Solving $k$-List Problems
and their Impact on Information Set Decoding

Dissertation Thesis

Leif Both
July 25, 2018
Erstgutachter: Prof. Dr. Alexander May
Zweitgutachter: Prof. Dr. Gregor Leander
Tag der mündlichen Prüfung: 13.07.2018
4.5 Representation Techniques ........................................ 74

5 Information Set Decoding with Nearest Neighbors 82
   5.1 A Link to the Approximate $k$-List Problem .................. 82
   5.2 A New Algorithm for Simple ISD .............................. 86
   5.3 Results and Comparison ........................................ 100
   5.4 Application: The McEliece Cryptosystem ..................... 104
   5.5 Application: Learning Parity with Noise ..................... 106
   5.6 Open Problems .............................................. 111

Bibliography ........................................................ 111

Bibliography ........................................................ 112
List of Algorithms

1.1 Sort-and-Match .................................................. 14
1.2 Filter .............................................................. 15
1.3 NN-Enumerate-Pairs ............................................. 17
1.4 NN-Meet-in-the-Middle ......................................... 17
1.5 NN-Search ......................................................... 18

2.1 Simple Sort-and-Match ......................................... 21
2.2 Schroeppel-Shamir ............................................. 23
2.3 4-Tree .............................................................. 25
2.4 k-Tree .............................................................. 28

3.1 Match-and-Filter ................................................ 35
3.2 Approx-k-Tree .................................................... 38
3.3 Gen-Approx-4-Tree ............................................. 42
3.4 Gen-Approx-k-Tree ............................................. 46
3.5 Approx-3-Tree .................................................... 51

4.1 SimpleISD .......................................................... 65
4.2 SimpleISDSolve .................................................. 65
4.3 BCD ................................................................. 69
4.4 AdvancedISD - Framework .................................... 72
4.5 AdvISDSolve ...................................................... 73
4.6 BJMM(m) .......................................................... 78

5.1 BJMM-NN(m) ...................................................... 84
5.2 Depth-2-ISDSolve ................................................ 90
5.3 Depth-m-ISDSolve .............................................. 95
At first sight, probability theory seems a pleasantly intuitive discipline. A simple game like coin tossing has two possible and equally likely outcomes. Thus the probability for each outcome is $\frac{1}{2}$ and after $n$ coin flips one expects each outcome to appear about $\frac{n}{2}$ times. However, common sense often fails in estimating probabilities for more complex events. Probably one of the most famous examples is the birthday paradox or birthday problem. A common variant deals with the question, how many people there have to be in a room such that at least two of them have the same birthday with high probability. Since the chance for two people being born on the same day of the year is $\frac{1}{365} \approx 0.03\%$ and thus pretty low, one would expect that a lot of people have to be in the room. The true result – for only 23 people the probability is already around 50% – is quite surprising but can be obtained easily using probability theory. First of all we have $23 \cdot 22 = 253$ pairs and for each two people the chance of not being born on the same day of the year is $1 - \frac{1}{365} = \frac{364}{365}$. The probability for all 253 pairs to have different birthdays is $\left(\frac{364}{365}\right)^{253} \approx 50\%$. Thus the chance for two of 23 people being born on the same day of the year is also $\approx 50\%$.

**Part I: k-List Problems.** In cryptography we consider a slightly different point of view on this problem. One of the most fundamental tools exactly exploits this birthday paradox. In these so-called birthday attacks we use the fact that a collision on a set of $N$ elements can be found in time $\sqrt{N}$. Analogously to the example above this is small compared to $N$. The analysis of these attacks is of major importance for evaluating secure instantiations of cryptographic problems. In the first part of this thesis (Chapter 2 and 3) we mainly cover generalizations of the birthday problem like the $k$-list problem first described by Wagner in 2002 [Wag02]. It has numerous applications like lattice sieving algorithms [AKS01, KS01] or the generic solving algorithms for subset sum by Howgrave-Graham and Joux [HGJ10] as well as Becker, Coron and Joux [BCJ11]. Further applications are solving algorithms for LPN [LF06, GJL14, ZJW16] and decoding algorithms for linear codes which we introduce separately as another main part of this thesis.
In the $k$-list problem one is given lists $L_1, \ldots, L_k$ with uniform and independent vectors from $\mathbb{F}_2^n$ and has to find a tuple $(x_1, \ldots, x_k) \in L_1 \times \cdots \times L_k$ such that $x_1 + \cdots + x_k = 0^n$ holds over $\mathbb{F}_2^n$. Intuitively a solution exists if $|L_1 \times \cdots \times L_k| \geq 2^n$ holds. For $k = 2$ we obtain the birthday problem again since $x_1 + x_2 = 0^n \iff x_1 = x_2$. Wagner also presented a solving algorithm, called the $k$-tree-algorithm, which finds a solution with special structure in a binary tree wise fashion and thus can be applied if $k$ is a power of two. The algorithm succeeds in expectation whenever $|L_i| \geq 2^{\frac{n}{2^{m+1}}} \cdot \log(k)$ holds for all $i = 1, \ldots, k$. Running time and memory consumption are up to a polynomial factor determined by the list sizes. However, in some applications one cannot guarantee sufficiently large list sizes. In [MS09] Minder and Sinclair’s presented a variant of the $k$-tree-algorithm which covers this case. In a nutshell the algorithm first expands the lists until there are sufficiently many solutions to run the $k$-tree-algorithm. One can adapt both Wagner’s as well as Minder’s and Sinclair’s algorithm to non-powers of two, i.e. to the case $k = 2^m + j, j < 2^m$. However, it is an open problem to improve the running times for this case upon $k = 2^m$.

In Chapter 3 we introduce another variant of the $k$-list problem which allows vectors summing up to small Hamming weight $w$ instead of an exact matching to the zero vector. This new variant is motivated by the fact that requiring exact matching is too strong for some applications and recent advances gained an edge through allowing small Hamming weight. Thus, in our approximate $k$-list problem one is given lists $L_1, \ldots, L_k$ with uniform and independent vectors from $\mathbb{F}_2^n$ again and has to find a tuple $(x_1, \ldots, x_k) \in L_1 \times \cdots \times L_k$ such that $\Delta(x_1 + \cdots + x_k) \leq w$ for some $w > 0$. Intuitively we obtain more solutions for growing weight $w$ which is why this problem should be easier to solve than the original $k$-list problem. There are $n^w$ possibilities to distribute $w$ ones over $n$ coordinates. Thus one can expect to find $n^w$ times as many solutions.

We present different variants of new algorithm which solves the approximate $k$-list problem if $k$ is a power of two. The easiest one simply uses the $k$-tree-algorithm on part of the coordinates only and hopes for the correct Hamming weight on the remaining coordinates. The other variants essentially combine techniques from the original $k$-tree-algorithm with filtering techniques and the nearest neighbor search by May-Ozerov [MO15]. This algorithm gets two lists $L_1, L_2$ as input and finds all pairs $(x_1, x_2) \in L_1 \times L_2$ with small Hamming distance. Although the running time of the nearest neighbor search algorithm increases with growing error $w$, the running time of our algorithm continually decreases for growing $w$ due to the rising number of solutions. We provide parameter sets which yield the lowest running time (resp. memory consumption) as well as parameter sets yielding a closed formula for the complexity.

It is conjectured that the 3-list problem cannot be solved exponentially faster.
than the 2-list problem [KPP14]. However, there exist several attempts which yield polynomial improvements [NS15, BDF17]. In contrast to this and to the k-list problem in general we are able to adapt our algorithm to instances where k is not a power of two but of the form \( k = 2^m + 2^m - 1 \) for some \( m \in \mathbb{N} \). We achieve an exponential speed-up for this case upon \( k = 2^m \) (e.g. \( k = 3 \) upon \( k = 2 \)) whenever \( 0 < w < \frac{n}{2} \). In a nutshell, our algorithm runs on lists of different sizes where two lists of equal size are combined via sort-and-match and the third list is filtered for the specific weight distribution.

**Part II: Information Set Decoding.** One application of k-tree-like algorithms is the decoding problem for random linear codes which is known to be NP-hard [BMvTT8].

It is even robust in the face of quantum computers and is therefore suitable for the construction of quantum-resistant cryptosystems as done by McEliece [McE87], Alekhnovich [Ale03] or Regev [Reg05]. We can define a linear code as a k-dimensional subspace of \( \mathbb{F}_2^n \). In the decoding problem one is given a noisy version \( x = c + e \) of a codeword \( c \) with error vector \( e \in \mathbb{F}_2^n \) of Hamming weight \( \Delta(e) = w \). The target is to find \( e \) and to recover the codeword \( c \). In typical settings the weight \( w \) is either bounded by the code distance \( d \) (full distance decoding) or by \( \frac{d}{2} \) (half distance decoding). The code distance is defined as the minimum Hamming distance between two codewords.

Decoding algorithms typically have a running time \( T(n, k, d) \) as a function of the three parameters \( n, k \) and \( d \). Given a random linear code we can use the Gilbert-Varshamow bound which yields that \( d \) is a function of \( n \) and \( k \) and therefore we can express the running time as a function \( T(n, k) \) of \( n \) and \( k \) only. Furthermore we mostly consider the worst case running time \( T(n) \) over all code rates \( \frac{k}{n} \). Until today, information set decoding algorithms provide the best complexities for decoding random linear codes. They offer exponential running times of the form \( T(n) = 2^{\tau n} \) where \( \tau \) is a constant which can be used as a metric to compare the different algorithms. Figure 1 illustrates the improvements over the last fifty years which are more significant for full distance decoding compared to half distance decoding.

![Fig. 1: Comparison of \( \tau \) for different algorithms in the full distance (top) and half distance (bottom) decoding setting.](image-url)
In this thesis we present most of the mentioned results, starting with the first information set decoding algorithm by Prange [Pra62] from 1962. In a nutshell, the algorithm applies linear algebra and reduces the dimension of the search space for $e$ which is then found via brute-force. This algorithm was improved by Stern [Ste88] and later by Bernstein, Lange and Peters [BLP11] employing meet-in-the-middle techniques. These algorithms also introduced a fixed weight distribution to the target vector $e$ where some coordinates are zero or have fixed weight. An outer loop applies random permutations to obtain this weight distribution while the solving algorithm exactly finds fitting vectors. In 2011, May, Meurer and Thomae [MMT11] used combinatorial representation techniques and a binary search tree to further improve the algorithm. Becker, Joux, May and Meurer [BJMM12] refined these techniques for decoding resulting in their BJMM-algorithm. Representations were originally introduced by Howgrave-Graham and Joux [HJ10] to improve generic algorithms for the well-known subset sum problem. The binary search tree works similar to the $k$-tree-algorithm by Wagner. In 2015, May and Ozerov [MO15] applied some nearest neighbor search leading to a speed-up for the BJMM. While their algorithm uses a binary search tree of depth 3, we show in Chapter 5, that depth 4 is in fact optimal for full distance decoding. Moreover we present a new algorithm which yields the currently best complexities for both full distance and half distance decoding. Since we heavily use nearest neighbor search on every layer, the improvement upon previous works is more significant for the high error regime of full distance decoding.

The so-called learning parity with noise (LPN) problem is closely related to the decoding problem. In the LPN problem one is given $n$ samples of the form $(a_i, \langle a_i, s \rangle + e_i)$ for $s \in \mathbb{F}_2^k$, $a_i \in \mathbb{F}_2^k$ and some error $e_i$ with $\Pr[e_i = 1] = \tau$. The target is to recover the secret $s$. Every such instance can be translated naturally to a decoding problem for a random linear code. The best way to solve large instances in practice is a hybrid approach introduced by Esser, Kübler and May in [EKM17]. In a nutshell, one first applies a dimension reduction algorithm such as the BKW-algorithm [BKW00]. This results in a new instance with large error close to $\frac{1}{2}$ which is then solved by a decoding algorithm. However this is the perfect setting for our algorithm which performs especially well for high errors.

**Overview.** We conclude the introduction with an overview of this thesis.

- Chapter 1 starts with some general notation and definitions. We furthermore introduce basic algorithms which we use as subroutines throughout this thesis. We also devise some Lemmas which provide running times and memory consumptions for these algorithms.
• Chapter 2 introduces the \( k \)-list problem as well as the \( k \)-tree-algorithm by Wagner [Wag02]. We cover several variations and conclude with some applications.

• Chapter 3 is mainly based on a joint work with Alexander May [BM17a]. We introduce the approximate \( k \)-list problem and present as well as analyze different solving algorithms. We conclude with an application – the parity check problem – and a short discussion on open problems.

• In Chapter 4 we introduce the syndrome decoding problem and give an overview of different information set decoding algorithms. We start with Prange’s algorithm [Pra62] and continue with Stern’s algorithm [Ste88] as well as the variant by Bernstein, Lange and Peters [BLP11]. We then describe the information set decoding algorithm by Finiasz and Sendrier [FS09] before we introduce representation techniques and the BJMM-algorithm [BJMM12].

• Finally, Chapter 5 connects state-of-the-art information set decoding algorithms with the (approximate) \( k \)-tree-algorithm and nearest neighbor search. We start with the improved BJMM variant by May and Ozerov [MO15] which was later optimized in another joint work with Alexander May [BM17b]. We furthermore describe a new decoding algorithm introduced in [BM18] which is a joint work with Alexander May again. Next, we analyze the complexity of this algorithm in attacks on typical instances of the McEliece cryptosystem as well as the LPN problem. For the latter we especially consider the hybrid-algorithm by Esser, Kübler and May [EKM17]. We conclude again with a discussion on open problems.
Chapter 1

Preliminaries

In this section we present some general notation and definitions. We also discuss how the complexity of algorithms is analyzed in this thesis and present a collection of simple algorithms which are used as subroutines later.

1.1 Notation

We denote by $F_2$ the finite field with 2 elements, i.e. $F_2 := \{0, 1\}$. Throughout this thesis vectors are denoted by bold letters, e.g. $x \in F_2^n$ while matrices are denoted by capital letters, e.g. $G \in F_2^{k \times n}$. Thus the vector $x$ can also be interpreted as a bit string of length $n$. In contrast to matrices, $L \subset F_2^n$ denotes a list of vectors from $F_2^n$. The Hamming weight $\Delta(x)$ of a vector $x \in F_2^n$ is defined as the number of nonzero entries of $x$. For two vectors $x, y \in F_2^n$ we define the Hamming distance as $\Delta(x, y) := \Delta(x + y)$. In this case the operator $+$ denotes the bitwise XOR while for $x + y, x, y \in \mathbb{R}^n$ it denotes the addition over $\mathbb{R}^n$. For convenience we furthermore denote the all-zero vector by $0 := 0^n$ and we do not distinguish between column and row vectors, i.e. $F_2^n = F_2^{n \times 1} = F_2^{1 \times n}$. The logarithm to base 2 is denoted by $\log(x) := \log_2(x)$. We write $x \in_R S$ if we choose $x$ uniformly from a finite set $S$. For any $x > 0$ we define the set $\lfloor x \rfloor := \{0, 1, 2, \ldots, \lfloor x \rfloor\}$.

Projections. Let $x \in F_2^n$ be a vector with $n$ coordinates and $\ell := (\ell_1, \ldots, \ell_m) \in \mathbb{N}^m$ with $\sum_{i=1}^m \ell_i = n$. We denote by $x^{(i)}$ the $i$-th entry of $x$ for $i = 1, \ldots, n$ and by $x^{[j]}$ the projection of $x$ on its $j$-th block of size $\ell_j$ for $j = 1, \ldots, m$ (see Fig. 1.1), i.e.

$$x^{[j]} := (x^{(1 + \sum_{i=1}^{j-1} \ell_i)}, \ldots, x^{(\sum_{i=1}^{j} \ell_i)}).$$

For a set of indices $I \subset [n] \setminus \{0\}$ we extend this notation to projections onto more than one block ($x_I \in F_2^{\sum_{i \in I} \ell_i}$) and to projections for lists ($L_{[j]} \in F_2^{\ell_j}$).
Example 1.1. Let \( x := 11010101 \in \mathbb{F}_8^2 \). Then we define by \( \ell := (2, 2, 1, 3) \) the projections

\[ x_{[1]} = 11, \quad x_{[2]} = 01, \quad x_{[3]} = 0, \quad x_{[4]} = 101. \]

Checksums. Let \((x_1, \ldots, x_t) \in \mathbb{F}_2^n \times \cdots \times \mathbb{F}_2^n\) be a tuple of vectors. We define its checksum by \((x_1, \ldots, x_t)^{+ n} := \sum_{i=1}^t x_i\). We extend this notation again to lists of tuples \( L \subset (\mathbb{F}_2^n)^t \). For convenience we furthermore flatten tuples of tuples, i.e. \(((x_1, \ldots, x_t), \ldots, (x_{(s-1)t+1}, \ldots, x_{st})) = (x_1, \ldots, x_{st})\).

Example 1.2. Let \( x_1 := 11010101 \) and \( x_2 := 01110100 \in \mathbb{F}_8^2 \). Then we have

\[ (x_1, x_2)^{+ n} = x_1 + x_2 = 10100001. \]

We moreover introduce the following technical tools which are used extensively throughout this thesis.

Definition 1.1 (Binary Entropy Function). We define the binary entropy function \( H : [0, 1] \to [0, 1] \) as

\[
H(x) := \begin{cases} 
-x \log(x) - (1-x) \log(1-x) & x \in (0,1) \\
0 & \text{else}
\end{cases}
\]

Note that this function is symmetric around \( x = \frac{1}{2} \) and strictly concave (see Fig. 1.2).

We also define the 'right inverse' of \( H(x) \) as

\[ H^{-1}(y) := \text{argmin}_{x \in [0,1]} \{ H(x) = y \}. \]

For our analysis we use the following standard estimation which can be found in [MU05].

Lemma 1.1. For \( n \in \mathbb{N} \) and \( k \in [n] \) it holds that

\[ \frac{2^{H(k/n)}}{n+1} \leq \binom{n}{k} \leq 2^{H(k/n)}. \]
1.2 Algorithms

Since most algorithms in this thesis are probabilistic or operate on a random input we provide an average case analysis of them. Consider some algorithm $A$ which is used to solve some mathematical problem. Given an instance $I$ as input it returns a set of solutions $S$ for the problem. We say that $A$ in expectation solves the problem in time $T$ if the following three conditions hold.

- $A$ is given a uniformly random input $I$,
- terminates in expected time $T$
- and finds at least one solution in expectation, that is $\mathbb{E}[|S|] \geq 1$.

The probability space is taken over the instance $I$ of the problem and $A$’s internal coin tosses.

We furthermore neglect polynomial factors in our analysis because our algorithms have exponential complexities as a function of some sufficiently large input parameter $n$. Employing commonly used notation, $2^{\lambda n}$ polynomial time operations take time $2^{(\lambda+\varepsilon)n}$ while storing a list of $2^{\lambda n}$ vectors uses $2^{(\lambda+\varepsilon)n}$ memory for any constants $\varepsilon, \lambda > 0$. The $\lambda$ may be a function of some other parameters beside $n$ but constant in $n$. The term $\varepsilon$ takes account of polynomial factors. However, some complexities are going to be the result of numerical optimizations. In this case we state complexities in the form $2^{\lambda n}$ where $\lambda$ is a floating point number rounded up to some digits after the decimal point. Here, rounding takes care of polynomial factors omitted in the analysis to provide an asymptotic upper bound. Throughout this thesis we always assume that $n$ is large and therefore the terms $\lambda n$ are integers.
Below we present several simple algorithms and provide running times, memory consumption and output sizes. Since we need those algorithms as subroutines in different settings throughout this thesis, we present them in a general form. We sacrifice simplicity in this section but benefit later when we come to more complex algorithms.

**Sort-and-Match-Algorithm.** The first algorithm presented in this section is used to find equal elements of length \( n \) in two lists \( L_1, L_2 \). This is essentially achieved through sorting the first list \( L_1 \) as a first step and then using binary search for all elements in \( L_2 \). Intuitively, this can be done in time \( \max\{|L_1|, |L_2|, |L_1 \times L_2| \cdot 2^{-n}\} \) returning \(|L_1 \times L_2| \cdot 2^{-n}\) elements. Algorithm 1.1 presents a generalized version allowing us to find elements which match on their projection only or which add up to some fixed vector (also see Fig. 1.3).

![Fig. 1.3: The Sort-and-Match-algorithm on 3 coordinates.](image)

**Algorithm 1.1: Sort-and-Match**

Global: \( n \in \mathbb{N}, \ell = (\ell_1, \ldots, \ell_m) \in \mathbb{N}^m \) with \( \sum_{i=1}^{m} \ell_i = n \)

Input: \( L_1, L_2 \subset (\mathbb{F}_2^n)^t, I \subset [m] \setminus \{0\}, y \in \{0, 1\}^{\delta n} \) with \( \delta := \frac{1}{n} \sum_{i \in I} \ell_i \)

Output: \( L \subset (\mathbb{F}_2^n)^t \times (\mathbb{F}_2^n)^t \)

1. Sort list \( L_1 \) with regard to its projection \( (L_1)^+_{|I|} \)
2. for \( x_2 \in L_2 \) do
   1. if \( \exists x_1 \in L_1 : (x_1 + x_2)^+_{|I|} = y \) then \( L \leftarrow L \cup \{(x_1, x_2)\} \)
3. return \( L \)

**Lemma 1.2.** Let \( n, t \in \mathbb{N}, \ell = (\ell_1, \ldots, \ell_m) \in \mathbb{N}^m \) with \( \sum_{i=1}^{m} \ell_i = n \), \( I \subset [m] \setminus \{0\} \), \( \delta := \frac{1}{n} \sum_{i \in I} \ell_i \) and \( \lambda \in [0, 1] \). For any constant \( \varepsilon > 0 \), given two lists \( L_1, L_2 \subset (\mathbb{F}_2^n)^t \) of equal size \( 2^{\lambda n} \) with uniform and independent vectors, Sort-and-Match outputs a list
of expected size $2^{(2\lambda-\delta)n}$ containing pairs $(x_1, x_2) \in L_1 \times L_2$ satisfying $(x_1 + x_2)^+_{[I]} = y$.

The expected running time and memory consumption are $2^{(\max\{\lambda, 2\lambda - \delta\} + \varepsilon)n}$.

**Proof.** Let $S_{i,j} \in \{0, 1\}$ be a random variable which is equal to 1 if and only if the $i$-th element in $L_1$ and the $j$-th element in $L_2$ satisfy the condition in step 2 of Alg. 1.1. Then

$$E[S_{i,j}] = \Pr[S_{i,j} = 1] = \frac{\Pr[(x_1 + x_2)^+_{[I]} = y]}{|L_1| \cdot |L_2|} = 2^{-\delta n}$$

since the elements are uniformly chosen and therefore their checksum is a uniform vector. Thus we have

$$E[|L|] = \sum_{i,j} E[S_{i,j}] = |L_1| \cdot |L_2| \cdot 2^{-\delta n} = 2^{(2\lambda-\delta)n}.$$

The sorting in step 1 takes time $2^{(\lambda + \varepsilon)n}$ while the binary searches in step 2 can be done in expected time $2^{(\max\{\lambda, 2\lambda - \delta\} + \varepsilon)n}$ for any constant $\varepsilon > 0$. This results in the claimed expected running time. The memory consumption is determined by the list sizes. \hfill \Box

**Filter-Algorithm.** This algorithm filters the input list $L$ for elements with some specific Hamming weight $w$ (also see Fig. 1.4 for the generalized variant). Intuitively this can be done in time $|L|$ returning a list containing $\binom{n}{w} \cdot |L|$ elements.

![Fig. 1.4: The Filter-algorithm for weight 2 on 3 fixed coordinates.](image)

**Algorithm 1.2: Filter**

<table>
<thead>
<tr>
<th>Global</th>
<th>$n \in \mathbb{N}$, $\ell = (\ell_1, \ldots, \ell_m) \in \mathbb{N}^m$ with $\sum_{i=1}^m \ell_i = n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>$L \subset (\mathbb{F}_2^n)^t$, $I \subset [m]\setminus{0}$, $w \in \mathbb{N}$.</td>
</tr>
<tr>
<td>Output</td>
<td>$L' \subset (\mathbb{F}_2^n)^t$</td>
</tr>
<tr>
<td>1 $L' \leftarrow \emptyset$</td>
<td></td>
</tr>
<tr>
<td>2 for $x \in L$ do</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if $\Delta(x_{[I]}^+_{[N]}) \leq w$ then $L' \leftarrow L' \cup {x}$</td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>return $L'$</td>
<td></td>
</tr>
</tbody>
</table>
Lemma 1.3. Let \( n, t \in \mathbb{N} \), \( \ell = (\ell_1, \ldots, \ell_m) \in \mathbb{N}^m \) with \( \sum_{i=1}^m \ell_i = n \), \( I \subset [m] \setminus \{0\} \), \( \delta := \frac{1}{n} \sum_{i \in I} \ell_i \), \( \lambda \in [0, 1] \) and let \( \gamma \in [0, \frac{1}{2}] \) with \( \gamma n \in \mathbb{N} \). For any constant \( \varepsilon > 0 \), given a list \( L \subset (\mathbb{F}_2^m)^t \) of size \( 2^{\lambda n} \) with uniform and independent vectors, FILTER outputs a list \( L' \) containing vectors \( x \in L \) satisfying \( \Delta(x_i^n) \leq \gamma n \) where

\[
\frac{2^{(\lambda + H(\frac{\gamma}{2})\delta - \delta) n}}{\text{poly}(n)} \leq \mathbb{E}[|L'|] \leq 2^{(\lambda + H(\frac{\gamma}{2})\delta - \delta + \varepsilon) n}.
\]

The running time and memory consumption are \( 2^{(\lambda + \varepsilon)n} \).

Proof. The running time and memory consumption are determined by the number of iterations \( |L| \) and the polynomial time operation in each step resulting in the claimed complexity. Note that we have \( |L'| \leq |L| \) and thus the output list \( L' \) does not influence the overall complexity. Let \( S_j \in \{0, 1\} \) be a random variable which is equal to 1 if and only if the \( j \)-th element in \( L \) satisfies the condition in step 2 of Alg. 1.2. Since its checksum is a uniform vector and using Lemma 1.1 we have

\[
\mathbb{E}[S_j] = Pr[S_j = 1] = \sum_{k=0}^{\gamma n} \binom{\delta n}{k} \cdot 2^{-\delta n} \geq \binom{\delta n}{\gamma n} \cdot 2^{-\delta n} \geq \frac{2^{H(\frac{\gamma}{2})-1} \delta n}{\delta n + 1}
\]

and thus

\[
\mathbb{E}[|L'|] = \mathbb{E} \left[ \sum_j S_j \right] = \sum_j \mathbb{E}[S_j] \geq |L| \cdot \frac{2^{H(\frac{\gamma}{2})-1} \delta n}{\delta n + 1} = \frac{2^{(\lambda + H(\frac{\gamma}{2})\delta - \delta) n}}{\text{poly}(n)}.
\]

Analogously we can show that for any constant \( \varepsilon > 0 \)

\[
\mathbb{E}[S_j] = \sum_{k=0}^{\gamma n} \binom{\delta n}{k} \cdot 2^{-\delta n} \leq O(n) \cdot \binom{\delta n}{\gamma n} \cdot 2^{-\delta n} \leq 2^{(\lambda + H(\frac{\gamma}{2})\delta - \delta + \varepsilon) n}
\]

since \( \binom{\delta n}{k} \) is strictly increasing in \( k \) for \( k \leq \gamma n \). \( \square \)

Nearest Neighbor Search. Given two lists \( L_1, L_2 \) a nearest neighbor search finds elements \( x_i \in L_i \), \( i = 1, 2 \) with some upper bounded Hamming distance, i.e. \( \Delta(x_1, x_2) \leq w \). A naive algorithm (see. Alg. 1.3) simply tests the distance of all pairs \( (x_1, x_2) \in L_1 \times L_2 \) in time \( |L_1 \times L_2| \).
Algorithm 1.3: NN-Enumerate-Pairs

Input : $L_1, L_2 \subset F_2^n$, $w \in \mathbb{N}$.
Output : $L \subset F_2^n \times F_2^n$

$L \leftarrow \emptyset$

1 for $(x_1, x_2) \in L_1 \times L_2$ do
   if $\Delta(x_1, x_2) \leq w$ then
      $L \leftarrow L \cup \{(x_1, x_2)\}$
   end

return $L$

Applying a meet-in-the-middle technique (see Alg. 1.4) results in a classic time-memory trade-off where the running time is reduced at the cost of a higher memory consumption as we show next. Intuitively, the running time is determined by the list size $|L_2'| = (\frac{n}{2}) \cdot |L_1|$ and the output list size $(\frac{n}{2}) \cdot |L_1 \times L_2|$.

Algorithm 1.4: NN-Meet-in-the-Middle

Input : $L_1, L_2 \subset F_2^n$, $w \in \mathbb{N}$.
Output : $L \subset F_2^n \times F_2^n$

$L, L_2' \leftarrow \emptyset$

1 for $x_2 \in L_2$, $e \in F_2^n$ with $\Delta(e) \leq \frac{w}{2}$ do
   $L_2' \leftarrow L_2' \cup \{(x_2 + e, x_2)\}$

end

2 for $x_1 \in L_1$, $e \in F_2^n$ with $\Delta(e) \leq \frac{w}{2}$ do
   if $(x_1 + e, x_2) \in L_2'$ then
      $L \leftarrow L \cup \{(x_1, x_2)\}$
   end

return $L$

A more efficient way to find nearest neighbors uses projections and was introduced in [MO15]. Throughout this thesis we use their algorithm as a black box.

Theorem 1.1 (May-Ozerov [MO15]). Let $n \in \mathbb{N}$, $\lambda \in [0, 1 - H(\frac{1}{2})]$ and $\gamma \in [0, \frac{1}{2}]$ with $\gamma n \in \mathbb{N}$. For any constant $\varepsilon$, given two lists $L_1, L_2 \subset F_2^n$ of equal size $2^{\lambda n}$ with uniform and independent vectors, one can find all but a negligible fraction of pairs $(x_1, x_2) \in L_1 \times L_2$ satisfying $\Delta(x_1, x_2) \leq \gamma n$ in time

$$2^{(\tau(\lambda, \gamma) + \varepsilon)n}, \text{ where } \tau(\lambda, \gamma) := (1 - \gamma) \left(1 - H \left(\frac{H^{-1}(1 - \lambda) - \frac{\gamma}{2}}{1 - \gamma}\right)\right).$$

We would like to point out that May and Ozerov originally showed the theorem for the case $\Delta(x_1, x_2) = \gamma n$ in [MO15]. Running the May-Ozerov nearest neighbor search
for all \( O(n) \) integers 0, 1, \ldots, \( \gamma n \) yields the generalized version in Theorem 1.1. Since the function \( y \) is strictly increasing in \( \gamma \) we only get a polynomial overhead of \( O(n) \) which adds to the constant \( \varepsilon \).

Note that Theorem 1.1 cannot be applied for parameters which do not satisfy the condition \( \lambda < 1 - H(\frac{\delta}{2}) \). If that is the case we choose one of the simple nearest neighbor searches Alg. 1.3 or 1.4. Analogously to the previous algorithms we generalize the nearest neighbor search to projections and checksums (see Fig. 1.5). The following algorithm combines all nearest neighbor search algorithms and is used as a subroutine throughout the thesis. Figure 1.6 compares the running times for the different algorithms.

![Image](https://via.placeholder.com/150)

**Fig. 1.5:** The NN-Search-algorithm on 3 fixed coordinates for weight \( \leq 1 \).

**Algorithm 1.5: NN-Search**

<table>
<thead>
<tr>
<th>Global</th>
<th>: ( n \in \mathbb{N}, \ell = (\ell_1, \ldots, \ell_m) \in \mathbb{N}^m ) with ( \sum_{i=1}^{m} \ell_i = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>: ( L_1, L_2 \subset (\mathbb{F}_n^2)^t, I \subset [m]{0}, w \in \mathbb{N} ).</td>
</tr>
<tr>
<td>Output</td>
<td>: ( L \subset (\mathbb{F}_n^2)^t \times (\mathbb{F}_n^2)^t )</td>
</tr>
</tbody>
</table>

1. Choose the fastest nearest neighbor algorithm to determine the set \( L \leftarrow \{ (x_1, x_2) \in L_1 \times L_2 \mid \Delta((x_1 + x_2)^t_{[I]^t}) \leq w \} \)

   return \( L \)

**Lemma 1.4.** Let \( n, t \in \mathbb{N}, \ell = (\ell_1, \ldots, \ell_m) \in \mathbb{N}^m \) with \( \sum_{i=1}^{m} \ell_i = n, I \subset [m]\{0\} \), \( \delta := \frac{1}{n} \sum_{i \in I} \ell_i, \lambda \in [0, 1] \) and \( \gamma \in [0, \frac{1}{2}] \) with \( \gamma n \in \mathbb{N} \). For any constant \( \varepsilon > 0 \) , given two lists \( L_1, L_2 \subset (\mathbb{F}_n^2)^t \) of equal size \( 2^\lambda n \) with uniform and independent vectors, NN-Search outputs a list \( |L| \) of pairs \( (x_1, x_2) \in L_1 \times L_2 \) satisfying \( \Delta((x_1 + x_2)^t_{[I]^t}) \leq \gamma n \) where

\[
\frac{2(2^{2\lambda + H(\frac{\delta}{2})\delta - \delta})n}{\text{poly}(n)} \leq \mathbb{E}[|L|] \leq 2^{(2^{2\lambda + H(\frac{\delta}{2})\delta - \delta} + \varepsilon)n}.
\]
The expected running time is upper bounded by $2^{\tau_{\text{NN}}(\lambda, \delta, \gamma) + \epsilon} n$ where

$$
\tau_{\text{NN}}(\lambda, \delta, \gamma) := \begin{cases} 
\tau(\frac{\lambda}{2}, \frac{\gamma}{2}) & \text{if } \frac{\tau(\frac{\lambda}{2}, \frac{\gamma}{2})}{2} < \frac{\lambda}{2} < 1 - H(\frac{\gamma}{2}) \\
\min \left\{ 2\lambda, \max \left\{ H(\frac{\gamma}{2})\delta + \lambda, 2\lambda + H(\frac{\gamma}{2})\delta - \delta \right\} \right\} & \text{else}
\end{cases}
$$

with $\tau(\cdot, \cdot)$ defined as in Theorem 1.1.

**Proof.** The bounds for the expected number of solutions $E[|L|]$ are determined analogously to Lemma 1.3.

Algorithm 1.3 simply enumerates all pairs in $L_1 \times L_2$ and checks for the correct Hamming distance. This can clearly be done in time $2^{(2\lambda+\epsilon)n}$. Moreover, Alg. 1.4 runs $2^{\lambda n} \cdot (\frac{\delta n}{2})$ iterations of polynomial time operations in steps 1 and 2. Combined with the expected size of the output list $|L|$ this yields a running time of

$$
\max \left\{ 2^{(H(\frac{\gamma}{2})\delta + \lambda + \epsilon)n}, 2^{(2\lambda + H(\frac{\gamma}{2})\delta - \delta + \epsilon)n} \right\}.
$$

For $\frac{\lambda}{2} < 1 - H(\frac{\gamma}{2})$ we can use May-Ozerov which yields the running time from Theorem 1.1. However, in the case $\frac{\tau(\frac{\lambda}{2}, \frac{\gamma}{2})}{2} > \frac{\lambda}{2}$ we use the more efficient Alg. 1.3. For $\frac{\lambda}{2} \geq 1 - H(\frac{\gamma}{2})$ we again choose the more efficient algorithm for the given parameters. □

![Fig. 1.6: Running time exponents $\log(T)/n$ for May-Ozerov (solid), NN-Enumerate Pairs (dotted) and NN-Meet-in-the-Middle (dashed).](image)
Chapter 2

The $k$-List Problem

This chapter contains basic combinatorial problems forming the starting point on which our work is based. In Section 2.1 we introduce the most basic one, namely the birthday problem, being one of the best-known problems in cryptology. It is followed by its generalization, the $k$-list problem, in Section 2.2.

2.1 Origins: The Birthday Problem

The birthday problem is one of the most famous problems in cryptography. In a nutshell, one has to find collisions between random elements. Intuitively, if there are a total of $N$ elements, a collision can be expected after around $\sqrt{N}$ random elements have been chosen. In this work we use the following definition.

Definition 2.1 (Birthday Problem). Let $L_1, L_2 \subset \mathbb{F}_2^n$ be two lists with uniform and independent vectors. Given $L_1, L_2$, one has to find two elements $x_1 \in L_1$ and $x_2 \in L_2$ satisfying

$$x_1 + x_2 = 0.$$  

(2.1)

A variant of this problem comes with only one list $L$ where ones has to find two distinct elements $x_1, x_2 \in L$ satisfying Eq. 2.1. The following simple algorithm finds all existing solutions to the birthday problem for given lists $L_1, L_2$. Note that this is a special case of Sort-and-Match (Alg. 1.1) with $t = 1$, $\ell = (n)$, $I = \{1\}$ and $y = 0$.

20
Algorithm 2.1: Simple Sort-and-Match

Input : \( L_1, L_2 \subset F_2^n \)
Output : \( L \subset F_2^n \times F_2^n \)

1 Sort list \( L_1 \)
2 for \( x \in L_2 \) do
   if \( x \in L_1 \) then \( L \leftarrow L \cup \{x\} \)
end
return \( L \)

The following proposition shows that the birthday problem can be solved in expectation by above algorithm if the lists \( L_1, L_2 \) are sufficiently large, which is the case if \( |L_1 \times L_2| \geq 2^n \).

**Proposition 2.1.** For any constant \( \varepsilon > 0 \) the birthday problem can in expectation be solved in time \( T = 2^{(\frac{1}{2} + \varepsilon)n} \).

**Proof.** Let \( |L_1| = |L_2| = 2^\frac{n}{2} \). The probability for two random elements \((x_1, x_2) \in L_1 \times L_2\) to form a solution of the birthday problem is \( 2^{-n} \). Since Eq. 2.1 is equivalent to \( x_1 = x_2 \), Alg. 2.1 finds all existing solutions. Analogously to the proof of Lemma 1.2 we can show that the expected number of solutions is

\[
E[|L|] = \frac{2^\frac{n}{2} \cdot 2^\frac{n}{2}}{2^n} = 1
\]

i.e. one solution exists in expectation. Thus, for any constant \( \varepsilon > 0 \), Alg. 2.1 solves the birthday problem in expected time \( T = 2^{(\frac{1}{2} + \varepsilon)n} \) as this is the running time for both the sorting in step 1 and the \( |L_2| = 2^\frac{n}{2} \) binary searches in step 2.

In literature, Alg. 2.1 is often referred to as the merge-join operation. Another possibility is a hash-join where one list is stored in a hash table and the other list is scanned for elements in the hash table. This requires \( |L_1| + |L_2| \) computation steps, slightly less than the \( O(n \log n) \), \( n = \max(|L_1|, |L_2|) \) steps for merge-join. Nonetheless the asymptotic complexity is the same for equal list sizes.

**Applications.** There are numerous applications of the birthday problem. An important one is to find collisions for a hash function \( h : F_2^n \rightarrow F_2^n \), i.e. to find \( x_1 \neq x_2 \) such that \( h(x_1) = h(x_2) \). We simply store \( 2^\frac{n}{2} \) elements of the form \( h(\cdot) \) for different inputs in each lists \( L_1 \) and \( L_2 \). If we assume that \( h \) behaves like a random function, the lists contain elements chosen uniformly and independently and we have created an instance of the birthday problem. Following Proposition 2.1 we can find a collision for the hash function \( h \) in time \( 2^{(\frac{1}{2} + \varepsilon)n} \) for any constant \( \varepsilon > 0 \).
2.2 Introducing the $k$-List Problem

The following generalized version of the birthday problem was introduced by Wagner in 2002 [Wag02]. His $k$-list problem or generalized birthday problem allows an arbitrary number $k$ of initial lists searching for $k$ vectors summing up to $0$.

**Definition 2.2 ($k$-List Problem).** Let $L_1, \ldots, L_k \subset \mathbb{F}_2^n$ be $k$ lists with uniform and independent vectors. Given those lists, one has to find elements $(x_1, \ldots, x_k) \in L_1 \times \ldots \times L_k$ satisfying

$$x_1 + \ldots + x_k = 0. \quad (2.2)$$

Analogously to the birthday problem a solution exists in expectation if $|L_1 \times \ldots \times L_k| \geq 2^n$ or $|L_i| = 2^n$, $i = 1, \ldots, k$ for balanced lists which is therefore a lower bound for the complexity of this problem. Furthermore this problem can be viewed as a single list problem again, where we search for $k$ distinct elements summing up to the zero vector. For $k = 2^m$ with $m \in \mathbb{N}$ the $k$-List problem obviously can be reduced to the birthday problem and thus be solved in time $2^{(1+\varepsilon)n}$ for any constant $\varepsilon > 0$. In a nutshell, a naive algorithm uses a meet-in-the-middle approach and combines the first (resp. last) $\frac{k}{2}$ lists to one list containing all sums $x_1 + \ldots + x_{\frac{k}{2}}$ (resp. $x_{\frac{k}{2}+1} + \ldots + x_k$) resulting in an instance of the birthday problem with two lists of size $2^{\frac{n}{2}}$ containing elements chosen uniformly and independently.

The Schroeppel-Shamir-Algorithm. Long before Wagner defined the $k$-List problem, Schroeppel and Shamir [SS81] already described a method (see Fig. 2.1) to solve the $k = 4$ case, i.e. find vectors $(x_1, x_2, x_3, x_4) \in L_1 \times L_2 \times L_3 \times L_4$ satisfying

$$x_1 + x_2 + x_3 + x_4 = 0.$$

Their algorithm (see also Alg. 2.2) starts with four lists $L_1^{(0)}, L_2^{(0)}, L_3^{(0)}, L_4^{(0)}$ of size $2^n$. In the first step **Sort-and-Match** on the lists $L_1^{(0)}, L_2^{(0)}$ (resp. $L_3^{(0)}, L_4^{(0)}$) finds all pairs of vectors adding up to some vector $y \in \mathbb{F}_2^n$ on the first $\frac{n}{2}$ bits. Those pairs are stored in two new lists

$$L_1^{(1)} = \{(x_1, x_2) \in L_1^{(0)} \times L_2^{(0)} \mid (x_1 + x_2)[1] = y\}$$
$$L_2^{(1)} = \{(x_3, x_4) \in L_3^{(0)} \times L_4^{(0)} \mid (x_3 + x_4)[1] = y\}.$$

In the second step another **Sort-and-Match** on those two lists finds vectors which add up to zero on the last $\frac{n}{4}$ bits while the first bits add up to zero.
The output is a list

\[ L_1^{(2)} = \{(x_1, x_2, x_3, x_4) \in L_1^{(1)} \times L_2^{(1)} \mid \sum_{j=1}^{4} x_j = 0\} \]

of solutions to the 4-list problem. These two steps are repeated for every \( y \in F_2^{n/4} \) eventually finding a solution. The following theorem shows that the 4-List problem can be solved by Alg. 2.2 in expectation as fast as with the meet-in-the-middle-approach but with a lower memory consumption.

**Algorithm 2.2: Schroepel-Shamir**

| Input | \( L_j^{(0)} \subset F_2^n, |L_j^{(0)}| = 2^{\frac{3n}{4}} \) for \( j = 1, 2, 3, 4 \) |
|-------|--------------------------------------------------|
| Output| \( (x_1, x_2, x_3, x_4) \in F_2^n \times F_2^n \times F_2^n \times F_2^n \) |
| \( \ell = \left(\frac{n}{4}, \frac{3n}{4}\right) \) | for \( y \in F_2^{n/4} \) do |
| 1 | \( L_1^{(1)} \leftarrow \text{Sort-And-Match}(L_1^{(0)}, L_2^{(0)}, 1, y) \) |
| 1 | \( L_2^{(1)} \leftarrow \text{Sort-And-Match}(L_3^{(0)}, L_4^{(0)}, 1, y) \) |
| 2 | \( L_1^{(2)} \leftarrow \text{Sort-And-Match}(L_1^{(1)}, L_2^{(1)}, 2, 0) \) |
| | if \( |L_1^{(2)}| > 0 \) then return some \((x_1, x_2, x_3, x_4) \in L_1^{(2)}\) |
| | end |
| | return \( \perp \) |

**Theorem 2.1.** For any constant \( \varepsilon > 0 \) the 4-list problem can in expectation be solved in time \( T = 2^{(\frac{3}{4} + \varepsilon)n} \) using \( M = 2^{(\frac{1}{4} + \varepsilon)n} \) memory.
Proof. Let $S_0 := |L_j^{(0)}| = 2^{\frac{j}{4}}$, $j = 1, 2, 3, 4$. By Lemma 1.2 we have

$$S_1 := \mathbb{E}[|L_j^{(1)}|] = \frac{S_0^2}{2\pi} = 2^{\frac{j}{4}}, \quad j = 1, 2 \quad \text{and} \quad \mathbb{E}[|L^{(2)}|] = \frac{S_1^2}{2\pi} = 2^{\frac{j}{4}}.$$ 

Thus the memory consumption of algorithm 2.2 as well as the running time of step 1 and 2 is $2^{\left(\frac{j}{4} + \varepsilon\right)n}$ for any $\varepsilon > 0$ using Lemma 1.2 again. We brute force over all $y \in \mathbb{F}_{2^n}^2$ adding a factor of $2^{\frac{j}{4}}$ to the running time. We obtain a total expected running time $T = 2^{\left(\frac{j}{4} + \varepsilon\right)n}$ and memory consumption $M = 2^{\left(\frac{j}{4} + \varepsilon\right)n}$.

A solution to the 4-list problem exists in expectation if $|L_1^{(0)} \times L_2^{(0)} \times L_3^{(0)} \times L_4^{(0)}| \geq 2^n$ which is satisfied. For every such solution $(x_1, x_2, x_3, x_4) \in L_1^{(0)} \times L_2^{(0)} \times L_3^{(0)} \times L_4^{(0)}$ we have

$$x_1 + x_2 + x_3 + x_4 = 0 \iff x_1 + x_2 = x_3 + x_4 \Rightarrow (x_1 + x_2)_{[1]} = (x_3 + x_4)_{[1]} = y$$

for some $y \in \mathbb{F}_{2^n}^2$. Thus in one of the iterations the solution survives step 1 i.e. $(x_1, x_2) \in L_1^{(1)}$ and $(x_3, x_4) \in L_2^{(1)}$ as well as step 2. Hence, Alg. 2.2 finds all existing solutions.

In the following section we see a more efficient way to solve the $k$-List Problem for $k \geq 4$, especially if $k$ is a power of two.

### 2.3 The $k$-Tree Algorithm

Besides being the first who defined the generalized birthday problem, Wagner also brought up a new algorithm, namely the $k$-tree-algorithm which is still the current state-of-the-art for solving this problem. In a nutshell, the initial lists $L_1, \ldots, L_k$ are combined in a tree-wise fashion using SORT-AND-MATCH, finding partially matching vectors at first, leading to special structured solutions of the $k$-list problem at the end. This structure allows them to be found more efficiently than in previous approaches. However, the algorithm only finds this special solutions, possibly throwing away any other solutions.

**The Case $k = 4$.** Before describing the general version of the algorithm we look at the simplest case. The reader is advised to follow the description via Fig. 2.2.

We first introduce a split into $\ell_1$ and $\ell_2 := n - \ell_1$ coordinates to all vectors in the four initial lists named $L_1^{(0)}, L_2^{(0)}, L_3^{(0)}, L_4^{(0)}$. The parameter $\ell_1$ and the size of the initial
lists are subject to optimization. In the first step a \texttt{Sort-and-Match} on the lists \( L_1^{(0)}, L_2^{(0)} \) (resp. \( L_3^{(0)}, L_4^{(0)} \)) finds all pairs of vectors which are equal on the first \( \ell_1 \) bits. Those pairs are stored in two new lists

\[
L_1^{(1)} = \{(x_1, x_2) \in L_1^{(0)} \times L_2^{(0)} \mid (x_1 + x_2)[1] = 0\}
\]
\[
L_2^{(1)} = \{(x_3, x_4) \in L_3^{(0)} \times L_4^{(0)} \mid (x_3 + x_4)[1] = 0\}.
\]

In the second step another \texttt{Sort-and-Match} on those two lists finds vectors which add up to zero on the last \( \ell_2 \) bits while the first bits add up to zero by construction. The output is a list

\[
L_1^{(2)} = \{(x_1, x_2, x_3, x_4) \in L_1^{(1)} \times L_2^{(1)} \mid \sum_{j=1}^{4} x_j = 0\}
\]

of solutions to the 4-list problem. More details can be found in Alg. 2.3. The following example shows, how the 4-Tree-algorithm finds a solution to the 4-list problem.

\begin{algorithm}
\caption{4-Tree}
\begin{algorithmic}
\STATE \textbf{Input} : \( L_j^{(0)} \subset \mathbb{F}_2^n \) for \( j = 1, 2, 3, 4 \)
\STATE \textbf{Output} : \( (x_1, x_2, x_3, x_4) \in \mathbb{F}_2^{\ell_1} \times \mathbb{F}_2^{\ell_2} \times \mathbb{F}_2^{\ell_1} \times \mathbb{F}_2^{\ell_2} \)
\STATE \textbf{Parameters} : \( \ell = (\ell_1, \ell_2) \in \mathbb{N}^2 \) with \( \ell_2 := n - \ell_1 \)
\STATE \( L_1^{(1)} \leftarrow \texttt{Sort-and-Match}(L_1^{(0)}, L_2^{(0)}, 1, 0) \)
\STATE \( L_2^{(1)} \leftarrow \texttt{Sort-and-Match}(L_3^{(0)}, L_4^{(0)}, 1, 0) \)
\STATE \( L_1^{(2)} \leftarrow \texttt{Sort-and-Match}(L_1^{(1)}, L_2^{(1)}, 2, 0) \)
\IF{\( |L_1^{(2)}| > 0 \)} \RETURN some \( (x_1, x_2, x_3, x_4) \in L_1^{(2)} \) \ENDIF
\end{algorithmic}
\end{algorithm}
Example 2.1. Let

\[ L_1^{(0)} = \{101101, 001011, 111000, 001110\}, \quad L_2^{(0)} = \{101011, 101110, 011010, 110110\}, \]
\[ L_3^{(0)} = \{001111, 011100, 100000, 010010\}, \quad L_4^{(0)} = \{010100, 111111, 001101, 101110\} \]

be four lists with elements from the \( \mathbb{F}_2^n \). In the first step the algorithms finds vectors matching on the first two coordinates and store them into new lists

\[ L_1^{(1)} = \{(101101, 101011), (101101, 101110), (111000, 110110)\} \]
\[ L_2^{(1)} = \{(001111, 001101), (011100, 010100), (100000, 101110), (010010, 010100)\}. \]

Notice that all pairs in those lists add to zero on the first two coordinates (e.g. \( 101101, 101011 = 000110 \)). In a second step vectors whose checksums match on the last four coordinates found and stored in the final list

\[ L_1^{(2)} = \{(101101, 101011, 010010, 010100), (111000, 110110, 100000, 101110)\}. \]

Since they already match on the first coordinates, they form solutions to the 4-list problem. However, \( (101101, 110110, 001111, 010100) \) is another solution which is not found by the algorithm.

The following proposition shows that the 4-List problem can be solved by Alg. 2.3 in expectation.

Proposition 2.2. For any constant \( \varepsilon > 0 \) the 4-list problem can in expectation be solved in time and memory \( T = M = 2^{(\frac{2}{3} + \varepsilon)n} \).

Proof. Let \( S_0 := |L_j^{(0)}| = 2^\frac{n}{3}, \ j = 1, 2, 3, 4 \) and consider Alg. 2.3 with parameters \( \ell_1 = \frac{n}{3}, \ell_2 = \frac{2n}{3} \). By Lemma 1.2 we have

\[ S_1 := \mathbb{E}[|L_j^{(1)}|] = \frac{S_0^2}{2^\frac{n}{3}} = 2^\frac{2n}{3}, \ j = 1, 2 \quad \text{and} \quad \mathbb{E}[|L^{(2)}|] = \frac{S_1^2}{2^\frac{n}{3}} = 1. \]

Thus we obtain a solution in expectation. Furthermore all lists on the levels 0,1 have the same expected size \( 2^\frac{2n}{3} \) resulting in a total expected running time and memory consumption

\[ T = M = 2^{(\frac{2}{3} + \varepsilon)n} \]

using Lemma 1.2 again.  \( \square \)

Hence, this algorithm provides a lower running time compared to Alg. 2.2 at a cost of higher memory consumption.
The Case $k = 2^m$. The above idea of partially matching can be generalized if $k$ is a power of two. The reader is advised to follow the description again via Fig. 2.3 which shows the algorithm for $k = 8$.

![Diagram of the 8-Tree-algorithm](image)

Fig. 2.3: The 8-Tree-algorithm.

We introduce a split into $m$ blocks to the vectors in the initial lists $L_1^{(0)}, \ldots, L_k^{(0)}$ where the $i$-th block has length $\ell_i$ with $\sum_{i=1}^{m} \ell_i = n$. The goal is to construct a solution of some specific form similar to the 4 list case. The parameters $\ell_i$ as well as the size of the initial lists are subject to optimization again.

In the first step a Sort-and-Match pairwise merges the initial lists, resulting in new lists

$$L_j^{(1)} = \{(x_{2j-1}, x_{2j}) \in L_{2j-1}^{(0)} \times L_{2j}^{(0)} \mid (x_{2j-1} + x_{2j})_1 = 0\}$$

for $j = 1, \ldots, 2^{m-1}$. This pairwise merge is repeated in a tree wise fashion, matching on $\ell_i$ coordinates on level $i$ while the previous coordinates match to zero by construction. This yields

$$L_j^{(i)} = \{(x_{1,j}, \ldots, x_{2^i,j}) \in L_{2j-1}^{(i-1)} \times L_{2j}^{(i-1)} \mid \sum_{h=1}^{2^i} (x_{h,j})_i = 0\}$$

on levels $i = 2, \ldots, m - 1$ for $j = 1, \ldots, 2^{m-i}$ where $x_{h,j} := x_{2^i(j-1)+h}$ for $h = 1, \ldots, 2^i$. 

27
The final list
\[ L_1^{(m)} = \{(x_1, \ldots, x_k) \in L_1^{(m-1)} \times L_2^{(m-1)} \mid \sum_{i=1}^k x_i = 0\}. \]
contains solutions of the k-list problem. The following theorem shows that the k-List problem can be solved by Alg. 2.4 in expectation.

Algorithm 2.4: k-Tree

| Input | : \( L_j^{(0)} \subset \mathbb{F}_2^n \) for \( j = 1, \ldots, k = 2^m, m \in \mathbb{N} \) |
| Output | : \((x_1, \ldots, x_k) \in (\mathbb{F}_2^n)^k\) |
| Parameters | : \( \ell = (\ell_1, \ldots, \ell_m) \in \mathbb{N}^m \) |
| for \( i = 1, \ldots, m \) do |
| \( \ell_j^{(i)} \leftarrow \text{SORT-AND-MATCH}(L_{2j-1}^{(i-1)}, L_{2j}^{(i-1)}, i, 0) \) |
| for \( j = 1, \ldots, 2^{m-i} \) do |
| 1 |
| if \( |L_1^{(m)}| > 0 \) then return some \((x_1, \ldots, x_k) \in L_1^{(m)}\) else return \(\perp\) |

Theorem 2.2. For any constant \( \varepsilon > 0 \), constant \( m \in \mathbb{N} \) and \( k = 2^m \) the k-list problem can in expectation be solved in time and memory \( T = M = 2^{(\frac{1}{m+1} + \varepsilon)n} \).

Proof. Let \( S_0 := |L_j^{(0)}| = 2^{\frac{n}{m+\varepsilon}}, j = 1, \ldots, k \) and consider Alg. 2.4 with parameters
\[ \ell_m = \frac{2n}{m+1}, \quad \ell_i = \frac{n}{m+1}, \quad i = 1, \ldots, m-1. \]

By Lemma 1.2 we have
\[ S_i := \mathbb{E}[|L_j^{(i)}|] = \frac{S_{i-1}^2}{2^{\ell_i}} = 2^{\frac{n}{m+\varepsilon}}, j = 1, \ldots, 2^{m-i} \]
on layers \( i = 1, \ldots, m-1 \) and
\[ \mathbb{E}[|L_1^{(m)}|] = \frac{S_{m-1}^2}{2^{\ell_m}} = 1. \]

Thus we obtain a solution in expectation. Furthermore all lists on level \( i = 0, \ldots, m-1 \) have the same expected size \( S_i = 2^{\frac{n}{m+\varepsilon}} \) resulting in a total expected running time and memory consumption
\[ T = M = 2^{(\frac{1}{m+1} + \varepsilon)n} \]
using Lemma 1.2 again. \( \square \)
The result also holds for bigger initial lists since one can simply discard elements and shrink the list size to $2^{\frac{m}{k+1}}$ if necessary. Figure 2.4 visualizes the running times of the $k$-tree-algorithm.

![Figure 2.4: Dependence of the $k$-Tree algorithm on $m$.](image)

We would like to point out that above algorithm can be easily adapted to find a solution to the equation

$$x_1 + \ldots + x_k = c$$

for some vector $c \in \mathbb{F}_2^n$. We simply define the list $L'_k := \{x + c \mid x \in L_k\}$ and use the $k$-tree-algorithm on the lists $L_1, \ldots, L_{k-1}, L'_k$. The returned solution satisfies

$$x_1 + \ldots + x_{k-1} + x_k + c = 0$$

which is equivalent to Eq. 2.3.

**Decreasing List Sizes.** While information theoretically a solution to the $k$-list problem exists if $|L_1| = \ldots = |L_k| = 2^\frac{n}{k}$, the $k$-Tree-algorithm increases this bound to $2^{\frac{m}{k+1}}$. However, in some applications where the initial lists are fixed and contain less elements, Wagner’s algorithm cannot be applied. Minder and Sinclair showed in [MS09], how to find a solution given lists of smaller size. In their *extended k-tree-algorithm* one compensates for the smaller initial lists by eliminating less bits. Hence, the list size grows throughout the search tree until some sufficient size is reached. For example, if we are given four lists of size $2^\frac{n}{4} < 2^\frac{n}{k}$ only, the extended-$k$-tree-algorithm uses *Sort-and-Match* on $\frac{n}{4}$ bits in the first step resulting in lists of expected size $2^\frac{n}{2}$. In the final step a *Sort-and-Match* on the remaining $\frac{3}{4}n$ coordinates leaves 1 solution in expectation again. However, the running time increases from $2^\frac{n}{k}$ to $2^\frac{n}{2}$.
Finding a Random Solution. As mentioned before, Alg. 2.4 only finds solutions with some specific structure. We show now that this restriction can be eliminated. To give an idea how to find a solution with random structure we describe a variant of the 8-tree-algorithm (see Fig. 2.5). In the first step the lists $L_1^{(0)}$ and $L_2^{(0)}$ (resp. $L_3^{(0)}$ and $L_4^{(0)}$) are merged via Sort-and-Match again. Instead matching to zero, one now matches to some random value $y_{1,1} \in \mathbb{F}^{\ell_1}$ on the first $\ell_1$ bits. The same procedure is repeated on the other half of the tree, matching to another random value $y_{1,2} \in \mathbb{F}^{\ell_1}$. The resulting lists $L_1^{(1)}$ and $L_2^{(1)}$ (resp. $L_3^{(1)}$ and $L_4^{(1)}$) are merged in the same way matching to some random value $y_{2,1} \in \mathbb{F}^{\ell_2}$ on the next $\ell_2$ bits. Note that the first $\ell_1$ bits still match to zero. In the last step we match to zero on the remaining $\ell_3$ bits.

However, this algorithm still finds solution of some specific (now random) structure only. A single existing golden solution is only found with negligible probability. This is in contrast to the algorithm by Schroeppel and Shamir (Alg. 2.2), which finds all existing solutions. Hence the gain in speed comes at a cost of flexibility. Nevertheless this is still sufficient for many applications as we see later.

The case $k = 2^m + j$. For choices $k = 2^m + j, j \geq 0$ we can bound the complexity by the case $2^m$. In a nutshell, we choose random values $x_i \in L_i$ for $i = 2^m + 1, \ldots, 2^m + j$ and define $c := \sum_{i=2^m+1}^{2^m+j} x_i$. Then we use the same strategy as for Eq. 2.3.

Large $k$: Gaussian Elimination. It was already shown by Bellare et al. in [BM97] that, for sufficiently large $k \geq n$, the $k$-list problem can be solved in polynomial
time via Gaussian elimination. This algorithm only needs two elements per list, i.e. 
\[ L_i := \{x_{i,0}, x_{i,1}\} \text{ for } i = 1, \ldots, k \] and we discard any additional elements. Furthermore we define \( b_i \in \mathbb{F}_2 \) with \( b_i = 0 \) for \( i = 1, \ldots, k \) if and only if \( x_{i,0} \in L_i \) is chosen for the solution vector. Thus we have to find some \( b := (b_1, \ldots, b_k) \in \mathbb{F}_2^k \) such that

\[
\sum_{i=1}^{k} b_i x_{i,1} + (1 - b_i) x_{i,0} = 0 \iff \sum_{i=1}^{k} b_i (x_{i,1} - x_{i,0}) = -\sum_{i=1}^{k} x_{i,0}.
\]

This linear equation in \( b \) can be solved via Gaussian elimination with high probability in polynomial time in \( n \) and \( k \). If there is no solution we can simply discard the lists and choose two new elements per list.

Wagner claimed that the \( k \)-list problem can be solved in subexponential time for the case \( k = 2\sqrt{n} \). This is followed from Theorem 2.2 for \( m = \sqrt{n} \) assuming 
\[
T = k \cdot 2^{\left(\frac{1}{\sqrt{\pi}}+\epsilon\right)n} = 2^{\sqrt{n} \cdot 2^{\left(\frac{1}{\sqrt{\pi}}+\epsilon\right)n}} = 2^{(2+\epsilon)\sqrt{n}}.
\]
However for \( n \geq 16 \) we have \( 2^{\sqrt{n}} \geq n \) and thus polynomial complexity already which makes this observation obsolete.

### 2.4 Applications

Wagner described several applications for his \( k \)-Tree-algorithm in [Wag02]. In this section we take a closer look at two of them. The first one is collision search for hash functions which we present here as an introductory example. The second application is the finding of low weight polynomials which is also an application for our approximate \( k \)-tree algorithm described in the next chapter.

**Finding Hash Collisions.** The \( k \)-Tree-algorithm can be applied to various constructions of hash functions, e.g. AdHash by Bellare et al [BM97]. Let \( n \) be a public parameter, \( D \) a set and \( h : \mathbb{N} \times D \to \mathbb{F}_2^n \) some hash function. We then define \( H : D^k \to \mathbb{F}_2^n \) as

\[
H(x) := \sum_{i=1}^{k} h(i, x_i)
\]

where \( x = (x_1, \ldots, x_k) \) with \( x_i \in D \) and w.l.o.g. \( k = 2^m \) for \( m \in \mathbb{N} \). An attack based on the \( k \)-tree-algorithm is straight forward. One defines the lists

\[
L_i := \{h(i, x_i) \mid x_i \in D\} \quad \text{with} \quad |L_i| = 2^{\frac{n}{\sqrt{\pi}}}, \quad i = 1, \ldots, k.
\]

In expectation, one collision

\[
H(x) := h(1, x_1) + \ldots + h(k, x_k) = H(y) =: c \quad \text{for some } y \in D^k
\]
Finding Low Weight Polynomials. Low weight polynomials are essential parts of so-called fast correlation attacks on stream ciphers [MS89, CT00, CJM02, LJ14, ZXM15]. For some irreducible polynomial $P(X)$ over $\mathbb{F}_2$ and of degree $n$ we define by $\mathbb{F} := \mathbb{F}_2[X]/P[X]$ a finite field of size $2^n$ induced by $P(X)$. Furthermore, let $\Delta(P(X))$ denote the weight of $P(X)$, i.e. the number of non-zero coefficients. We define the following problem.

**Definition 2.3 (Parity Check Problem).** Let $n, d, N \in \mathbb{N}$ and let $P(X)$ be an irreducible polynomial over $\mathbb{F}_2$ of degree $n$. One has to find a multiple $Q(X)$ of $P(X)$ of weight smaller than $d$ and degree smaller than $N$. Typically one has $d \ll n$ and $N \gg n$.

Let us choose $k = 2^m = 2^{\lceil \log d \rceil}$ as the number of lists given to the $k$-Tree-algorithm and some fixed polynomial $C(X) \mod P(X)$ of weight $d - k$. We fill each list with $2^{\frac{n}{m} + 1}$ elements of the form $X^a \mod P(X) \in \mathbb{F}$, $a \in \{1, 2, \ldots, N\}$.

The $k$-tree-algorithm now operates on the coefficient vectors returning one expected solution satisfying

$$S(X) := X^{a_1} + \ldots + X^{a_k} = C(X) \mod P(X).$$

It follows that

$$S(X) + C(X) = 0 \mod P(X),$$

i.e. $S(X) + C(X)$ is a multiple of $P(X)$ with weight $\leq |S(X)| + |C(X)| \leq d$. Furthermore the solution has maximum degree $N$ since $a_i \leq N$ and $C(X)$ is reduced modulo $P(X)$. We discuss the running times for specific parameters in the next chapter where we also compare this algorithm to our approximate $k$-tree-algorithm.
Chapter 3

The Approximate $k$-List Problem

In some recent applications of the $k$-list problem (Def. 2.2) – e.g. solving LPN [GJL14, GJS15, KF15] or the decoding problem we are going to introduce in Chapter 4 – the requirement that all vectors $x_i$ sum up to the target vector $0$ is too strong. Instead, some low Hamming weight is allowed. This motivates a more flexible variant of the $k$-list problem, which was introduced for the first time in a joint work with Alexander May [BM17a]. In our new problem we allow some error positions $w$ in the target vector. Since we obtain more potential solutions compared to the original $k$-list problem, this problem should be easier to solve. In this chapter we analyze this new problem and provide different solving algorithms. The results are mainly based on a previous publication [BM17a].

3.1 A Relaxed Version of the $k$-List Problem

First, let us formally define the approximate $k$-list problem.

**Definition 3.1 (Approximate $k$-List Problem).** Let $w \in [\frac{n}{2}]$ be the target weight and let $L_1, \ldots, L_k \subset \mathbb{F}_2^n$ be $k$ lists with uniform and independent vectors. Given those lists, one has to find elements $(x_1, \ldots, x_k) \in L_1 \times \ldots \times L_k$ satisfying

$$\Delta(x_1 + \ldots + x_k) \leq w.$$  \hfill (3.1)

We often assume that $w$ is linear in $n$, i.e. $w := \gamma n$ for some $\gamma \in [0, \frac{1}{2}]$. Analogously to the $k$-list problem, the Approximate $k$-list problem has a solution in expectation, whenever $|L_1 \times \ldots \times L_k| \geq 2^n \cdot \left(\frac{n}{w}\right)^{-1}$ holds.
**Match-and-Filter-Algorithm.** Before we present our new algorithm, we describe a very simple and easy to implement variant, called MATCH-AND-FILTER (see Figure 3.1). It is based on Leurent’s near-collision search for hash functions [Leu12] which is adapted to the approximate \( k \)-list setting. The overall idea is to find solutions to the \( k \)-list problem on a fraction of all coordinates and filtering for the correct Hamming weight on the remaining (so far truncated) parts.

![Fig. 3.1: The MATCH-AND-FILTER-algorithm for \( k = 4 \)](image)

In detail, the algorithm (see also Alg. 3.1) starts with \( k = 2^m \) lists \( L_1^{(0)}, \ldots, L_k^{(0)} \) of size \( 2^c \) for some \( c \). Those lists are now pairwise combined with the SORT-AND-MATCH-algorithm (Alg. 1.1) in order to find elements matching on \( c \) bits. Just like in the \( k \)-Tree-algorithm (Alg. 2.4), this is repeated until only one list \( L_1^{(m)} \), containing vectors whose sum is zero on the first \( m \cdot c \) bits, is left. Then the algorithm filters for the correct weight \( w \) on the remaining bits using the FILTER-algorithm (Alg. 1.2). The expected list size remains \( 2^c \) throughout the whole sort-and-match part. Thus the expected number of solutions in the final list is

\[
\mathbb{E}[|L_1^{(m)}|] = 2^c \cdot \frac{\sum_{i=0}^{w} \binom{n-m \cdot c}{i}}{2^{n-m \cdot c}} \tag{3.2}
\]

since we filter for weight \( \leq w \) on \( n - m \cdot c \) bits in a list of size \( 2^c \). It remains to find a minimal value for \( c \) such that \( \mathbb{E}[|L_1^{(m)}|] \geq 1 \) numerically. The expected running time \( T \) and space consumption \( M \) are determined by the list sizes. Therefore we have \( T = M = 2^c + \varepsilon n \).
Algorithm 3.1: Match-and-Filter

**Input:** \( L_j^{(0)} \subset \mathbb{F}_2^n, |L_j^{(0)}| = 2^c \) for \( j = 1, \ldots, k = 2^m, w \in \mathbb{N} \)

**Output:** \((x_1, \ldots, x_k) \in (F_2^n)^k\)

**Parameters:** \( \ell = (c, \ldots, c, n - m \cdot c) \in \mathbb{N}^{m+1} \) for \( i = 1, \ldots, m \)

for \( i = 1, \ldots, m \) do
   for \( j = 1, \ldots, 2^{m-i} \) do
      \( L_j^{(i)} \leftarrow \text{Sort-And-Match}(L_j^{(i-1)}, L_j^{(i-1)}, i, 0) \)
   end
end

2 \( L_1^{(m)} \leftarrow \text{Filter}(L_1^{(m)}, m + 1, w) \)

if \( |L_1^{(m)}| > 0 \) then return some \((x_1, \ldots, x_k) \in L_1^{(m)}\) else return \( \perp \)

Using Nearest Neighbors. For \( k = 2 \) the approximate \( k \)-list problem is very similar to finding nearest neighbors. Hence, our approximate 2-tree-algorithm uses the May-Ozerov nearest neighbor search on two lists \( L_1, L_2 \). The list sizes are chosen sufficiently large, such that there exists a solution in expectation, i.e. \( \exists (x_1, x_2) \in L_1 \times L_2 \) with Hamming distance \( \Delta(x_1, x_2) \leq w \). Based on [MO15] we obtain the following result for the approximate 2-list problem which is of importance for solving the general approximate \( k \)-list problem.

**Lemma 3.1.** For any constant \( \varepsilon > 0 \) and \( 0 \leq \gamma \leq \frac{1}{2} \) with \( \gamma n \in \mathbb{N} \), the approximate 2-list problem with target weight \( \gamma n \) can in expectation be solved in time

\[
T = 2^{(\tau_2(\gamma) + \varepsilon)n} \quad \text{with} \quad \tau_2(\gamma) := (1 - \gamma) \left( 1 - H \left( \frac{H^{-1}(\frac{1+H(\gamma)}{2}) - \frac{\gamma}{2}}{1-\gamma} \right) \right),
\]

using memory \( M = 2^{(1-H(\gamma) + \varepsilon)n} \).

**Proof.** Let \( |L_1| = |L_2| = 2^{\lambda n} \) with \( \lambda := \frac{1-H(\gamma)}{2} \). By Lemma 1.4 the expected number of solutions can be upper bounded by

\[
E[|L|] \geq \frac{2^{(2\lambda + H(\gamma) - 1)n}}{\text{poly}(n)} = \frac{1}{\text{poly}(n)}.
\]

Increasing the initial list sizes \( |L_1|, |L_2| \) by only a polynomial factor (which adds to the constant term \( \varepsilon \)), we obtain a solution in expectation. Notice, that the restriction of
Theorem 1.1 is satisfied since
\[ \lambda = \frac{1 - H(\gamma)}{2} < 1 - H(\gamma) < 1 - H\left(\frac{\gamma}{2}\right). \]
Thus, by Lemma 1.4, the May-Ozerov nearest neighbor search in expectation solves the given instance of the approximate 2-list problem in time
\[ T = 2^{\tau(\lambda, \gamma) + \epsilon}n = 2^{\tau_2(\gamma) + \epsilon}n \]
and the memory consumption is determined by the list sizes.

\[ \Box \]

3.2 Solving the Approximate k-list Problem

The overall idea of our approximate k-tree-algorithm is based on the original k-Tree-algorithm (Alg. 2.4) by Wagner [Wag02], i.e. we find partially matching vectors in a tree-wise fashion. However we use a more relaxed strategy where we only filter for some small error in parts of the tree. In addition to this we make use of the NN-Search-algorithm analogously to the 2-tree-algorithm described in the previous section. As a result our algorithm uses the allowed error weight to its advantage and compensates for the higher amount of solutions with smaller initial list sizes.

The Approximate k-tree-algorithm. Our algorithm combines the techniques used in MATCH-AND-FILTER (Alg. 3.1) with the advantages of neighbor search. In a nutshell, the algorithm for \( k = 2^m \) repeats the SORT-AND-MATCH step several times followed by a nearest neighbor search for the target weight (see Figure 3.2 for the case \( k = 8 \)).

Let us describe the algorithm in detail. We introduce a split into \( m \) blocks to all vectors in the \( k = 2^m \) initial lists named \( L_1^{(0)}, \ldots, L_k^{(0)} \). The first \((m - 2)\) blocks contain \( c \) bits each, the next block \( \ell \) and the rightmost block \( \ell_m := n - (m - 2)c - \ell \) bits. The initial lists are of size \( 2^c \), where \( c \) and \( \ell \) are parameters subject to optimization. In the first step, SORT-AND-MATCH combines two lists at a time and finds matching vectors on the first block of \( c \) coordinates in the creation of the lists \( L_1^{(1)} \) for \( j = 1, \ldots, 2^{m-1} \) on level 1. This is repeated until only two lists \( L_1^{(m-1)}, L_2^{(m-1)} \) are left on level \( m - 1 \). We match on \( c \) bits on level \( i \) for \( i = 1, \ldots, m - 2 \) and on \( \ell \) bits on level \( m - 1 \). For the lists on level \( i \) we have
\[ L_j^{(i)} = \{(x_{1,j}, \ldots, x_{2^i,j}) \in L_2^{(i-1)} \times L_2^{(i-1)} \mid \sum_{h=1}^{2^i} (x_{h,j})_{[i]} = 0\} \]
for $j = 1, \ldots, 2^{m-i}$ where $x_{h,j} := x_{2^i(j-1)+h}$ for $h = 1, \ldots, 2^i$. As a last step we construct the final list

$$L_1^{(m)} = \{(x_1, \ldots, x_k) \in L_1^{(m-1)} \times L_2^{(m-1)} | \Delta \left( \sum_{j=1}^{k} (x_j)_{[m]} \right) \leq \gamma n \}.$$ 

via NN-Search on the last $\ell_m$ bits for weight $\leq w$ out of those two lists. Since we all elements match to zero on the previous coordinates, this last list only contains solutions to the approximate $k$-list problem with target weight $w$. This concludes the description of the algorithm (also see Alg. 3.2). We would like to point out that for $w = 0$ the algorithm collapses to the original $k$-Tree-algorithm and therefore we achieve the same running times. Note that this algorithm only works for $k > 2$ while for $k = 2$ we have the result of Lemma 3.1 As we see later, the choice of eliminating $\ell$ instead of $c$ bits in level $m-1$ is an adjustment to compensate for the increased running time of NN-Search and balances the running times. For $k = 2$ we cannot do this balancing since we only have one computation step - a nearest neighbor search. The following example shows, how the APPROX-4-Tree-algorithm finds a solution to the approximate $4$-list problem.

Fig. 3.2: The approximate $k$-tree-algorithm, $k = 8$. 

$L_1^{(0)} L_2^{(0)} L_3^{(0)} L_4^{(0)} L_5^{(0)} L_6^{(0)} L_7^{(0)} L_8^{(0)}$

$L_1^{(1)} L_2^{(1)} L_3^{(1)} L_4^{(1)}$

$L_1^{(2)} L_2^{(2)}$

$L_1^{(3)}$

$c \ell \ell_s$

$L_1^{(0)} L_2^{(0)} L_3^{(0)} L_4^{(0)} L_5^{(0)} L_6^{(0)} L_7^{(0)} L_8^{(0)}$

$L_1^{(1)} L_2^{(1)} L_3^{(1)} L_4^{(1)}$

$L_1^{(2)} L_2^{(2)}$

$L_1^{(3)}$

$c \ell \ell_s$
Algorithm 3.2: Approx-k-Tree

Input : $L^{(0)}_j \subset \mathbb{F}_2^2$ with $|L^{(0)}_j| = 2^c$ for $j = 1, \ldots, k = 2^m$, $m \in \mathbb{N}_{>1}$, $w \in \mathbb{N}$
Output : $(x_1, \ldots, x_k) \in (\mathbb{F}_2^2)^k$

Parameters: $\ell := (c, \ldots, c, \ell_m) \in \mathbb{N}^m$ with $\ell_m := \ell, n - (m - 2)c - \ell$

for $i = 1, \ldots, m - 1$ do
  for $j = 1, \ldots, 2^{m-i}$ do
    $L^{(i)}_j \leftarrow \text{Sort-And-Match}(L^{(i-1)}_{2j-1}, L^{(i-1)}_{2j}, i, 0)$
  end
end

$L^{(m)}_1 \leftarrow \text{NN-Search}(L^{(m-1)}_1, L^{(m-1)}_2, m, w)$
if $|L^{(m)}_1| > 0$ then return some $(x_1, \ldots, x_k) \in L^{(m)}_1$ else return ⊥

Example 3.1. Let

$L^{(0)}_1 = \{0111, 1100, 0010\}, \quad L^{(0)}_2 = \{1000, 1011, 0101\},$
$L^{(0)}_3 = \{0011, 1101, 0001\}, \quad L^{(0)}_4 = \{1001, 1000, 1110\}$

be the four initial list with elements from $\mathbb{F}_2^2$ and let $w = 1$. In the first step the algorithm finds vectors matching on the first block of size $\ell$. Assuming we have $\ell = 2$, then

$L^{(1)}_1 = \{(0111, 0101)\}, \quad L^{(1)}_2 = \{(1101, 1110)\}$.

Notice that all pairs in those lists add to zero on the first two coordinates (e.g. 0111 + 0101 = 0010). The nearest neighbor search on the remaining two bits for weight ≤ 1 returns

$L^{(2)}_1 = \{(0111, 0101, 1101, 1110)\}$.

Further solutions like (0010, 1000, 0001, 1001) do not survive the first step and are not found by the algorithm.

Theorem 3.1. Let $k = 2^m$ with $m \in \mathbb{N}_{>1}$, $0 \leq \gamma \leq \frac{1}{2}$ with $\gamma n \in \mathbb{N}$ and $\gamma_m := \frac{\gamma}{(m - 2)\lambda - \delta}$ with $\lambda, \delta \in [0, 1]$ such that one of the two conditions hold.

$$\lambda = \frac{2\tau_2(\gamma_m)}{1 + H(\gamma_m) + 2m\tau_2(\gamma_m)}$$

and

$$\delta = \frac{H(\gamma_m) + 4\tau_2(\gamma_m) - 1}{1 + H(\gamma_m) + 2m\tau_2(\gamma_m)}$$
or
\[
\lambda = \delta = \frac{1 - H(\gamma_m)}{1 + m + (1 - m) \cdot H(\gamma_m)}. \tag{3.4}
\]
For any constant \( \varepsilon > 0 \), the approximate \( k \)-list problem with target weight \( \gamma n \) can in expectation be solved in time and memory \( T = M = 2^{(\lambda + \varepsilon)n} \) under Condition 3.3 respectively time \( T = 2^{(\gamma_m(1 - (m - 2)\lambda - \delta) + \varepsilon)n} \) and memory \( M = 2^{(\lambda + \varepsilon)n} \) under Condition 3.4.

**Proof.** We use the previously described algorithm and set \( w := \gamma n \), \( c := \lambda n \), \( \ell := \delta n \).

By Lemma 1.2 and Lemma 1.4 we have
\[
S_0 := |L_j(0)| = 2^c = 2^{\lambda n}, \quad j = 1, \ldots, 2^m
\]
\[
\mathbb{E}[|L_j^{(i)}|] = \frac{S_0^2}{2^{\lambda n}} = 2^{\lambda n} = S_0, \quad j = 1, \ldots, 2^{m-1}, i = 1, \ldots, m - 2
\]
\[
S_{m-1} := \mathbb{E}[|L_j^{(m-1)}|] = \frac{S_0^2}{2^{\lambda n}} = 2^{(2\lambda - \delta)n} = 2^{1 - \frac{H(\gamma_m)}{2}}\ell, \quad j = 1, 2
\]
\[
\mathbb{E}[|L_j^{(m)}|] = \sum_{i=0}^{\gamma_m} \frac{\binom{\ell}{i}}{2^{\ell}} \geq \frac{1}{\text{poly}(n)}
\]
since \( \gamma_m = \frac{2^m}{\ell_m} \) and \( 2^{(1 - H(\gamma_m))\ell_m + (H(\gamma_m) - 1)\ell_m} = 1 \). Thus, following the argumentation of Lemma 3.1, the algorithm returns in expectation one solution for the approximate \( k \)-list problem with target weight \( \gamma n \). By Lemma 1.2 the Sort-and-Match calls in step 1 take time
\[
T_0 := 2^{\max\{\lambda, 2\lambda - \delta, \gamma\}}n.
\]
Using Lemma 3.1, NN-Search for weight \( w \) (i.e. relative weight \( \gamma_m \)) on \( \ell_m \) bits in step 2 runs in time
\[
T_1 := 2^{\gamma_m \ell_m}.
\]
Finally, using Condition 3.3, we have \( T_0 = T_1 \) as well as \( S_0 \geq S_{m-1} \) and thus
\[
T = \max\{T_0, T_1\} = 2^{(\lambda + \varepsilon)n} = M.
\]
For Condition 3.4 we obtain \( S_0 = S_{m-1} \) and \( T_1 \geq T_0 \) using the fact that the initial list size is a lower bound of the running time for the May-Ozerov nearest neighbor search. Hence,
\[
M = 2^{(\lambda + \varepsilon)n}, \quad T = T_1.
\]

Because \( \gamma_m \) is a function of \( \lambda \) and \( \delta \) we cannot derive a closed formula for \( T \) and \( M \).
However we can determine the optimal values for different instances of the approximate $k$-list problem through numerical optimization. We state the results of this optimization and a comparison to our other algorithms in Section 3.4. For Eq. 3.3 we have balanced running times but unbalanced list sizes over the different levels. Thus this case yields a minimized running time. The case defined by Eq. 3.4 yields balanced list sizes and a higher running time and thus minimized memory consumption. We have a classical time-memory trade-off where we can also adjust the parameters to achieve any cases between optimal time and optimal memory (see Fig. 3.3). Notice that $T = M$ for minimal running time.

![Fig. 3.3: Time-Memory trade-off for $m = 2$.](image)

### 3.3 A Provable Variant

Before we give running times and memory consumptions for our Approx-$k$-Tree-algorithm derived from numerical optimization, let us first present a variant for which we are able to provide a closed formula for the complexity. We see later that the new algorithm is a generalization of Approx-$k$-Tree. The main difference lies in a different choice of parameters and in an additional filtering step.

**The Case $k = 4$.** Before generalizing our approximate $k$-tree-algorithm to arbitrary powers of two $k = 2^m$, let us first look at the easiest case $k = 4$ (see Figure 3.4).

We denote the target weight by $w := \gamma n$ and introduce a split into 4 blocks to all vectors in the four initial lists named $L^{(0)}_1, L^{(0)}_2, L^{(0)}_3, L^{(0)}_4$. The first block consists of $\ell_1$, the second and third block of $\ell'_1$ and the rightmost block of $\ell_2 := n - \ell_1 - 2\ell'_1$ coordinates. The initial lists are of size $2^c$, where $c$ as well as the parameters $\ell_1, \ell'_1$ are subject to optimization. In the first step a Sort-and-Match on the lists $L^{(0)}_1, L^{(0)}_2$.
Fig. 3.4: The APPROX-4-TREE-algorithm

(resp. $L_3^{(0)}$, $L_4^{(0)}$) finds all pairs of vectors which are equal on block 1 and 2 (resp. block 1 and 3). Subsequent filtering using the FILTER-algorithm for relative weight $\leq \gamma$ on $L_1^{(1)}$ (resp. $L_2^{(1)}$) on block 3 (resp. block 2) results in two lists

\[ L_1^{(1)} = \{ (x_1, x_2) \in L_1^{(0)} \times L_2^{(0)} | (x_1 + x_2)_{[1,2]} = 0, \Delta((x_1 + x_2)_{[3]}) \leq \gamma \ell_1' \} \]
\[ L_2^{(1)} = \{ (x_3, x_4) \in L_3^{(0)} \times L_4^{(0)} | (x_3 + x_4)_{[1,3]} = 0, \Delta((x_3 + x_4)_{[2]}) \leq \gamma \ell_1' \}. \]

Note that the search tree is asymmetric on this layer. The first block in the lists $L_1^{(1)}$ and $L_2^{(1)}$ have the same property while the second and third block contrast each other. If we now add elements from these lists, the first $\ell_1$ bits add up to zero by construction while we obtain relative weight $\leq \gamma$ on the next $2\ell_1'$ bits. This is used in the second step. We apply the NN-SEARCH-algorithm for the lists $L_1^{(1)}$, $L_2^{(1)}$ finding vectors summing up to some relative weight $\leq \gamma_2 \leq \frac{n - 2\ell_1'}{\ell_2} \gamma$ on the last $\ell_2$ bits. Thus the final list

\[ L_1^{(2)} = \{ (x_1, x_2, x_3, x_4) \in L_1^{(1)} \times L_2^{(1)} | \Delta(\sum_{j=1}^4 (x_j)_{[4]}) \leq \gamma_2 \ell_2 \}. \]

contains solutions to the approximate 4-list problem since $2\gamma \ell_1' + \gamma_2 \ell_2 \leq \gamma n$. The following proposition shows that the approximate 4-list problem can be solved by Alg. 3.3 in expectation.
Algorithm 3.3: Gen-Approx-4-Tree

Input : \( L_j^{(0)} \subset F_2^n \) for \( j = 1, 2, 3, 4 \), \( w \in \mathbb{N} \)
Output : \((x_1, x_2, x_3, x_4) \in F_2^n \times F_2^n \times F_2^n \times F_2^n \)
Parameters : \( \ell := (\ell_1, \ell_1', \ell_2, \ell_2') \in \mathbb{N}^4 \), with \( \ell_2 := n - \ell_1 - 2\ell_1', \gamma_2 \in [0, 1] \)

Define \( \gamma := \frac{w}{n} \)

1. \( L_1^{(1)} \leftarrow \text{Sort-And-Match}(L_1^{(0)}, L_2^{(0)}, \{1, 2\}, 0) \)
2. \( L_2^{(1)} \leftarrow \text{Sort-And-Match}(L_3^{(0)}, L_4^{(0)}, \{1, 3\}, 0) \)
3. \( L_1^{(1)} \leftarrow \text{Filter}(L_1^{(1)}, 3, \gamma \ell_1') \)
4. \( L_2^{(1)} \leftarrow \text{Filter}(L_2^{(1)}, 2, \gamma \ell_1') \)
5. \( L_2^{(2)} \leftarrow \text{NN-Search}(L_1^{(1)}, L_2^{(1)}, 4, \gamma_2 \ell_2) \)

if \( |L_1^{(2)}| > 0 \) then return some \((x_1, x_2, x_3, x_4) \in L_1^{(2)} \) else return \( \perp \)

Proposition 3.1. Let \( 0 \leq \gamma \leq \frac{1}{2} \) with \( \gamma n \in \mathbb{N} \), \( \gamma^* > 0 \) with \( H(\gamma^*) = \frac{1}{1+2\tau_2(\gamma^*)} \) and

\[
\tau_4(\gamma) := \begin{cases} 
\frac{2-2H(\gamma)}{1-H(\gamma)+4\tau_2(\gamma)-2H(\gamma)\tau_2(\gamma)} \tau_2(\gamma) & \gamma \leq \gamma^* \\
\frac{2-H(\gamma)}{1+4\tau_2(\gamma)} \tau_2(\gamma) & \gamma \geq \gamma^* 
\end{cases}
\]

For any constant \( \varepsilon > 0 \), the approximate 4-list problem with target weight \( \gamma n \) can in expectation be solved in time and memory \( T = M = 2^{(\tau_4(\gamma)+\varepsilon)n} \).

Proof. We use the previously described algorithm and set

\[
\gamma_2 := \gamma, \quad w := \gamma n, \quad c := \tau_4(\gamma)n \\
\ell_1 := \begin{cases} 
\frac{1-H(\gamma)-2H(\gamma)\tau_2(\gamma)}{1-H(\gamma)+4\tau_2(\gamma)-2H(\gamma)\tau_2(\gamma)} n & \gamma \leq \gamma^* \\
0 & \gamma \geq \gamma^* 
\end{cases} \\
\ell_1' := \begin{cases} 
\frac{H(\gamma)+2\tau_2(\gamma)-1}{H(\gamma)+4\tau_2(\gamma)-2H(\gamma)\tau_2(\gamma)} n & \gamma \leq \gamma^* \\
\frac{H(\gamma)+2\tau_2(\gamma)-1}{2+8\tau_2(\gamma)} n & \gamma \geq \gamma^* 
\end{cases}
\]

Let \( S_0 := |L_j^{(0)}| = 2^c \) for \( j = 1, 2, 3, 4 \) and

\[
S_1 := 2^{\frac{1-H(\gamma)}{2}} \ell_2 = 2^{2c-\ell_1+(H(\gamma)-2)\ell_1}.
\]
By Lemma 1.2, Lemma 1.3 and Lemma 1.4 we have

\[
E[L_j^{(1)}] = \frac{S_0^2}{2^{c+\ell_1}} \cdot 2^{\ell_1} \sum_{i=0}^{\ell_1} \bigg( \frac{\gamma \ell_1'}{2^{\ell_2}} \bigg) \Rightarrow \frac{S_1}{\text{poly}(n)} \leq E[L_j^{(1)}] \leq S_1 \cdot 2^{\epsilon n}, j = 1, 2
\]

\[
E[L_1^{(2)}] = E[L_j^{(1)}]^2 \geq \frac{S_0^2 \ell_2}{\text{poly}(n)} \cdot \frac{\gamma \ell_2}{2^{\ell_2}} \geq \frac{1}{\text{poly}(n)}
\]

since \(2^{(1-H(\gamma))\ell_2+(H(\gamma_2)-1)\ell_2} = 1\). Furthermore, every element in \(L_1^{(2)}\) has Hamming weight \(\leq 2\gamma \ell_1' + \gamma \ell_2 = \gamma (n - \ell_1) \leq \gamma n\). Therefore, according to Lemma 3.1, the algorithm returns one solution for the approximate 4-list problem with target weight \(w = \gamma n\) in expectation.

By Lemma 1.2 and 1.3 the Sort-and-Match steps and the filtering in steps 1 − 4 take time

\[T_0 := 2^{\max\{c, 2c - \ell_1 - \ell_1'\} + \epsilon n}.\]

Since \(S_1 = 2^{\frac{1-H(\gamma)}{2} \ell_2}\), NN-Search for relative weight \(\gamma\) on \(\ell_2\) bits in step 5 is equivalent to our approximate 2-tree-algorithm. Thus we can apply Lemma 3.1 again and obtain the running time

\[T_1 := 2^{\tau_2(\gamma)\ell_2 + \epsilon n} = 2^{(\tau_4(\gamma) + \epsilon)n}.\]

For \(\gamma \leq \gamma^*\) we have \(c = 2c - \ell_1 - \ell_1'\) while for \(\gamma \geq \gamma^*\) we can use the fact that \(H(\gamma) \geq \frac{1}{2^{\tau_2(\gamma)+1}}\) (see Fig. 3.5 for a numerical comparison) which is equivalent to \(c \geq 2c - \ell_1 - \ell_1'\). Thus, with \(c = \tau_4(\gamma)n\), we have

\[T_0 = 2^{(\tau_4(\gamma) + \epsilon)n} = T_1 \Rightarrow T = \max\{T_0, T_1\} = 2^{(\tau_4(\gamma) + \epsilon)n}.
\]

The expected memory consumption is determined by the list sizes, i.e.

\[M = 2^{(\tau_4(\gamma) + \epsilon)n}.\]

\[\square\]

**The Case \(k = 2^m\).** Our algorithm for \(k = 4\) already illustrates the overall idea of approximate matching. In a nutshell, the algorithm for \(k = 2^m\) repeats the Sort-and-Match step several times followed by one Filter step and the NN-SEARCH-algorithm for the remaining to lists (see Figure 3.6).

Let us describe the general case in detail. Instead of 4 blocks we now split all vectors.
into $3(m-1)+1$ blocks defined by the vector

$$\ell := (\ell_1, \ell'_1, \ell_1', \ell_2, \ell'_2, \ldots, \ell_{m-1}, \ell'_{m-1}, \ell_m) \text{ with } \ell_m := n - \sum_{i=1}^{m-1} (\ell_i + 2\ell'_i).$$

We also introduce the sets

$$I_{i,j} := \begin{cases} \{3i - 2, 3i - 1\} & j \leq 2^{m-i-1} \\
\{3i - 2, 3i\} & j > 2^{m-i-1} \end{cases} \quad \text{for } j = 1, \ldots, 2^{m-i} \text{ and } i = 1, \ldots, m-1. \tag{3.5}$$

Starting with the $k = 2^m$ initial lists $L_1^{(0)}, \ldots, L_k^{(0)}$ of size $2^c$ the algorithm combines two lists at a time using the Sort-and-Match-algorithm to find matching elements on $\ell_i + \ell'_i$ coordinates of the blocks determined by the set $I_{i,j}$ in the creation of the list $L_j^{(i)}$ for $j = 1, \ldots, 2^{m-i}$ on level $i$. This is repeated until only two lists are left. For the lists on level $i = 1, \ldots, m-2$ we have

$$L_j^{(i)} = \begin{cases} \{(x_{1,j}, \ldots, x_{2j}) \in L_{2j-1}^{(i-1)} \times L_{2j}^{(i-1)} | \sum_{h=1}^{2^i} (x_{h,j})_{[3i-2, 3i-1]} = 0\} & j \leq 2^{m-i-1} \\
\{(x_{1,j}, \ldots, x_{2j}) \in L_{2j-1}^{(i-1)} \times L_{2j}^{(i-1)} | \sum_{h=1}^{2^i} (x_{h,j})_{[3i-2, 3i]} = 0\} & j > 2^{m-i-1} \end{cases}$$

where $x_{h,j} := x_{2(j-1)+h}$ for $h = 1, \ldots, 2^i$. The first (resp. second) list is then filtered for relative weight $\leq \gamma$ on the blocks $3, 6, \ldots, 3(m-1)$ (resp. $2, 5, \ldots, 3(m-1) - 1$)
The Approx-8-Tree-algorithm

returning the lists

\[ L_1^{(m-1)} = \{(x_1, \ldots, x_2^{m-1}) \in L_1^{(m-2)} \times L_2^{(m-2)} \mid \sum_{h=1}^{2^{m-1}} (x_h)[3m-5,3m-4] = 0, \Delta \left( \sum_{h=1}^{2^{m-1}} (x_h)[3,6,\ldots,3(m-1)] \right) \leq \gamma \ell'_{m-1} \} \]

\[ L_2^{(m-1)} = \{(x_{2^{m-1}+1}, \ldots, x_{2^m}) \in L_3^{(m-2)} \times L_4^{(m-2)} \mid \sum_{h=2^{m-1}+1}^{2^m} (x_h)[3m-5,3m-3] = 0, \Delta \left( \sum_{h=2^{m-1}+1}^{2^m} (x_h)[2,5,\ldots,3(m-1)-1] \right) \leq \gamma \ell'_{m-1} \} \]

As a last step we construct the final list

\[ L_1^{(m)} = \{(x_1, \ldots, x_k) \in L_1^{(m-1)} \times L_2^{(m-1)} \mid \Delta \left( \sum_{j=1}^{k} (x_j)[3(m-1)+1] \right) \leq \gamma m \ell_m \} \]

via NN-Search on the last \( \ell_m \) bits for relative weight \( \leq \gamma_m \leq \frac{n-2 \sum_{i=1}^{m-1} \ell_i}{\ell_m} \gamma \) out of those two lists. Analogously to the 4-list case this last list only contains solutions to the approximate \( k \)-list problem with target weight \( \gamma n \). The following theorem shows that the approximate \( k \)-List problem can be solved by Alg. 3.4 in expectation for \( k = 2^m \).
We use the previously described algorithm and set

\[ H \] for \[ 1 \]

\[ \ell_m := n - \sum_{i=1}^{m-1} (\ell_i + 2\ell_i'), \quad \gamma_m \in [0, 1] \]

Define \( \gamma := \frac{w}{n} \) and \( I_{i,j} \) for \( j = 1, \ldots, 2^{m-i}, i = 1, \ldots, m-1 \) as in (3.5).

1. for \( i = 1, \ldots, m-1 \) do
   1. for \( j = 1, \ldots, 2^{m-i} \) do
      \[ L_j^{(i)} \leftarrow \text{SORT-AND-MATCH}(L_{2j-1}^{(i-1)}, L_{2j}^{(i-1)}, I_{i,j}, 0) \]
   end
2. for \( i = 1, \ldots, m-1 \) do
   \[ L_{1}^{(m-1)} \leftarrow \text{FILTER}(L_{1}^{(m-1)}, 3i, \gamma \ell_{i}') \]
   \[ L_{2}^{(m-1)} \leftarrow \text{FILTER}(L_{2}^{(m-1)}, 3i - 1, \gamma \ell_{i}') \]
end
3. \( L_{1}^{(m)} \leftarrow \text{NN-SEARCH}(L_{1}^{(m-1)}, L_{2}^{(m-1)}, 3(m - 1) + 1, \gamma m \ell_m) \)
if \( |L_{1}^{(m)}| > 0 \) then return some \((x_1, \ldots, x_{2^m}) \in L_{1}^{(m)}\) else return \( \perp \)

**Theorem 3.2.** Let \( k = 2^m > 2 \) with \( m \in \mathbb{N}, 0 \leq \gamma \leq \frac{1}{2} \) with \( \gamma n \in \mathbb{N}, \gamma^* > 0 \) with 

\[ H(\gamma^*) = \frac{1+2(2m-4)\tau_2(\gamma^*)}{1+2(2m-2)\tau_2(\gamma^*)} \text{ and} \]

\[ \tau_k(\gamma) := \begin{cases} \frac{2-2H(\gamma)}{1-H(\gamma)+2m\tau_2(\gamma)+(2-2m)H(\gamma)\tau_2(\gamma)} \tau_2(\gamma) & \gamma \leq \gamma^* \ 
\frac{2-2H(\gamma)}{1+2m\tau_2(\gamma)} \tau_2(\gamma) & \gamma \geq \gamma^* \end{cases} \]  

(3.6)

For any constant \( \varepsilon > 0 \), the approximate \( k \)-list problem with target weight \( \gamma n \) can in expectation be solved in time and memory \( T = M = 2^{(\tau_k(\gamma^*)+\varepsilon)n} \).

**Proof.** We use the previously described algorithm and set

\[ \gamma_m := \gamma, \quad w := \gamma n, \quad c := \tau_k(\gamma)n \]  

(3.7)

\[ \ell_1 = \ldots = \ell_{m-1} := \begin{cases} 1-H(\gamma)+2m\tau_2(\gamma)+(2-2m)H(\gamma)\tau_2(\gamma) & (m-1)(1-H(\gamma)+2m\tau_2(\gamma)+(2-2m)H(\gamma)\tau_2(\gamma))n 
\geq \gamma^* \ 
0 & \gamma \leq \gamma^* \end{cases} \]

\[ \ell_1' = \ldots = \ell_{m-2}' := \begin{cases} H(\gamma)+2\tau_2(\gamma)-1 & (m-1)(1-H(\gamma)+2m\tau_2(\gamma)+(2-2m)H(\gamma)\tau_2(\gamma))n 
\geq \gamma^* \ 
c & \gamma \leq \gamma^* \end{cases} \]

\[ \ell_1'' := \begin{cases} \ell_1' & \gamma \leq \gamma^* 
H(\gamma)+(2-2m)\tau_2(\gamma)+(2m-4)H(\gamma)\tau_2(\gamma)-1 & 2+4m\tau_2(\gamma) \end{cases} \]

\[ \ell \geq \gamma^* \]

46
Let \( S_0 := |L_j^{(0)}| = 2^c, \quad j = 1, \ldots, 2^m \) and

\[
S_{m-1} := 2^{1-H(\gamma)} \ell_m = 2^{2c-\ell_{m-1}+(H(\gamma)-2)\ell_{m-1}}.
\]

By Lemma 1.2, Lemma 1.3 and Lemma 1.4 we have

\[
E[|L_j^{(i)}|] = \frac{S_0}{2^{i+\ell_i}} = S_0, \quad j = 1, \ldots, 2^{m-1}, i = 1, \ldots, m-2
\]

\[
E[|L_j^{(m-1)}|] = \frac{S_0}{2^{\ell_m+\ell_{m-1}}}, \sum_{i=0}^{\infty} \frac{\ell_{m-1}}{2^{\ell_{m-1}}}
\]

\[
\Rightarrow \frac{S_{m-1}}{\text{poly}(n)} \leq E[|L_j^{(m-1)}|] \leq S_{m-1} \cdot 2^{cn}, \quad j = 1, 2
\]

\[
E[|L_j^{(m)}|] = E[|L_j^{(m-1)}|] \cdot \sum_{i=0}^{\infty} \frac{\ell_m}{2^{\ell_m}} \geq \frac{S_{m-1}}{\text{poly}(n)} \cdot \sum_{i=0}^{\infty} \frac{\ell_m}{2^{\ell_m}} \geq \frac{1}{\text{poly}(n)}
\]

since \( 2^{(1-H(\gamma))\ell_m+(H(\gamma)-1)\ell_m} = 1 \). Furthermore every element in \( L_j^{(m)} \) has hamming weight \( 2\gamma \sum_{i=1}^{m-1} \ell'_i + \ell_m \leq \gamma n \). Therefore, as stated in Lemma 3.1, the algorithm in expectation returns one solution for the approximate \( k \)-list problem with target weight \( \gamma n \). By Lemma 1.2 the Sort-and-Match steps in each iteration in step 1 take time

\[
T_i := 2^{\max\{c,2c-\ell_{i+1}-\ell'_i+cn\}} = 2^{(\tau_k(\gamma)+\varepsilon)n}, \quad i = 0, \ldots, m - 3
\]

\[
T_{m-2} := 2^{\max\{c,2c-\ell_{m-1}-\ell'_m+cn\}}
\]

on level \( i \) while by Lemma 1.3, \( T_{m-2} \) is also an upper bound for the running time of the filterings in step 2. Since \( S_{m-1} = 2^{1-H(\gamma)}\ell_m \), the nearest Neighbor search for relative weight \( \gamma \) on \( \ell_m \) bits in step 3 is equivalent to our approximate 2-tree-algorithm. Thus we can apply Lemma 3.1 again and obtain the running time

\[
T_{m-1} := 2^{\tau_k(\gamma)\ell_m+cn} = 2^{(\tau_k(\gamma)+\varepsilon)n}
\]

For \( \gamma \leq \gamma^* \) we have \( c = 2c - \ell_{m-1} - \ell'_m \) while for \( \gamma \geq \gamma^* \) we can use the fact that \( H(\gamma) \geq \frac{1+(2m-4)\tau_k(\gamma)}{1+(2m-2)\tau_k(\gamma)} \) which is equivalent to \( c \geq 2c - \ell_{m-1} - \ell'_m \). Since \( c = (\tau_k(\gamma))n \) we have \( T_0 = \ldots = T_{m-1} \) and therefore the expected run-time is \( T = 2^{(\tau_k(\gamma)+\varepsilon)n} \). Since the memory consumption is determined by the list sizes and \( S_0 \geq S_{m-1} \), we have \( M = 2^{(\tau_k(\gamma)+\varepsilon)n} \).

Note that this result only holds for \( k > 2 \) while for \( k = 2 \) we have the result of Lemma 3.1 again. For the latter case we cannot balance running time and memory.

47
since we only have one computation step - a nearest neighbor search. Nevertheless Alg. 3.4 can be applied for the case \( k = 2 \). Figure 3.7 visualizes the relative running time exponents \( \tau_k(\gamma) \) of our Gen-Approx-\( k \)-Tree-algorithm with parameters as defined in Theorem 3.2. We would like to point out that for \( \gamma = 0 \) the algorithm collapses to the original \( k \)-Tree-algorithm again and therefore we achieve the same running times. Furthermore both cases in Eq. 3.6 are equal for \( \gamma = \gamma^* \). Therefore the running time exponent’s function of \( \gamma \) is continuous.

For \( \gamma \leq \gamma^* \) the algorithm returns a solution of weight \( \gamma \cdot (2(m-1)\ell'_1 + \ell_m) = \gamma(n-(m-1)\ell_1) \) which is strictly smaller than \( \gamma n \). Thus the choice of parameters given by Eq. 3.7 in Theorem 3.2 is not optimal for this case. We sacrificed optimality here in order to obtain a closed formula for the running time. Choosing

\[
\ell_1 = \ldots = \ell_{m-2} = c, \quad \ell_{m-1} = \ell \\
\ell'_1 = \ldots = \ell'_{m-1} = 0 \\
\gamma_m = \frac{\gamma n}{\ell_m} = \frac{\gamma n}{n - (m-2)c - \ell'}
\]

we omit the filtering in step 2 and therefore achieve a symmetric search tree again. In fact, this results in our previous algorithm Approx-\( k \)-Tree (Alg. 3.2) which is therefore a special case of Gen-Approx-\( k \)-Tree (Alg. 3.4).

The analysis in this chapter only holds if the initial lists are large enough or the size can be freely chosen. In the last chapter we already introduced the extended \( k \)-tree-algorithm by Minder and Sinclair [MS09] which adapts Wagner’s \( k \)-Tree-algorithm to the case, where one starts with smaller sizes. The same techniques can be adapted here.
3.4 Comparison of Our Algorithms

In this chapter we introduced several algorithms for the approximate $k$-list problem. We started with the simple Match-and-Filter-algorithm (Alg. 3.1) and continued with our new APPROX-$k$-Tree-algorithm (Alg. 3.2). For the latter we described the generalization Gen-APPROX-$k$-Tree (Alg. 3.4) which provides provable complexity (Lemma 3.1 and Theorem 3.2) for specific parameters. Let us now compare those different algorithms. Fig. 3.8 provides complexities for different choices of $m$ and $\gamma$ while Fig. 3.9 takes a closer look at the case $m = 3$.

First, we would like to mention that the simple Match-and-Filter-algorithm achieves better running times than our Gen-APPROX-$k$-Tree-algorithm. However, it
does not provide a closed formula. Another downside of Match-and-Filter is the fact that this algorithm is more restricting, fixing a larger region of all-zero bits in the target vector. For applications, where one searches specific target vectors, this might be an issue. Furthermore, Approx-$k$-Tree with minimized running time is superior for arbitrary choices of $\gamma$ and $m > 1$. This holds analogously for the variant with minimized memory which has the lowest memory consumption of all algorithms and a lower running time in most cases compared to the provable variant. Hence, the results of Lemma 3.1 and Theorem 3.2 can be seen as upper bounds for the approximate $k$-list problem while Theorem 3.1 yields more practical results. Notice, that we have $M = T$ for Match-and-Filter, Approx-$k$-Tree with minimized running time as well as Gen-Approx-$k$-Tree.

### 3.5 Improvements for $k = 2^m + 2^{m-1}$

In this section we propose algorithms for instances of the approximate $k$-list problem where $k$ is of the special form $2^m + 2^{m-1}$ for $m \geq 1$. We show that we can improve upon the case $2^m$ – in contrast to the $k$-list problem.

**The case $k = 3$.** Let us first look at the simplest case again (see Figure 3.10).

We introduce a split into 2 blocks to all vectors in the three initial lists named $L_1^{(0)}, L_2^{(0)}, L_3^{(0)}$. The first block consists of $\ell_1$ and the second block of $\ell_2 := n - \ell_1$ coordinates. The first two lists $L_1^{(0)}, L_2^{(0)}$ are of size $2^{c_1}$ while the third list $L_3^{(0)}$ is of size $2^{c_3}$. The parameters $c_1, c_3$ and $\ell_1$ are subject to optimization. In the first step a Sort-and-Match on the lists $L_1^{(0)}, L_2^{(0)}$ finds all pairs of vectors which are equal on the first $\ell$ bits. The list $L_3^{(0)}$ is filtered for elements which have relative weight $\leq \gamma_1$ on
those bits. This results in two lists

\[ L_1^{(1)} = \{(x_1, x_2) \in L_1^{(0)} \times L_2^{(0)} \mid (x_1 + x_2)_{[1]} = 0\} \]
\[ L_2^{(1)} = \{x_3 \in L_3^{(0)} \mid \Delta((x_3)_{[1]}) \leq \gamma_1 \ell_1\}. \]

In the second step we use NN-Search for those two lists, finding vectors adding up to some relative weight \( \leq \gamma_2 \leq \frac{\gamma n - \gamma_1 \ell_1}{\ell_2} \) on the last \( \ell_2 \) bits. Notice that all elements have relative weight \( \leq \gamma_1 \) on the first \( \ell_1 \) bits by construction. Thus the final list

\[ L_1^{(2)} = \{(x_1, x_2, x_3) \in L_1^{(1)} \times L_2^{(1)} \mid \Delta(\sum_{j=1}^{3} (x_j)_{[2]}) \leq \gamma_2 \ell_2\}. \]

contains solutions to the approximate 3-list problem since \( \gamma_1 \ell_1 + \gamma_2 \ell_2 \leq \gamma n \). More details can be found in Alg. 3.5. The following example shows, how the Approx-3-Tree-algorithm finds a solution to the approximate 3-list problem.

**Algorithm 3.5: Approx-3-Tree**

Input : \( L_j^{(0)} \subset \mathbb{F}_2^n \), for \( j = 1, 2, 3 \), \( w \in \mathbb{N} \)

Output : \((x_1, x_2, x_3) \in \mathbb{F}_2^n \times \mathbb{F}_2^n \times \mathbb{F}_2^n\)

Parameters: \( \ell = (\ell_1, \ell_2) \in \mathbb{N}^2 \) with \( \ell_2 := n - \ell_1 \), \( \gamma_1, \gamma_2 \in [0, 1] \)

1. \( L_1^{(1)} \leftarrow \text{Sort-And-Match}(L_1^{(0)}, L_2^{(0)}, 1, 0) \)
2. \( L_2^{(1)} \leftarrow \text{Filter}(L_3^{(0)}, 1, \gamma_1 \ell_1) \)
3. \( L_1^{(2)} \leftarrow \text{NN-Search}(L_1^{(1)}, L_2^{(1)}, 2, \gamma_2 \ell_2) \)

if \( |L_1^{(2)}| > 0 \) then return some \((x_1, x_2, x_3) \in L_1^{(2)}\) else return \( \perp \)
Example 3.2. Let
\[ L_1^{(0)} = \{0111, 1100, 0010\}, \quad L_2^{(0)} = \{1000, 1011, 0101\}, \]
\[ L_3^{(0)} = \{1100, 1011, 1111\} \]
be the three initial list with elements from \( \mathbb{F}_2^4 \) and let \( w \leq 2 \). In the first step the algorithm finds vectors in \( L_1^{(0)}, L_2^{(0)} \) matching on the first block of size \( \ell_1 \). Next the third list is filtered for weight \( \gamma_1 \ell_1 \). Assuming we have \( \ell_1 = 2 \) and \( \gamma_1 \ell_1 = 1 \), then
\[ L_1^{(1)} = \{(0111, 0101)\}, \quad L_2^{(1)} = \{1011\}. \]
Thus the nearest neighbor search on the remaining two bits for weight \( \leq 1 \) returns
\[ L_1^{(2)} = \{(0111, 0101, 1011)\}. \]
Solutions like \( (1100, 1000, 1100) \) do not survive the first step and are not found by the algorithm.

The following theorem shows that the approximate 3-List problem can be solved by Alg. 3.5 in expectation.

**Theorem 3.3.** Let \( 0 \leq \gamma \leq \frac{1}{2} \) with \( \gamma n \in \mathbb{N} \), \( \gamma^* > 0 \) with \( \tau_2(\gamma^*) = \frac{H(\gamma^*) - H(\gamma^*)^2}{4H(\gamma^*)-2} \) and
\[
\tau_3(\gamma) := \begin{cases} 
2-2H(\gamma) & \gamma \leq \gamma^* \\
\frac{2H(\gamma)^2 - H(\gamma^*)}{H(\gamma) + 4\tau_2(\gamma)} & \gamma \geq \gamma^* 
\end{cases}.
\]
For any constant \( \varepsilon > 0 \) the approximate 3-list problem with target weight \( w := \gamma n \) can in expectation be solved in time and memory \( T = M = 2^{(\tau_3(\gamma) + \varepsilon)n} \).
Proof. We use the previously described algorithm and set

\[
\begin{align*}
\gamma_1 &= \gamma_2 := \gamma, \quad w := \gamma n \\
c_1 &:= \begin{cases} 
H(\gamma)^2 - H(\gamma) + 2\tau_2(\gamma) & \gamma \leq \gamma^* \\
\tau_3(\gamma) n & \gamma \geq \gamma^* 
\end{cases} \\
c_3 &:= \begin{cases} 
\tau_3(\gamma) n & \gamma \leq \gamma^* \\
(1 - H(\gamma))(H(\gamma) + 4\tau_2(\gamma)) n & \gamma \geq \gamma^* 
\end{cases} \\
\ell_1 &:= \begin{cases} 
H(\gamma) + 2\tau_2(\gamma) - 1 & \gamma \leq \gamma^* \\
H(\gamma) + 4\tau_2(\gamma) - 1 & \gamma \geq \gamma^* 
\end{cases}
\end{align*}
\]

Let \( S_{0,1} := |L_{j_1}^{(0)}| = 2^{c_1}, j = 1, 2, S_{0,3} := |L_{j_3}^{(0)}| = 2^{c_3} \) and

\[
S_1 := 2^{\frac{1 - H(\gamma)}{2} \ell_2} = 2^{2c_1 - \ell_1} = 2^{c_3 + (H(\gamma) - 1) \ell_1}.
\]

By Lemma 1.2 Lemma 1.3 and Lemma 1.4 we have

\[
\begin{align*}
\mathbb{E}[|L_{j_1}^{(1)}|] &= \frac{S_{0,1}^2}{2^{\ell_1}}, \quad \mathbb{E}[|L_{j_2}^{(1)}|] = S_{0,3} \cdot \sum_{i=0}^{\gamma \ell_1} \frac{\binom{\ell_1}{i}}{2^{\ell_1}} \\
\frac{S_1}{\text{poly}(n)} &\leq \mathbb{E}[|L_{j_1}^{(1)}|] \leq S_1 \cdot 2^{c_n}, \quad j = 1, 2 \\
\mathbb{E}[|L_{j_1}^{(2)}|] &= \mathbb{E}[|L_{j_1}^{(1)}|^2] \cdot \sum_{i=0}^{\gamma \ell_2} \frac{\binom{\ell_2}{i}}{2^{\ell_2}} \geq \frac{S_1^2}{\text{poly}(n)} \cdot \sum_{i=0}^{\gamma \ell_2} \frac{\binom{\ell_2}{i}}{2^{\ell_2}} \geq \frac{1}{\text{poly}(n)}
\end{align*}
\]

since \( 2^{(1 - H(\gamma)) \ell_2 + \ell_2(H(\gamma) - 1)} = 1 \). Furthermore every element in \( L_{j_1}^{(2)} \) has hamming weight \( \leq \gamma \ell_1 + \gamma \ell_2 = \gamma n \) and therefore the algorithm returns one solution for the approximate 3-list problem with approximation \( \gamma \) in expectation. The Sort-And-Match step 1 takes time

\[
T_{0,1} := 2^{\max\{c_1, 2c_1 - \ell_1\} + c_n}
\]

by Lemma 1.2 while by Lemma 1.3 the Filtering step 2 runs in time

\[
T_{0,3} := 2^{c_3 + c_n}.
\]

Since \( S_1 = 2^{\frac{1 - H(\gamma)}{2} \ell_2} \), the nearest neighbor search for relative weight \( \gamma \) on \( \ell_2 \) bits in step 3 is equivalent to our approximate 2-tree-algorithm. Thus we can apply Lemma 3.1.
again and obtain the running time

\[
T_1 := 2^{\gamma_2(\gamma)\ell_2 + \varepsilon n} = \begin{cases} 
2^{c_3 + \varepsilon n} & \gamma \leq \gamma' \\
2^{c_1 + \varepsilon n} & \gamma \geq \gamma'
\end{cases} = 2^{(\tau_3(\gamma) + \varepsilon)n}
\]

We furthermore have

\[
S_{0,3} \geq S_1 \Leftrightarrow c_3 \geq 2c_1 - \ell_1 \quad \text{and} \quad 
\gamma \geq \gamma^* \Leftrightarrow \tau_2(\gamma) \geq \frac{H(\gamma) - H(\gamma)^2}{4H(\gamma) - 2} \Leftrightarrow c_1 \geq c_3
\]

and thus the overall running time is

\[
T = \max\{T_{0,1}, T_{0,3}, T_1\} = 2^{\max\{c_1, c_3\} + \varepsilon n} = \begin{cases} 
2^{c_3(1+\varepsilon)} & \gamma \leq \gamma' \\
2^{c_1(1+\varepsilon)} & \gamma \geq \gamma'
\end{cases} = 2^{(\tau_3(\gamma) + \varepsilon)n}
\]

Since \(S_{0,3} \geq S_1\) the memory consumption in both cases is determined by the list sizes \(S_{0,1}\) and \(S_{0,3}\), i.e.

\[
M = 2^{\max\{c_1, c_3\} + \varepsilon n} = 2^{(\tau_3(\gamma) + \varepsilon)n}.
\]

Again, both cases in Eq. 3.3 are equal for \(\gamma = \gamma^*\). Therefore the running time exponent’s function of \(\gamma\) is continuous. Analogously to Theorem 3.2 we chose \(\gamma_1 = \gamma_2 = \gamma\) to obtain a closed formula for running time and memory consumption. Numerical optimizations of the two parameters further minimize the running time similar to the result of Theorem 3.1.

The results are presented and compared to the APPROX-2-TREE-algorithm (see Lemma 3.1) in Figure 3.11 and Figure 3.12. The APPROX-3-TREE-algorithm provides a decreased running time exponent for all non-trivial choices of \(\gamma\). The biggest improvement is achieved around \(\gamma^*\).

<table>
<thead>
<tr>
<th>(\gamma)</th>
<th>(k = 2)</th>
<th>(k = 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.3334</td>
<td>0.3334</td>
</tr>
<tr>
<td>0.1</td>
<td>0.3004</td>
<td>0.2818</td>
</tr>
<tr>
<td>0.2</td>
<td>0.1774</td>
<td>0.1558</td>
</tr>
<tr>
<td>0.3</td>
<td>0.0861</td>
<td>0.0721</td>
</tr>
<tr>
<td>0.4</td>
<td>0.0244</td>
<td>0.0210</td>
</tr>
</tbody>
</table>

Fig. 3.11: Comparison of \(\frac{\log(T)}{n}\) for APPROX-3-TREE and its 2-tree counterpart.
The Case $k = 2^m + 2^{m-1}$. The idea from above can be generalized leading to an algorithm which combines the techniques used in Approx-$k$-Tree and Approx-3-Tree. We do not give a detailed description of some algorithm for $k = 2^m + 2^{m-1}$ here, since the notation would get too confusing. We hope the reader already acquired a good intuition for our algorithms throughout the previous sections and provide examples for $k = 6$ and $k = 12$ lists in Figure 3.14 and 3.15. Similar to the 3-list case, we do some Sort-and-Match on the left side of the binary search tree while we filter on the smaller, right side. Notice, that for the general case we have more than 2 lists on the right side which are combined via Sort-and-Match first. Figure 3.13 provides results of numerical optimizations for both $k = 6$ and $k = 12$ and compares them to their 4-tree respectively 8-tree counterpart. As for $k = 3$ we get a small but remarkable improvement.

<table>
<thead>
<tr>
<th>$\gamma$</th>
<th>$k = 4$</th>
<th>$k = 6$</th>
<th>$k = 8$</th>
<th>$k = 12$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.3334</td>
<td>0.3334</td>
<td>0.2500</td>
<td>0.2500</td>
</tr>
<tr>
<td>0.1</td>
<td>0.2001</td>
<td>0.1975</td>
<td>0.1564</td>
<td>0.1532</td>
</tr>
<tr>
<td>0.2</td>
<td>0.1204</td>
<td>0.1152</td>
<td>0.0961</td>
<td>0.0926</td>
</tr>
<tr>
<td>0.3</td>
<td>0.0611</td>
<td>0.0573</td>
<td>0.0506</td>
<td>0.0479</td>
</tr>
<tr>
<td>0.4</td>
<td>0.0190</td>
<td>0.0180</td>
<td>0.0167</td>
<td>0.0159</td>
</tr>
</tbody>
</table>

Fig. 3.13: Running time exponent $\frac{\log T}{n}$ for $k = 4, 6, 8$ and 12.
Fig. 3.14: An algorithm for the approximate 6-list problem.

Fig. 3.15: An algorithm for the approximate 12-list problem.
3.6 Application: Finding Low Weight Polynomials

In Section 2.4 we showed that Wager’s original $k$-Tree-algorithm has various applications like finding collisions for specific hash functions or solving the parity check problem. Our algorithms presented in this chapter can analogously be used to find near collisions for those hash functions, but is more useful in the context of the parity check problem. In this section we show that our algorithms improve upon the results obtained via the original $k$-Tree-algorithm and its variant the extended $k$-tree-algorithm. Since we aim for very small weight in general, we use MATCH-AND-FILTER which is the simplest algorithm presented in this chapter. Another reason for this choice is that we cannot easily compute the polynomial overhead for the May-Ozerov nearest neighbor search.

Solving the Parity Check Problem. First, recall the definition of the parity check problem (Def. 2.3) from Section 2.4.

**Definition 3.2 (Parity Check Problem).** Let $n,d,N \in \mathbb{N}$ and let $P(X)$ be an irreducible polynomial over $\mathbb{F}_2$ of degree $n$. One has to find a multiple $Q(X)$ of $P(X)$ of weight smaller than $d$ and degree smaller than $N$. Typically one has $d \ll n$ and $N \gg n$.

The connection to the approximate $k$-list problem is similar to the one in Section 2.4. Let again $k = 2^m = 2^{\lfloor \log d \rfloor}$ be the number of lists given to the algorithm and let $w := d - k$ be the target weight. We fill each list with $2^c$ elements of the form $X^a \mod P(X) \in \mathbb{F}_2$, $a \in \{1,2,\ldots,N\}$.

We now pairwise merge the lists via SORT-AND-MATCH on $c$ coordinates and repeat this until only one list is left. It contains elements of the form $X^{a_1} + \ldots + X^{a_k}$ whose $m \cdot c$ leftmost coefficients add to zero. In the last step, we filter for weight $w$ on the remaining coordinates and eventually find a list $L'$ containing small-weight polynomials of the form

$$S(X) := X^{a_1} + \ldots + X^{a_k} = C(X) \mod P(X).$$

Since we have

$$S(X) + C(X) = 0 \mod P(X),$$

$S(X) + C(X)$ is a multiple of $P(X)$ with maximum degree $N$. Furthermore we have

$$|S(X) + C(X)| \leq |S(X)| + |C(X)| \leq k + w = d.$$
Using Eq. 3.2 we obtain

\[ E[|L|] = 2^{(m+1)c-n} \sum_{i=0}^{w} \binom{n-m \cdot c}{i}. \]

Solving \( E[|L|] = 1 \) yields the required initial list size \( 2^c \) and therefore the maximum degree \( N = 2^c \). For our analysis in this section, we assume that Match-and-Filter uses time and memory (number of lists \( \cdot \) size of a list), i.e. \( T = M = kN \).

**Comparison with the \( k \)-tree and the extended \( k \)-tree-algorithm.** As shown in Section 2.4, the \( k \)-Tree-algorithm by [Wag02] returns a polynomial of degree \( N = 2^{\frac{n}{d+1}} \) determined by the number of lists \( k = 2^m \) in time and memory \( T = M = kN \). Since the extended \( k \)-tree-algorithm by Minder and Sinclair [MS09] also works for smaller list sizes than Wagner’s algorithm, we automatically obtain polynomials of lower degree. Figure 3.16 provides results for the \( k \)-Tree-algorithm and minimized running time for Match-and-Filter as well as the corresponding degree \( N \). For a good comparability the presented running times for the extended \( k \)-tree-algorithm correspond to the same fixed degrees, although other degrees are possible. Additionally, we present the degree \( N \) as obtained by Match-and-Filter in the same running time as the extended \( k \)-tree-algorithm.

<table>
<thead>
<tr>
<th>( d )</th>
<th>( k )</th>
<th>( k )-Tree [Wag02]</th>
<th>( \log(N) )</th>
<th>( \log(T) )</th>
<th>Extended ( k )-Tree [MS09]</th>
<th>( \log(N) )</th>
<th>( \log(T) )</th>
<th>Match-and-Filter Min.</th>
<th>( \log(T) )</th>
<th>( \log(N) )</th>
<th>Fixed</th>
<th>( \log(T) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>60</td>
<td>61</td>
<td>60</td>
<td>61</td>
<td>60</td>
<td>61</td>
<td>60</td>
<td>61</td>
<td>60</td>
<td>61</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>60</td>
<td>61</td>
<td>60</td>
<td>61</td>
<td>57</td>
<td>58</td>
<td>57</td>
<td>58</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>40</td>
<td>42</td>
<td>40</td>
<td>42</td>
<td>40</td>
<td>42</td>
<td>40</td>
<td>42</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>40</td>
<td>42</td>
<td>39</td>
<td>43</td>
<td>39</td>
<td>41</td>
<td>36</td>
<td>43</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>40</td>
<td>42</td>
<td>37</td>
<td>47</td>
<td>37</td>
<td>39</td>
<td>32</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>40</td>
<td>42</td>
<td>36</td>
<td>49</td>
<td>36</td>
<td>38</td>
<td>28</td>
<td>49</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>30</td>
<td>33</td>
<td>30</td>
<td>33</td>
<td>30</td>
<td>33</td>
<td>30</td>
<td>33</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>30</td>
<td>33</td>
<td>29</td>
<td>33</td>
<td>29</td>
<td>32</td>
<td>27</td>
<td>33</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>8</td>
<td>30</td>
<td>33</td>
<td>24</td>
<td>38</td>
<td>24</td>
<td>27</td>
<td>14</td>
<td>38</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3.16: Comparison of the bit complexities and maximum degrees for finding a multiple of a polynomial of degree \( n = 120 \).

First we notice that all three algorithms collapse to \( k \)-Tree if \( d \) is a power of two and thus \( k = d \). Otherwise, the extended \( k \)-tree-algorithm can obtain lower degrees \( N \) compared to \( k \)-Tree at a cost of a higher complexity. However, Match-and-Filter yields better complexities providing the same degree \( N \). Furthermore, Match-and-Filter provides a lower degree than Minder-Sinclair in the same time. Moreover,
their algorithm achieves a minimum degree of $2^n$ (as k-Tree) only, while we can go below this bound for $d \neq 2^n$. The reason is that our approximate search returns more solutions than an exact collision search allowing smaller initial lists.

Low weight polynomials are of major importance in fast correlation attacks. Since its degree determines the number of output bits an attacker has to know in the attack, a low degree is preferable. Investing more time for finding such polynomials usually pays off in total. Our algorithm improves over k-TREE and the extended k-tree-algorithm. Currently most fast correlation attacks are restricted to weight 4 or 5, while our algorithm provides more flexibility for the choice of weights.

3.7 Open Problems

We conclude this chapter with some open problems.

**Arbitrary** $k$. In contrast to the original $k$-list problem we were able to provide improved results for $k \neq 2^m$, namely for special $k = 2^m + 2^{m-1}$. However, we were not able to find algorithms for choices like $k = 5$ or $k = 7$ which improve upon $k = 4$ respectively $k = 6$. Moreover the algorithms for $k = 2^m + 2^{m-1}$ only provide a minor improvement upon $k = 2^m$. Thus it remains to find algorithms for arbitrary choices of $k$ with continuously decreasing running time for growing $k$.

**Alternative Nearest Neighbor Search.** Throughout this chapter we always used the nearest neighbor search by May and Ozerov. However, we handled this algorithm as a black box and therefore it can be replaced by any other nearest neighbor search. An algorithm superior to May-Ozerov would immediately improve our results. The existence of such an algorithm is an open question. In addition to this the restriction $\lambda < 1 - \frac{2}{\gamma}$ prevented us from using nearest neighbor search on every layer of the search tree like we do in our algorithm (Alg. 5.3) presented in Chapter 5 for the decoding problem. Hence, an alternative algorithm might pave the way for a whole new approach to the approximate $k$-list problem.

**Golden Solution.** Analogously to the $k$-Tree-algorithm we search for solutions with a special structure possibly throwing away an exponential amount of solutions. Although the structure can be randomized (see. Section 2.2), this has to be predetermined before our actual algorithm starts. Thus it is not designed to find some specific golden solution like it is possible with the algorithm by Schroeppele and Shamir (Alg. 2.2). As for the $k$-list problem this raises the question if there exists an efficient algorithm which finds a golden solution or even all solutions.
Failure Probability. Our analysis in this as well as in the last chapter followed Wagner’s work [Wag02] and concentrated on expected values for both complexity and the number of solutions. Although this already gives good estimates it remains open to analyze the failure probability of our algorithms like it was done for the extended $k$-tree-algorithm (and thus for $k$-Tree) in [MS09].
Chapter 4

Information Set Decoding

The decoding of random linear codes is a major problem in complexity theory. Since it is an NP-hard problem, it is especially interesting for the construction of cryptographic systems. The most efficient solving algorithms belong to the class of information set decoding (ISD) algorithms. After an introduction to linear codes, this chapter provides an overview over the history of ISD algorithms. We start with Prange’s algorithm [Pra62] which is the basis for most later work. We then continue with Stern’s algorithm [Ste88] which introduced meet-in-the-middle techniques to information set decoding as well as ball collision decoding by Bernstein et al. [BLP11]. We conclude with the description of an ISD algorithm by Finiasz and Sendrier [FS09] and the BJMM-algorithm [BJMM12] which is a follow up work of [MMT11].

4.1 Syndrome Decoding

Let us give some basic definitions first.

**Definition 4.1 (Linear Code).** A linear code $C$ is a $k$-dimensional subspace of $F_2^n$. The distance of $C$ is defined by $d := \min_{c \neq c' \in C} \{\Delta(c, c')\}$.

One can specify a linear code $C$ via a generator matrix $G \in F_2^{k \times n}$ or a parity check matrix $P \in F_2^{(n-k) \times n}$ via

$$C := \{xG \in F_2^n \mid x \in F_2^k\} \quad \text{or} \quad C := \{c \in F_2^n \mid Pc = 0\}.$$  

If $G \in R^{k \times n}$ or $P \in R^{(n-k) \times n}$, i.e. each matrix entry is chosen uniformly at random from $F_2$, then we call $C$ a random linear code.

The main computational problem in coding theory is the syndrome decoding problem. The security of many cryptographic constructions relies on the hardness of this problem. It is named after the so called syndrome of some arbitrary vector.
Definition 4.2 (Syndrome). Let $C \subseteq \mathbb{F}_2^n$ be a linear code with parity check matrix $P \in \mathbb{F}_2^{(n-k) \times n}$ and let $y := c + e$ for $c \in C$ and $e \in \mathbb{F}_2^n$. We define the syndrome $s \in \mathbb{F}_2^{n-k}$ of $y$ by

$$s := Py = Pc + Pe = Pe. \quad (4.1)$$

We now give a formal definition of the syndrome decoding problem.

Definition 4.3 (Syndrome Decoding Problem). Let $P \in \mathbb{F}_2^{(n-k) \times n}$ be a parity check matrix specifying a random linear code $C$. Given $P$, an (erroneous) codeword $y \in \mathbb{F}_2^n$ and some Hamming weight $w \in [n]$ one has to find an error vector $e \in \mathbb{F}_2^n$ with $y + e \in C$ and $\Delta(e) = w$.

We denote an instance of the syndrome decoding problem by $(P, s, w)$ with $s = Py$. The error vector $e \in \mathbb{F}_2^n$ solves $(P, s, w)$ if and only if $s = Pe$ and $\Delta(e) \leq w$.

If $w \geq \frac{n}{2}$, a randomly chosen preimage $e$ satisfying $s = Pe$ has weight $\leq w$ with good probability and thus solves the instance. On the other hand, if $w$ is constant, the problem becomes easy since we can simply use Brute-Force and enumerate all possible vectors $e \in \mathbb{F}_2^n$ with $\Delta(e) = w$. Thus we mostly use the notation of error rates $\frac{w}{n}$ and code rates $\frac{k}{n}$. In this chapter we look at the two most used settings where $w$ is either upper bounded by the code distance $d$ (full distance decoding) or by $\frac{d}{2}$ (half distance decoding). Notice, a solution to the syndrome decoding problem is unique for the latter setting. Any algorithm which aims to solve an instance of the syndrome decoding problem in either full distance or half distance decoding, has a running time depending on the parameters $n, k, d$. However, for random linear codes, we can use the Gilbert-Varshamov bound

$$\frac{k}{n} \approx 1 - H\left(\frac{d}{n}\right)$$

and thus the running time becomes a function $T(n, k)$ of $n$ and $k$ only. Furthermore we often compare worst case running times maximized over all rates $\frac{k}{n}$ resulting in a running time $T(n)$ only. All algorithms run in time exponential in $n$, i.e. $T(n) = 2^{(\tau+\varepsilon)n}$ for some $\tau > 0$ and any constant $\varepsilon > 0$. The constant $\tau$ is used as a metric to compare and evaluate different algorithms. Like in the previous chapters, polynomial factors contribute to the error term $\varepsilon$ again.

The following lemma helps us to find equivalent instances for a given syndrome decoding problem.

Lemma 4.1. Let $(P, s, w) \in \mathbb{F}_2^{(n-k) \times n} \times \mathbb{F}_2^{n-k} \times [n]$ be an instance of the syndrome decoding problem, $P' := GP\pi$ and $s' := Gs$ for some invertible matrix $G \in \mathbb{F}_2^{(n-k) \times (n-k)}$. Compare this to the previous chapters,
and some permutation \( \pi \in \mathbb{F}_2^{n \times n} \). We then have

\[ e \text{ is a solution of } (P, s, w) \iff \pi^{-1}e \text{ is a solution of } (P', s', w). \]

Proof. The statement directly follows from \( \Delta(e) = \Delta(\pi^{-1}e) \) and

\[ Pe = s \iff GP\pi^{-1}e =Gs \iff P'\pi^{-1}e = s'. \]

Thus linear algebra transformations on the parity check matrix \( P \) result in an equivalent instance of the syndrome decoding problem. As we only consider random linear codes, the only structure a decoding algorithm can exploit are the known weight \( w \) and the vector space structure. The main idea of ISD algorithms is to exploit the latter. Following above lemma one can reduce the dimension of the search space through applying linear algebra transformations on the parity check matrix \( P \).

### 4.2 Simple Information Set Decoding

A first major step towards solving the syndrome decoding problem was achieved by Prange in 1962 [Pra62]. He introduced the first information set decoding algorithm forming the basis for today’s improvements. In a nutshell, one uses Gaussian elimination to reduce the search space’s dimension from \( n \) down to \( k \) and hopes that a permutation of \( e \) contains only zeros on \( k \) coordinates.

**Prange’s Algorithm.** Let us explain the idea of Prange’s algorithm (see Fig. 4.1) in more detail now. As a first step one applies some column permutation matrix \( \pi \in \mathbb{F}_2^{n \times n} \) to \( P \) to enforce a special weight distribution on \( \pi^{-1}e \). Secondly one realizes the dimension reduction through Gaussian elimination, i.e. choosing some invertible \( G \in \mathbb{F}_2^{(n-k) \times (n-k)} \) such that \( GP\pi = (\bar{P} \mid I_{n-k}) \), where \( I_{n-k} \) is the \((n-k)\)-dimensional identity matrix. In total we transform Eq. 4.1 into

\[ \bar{s} := Gs = GP\pi^{-1}e = (\bar{P} \mid I_{n-k})\pi^{-1}e = \bar{P}e' + e'', \quad (4.2) \]

where \( \pi^{-1}e =: (e', e'') \in \mathbb{F}_2^k \times \mathbb{F}_2^{n-k} \). We call the instance given by above equation the standard form of \((P, s, w)\).

**Definition 4.4 (Standard Form).** Let \((P, s, w) \in \mathbb{F}_2^{(n-k) \times n} \times \mathbb{F}_2^{n-k} \times [n]\) be an instance of the syndrome decoding problem. We say that \((\bar{P}, \bar{s}, w) \in \mathbb{F}_2^{(n-k) \times k} \times \mathbb{F}_2^{n-k} \times [n]\)
Fig. 4.1: Dimension reduction and weight distribution for Prange’s algorithm.

is a standard form of \((P, s, w)\) if there exists some invertible \(G \in \mathbb{F}_2^{(n-k) \times (n-k)}\) and \(\pi \in \mathbb{F}_2^{n \times n}\) such that

\[
GP\pi = (\bar{P} | \bar{I}_{n-k}) \quad \text{and} \quad Gs = \bar{s}.
\]

The vector \((e', e'') \in \mathbb{F}_2^k \times \mathbb{F}_2^{n-k}\) solves the standard form \((\bar{P}, \bar{s}, w)\) if and only if \(\bar{s} = \bar{P}e' + e''\) and \(\Delta((e', e'')) = w\).

It directly follows from Lemma 4.1 that solving the dimension-reduced standard form is equivalent to solving the original instance.

**Corollary 4.1.** Let \((P, s, w)\) be an instance of the syndrome decoding problem with standard form \((\bar{P}, \bar{s}, w)\). Then

\[
e \text{ is a solution of } (P, s, w) \iff \pi^{-1}e \text{ is a solution of } (\bar{P}, \bar{s}, w).
\]

This is the first common idea of ISD algorithms. The second common point is the special weight distribution on the error vector enforced by the permutation \(\pi\).

**Definition 4.5 (Good Permutation I).** Let \(e \in \mathbb{F}_2^n\) with \(\Delta(e) = w\) and \(k, p \in \mathbb{N}\). We call a permutation \(\pi \) good for \(e\) with respect to Alg. 4.2 and parameters \((k, p)\), if

\[
\pi^{-1}e = (e', e'') \in \mathbb{F}_2^k \times \mathbb{F}_2^{n-k}
\]

with

\[
\Delta(e') = p, \quad \Delta(e'') = w - p.
\]

A random permutation \(\pi\) is good with probability

\[
P_\pi = \frac{{k\choose p}{n-k\choose w-p}}{{n\choose w}}.
\]

Hence, given a good permutation \(\pi\), it is sufficient to find some \(e' \in \mathbb{F}_2^k, \Delta(e') = p\) such that

\[
\Delta(\bar{P}e', \bar{s}) = \Delta(e'') = w - p. \tag{4.3}
\]
In a nutshell one can divide ISD algorithms into two parts. First there is an overall framework (see Alg. 4.1), which converts the decoding instance into its standard form and enforces a special weight distribution on the error vector $e$. Second there is a solver (see Alg. 4.2) for this standard form which is only successful if $e$ has the correct weight distribution. Prange used a simple brute force algorithm, choosing $p = 0$ while Lee and Brickell [LB88] optimized $p$ depending on the code rate $\frac{k}{n}$ and the relative weight $\frac{w}{n}$. However, Peters [Pet11] showed that the improvement upon Prange is only a polynomial factor. The following theorem states upper bounds for the time complexities of full distance decoding and half distance decoding based on Prange’s work.

Algorithm 4.1: SimpleISD

\begin{algorithm}
\caption{SimpleISD}
\begin{algorithmic}
\REQUIRE $P \in \mathbb{F}_2^{(n-k) \times n}$, $s \in \mathbb{F}_2^{n-k}$, $w \in \mathbb{N}$
\ENSURE $e \in \mathbb{F}_2^n$
\REPEAT
\STATE \STATE $\pi \leftarrow$ random permutation on $\mathbb{F}_2^n$
\STATE $(\cdot \mid Q) \leftarrow P\pi$ (permute columns) \Comment{$Q \in \mathbb{F}_2^{(n-k) \times (n-k)}$}
\UNTIL $Q$ is invertible
\STATE $(\bar{P} \mid I_{n-k}) \leftarrow G\pi$ and $\bar{s} \leftarrow Gs$ \Comment{$G \in \mathbb{F}_2^{(n-k) \times (n-k)}$}
\STATE $(e', e'') = \text{ISDSolve}(\bar{P}, \bar{s}, w)$ \Comment{See e.g. Algorithm 4.2.}
\UNTIL $(e', e'') \neq \bot$
\RETURN $\pi(e', e'')$
\end{algorithmic}
\end{algorithm}

Algorithm 4.2: SimpleISDSolve

\begin{algorithm}
\caption{SimpleISDSolve}
\begin{algorithmic}
\REQUIRE $P \in \mathbb{F}_2^{(n-k) \times k}$, $s \in \mathbb{F}_2^{n-k}$, $w \in \mathbb{N}$
\ENSURE $(e', e'') \in \mathbb{F}_2^k \times \mathbb{F}_2^{n-k}$
\PARAMETERS $p \in \mathbb{N}$
\FOR{$e' \in \mathbb{F}_2^k$ with $\Delta(e') = p$}
\STATE $e'' \leftarrow \bar{P}e' + \bar{s}$
\IF{$\Delta(e'') = w - p$} \textbf{return}$(e', e'')$
\ENDIF
\ENDFOR
\RETURN $\bot$
\end{algorithmic}
\end{algorithm}

Theorem 4.1. The syndrome decoding problem can in expectation be solved in time $2^{0.1207n}$ for full distance decoding and $2^{0.05752n}$ for half distance decoding. The memory consumption is polynomial in $n$.

\textbf{Proof.} Alg. 4.1 runs $\binom{k}{p}$ iterations and is successful if and only if $\pi$ is a good permutation.
which happens with probability $P_\pi$. For $p = 0$ we can expect to find a solution after

$$P_\pi^{-1} \cdot \binom{k}{0} = \binom{n}{w} \cdot \frac{(n-k)}{w} \leq 2^{(H(\frac{w}{n}) - H(\frac{w-k}{n}) + \varepsilon)n} \text{ iterations.}$$

Each iteration as well as step 2 take polynomial time and therefore contribute to the error term $\varepsilon$ only. Furthermore the matrix $Q$ in Alg. 4.1 is invertible with constant probability of at least $\frac{1}{4}$. Thus the finding of $Q$ in step 1 also contributes to the error term $\varepsilon$ only and the running time is determined by the number of iterations. For full distance decoding the worst-case running time is achieved at a rate of

$$\frac{k}{n} = 0.455 \text{ with relative distance } \frac{w}{n} = \frac{d}{n} = H^{-1} \left( 1 - \frac{k}{n} \right) = 0.1255.$$  

resulting in the claimed running time. For half distance decoding the running time can be shown analogously for a worst-case code rate

$$\frac{k}{n} = 0.468 \text{ with relative distance } \frac{w}{n} = \frac{d}{2n} = H^{-1} \left( 1 - \frac{k}{n} \right) = 0.06046.$$  

Notice that we can simply use $\frac{w}{n-k} = \frac{n}{1-rac{k}{n}}$ and $\left( \frac{n-k}{n} \right) = 1 - \frac{k}{n}$ in order to insert the code and error rates. Alg. 4.1 and Alg. 4.2 clearly have polynomial memory consumption. The correctness follows from Corollary 4.1. \hfill \Box

### 4.3 Ball Collision Decoding

In this section we describe a solving algorithm which leads to exponential improvement upon Prange’s algorithm and replaces Alg. 4.2 in the SIMPLE-SD framework. In a nutshell, one computes the candidates for $e'$ using a meet-in-the-middle technique (see Fig. 4.2) which is a typical time-memory trade-off. This approach was considered first by Stern [Ste88] in 1989.

![Fig. 4.2: Dimension reduction and weight distribution for Stern’s algorithm.](image-url)
It was refined by Bernstein, Lange and Peter’s ball collision decoding algorithm [BLP11] (see Fig. 4.3) which is a generalization of Stern’s variant.

\[ \text{Ball Collision Decoding.} \]

We now give a more detailed description of the ball collision decoding algorithm (see Alg. 4.3) which is used as a subroutine in Alg. 4.1 and thus replaces Alg. 4.2. Recall that we want to find some \( e' \in \mathbb{F}_2^n, \Delta(e') = p \) such that Eq. 4.3 holds. The algorithm introduces a split to \( e' \) into two parts of length \( \frac{k}{2} \) and enforces weight \( \frac{p}{2} \) on both vectors. Furthermore the second part \( e'' \) of the error is split into three parts where the first two parts have length \( \frac{\ell}{2} \) and the last one has length \( n-k-\ell \). The first two parts have enforced weight \( \frac{q}{2} \) while the last coordinates have weight \( w - p - q \). The parameters \( \ell, p, q \) are subject to optimization.

The algorithm only outputs a solution \((e', e'') \neq \perp\) if the permutation \( \pi \) in Alg. 4.1 induces the correct weight distribution. In this case, there exist vectors of the form

\[
\begin{align*}
\mathbf{e}'_1 &\in \mathbb{F}_2^{k/2} \times 0^{k/2}, \quad \mathbf{e}'_2 \in 0^{k/2} \times \mathbb{F}_2^{k/2} \quad \text{with} \quad \Delta(\mathbf{e}'_1) = \Delta(\mathbf{e}'_2) = \frac{p}{2}, \\
\mathbf{e}''_1 &\in \mathbb{F}_2^{\ell/2} \times 0^{\ell/2} \times 0^{n-k-\ell}, \quad \mathbf{e}''_2 \in 0^{\ell/2} \times \mathbb{F}_2^{\ell/2} \times 0^{n-k-\ell} \quad \text{with} \quad \Delta(\mathbf{e}''_1) = \Delta(\mathbf{e}''_2) = \frac{q}{2},
\end{align*}
\]

such that the vector \((\mathbf{e}', \mathbf{e}'')\) with \( \mathbf{e}' = \mathbf{e}'_1 + \mathbf{e}'_2 \) and \( \mathbf{e}'' = \mathbf{e}''_1 + \mathbf{e}''_2 + (0^{\ell}, \cdot) \) solves the standard form. Thus Eq. 4.2 can be rewritten as

\[
\begin{align*}
(\bar{P} \mathbf{e}'_1 + \mathbf{e}''_1)_{[1,2]} &= (\bar{P} \mathbf{e}'_2 + \mathbf{e}''_2 + \bar{s})_{[1,2]} \\
\Delta((\bar{P} \mathbf{e}'_1)_{[3]},(\bar{P} \mathbf{e}'_2 + \bar{s})_{[3]}) &= w - p - q.
\end{align*}
\]

**Definition 4.6 (Good Permutation II).** Let \( \mathbf{e} \in \mathbb{F}_2^n \) with \( \Delta(\mathbf{e}) = w \) and \( k, \ell, p, q \in \mathbb{N} \). We call a permutation \( \pi \) good for \( \mathbf{e} \) with respect to Alg. 4.3 and parameters \((k, \ell, p, q)\),
A random permutation \( \pi \) with which fixes \( \pi^{-1} e = (e_1', e_2', e_3'', e_4'') \in \mathbb{F}_2^{k/2} \times \mathbb{F}_2^{k/2} \times \mathbb{F}_2^{\ell/2} \times \mathbb{F}_2^{\ell/2} \times \mathbb{F}_2^{n-k-\ell} \) with
\[
\Delta(e_1') = \frac{p}{2}, \quad \Delta(e_2') = \frac{q}{2}, \quad i = 1, 2 \text{ and } \Delta(e_3'') = w - p - q.
\]
A random permutation \( \pi \) is good with probability
\[
P_\pi = \binom{\binom{k/2}{p/2} \binom{\ell/2}{q/2} (n-k-\ell)}{w}.
\]

Notice, that we have for \( e' \) (and analogously for \( e'' \)) that \( e_1' = (e_1')_1, e_2' = (e_2')_1 \) and \( e' = (e_1', e_2') = e_1' + e_2' \). Thus both \( e_1', e_2' \) and \( e_1', e_2' \) represent the vector \( e' \) in different ways. The algorithm now finds solutions to Eq. 4.5 and Eq. 4.6. First, it enumerates all possible vectors satisfying Eq. 4.4 and constructs the lists
\[
L_1 \subset \mathbb{F}_2^{n-k} \times (\mathbb{F}_2^{k/2} \times \mathbb{F}_2^{k/2}) \times (\mathbb{F}_2^{\ell/2} \times \mathbb{F}_2^{\ell/2} \times \mathbb{F}_2^{n-k-\ell})
\]
\[
L_2 \subset \mathbb{F}_2^{n-k} \times (\mathbb{F}_2^{k/2} \times \mathbb{F}_2^{k/2}) \times (\mathbb{F}_2^{\ell/2} \times \mathbb{F}_2^{\ell/2} \times \mathbb{F}_2^{n-k-\ell})
\]
with
\[
L_1 = \{ (\bar{P}e_1' + e_1', e_1', e_1'') \mid \Delta(e_1') = \frac{p}{2}, \Delta(e_1'') = \frac{q}{2} \}
\]
\[
L_2 = \{ (\bar{P}e_2' + e_2', \bar{s}, e_2', e_2'') \mid \Delta(e_2') = \frac{p}{2}, \Delta(e_2'') = \frac{q}{2} \}.
\]
The exact matching of Eq. 4.5 is then solved via Sort-and-Match on the two lists searching for vectors matching on the first \( \ell \) coordinates. The resulting list is then filtered for weight \( w - p - q \) on the next \( n - k - \ell \) coordinates to satisfy the approximate matching of Eq. 4.6. The algorithm obtains a final list \( L \subset \mathbb{F}_2^{n-k-\ell} \times \mathbb{F}_2^{k} \times \mathbb{F}_2^{\ell} \times \mathbb{F}_2^{n-k-\ell} \) with
\[
L = \{ (0, e_3'', e', e_1', 0) \mid \Delta(e') = p, e'' = \bar{P}e' + \bar{s}, \Delta(e''') = w - p \}.
\]
Thus any element in \( L \) yields a solution of the syndrome decoding problem in standard form \((\bar{P}, \bar{s}, w)\). In contrast to the algorithms in Chapter 2 and 3 we are interested in the sum of two vectors after the Sort-and-Match step instead of the vectors. We therefore store the checksums of the tuples in \( L \).

There are two special choices of parameters \( p, \ell, q \). For \( p = \ell = q = 0 \) the two lists \( L_1, L_2 \) only contain the zero vector and the filtering finds \( e'' \in \mathbb{F}_2^{n-k} \) with weight \( w \). Thus this corresponds to Prange’s algorithm. For \( q = 0 \) we obtain Stern’s algorithm which fixes \( \ell \) zero bits in the target vector \( e'' \) instead of enforcing some weight \( q \). This additional parameter leads to an exponential improvement by [BLP11] upon [Ste88].
Algorithm 4.3: BCD

Input: $\bar{P} \in \mathbb{F}_2^{(n-k)\times k}, \bar{s} \in \mathbb{F}_2^{n-k}, w \in \mathbb{N}$

Output: $(e', e'') \in \mathbb{F}_k^{2} \times \mathbb{F}_n^{n-k}$

Parameters: $\ell := (\ell, n-k-\ell, k, \ell, n-k-\ell) \in \mathbb{N}^5$, $p \in [w], q \in [w-p]$

1. Create lists $L_1, L_2$ as defined in (4.7)
2. $L \leftarrow \text{Sort-and-Match}(L_1, L_2, 1, 0)$
3. $L \leftarrow \text{Filter}(L, 2, w-p-q)$
   If $|L| > 0$ then return $(e', e'')$ for some $(0, e''_{[3]}, e', e''_{[1,2]}, 0) \in L$
   Else return ⊥

Proposition 4.1. The syndrome decoding problem can in expectation be solved in time $2^{(1.1630n)}$ and memory $2^{(0.0350n)}$ for full distance decoding as well as $2^{(1.0558n)}$ and memory $2^{(0.0139n)}$ for half distance decoding.

Proof. BCD enumerates vectors of the form given by Eq. 4.4 and keeps those which satisfy Eq. 4.5 and Eq. 4.6 in the list $L$. Given a good permutation $\pi$ (Definition 4.6) this list contains a vector corresponding to a solution of the standard form $(\bar{P}, \bar{s}, w)$ since the algorithm covers the whole search space. The construction of the lists $L_1, L_2$ in step 1 can be done in time

$$T_0 = S_0 = 2^{2\lambda n} := |L_1| = |L_2| = \binom{k/2}{\ell/2} \binom{\ell/2}{p/2} \binom{\ell/2}{q/2}.$$ 

The Sort-and-Match in step 2 results in a list of size $S_1 := 2^{(2\lambda - \frac{\lambda}{2})n}$. Together with the filtering in step 3 this takes time

$$T_1 := 2^{(\max\{\lambda, 2\lambda - \frac{\lambda}{2}\} + \epsilon)n}.$$ 

by Lemma 1.2 and Lemma 1.3. According to Definition 4.6, one can expect to find a good permutation after $P_{\pi}^{-1}$ iterations resulting in an overall expected running time

$$T = P_{\pi}^{-1} \cdot \max\{T_0, T_1\} = P_{\pi}^{-1} \cdot T_1$$

The expected memory consumption is $M = \max\{S_0, S_1\}$ since we do not need to store the list $L$ after step 2.

We now use Lemma 1.1 to upper bound $\lambda$ and $P_{\pi}^{-1}$ as well as numerical optimization to obtain parameters yielding the claimed complexities. For full distance decoding the
worst-case running time is achieved at a rate of

\[ \frac{k}{n} = 0.446 \text{ with relative distance } \frac{w}{n} = \frac{d}{n} = H^{-1} \left( 1 - \frac{k}{n} \right) = 0.1288 \]

The running times is then minimized choosing relative parameters

\[ \frac{p}{n} = 0.0094, \quad \frac{\ell}{n} = 0.0355, \quad \frac{q}{n} = 0.00075. \]

The resulting list sizes are \( S_0 = S_1 = 2^{0.03550n} \) while the probability for the correct weight distribution is \( P_\pi = 2^{-0.08079n} \). This yields the claimed running time and memory consumption. For half distance decoding the running time and memory consumption can be shown analogously for a worst-case code rate

\[ \frac{k}{n} = 0.465 \text{ with relative distance } \frac{w}{n} = \frac{d}{2n} = H^{-1} \left( 1 - \frac{k}{n} \right) = 0.06099 \]

and relative parameters

\[ \frac{p}{n} = 0.0031, \quad \frac{\ell}{n} = 0.0139, \quad \frac{q}{n} = 0.00011. \]

\[ \square \]

Analogously to the algorithms in Section 3.2, we do not know explicit formulas for optimal parameters \( p, \ell, q \). Hence, in contrast to Theorem 4.1, we cannot give a closed formula for the running time exponent. This is also the case for subsequent results in this chapter. Instead, we run numerical optimizations which returns parameters leading to a minimal running time for different code rates.

**Comparison to Stern’s Algorithm.** For Stern’s algorithm where \( q = 0 \), full distance decoding can be solved in worst-case time \( 2^{0.1166n} \) and memory \( 2^{0.03119n} \) choosing parameters \( \frac{p}{n} = 0.0088, \frac{\ell}{n} = 0.0312 \) at rate \( \frac{k}{n} = 0.445 \). Half distance decoding can be solved in worst-case time \( 2^{0.05564n} \) and memory \( 2^{0.01308n} \) choosing parameters \( \frac{p}{n} = 0.0030, \frac{\ell}{n} = 0.0131 \) at rate \( \frac{k}{n} = 0.467 \). Thus BCD is faster than Stern’s algorithm in the worst case. Bernstein, Lange and Peters provided a proof for the general superiority of the BCD-algorithm over Stern’s algorithm in [BLP11]. Therefore, for all choices of \( 0 < k < n \) and \( 0 < w < n \), Alg. 4.3 provides a lower running time exponent if one chooses \( q > 0 \) upon the choice \( q = 0 \).
4.4 Advanced Information Set Decoding

In this section we introduce another framework for information set decoding. This is also the foundation for more advanced techniques, which reduce the running time exponent significantly. In a nutshell, the new framework offers a variant of the standard form from Definition 4.4 (see Fig. 4.4). Furthermore a new way of splitting the error vector \( e' = e'_1 + e'_2 \) is introduced where \( e'_1 \) and \( e'_2 \) may overlap. This framework was considered first by Finiasz and Sendrier [FS09].

![Dimension reduction and weight distribution for advanced ISD.](image)

**Definition 4.7 (Advanced Standard Form).** Let \( (P, s, w) \in \mathbb{F}_2^{(n-k)\times n} \times \mathbb{F}_2^{n-k} \times [n] \) be an instance of the syndrome decoding problem. We say that \( (\tilde{P}, \tilde{s}, w) \in \mathbb{F}_2^{(n-k)\times(k+\ell)} \times \mathbb{F}_2^{n-k} \times [n] \) is an advanced standard form of \( (P, s, w) \) if there exists some invertible \( G' \in \mathbb{F}_2^{(n-k)\times(n-k)} \) and \( \pi \in \mathbb{F}_2^{n\times n} \) such that

\[
G'P\pi = \begin{pmatrix} \tilde{P} & 0 \\ \bar{I}_{n-k-\ell} & \end{pmatrix}
\]

and \( G's = \tilde{s} \).

The vector \( (e', e'') \in \mathbb{F}_2^{k+\ell} \times \mathbb{F}_2^{n-k-\ell} \) solves the advanced standard form \( (\tilde{P}, \tilde{s}, w) \) if and only if \( \tilde{s} = \tilde{P}e' + (0, e'') \) and \( \Delta(e', e'') = w \).

Analogously to the standard form, it follows directly from Lemma 4.1 that the advanced standard form is equivalent to the original instance.

**Corollary 4.2.** Let \( (P, s, w) \) be an instance of the syndrome decoding problem with advanced standard form \( (\tilde{P}, \tilde{s}, w) \). Then

\[
e \text{ is a solution of } (P, s, w) \iff \pi^{-1}e \text{ is a solution of } (\tilde{P}, \tilde{s}, w).
\]

Thus the first part of every algorithm in the advanced ISD framework is to convert an instance of the syndrome decoding problem into its advanced standard form (see Alg. 4.4).
Algorithm 4.4: ADVANCEDISD - FRAMEWORK

<table>
<thead>
<tr>
<th>Input</th>
<th>$P \in \mathbb{F}<em>2^{(n-k)\times n}$, $s \in \mathbb{F}</em>{2^{n-k}}$, $w \in \mathbb{N}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>$e \in \mathbb{F}_2^n$</td>
</tr>
</tbody>
</table>

```
repeat
1 repeat
   \pi \leftarrow \text{random permutation on } \mathbb{F}_2^n
   \begin{pmatrix} 0 & \cdot & Q' \end{pmatrix} \leftarrow P \pi \quad \triangleright Q' \in \mathbb{F}_2^{(n-k-\ell)\times(n-k-\ell)}
   \text{until } Q' \text{ is invertible}
2 \begin{pmatrix} \tilde{P} & 0 & I_{n-k-\ell} \end{pmatrix} \leftarrow G' \pi \pi \quad \triangleright G' \in \mathbb{F}^{(n-k)\times(n-k)}
   \tilde{s} \leftarrow G' s
3 \begin{pmatrix} e' \cdot e'' \end{pmatrix} = \text{AdvISDSolve} (\tilde{P}, \tilde{s}, w) \quad \triangleright \text{See Algorithm 4.2.}
until (e', e'') \neq \bot
return \pi(e', e'')
```

We now describe a simple solving algorithm (see Alg. 4.5) in the ADVANCEDISD-framework. It was presented first by Finiasz and Sendrier [FS09]. The algorithm enforces weight $\Delta(e') = p$ and $\Delta(e'') = w - p$ again.

**Definition 4.8 (Good Permutation III).** Let $e \in \mathbb{F}_2^n$ with $\Delta(e) = w$ and $k, p, \ell \in \mathbb{N}$. We call a permutation $\pi$ good for $e$ with respect to Alg. 4.5 and parameters $(k, p, \ell)$, if $\pi^{-1} e = (e', e'') \in \mathbb{F}_2^{k+\ell} \times \mathbb{F}_2^{n-k-\ell}$ with

$$\Delta(e') = p, \quad \Delta(e'') = w - p.$$ 

A random permutation $\pi$ is good with probability

$$P_\pi = \binom{k+\ell}{p} \binom{n-k-\ell}{w-p}.\binom{n}{w}.$$

The algorithm furthermore introduces a split into two blocks of length $\ell$ and $n-k-\ell$ to the sum $\tilde{P}e' + \tilde{s}$ and constructs the error vector $e'$ as a sum $e' = e'_1 + e'_2$ of two vectors $e'_1, e'_2 \in \mathbb{F}_2^{k+\ell}$ with full length and weight $\frac{p}{2}$. Hence, given a good permutation $\pi$, it is sufficient to find some $e'_i \in \mathbb{F}_2^{k+\ell}$, $\Delta(e'_i) = \frac{p}{2}$, $i = 1, 2$ such that

$$\begin{align*}
(\tilde{P}e'_1)_{[1]} &= (\tilde{P}e'_2 + \tilde{s})_{[1]} \quad \text{and} \\
\Delta((\tilde{P}e'_1)_{[2]}), (\tilde{P}e'_2 + \tilde{s})_{[2]}) &= w - p.
\end{align*}$$

Therefore we have an exact matching on $\ell$ coordinates by Eq. 4.8 and approximate matching on the remaining $n - k - \ell$ coordinates by Eq. 4.9. Note that, in contrast to
Simple ISD, the two vectors overlap and thus do not necessarily add up to weight $p$. The algorithm samples vectors of length $k + \ell$ and weight $\frac{p}{2}$ to construct lists

$$L_1 = \{ (\tilde{P}e'_1, e'_1) \in \mathbb{F}_2^{n-k} \times \mathbb{F}_2^{k+\ell} | \Delta(e'_1) = \frac{p}{2} \}$$

$$L_2 = \{ (\tilde{P}e'_2 + \tilde{s}, e'_2) \in \mathbb{F}_2^{n-k} \times \mathbb{F}_2^{k+\ell} | \Delta(e'_2) = \frac{p}{2} \}. \quad (4.10)$$

The exact matching of Eq. 4.8 is then solved via Sort-and-Match on the two lists searching for vectors matching on the first $\ell$ coordinates. The resulting list is then filtered for weight $w - p$ on the next $n - k - \ell$ coordinates to satisfy the approximate matching of Eq. 4.9. The algorithm obtains a final list

$$L = \{ (0, e''', e') \in 0^{\ell} \times \mathbb{F}_2^{n-k-\ell} \times \mathbb{F}_2^{k+\ell} | \Delta(e') = p, (0, e'') = \tilde{P}e' + \tilde{s}, \Delta(e'') = w - p \}.$$

Thus any element in $L$ yields a solution of the syndrome decoding problem in advanced standard form $(\tilde{P}, \tilde{s}, w)$. Note that every element in the final list also contains a zero $\ell$ bit string which is the result of finding a solution to the exact matching but which is not a part of the target vector $e$ itself.

**Algorithm 4.5: AdvISDSolve**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tilde{P} \in \mathbb{F}_2^{(n-k)\times(k+\ell)}, \tilde{s} \in \mathbb{F}_2^{n-k}, w \in \mathbb{N}$</td>
<td>$(e', e'') \in \mathbb{F}_2^{k+\ell} \times \mathbb{F}_2^{n-k-\ell}$</td>
<td>$\ell = (\ell, n - k - \ell, k + \ell) \in \mathbb{N}^3, p \in [w]$</td>
</tr>
<tr>
<td>Create lists $L_1, L_2$ as defined in (4.10)</td>
<td>$L \leftarrow \text{Sort-and-Match}(L_1, L_2, 1, 0)^\dagger$</td>
<td>1</td>
</tr>
<tr>
<td>2 $L \leftarrow \text{Sort-and-Match}(L_1, L_2, 1, 0)^\dagger$</td>
<td>2 $L \leftarrow \text{Filter}(L, 2, w - p)$</td>
<td>3 if $</td>
</tr>
</tbody>
</table>

It is not hard to see how the BCD-algorithm transforms into this algorithm through removing the parameter $q$ and choosing a larger parameter $p$. In fact it was shown in [MMT11] that Alg. 4.5 achieves at least the same complexity as Alg. 4.3. However, the description of the algorithm becomes much simpler in the new framework. Another difference is, that the algorithm by Finiasz and Sendrier samples sufficiently many vectors until the whole search space for $e'$ is covered instead of enumerating all possible vectors. Furthermore their algorithm allows different vectors to add up to the same $e'$. In the next section we show how to exploit this property.
4.5 Representation Techniques

In the BCD-algorithm, the vector $e' \in \mathbb{F}_2^k$, $\Delta(e') = p$ is constructed via enumerating all vectors $e'_1 \in \mathbb{F}_2^{k/2} \times \mathbb{F}_2^{k/2}$, $e'_2 \in \mathbb{F}_2^{k/2} \times \mathbb{F}_2^{k/2}$ with $\Delta(e'_1) = \Delta(e'_2) = \frac{p}{2}$ and searching for a pair $(e'_1, e'_2)$ such that $e' = e'_1 + e'_2$. Given a good permutation $\pi$, there exists exactly one matching pair. In contrast to this the algorithm by Finiasz and Sendrier allows the $\frac{p}{2}$ 1’s to be distributed over the whole length $k + \ell$ of the two vectors. As we will see, this leads to an exponential number of matching pairs which we call representations of the vector $e'$ (see the top two examples in Fig. 4.5). Representations where originally introduced for solving algorithms of the subset sum problem by Howgrave-Graham and Joux [HGJ10] and improved by Becker, Coron and Joux [BCJ11].

As mentioned before the algorithm of Finiasz and Sendrier does not fully exploit their existence. This was first achieved by May, Meurer and Thomae in [MMT11] gaining an exponential improvement upon previous works. Finally, the algorithm by Becker, Joux, May and Meurer [BJMM12] fully exploited these techniques allowing higher weight $> \frac{p}{2}$ for the vectors $e'_1, e'_2$ (see bottom two examples in Fig. 4.5). This leads to an even higher amount of representations gained by additional 1’s which cancel out if they are at the same position.

$$
\begin{align*}
0100100000011 & \quad e'_2 \quad 1010000100010 \\
1010000110000 & \quad e'_1 \quad 0100101000001 \\
\hline
1110100110011 & \quad e' \quad 1110100110011
\end{align*}
$$

$$
\begin{align*}
0101100010010 & \quad e'_2 \quad 1110010001001 \\
1011000100110 & \quad e'_1 \quad 0001101101010 \\
\hline
1110100110011 & \quad e' \quad 1110100110011
\end{align*}
$$

Fig. 4.5: Possible representations of a vector $e'$. The gray parts mark the ones which cancel out each other.

**Definition 4.9 (Representation).** Let $e' \in \mathbb{F}_2^k$ with $\Delta(e') = p$. A pair of vectors $(e'_1, e'_2) \in \mathbb{F}_2^k \times \mathbb{F}_2^k$ with $\Delta(e'_1) = \Delta(e'_2) \geq \frac{p}{2}$ and $e' = e'_1 + e'_2$ is called a representation of $e'$. 74
The following lemma estimates the number of representations of a vector.

**Lemma 4.2.** Let \( e' \in F_2^k \) with \( \Delta(e') = p, p_1 \geq \frac{k}{2} \) and

\[
\mathcal{R} := \{(e_1', e_2') \in F_2^k \times F_2^k \mid e_1' + e_2' = e' \land \Delta(e_1') = \Delta(e_2') = p_1\}
\]

be the set containing all representations of \( e' \) with vectors of weight \( p_1 \). There exist a total number of

\[
|\mathcal{R}| = \left( \begin{array}{c} p \\ \frac{p}{2} \end{array} \right) \left( \begin{array}{c} k - p \\ p_1 - \frac{p}{2} \end{array} \right)
\]

representations.

**Proof.** Because the \( k - p \) 0’s in \( e' \) can only be represented as \( 0 + 0 \) or \( 1 + 1 \), an equal number of \( x \) 1’s in \( e_1' \) and \( e_2' \) must be in the same positions. Fixing these \( x \) 1’s (and \( k - p - x \) 0’s) in \( e_1' \) fully determines the entries in \( e_2 \) on those \( k - p \) coordinates. Hence, there exist \( \binom{k-p}{x} \) possible combinations for this part. The \( p \) 1’s in \( e' \) can only be represented as \( 0 + 1 \) or \( 1 + 0 \). Furthermore the number of 1’s must be equal on this part, too since both vectors \( e_1' \) and \( e_2' \) have the same weight \( p_1 \) and same the number of 1’s on the other part. Thus they must exactly have weight \( \frac{p}{2} \) on those coordinates. This leads to another \( \binom{p}{\frac{p}{2}} \) combinations for this part and with \( \frac{p}{2} + x = p_1 \Rightarrow x = p_1 - \frac{p}{2} \) this yields the claimed number of representations. \( \square \)

**The BJMM Algorithm.** We now give a detailed description of the BJMM-algorithm which replaces Alg. 4.5 in the AdvancedISD-framework (Alg. 4.4). The reader is advised to follow via Fig. 4.6. We skip the MMT-algorithm at this point since it is a non-optimal special case of the BJMM. In a nutshell the algorithm consists of several **Sort-And-Match** steps forming a binary search tree of depth \( m \) similar to the \( k \)-tree-algorithm (Alg. 2.4) in Chapter 2. The parameter \( m \) is subject to optimization. Starting with a meet-in-the-middle like merging similar to Stern’s algorithm in Section 4.3, this results in a final list containing elements of the form \( (\tilde{P} e' + \tilde{s}, e') \) solving Eq. 4.8. In a last step, Eq. 4.9 is checked naively.

**Construction of the Solution.** Let us first describe how the solution of the syndrome decoding problem is constructed. This essentially describes the algorithm in reverse order. Analogously to the algorithm by Finiasz and Sendrier, the BJMM constructs the first part of the error vector \( e' = e_1^{(m-1)} + e_2^{(m-1)} \in F_2^{k+\ell} \) with \( \Delta(e') = p_m \) for some weight \( p_m \) and \( e_1^{(m-1)}, e_2^{(m-1)} \in F_2^{k+\ell} \). Additionally the corresponding vectors \( \tilde{P} e_1^{(m-1)} \) and \( \tilde{P} e_2^{(m-1)} \) satisfy Eq. 4.8 and Eq. 4.9 yielding a solution

\[
(e', e'') \text{ with } \Delta(e'') = w - p_m \text{ and } (0, e'') = \tilde{P} e' + \tilde{s} = \tilde{P} e_1^{(m-1)} + \tilde{P} e_2^{(m-1)} + \tilde{s}.
\]
The algorithm now introduces a split into $m$ blocks to the all zero part, where block $i$ contains $\ell_i$ coordinates with $\ell := \sum_{i=1}^{m} \ell_i$. The final list of all solutions is constructed out of two lists $L_{1}^{(m-1)}, L_{2}^{(m-1)}$ on level $m-1$ containing candidates of the form $(\tilde{P}e_{1}^{(m-1)}, e_{1}^{(m-1)})$, $(\tilde{P}e_{2}^{(m-1)} + \tilde{s}, e_{2}^{(m-1)})$ respectively, where

$$\Delta(e_{1}^{(m-1)}) = \Delta(e_{2}^{(m-1)}) = p_{m-1} \geq \frac{p_m}{2}.$$ 

By Lemma 4.2 there exist

$$R_m := \left( \frac{p_m}{p_m/2} \right) \left( \frac{k + \ell - p_m}{p_{m-1} - p_m/2} \right)$$

representations $e' = e_{1}^{(m-1)} + e_{2}^{(m-1)}$. Since only one representation is needed in expectation, it is sufficient for the algorithm to construct an $\frac{1}{R_m}$-fraction of all representations. This is done by keeping specific vectors in $L_{1}^{(m-1)}$ and $L_{2}^{(m-1)}$. In detail, only vectors are kept where the first $m-1$ blocks of $\tilde{P}e_{1}^{(m-1)}$, $\tilde{P}e_{2}^{(m-1)} + \tilde{s}$ are zero. Hence, all
elements in $L_1^{(m-1)} \times L_2^{(m-1)}$ contain zeros on the first $m - 1$ blocks by construction, i.e. a fraction is kept. Choosing $\sum_{i=1}^{m-1} \ell_i := \lceil \log R_m \rceil$ one solution survives in expectation. A simple Sort-And-Match on $L_1^{(m-1)}$ and $L_2^{(m-1)}$ for matching elements on the $\ell_m$ bits of block $m$ as well as subsequent filtering for weight $w - p_m$ on the next $n - k - \ell$ coordinates and for weight $p_m$ on the last $k + \ell$ coordinates returns the final list $L_1^{(m)}$.

On the above levels 2, …, $m$ these steps are repeated recursively. The first list $L_1^{(i)}$ on every level $i = 2, \ldots, m$ is constructed via Sort-And-Match on lists $L_1^{(i-1)}$, $L_2^{(i-1)}$ for zero entries only in block $i$ and subsequent filtering for weight $p_i$ on the last $k + \ell$ bits. As the latter lists already contain elements with zeros on the first $i - 1$ blocks, elements in $L_1^{(i)}$ are zero on all coordinates of the first $i$ blocks. Furthermore the number of eliminated elements determined by the $\ell_i$ as well as the representations of the vectors determined by the $p_i$ are balanced again. The remaining lists on each level are created analogously.

The list $L_1^{(1)}$ is constructed out of lists $L_1^{(0)}$, $L_2^{(0)}$ using the meet-in-the-middle technique by Stern. The algorithm represents $\tilde{e}_1^{(1)} \in \mathbb{F}_2^{k+\ell}$ with $\Delta(\tilde{e}_1^{(1)}) = p_1$ as $\tilde{e}_1^{(1)} = e_1^{(0)} + e_2^{(0)}$ where

$$\Delta(e_1^{(0)}) = s_0 + e_1^{(0)} - \frac{p_1}{2} \text{ and } e_1^{(0)} \in \mathbb{F}_2^{k+\ell} \times \mathbb{F}_2\times \frac{1}{2} \Delta(e_1^{(0)})$$

Elements $(\hat{e}_1^{(0)}, e_1^{(0)}) \in L_1^{(0)}$ and $(\hat{e}_2^{(0)}, e_2^{(0)}) \in L_1^{(0)}$ now add up to weight $p_1$ on the last $k + \ell$ bits by construction while a Sort-And-Match on the first $\ell_1$ bits completes the construction of $L_1^{(1)}$.

**Description of the BJMM-Algorithm with Arbitrary Depth.** We now describe the full algorithm (see Alg. 4.6) in the correct order. On level 0, the algorithm starts with lists

$$(4.11) \quad L_j^{(0)} = \{(\hat{e}_j^{(0)}, e_j^{(0)}) \in \mathbb{F}_2^{n-k} \times \mathbb{F}_2\times (k+\ell)/2 \mid \Delta(e_j^{(0)}) = p_1/2\}$$

$$(4.12) \quad L_j^{(0)} = \{(\hat{e}_j^{(0)}, e_j^{(0)}) \in \mathbb{F}_2^{n-k} \times \mathbb{F}_2\times (k+\ell)/2 \mid \Delta(e_j^{(0)}) = p_1/2\}$$

$$(4.13) \quad L_j^{(0)} = \{(\hat{e}_j^{(0)}, e_j^{(0)}) \in \mathbb{F}_2^{n-k} \times \mathbb{F}_2\times (k+\ell)/2 \mid \Delta(e_j^{(0)}) = p_1/2\}$$

for $j_1 = 1, 3, \ldots, 2^{m-1}-1$ and $j_2 = 2, 4, \ldots, 2^m-2$. These lists are pairwise combined using Sort-And-Match on the first block of length $\ell_1$. This step in combination with filtering for the correct weight $p_i$ on the last $k + \ell$ coordinates is repeated on every level $i$ in a tree wise fashion until a single list is left. In detail the algorithm combines the two lists $L_1^{(i-1)}$, $L_2^{(i-1)}$ via Sort-And-Match on block $i$ of size $\ell_i$ plus filtering for
weight $p_i$ on the last block to obtain the first list

$$L_1^{(i)} = \{(\tilde{P}e_1^{(i)}, e_1^{(i)}) \in \mathbb{F}_2^{n-k} \times \mathbb{F}_2^{k+\ell} | \Delta(e_1^{(i)}) = p_i, \ (\tilde{P}e_1^{(i)})_{[1, \ldots, i]} = 0\}$$

on each level $i = 1, \ldots, m$. The other lists $L_j^{(i)}$ are created analogously. Note again that on level $i$ the blocks $1, \ldots, i - 1$ add to the zero vector by construction which is why canceling out the $i$-th block is sufficient. Eventually, filtering on the last list $L_1^{(m)}$ for weight $w - p_m$ on $n - k - \ell$ bits leads to the final list

$$L_1^{(m)} = \{(0, e'', e') \in 0^\ell \times \mathbb{F}_2^{n-k} \times \mathbb{F}_2^{k+\ell} | \Delta(e') = p_m, (0, e'') = \tilde{P}e' + \tilde{s}, \Delta(e'') = w - p_m\}.$$

Thus any element in this list yields a solution of the given syndrome decoding problem in advanced standard form $(\tilde{P}, \tilde{s}, w)$. In contrast to the algorithms in Chapter 2 and 3 we are interested in the sum of two vectors after each Sort-and-Match step instead of the vectors. We therefore store the checksums of the tuples each time. We now show that the BJMM-algorithm exponentially improves the complexity upon BCD.

**Algorithm 4.6: BJMM($m$)**

\begin{algorithm}
\begin{enumerate}
\item Create lists $L_j^{(1)}, j = 1, \ldots, 2^m$ as defined in (4.11).
\item for $i = 1, \ldots, m$ do
  \begin{enumerate}
  \item for $j = 1, \ldots, 2^{m-i}$ do
    \begin{enumerate}
    \item $L_j^{(i)} \leftarrow$ Sort-and-Match($L_{2j-1}^{(i-1)}, L_{2j-1}^{(i-1)}, \epsilon, 0^\ell$)
    \item $L_j^{(i)} \leftarrow$ Filter($L_j^{(i)}, m + 2, p_i$)
    \end{enumerate}
  \end{enumerate}
\item $L_1^{(m)} \leftarrow$ Filter($L_1^{(m)}, m + 1, w - p_m$)
\item if $|L_1^{(m)}| > 0$ then return $(e', e'')$ for some $(0, e'', e') \in L_1^{(m)}$
\item else return $\perp$
\end{enumerate}
\end{algorithm}

**Proposition 4.2.** The syndrome decoding problem can in expectation be solved in time $2^{0.1020n}$ and memory $2^{0.07280n}$ for full distance decoding and in time $2^{0.04949n}$ and memory $2^{0.02617n}$ for half distance decoding.

The following Lemma proofs that Alg. 4.6 constructs a non-empty list of solutions if the induced permutation $\pi$ in Alg. 4.4 is good, i.e. $\pi$ satisfies Def. 4.8.
Lemma 4.3 (Correctness of BJMM). Let \( e \) be a solution to the instance \((P, s, w) \in \mathbb{F}_2^{(n-k) \times n} \times \mathbb{F}_2^{m-k} \times [n]\) of the syndrome decoding problem with advanced standard form \((\tilde{P}, s, w)\). Let \( \pi \) be a good permutation for \( e \) as given by Def. 4.8 with parameters \((k, p_m, \ell)\). If Alg. 4.6 runs with parameters \( \ell_i, p_i \in \mathbb{N} \) for \( i = 1, \ldots, m \) satisfying

\[
\sum_{i=1}^{m} \ell_i = \ell \quad \text{and} \quad \left( p_i \right) \left( k + \ell - p_i \right) \left( p_{i-1} - p_i/2 \right) \geq \prod_{h=1}^{i-1} 2^{\ell_h} \forall i = 2, \ldots, m \tag{4.12}
\]

then \((0, e'', e') \in L^{(m)}_1\) for \( \pi^{-1}e = (e', e'') \).

Proof. Let \((e', e'')\) be the solution of the advanced standard form we are looking for. Since we assume that \( \pi \) is good, we have \( \Delta(e'') = w - p_m \). Furthermore, \( e' \) fully determines \( e'' \) by \( \tilde{P} e' + \tilde{s} = (0, e'') \) and the algorithm exactly finds such candidates in the first component if \( \sum_{i=1}^{m} \ell_i = \ell \). It is therefore sufficient to show that the desired \( e' \) is also constructed by the algorithm.

On the first level, all vectors \( e_1^{(0)}, e_2^{(0)} \) are combined to obtain \( e_1^{(1)} = e_1^{(0)} + e_2^{(0)} \). Since all sums are different by the definition of \( e_1^{(0)}, e_2^{(0)} \), there are \( \binom{k+l}{p_i/2} \) different vectors \( e_1^{(1)} \). Up to polynomial factors, these are all \( \binom{k+s}{p_i} \) possible vectors \( e_1^{(1)} \) (and analogously \( e_j^{(1)}, j = 2, \ldots, 2^{m-1} \)) by Lemma 1.1.

Let us now look at the construction of \( e_1^{(1)} \) with weight \( p_i \) on level \( i \) as the sum of two vectors \( e_1^{(i-1)}, e_2^{(i-1)} \) with weight \( p_{i-1} \geq p_i \). By Lemma 4.2, the vector has

\[
R_i := \left( p_i \right) \left( k + \ell - p_i \right) \left( p_{i-1} - p_i/2 \right) \tag{4.13}
\]

representations, while it is sufficient to have one representation, i.e. a \( \frac{1}{R_i} \)-fraction in \( L^{(i-1)}_1 \times L^{(i-1)}_2 \). Recall that \( (\tilde{P} e_1^{(1)}) [h] = 0 \) for \( h = 1, \ldots, i \) by definition of \( e_1^{(i)} \). Thus all representations of this vector satisfy \( (\tilde{P} e_1^{(i-1)}) [h] = (\tilde{P} e_2^{(i-1)}) [h] \) for \( h = 1, \ldots, i \). Furthermore, following from the construction on prior levels, the lists on level \( i - 1 \) only contain vectors satisfying \( (\tilde{P} e_1^{(i-1)}) [h] = (\tilde{P} e_2^{(i-1)}) [h] = 0 \) for \( h = 1, \ldots, i - 1 \). Hence, by randomness of \( \tilde{P} \), a vector \( e_1^{(i-1)} + e_2^{(i-1)} \) is a representation of \( e_1^{(i)} \) with probability

\[
p_{i,m} := \frac{2^{\ell_i}}{\prod_{h=1}^{i} 2^{\ell_h}} = \frac{1}{\prod_{h=1}^{i-1} 2^{\ell_h}}.
\]

Thus an expected number of \( R_i \cdot p_{i,m} \) representations is kept in \( L^{(i)}_1 \). This holds for all vectors on all levels \( i = 2, \ldots, m - 1 \) and the final vector \( e' \) on level \( m \). Under condition 4.12 there is at least one representation in expectation for every vector on every layer. To sum up, the solution \( e' \) survives throughout the algorithm. \( \square \)
We would like to point out that above Lemma 4.3 is not vacuous. Using Lemma 1.1, Eq. 4.12 becomes up to polynomial factors
\[
\sum_{i=1}^{m} \ell_i = \ell \quad \text{and} \quad p_i + (k + \ell - p_i) \cdot H \left( \frac{p_i - 1 - p_i/2}{k + \ell - p_i} \right) \geq \sum_{h=1}^{i-1} \ell_h \quad \forall i = 2, \ldots, m.
\]
For any parameters \(\ell_i\) such that \(\sum_{i=1}^{m} \ell_i = \ell\) and \(p_i - 1 = k + \ell - p_i\) for \(i = 2, \ldots, m\) this is equivalent to
\[
k + \ell \geq \sum_{h=1}^{i-1} \ell_h, \quad \forall i = 2, \ldots, m.
\]
which is always fulfilled. Next we provide formulas for running time and memory consumption.

**Lemma 4.4 (Complexity of BJMM).** Let \((P, s, w) \in \mathbb{F}_2^{(n-k)\times n} \times \mathbb{F}_2^{n-k} \times [n]\) be an instance of the syndrome decoding problem. For any \(\varepsilon > 0\) and parameters \(\ell_i, p_i \in \mathbb{N}\), \(i = 1, \ldots, m\), satisfying condition 4.12, Alg. 4.4 with Alg. 4.6 as a subroutine runs in expected time \(T\) with memory consumption \(M\), where
\[
T = P^{-1}_\pi \max_{0 \leq i \leq m-1} \{T_i\}, \quad M = \max_{0 \leq i \leq m-1} \{S_i\}
\]
with
\[
S_0 := \left( \frac{k+\ell}{p_1/2} \right), \quad S_i := \left( \frac{k+\ell}{p_i} \right) \cdot \prod_{h=1}^{i} 2^{\ell_h}, \quad i = 1, \ldots, m-1
\]
and
\[
T_i := 2^{(\max(\lambda, 2\lambda_i - \ell_i + 1) + \varepsilon)n}, \quad \lambda_i := \log(S_i) / n, \quad i = 0, \ldots, m-1.
\]

**Proof.** Let us first consider Alg. 4.6. The lists \(L_j^{(0)}\), \(j = 1, \ldots, 2^m\) on level 0 have size \(S_0\) and can be determined in time \(S_0\). On each level \(i = 1, \ldots, m-1\) we expect to construct a \(1/\sum_{h=1}^{i} \ell_h\)-fraction of all \(\binom{k+\ell}{p_i}\) vectors with length \(k + \ell\) and weight \(p_i\) by randomness of \(P\). Thus the resulting lists have expected size
\[
\mathbb{E}[|L_j^{(i)}|] = S_i, \quad j = 1, \ldots, 2^{m-i}
\]
Every Sort-and-Match in step 2 on level \(i\) and \(\ell_{i+1}\) coordinates as well as subsequent filtering for weight \(p_i\) on \(k + \ell\) bits can be done in time \(T_i\) for \(i = 0, \ldots, m-1\) using Lemma 1.2 and Lemma 1.3. Finally, Alg. 4.4 finds a good permutation \(\pi\) with probability \(P_\pi\). Hence, using \(T_0 \geq S_0\), the algorithms terminate in expected time \(T\) and use expected memory \(M\). \(\Box\)

Now we can apply the two lemmas and proof Prop. 4.2.
Proof (of Proposition 4.2). We use Lemma 1.1 to upper bound the $\lambda_i$ and the binomial coefficients. Numerical optimization yields the following parameters and the claimed complexities. For full distance decoding the worst-case running time is achieved a rate of

$$\frac{k}{n} = 0.426 \text{ with relative distance } \frac{w}{n} = \frac{d}{n} = H^{-1} \left( 1 - \frac{k}{n} \right) = 0.1361.$$ 

In this case the running time is minimized for $m = 3$ and choosing the relative parameters

$$p_1 = 0.01816, \quad p_2 = 0.03418, \quad p_3 = 0.04980, \quad \ell = 0.1902.$$ 

This results in $R_2 = 2^{0.04545n}$ and $R_3 = 2^{0.1182n}$ representations. We furthermore choose

$$\ell_1 := \log(R_2), \quad \ell_2 := \log(R_3) - \ell_1, \quad \ell_3 := \ell - \ell_1 - \ell_2$$

such that Eq. 4.12 is met and therefore the algorithms returns a solution in expectation. The resulting list sizes and running times are

$$S_0 = 2^{0.05908n}, \quad S_1 = 2^{0.07270n}, \quad S_2 = 2^{0.07243n},$$
$$T_0 = 2^{0.07270n}, \quad T_1 = 2^{0.07275n}, \quad T_2 = 2^{0.07275n}$$

while the probability for the correct weight distribution is $P_s = 2^{-0.02922n}$. This yields the claimed running time and memory consumption. For half distance decoding the running time and memory consumption can be shown analogously for a worst-case code rate

$$\frac{k}{n} = 0.448 \text{ with relative distance } \frac{w}{n} = \frac{d}{2n} = H^{-1} \left( 1 - \frac{k}{n} \right) = 0.06401,$$

depth $m = 3$ and relative parameters

$$p_1 = 0.004286, \quad p_2 = 0.008410, \quad p_3 = 0.01140, \quad \ell = 0.0618.$$ 

$\square$
In this chapter we present improved variants of the BJMM-algorithm which use a nearest neighbor search similar to the algorithms for the approximate $k$-list problem described in Chapter 3. We will see that both classes of algorithms share many commonalities. We first present a BJMM variant with nearest neighbor search which was first presented by May and Ozerov in [MO15] and improved in our work [BM17b]. Then we describe a new algorithm we presented in [BM18] which goes back to the SimpleISD-framework. Both are joint works with Alexander May.

### 5.1 A Link to the Approximate $k$-List Problem

Recall that the BJMM (Alg. 4.6) essentially reduces the weight successively through eliminating length $\ell_i$ blocks of bits in the vector $\tilde{P}e'$ until the target $w - p_m$ is reached. This is done in a binary search tree fashion. However, this is very similar to the Match-and-Filter-algorithm (Alg. 3.1) for the approximate $k$-list problem where blocks of length $c$ are eliminated per step until the target weight $w$ is reached. The first main difference is the additional part $e'$, which is carried through all lists and whose weight provides an additional restriction. In addition to this we are given the promise of an existing golden solution induced by the good permutation, while in the approximate $k$-list setting we choose list sizes such that we can expect a solution.

**BJMM with Nearest Neighbors.** However the general structure of the two algorithms is the same. In Section 3.2 we presented a variant of our approximate $k$-tree-algorithm which is essentially a combination of the Match-and-Filter-algorithm and the NN-SEARCH. This resulted in a small but remarkable improvement of the
complexity. Therefore it seems reasonable to combine the BJMM with nearest neighbor search, too. This was first suggested by May and Ozerov [MO15] and results in a slightly different algorithm (Alg. 5.1). In detail, the two lists \( L_1^{(m-1)} \), \( L_2^{(m-1)} \) on level \( m-1 \) are combined directly via NN-Search for weight \( w - p_m \) now instead of Sort-and-Match and consecutive filtering for weight \( w - p_m \) (see Fig. 5.1). Since its condition is always applied, the May-Ozerov nearest neighbor search can be used.

\[
T_{m-1} := 2(\tau(\frac{n-m-1}{n})^3+\varepsilon)n \quad \text{where} \quad \delta := \frac{n-k-\ell}{n} \quad \text{and} \quad \gamma := \frac{w-p_m}{n}.
\]

Let us take a closer look at the parameter \( m \), the number of levels in the binary search tree. For the original BJMM without NN-Search numerical optimizations showed that \( m = 3 \) is optimal and a higher amount of levels does not decrease the running time anymore. May and Ozerov also carried on with three levels in [MO15] improving the complexities upon [BJMM12] in both full distance and half distance decoding. However, as we were able to show in [BM17b], the running time for full distance decoding can be further reduced if we increase the number of levels to \( m = 4 \).
Algorithm 5.1: BJMM-NN\((m)\)

**Input** : \(\tilde{P} \in \mathbb{F}_2^{(n-k) \times (k+\ell)}, \tilde{s} \in \mathbb{F}_{2^{n-k}}, w \in \mathbb{N}\)

**Output** : \((e', e'') \in \mathbb{F}_2^{k+\ell} \times \mathbb{F}_2^{n-k-\ell}\)

**Parameters**:
- \(\ell := (\ell_1, \ldots, \ell_{m-1}, n-k, k+\ell) \in \mathbb{N}^{m+1}\) with \(\ell := \sum_{i=1}^{m-1} \ell_i\), \(p_1, \ldots, p_m \in \mathbb{N}\).
- \(L_j^{(0)}, j = 1, \ldots, 2^m\) as defined in (4.11).
- for \(i = 1, \ldots, m-1\) do
  - for \(j = 1, \ldots, 2^{m-1}\) do
    - \(L_j^{(i)} \leftarrow \text{SORT-AND-MATCH}(L_j^{(i-1)}, L_j^{(i-1)}, i, 0)^n\)
    - \(L_j^{(i)} \leftarrow \text{Filter}(L_j^{(i)}, m+1, p_i)\)
  - end
- \(L_1^{(m)} \leftarrow \text{NN-Search}(L_1^{(m-1)}, L_2^{(m-1)}, m, w-p_m)^n\)
- \(L_1^{(m)} \leftarrow \text{Filter}(L_1^{(m)}, m+1, p_m)\)
- if \(|L_1^{(m)}| > 0\) then return \((e', e'')\) for some \((0, e'', e') \in L_1^{(m)}\)
- else return \(\perp\)

**Proposition 5.1.** The syndrome decoding problem can in expectation be solved in time \(2^{0.0953n}\) and memory \(2^{0.0915n}\) for full distance decoding as well as time \(2^{0.0473n}\) and memory \(2^{0.0363n}\) for half distance decoding.

**Proof.** We use Lemma 1.1 to upper bound the \(\lambda_i\) and the binomial coefficients. Numerical optimization yields the following parameters and the claimed complexities. For full distance decoding the worst-case running time is achieved at a rate of

\[
\frac{k}{n} = 0.423 \quad \text{with relative distance} \quad \frac{w}{n} = \frac{d}{n} = H^{-1}\left(1 - \frac{k}{n}\right) = 0.1373.
\]

In this case the running time is minimized for \(m = 4\) and choosing the relative parameters

\[
\frac{p_1}{n} = 0.0298, \quad \frac{p_2}{n} = 0.0521, \quad \frac{p_3}{n} = 0.0734, \quad \frac{p_4}{n} = 0.0825, \quad \frac{\ell}{n} = 0.2635.
\]

This results in \(R_2 = 2^{0.0856n}\), \(R_3 = 2^{0.1771n}\) and \(R_4 = 2^{0.2635n}\) representations. Defining

\[
\ell_1 := \log(R_2), \quad \ell_2 := \log(R_3) - \ell_1, \quad \ell_3 := \ell - \ell_1 - \ell_2
\]

Eq. 4.12 is met and therefore the algorithm returns a solution in expectation. The
The resulting list sizes and running times are

\[ S_0 = 2^{0.0915n}, \quad S_1 = 2^{0.0915n}, \quad S_2 = 2^{0.0888n}, \quad S_3 = 2^{0.0732n}, \]

\[ T_1 = 2^{0.0915n}, \quad T_2 = 2^{0.0915n}, \quad T_3 = 2^{0.0913n}, \quad T_4 = 2^{0.0915n} \]

while the probability for the correct weight distribution is \( P_\pi = 2^{-0.0038n} \). This yields the claimed running time and memory consumption. For half distance decoding the running time and memory consumption can be shown analogously for a worst-case code rate

\[ \frac{k}{n} = 0.474 \text{ with relative distance } \frac{w}{n} = \frac{d}{2n} = H^{-1}\left(1 - \frac{k}{n}\right) = 0.05942, \]

depth \( m = 3 \) and relative parameters

\[ \frac{p_1}{n} = 0.0090, \quad \frac{p_2}{n} = 0.0150, \quad \frac{p_3}{n} = 0.0177, \quad \frac{\ell}{n} = 0.0663. \]

\[ \square \]

**Results.** Let us take a look how the algorithm behaves for different numbers of layers. Fig. 5.2 provides running time and memory consumption for full distance decoding as well as half distance decoding for different choices of \( m \).

<table>
<thead>
<tr>
<th>( m )</th>
<th>( \log(P_\pi^{-1})/n )</th>
<th>( \log(T)/n )</th>
<th>( \log(M)/n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.0211</td>
<td>0.1003</td>
<td>0.0792</td>
</tr>
<tr>
<td>3</td>
<td>0.0094</td>
<td>0.0967</td>
<td>0.0873</td>
</tr>
<tr>
<td>4</td>
<td>0.0038</td>
<td>0.0953</td>
<td>0.0915</td>
</tr>
<tr>
<td>5</td>
<td>0.0043</td>
<td>0.0953</td>
<td>0.0910</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( m )</th>
<th>( \log(P_\pi^{-1})/n )</th>
<th>( \log(T)/n )</th>
<th>( \log(M)/n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.0210</td>
<td>0.0492</td>
<td>0.0282</td>
</tr>
<tr>
<td>3</td>
<td>0.0110</td>
<td>0.0473</td>
<td>0.0363</td>
</tr>
<tr>
<td>4</td>
<td>0.0122</td>
<td>0.0473</td>
<td>0.0351</td>
</tr>
</tbody>
</table>

Fig. 5.2: Upper bounds for running time and memory exponents of the BJMM-NN(\( m \)).

Interestingly, the running time \( P_\pi^{-1} \) for finding a suitable permutation \( \pi \) of the outer loop decreases with increasing \( m \) until \( m \) yields minimal running time. While it is significant for \( m = 2 \), it nearly vanishes for \( m = 4 \) in full distance decoding and nearly all the work is done in the inner loop of Alg. 5.1. Thus, for \( m = 5 \), there is not much work anymore which can be shifted to the inner loop. This may be an explanation why the algorithm finds it optimum for a certain number of levels. Another reason for this can be found in Fig. 5.3 which provides more detailed information for the time and space consumption on each level of the search tree. For \( m = 2, 3, 4 \) the initial lists of size \( S_0 \) (and the lists after the first step of size \( S_1 \) for \( m = 4 \)) are the biggest. Throughout the algorithm the list sizes decrease until a minimum list size \( S_{m-1} \) is reached. This is
a compensation for the higher running time $T_m$ of NN-SEARCH, which runs on those lists. In contrast to this the running times $T_i$, $i = 0, \ldots, m$ are balanced throughout the algorithm. However, for $m = 5$ this does not hold and $S_1$ is much smaller than $S_2$. The optimization cannot balance the first list sizes anymore and therefore cannot provide a further improvement.

\[
\begin{array}{cccccc}
  m & \log(T_1)/n & \log(T_2)/n & \log(T_3)/n & \log(T_4)/n & \log(T_5)/n \\
  2 & 0.0792 & 0.0792 & - & - & - \\
  3 & 0.0873 & 0.0873 & 0.0873 & - & - \\
  4 & 0.0915 & 0.0915 & 0.0913 & 0.0915 & - \\
  5 & 0.0910 & 0.0909 & 0.0910 & 0.0910 & 0.0910 \\
\end{array}
\]

\[
\begin{array}{cccccc}
  m & \log(S_0)/n & \log(S_1)/n & \log(S_2)/n & \log(S_3)/n & \log(S_4)/n \\
  2 & 0.0792 & 0.0624 & - & - & - \\
  3 & 0.0873 & 0.0873 & 0.0692 & - & - \\
  4 & 0.0915 & 0.0915 & 0.0888 & 0.0731 & - \\
  5 & 0.0910 & 0.0727 & 0.0909 & 0.0881 & 0.0725 \\
\end{array}
\]

Fig. 5.3: Exponents for running time and memory consumption for all search tree levels in full distance decoding.

In contrast to the Approx-$k$-Tree-algorithm in Chapter 3 the choice $m = 4$ (resp. $m = 3$) for full distance (resp. half distance) decoding is optimal and a further increment of the number of layers does not yield reduced running times. This is another evidence that despite the many commonalities between the settings, there are still some differences. In Section 5.2, we present in fact an algorithm, which improves upon BJMM-NN since it uses the NN-SEARCH-algorithm on every level in the binary search tree and is thus less restrictive.

### 5.2 A New Algorithm for Simple ISD

In the previous chapter we presented algorithms in the SimpleISD-framework as well as in the AdvancedISD-framework. One common thing found in most of the algorithms is some sort of exact matching on $\ell$ coordinates among the candidates $(e_1', e_2')$ and subsequent filtering for specific weight. This offers the possibility to use techniques similar to the (approximate) $k$-list problem.

However, the AdvancedISD-framework has two big disadvantages compared to SimpleISD. First, the length of the vector $e'$ is increased from $k$ to $k + \ell$ which increases the search space. Moreover this leads to a more restrictive weight distribution since it fixes small weight $p \ll w$ on $k + \ell$ instead of $k$ coordinates. It becomes harder to find a permutation $\pi$ which induces this weight distribution increasing the number of
necessary iterations in Alg. 4.4. This raises the question whether it is possible to take the best out of both worlds, namely using a binary search tree and NN-SEARCH in the SIMPLEISD-framework. For example, it is possible to apply a BJMM-like binary search tree directly to the exact matching part of Stern’s algorithm from Section 4.3 in the SIMPLEISD-framework. However, this restricts the weight distribution greatly because we fix \( \ell \) zero bits in the \( e'' \) vector. In contrast to this the exact matching in ADVANCEDISD does not affect the target vector \( e \), i.e. no zero bits are fixed, which is a huge advantage. Therefore it is reasonable to omit the exact matching completely.

In this section we describe a new algorithm first presented in our work \([BM18]\) which goes back to the SIMPLEISD-framework (Alg. 4.1) and Eq. 4.3 (\( \Delta(\bar{P}e', \bar{s}) = w - p \)). Splitting \( e' = e_1^{(1)} + e_2^{(1)} \) for \( e_1^{(1)}, e_2^{(1)} \in \mathbb{F}_2^k \) yields

\[
\Delta(\bar{P}e_1^{(1)}, \bar{P}e_2^{(1)} + \bar{s}) = w - p. \tag{5.1}
\]

Our algorithm constructs two lists \( L_1^{(1)}, L_2^{(1)} \) with entries

\[
(\bar{P}e_1^{(1)}, e_1^{(1)}) \text{ and } (\bar{P}e_2^{(1)} + \bar{s}, e_2^{(1)}) \text{ such that }
\Delta(e') = \Delta(e_1^{(1)} + e_2^{(1)}) = p \text{ and } \Delta(e'') = w - p.
\]

which therefore yields a solution \((e', e'')\) of Eq. 4.3 and thus for the decoding problem in standard form. We construct the two lists \( L_1^{(1)}, L_2^{(1)} \) with a binary search tree of depth \( m \) again.

The Depth-2 Algorithm. First, let us have a detailed look at the easiest case \( m = 2 \) which already shows the main idea. This algorithm is used as a subroutine in the SIMPLEISD-framework (Alg. 4.4) and replaces the solving algorithm (Alg. 4.5). We advise the reader to follow the description via Fig. 5.4.

We introduce a split into \( \ell_1 \) and \( \ell_2 := n - k - \ell_1 \) coordinates to the vector \( e'' \) of length \( n - k \) and enforce weight \( w_1 \) on the first as well as weight \( w_2 := w - p - w_1 \) on the second block. The parameters \( \ell_1, w_1 \) are subject to optimization. Thus our algorithm only finds a solution \((e', e'')\) if \( e' \) has weight \( p \) and satisfies \( \bar{P}e' + \bar{s} = e'' = (e''[1], e''[2]) \) with \( \Delta(e''[1]) = w_1 \) and \( \Delta(e''[2]) = w_2 \). The column permutation \( \pi \) in Alg. 4.1 has to induce this specific weight distribution.

Definition 5.1 (Good Permutation IV). Let \( e \in \mathbb{F}_2^n \) with \( \Delta(e) = w \) and \( k, p \in \mathbb{N} \) as well as \( \ell_1, \ell_2 \in \mathbb{N} \) with \( \ell_1 + \ell_2 = n - k \), and \( w_1, w_2 \in \mathbb{N} \) with \( w_1 + w_2 = w - p \). We call a permutation \( \pi \) good for \( e \) with respect to Alg. 5.2 and parameters \((k, p, w_1, \ell_1, w_2, \ell_2)\),

87
if $\pi^{-1}e = (e', e''_{[1]}, e''_{[2]}) \in \mathbb{F}_2^k \times \mathbb{F}_2^{\ell_1} \times \mathbb{F}_2^{\ell_2}$ with

$$\Delta(e') = p, \quad \Delta(e''_{[1]}) = w_1 \text{ and } \Delta(e''_{[2]}) = w_2.$$ 

A random permutation $\pi$ is good with probability

$$P_{\pi} = \binom{k}{w_1} \binom{\ell_1}{w_2} \binom{\ell_2}{w}.$$ 

Hence, our final list $L^{(2)}_1$ contains vectors of the form $(\bar{P}e' + \bar{s}, e')$ such that $\Delta(e') = p, \Delta(e''_{[1]}) = w_1$ and $\Delta(e''_{[2]}) = w_2$. We construct this list out of two lists $L^{(1)}_1, L^{(1)}_2$ containing elements $(\bar{P}e^{(1)}_1, e^{(1)}_1), (\bar{P}e^{(1)}_2, e^{(1)}_2)$ respectively. We enforce weight $\Delta(e^{(1)}_1) = \Delta(e^{(1)}_2) = p_1 \geq \frac{p}{2}$ and weight $\Delta((\bar{P}e^{(1)}_1)_{[1]}) = \Delta((\bar{P}e^{(1)}_2)_{[1]}) = w_1^{(1)}$ on the first $\ell_1$ coordinates. The parameters $p_1$ and $w_1^{(1)}$ are again subject to optimization. In the construction of the final list $L^{(2)}_1$ we run NN-SEARCH on $\ell_2$ bits for weight $w_2$ as a first step. Filtering for weight $p$ on $k$ coordinates and weight $w_1$ on $\ell_1$ coordinates completes the computation.

Furthermore, we construct $e^{(1)}_1 = e^{(0)}_1 + e^{(0)}_2$ where

$$\Delta(e^{(0)}_1) = \Delta(e^{(0)}_2) = \frac{p_1}{2} \text{ and } e^{(0)}_1 \in \mathbb{F}_2^{k/2} \times \mathbf{0}^{k/2}, e^{(0)}_2 \in \mathbf{0}^{k/2} \times \mathbb{F}_2^{k/2}.$$ 

Thus the first list $L^{(1)}_1$ is constructed out of lists $L^{(0)}_1, L^{(0)}_2$ containing elements $(\bar{P}e^{(0)}_1, e^{(0)}_1)$ and $(\bar{P}e^{(0)}_2, e^{(0)}_2)$ using the meet-in-the-middle technique again. The lists are combined via the NN-SEARCH-algorithm on the first $\ell_1$ coordinates for weight $w_1^{(1)}$.
to obtain the list $L_1^{(1)}$ where the vectors $e_1^{(0)}$ and $e_2^{(0)}$ add to a vector of the correct weight $p_1$ by construction.

**Description of the Depth-2 Algorithm.** Let us now describe the whole algorithm (see Alg. 5.2) in detail. On level 0 we start with four lists

$$L_1^{(0)} = \{(\bar{P}e_1^{(0)}, e_1^{(0)}) \in \mathbb{F}_2^{n-k} \times \mathbb{F}_2^{k/2} \times \mathbb{O}^{k/2} | \Delta(e_1^{(0)}) = p_1/2\},$$

$$L_2^{(0)} = \{(\bar{P}e_2^{(0)}, e_2^{(0)}) \in \mathbb{F}_2^{n-k} \times \mathbb{O}^{k/2} \times \mathbb{F}_2^{k/2} | \Delta(e_2^{(0)}) = p_1/2\},$$

$$L_3^{(0)} = \{(\bar{P}e_3^{(0)}, e_3^{(0)}) \in \mathbb{F}_2^{n-k} \times \mathbb{F}_2^{k/2} \times \mathbb{O}^{k/2} | \Delta(e_3^{(0)}) = p_1/2\},$$

$$L_4^{(0)} = \{(\bar{P}e_4^{(0)} + \bar{s}, e_4^{(0)}) \in \mathbb{F}_2^{n-k} \times \mathbb{O}^{k/2} \times \mathbb{F}_2^{k/2} | \Delta(e_4^{(0)}) = p_1/2\}.$$

Note that $L_1^{(0)} = L_3^{(0)}$. The lists are pairwise combined via NN-Search on the first $\ell_1$ coordinates for weight $w_1^{(1)}$ yielding the two lists on level 1

$$L_1^{(1)} = \{(\bar{P}e_1^{(1)}, e_1^{(1)}) \in \mathbb{F}_2^{n-k} \times \mathbb{F}_2^k | \Delta(e_1^{(1)}) = p_1, \Delta((\bar{P}e_1^{(1)})_{[1]} = w_1^{(1)}\},$$

$$L_2^{(1)} = \{(\bar{P}e_2^{(1)} + \bar{s}, e_2^{(1)}) \in \mathbb{F}_2^{n-k} \times \mathbb{F}_2^k | \Delta(e_2^{(1)}) = p_1, \Delta((\bar{P}e_2^{(1)} + \bar{s})_{[1]} = w_1^{(1)}\}.$$

Another nearest neighbor search on $\ell_2$ bits for weight $w_2$ and subsequent filtering for weight $p$ on the last $k$ coordinates as well as weight $w_1$ on the first $\ell_1$ bits leads to the final list

$$L^{(2)} = \{(e'', e') \in \mathbb{F}_2^k \times \mathbb{F}_2^{n-k} | \Delta(e') = p, e'' = \bar{P}e' + \bar{s}, \Delta(e'') = w - p\}.$$

Thus any element in this list refers to a solution $(e', e'')$ of the given syndrome decoding problem in standard form $(\bar{P}, \bar{s}, w)$. Again, we keep checksums of the tuples after each nearest neighbor search since we are interested in the sum of every two vectors only. This algorithm with depth $m = 2$ already beats the bound $2^{0.102n}$ of the original BJMM-algorithm with depth 3.

89
then in expectation we have

Lemma 5.1 (Correctness of Depth-2-ISDSolve). Let \( e \) be a solution to the instance \( (P, s, w) \in F_2^{(n-k) \times n} \times F_2^{n-k} \times [n] \) of the syndrome decoding problem with standard form \( (\bar{P}, \bar{s}, w) \). Let \( \pi \) be a good permutation for \( e \) as given by Def. 5.1 with parameters \( (k, p, w_1, \ell_1, w_2, \ell_2) \). If Alg. 5.2 runs with parameters \( p_1, w_1^{(1)} \in \mathbb{N} \) satisfying

\[
\frac{p}{p/2} \left( \frac{k-p}{p_1-p/2} \right) \geq \frac{2^{|\ell_1|}}{(w_1^{(1)}-w_1/2)},
\]

then in expectation we have \( (e'', e') \in L_2^{(2)} \) for \( \pi^{-1} e = (e', e'') \).

Proof. Let \( (e', e'') \) be the solution of the standard form we are looking for. Since we assume that \( \pi \) is good, we have \( \Delta(e'') = w - p \). Furthermore \( e' \) fully determines \( e'' \) by \( \bar{P}e' + \bar{s} = e'' \) and the algorithm exactly finds such candidates in the first component. It is therefore sufficient to show that the desired \( e' \) is constructed by the algorithm.

On the first level, all vectors \( e_1^{(0)}, e_2^{(0)} \) are combined to obtain \( e_1^{(1)} = e_1^{(0)} + e_2^{(0)} \). Up to polynomial factors all vectors \( e_1^{(1)} \) and \( e_2^{(1)} \) respectively are constructed this way analogously to Lemma 4.3.

Let us now look at the construction of \( e' \) with weight \( p \) as the sum of two vectors.
\( \mathbf{e}_1^{(1)}, \mathbf{e}_2^{(1)} \) with weight \( p_1 \geq p \). By Lemma 4.2, the vectors has

\[
R_2 := \left( \frac{p}{p/2} \right) \left( \frac{k + \ell - p}{p_1 - p/2} \right)
\]

representations, while it is sufficient to keep one representation, i.e. a \( \frac{1}{R_2} \)-fraction in \( L_1^{(1)} \times L_2^{(1)} \). Recall that \( \Delta((\bar{P}\mathbf{e} + \bar{s})_{[1]}) = w_1 \) by definition of \( \mathbf{e}' \). Furthermore the lists on level 1 only contain vectors \((\bar{P}\mathbf{e}_1^{(1)}, \mathbf{e}_1^{(1)}) \in L_1^{(1)}, (\bar{P}\mathbf{e}_2^{(1)} + \bar{s}, \mathbf{e}_2^{(1)}) \in L_2^{(1)} \) satisfying \( \Delta((\bar{P}\mathbf{e}_1^{(1)})_{[1]}) = \Delta((\bar{P}\mathbf{e}_2^{(1)} + \bar{s})_{[1]}) = w_1^{(1)} \), i.e. we enforce weight \( w_1^{(1)} \) on \( \ell_1 \) bits. Let \( E \) be the event that there exists a representation of

\[
\mathbf{e}''_1 = (\bar{P}\mathbf{e}_1^{(1)} + \bar{P}\mathbf{e}_2^{(1)} + \bar{s})_{[1]} \text{ with } \Delta((\bar{P}\mathbf{e}_1^{(1)})_{[1]}) = \Delta((\bar{P}\mathbf{e}_2^{(1)} + \bar{s})_{[1]}) = w_1^{(1)}.
\]

There are a total of \( 2^{\ell_1} \) possible representations for \( \mathbf{e}''_1 \). By Lemma 4.2, \( \left( \frac{w_1}{w_1/2} \right) \left( \frac{\ell_1 - w_1}{w_1^{(1)} - w_1/2} \right) \) of them have the correct weight \( w_1^{(1)} \) for \((\bar{P}\mathbf{e}_1^{(1)})_{[1]}, (\bar{P}\mathbf{e}_2^{(1)} + \bar{s})_{[1]} \). Hence, by randomness of \( \bar{P} \) a vector \( \mathbf{e}_1^{(1)} + \mathbf{e}_2^{(1)} \) is a representation of \( \mathbf{e}' \) with probability

\[
p_{2,2} := \Pr[E] = \frac{\left( \frac{w_1}{w_1/2} \right) \left( \frac{\ell_1 - w_1}{w_1^{(1)} - w_1/2} \right)}{2^{\ell_1}}.
\]

Thus an expected number of \( R_2 \cdot p_{2,2} \) representations is kept in then final list \( L_1^{(2)} \). Under condition 5.3 we have \( R_2 \cdot p_{2,2} \geq 1 \), i.e. there survives at least one representation of the solution \( \mathbf{e}' \) in expectation.

We would like to point out that above Lemma is not vacuous. With \( p_1 = k/2 \) and \( w_1^{(1)} = \ell_1/2 \), condition 5.3 collapses to

\[
2^{p+k-p} \geq 2^{\ell_1 - w_1} \quad \Leftrightarrow \quad k \geq 0,
\]

which is trivially fulfilled. We used Lemma 1.1 here once again.

**Lemma 5.2 (Complexity of Depth-2-ISDSolve).** Let \((P,s,w) \in \mathbb{F}_2^{(n-k) \times n} \times \mathbb{F}_2^{-k} \times [n]\) be an instance of the syndrome decoding problem. For any constant \( \varepsilon > 0 \), parameters \( p, p_1, \ell_1, w_1^{(1)}, w_2 \), satisfying condition 5.3, and \( \ell_2 := n - k - \ell_1, w_2 := w - p - w_1 \), Alg. 4.1 with Alg. 5.2 as a subroutine runs in expected time \( T \) and list sizes \( S_0, S_1 \) on levels 0, 1 where

\[
T = P^{-1} \cdot \max\{T_0, T_1\}, \quad S_0 := \left( \frac{k}{2^{p_1/2}} \right), \quad S_1 := \left( \frac{k}{p_1} \right) \cdot \left( \frac{\ell_1}{2^{\ell_2}} \right),
\]

with

\[
T_0 := 2^{\left( r N \left( \frac{\log(\varepsilon)}{n\theta} + \frac{k}{\ell_1} \right) \right) \varepsilon n} \quad \text{and} \quad T_1 := 2^{\left( r N \left( \frac{\log(\varepsilon)}{n\theta} + \frac{k}{\ell_1} \right) \right) \varepsilon n}.
\]

91
Proof. Let us first consider Alg. 5.2. The lists \( |L_j^{(0)}| \), \( j = 1, 2, 3, 4 \) on level 0 have size \( S_0 \) and are created in time \( T_0 \). The nearest neighbor search on \( \ell_1 \) bits for weight \( w_1^{(1)} \) in step 2 can be done in time \( T_0 \) by Lemma 1.4. We expect to construct a \( \binom{k}{p_1} \) fraction of all \( \binom{k}{p_1} \) vectors with length \( k \) and weight \( p_1 \). Thus, by randomness of \( \bar{P} \), the resulting lists have expected size

\[
\mathbb{E}[|L_j^{(1)}|] = S_1, j = 1, 2.
\]

The two lists are combined via NN-Search on \( \ell_2 \) bits for weight \( w_2 \) again in step 3 which takes time \( T_1 \) using Lemma 1.4 again and results in the final list \( L^{(2)} \). The filtering (step 4) takes time \( \leq T_1 \) and can be done on the fly i.e. we do not need to store the final list since we can return the first found solution.

Thus Alg. 5.2 runs in time \( \max\{T_0, T_1\} \) using \( T_0 \geq S_0 \). Since Alg. 4.1 finds a good permutation \( \pi \) with probability \( P_\pi \) one can expect to find it after an expected number of \( P_\pi^{-1} \) iterations. Hence, the algorithm terminates in expected time

\[
T = P_\pi^{-1} \cdot \max\{T_0, T_1\}.
\]

We are now able to apply the two lemmas and proof Prop. 5.2.

Proof of Proposition 5.2. We use Lemma 1.1 to upper bound the \( \lambda_i \) and the binomial coefficients. Numerical optimization yields the following parameters and the claimed complexities. For full distance decoding the worst-case running time is achieved a rate of

\[
\frac{k}{n} = 0.43 \text{ with relative distance } \frac{w}{n} = \frac{d}{n} = H^{-1} \left( 1 - \frac{k}{n} \right) = 0.1346.
\]

In this case the running time is minimized for the relative parameters

\[
\frac{p}{n} = 0.03730, \quad \frac{p_1}{n} = 0.02645,
\]

resulting in \( R_1 = 2^{0.09254n} \) representations. We furthermore choose

\[
\frac{\ell_1}{n} = 0.1553 \text{ and } \frac{w_1}{n} = 0.01970
\]

and use condition Eq.(5.3) from Lemma 5.1 to obtain

\[
\frac{w_1^{(1)}}{n} = 0.01765.
\]
The resulting running times and list sizes are

\[ T_0 = 2^{0.08483n}, \quad T_1 = 2^{0.07168n}, \quad S_0 = 2^{0.06741n}, \quad S_1 = 2^{0.01334n} \]

using the May-Ozerov nearest neighbor search on both layers. The probability for the correct weight distribution satisfying Def. 5.1 is \( P_\pi = 2^{-0.01334n} \). This yields the claimed running time and memory consumption.

### The Depth-\( m \) Algorithm.

Our algorithm with depth \( m = 2 \) already illustrates the overall idea but does not yield improvements compared to previous results yet. We now generalize the algorithm to arbitrary depth. As this is straight-forward for the most part, we do not go into too much detail and mainly point out the subtle differences. The algorithm is again used as a subroutine in the SimpleISD-framework (Alg. 4.1) and replaces the solving algorithm (Alg. 4.2). We advise the reader to follow the description via Fig. 5.5 which shows the algorithm for \( m = 3 \).

![Fig. 5.5: Our algorithm for depth 3.](image)

We now enforce \( \Delta(e') = p_m \) and split the vector \( e'' \) into \( m \) blocks \( e'' = (e''_1, \ldots, e''_m) \) instead of only 2 blocks where each block has length \( \ell_i \) and enforced weight \( w_{i}^{(m)} \). The parameters \( \ell_i, w_{i}^{(m)} \) are subject to optimization with the restriction that \( \sum_{i=1}^{m} \ell_i = n - k \) and \( \sum_{i=1}^{m} w_{i}^{(m)} = w - p_m \). Again, our algorithm successfully finds a solution if the
column permutation $\pi$ in Alg. 4.1 induces the correct weight distribution.

**Definition 5.2 (Good Permutation V).** Let $e \in \mathbb{F}_2^m$ with $\Delta(e) = w$ and $k, p_m \in \mathbb{N}$. Furthermore let $\ell_1, \ldots, \ell_m \in \mathbb{N}$ with $\sum_{i=1}^{m} \ell_i = n - k$, and $w_1^{(m)}, \ldots, w_m^{(m)} \in \mathbb{N}$ with $\sum_{i=1}^{m} w_i^{(m)} = w - p_m$. We call a permutation $\pi$ good for $e$ with respect to Alg. 5.3 and parameters $(k, p_m, (w_i^{(m)}, \ell_i)_{i=1, \ldots, m})$, if $\pi^{-1} e = (e', e''_{[1]}, \ldots, e''_{[m]}) \in \mathbb{F}_2^k \times \mathbb{F}_2^k \times \cdots \times \mathbb{F}_2^k$ with

$$\Delta(e') = p_m, \quad \Delta(e''_{[i]}) = w_i^{(m)}, i = 1, \ldots, m.$$  

A random permutation $\pi$ is good with probability

$$P_\pi = \frac{\binom{k}{p_m} \prod_{i=1}^{m} \binom{\ell_i}{w_i}}{\binom{n}{w}}.$$  

Our goal is to have elements of this form only on the final list $L_1^{(m)}$.

**Description of the Depth-$m$ Algorithm.** Let us describe the whole algorithm (see Alg. 5.3) in detail. There are a total of $2^m$ lists on level 0 which are of the form

$$L_j = \{(\bar{P} \bar{e}_j(0), e_j(0)) \in \mathbb{F}_2^{n-k} \times \mathbb{F}_2^{k/2} \times \mathbb{F}_2^{k/2} \mid \Delta(e_j(0)) = p_1/2\},$$

(5.5)

$$L_{j_2} = \{(\bar{P} \bar{e}_{j_2}(0), e_{j_2}(0)) \in \mathbb{F}_2^{n-k} \times \mathbb{F}_2^{k/2} \times \mathbb{F}_2^{k/2} \mid \Delta(e_{j_2}(0)) = p_1/2\},$$

$$L_{2m} = \{(\bar{P} \bar{e}_{2m}(0), \bar{s}, e_{2m}(0)) \in \mathbb{F}_2^{n-k} \times \mathbb{F}_2^{k/2} \times \mathbb{F}_2^{k/2} \mid \Delta(e_{2m}(0)) = p_1/2\}$$

for $j_1 = 1, 3, \ldots, 2^m - 1$ and $j_2 = 2, 4, \ldots, 2^m - 2$. The algorithm combines two lists at a time in a binary tree wise fashion until the final list $L^{(m)}$ is left. In detail the first list $L_i^{(0)}$ on level $i = 1, \ldots, m - 1$ is constructed via NN-SEARCH for the lists $L_1^{(i-1)}$ and $L_2^{(i-1)}$ for weight $w_i^{(i)}$ on the $\ell_i$ coordinates of the $i$-th block. Subsequent filtering for weight $p_i$ on the last $k$ coordinates and a specific weight distribution on the remaining coordinates results in

$$L_i^{(i)} = \{(\bar{P} \bar{e}_i^{(i)}, e_i^{(i)}) \in \mathbb{F}_2^{n-k} \times \mathbb{F}_2^k \mid \Delta(e_i^{(i)}) = p_i, \Delta((\bar{P} \bar{e}_i^{(i)})_{[i]}) = w_i^{(i)}, h = 1, \ldots, i\}.$$  

The other lists $L_j^{(i)}$, $j = 2, \ldots, 2^i$ are created analogously. In the last step we then run another nearest neighbor search on the level $m - 1$ lists $L_1^{(m-1)}, L_2^{(m-1)}$ for weight $w_m^{(m)}$ on $\ell_m$ coordinates. With subsequent filtering for weight $p_m$ on the last $k$ coordinates and weight $w_i^{(m)}$ for every projection $e''_{[i]}, i = 1, \ldots, m - 1$ we obtain

$$L_1^{(m)} = \{(e'', e') \in \mathbb{F}_2^{n-k} \times \mathbb{F}_2^k \mid \Delta(e') = p_m, e'' = \bar{P} \bar{e} + \bar{s}, \Delta(e'') = w - p_m\}.$$  

94
Thus any element in this list yields a solution \((e',e'')\) of the given syndrome decoding problem in standard form \((\hat{P}, \hat{s}, w_m)\).

### Algorithm 5.3: Depth-\(m\)-ISDSolve

**Input:** \(\hat{P} \in \mathbb{F}_2^{n \times (n-k)}, \hat{s} \in \mathbb{F}_2^{n-k}, w \in \mathbb{N}\)

**Output:** \((e',e'') \in \mathbb{F}_2^n \times \mathbb{F}_2^k\)

**Parameters:** \(\ell := (\ell_1, \ldots, \ell_{m}, k) \in \mathbb{N}^{m+1}\) with \(\sum_{i=1}^{m} \ell_i := n-k\),

\(p_1, \ldots, p_m, w_1^{(m)}, \ldots, w_m^{(m)} \in \mathbb{N}\) with \(\sum_{i=1}^{m} w_i^{(m)} := w - p_m\).

Choose optimal \(w_j^{(i)}\) such that condition (5.6) holds.

1. Create lists \(L_j^{(0)}, j = 1, \ldots, 2^m\) as defined in (5.5).
2. \(L_j^{(1)} \leftarrow \text{NN-Search}(L_j^{(0)}, L_{2j-1}^{(0)}, 1, w_1^{(1)})^+, j = 1, \ldots, 2^{m-1}\)
3. For \(i = 2, \ldots, m\) do
   - \(L_j^{(i)} \leftarrow \text{NN-Search}(L_j^{(i-1)}, L_{2j}^{(i-1)}, i, w_i^{(i)})^+\)
   - \(L_j^{(i)} \leftarrow \text{Filter}(L_j^{(i)}, h, w_h^{(i)}), h = 1, \ldots, i-1\)
   - \(L_j^{(i)} \leftarrow \text{Filter}(L_j^{(i)}, m+1, p_i)\)

If \(|L_j^{(m)}| > 0\) then return \((e',e'')\) for some \((e'', e') \in L^{(m)}\) else return \(\perp\)

### Theorem 5.1.

The syndrome decoding problem can in expectation be solved in time \(2^{0.0885n}\) and memory \(2^{0.0736n}\) for full distance decoding as well as time \(2^{0.0465n}\) and memory \(2^{0.0294n}\) for half distance decoding.

We again proof first that our algorithm (Alg. 5.3) constructs a non-empty set of solutions of the induced permutation in Alg. 4.1 satisfies Def. 5.2.

### Lemma 5.3 (Correctness of Depth-\(m\)-ISDSolve).

Let \(e\) be a solution to the instance \((\hat{P}, \hat{s}, w) \in \mathbb{F}_2^{n \times (n-k)} \times \mathbb{F}_2^{n-k} \times \mathbb{N}\) of the syndrome decoding problem with standard form \((\hat{P}, \hat{s}, w)\). Let \(\pi\) be a good permutation for \(e\) as given by Def. 5.2 with parameters \((k, p_n, (w_i^{(m)}, \ell_i), i = 1, \ldots, m)\). If Alg. 5.3 runs with parameters \(p_i, w_i^{(i)} \in \mathbb{N}\), for \(j = 1, \ldots, i\), \(i = 1, \ldots, m-1\) satisfying

\[
\binom{p_i}{p_i/2} \binom{k-p_i}{p_i-1-p_i/2} \geq \prod_{h=1}^{i-1} \left( w_h^{(i)} / 2 \right) \left( w_h^{(i-1)} / 2 \right) \left( \ell_h - w_h^{(i)} / 2 \right), \quad \forall i = 2, \ldots, m \tag{5.6}
\]

then on expectation we have \((e'', e') \in L^{(m)}\) for \(\pi^{-1} e = (e', e'')\).
Proof. Let \((e', e'')\) be the solution of the standard form we are looking for. Similar to Lemma 5.1 it is sufficient to show that our algorithm constructs the desired \(e'\) as a sum of two vectors \(e_1^{(m-1)}\) and \(e_2^{(m-1)}\).

We have \(e_j^{(i)} = e_{2j-1}^{(i-1)} + e_{2j}^{(i-1)},\) for all \(j = 1, \ldots, 2^i, i = 1, \ldots, m - 1\) in our algorithm. Analogously to Lemma 5.1, we obtain up to a polynomial factor all \(2^k\) vectors \(e_j^{(1)} \in F_2^n\) on level 1.

For the construction of the remaining vectors, we exemplarily look at the vector \(e_1^{(i)}\) with weight \(p_i\) on level \(i\) via \(e_1^{(i-1)} + e_2^{(i-1)}\) with \(e_1^{(i-1)}, e_2^{(i-1)}\) having weight \(p_{i-1} \geq p_i/2\). The reasoning can later be generalized to all vectors on this layer and all layers \(i = 2, \ldots, m - 1\) as well as the final vector \(e'\) on level \(m\). By Lemma 4.2 the vector has

\[
R_i := \left( \frac{p_i}{p_{i-1}} \right) \left( \frac{k - p_i}{p_i - p_{i-1}/2} \right)
\]

representations, while it suffices to keep one of these representations, i.e. a \(\frac{1}{2^{n/2}}\)-fraction in \(L_1^{(i-1)} \times L_2^{(i-1)}\). Recall that our target vector \(e_1^{(i)}\) satisfies \(\Delta(\hat{P}e_1^{(i)}_{[h]})) = w_h^{(i)}\), \(h = 1, \ldots, i - 1\). On level \(i - 1\) we compute only those elements \((\hat{P}e_1^{(i-1)}, e_1^{(i-1)}) \in L_1^{(i-1)}\) satisfying \(\Delta((\hat{P}e_1^{(i-1)}_{[h]})) = w_h^{(i-1)}, h = 1, \ldots, i - 1\) (analogous for \(L_2^{(i-1)}\)) i.e. we enforce weight \(w_h^{(i-1)}\) on \(\ell_h\) bits for \(h = 1, \ldots, i - 1\). Let \(E\) be the event that there exists a representation of

\[(\hat{P}e_1^{(i)}_{[h]} = (\hat{P}e_1^{(i-1)} + \hat{P}e_2^{(i-1)})_{[h]}\) with \(\Delta((\hat{P}e_1^{(i-1)}_{[h]})) = \Delta((\hat{P}e_2^{(i-1)}_{[h]})) = w_h^{(i-1)}\).

for \(h = 1, \ldots, i - 1\). There are a total of \(2^{2k}\) possible representations. By Lemma 4.2, \(\left(\frac{w_h^{(i)}}{2}\right)(\frac{\ell_h - w_h^{(i)}}{2})\) of them have the correct weight \(w_h^{(i-1)}\) for \((\hat{P}e_1^{(i-1)}_{[h]}), (\hat{P}e_2^{(i-1)}_{[h]}).\)

Hence, by randomness of \(\hat{P}\) a vector is representation of \(e_1^{(i)}\) with probability

\[p_{i,m} := \Pr[E] = \prod_{h=1}^{i-1} \frac{\left(\frac{w_h^{(i)}}{2}\right)(\frac{\ell_h - w_h^{(i)}}{2})}{2^{2k}}.
\]

Therefore there survives at least one representation of \(e_1^{(i)}\) whenever \(R_i \cdot p_{i,m} \geq 1\) holds. Generalizing this to all layers yields to one expected solution in the final list if Condition 5.6 holds.

Analogously to Lemma 5.1 it can be shown that Condition 5.6 is trivially fulfilled for \(p_i = k/2\) and \(w_h^{(i)} = \ell_h/2, h = 1, \ldots, i - 1, i = 2, \ldots, m\).

Lemma 5.4 (Complexity of Depth-\(m\)-ISDSolve). Let \((P, s, w) \in F_2^{(n-k) \times n} \times F_2^{n-k} \times \mathbb{N}\) be an instance of the syndrome decoding problem. For any constant \(\varepsilon > 0\) and
parameters satisfying Condition 5.7, Alg. 4.1 with subroutine Alg. 5.3 runs in expected

time $T$ with list sizes $S_i$ on level $i = 0, \ldots, m - 1$, where

$$
T = P^{-1}_\pi \cdot \max_{0 \leq i < m} \{ T_i \}, \quad S_0 := \left(\frac{k}{2}\right),
$$

$$
S_i = \binom{k}{p_i} \cdot \frac{\ell^{(i)}_{\pi}}{2^n} \cdot \prod_{h=1}^{i-1} \frac{w^{(h)}_{\pi} \cdot \binom{\ell^{(i)}_{\pi}}{w^{(h)}_{\pi}} \cdot \binom{\ell^{(i)}_{\pi} - w^{(h)}_{\pi}}{w^{(i)}_{\pi}/2}}{w^{(h)}_{\pi}/2} \cdot i = 1, \ldots, m - 1
$$

with $T_i := 2^{\left(n \log(\lambda_i, \delta_{i+1}, \gamma_{i+1}) \right)}$

and $\lambda_i := \frac{\log(S_i)}{n}$, $\delta_{i+1} := i_{i+1}/p_i$, $\gamma_{i+1} := \frac{w^{(i+1)}}{n}$ for $i = 0, \ldots, m - 1$.

Proof. Let us first consider Alg. 5.3. The lists on level 0 have size $S_0$ and are created in time $S_0$. On every layer $i = 1, \ldots, m - 1$ the projection $(\hat{P}e^{(i)}_{\pi})[i]$ with weight $w^{(i)}_{\pi}$ is the sum of two uniform vectors since $\hat{P}$ is uniform. In contrast to this, the projection $(\hat{P}e^{(i)}_{\pi})[h]$ with weight $w^{(i)}_{\pi}$ is the sum of two uniform vectors of specific weight $w^{(i-1)}_{\pi}$ for every $h = 1, \ldots, i - 1$. If we fix the first vector, there are $\binom{\ell^{(i-1)}_{\pi}}{w^{(i-1)}_{\pi}}$ possible second vectors out of which $\binom{w^{(i-1)}_{\pi} - w^{(i-1)}_{\pi}}{w^{(i-1)}_{\pi}/2}$ yield the correct weight $w^{(i)}_{h, \pi}$. Therefore, the expected list size on layer $i = 1, \ldots, m - 1$ is

$$
\mathbb{E}\left[\left|L^{(i)}_{j}\right|\right] = \|\{x \in \mathbb{F}_2^n : \Delta(x) = p_i\}| \cdot \mathbb{P}_x[\Delta(x) = w^{(i)}_{\pi}] \cdot \prod_{h=1}^{i-1} \mathbb{P}_{x+y \in \mathbb{F}_2^h}[\Delta(x+y) = w^{(i)}_{\pi} | \Delta(x) = w^{(i-1)}_{\pi}] = \binom{k}{p_i} \cdot \frac{\ell^{(i)}_{\pi}}{2^n} \cdot \prod_{h=1}^{i-1} \frac{w^{(h)}_{\pi} \cdot \binom{\ell^{(i)}_{\pi}}{w^{(h)}_{\pi}} \cdot \binom{\ell^{(i)}_{\pi} - w^{(h)}_{\pi}}{w^{(i)}_{\pi}/2}}{w^{(h)}_{\pi}/2} = S_i \text{ for } j = 1, \ldots, 2^{m-i}.
$$

The lists are recursively constructed via the NN-SEARCH-algorithm and subsequent filtering. By Lemma 1.4 the nearest neighbor searches take time $T_i$ on layer $i = 0, \ldots, m - 1$. The output lists sizes for the nearest neighbor searches are upper bounded by $T_i$ which is therefore an upper bound for the running time of the filtering steps on each level. Since we also have $S_0 \leq T_0$, Alg. 5.3 runs in time $\max_{0 \leq i < m} \{ T_i \}$. For the total running time we have to take account of the outer loop in Alg. 4.1. A good permutation is found with probability $P_\pi$. Therefore we can expect that the algorithm terminates after $P^{-1}_\pi$ iterations and thus after time $T$.

Proof of Theorem 5.1. We use Lemma 1.1 to upper bound the $\lambda_i$ and the binomial coefficients. Numerical optimization yields the following parameters and the claimed
complexities. For full distance decoding the worst-case running time is achieved a rate of
\[ \frac{k}{n} = 0.46 \]
with relative distance
\[ \frac{w}{n} = \frac{d}{n} = H^{-1} \left( 1 - \frac{k}{n} \right) = 0.1237. \]
For this code rate the running time is minimized for the relative parameters
\[ \frac{p_1}{n} = 0.00559, \quad \frac{p_2}{n} = 0.01073, \quad \frac{p_3}{n} = 0.02029, \quad \frac{p_4}{n} = 0.03460, \]
resulting in \( R_2 = 2^{0.01357n} \), \( R_3 = 2^{0.02668n} \) and \( R_4 = 2^{0.06028n} \) representations. We furthermore choose
\[ \ell_1 = 0.0366, \quad \ell_2 = 0.0547, \quad \ell_3 = 0.0911, \]
\[ \frac{w_1}{n} = 0.0066, \quad \frac{w_2}{n} = 0.0099, \quad \frac{w_3}{n} = 0.0114, \quad \frac{w_1(3)}{n} = 0.0232. \]
Optimization showed that
\[ w_1 = \frac{w_1(2)}{2}, \quad w_2 = \frac{w_2(3)}{2} \]
is a good choice which yields
\[ \frac{w_1(1)}{n} = 0.011515, \quad \frac{w_1(2)}{n} = 0.023029, \]
\[ \frac{w_2(2)}{n} = 0.016676, \quad \frac{w_2(3)}{n} = 0.033351, \quad \frac{w_3(3)}{n} = 0.009993 \]
using condition Eq.(5.6) from Lemma 5.3. The resulting list sizes are
\[ S_0 = 2^{0.02179n}, \quad S_1 = 2^{0.03987n}, \quad S_2 = 2^{0.05987n}, \quad S_3 = 2^{0.05975n}. \]
The lists on layer 0 are combined with the nearest neighbor search of Alg. 1.3 in time
\[ T_0 = 2^{0.04359n}, \]
as the condition for May-Ozerov is not satisfied and Alg. 1.4 is less efficient in this case. On layer 1 we use Alg. 1.4 in time
\[ T_1 = 2^{0.07356n}, \]
which is also the space consumption for this step since we blow up the lists in this meet-in-the-middle-algorithm. On the remaining layers, we use May-Ozerov nearest
neighbor search which yields

\[ T_2 = 2^{0.07365n}, \quad T_3 = 2^{0.07359n}. \]

The probability for the correct weight distribution satisfying Def. 5.2 is \( P_\pi = 2^{-0.01485n} \).

This yields the claimed running time and memory consumption.

The complexity for half distance decoding can be shown analogously for a code rate \( \frac{k}{n} = 0.47 \) with relative distance \( \frac{w}{n} = d/n = H^{-1} \left( 1 - \frac{k}{n} \right) = 0.06011 \)

using the parameters

\[
\begin{align*}
\frac{p_1}{n} &= 0.002038, \quad \frac{p_2}{n} = 0.003855, \quad \frac{p_3}{n} = 0.007490, \quad \frac{p_4}{n} = 0.012200, \\
\frac{\ell_1}{n} &= 0.0125, \quad \frac{\ell_2}{n} = 0.0204, \quad \frac{\ell_3}{n} = 0.0350, \\
\frac{w_1}{n} &= 0.0012, \quad \frac{w_2}{n} = 0.0019, \quad \frac{w_3}{n} = 0.0019, \\
\frac{w_1^{(1)}}{n} &= 0.003581, \quad \frac{w_1^{(2)}}{n} = 0.007161, \quad \frac{w_1^{(3)}}{n} = 0.0062, \\
\frac{w_2^{(2)}}{n} &= 0.005906, \quad \frac{w_2^{(3)}}{n} = 0.011812, \quad \frac{w_3^{(3)}}{n} = 0.002200.
\end{align*}
\]
5.3 Results and Comparison

Before we compare the different decoding algorithms, let us first give parameter sets for our algorithm \textsc{Depth-4-ISDSolve} for both full distance and half distance decoding (see Figure 5.6). However, since our analysis is purely asymptotical, we advise the reader to choose parameters in dependence of the implementation in practice and use the given sets as a starting point only. Recall the proof of Theorem 5.1 to obtain the parameters \( w_h^{(i)} \) for \( h = 1, \ldots, i, i = 1, 2, 3 \).

![Fig. 5.6: Parameter sets for \textsc{Depth-4-ISDSolve} for minimal running time and varying code rates in the full distance (top) and half distance (bottom) decoding setting.](image)

\begin{table}[h]
  \centering
  \begin{tabular}{c|cccccccccc}
    k & 0.1 & 0.2 & 0.3 & 0.4 & 0.46 & 0.5 & 0.6 & 0.7 & 0.8 & 0.9 \\
    \hline
    \( p_1 \) & 0.0044 & 0.0063 & 0.0056 & 0.0057 & 0.0056 & 0.0055 & 0.0049 & 0.0041 & 0.0029 & 0.0015 \\
    \( p_2 \) & 0.0076 & 0.0114 & 0.0107 & 0.0110 & 0.0107 & 0.0105 & 0.0094 & 0.0077 & 0.0055 & 0.0029 \\
    \( p_3 \) & 0.0117 & 0.0172 & 0.0197 & 0.0205 & 0.0203 & 0.0200 & 0.0182 & 0.0148 & 0.0108 & 0.0058 \\
    \( p_4 \) & 0.0212 & 0.0310 & 0.0345 & 0.0354 & 0.0346 & 0.0338 & 0.0302 & 0.0243 & 0.0173 & 0.0090 \\
    \( q_1 \) & 0.0275 & 0.0375 & 0.0386 & 0.0387 & 0.0366 & 0.0356 & 0.0299 & 0.0250 & 0.0171 & 0.0085 \\
    \( q_2 \) & 0.0559 & 0.0809 & 0.0503 & 0.0543 & 0.0547 & 0.0546 & 0.0515 & 0.0425 & 0.0283 & 0.0135 \\
    \( q_3 \) & 0.0556 & 0.0770 & 0.0918 & 0.0933 & 0.0911 & 0.0899 & 0.0796 & 0.0667 & 0.0508 & 0.0297 \\
    \( q_4 \) & 0.0092 & 0.0104 & 0.0088 & 0.0076 & 0.0066 & 0.0060 & 0.0043 & 0.0029 & 0.0015 & 0.0005 \\
    \( \ell_{1}^{(3)} \) & 0.0186 & 0.0224 & 0.0117 & 0.0109 & 0.0099 & 0.0091 & 0.0073 & 0.0049 & 0.0024 & 0.0007 \\
    \( \ell_{2}^{(3)} \) & 0.0149 & 0.0158 & 0.0163 & 0.0135 & 0.0114 & 0.0104 & 0.0069 & 0.0045 & 0.0024 & 0.0008 \\
    \( \ell_{3}^{(3)} \) & 0.0135 & 0.0187 & 0.0257 & 0.0251 & 0.0232 & 0.0222 & 0.0168 & 0.0127 & 0.0086 & 0.0043 \\
  \end{tabular}
\end{table}

\begin{table}[h]
  \centering
  \begin{tabular}{c|cccccccccc}
    k & 0.1 & 0.2 & 0.3 & 0.4 & 0.46 & 0.5 & 0.6 & 0.7 & 0.8 & 0.9 \\
    \hline
    \( p_1 \) & 0.0012 & 0.0017 & 0.0020 & 0.0021 & 0.0020 & 0.0020 & 0.0019 & 0.0016 & 0.0012 & 0.0007 \\
    \( p_2 \) & 0.0017 & 0.0032 & 0.0038 & 0.0039 & 0.0039 & 0.0039 & 0.0036 & 0.0029 & 0.0022 & 0.0013 \\
    \( p_3 \) & 0.0003 & 0.0063 & 0.0073 & 0.0075 & 0.0075 & 0.0070 & 0.0056 & 0.0044 & 0.0025 \\
    \( p_4 \) & 0.0061 & 0.0105 & 0.0120 & 0.0124 & 0.0122 & 0.0121 & 0.0111 & 0.0090 & 0.0068 & 0.0037 \\
    \( q_1 \) & 0.0125 & 0.0109 & 0.0122 & 0.0125 & 0.0125 & 0.0121 & 0.0108 & 0.0103 & 0.0068 & 0.0037 \\
    \( q_2 \) & 0.0090 & 0.0176 & 0.0208 & 0.0209 & 0.0204 & 0.0200 & 0.0176 & 0.0149 & 0.0102 & 0.0055 \\
    \( q_3 \) & 0.1249 & 0.0285 & 0.0324 & 0.0344 & 0.0350 & 0.0350 & 0.0333 & 0.0299 & 0.0231 & 0.0140 \\
    \( q_4 \) & 0.0021 & 0.0015 & 0.0015 & 0.0013 & 0.0012 & 0.0011 & 0.0008 & 0.0007 & 0.0003 & 0.0001 \\
    \( \ell_{1}^{(3)} \) & 0.0015 & 0.0025 & 0.0025 & 0.0022 & 0.0019 & 0.0018 & 0.0013 & 0.0009 & 0.0005 & 0.0002 \\
    \( \ell_{2}^{(3)} \) & 0.0208 & 0.0026 & 0.0024 & 0.0021 & 0.0019 & 0.0018 & 0.0014 & 0.0011 & 0.0006 & 0.0002 \\
    \( \ell_{3}^{(3)} \) & 0.0065 & 0.0060 & 0.0062 & 0.0063 & 0.0062 & 0.0061 & 0.0055 & 0.0055 & 0.0034 & 0.0018 \\
  \end{tabular}
\end{table}
BJMM-NN vs. Our Algorithm. Let us compare the worst case complexities of our algorithm to the BJMM with NN-SEARCH (see Fig. 5.7). Our algorithm is superior to BJMM-NN for all depths $m = 2, 3, 4$ in the full distance (FD) decoding setting. For $m = 3$ we already beat the optimal BJMM-NN(4) variant. Another benefit is the moderate memory consumption of our algorithm which is clearly smaller than the running time. Note that we reduce the memory consumption for $m = 3$ compared to $m = 2$. However, it increases again for $m = 4$ since we have to use the meet-in-the-middle nearest neighbor search (Alg. 1.4) which costs extra memory. In the half distance (HD) decoding setting, we outperform the BJMM-NN for $m = 4$. Unfortunately this improvement is not as significant as in the FD setting. This is due to a strong dependency on the error-weight i.e. our algorithm behaves best for high errors. We suspect this is due to the use of the NN-SEARCH-algorithm on every layer, which needs a sufficiently large weight to show its strength. In contrast to BJMM-NN, the choice $m = 4$ is not necessarily optimal. Unfortunately we were not able to run optimizations for $m > 4$ since the parameter space is too large to run meaningful optimizations.

<table>
<thead>
<tr>
<th>$m$</th>
<th>BJMM-NN($m$)</th>
<th>Depth-$m$-ISDSolve</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\log(T)/n$</td>
<td>$\log(M)/n$</td>
</tr>
<tr>
<td>2</td>
<td>0.1003</td>
<td>0.0781</td>
</tr>
<tr>
<td>3</td>
<td>0.0967</td>
<td>0.0879</td>
</tr>
<tr>
<td>4</td>
<td>0.0953</td>
<td>0.0915</td>
</tr>
<tr>
<td>2</td>
<td>0.0491</td>
<td>0.0309</td>
</tr>
<tr>
<td>3</td>
<td>0.0473</td>
<td>0.0363</td>
</tr>
<tr>
<td>4</td>
<td>0.0473</td>
<td>0.0351</td>
</tr>
</tbody>
</table>

Fig. 5.7: Running time and memory consumption for different $m$.

Comparison of Running Times. In this thesis we presented several ISD algorithm. We started with Prange’s [Pra62] and Stern’s [Ste88] algorithm as well as ball collision decoding (BCD, Alg. 4.3) by Bernstein et al. [BLP11] in the SimpleISD-framework. We then introduced the AdvancedISD-framework and the algorithm by Finiasz and Sendrier [FS09] and continued with the BJMM [BJMM12] (Alg. 4.6) and its variant BJMM-NN [MO15, BM17b] with nearest neighbors (Alg. 5.1). We concluded with the description of our new algorithm Depth-$m$-ISDSolve (Alg. 5.3). Fig. 5.8 illustrates the running times as a function of a constant code rate $0 \leq \frac{k}{n} \leq 1$. We omitted Stern’s and the Finiasz-Sendrier-algorithm since it provides similar running times to ball collision decoding. The graphs have clear maxima around $\frac{k}{n} = 0.45$ which supports the meaningfulness of a worst-case analysis in this setting. Our algorithm provides the
best running times over the whole spectrum of code rates.

In contrast to this we provide a comparison of the worst case complexities in Figure 5.9. We added the MMT [MMT11] for completeness.

<table>
<thead>
<tr>
<th></th>
<th>$k/n$</th>
<th>$\log(T)/n$</th>
<th>$\log(M)/n$</th>
<th>$k$</th>
<th>$\log(T)/n$</th>
<th>$\log(M)/n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prange</td>
<td>0.455</td>
<td>0.1207</td>
<td>-</td>
<td>0.468</td>
<td>0.05752</td>
<td>-</td>
</tr>
<tr>
<td>Stern</td>
<td>0.445</td>
<td>0.1166</td>
<td>0.0312</td>
<td>0.467</td>
<td>0.05564</td>
<td>0.01308</td>
</tr>
<tr>
<td>BCD</td>
<td>0.446</td>
<td>0.1163</td>
<td>0.0355</td>
<td>0.465</td>
<td>0.05558</td>
<td>0.01390</td>
</tr>
<tr>
<td>MMT</td>
<td>0.45</td>
<td>0.1116</td>
<td>0.0541</td>
<td>0.47</td>
<td>0.05364</td>
<td>0.02160</td>
</tr>
<tr>
<td>BJMM(3)</td>
<td>0.426</td>
<td>0.1020</td>
<td>0.0728</td>
<td>0.448</td>
<td>0.04949</td>
<td>0.02617</td>
</tr>
<tr>
<td>BJMM-NN(4)</td>
<td>0.424</td>
<td>0.0953</td>
<td>0.0915</td>
<td>0.475</td>
<td>0.04730</td>
<td>0.03510</td>
</tr>
<tr>
<td>DEPTH-4-ISDSOLVE</td>
<td>0.46</td>
<td>0.0885</td>
<td>0.0738</td>
<td>0.47</td>
<td>0.0465</td>
<td>0.02940</td>
</tr>
</tbody>
</table>

Fig. 5.9: Worst case code rate, running time and memory consumption for different algorithms.

**Time-Memory Trade-offs.** Until now we only cared for minimal running time without considering low memory consumption. Figure 5.10 illustrates the behavior of the algorithm’s running times, if we restrict the memory to $M \leq 2^{\lambda n}$ for some $\lambda > 0$. For BJMM, BJMM-NN and our algorithm we analyzed the running time over
the whole spectrum from no memory restriction (right) to polynomial memory (left, \( \lambda \to 0 \)). Decreasing the available memory does not lead to a significantly increased running time at first. However, for small available memory the running times increase faster eventually converging to the run time of Prange’s algorithm for polynomial memory consumption. In contrast to this all algorithms have some maximum memory consumptions where a further increment does not reduce the running time anymore. Again, our algorithm with \( m = 3 \) or \( m = 4 \) is superior for all restrictions of memory while for \( m = 2 \) the curve is very similar to BJMM-NN(3).

![Figure 5.10: Running time depending on different memory restrictions 2^{\lambda n} in the full distance decoding setting.](image)

Finally, Figure 5.11 provides a closer look at running times for specific choices of memory restrictions. Running times are bold, when they are minimal already. Further increasing the memory does not have any effect in this case which is marked with a hyphen (-). Using \( 2^{0.0318n} \) memory like Stern’s algorithm, our algorithm has a running time of \( 2^{0.0969n} \). This is similar to the running time \( 2^{0.0967n} \) of BJMM-NN(3) which in contrast uses significantly more memory \( (2^{0.0879n}) \). Moreover for \( M = 2^{0.0355n} \) (memory of ball collision decoding) our algorithm provides the same running time \( 2^{0.0953n} \) as the BJMM-NN(4)-algorithm which uses \( 2^{0.0915n} \) memory.
<table>
<thead>
<tr>
<th>$\lambda$</th>
<th>0.0915</th>
<th>0.0879</th>
<th>0.0781</th>
<th>0.0729</th>
<th>0.0691</th>
<th>0.0541</th>
<th>0.0355</th>
<th>0.0318</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stern</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.1167</td>
<td></td>
</tr>
<tr>
<td>BCD</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.1163</td>
<td></td>
</tr>
<tr>
<td>MMT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.1116</td>
<td></td>
</tr>
<tr>
<td>BJMM(2)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.1054</td>
<td></td>
</tr>
<tr>
<td>BJMM(3)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.1020</td>
<td>0.1023</td>
<td>0.1043</td>
<td>0.1088</td>
</tr>
<tr>
<td>BJMM-NN(2)</td>
<td>-</td>
<td>-</td>
<td>0.1003</td>
<td>0.1008</td>
<td>0.1010</td>
<td>0.1030</td>
<td>0.1067</td>
<td>0.1077</td>
</tr>
<tr>
<td>BJMM-NN(3)</td>
<td>-</td>
<td>0.0967</td>
<td>0.0974</td>
<td>0.0978</td>
<td>0.0982</td>
<td>0.1003</td>
<td>0.1060</td>
<td>0.1073</td>
</tr>
<tr>
<td>BJMM-NN(4)</td>
<td>0.0953</td>
<td>0.0957</td>
<td>0.0960</td>
<td>0.0965</td>
<td>0.0968</td>
<td>0.0999</td>
<td>0.1060</td>
<td>0.1072</td>
</tr>
<tr>
<td>DEPTH-2-ISDSOLVE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.0983</td>
<td>0.1004</td>
<td>0.1063</td>
</tr>
<tr>
<td>DEPTH-3-ISDSOLVE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.0938</td>
<td>0.0997</td>
<td>0.1013</td>
</tr>
<tr>
<td>DEPTH-4-ISDSOLVE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.0886</td>
<td>0.0887</td>
<td>0.0904</td>
</tr>
</tbody>
</table>

Fig. 5.11: Running time exponents $\log(T)/n$ for different memory restrictions in the full distance decoding setting.

### 5.4 Application: The McEliece Cryptosystem

The public-key cryptosystem by McEliece [McE78] has been introduced in 1978 and is based on coding theory. There have been numerous attempts to break this system, for example by Stern [Ste88], Canteaut et al. [CC98, CS98] and Bernstein et al. [BLP08a]. Nevertheless it has shown remarkable strength. This also holds for known quantum attacks. In addition it comes with efficient algorithms for encryption and decryption running in time $O(b^2)$ for parameters which provide bit security $b$ [BLP11]. In contrast to this, the approved cryptosystem RSA comes with encryption in time $O(b^3)$ only and does not stand a chance against quantum computers. Because of its coding background, ISD algorithms are the main threat of the McEliece cryptosystem.

**The McEliece Cryptosystem.** Let $G \in \mathbb{F}_2^{k \times n}$ be a generator matrix for some linear code $C$ and $w \in [n]$ be some Hamming weight. We define the *encryption function* $\text{Enc}_G(x) : \mathbb{F}_2^k \rightarrow \mathbb{F}_2^n$ as

$$\text{Enc}_G(x) \leftarrow xG + e$$

for some uniform $e \in \mathbb{F}_2^n$ with $\Delta(e) = w$.

Recall that $c := mG + e$ is a codeword. The *decryption function* $\text{Dec}_{sk}(y) : \mathbb{F}_2^n \rightarrow \mathbb{F}_2^n$ for a ciphertext $y := \text{Enc}_G(x)$ using some secret key $sk$ is not relevant for our attack. The parameters of the cryptosystem are $n, k$ and $w$. Without loss of generality we assume that $G$ is given in systematic form $G = (I_k \mid \tilde{P}^T)$ and thus the parity check matrix of the linear code is $P = (\tilde{P} \mid I_{n-k})$. Hence, $(P, s, w)$ with $s := Py = Pe$ is an instance of the decoding problem defined by the parameters of the given McEliece cryptosystem. Solving this instance and therefore finding $e$ is equivalent to decrypting the ciphertext.
Attacking McEliece. Bernstein, Lange and Peters [BLP08b] suggest the following set of parameters for instantiations of McEliece.

<table>
<thead>
<tr>
<th>$b$</th>
<th>$n$</th>
<th>$k$</th>
<th>$w$</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>1632</td>
<td>1269</td>
<td>34</td>
</tr>
<tr>
<td>128</td>
<td>2960</td>
<td>2288</td>
<td>57</td>
</tr>
<tr>
<td>256</td>
<td>6624</td>
<td>5129</td>
<td>117</td>
</tr>
</tbody>
</table>

Fig. 5.12: McEliece parameters for different security levels $b$ as suggested in [BLP08b].

Thus it is reasonable to choose $\frac{k}{n} = 0.775$, $\frac{w}{n} = 0.02$ in order to estimate the complexity of ISD algorithms for these instances. Figure 5.13 provides the complexities for BJMM, BJMM-NN and our algorithm with different numbers of layers $m$.

<table>
<thead>
<tr>
<th>$m$</th>
<th>$\log(T)/n$</th>
<th>$\log(M)/n$</th>
<th>$m$</th>
<th>$\log(T)/n$</th>
<th>$\log(M)/n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.0370</td>
<td>0.0255</td>
<td>2</td>
<td>0.0362</td>
<td>0.0264</td>
</tr>
<tr>
<td>3</td>
<td>0.0362</td>
<td>0.0262</td>
<td>3</td>
<td>0.0350</td>
<td>0.0280</td>
</tr>
<tr>
<td>4</td>
<td>0.0362</td>
<td>0.0271</td>
<td>4</td>
<td>0.0350</td>
<td>0.0280</td>
</tr>
</tbody>
</table>

Fig. 5.13: Upper bounds for running time and memory exponents of BJMM($m$) (left), BJMM-NN($m$) (right) and Depth-$m$-ISDSolve (below) for $\frac{k}{n} = 0.775$, $\frac{w}{n} = 0.02$.

Plugging the different values of $n$ given by Fig. 5.12 into $\log(T) = 0.0362n$ for BJMM($m$), $\log(T) = 0.0350n$ for BJMM-NN($m$) and $\log(T) = 0.0347n$ for Depth-$m$-ISDSolve this yields

<table>
<thead>
<tr>
<th>$b$</th>
<th>BJMM(3)</th>
<th>BJMM-NN(4)</th>
<th>Depth-4-ISDSolve</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>59</td>
<td>57</td>
<td>57</td>
</tr>
<tr>
<td>128</td>
<td>107</td>
<td>104</td>
<td>103</td>
</tr>
<tr>
<td>256</td>
<td>240</td>
<td>232</td>
<td>230</td>
</tr>
</tbody>
</table>

Fig. 5.14: Running time exponents $\log(T)$ aiming for different security levels $b$.

Since our asymptotic analysis neglects all polynomial factors, these values are clearly not the bit security levels of the suggested McEliece instantiations in practice. However, they indicate that the proposed McEliece instances can be attacked with less effort than predicted ten years ago.
5.5 Application: Learning Parity with Noise

In this section we present another application for our algorithm besides the syndrome decoding problem itself. Every instance of the so called learning parity with noise (LPN) problem is naturally an instance of the syndrome decoding problem allowing us to apply our ISD algorithms directly. However, Esser, Kübler and May showed in [EKM17] that the best approach to solve large LPN instances in practice is by a hybrid approach. In a nutshell, one first use some dimension reduction algorithm, e.g. the BKW-algorithm [BKW00] followed by some decoding algorithm. This approach is slower that using BKW only but uses much less memory which makes it far more practical. The dimension reduction in the first step comes at a cost of a large error rate close to \( \frac{1}{2} \) for decoding. As mentioned before, our algorithm Depth-\( m \)-ISDSolve behaves best when attacking decoding instances with high error rate. Thus it is well suited for this hybrid setting. Let us first give a definition of the learning parity with noise problem.

**Definition 5.3 (LPN Problem).** Let \( \tau \in [0, \frac{1}{2}) \) be some error parameter, and let \( s \in \mathbb{F}_2^k \) be a secret vector. Given oracle access to samples of the form \((a_i, b_i) := (a_i, (a_i \cdot s) + e_i), \) for \( i = 1, 2, \ldots \)

where \( a_i \in \mathbb{R} \mathbb{F}_2^k \) and \( e_i \in \{0, 1\} \) with \( \Pr[e_i = 1] = \tau \), one has to find \( s \). We denote an instance of the LPN problem by \( \text{LPN}_{k, \tau} \).

Let \( n \) be the number of samples, which can be freely chosen. We can write an LPN instance as a matrix-vector tuple \((A, b) \in \mathbb{F}_2^n \times \mathbb{F}_2^k \) satisfying \( As = b + e \),

where \( e = (e_1, \ldots, e_n) \) and the \( i^{th} \) row of \( A \) and \( b \) represent the \( i^{th} \) LPN sample. By definition of LPN, \( A \) is the generator matrix of a random binary linear code with dimension \( k \) and length \( n \). Since we can choose \( n \) ourselves freely, we can make the rate \( \frac{k}{n} \) arbitrarily small. Furthermore \( b = As + e \) is a noisy codeword that can be decoded to \( b + e = As \) if the error vector \( e \in \mathbb{F}_2^n \) of expected weight \( \mathbb{E}(|\Delta(e)|) = \tau n =: w \) can be recovered. Typical LPN instances in cryptography come with \( \tau = \frac{1}{4} \) or \( \tau = \frac{1}{8} \).

**Straight Attack on LPN.** Let us first apply our algorithm Depth-\( m \)-ISDSolve directly to the \( \text{LPN}_{k, \tau} \) problem, i.e. we solve an instance \((P, Pb, w) \) of the syndrome decoding problem where \( P \in \mathbb{F}_2^{(n-k) \times n} \), \( b \in \mathbb{F}_2^n \) and \( w := \tau n \). We are free to choose the number of samples \( n \) and therefore the code rate \( \frac{k}{n} \) despite the fact that \( k \) is fixed.
contrast to the decoding setting, we now minimize the running time $T(n, k, \tau)$ of our decoding algorithm over all $n$. Figure 5.15 provides an overview how the number of samples $n$ influences our algorithm compared to Prange’s algorithm and the BJMM-NN. Increasing the amount of samples decreases the running time rapidly until some threshold value is reached. For Prange the running time converges against this value while it slowly increases again for BJMM-NN and our algorithm. The running time for Depth-$m$-ISDSOLVE reaches its minimum for an optimal number of samples around 140,000 for the cryptographically popular LPN$_{512, \frac{1}{4}}$-instances.

![Fig. 5.15: Comparison of Prange’s algorithm, BJMM-NN(4) and our algorithm with depth 4 for different numbers of samples for $k = 512, \tau = \frac{1}{4}$.](image)

We compare the minimum running times in Fig. 5.16. Notice that we suppress polynomial overheads for the running times but not for the number of samples $n$. We would like to point out again that decoding is not the best way to attack LPN instances. It is preferable to combine a variant of the BKW-algorithm like [GJL14] and a decoding algorithm as shown by Esser, Kübler and May [EKM17]. Since one has to solve instances of the syndrome decoding problem with high error rate $\frac{m}{n} = \tau$, our algorithm seems like a perfect choice. Optimizations showed that the typical instance LPN$_{512, \frac{1}{4}}$ is turned into LPN$_{117, \frac{255}{512}}$ instances by the BKW first and then subsequently solved via decoding. Figure 5.16 already provides us a good evidence that our algorithm can solves these instances much faster that previous algorithms like the BJMM-NN.
The asymptotic behavior of our algorithm on LPN-instances is shown in Fig. 5.17 for varying weights $\tau$ and compared to Prange’s algorithm as well as the BJMM-NM. This again illustrates the strength of our algorithm for high errors. The graph of our new algorithm’s complexity can be very well approximated by a line yielding the simple formula

$$T_{\text{LPN}}(k, \tau) = 2^{1.28k\tau}.$$  \hfill (5.8)

**Using Hybrid Techniques.** Let us take a closer look now how our algorithm behaves in the hybrid framework introduced in [EKM17] which is due to its low memory
requirements currently the best way to attack LPN instances in practice. The analysis is done analogously to [Kü18]. Starting with a LPN$_{k,\tau}$ instance, the first step is a naive dimension reduction where one only keeps samples with zero entries on $\alpha k$ fixed coordinates. This yields an instance LPN$_{(1-\alpha)k,\tau}$ of reduced dimension in time $T_1 := 2^{\alpha k}$

since we need $2^{\alpha k}$ queries in expectation to the oracle of LPN$_{k,\tau}$ in order to obtain one sample for LPN$_{(1-\alpha)k,\tau}$. In the second step the BKW is applied on $\beta k$ coordinates in time $T_2 := 2^{\beta k}$

yielding an instance LPN$_{(1-\alpha-\beta)k,\tau'}$ with $\tau' \approx \frac{1}{2}$ [EKM17]. This instance is solved using decoding in time $T_3$ dependent on the choice of algorithm. Using a naive decoding algorithm with running time $2^{-(1-\alpha-\beta)\log(1-\tau')k} \approx 2^{(1-\alpha-\beta)k}$ like in [EKM17] yields a minimal running time

$$T_{Naive} := \max\{T_1, T_2, T_3\} = 2^{\max\{\frac{1-\beta}{2}, \frac{\beta}{\log(1-\tau')}, \tau(\lambda)\}k}$$

for the choice $\alpha = \frac{1-\beta}{2}$ and optimal $\beta$. The memory consumption is determined by the BKW step, i.e. $M = T_2$. However, the hybrid framework really plays to its strengths in memory restricted environments where the memory consumption is bounded by $2^{\lambda k}$ for some $\lambda > 0$. Thus we have to choose $\beta$ such that $T_2 \leq 2^{\lambda k}$.

If we solve a LPN$_{k,\tau}$ instance using our algorithm with memory at most $2^{\lambda k}$, the running time (see Figure 5.18) can be well approximated by

$$2^{\tau(\lambda)k} \text{ with } \tau(\lambda) = \begin{cases} 7.39\lambda^3 - 3.91\lambda^2 - 0.53\lambda + 1.00 & \lambda \leq 0.402 \\ 0.635 & \text{else} \end{cases}$$

Thus, using Depth-$m$-ISDSolve in step 3 of the hybrid framework, the overall running time is

$$T_{Ours} := \max\{T_1, T_2, T_3\} = 2^{\max\{\frac{1-\beta}{2}, \frac{\beta}{\log(1-\tau')}, \tau(\lambda)\}(1-\alpha-\beta)k}$$

assuming we choose $\beta$ such that $T_2 \leq 2^{\lambda k}$. Finally, Figure 5.19 illustrates the running times $T_{Naive}$ and $T_{Ours}$ depending on $\lambda$ with optimized parameter $\alpha, \beta$. Our algorithm slightly reduces the running time in memory restricted environments. The improvement however vanishes if $\lambda$ is sufficiently large and does not restrict the decoding step or if $\lambda \to 0$, i.e. for polynomial memory.
Fig. 5.18: Running time of DEPTH-4-ISDSOLVE solving a LPN$_{k, \frac{233}{512}}$ instance with $M \leq 2^{\lambda k}$ memory.

Fig. 5.19: Running time of the hybrid-algorithm with $M \leq 2^{\lambda k}$ memory using different subroutines for the decoding in step 3.
5.6 Open Problems

Let us conclude this chapter taking a look on open problems. Since both BJMM-NN and our decoding algorithm Depth-$m$-ISDSolve are similar to our Approx-$k$-Tree-algorithm in Chapter 3, there are similar problems to discuss. Analogously do the previous chapter we used the nearest neighbor search by May and Ozerov whenever possible. An alternative nearest neighbor search or a way to get rid of the restriction $\lambda < 1 - H(\frac{4}{7})$ would improve our results immediately again. Moreover we rely on expected values in our analysis again and did not estimate failure probabilities.

Polynomial Factors. In the original work [EKM17] about the hybrid framework the authors only considered simpler decoding algorithms despite state-of-the-art solutions like the BJMM. This is due to the huge polynomial factors which add up to the complexity of the algorithms. The same holds for the May-Ozerov nearest neighbor search which is used in BJMM-NN and our algorithm Depth-$m$-ISDSolve. Thus it remains unclear how the different algorithms behave in practice. We leave this as an open problem. Some proper implementations as well as experimental results should be able to answer this question. Nevertheless our asymptotic analysis already provides good estimates.

Parameter Overload. One drawback of state-of-the-art decoding algorithms is their increasing number of parameters subject to optimization. Unfortunately this especially holds for our new algorithm. Its sheer number of parameters prevented us from finding optimal parameters for $m > 4$ perhaps yielding even lower running times. Therefore minimizing the running time for $m > 4$ as well as reducing the number of parameters is an open problem. Nonetheless it could be more preferable to develop new decoding algorithms from scratch. For example the parameters of any new algorithm could be found more intuitively if it has no outer loop with its dependencies on the parameters used in the solving algorithm.
Bibliography


