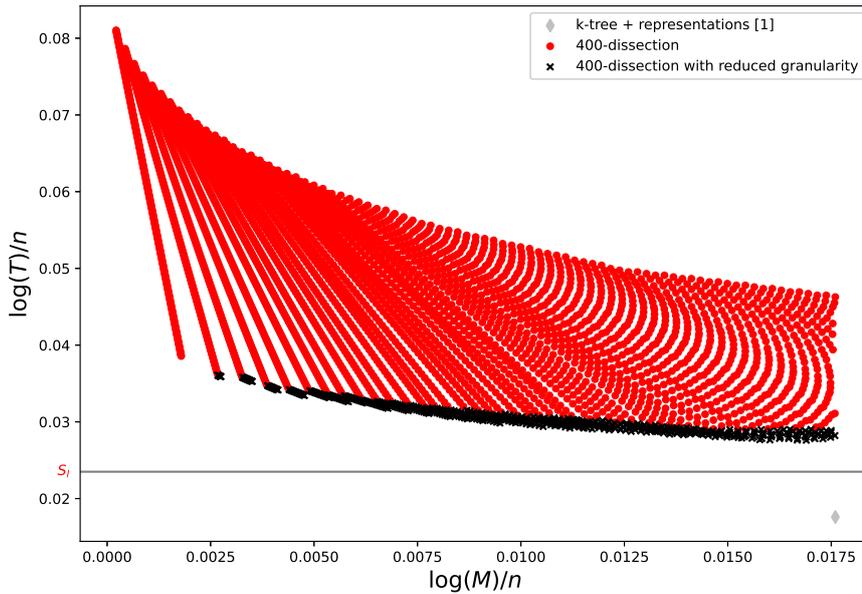


Fig. 4: Time-memory tradeoffs offered by the use of a 400-dissection within the PGE+SS framework with  $l = 0.04n$ . The grey line represents the desired number of solution  $S_l$  for this particular  $l$ .

#### 4.5 Comparison of the $k$ -tree & dissection frameworks

The  $k$ -tree and dissection frameworks may both be used to solve the same  $r$ -list problems. In this short section we wish to compare these two options and show in which regimes they respectively perform better. We again let by definition  $3^m$  be the initial lists size.

We start with an example, comparing a 16-dissection with a 16-tree algorithm. The 16-dissection of gain  $g = 4$  is split into an 11-dissection and a 5-dissection, with a total recursion depth equal to 5; the maximal target size for which this dissection may provide solutions in amortised constant time is thus equal to  $5m = (g + 1)m$ . The 16-tree algorithm has a total number of levels equal to 4, and the maximal target size for which it may provide solutions in amortised constant time is  $4m$ . Now considering a full-weight sub syndrome decoding problem of target size  $l$ , setting  $m$  to  $l/5$  (resp.  $l/4$ ) minimises the memory cost and granularity of the 16-dissection (resp. 16-tree algorithm) while allowing to find solutions in amortised constant time. In this case the dissection's granularity is  $3^{3l/5}$  while the one of the  $k$ -tree is  $3^{l/4}$ . It is thus mostly beneficial to use the more memory-efficient 16-dissection over a 16-tree algorithm when  $S_l \geq 3^{3l/5}$ , which asymptotically holds for  $l \lesssim 0.028n$ , while the granularity of the 16-tree itself will not be a limiting factor until the much larger value of  $l \approx 0.051n$ ; since  $S_l$  is decreasing with increasing  $l$  in this range, it means that a 16-tree is able to reach a lower time cost than a 16-dissection, but with a comparably higher memory cost.

More generally, we may compare a  $2^a$ -tree with a  $2^a$ -dissection: from Eq. (3) the gain of the dissection is approximately  $2^{(a+1)/2}$ , and it follows from Proposition 14 that its granularity is approximately  $3^{m2^{a/2}}$ , which is to be compared with the much lower  $3^m$  for the  $k$ -tree algorithm. The maximum target size for which the dissection may provide solutions in amortised constant time is then  $\approx m2^{(a+1)/2}$ , much larger than the  $k$ -tree algorithm at  $am$ . One may then again remark that the dissection is much more memory-efficient than the  $k$ -tree algorithm as it allows to return exponentially-more solutions in amortised constant time with the same memory usage, but that its efficient usage may be limited by an exponentially-larger granularity.

We conclude this comparison by plotting in Fig. 5 the best time-memory tradeoffs we obtained by applying the dissection &  $k$ -tree frameworks to ternary syndrome decoding in the Wave regime. In consistency with the above analysis, dissection performs significantly better than the  $k$ -tree algorithm in the low-memory regime where the total memory cost  $M \lesssim 3^{0.0073n}$ ; there is also little interest in using memory larger than  $\approx 3^{0.0025n}$  since doing so only very moderately decreases the time cost. All of those points correspond to small values for the  $l$  parameter for

which the dissection granularity matches the large number of required solutions. In the large-memory regime the dissection loses its interest and it becomes significantly outperformed by the  $k$ -tree algorithm whose fine granularity is not limiting until much larger values of  $l$ .

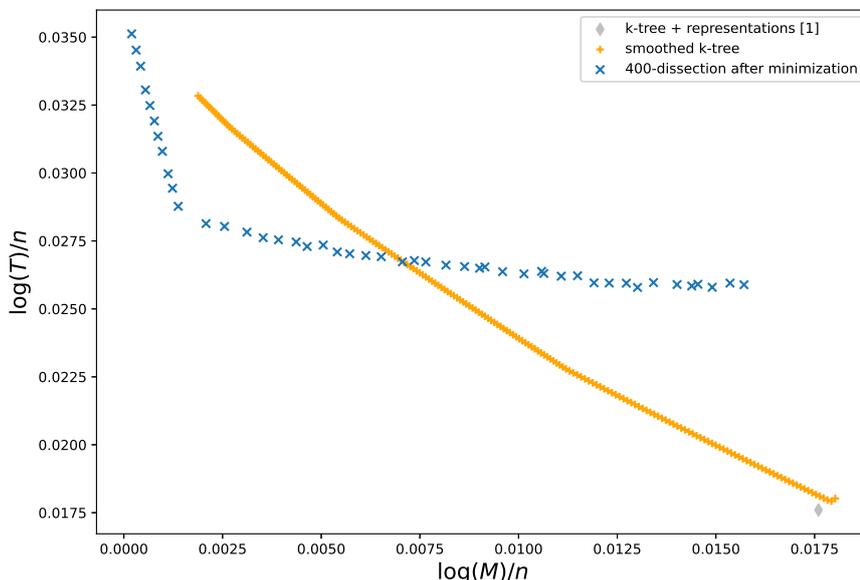


Fig. 5: Time-memory tradeoffs for the ternary syndrome decoding problem in the Wave regime from the  $k$ -tree & dissection frameworks. The results for the dissection are the best tradeoffs obtained from a 400-dissection after minimisation with  $l$ ,  $\mu$  and  $m$ .

## 5 Dissection in tree for syndrome decoding

We now present the “hybrid” *Multiple-Layer List Sum Algorithms* (which we will call “dissection in tree” for short) introduced by Dinur as a framework to solve generic generalised birthday problems [5], and apply it to ternary syndrome decoding. Similarly to the algorithms of the previous section, fully exploiting the framework in our particular case requires careful parameter selection and some modifications in particular to improve the granularity.

### 5.1 The main algorithm of dissection in tree

The idea behind dissection in tree is in fact quite straightforward: it consists in replacing the binary tree structure underlying the  $k$ -tree algorithm with an  $n$ -ary one and using (typically exhaustive)  $n$ -dissection to implement the merging of lists at each level. Similarly as in the  $k$ -tree framework, the merging is usually done w.r.t. targets whose sizes ensure that the expected list size is maintained constant through the tree, except possibly at the root level.

We first illustrate this in our case with a tree of three levels of 4-dissection, which we denote as a *3,4-dissection*; Fig. 6 shows the structure of the resulting tree. This instance provides some of the best tradeoffs we were able to obtain for syndrome decoding in the Wave regime.

Since in this case the number of leaves is equal to  $4^3 = 64$ , and again letting  $3^m$  denote by definition the cardinal of the lists, an immediate adaptation of the constraint from Eq. (2) gives here:

$$3^m \leq 2^{\frac{k+l}{64}} \quad (5)$$

To keep a constant expected size for the lists of the first two levels, the target size is set to  $3m$ . At the last level, the remaining target size is equal to  $l - 6m$ , where  $l$  again denotes

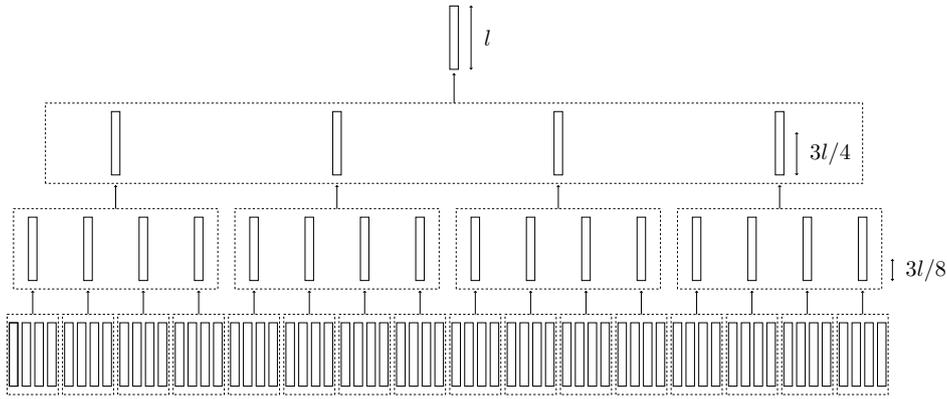


Fig. 6: Illustration of 4-dissection with three levels and  $M = 3^{l/8}$ .

the total target size. The expected number of returned solutions  $S$  for a thusly parameterised 3,4-dissection is then given by:

$$S = \frac{3^{4m}}{3^{l-6m}} = \frac{3^{10m}}{3^l}$$

Since the time cost of an exhaustive 4-dissection is  $\mathcal{O}(3^{2m})$ , one obtains the following constraint for the solutions to be returned in amortised constant time:

$$\frac{3^{10m}}{3^l} \geq 3^{2m} \quad \therefore m \geq \frac{l}{8},$$

and one would typically use the minimal admissible value  $m = \frac{l}{8}$ .

*Comparison with a 64-tree.* It is quite relevant to compare the performance of 3,4-dissection and a  $k$ -tree instance with 6 levels, since both instantiations have a similar structure and use the same number of lists. When applied to syndrome decoding and even without specific adaptation, the 3,4-dissection performs systematically better: as shown above, it is able to provide solutions to the sub decoding problem for a target of size  $l$  in amortised constant time with memory cost  $\mathcal{O}(3^{l/8})$ , while the 64-tree requires a memory of size  $\mathcal{O}(3^{l/6})$  to achieve the same. Informally one effect at play here is that using a dissection allows to find solutions that are less structured compared to a  $k$ -tree algorithm, and one thus does not require to increase the memory as much as the latter does to compensate for a high decimation of the solution space. One beneficial effect of a lower memory consumption is then that it leads to a wider range of target sizes: the constraint from Eq. (5) is “easier” to satisfy than Eq. (2), thus allowing for lower time cost for identical memory costs. There is however one downside to using dissection in tree: the granularity of  $3^{l/4} = 3^{2m} = 3^{2(l/8)}$  of the 3,4-dissection is coarser than the  $3^{l/6}$  of the 64-tree, which can be explained from the use of inherently coarser dissections to perform the merging. While this never makes 3,4-dissection “worse” than a 64-tree, it does prevent exploiting its full potential.

We summarise this comparison in Fig. 7, which plots the best time-memory tradeoffs obtained from raw 3,4-dissection and 64-tree and various values of  $l$  (shown in false colour). Two regimes are clearly observable for the 3,4-dissection whose coarse granularity makes it returning too many solutions for somewhat large values of  $l$ .

The analysis of a general raw  $h, r$ -dissection tree is a straightforward extension of the above example for the 3,4-dissection.

Letting again  $3^m$  be by definition the initial list size, enforcing equally-sized lists gives target sizes of  $m(r-1)$  at every level of the tree but the last, where it is  $l - m(r-1)(h-1)$ . Then if we let  $g = \text{gain}(r)$ , the cost of one dissection is equal to  $\mathcal{O}(3^{m(r-1-g)})$  and returning solutions in amortised constant time translates into the following:

$$\frac{3^{mr}}{3^{l-m(r-1)(h-1)}} \geq 3^{m(r-1-g)} \quad \therefore m \geq \frac{l}{(r-1)(h-1) + 1 + g}, \quad (6)$$

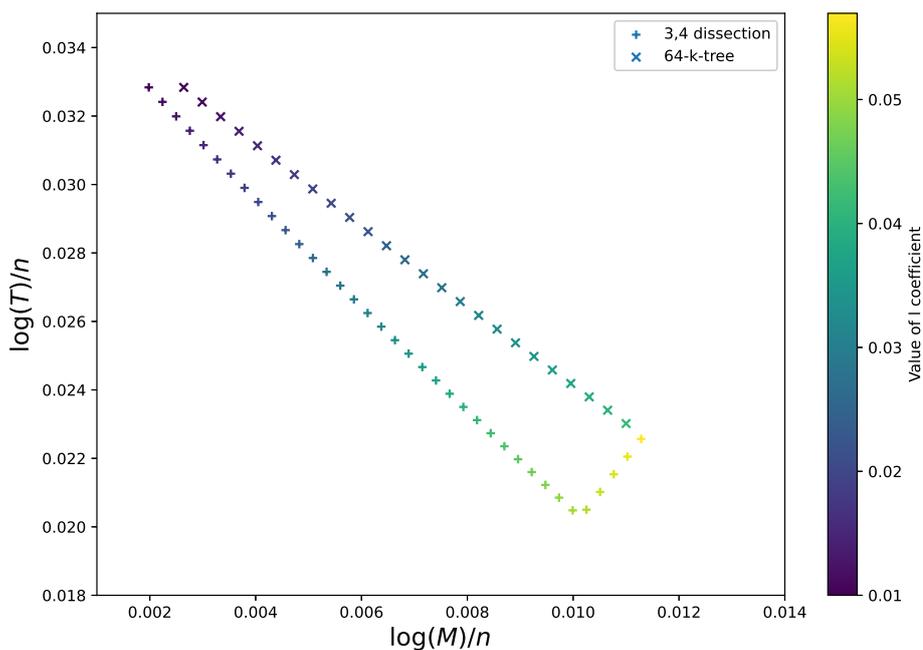


Fig. 7: Best time-memory tradeoff for the syndrome decoding problem in the Wave regime using raw 3,4-dissection and 64-tree.

or simply  $m = l/((r-1)(h-1) + 1 + g)$  when minimising the memory.

Finally, the straightforward generalisation of Eq. (5) is given by:

$$3^m \leq 2^{\frac{k+l}{r^h}} \quad (7)$$

We summarise these constraints in the parameter selection in Algorithm 4, Appendix E, an implementation of which being available at [https://github.com/charlotte-lefevre/TM\\_tradeoffs\\_SDP](https://github.com/charlotte-lefevre/TM_tradeoffs_SDP).

## 5.2 Improvements for syndrome decoding

We now present (and ultimately combine) two improvements to the dissection in tree: the first aims at reducing its granularity while the second is a straightforward adaptation of the smoothing technique. The price to pay for both are exponentially larger memory costs and thus less favourable tradeoffs.

**Improving the granularity of the dissection in tree.** In a raw dissection tree, the dissections performed at every level are exhaustive. To decrease the overall granularity, one idea would then be to consider non-exhaustive dissections so that fewer solutions are eventually returned. This however also requires to decrease the target sizes at every level to compensate, and thus also to increase the initial list sizes if one wishes to return solutions in amortised constant time.

Let  $\alpha$  be a new parameter s.t. the  $r$ -dissection now enumerates only  $3^{m(r-\alpha)}$  candidates from the product of the  $r$  input lists. Enforcing equally-sized lists gives target sizes of  $m(r-1-\alpha)$  at every level of the tree but the last, where it is  $l - m(h-1)(r-1-\alpha)$ . The expected number of returned solutions is then equal to:

$$S = \frac{3^{m(r-\alpha)}}{3^{l-m(h-1)(r-1-\alpha)}} \quad (8)$$

Each dissection now costs  $\mathcal{O}(3^{m(r-g-1-\alpha)})$  (provided that this is not lower than their granularity), and returning solutions in amortised constant time translates into the following:

$$\frac{3^{m(r-\alpha)}}{3^{l-m \times (h-1)(r-1-\alpha)}} \geq 3^{m(r-g-1-\alpha)}$$

$$\therefore m \geq \frac{l}{(h-1)(r-1) + 1 + g - \alpha(h-1)} \quad (9)$$

The memory increase for positive values of  $\alpha$  is then visible by comparing Eq. (6) and Eq. (9). It remains to determine the optimal  $\alpha$ , which in the amortised constant time regime is constrained by two phenomena:

1. The number of returned solutions must not be greater than necessary, *i.e.*  $S \leq S_l$ . Letting  $s = \log_3(S_l)$  and injecting the minimal value for  $m$  given by Eq. (9) into Eq. (8) gives (after a tedious computation):

$$[l - (h-1)s] \times \alpha \geq [l(r-g-1) - s(g+1 + (h-1)(r-1))] \quad (10)$$

2. The required number of solution at every level must not be lower than the granularity of the dissection. From Proposition 14 this gives:

$$r - g - 1 - \alpha > g - \mathit{gain}(g+1) \quad (11)$$

One would then pick the smallest value of  $\alpha$  satisfying both constraints to minimise the overall memory cost. We summarise this in the parameter selection Algorithms 6 and 7 from Appendix E, implementations being available at [https://github.com/charlotte-lefevre/TM\\_tradeoffs\\_SDP](https://github.com/charlotte-lefevre/TM_tradeoffs_SDP).

**Smoothing the dissection tree.** Since the dissection tree features a population constraint similar to the  $k$ -tree algorithm, we may adapt to it the smoothing technique from Proposition 9. This leads to the following:

**Proposition 16.** *Let  $l, r, h$  be fixed,  $g := \mathit{gain}(r)$ . If  $3^l > 2^{(k+l)/(r^{h-1})}$  and  $3^{l/(g+1+(r-1)(h-2))} < 2^{(k+l)/(r^{h-1})}$ , then one can use a smoothed tree with  $h$  levels of  $r$ -dissections to obtain  $2^{m(r-g-1)}$  solutions to the generalised birthday problem with  $r$  lists in amortised constant time, where:*

$$m = \frac{1}{(h-2)(r-1) + g} \left( l \log_2(3) - \frac{k+l}{r^{h-1}} \right).$$

The proof is similar to the one of Proposition 9 and given in Appendix D. The parameter selection with smoothing is given by Algorithm 5, Appendix E.

**Combination of the improvements.** There are settings where both previous improvements may be jointly necessary. This can be done by using a two-step process: the bottom level of the tree is used to satisfy a constraint of size  $t$ , which becomes a parameter, in a possibly non-exhaustive way as controlled by a parameter  $\beta$ . As in a smoothed tree, the goal is to produce intermediate lists of size  $3^m$  (where  $m$  is another parameter), starting from ones of size  $3^{\tilde{m}}$ ,  $\tilde{m} := \log_3(2^{\frac{k+l}{r^h}})$ . Then the  $h-1$  remaining levels are required to satisfy a target of size  $l-t$  with input lists of size  $3^m$ , in a possibly non-exhaustive way as controlled by a parameter  $\alpha$ .

Parameters leading to valid instances in amortised constant time must then satisfy the following constraints:

1. The expected list size is larger than  $3^m$  after the first level:

$$\frac{3^{\tilde{m} \times (r-\beta)}}{3^t} \geq 3^m \quad \therefore \beta \leq r - \frac{m+t}{\tilde{m}}$$

2. The parameter  $\alpha$  is constrained by Eqs. (10) and (11).
3. The parameter  $\beta$  is constrained by Eq. (11).
4. The cost is dominated by the upper part of the tree:

$$3^{\tilde{m} \times (c-\beta)} \leq 3^{m(c-\alpha)} \quad \therefore \beta \geq c - \frac{m}{\tilde{m}}(c-\alpha),$$

where  $c := r - g - 1$ .

When searching for valid parameterizations, it is best to first select the value for  $t$  and to take it as large as possible as this minimises the memory cost. This makes sense, intuitively, since in that case the tree is as close as possible to a balanced (non-smoothed) one.

The full parameter selection algorithm is given in [Algorithm 8](#), [Appendix E](#), and an implementation is provided at [https://github.com/charlotte-lefevre/TM\\_tradeoffs\\_SDP](https://github.com/charlotte-lefevre/TM_tradeoffs_SDP). The impact of the granularity improvements, also combined with smoothing, are illustrated for the 3,4-dissection in [Fig. 8](#). Thanks to these improvements, the 3,4-dissection is now applicable to larger memory regimes but at the cost of less favourable tradeoffs (clearly observable from the changes of the slopes).

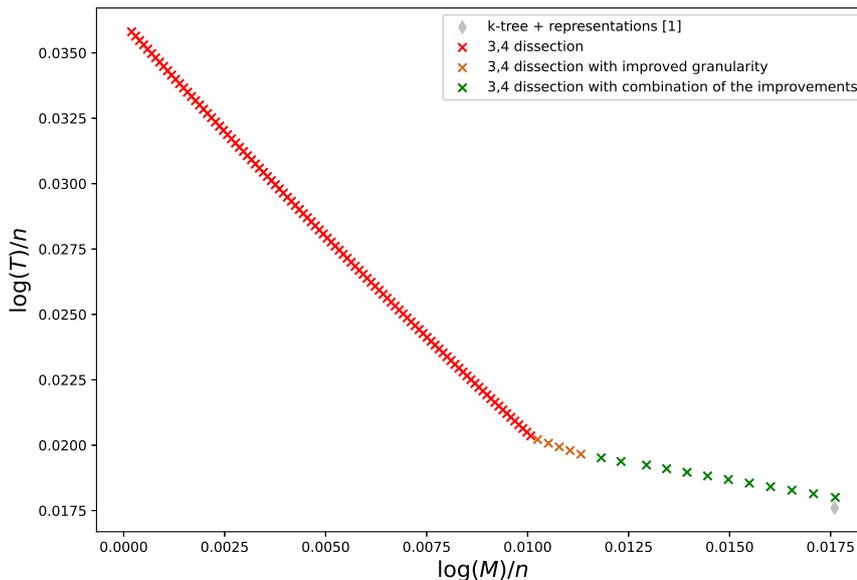


Fig. 8: Best time-memory tradeoff for the syndrome decoding problem in the Wave regime using 3,4-dissection. From  $M \approx 2^{0.01n}$ , the granularity of the 3,4-dissection becomes too coarse, so that non-exhaustive dissections are henceforth considered. Then at  $M \approx 2^{0.0112n}$ , [Eq. \(7\)](#) is no more satisfied, leading thus to the use of the smoothing technique.

### 5.3 Experimental results

As a proof of concept, we implemented the 3,4-dissection algorithm using Sage. The main aim here is to check that the practical number of iterations of the *Subset sum step* before finding a solution to the SDP coincides with the theoretical prediction. This implementation is not fully optimised and relies on a general-purpose finite-field linear algebra software packaged within Sage. This restricts its usage to relatively small parameters and we only considered instances up to  $n = 875$ , which in the Wave regime translates to  $k = 591$ ,  $l = 48$ . With this instantiation, one iteration of the *Subset sum step* combined with the *Probabilistic reconstruction step* takes on average 800 seconds on a (virtualised) i386 processor.<sup>7</sup> With 10 runs of the full algorithm, 5.7 iterations of the *Subset sum step* were necessary on average before finding a solution, which is somewhat consistent with the theoretically expected 2.9, especially given the small number of runs. Moreover, with the instantiation  $n = 560$ ,  $k = 379$ ,  $l = 34$ , the average number of iterations with 110 runs is 12.05, which comes very close to the 12.3 expected number of iterations.

The code of this proof-of-concept implementation is available at [https://github.com/charlotte-lefevre/TM\\_tradeoffs\\_SDP](https://github.com/charlotte-lefevre/TM_tradeoffs_SDP).

<sup>7</sup> The computer used for the tests has an Apple M1 processor, but Sage uses Apple’s Intel emulator.

## 6 Application to Wave

We summarise our best time-memory tradeoffs for solving the syndrome decoding problem in the Wave regime in Fig. 9. We do this in two settings: in Fig. 9b we use asymptotic estimates similar to the ones used in the previous sections, while Fig. 9a is an estimate in bit complexity for concrete proposed security parameters. The plots were all drawn using the parameter selection algorithms presented in Appendix E, and the code is available at [https://github.com/charlotte-lefevre/TM\\_tradeoffs\\_SDP](https://github.com/charlotte-lefevre/TM_tradeoffs_SDP).

From these figures it is notable that dissection in tree always outperforms  $k$ -tree instantiations (except in the regime where time and memory are about equal) and almost always outperforms dissection (it is about equivalent in the very low memory regime). For instance, the bit complexity estimate for the 3,4-dissection at  $M \approx 2^{90}$  is about  $2^{25}$  times less than using a smoothed  $k$ -tree algorithm with the same amount of memory. For low-memory regimes, the best instances use layered dissections with 2 levels, while from  $M \approx 2^{0.009n}$ , the best tradeoffs are obtained with 4, 4 or 3, 4-dissections.

Table 2: Bit cost estimates for various tradeoffs for solving the generic syndrome decoding problem,  $n = 7236$ ,  $k = 4892$ ,  $w = 6862$ .

| Time      | Memory     | Target tradeoff | Algorithm                       |
|-----------|------------|-----------------|---------------------------------|
| $2^{152}$ | $2^{144}$  | $T = M$         | $k$ -tree + representations [1] |
| $2^{162}$ | $2^{130}$  | $T = M^{5/4}$   | 3,4-dissection                  |
| $2^{177}$ | $2^{88.5}$ | $T = M^2$       | 3,4-dissection                  |
| $2^{194}$ | $2^{64.8}$ | $T = M^3$       | 2,11-dissection                 |
| $2^{213}$ | $2^{42.6}$ | $T = M^5$       | 2,16-dissection                 |
| $2^{247}$ | $2^{24.6}$ | $T = M^{10}$    | 2,29-dissection                 |

The bit costs of Fig. 9a correspond to Wave’s “new” parameters  $n = 7236$ ,  $k = 4892$ ,  $w = 6862$  [1], and were computed using the following assumptions or simplifications:

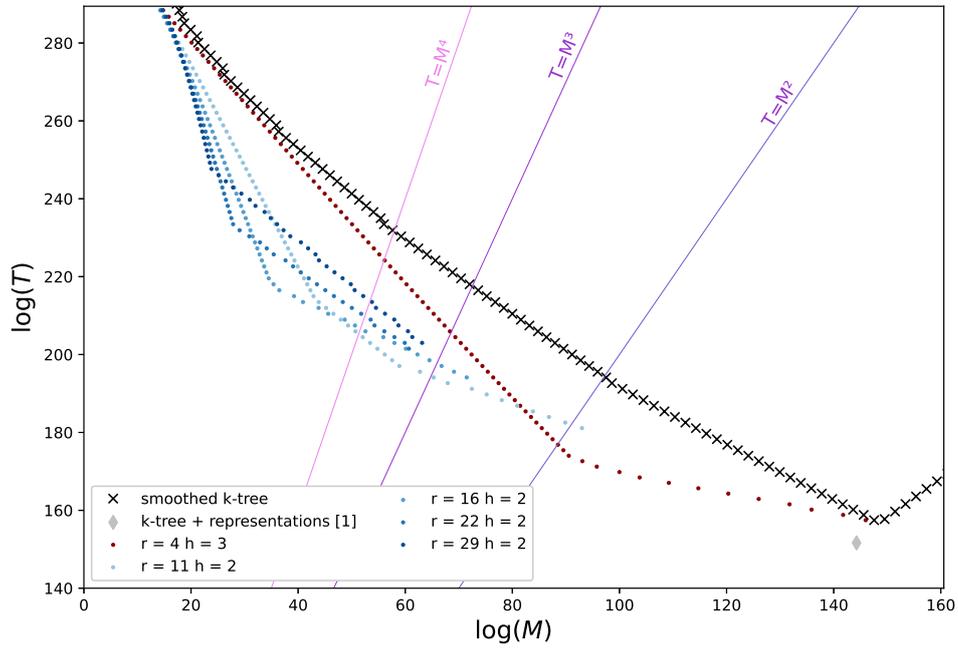
- Elements of  $\mathbb{F}_3$  are stored on 2 bits, and elementary operations in  $\mathbb{F}_3^n$  cost  $2n$  bit operations.
- Polynomial factors of the algorithms are taken into account.
- Polynomial factors in the estimate for  $S_l$  are taken into account.
- Computing  $L_1 \bowtie_{\vartheta} L_2$  for some lists  $L_1, L_2$  of elements of  $\mathbb{F}_3^n$  and some  $\vartheta$  costs  $2n(\#L_1 + \#L_2)$  as long as the size of the result is not larger than one of the input lists.

The last simplification implies a constant cost for memory access, which is an unrealistic underestimation for most of the considered memory sizes. The provided costs should thus not be interpreted as precise estimates but rather as intermediate points between asymptotic computations and a full and accurate modelling of an attack, which is out of the scope of this paper.

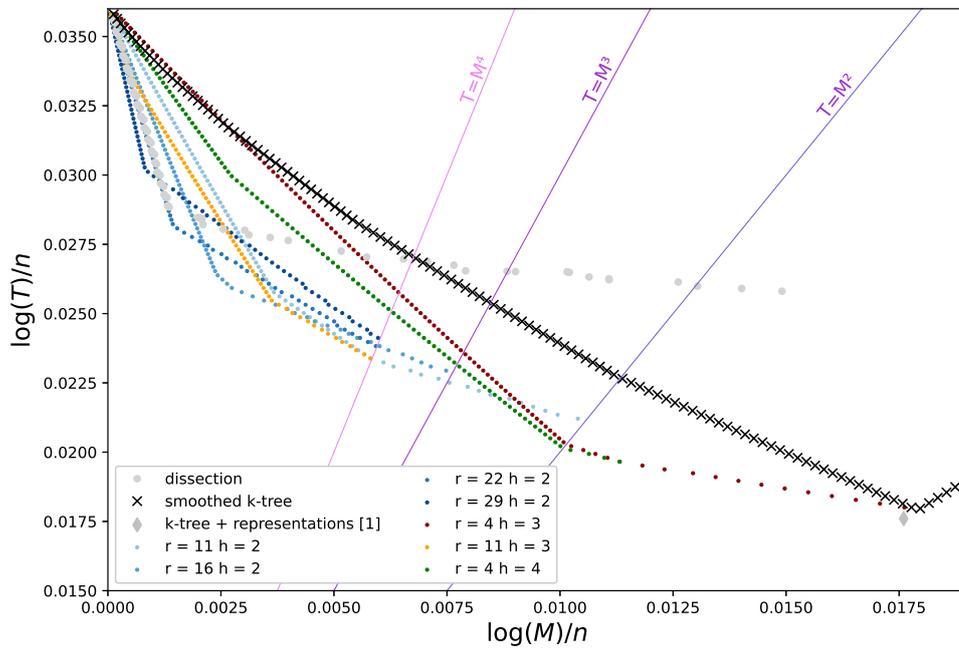
Finally, we list some of the most notable tradeoffs for concrete parameters in Table 2.

## Acknowledgements

The first author was partially supported by the French National Research Agency in the framework of the *Investissements d’avenir* programme (ANR-15-IDEX-02). The second author is in part supported by the Netherlands Organisation for Scientific Research (NWO) under grant OCENW.KLEIN.435. Part of this work was done when the second author was with Université Grenoble Alpes, and was partially supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) in the framework of the *Investissement d’avenir* programme. Finally we wish to thank the reviewers for all their comments.



(a) Bit cost for fixed parameters.



(b) Asymptotic cost.

Fig. 9: Summary of obtained time-memory tradeoffs. For the dissection in tree,  $r$  denotes the dissection used and  $h$  the number of levels.

## References

1. Rémi Bricout, André Chailloux, Thomas Debris-Alazard, and Matthieu Lequesne. Ternary Syndrome Decoding with Large Weight. In Kenneth G. Paterson and Douglas Stebila, editors, *SAC 2019*, volume 11959 of *Lecture Notes in Computer Science*, pages 437–466. Springer, 2019.
2. John T. Coffey and Rodney M. Goodman. The complexity of information set decoding. *IEEE Trans. Inf. Theory*, 36(5):1031–1037, 1990.
3. Thomas Debris-Alazard. *Cryptographie fondée sur les codes : nouvelles approches pour constructions et preuves ; contribution en cryptanalyse. (Code-based Cryptography: New Approaches for Design and Proof ; Contribution to Cryptanalysis)*. PhD thesis, Pierre and Marie Curie University, Paris, France, 2019.
4. Thomas Debris-Alazard, Nicolas Sendrier, and Jean-Pierre Tillich. Wave: A New Family of Trapdoor One-Way Preimage Sampleable Functions Based on Codes. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019*, volume 11921 of *Lecture Notes in Computer Science*, pages 21–51. Springer, 2019.
5. Itai Dinur. An algorithmic framework for the generalized birthday problem. *Des. Codes Cryptogr.*, 87(8):1897–1926, 2019.
6. Itai Dinur, Orr Dunkelman, Nathan Keller, and Adi Shamir. Efficient Dissection of Composite Problems, with Applications to Cryptanalysis, Knapsacks, and Combinatorial Search Problems. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 719–740. Springer, 2012.
7. Ilya Dumer. On minimum distance decoding of linear codes. In *Proc. 5th Joint Soviet-Swedish Int. Workshop Inform. Theory*, pages 50–52, 1991.
8. Andre Esser, Felix Heuer, Robert Kübler, Alexander May, and Christian Sohler. Dissection-BKW. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018*, volume 10992 of *Lecture Notes in Computer Science*, pages 638–666. Springer, 2018.
9. Nick Howgrave-Graham and Antoine Joux. New generic algorithms for hard knapsacks. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110, pages 235–256. Springer, 2010.
10. Alexander Meurer. *A coding-theoretic approach to cryptanalysis*. PhD thesis, Ruhr University Bochum, 2013.
11. Lorenz Minder and Alistair Sinclair. The Extended k-tree Algorithm. *J. Cryptol.*, 25(2):349–382, 2012.
12. Robert Niebuhr, Pierre-Louis Cayrel, Stanislav Bulygin, and Johannes Buchmann. On lower bounds for information set decoding over  $\mathbb{F}_q$ . In *SCC 2010*, volume 10, pages 143–157, 2010.
13. Christiane Peters. Information-Set Decoding for Linear Codes over  $\mathbb{F}_q$ . In Nicolas Sendrier, editor, *PQCrypto 2010*, volume 6061 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2010.
14. Eugene Prange. The use of information sets in decoding cyclic codes. *IRE Trans. Inf. Theory*, 8(5):5–9, 1962.
15. Richard Schroeppel and Adi Shamir. A  $T = \mathcal{O}(2^{n/2})$ ,  $S = \mathcal{O}(2^{n/4})$  algorithm for certain NP-complete problems. *SIAM J. Comput.*, 10(3):456–464, 1981.
16. Jacques Stern. A method for finding codewords of small weight. In Gérard D. Cohen and Jacques Wolfmann, editors, *Coding Theory and Applications, 3rd International Colloquium, Toulon, France, November 2-4, 1988, Proceedings*, volume 388 of *Lecture Notes in Computer Science*, pages 106–113. Springer, 1988.
17. Rodolfo Canto Torres. Asymptotic analysis of ISD algorithms for the q-ary case. In *Proceedings of the Tenth International Workshop on Coding and Cryptography WCC 2017*, 2017.
18. David A. Wagner. A Generalized Birthday Problem. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 2002.

## A Approximation of the number of required solution

Eq. (1) from Proposition 4 contains binomial coefficients which can be complex to evaluate. Here, we aim at approximating the former to obtain a simple and reasonable approximation. The corresponding formula is given in Proposition 18. In particular, this shows that the time cost of *Subset Sum step* is expected to be exponential.

**Lemma 17 ([3, Lemma 1.2]).** *Let  $n, w \in \mathbb{N}$  with  $w < n$ . Then one has*

$$\binom{n}{w} 2^w \underset{w = \mathcal{O}(n)}{\overset{n \rightarrow +\infty}{\sim}} \frac{1}{\sqrt{2\pi w(1-w/n)}} 3^{nh_3(w/n)},$$

where  $h_3$  is the ternary entropy function, defined over  $]0, 1[$  by:

$$h_3(x) = -(1-x) \log_3(1-x) - x \log_3\left(\frac{x}{2}\right).$$

**Proposition 18.** *Let  $\mathbf{H}_1, \mathbf{H}_2, \mathbf{s}'_1, \mathbf{s}'_2, \mathbf{e}'_1, n, w, k, l$  as defined in Proposition 4. Then one has:*

$$\Pr[\text{wt}(\mathbf{e}'_1) = w - k - l] \underset{n \rightarrow +\infty}{\approx} 3^{(n-k-l) \times h_3\left(\frac{w-k-l}{n-k-l}\right) + l - n + k}.$$

This proposition is a consequence of Lemma 17 applied to the two binomial coefficients appearing in Proposition 4. The polynomial factor in Lemma 17 has been ignored, hence the approximation symbol. This formula is used in practice to compute the value of  $S_l$  when the code length is not fixed.

## B The meet-in-the-middle algorithm and its application

The meet-in-the-middle is a basic building block for time-memory trade-offs. In this section we describe this algorithm embedded in the PGE+SS framework to solve the *Subset sum problem*. This combination is also known under the name of Stern/Dumer algorithm [16,7]. The meet-in-the-middle requires two lists denoted by  $L_1, L_2$ . In the context of the SDP, the way to build the lists is described in Section 4.1, where  $\mathcal{W}_1 := \left\{ \left( \mathbf{x} \mathbf{0}^{\left(\frac{k+l}{2}\right)} \right) \middle| \mathbf{x} \in \mathbb{F}_3^{\frac{k+l}{2}}, \text{wt}(\mathbf{x}) = \frac{k+l}{2} \right\}$ , and  $\mathcal{W}_2 := \left\{ \left( \mathbf{0}^{\left(\frac{k+l}{2}\right)} \mathbf{x} \right) \middle| \mathbf{x} \in \mathbb{F}_3^{\frac{k+l}{2}}, \text{wt}(\mathbf{x}) = \frac{k+l}{2} \right\}$ . Moreover, any element  $\mathbf{l}_1$  in  $L_1$  (resp.  $\mathbf{l}_2$  in  $L_2$ ) must be associated to the vector  $\mathbf{e}'$  in  $\mathcal{W}_1$  (resp.  $\mathbf{e}''$  in  $\mathcal{W}_2$ ) such that  $\mathbf{e}' \mathbf{H}^T = \mathbf{l}_1$  (resp.  $\mathbf{e}'' \mathbf{H}^T - \mathbf{s} = \mathbf{l}_2$ ). We denote this link by  $\mathbf{e}' =: \text{associated}(\mathbf{l}_1)$ . The meet-in-the-middle algorithm is then described as follows:

For every  $\mathbf{l}_2 \in L_2$ :

If there exists  $\mathbf{l}_1 \in L_1$  s.t  $\mathbf{l}_1 = -\mathbf{l}_2$ :

- $\mathbf{e}' \leftarrow \text{associated}(\mathbf{l}_1)$
- $\mathbf{e}'' \leftarrow \text{associated}(\mathbf{l}_2)$
- process  $\mathbf{e}' + \mathbf{e}''$  in the *Probabilistic reconstruction step*

In practice, searching for a colliding element within a list can be done in a constant time, and this makes the meet-in-the-middle much more efficient than a memoryless exhaustive search in terms of time cost.

As the parity-check matrix  $\mathbf{H}$  is sampled uniformly at random, an heuristic hypothesis made here is that the lists  $L_1$  and  $L_2$  contain random uniform vectors. Therefore, the expected numbers of colliding elements is approximated by:

$$\frac{\#L_1 \#L_2}{3^l} \tag{12}$$

Under the constraint that solutions must be returned in amortised constant time, the minimal memory usage is given when  $\#L_1 = \#L_2 = 3^l$ , and this also minimises the granularity of the algorithm. Thus the memory and time costs are given by  $M = 3^l, T = 3^l \cdot \max\left(1, \frac{S_l}{3^l}\right)$ .

*Remark 19.* It is also possible to consider a meet-in-the-middle with  $r$  lists  $L_1, L_2, \dots, L_r$  for any  $r \geq 2$  by searching an element in  $L_1$  equal to  $-\mathbf{l}_2 - \mathbf{l}_3 - \dots - \mathbf{l}_r$  for every tuple  $(\mathbf{l}_1, \mathbf{l}_2, \dots, \mathbf{l}_r) \in L_1 \times L_2 \times \dots \times L_r$ .

## C Recursive construction of the exhaustive dissection

All statements presented in this section have already been proved by Dinur *et al.* [6]. Here we wish to provide a self-contained description of the exhaustive  $r$ -dissection framework for any  $r \geq 4$ . We will see that an  $r$ -dissection of gain  $g$  is split into an  $(r - g - 1)$ -dissection of gain  $g - 1$  and a  $(g + 1)$ -meet-in-the middle. However, in order to understand the construction of the dissection, we rather start from the  $r$ -dissection of gain equal to  $g - 1 \geq 0$  (by definition), then build from it the  $(r + g + 1)$ -dissection. We then show by induction on the gain that the newly-built dissection has a gain equal to  $g$  and show a few properties about the amortised time and the memory cost. In the last paragraph, we give advice about how to find concretely the gain (thus the cutting) of an  $r$ -dissection for any  $r \geq 4$ .

Let  $m$  be s.t the size of the list  $L_i$  is equal to  $3^m$  for any  $i$ . The base case, *i.e.* the 4-dissection (where  $g = 1$ ) has already been investigated in Section 4.3. Let then  $g > 1$  be such that the

$r$ -dissection has a gain equal to  $g - 1$ . Assume by induction that the former is able to output all of the  $3^{m(r-g)}$  solutions with a target size equal to  $mg$  in amortised constant time; with a memory cost equal to  $3^m$ . Now, we can describe the  $(r + g + 1)$ -dissection with a target  $\mathbf{s} \in \mathbb{F}_3^{m(g+1)}$ <sup>8</sup>. We will show that the latter has a gain equal to  $g$  and that solutions are returned in amortised constant time with a memory cost of  $3^m$ . Figure 10 illustrates the following description of the  $(r + g + 1)$ -dissection:

For every intermediate target  $\mathbf{t} \in \mathbb{F}_3^{mg}$ :

- (i) Run a  $(g+1)$  exhaustive meet-in-the-middle with the lists  $L_1, \dots, L_{g+1}$ , available memory  $3^m$  and target equal to  $\mathbf{s}_1 - \mathbf{t} \in \mathbb{F}_3^{mg}$ , where  $\mathbf{s}_1$  comprises the  $gm$  last coordinates of  $\mathbf{s}$ . The  $3^m$  (expected) solutions are stored in a list called  $L$ . The time cost of this step is equal to  $3^{gm}$ .
- (ii) Run an  $r$ -dissection with the lists  $L_{g+2}, \dots, L_{r+g+1}$  and the target  $\mathbf{t} \in \mathbb{F}_3^{mg}$ . By induction, its memory cost is equal to  $3^m$ , the time cost is given by  $3^{m(r-g)}$ , and the solutions are returned in amortised constant time.
- (iii) Every solution returned by the  $r$ -dissection has shape  $(\mathbf{y} \mathbf{t})$  with  $\mathbf{y} \in \mathbb{F}_3^m$ . The final step consists in searching in  $L$  an element equal to  $(\mathbf{s}_2 - \mathbf{y} \mathbf{s}_1 - \mathbf{t})$ , where  $\mathbf{s}_2$  comprises the  $m$  first coordinates of  $\mathbf{s}$ . This step can be done in a constant time per solution.

The solutions returned by the  $r$ -dissection are generated on-the-fly, so that no extra-memory in step (iii) is required to store them. One can check by induction that  $r - g \geq g$ , so that the time cost of the  $r$ -dissection dominates the one the  $(g + 1)$ -meet-in-the middle. An immediate consequence is that solutions to the  $(r + g + 1)$ -list problem are returned in amortised constant time, so that  $T = 3^{mr}$ : the gain of the  $(r + g + 1)$ -dissection is thus equal to  $g$ . Finally, one can easily check that the memory cost is  $M = 3^m$ .

*Remark 20.* The previous description does not tell how to build the  $r$ -dissection for any  $r \geq 4$  (e.g. with  $r = 5$ ). Starting from the  $r$ -dissection of gain  $g$ , it is possible to define the  $(r + a)$ -dissection for any  $a < g + 2$ . The idea is to run an  $r$ -dissection with lists  $L_{a+1}, \dots, L_{a+r}$  and target  $\mathbf{s} - \mathbf{l}_1 - \mathbf{l}_2 - \dots - \mathbf{l}_a$ ; and repeat the procedure for every tuple  $(\mathbf{l}_1, \mathbf{l}_2, \dots, \mathbf{l}_a) \in L_1 \times L_2 \times \dots \times L_a$ . The gain of this dissection is also equal to  $g$ , and one can show that the properties previously shown are still true. Therefore the  $r$ -dissection is well defined for any  $r \geq 4$ .

In practice, to find the gain of a  $r$ -dissection for a fixed  $r \geq 4$ , one can use the *magic sequence*  $(M_g)_{g \in \mathbb{N}}$  introduced by Dinur *et al.* which is by construction such that  $\forall g \in \mathbb{N}, M_g = \min\{r \mid \text{gain}(r) = g\}$ . It has been proven that  $M_g = \frac{(g+1)(g+2)}{2} + 1$  [6]. Finally, one can remark that  $\text{gain}(r) = \max\{g \mid M_g \leq r\}$  so that the splitting of an  $r$ -dissection into a  $(r - \text{gain}(r) - 1)$ -dissection and a  $(\text{gain}(r) + 1)$ -meet-in-the-middle is easy to determine.

## D Proof of Proposition 16

The aim here is to prove Proposition 16. The proof is very close to the case of the smoothing of the  $k$ -tree algorithm (see [1, Proposition 4] and Proposition 9).

**Proposition 16.** *Let  $l, r, h$  be fixed,  $g := \text{gain}(r)$ . If  $3^l > 2^{(k+l)/(r^{h-1})}$  and  $3^{l/(g+1+(r-1)(h-2))} < 2^{(k+l)/(r^{h-1})}$ , then one can use a smoothed tree with  $h$  levels of  $r$ -dissections to obtain  $2^{m(r-g-1)}$  solutions to the generalised birthday problem with  $r$  lists in amortised constant time and memory cost  $2^m$ , where:*

$$m = \frac{1}{(h-2)(r-1) + g} \left( l \log_2(3) - \frac{k+l}{r^{h-1}} \right).$$

*Proof.* The idea is exactly the same as the one in Section 4.2: the constraint size  $w$  from level 1 to 2 is reduced, and this allows the list size on level 2 to be larger than the one on level 1. Then from level 2, a (non-smoothed) dissection in tree is run to obtain solutions in amortised constant time.

Let  $\varsigma := l \log(3)$  normalise in base 2 the size of the dimension- $l$  ternary constraint that one wishes to solve and  $\tau := (k+l)/r^h$  be the logarithm of the maximum size of  $r$  lists of full-weight vectors partitioning the domain. We aim at finding initial and subsequent constraints  $w, w' = w'' = \dots = w^{(h-1)}$  and  $w^{(h)}$  s.t the following constraints are satisfied:

<sup>8</sup> Larger targets can be considered, but in this case the amortised time increases exponentially in the overhead of the target size .

1. The first level dissections return lists with an expected size of  $2^m$ :  $m = r\tau - w$ .
2. The list cardinal is preserved from level 2 to level h-1:  $rm - w' = m$ .
3. The constraints sum to  $\varsigma$ :  $w + (h - 2)w' + w^{(h)} = \varsigma$ .
4. Solutions are returned in amortised constant time:  $rm - w^{(h)} \geq m(r - g - 1)$ .

As we want to minimise the memory usage, the inequality in constraint 4 becomes an equality. By using respectively constraints 3,1,2 to substitute  $w^{(h)}$ ,  $w$  and  $w'$  in constraint 4, one can obtain the  $m$  of the proposition. If  $3^l > 2^{\frac{k+l}{r^{h-1}}}$  and  $3^{\frac{l}{g+1+(r-1)(h-2)}} < 2^{\frac{k+l}{r^{h-1}}}$ , one can check that  $w \geq 0$  and  $m \geq 0$ .

□

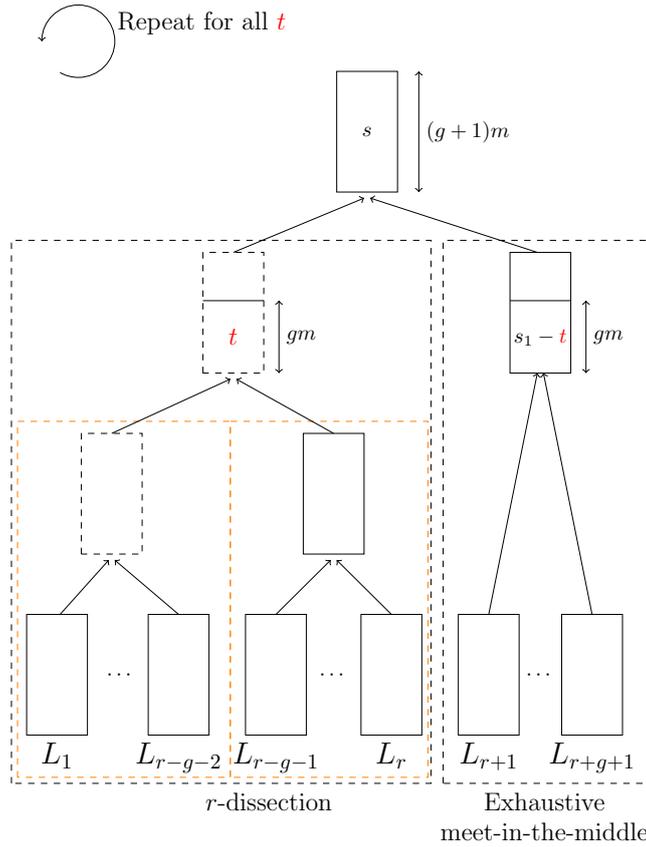


Fig. 10: Recursive dissection construction where the leaves list cardinal is defined to be equal to  $3^m$ . A list drawn with dashed lines is not stored in memory and processed on-the-fly.

## E Algorithms

We present here algorithms useful to evaluate the memory usage and time cost of the algorithms described in this paper.

### E.1 From Section 4.2

---

**Algorithm 1:** *find\_param\_kTree*( $n, l, k, w$ )
 

---

**Input:**  $n$ : code length,  $l$ : size of the sub-problem,  $k$ : code dimension,  $w$ : target weight of SDP

**Output:**  $(M, T, a)$  required memory, time and number of levels or  $\perp$  if algorithm fails

```

1  $S_l \leftarrow Nb\_required\_sol(n, l, k, w)$  // see Proposition 4
2  $a \leftarrow 1$ 
3 if  $3^{l/a} > 2^{\frac{k+l}{2^a}}$  then
4 | return  $\perp$ 
5 end
6 while  $3^{l/a} \leq 2^{\frac{k+l}{2^a}}$  do
7 |  $a \leftarrow a + 1$ 
8 end
9  $a \leftarrow a - 1$ 
10 return  $(3^{l/a}, \max(S_l, 3^{l/a}), a)$ 

```

---

**Algorithm 2:** *find\_param\_SmoothedkTree*( $n, l, k, w$ )
 

---

**Input:**  $n$ : code length,  $l$ : size of the sub-problem,  $k$ : code dimension,  $w$ : target weight of SDP

**Output:**  $(M, T, a)$  required memory, time and number of levels or  $\perp$  if algorithm fails

```

1  $S_l \leftarrow Nb\_required\_sol(n, l, k, w)$  // see Proposition 4
2  $a \leftarrow 3$ 
3 if  $3^{\frac{l}{a-1}} > 2^{\frac{k+l}{2^{a-1}}}$  then
4 | return  $\perp$ 
5 end
6 while  $3^{\frac{l}{a-1}} \leq 2^{\frac{k+l}{2^{a-1}}}$  do
7 |  $a \leftarrow a + 1$ 
8 end
9  $a \leftarrow a - 1$ 
10  $m \leftarrow \frac{1}{a-2} (l \cdot \log_2(3) - \frac{k+l}{2^{a-1}})$ 
11 return  $(2^m, \max(S_l, 2^m), a)$ 

```

---

**E.2 From Section 4.3**

---

**Algorithm 3:** *find\_param\_Dissection*( $r, n, l, k, w, m, \mu$ )

---

**Input:**  $r$ : number of lists,  $n$ : code length,  $l$ : size of the sub-problem,  $k$ : code dimension,  
 $w$ : target weight of SDP,  $m$ :  $\log_3$  of base lists size,  $\mu$  s.t memory cost is  $3^{m\mu}$

**Output:**  $(M, T)$  memory and time or  $\perp$  if algorithm fails

```

1  $g \leftarrow \text{gain}(r, \mu)$ 
2  $S_l \leftarrow \text{Nb\_required\_sol}(n, l, k, w)$  // see Proposition 4
3 if  $m > \log_3(2^{\frac{k+l}{r}})$  then
4 | return  $\perp$ 
5 end
/*  $s$  is the  $\log_3$  of required solutions in the large sub-dissection divided
   by  $m$  */
6  $s \leftarrow \frac{\log_3(S_l)}{m} + \max(0, -g + \frac{l}{m} - \mu)$ 
/* If no granularity issues */
7 if  $s > \max(\mu, (g - \text{gain}(g + \mu, \mu)))$  then
8 | return  $(3^{m\mu}, 3^{ms})$ 
9 end
/* Else, try to reduce granularity by tweaking the dissection */
10 else if  $s \geq \max[g - 2\mu - 2 \text{gain}(g, \mu) + \text{gain}(g + \mu, \mu), \mu]$  then
11 | return  $(3^{m\mu}, 3^{m[s+g-\text{gain}(g+\mu,\mu)]/2})$ 
12 end
/* Else, the dissection granularity could not be lowered */
13 else
14 | return  $(3^{m\mu}, 3^{m(g-\text{gain}(g+\mu,\mu))})$ 
15 end
```

---

**E.3 From Section 5**

---

**Algorithm 4:** *find\_param\_DissectTree*( $r, h, l, n, k, w$ )

---

**Input:**  $r$ : number of lists,  $h$ : number of levels,  
 $l$ : size of the sub-problem,  $n$ : code length,  $k$ : code dimension,  
 $w$ : target weight of SDP

**Output:**  $(M, T)$  memory and time using  $h, r$ -dissection tree or  $\perp$  if algorithm fails

```

1  $g \leftarrow \text{gain}(r)$ 
2  $S_l \leftarrow \text{Nb\_required\_sol}(n, l, k, w)$  // see Proposition 4
3  $m \leftarrow \frac{l}{(r-1)(h-1)+g+1}$ 
4 if  $\log_2(3) \cdot m > \frac{k+l}{r^h}$  then // constraint (7)
5 | return  $\perp$ 
6 end
7 return  $(3^m, \max(S_l, 3^{(r-g-1)m}))$ 
```

---



---

**Algorithm 5:** *find\_param\_SmoothedAlgo*( $r, h, l, n, k, w$ )

---

**Input:**  $r$ : number of lists,  $h$ : number of levels,  
 $l$ : size of the sub-problem,  $n$ : code length,  $k$ : code dimension,  
 $w$ : target weight of SDP

**Output:**  $(M, T)$  memory and time using smoothed  $h, r$ -dissection tree or  $\perp$  if algorithm fails

```

1  $g \leftarrow \text{gain}(r)$ 
2  $S_l \leftarrow \text{Nb\_required\_sol}(n, l, k, w)$  // see Proposition 4
3 if  $\log_2(3) \times \frac{l}{g+1+(r-1)(h-2)} < \frac{k+l}{r^{h-1}}$  and  $\log_2(3) \times l > \frac{k+l}{r^{h-1}}$  then
4 |  $m \leftarrow \frac{1}{(h-2)(r-1)+g} (\log_2(3) \cdot l - \frac{k+l}{r^{h-1}})$ 
5 | return  $(2^m, \max(S_l, 2^{(r-g-1)m}))$ 
6 end
7 return  $\perp$  // can not apply smoothing technique
```

---

---

**Algorithm 6:** *find\_param\_DissecTree\_betterGranularity*( $r, h, l, n, k, w$ )

---

**Input:**  $r$ : number of lists,  $h$ : number of levels,  
 $l$ : size of the sub-problem,  $n$ : code length,  $k$ : code dimension,  
 $w$ : target weight of SDP  
**Output:**  $(M, T)$  memory and time using  $h, r$ -dissection tree with improved granularity  
or  $\perp$  if algorithm fails

```

1  $g \leftarrow \text{gain}(r)$ 
2  $S_l \leftarrow \text{Nb\_required\_sol}(n, l, k, w)$  // see Proposition 4
3  $\alpha \leftarrow \text{ChooseBestAlpha}(r, h, l, S_l)$  // Algorithm 7
4 if  $\alpha = \perp$  then // cannot find suitable  $\alpha$ 
5 | return  $\perp$ 
6 end
7  $m \leftarrow \frac{l}{(r-1)(h-1)+1+g-\alpha(h-1)}$ 
8 if  $\log_2(3) \cdot m > \frac{k+l}{r^h}$  then // constraint (7)
9 | return  $\perp$ 
10 end
11 return  $(3^m, S_l)$ 

```

---



---

**Algorithm 7:** *ChooseBestAlpha*( $r, h, t, S$ )

---

**Input:**  $r$ : number of lists,  $h$ : number of levels,  
 $t$ : size of the target,  $S$ : number of wanted solutions  
**Output:**  $\alpha$  minimal possible parameter or  $\perp$  if algorithm fails  
/\* minimise alpha according to Eq. (10) \*/

```

1  $g \leftarrow \text{gain}(r)$ 
2 Left  $\leftarrow t - \log_3(S)(h-1)$ 
3 Right  $\leftarrow -\log_3(S)[g+1+(h-1)(r-1)] + t(r-g-1)$ 
4 if Left  $\geq 0$  then
5 |  $\alpha \leftarrow \max\left(0, \frac{\text{Right}}{\text{Left}}\right)$ 
6 end
7 else if Right  $\geq 0$  then
8 | return  $\perp$ 
9 end
10 else
11 |  $\alpha \leftarrow 0$ 
12 end
    /* check if there are granularity problems (Eq. (11)) */
13 if  $\alpha > r - 2g - 1 + \text{gain}(g+1)$  then
14 | return  $\perp$ 
15 end
16 return  $\alpha$ 

```

---

---

**Algorithm 8:** *find\_param\_CombinedAlgo*( $r, h, l, n, k, w, step$ )

---

**Input:**  $r$ : number of lists,  $h$ : number of levels, $l$ : size of the sub-problem,  $n$ : code length,  $k$ : code dimension, $w$ : target weight of SDP,  $step$ : a precision parameter**Output:**  $(M, T)$  memory and time using the combined  $h, r$ -dissection tree or  $\perp$  if algorithm fails

```

1  $\tilde{m} \leftarrow \log_3 \left( 2^{\frac{k+l}{r^h}} \right)$ 
2  $g \leftarrow gain(r)$ 
3  $S_l \leftarrow Nb\_required\_sol(n, l, k, w)$  // see Proposition 4

   /* target max corresponds to find_param_DissectTree */
4  $t \leftarrow (r-1) \times \frac{l}{(h-1)(r-1)+g+1}$ 
5 while  $t > 0$  do
6    $\alpha \leftarrow ChooseBestAlpha(r, h-1, l-t, S_l)$ 
7   if  $\alpha = \perp$  then
8     | continue
9   end
10   $max\beta \leftarrow \min(r-2g-1 + gain(g+1), r - \frac{m+t}{\tilde{m}})$ 
11   $min\beta \leftarrow r-g-1 - \frac{m}{\tilde{m}}(r-g-1-\alpha)$ 
12  if  $max\beta < min\beta$  or  $max\beta < 0$  then // UNSAT constraints
13    |  $t \leftarrow t - step$ 
14    | continue
15  end

   /* similar to find_param_DissectTree_betterGranularity */
16   $m \leftarrow \frac{l-t}{(r-1)(h-2)+g+1-\alpha(h-2)}$ 
17  return  $(3^m, S_l)$ 
18 end
19 return  $\perp$  // no suitable target has been found

```

---