Noname manuscript No.
(will be inserted by the editor)

Graphs and Self-dual additive codes over $GF(4)$

Mithilesh Kumar · Håvard Raddum · Srimathi Varadharajan

the date of receipt and acceptance should be inserted later

Abstract We initiate the study of self-dual codes over $GF(4)$ whose corresponding graphs have fixed rankwidth. We show that by combining the structural properties of rankwidth 1 graphs, the classification of corresponding codes becomes significantly faster.

We give a new algorithm for computing weight enumerators using Binary Decision Diagrams (BDD), which has similar complexity to brute force $\mathcal{O}(2^k)$ but has the benefit that we automatically get complexity $\mathcal{O}(2^{\min\{k,n-k\}})$ (for $k > n/2$) without needing to consider the dual code.

We show that the minimum distance of a code is at least 3 if and only if the corresponding graph does not contain any pendant vertex or any twin-pairs. We also give an algorithm for computing an approximate minimum distance in codes corresponding to general graphs.

Keywords Stabilizer Code · Self-dual · Rankwidth · Binary Decision Diagram · Minimum Distance

Mathematics Subject Classification (2010) 94B25

1 Introduction

The implications of quantum computation and information in coding theory and cryptography needs to be explored more extensively. While the classical error correction is well understood with decades of research, the quantum error correction needs more time and effort. In this regard, stabilizer codes are used for quantum error correction [14]. The corresponding quantum states (called stabilizer states) can be represented using graphs (called graph states) [16]. Graph states can be used as a resource for measurement-based quantum computation [21]. These quantum stabilizer codes can be represented as self-dual additive codes over $GF(4)$ [5]. Furthermore, there is a one-to-one correspondence between self-dual additive codes over $GF(4)$ and simple undirected graphs [25,18,4]. Certain local operations on graphs preserve the equivalence of self-dual additive codes over $GF(4)$. There is

Simula UiB, Thormøhlensgate 55, 5008 Bergen, Norway

a long list of papers [12,1,7,6,8,9] that tried to classify such codes and in this paper, we continue on this path.

Trying to classify codes by considering all possible graphs makes the problem extremely hard. This is evident since self-dual additive codes over $GF(4)$ has been classified only for n up to 12 [7]. In this paper, we initiate the study of self-dual codes whose corresponding graphs have fixed rankwidth. The graph parameter rankwidth is preserved under local complementation. For rankwidth 1, the class of graphs is exactly the distance-hereditary graphs. We show that by combining the structural properties of these graphs with the algorithm used in [13,7], the classification of corresponding codes becomes significantly faster.

There are two computationally heavy steps in the above algorithm: graph isomorphism and weight enumeration. For a fixed k, testing graph isomorphism for graphs of rankwidth k is polynomial in the size of the graph [15], and it is in fact linear in n for graphs of rankwidth 1 [24]. Hence, looking at the problem of classification of such codes in terms of rankwidth has additional advantages.

Another important step in the classification algorithm from [7] was computing weight-enumerators for a given code. The algorithm in [7] to compute the weight-enumerator for a linear $[n, k]$ code is essentially a brute-force search with complexity $\mathcal{O}(2^k)$. If $k > n/2$, it is necessary to go via the dual code to do the weight enumeration efficiently. We use Binary Decision Diagrams (BDD) to compute the weight-enumerators instead. The algorithm using BDD for weight enumeration has similar complexity to brute force, but has the benefit that we automatically get complexity $\mathcal{O}(2^{min\{k,n-k\}})$ without needing to consider the dual code.

The minimum distance of codes corresponding to distance-hereditary graphs is 2. We show that the minimum distance of a code is at least 3 if and only if the corresponding graph does not contain any pendant vertex or any twin-pairs. We also give an algorithm for computing an approximate minimum distance in codes corresponding to general graphs and leave some interesting open problems.

2 Preliminaries

Let $\mathbb{F}$ be a finite alphabet set (e.g. $\mathbb{F} = \{0, 1\}$ for binary codes) with size $|\mathbb{F}| =$ q. A *code* of length n is any subset of $\mathbb{F}^n$. For q a prime power and $\mathbb{F} = \mathbb{F}_q$ a finite field of q elements, a *linear code* is a linear subspace of the vector space $\mathbb{F}^n$. Let $GF(4)$ denote the finite field of four elements. The elements of $GF(4)$ are represented as $\{0, 1, \omega, \omega^2\}$ such that $\omega^2 = \omega + 1$. An *addtive code* C over $GF(4)$ of length n is an additive subgroup of $GF(4)^n$. If C contains 2^k codewords for some $0 \leq k \leq n$ then C can be represented via a $k \times n$ *generator matrix*, with entries from $GF(4)$. For any element $x \in GF(4)$, the *conjugate* of x is defined as $\bar{x} := x^2$. We define a function $Tr : GF(4) \to GF(2)$, known as *trace map* as for any $x \in GF(4), Tr(x) := x + \bar{x}$. The set $GF(4)^n$ denotes the set of all vectors of length n with each entry from $GF(4)$. The *dot product* of two vectors $u, v \in GF(4)^n$ is defined as The *Hermitian trace inner product* of two vectors over $GF(4)$ of length n, $u := (u_1, u_2, \ldots, u_n)$ and $v := (v_1, v_2, \ldots, v_n)$, is defined as

$$u * v := Tr(u \cdot \bar{v}) = \sum_{i=1}^{n} Tr(u_i \bar{v}_i) = \sum_{i=1}^{n} (u_i v_i^2 + u_i^2 v_i) \mod 2 \tag{1}$$

The *dual* of a code C denoted as $C^\perp$ is defined as

$$C^\perp = \{u \in \mathbb{F}^n \mid u * v = 0 \ \forall \ v \in C\}.$$

If $C = C^\perp$ then C is said to be *self dual.* If C is self dual, then $|C| = |\mathbb{F}|^{n/2}$. The *Hamming distance* between two codewords $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ is the number of places where they differ, and is denoted by $\text{dist}(x, y)$. The *Hamming weight* of a codeword $x = (x_1, \ldots, x_n)$ is the number of nonzero x_i and is denoted by $\text{wt}(x)$. We have the relation $\text{dist}(x, y) = \text{wt}\ (x-y)$. The *minimum distance* of the code C is the minimal Hamming distance between any two distinct codewords of C. Since C is an additive code, the minimum distance is also given by the smallest nonzero weight of any code word in C. A linear code with minimum distance d is called an $[n, k, d]$ code.

2.1 Graphs

A *graph* is a pair $G = (V, E)$ where V is a set of *vertices*, and $E \subseteq V \times V$ is a set of *edges*. A graph with n vertices can be represented by an $n \times n$ binary matrix called the *adjacency matrix* A such that $A_{ij} = 1$ if and only if $v_i v_j \in E$, and $A_{ij} = 0$ otherwise. In this paper we consider *simple undirected* graphs that have no loops, so the diagonal entries are 0, and no multiple edges. The *open neighbourhood* of $v \in V$, denoted by $N(v)$, is the set of vertices connected to v by an edge. The *closed neighbourhood* of $v \in V$, denoted by $N[v]$, is the set of vertices connected to v by an edge along with v itself. The number of vertices incident to v is called the degree of a vertex v. A *path* is a sequence of vertices, $(v_1, v_2, ..., v_i)$ such that $v_j, v_{j+1} \in E$. A graph is *connected* if there is a path from any vertex to any other vertex in the graph. A *cycle* is a path that has same vertex as start and end point. A *tree* is a connected graph that does not contain any cycle. Vertices of degree 1 in a tree are called *leaves.* We can designate any vertex in a tree as a *root* vertex. Usually the root vertex is taken as a start or reference point in the tree. The *complement* or the *inverse* of G is a graph H such that two distinct vertices of H are adjacent if and only if they are not adjacent in G. A subgraph H of a graph G is obtained by deleting vertices and edges from G. The subgraph H is called *induced* if H can be obtained from G by deleting only vertices from G. A vertex of degree 1 is called a *pendant.* A pair of vertices u, v are called *true twins* if $N[u] = N[v]$ and are called false twins if $N(u) = N(v)$. When we call a pair of vertices as *twin-pair*, then they can either be true-twins or false-twins.

Local complementation (LC) on v denoted by $G{*}v$ replaces the induced subgraph of G on $N(v)$ by its complement. See Figure 1 for an example.

A *Graph code* is an additive code over GF(4) that has a generator matrix of the form $C = A + \omega I$, where I is the identity matrix and A is the adjacency matrix of a simple undirected graph. A graph code is always self dual since its generator matrix has full rank over GF(2).

Theorem 1 ([25, 18]) *Every self-dual additive code over GF(4) is equivalent to a graph code.*

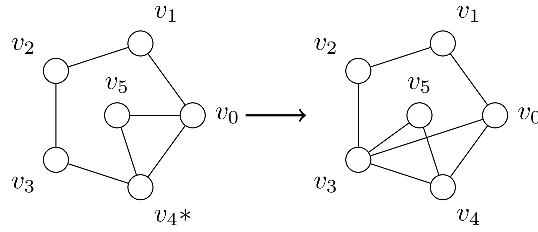

Fig. 1: Local complementation at v_4

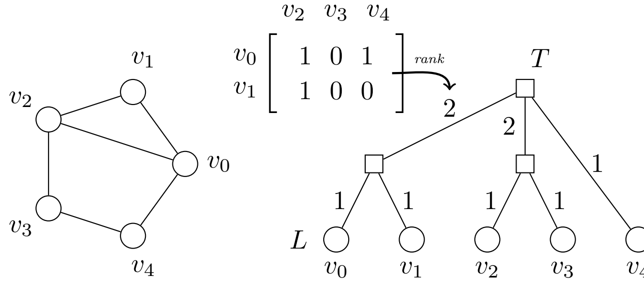

Fig. 2: A rank decomposition (T, L) of a graph. The rankwidth of T is 2. It so happens that the rankwidth of this graph is 2 as well.

2.2 Rankwidth

Let G be a simple undirected graph. A rank decomposition of G is a pair (T, L) of tree T with leaves L such that every internal node in the tree has degree 3 and there is a bijection between the set of leaves L of T to the set of vertices V of G. See Figure 2. For every edge e in T, we can associate a partition (A, B) of V, where A and B are set of leaves in the two sub-trees that result after deleting e. The weight of an edge e in T is the rank of the adjacency matrix of bipartite graph $((A, B), E)$ where E is the set of edges whose one endpoint is in A and other in B. The rankwidth of the decomposition (T, L) is the largest weight of an edge in T. The rankwidth $rw(G)$ of G is the smallest rankwidth of any tree decomposition of G. Local complementation preserves the rankwidth of a graph.

Theorem 2 **([19])** *Given a graph G and a vertex $v \in V(G)$, $rw(G) = rw(G * v)$.*

The *distance* between two nodes is the length of the shortest path between them. A distance-hereditary graph is a graph in which the distances in any connected induced subgraph are the same as they are in the original graph.

Theorem 3 **([19])** *G is distance-hereditary if and only if the rankwidth of G is at most 1.*

2.3 Weight enumerators

When using a code, it is important to know the probability of correct decoding. In practical situation it is necessary to know the weight distribution of the the code. Let A_i denote the number of vectors in a code C having Hamming weight equal to i. Then $A_0, A_1, A_2...$ is called the *weight distribution* of the code. The *Hamming weight enumerator* of C may be defined by a bivariate polynomial

$$W_c(x,y) = \sum_{u \in C} x^{n-wt(u)} y^{wt(u)} = \sum_{i=0}^{n} A_i x^{n-i} y^i. \qquad (2)$$

The weight enumerators of a code and its dual are related via the following theorem from [11, p. 127]

Theorem 4 *If C is an $[n,k]$ binary linear code with dual code $C^\perp$ then*

$$W_{C^\perp}(x,y) = \frac{1}{|C|} W_C(x+y, x-y)$$

where $|C| = 2^k$ is the number of codewords in $|C|$.

2.4 Binary Decision Diagrams

The data structure we use for calculating the weight enumerators of self dual additive code over $GF(4)$ are Binary Decision Diagrams (BDD). Binary Decision Diagrams are used in various applications, such as representing system of Boolean equations [23], application to permutations [17], or representing integer multiplication [20]. In this paper, we describe the basic operations we must do on a BDD in order to compute the weight enumerators of self dual codes of $GF(4)$.

2.4.1 Description of BDD

A Binary Decision Diagram (BDD) is a directed acyclic graph with a root node at the top and a true-node at the bottom. The nodes in a BDD are arranged in horizontal *levels*, and we visualize a BDD by drawing the levels in a top-down fashion. There is only one node on the highest level, called the *top node* or the *root node*, and there is only one node on the lowest level, called the *bottom node* or the *true-node*.

All edges in the BDD are directed downwards, with an edge always going between nodes on different levels. In other words, no edge is drawn between the nodes on the same level. Each node, except for the bottom node, has one or two outgoing edges, called the *0-edge* and/or the *1-edge*. The bottom node only has incoming edges and no outgoing edges. 0-edges are drawn as dotted lines, while 1-edges are drawn as solid lines. All the operations on BDD are done over GF(2). The transition from GF(4) to GF(2) is explained in Section 3.

A level in a BDD is usually associated with a single variable over GF(2). In our case, we allow a *linear combination* of variables over GF(2) associated with each level. A *path* in a BDD is a sequence of consecutive edges, where the end node of one edge is the start node for the next edge. A *complete* path starts in the top

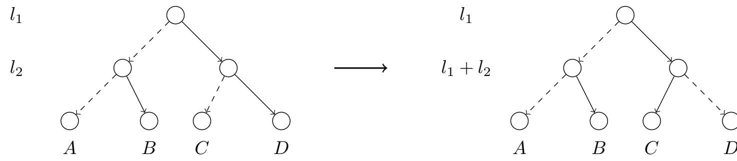

Fig. 3: Adding levels in a BDD

node and ends in the bottom node. We regard each edge in a path to assign a value over GF(2). If an e-edge starts from a node on a level associated with linear combination l, it yields the linear equation $l = e$ (for $e \in \{0, 1\}$).

An edge need not go to a node on the level directly below. In that case we say that the edge *jumps* over some levels. For an edge that jumps over some levels, we can always insert nodes on each jumped level. The inserted nodes should have the 0- and 1-edges pointing to the same nodes. In the left BDD of Figure 5 the last edge jumps over n levels, while the BDDs in Figure 6 have nodes inserted for the jumping edge. Both ways of drawing the BDDs are equivalent.

2.4.2 Adding Levels

The add operation allows us to add one linear combination for a level onto the linear combination for the level directly below, and change the BDD accordingly to keep the set of binary vectors encoded by the BDD unchanged. We explain the general case of adding levels using Figure 3. This figure shows the case when all edges are present. For cases where some edges are missing we can just imagine that the missing edges go to some "ghost" nodes, change the edges according to Figure 3, and then remove the ghost nodes.

Let l_1 and l_2 be the linear combination for two adjacent levels, with l_1 on the top. We want to add l_1 onto l_2. In the general case we have two outgoing edges, the 0-edge and the 1-edge, from the node on level l_1. By choosing values for l_1 and l_2 we end up in one of the four nodes labelled A,B,C,D in Figure 3. When we add l_1 to l_2, the lower level gets associated with the linear combination $l_1 + l_2$ and the choice of values for l_1 and l_2 must send us to the same node. For instance, $l_1 = 1$ and $l_2 = 0$ leads to node C in the left BDD. That choice of values gives $l_1 + l_2 = 1$, so after adding the levels the values $l_1 = 1$ and $l_1 + l_2 = 1$ must also end up in the node C in the right BDD. To preserve the set of vectors encoded in the BDD when replacing l_2 by $l_1 + l_2$ we must flip the outgoing edges from the node pointed to by the 1-edge from the node on level l_1.

2.4.3 Swapping Levels

How to swap the variables on two adjacent levels in a BDD and change the nodes and edges such that the resulting BDD encodes exactly the same set of vectors is explained in [22]. By swapping levels the linear combination associated with level i is swapped to level $i + 1$ and vice versa without affecting the set of vectors encoded by the BDD. We explain the general case of swapping levels using Figure 4.

Let l_1 and l_2 be the linear combination for two adjacent levels, with l_1 being above l_2. We want to swap these linear combinations without disturbing the set

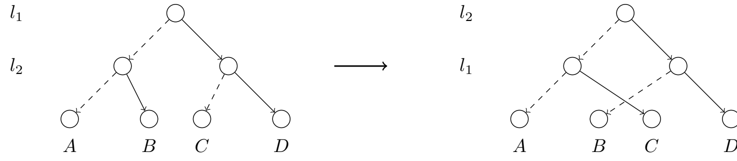

Fig. 4: Swapping levels in a BDD

of vectors encoded in the BDD. For the node on level l_1, we have two outgoing edges. As seen previously choosing values for l_1 and l_2 we end up in one of the four nodes labelled A,B,C,D in Figure 4. In the left BDD before swapping, the choice of the value $l_1 = 0$ and $l_2 = 1$ leads us to the node B. After we swap l_1 and l_2, the choice of values $l_1 = 0$ and $l_2 = 1$ must still leads to the node B. This is achieved by swapping edges on the lower level of the paths where l_1 and l_2 have different values.

The swap and add operations have linear time complexity in the number of nodes on the two affected levels, so it is very cheap to do when this number is small. The drawback is that the number of nodes on the lower of the two levels may, in the worst case, double during the operation. This happens when there is only one node on the lower level, but after add or swap is done two nodes are needed to keep the paths intact. Hence repeatedly doing swap or add operations through the BDD may lead to exponential growth in the number of nodes. The number of nodes may also decrease during this process, but finding the order of the linear combinations giving the smallest BDD is an NP-hard problem [3].

3 Construction of BDD

Let G be an $n \times n$ generator matrix of a code C over $GF(4)$. We compute the set of all code words by considering all possible linear combinations over $GF(2)$ of the rows of G. This is done by first expanding the matrix into an $n \times 2n$ matrix G' over $GF(2)$ by associating (mapping) each $GF(4)$-element to two bits as follows: $0 = (00)$, $1 = (01)$, $\omega = (10)$, $\omega^2 = (11)$.

$$G = \begin{bmatrix} \omega\ 1\ .\ .\ .\ . \\ 1\ \omega\ 0\ .\ .\ . \\ .\ 0\ \omega\ 1\ .\ . \\ .\ .\ .\ .\ .\ . \\ .\ .\ .\ .\ .\ . \\ .\ .\ .\ .\ .\ \omega \end{bmatrix}_{n\times n} \Longrightarrow G' = \begin{bmatrix} 1\,0\,0\,1\ .\ .\ .\ . \\ 0\,1\,1\,0\,0\,0\ .\ . \\ .\ .\ 0\,0\,1\,0\ .\ . \\ .\ .\ .\ .\ .\ .\ .\ . \\ .\ .\ .\ .\ .\ .\ .\ . \\ .\ .\ .\ .\ .\ .\ 1\,0 \end{bmatrix}_{n\times 2n}$$

Now we multiply all binary strings $(c_1, c_2, c_3, ...c_n)$ of length n to the matrix G' to get the set of all code words $(x_1, x_2, x_3, ...x_{2n-1}, x_{2n})$.

$$(c_1, c_2, c_3, ...c_n) \begin{bmatrix} 1\,0\,0\,1\ .\ .\ .\ . \\ 0\,1\,1\,0\,0\,0\ .\ . \\ .\ .\ 1\,0\,0\,1\ .\ . \\ .\ .\ .\ .\ .\ .\ .\ . \\ .\ .\ .\ .\ .\ .\ .\ . \\ .\ .\ .\ .\ .\ .\ 1\,0 \end{bmatrix} = (x_1, x_2, x_3, ...x_{2n-1}, x_{2n})$$

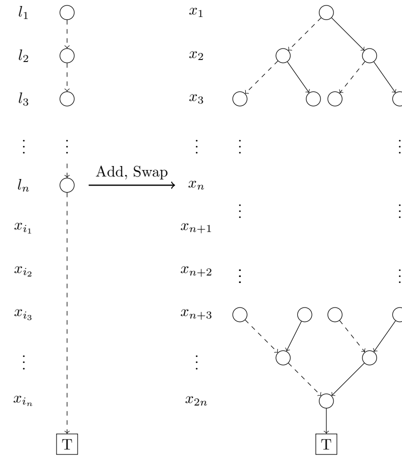

Fig. 5: BDD after adding and swapping to resolve all linear combinations into single variables, sorted on the levels.

In order to construct the BDD that has all code words as paths, we introduce the parity check matrix H. The parity check matrix describes the set of linear relations that the coordinates of each code word must satisfy. If x is a code word and H is the parity check matrix then $xH^T = 0$. For all code words in the code, we have the following relations:

$$(x_1, x_2, x_3, ...x_{2n}) \begin{bmatrix} 1\,0\,0\,1\,.\,. \\ 0\,1\,1\,0\,0\,0 \\ .\,.\,1\,0\,0\,1 \\ .\,.\,.\,.\,.\,. \\ .\,.\,.\,.\,.\,. \\ .\,.\,.\,.\,.\,. \\ .\,.\,.\,.\,.\,. \\ .\,.\,.\,.\,.\,. \end{bmatrix}_{2n \times n} = (0, 0, 0.....0, 0)$$

Let the linear equations given by $xH^T = 0$ be $l_i = 0$ for $1 \leq i \leq n$. A BDD that encodes all code words of the code, i.e. x-vectors satisfying all $l_i = 0$ is given in the left BDD of Figure 5. In that BDD, the x_{i_j} are free variables such that the l_i's can not be written as a sum of the x_{i_j}'s. In other words, considering the free variables as linear combinations too, all linear combinations in the BDD are independent. The free variables can take any value, and the l_i's represent linear combinations of the coordinates that must be 0 in order to be a code word.

We now use add and swap operations on this basic BDD to resolve the linear combinations l_i. By resolving the linear combinations we mean that we add together some of the linear combinations and free variables to transform l_i into a single variable. We use the swap operation to move levels that need to be added so they are adjacent to each other, and the add operation to do the actual addition.

We apply the operations until only single variables appear on all levels. We sort the levels in order such that x_1 appears on the top and x_{2n} appears on the lowest level, as shown in the right BDD of Figure 5. We have given an explicit

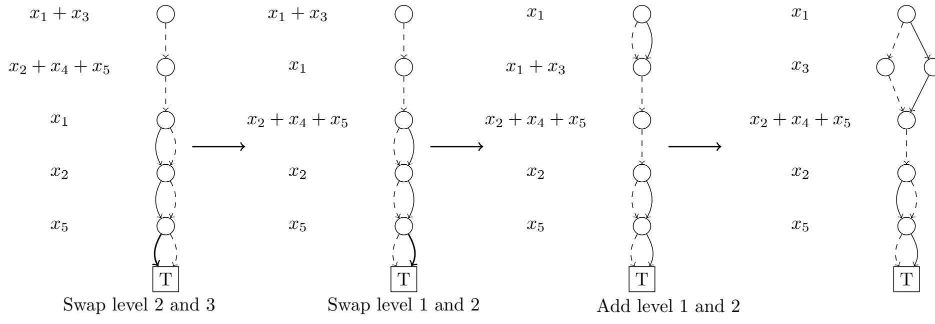

Fig. 6: Example: Performing add and swap to a BDD

example of how add and swap operations are performed in a BDD, see Figure 6. Now the paths of the BDD represent all code words in C.

Complexity: In this paper we are concerned with the special case where the codes are of length $2n$ and dimension n. However, constructing the BDD representing a binary code can be done for any length n and any dimension $k \leq n$. We explain here the complexity, in terms of number of nodes in the final BDD, in the general case for an $[n, k]$ linear code C over $GF(2)$.

Lemma 1 *The number of nodes on any level of the final BDD after resolving all linear combinations for a code C is at most 2^k.*

Proof The number of code words in C is 2^k, and so the total number of paths in the BDD is also 2^k. There are no edges between nodes on the same level, so all nodes on any level are part of different paths. Hence the number of nodes on any level can not be more than 2^k.

Lemma 2 *The number of nodes on any level of the final BDD after resolving all linear combinations for a code C is at most 2^{n-k}.*

Proof The number of nodes on any level of the basic BDD before resolving any linear combinations is 0 or 1. Applying the swap or add operation will at most double the number of nodes on the lower of the affected levels. We resolve one linear combination by adding certain levels in the BDD. Starting with the lowest level and moving levels upwards, adding as needed, we see that each level is involved in the resolution of a linear combination only once. So the total number of nodes in any level of the BDD after resolving one linear constraint may at most double. Since we are resolving $n - k$ linear combinations, the total number of nodes in the final BDD will be at most 2^{n-k}.

Combining lemmas 1 and 2 we get the following result.

Theorem 5 *The number of nodes in the final BDD representing the code words of a binary linear $[n, k]$ code is of order $\mathcal{O}(2^{\min\{k,n-k\}})$.*

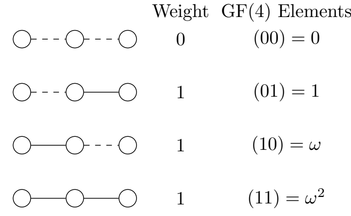

Fig. 7: Weight of consecutive edges in a BDD

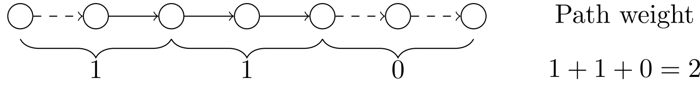

Fig. 8: Example of computing weight of a path

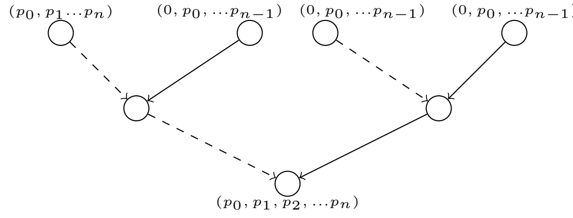

Fig. 9: Shift weight vector by one to the right when the pair of edges indicate a non-zero $GF(4)$-element.

3.1 Algorithm for computing weight enumerators

Recall that pairs of coordinates (x_{2i-1}, x_{2i}) actually represent one element in $GF(4)$. A path in the BDD with resolved and sorted levels has length $2n$, but represents a code word of length n with elements from $GF(4)$. When computing the weight enumeration we therefore count how many non-zero $GF(4)$-elements a path (code word) represents. Figure 7 shows how the different elements of $GF(4)$ are represented as paths in the BDD, and Figure 8 gives an example of how to count the weight of a code word represented as a path.

Now we describe our algorithm for computing weight enumerators using BDD. We explain the whole process, where we start with a graph G, and want to compute the weight enumeration of its corresponding code over $GF(4)$.

Step 1: Given the adjacency matrix A of a graph G, we obtain the generator matrix $G = A + \omega I$ over GF(4). Note all operations are over GF(2).

Step 2: We transform the generator matrix over GF(4) to GF(2) by mapping two bits to each GF(4) element in the matrix as follows: 0 = (00), 1 = (01), ω = (10), ω^2 = (11).

Step 3: We obtain the parity check matrix H, from the generator matrix over GF(2) and, get the parity check equations $l_1 = l_2 = \ldots = l_n = 0$.

Step 4: Construct the BDD for $l_1 = l_2 = \ldots = l_n = 0$ with $x_{i_1}, x_{i_2}, \ldots, x_n$ as variables.

Step 5: Now we apply add and swap operations to the BDD to resolve and sort the linear combinations, see figures 5 and 6.

Step 6: **Weight enumeration**: Each node in the BDD has a vector of length $(n+1)$ of integer values, denoted as $(p_0, p_1, p_2, \dots, p_n)$. For a given node, p_i indicates the number of paths of weight i below this node.

1. Start with setting $(1, 0, 0, \dots, 0)$ as the vector for the true-node at the bottom. We say there is one path of weight 0 from the true-node to itself (the empty path).
2. We compute the vectors for the other nodes in a recursive way, from the lower levels to higher ones. When a pair of edges from a node T to A contribute 0 to the path weight, the weight distribution below T along this path is the same as for A. In other words, prepending the partial code words with a zero does not change the weight distribution. When the pair of edges from T to A contribute 1 to the weight, the paths of weight i below A become paths of weight $i+1$ below T. Hence the weight enumeration vectors for T are obtained by shifting the vector for A by one position to the right, as shown in Figure 9.
3. Assuming all weight distribution vectors have been computed for the nodes on one level, compute the weight distribution for the nodes two levels above by adding all the weight contributions, shifting them by one position to the right as needed. This is shown in Figure 10.
4. Compute weight distributions for all nodes in the BDD, moving upwards two levels at the time. In the end, the vector for the root node gives the weight distribution of the whole code.

The complexity of computing the weight enumeration of a given code represented as a BDD is $\mathcal{O}(N)$, where N is the number of nodes in the BDD and adding two integer vectors counts as a unit operation. In terms of single integer additions, the complexity is $\mathcal{O}(nN)$.

We have described the algorithm for computing the weight enumeration when the code represented as a BDD is regarded as being over $GF(4)$. Going back to the general case of an $[n, k]$ linear code over $GF(2)$, we can easily modify the algorithm to compute the weight distribution for any binary linear code when it is represented as paths in a BDD.

Computing the weight distribution in general is a hard problem, that can only be solved by brute force. The naive way of doing it (without the BDD representation) is to run through all the code words and count their weights. This has complexity $\mathcal{O}(2^k)$. If $k > n/2$, the complexity of doing weight enumeration becomes bigger than it needs to be. Then one can compute the weight distribution for the dual code (of dimension $n-k$ and complexity $\mathcal{O}(2^{n-k}) < \mathcal{O}(2^k)$), and use Theorem 4 to find the weight distribution of the given code.

Theorem 5 shows the advantage of using the BDD approach to calculate the weight enumeration: We have a single algorithm that automatically gets the lowest complexity possible, regardless of whether k is bigger or smaller than $n/2$.

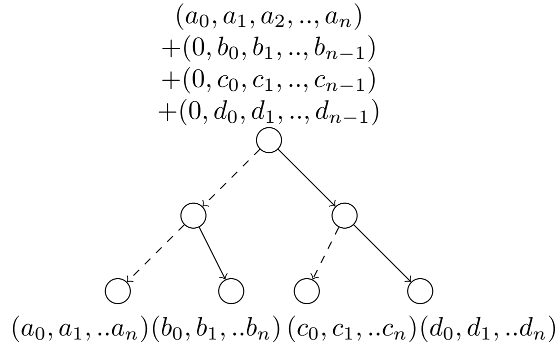

Fig. 10: Computing weight enumeration for one node.

4 Classification for Rankwidth 1 graphs

The algorithm for classifying self-dual codes corresponding to general graphs as described in [7]: Let L_{n-1} be the set of representatives for classes of graphs on $n-1$ vertices corresponding to equivalent self-dual codes.

- Compute the set of graphs E_n by adding a vertex to each graph in L_{n-1} in $2^{n-1}-1$ ways i.e. making the vertex adjacent to every possible non-empty subset of the vertex set.
- For each set of of isomorphic graphs keep only one graph in E_n.
- Use weight-enumerators to partition the set E_n i.e. graphs corresponding to same weight-enumerators are put in one class.
- Partition each class in E_n by checking for self-dual equivalence.
- Output L_n that contains one graph from each class in E_n.

We utilize the following definition of distance hereditary graphs.

Theorem 6 **([2])** *Let G be a finite graph with at least two vertices. Then G is distance-hereditary if and only if G is obtained from an edge by a sequence of one of vertex extensions: add vertex as a pendant, add vertex as a true-twin to an existing vertex and add vertex as a false-twin to an existing vertex.*

Let $\mathcal{G}_{n-1}$ be all connected graphs of rankwidth 1 on $n-1$ vertices. Then $\mathcal{G}_n$ can be obtained by adding a vertex to each graph in $\mathcal{G}_{n-1}$ as a pendant or a twin to some vertex. Consider $\mathcal{C}$ to be the orbit of a graph $G \in \mathcal{G}_{n-1}$. Let $G_1, G_2 \in \mathcal{C}$. Then there is a sequence of LC operations S that can take G_1 to G_2. Let $\mathcal{E}_1$ and $\mathcal{E}_2$ be the $3(n-1)$ extensions of G_1 and G_2 obtained via adding pendants or twins. We show that via applying S on any graph in $\mathcal{E}_1$, we end-up with a graph in $\mathcal{E}_2$ implying that $\mathcal{E}_1$ and $\mathcal{E}_2$ are *LC-equivalent.*

Let u be a new vertex added to G_1 as a pendant or twin to a vertex $v \in V(G_1)$. The LC operations at vertices in G_1 switch the role of u relative to v as a pendant or a twin. At the same time, G_1 changes to G_2 after S has been performed. Then, u can be seen as being attached to G_2 as a pendant or twin (according to what happens after apply S to $G_1 + u$). Hence, any graph $\mathcal{E}_2$ can be seen as being obtained from a graph in $\mathcal{E}_1$ via applying S. This implies that instead of considering extensions

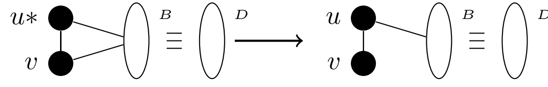

Fig. 11: Here filled circles denote vertices and ellipses denote set of vertices. An edge from a vertex to a set denotes that vertex is adjacent to every vertex in the set. The vertices u and v are true twins. After LC at u, the degree of v reduces to 1.

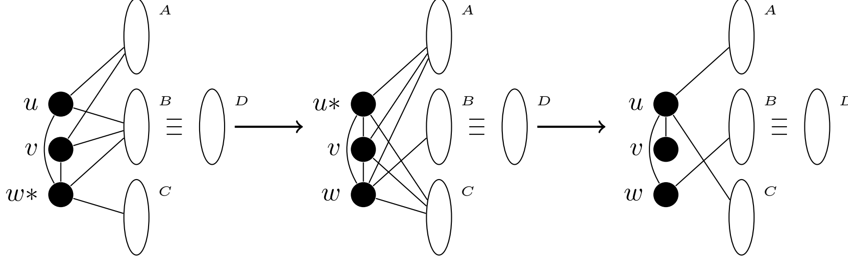

Fig. 12: The vertices u and v are false-twins and w is a common neighbor of u and v. After LC operation at w followed by LC operation at u reduces the degree of v to 1.

of $\mathcal{C}$, we need only consider extensions of just one representative from $\mathcal{C}$. Let L_{n-1} be the set of representatives of all orbits in $\mathcal{G}_{n-1}$.

Since rankwidth is preserved by LC operations, the graphs in the sets L_{n-1}, E_n and L_n must be of rankwidth 1. Hence by above discussion, in the computation of E_n from L_{n-1}, the vertex must be added as a pendant or a false-twin or a true-twin. There are at most $3(n-1)$ ways to do that. So instead of branching in $2^{n-1}-1$ ways, we need only branch in at most $3n-3$ ways. Furthermore, the isomorphism testing in E_n is linear in n for rankwidth 1 graphs.

5 Minimum Distance

Glynn et al [13] showed that the minimum distance of a code is equal to one plus the minimum vertex degree over all graphs in the corresponding LC orbit.

Lemma 3 *If a connected graph contains a twin-pair, then the minimum distance of the corresponding code is 2.*

Proof Let u, v be a true-twin pair. Then, after LC operation at u, the degree of v in the resulting graph is 1. See Figure 11.

If u, v is a false-twin pair, then after LC operation at a common vertex w, they become true-twins in the resulting graph. Then, as in the above case, a pendant results after an LC operation at u. See Figure 12. Since LC operation preserves connectivity, this is the minimum possible degree of a vertex over the entire LC orbit of the graph. Hence, the minimum distance of the corresponding code is 2.

Lemma 4 *Codes with corresponding graphs of rankwidth 1 have minimum distance 2.*

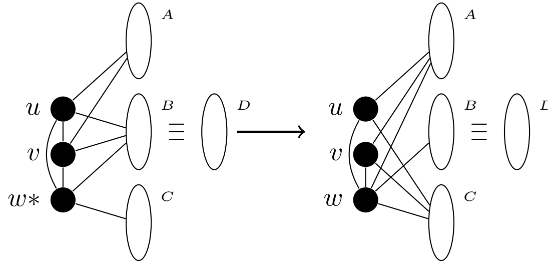

Fig. 13: The vertex w is in the common neighborhood of true-twins u and v. LC at w makes u, v into a false-twin pair.

Proof The rankwidth 1 graphs are exactly distance hereditary graphs which can be constructed recursively by adding a pendant or a twin-pair. Such graphs will have either a pendant or a twin-pair (just look at the last step in the construction of the graph). Hence, by Lemma 3, codes with corresponding graphs of rankwidth 1 have minimum distance 2.

Lemma 5 *If a graph contains a twin-pair, then every graph in its LC orbit will contain a twin-pair or a pendant.*

Proof Let u and v be true-twin pairs. An LC operation at either u or v will yield a pendant. An LC operation at $w \notin N(u)$ does not effect $N(u)$, hence, u and v continue as twin pairs. Let $w \in N(u)/v$. Now, after LC at w, the pair u and v become false-twins. See Figure 13.

Now, suppose u and v are false-twins. They become true twins if an LC operation is done on w, see Figure 12. An LC operation at either u or v or at a vertex not in their common neighborhood does not change the neighborhood of u or v and hence, they continue as false-twins. This concludes the proof of the lemma.

Lemma 6 *If G does not have a pendant or a twin-pair, then no graph in the LC orbit of G will have a twin-pair.*

Proof We prove this by contradiction. Suppose G does not contain any twin-pair or pendant, but an LC operation at u created the twins v and w. By Lemma 5, all graphs in the orbit must have either a pendant or a twin-pair. Since G lies in the orbit, it must have a pendant or twin-pair, contradicting the assumption.

Combining Lemma 3, Lemma 5 and Lemma 6 gives the following theorem.

Theorem 7 *The minimum distance of a self-dual additive code over $GF(4)$ is at least 3 if and only if the corresponding graph G has no pendants or twin-pairs.*

5.1 An approximation algorithm for minimum distance

The problem of computing the minimum distance of a binary linear code is NP-hard [26]. In addition, the problem is hard to approximate within any constant

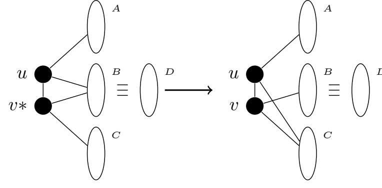

Fig. 14: Degree of u decreases when $|C| < |B|$.

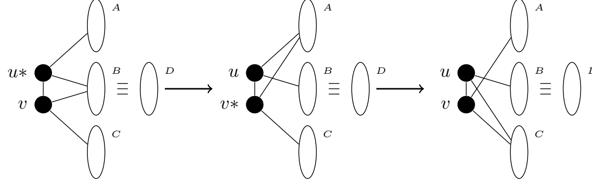

Fig. 15: Degree of u decreases when $|C| < |A|$

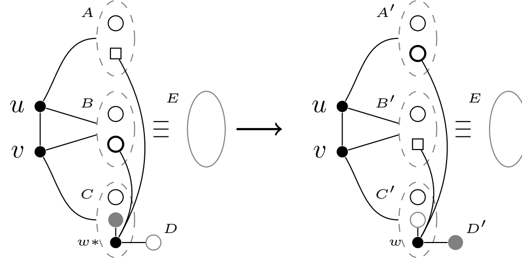

Fig. 16: Taking LC operation at $w \in C$. If $|C'|$ is less than either $|A'|$ or $|B'|$, then the degree of u can be decreased.

factor in random polynomial time [10]. For self-dual codes over $GF(4)$, the minimum distance is $1+\delta$ where δ is the minimum degree of any vertex in any graph in the LC orbit of the graph corresponding to the code. It is possible to get the minimum distance from the weight enumerator polynomial or from the LC orbit, but both these approaches take exponential time. In this section we discuss a heuristic approach to get some upper bound on the minimum distance.

Computing δ is equivalent to finding a sequence of LC operations starting at some vertex u such that at the end, there is a vertex of degree δ in the resulting graph. Clearly, finding this sequence is hard. The strategy we use is to pick a vertex in the graph and try to decrease its degree as much as possible via LC operations. Consider Figure 14. After LC operation at v, the degree of u changes by $(1+|A|+|C|)-(1+|A|+|B|) = |C|-|B|$. The degree of u decreases if $|B| > |C|$. If $|A| > |C|$ and $|B| < |C|$, then applying LC at u followed by LC at v decreases the degree of u by $|A| - |C|$, see Figure 15.

What if $|C|$ is larger than both $|A|$ and $|B|$? Then, either we can try LC operation at some other vertex in the neighborhood of u or try to decrease $|C|$. The size of $|C|$ can only be decreased by an LC operation at a vertex in C, see Figure 16. If $|C'|$ is still larger than $|A'|$ and $|B'|$, then we can try to decrease the size of $|D|$

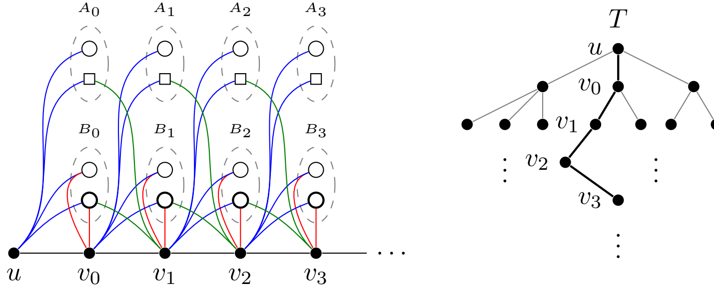

Fig. 17: Decomposition of the graph G along a path $uv_0v_1v_2v_3\dots$ in the BFS-tree T with u as root vertex. Black-filled circles represent vertices. Other shapes represent sets. An edge from a vertex to a set represents that every vertex in the set is a neighbor of the vertex. Edges between sets have not been shown.

Vertex	Partition of neighborhood	Conditions for LC
v_0	A_0 $\quad B_0$ $\quad C_1$	$\lvert C_1\rvert < \max\{\lvert A_0\rvert, \lvert B_0\rvert\}$
v_1	A_{00} A_{01} $\quad B_{00}$ B_{01} $\quad A_1$ B_1 $\quad C_2$	$\lvert A_1\rvert + \lvert C_2\rvert < t_1 := \max\{\lvert A_{00}\rvert + \lvert B_{01}\rvert, \lvert B_{00}\rvert + \lvert A_{01}\rvert\}$
v_2	A_{10} A_{11} $\quad B_{10}$ B_{11} $\quad A_2$ B_2 $\quad C_3$	$\lvert A_2\rvert + \lvert C_3\rvert < t_2 := t_1 - (\lvert A_{10}\rvert + \lvert B_{11}\rvert)$
v_3	A_{20} A_{21} $\quad B_{20}$ B_{21} $\quad A_3$ B_3 $\quad C_4$	$\lvert A_3\rvert + \lvert C_4\rvert < t_3 := t_2 - (\lvert A_{20}\rvert + \lvert B_{21}\rvert)$
	$\vdots$	

Fig. 18: At each vertex v_i store sets A_i, B_i, C_i and t_i. If $|A_3| + |C_4| < t_3$, then $|A_2| + |C_3| < t_2$ which in turn implies $|A_1| + |C_2| < t_1$. Hence, just by looking at v_3 we can decide whether degree of u can be decreased.

first by taking an LC operation at a vertex in D, then consider LC at w. Following this chain of thought, we see that the sequence of LC operations correspond to a path in a Breadth-First-Search (BFS) tree T of G with u as the root vertex. A BFS tree is constructed as follows: Pick u as root. At each layer i, the vertices in layer i are neighbors of vertices in layer $i-1$ that have not been already placed in some layer.

We aim to use this tree to find a path that gives a sequence of LC operations to decrease the degree of u. If no such path exists, then the algorithm reports the degree of u as a candidate for δ. See Figure 17. Now, we state the algorithm:

1: Construct BFS tree For $u \in V(G)$, construct Breadth-First-Search tree T with u as the root node. The neighborhood of a vertex at layer i in T lie only in layers $i-1, i$ and $i+1$. Note that we would require to reconstruct the tree after LC operations along the path. The tree T is used to guide the sequence of LC operations to be applied to decrease the degree of u.

2: Partition neighborhoods At each node of the tree, we store some information that can be used to check whether LC along the path to the root will decrease the degree of u.
At the root node u, we have $C = N(u)$.
At the second layer in the tree, for each vertex $v_0 \in C_0 = N(u)$, the neighborhood of v_0 can be partitioned as (apart from u) as (A_0, B_0, C_1) where $B_0 = C_0 \cap N(v_0), A_0 = C_0 - B_0$ and $C_1 = N(v_0) - (u \cup C_0)$.
At the ith layer with $j = i-1, k = i+1$, for each vertex $v_i \in C_i$, partition the neighborhood of v_i as (apart from v_j) $(A_{j0}, A_{j1}, B_{j0}, B_{j1}, A_i, B_i, C_k)$ where $A_{j0} = A_j - N(v_i), A_{j1} = A_j - A_{j0}, B_{j0} = B_j - N(v_i), B_{j1} = B_j - B_{j0}, B_i = C_i - N(v_i), A_i = C_i - B_i, C_k = N(v_i) - (v_j \cup A_{j1} \cup B_{j1} \cup B_i)$. See Figure 17.
3: Book keeping For each $i \geq 2$ with $j = i-1, k = i+1$, at each vertex v_i, we store the sets A_i, B_i, C_k and the values $a_i := |A_{j0}| + |B_{j1}|, t_i := t_j - a_i$ and $|A_i| + |C_k|$.
4: Check for LC If $|A_i| + |C_k| \leq t_i$, then an LC operation along the path from v_i to the root will decrease the degree of u. See Figure 18. Then apply LC operations along this path and construct the BFS-tree T for the new graph and repeat.
5: Return degree of u If there does not exist any vertex in the tree with $|A_i| + |C_k| \leq t_i$, then return the current degree of u as δ_u.
6: Terminate Finally, the algorithm outputs the smallest δ_u over all vertices in the graph.

Running Time: For a given graph G, the BFS-tree can be constructed in linear time. For each vertex v, the partition for neighborhood of v can be computed in polynomial time and it will take polynomial space to store the necessary information. Hence, in polynomial time we can decide whether there exists a path in the tree along which LC operations decreases the degree of u. Hence, the algorithm terminates in polynomial time.

6 Conclusion and open problems

In this paper we have shown that classification of self-dual codes over GF(4) is essentially a graph-theoretic problem and a lot remains to be explored. We showed that structural properties of rankwidth 1 graphs can make the classification of corresponding codes faster. Graphs of rankwidth 2 are not fully understood yet. It would be interesting to follow up with graph classes that can be constructed recursively like distance-hereditary graphs.

We used BDDs to obtain weight-enumerator polynomials which although improves on the known algorithm is still exponential. Since minimum distance can be obtained from this polynomial, we can not hope to find the weight enumeration any faster. It would be interesting to improve the algorithm though. We also characterize when a graph will have minimum degree at least 2 over the entire LC orbit. Can we find conditions for minimum degree at least 3?

References

1. Bachoc, C., Gaborit, P.: On extremal additive I_4 codes of length 10 to 18. Electronic Notes in Discrete Mathematics **6**, 55–64 (2001). URL `https://doi.org/10.1016/S1571-0653(04)00157-X`

2. Bandelt, H., Mulder, H.M.: Distance-hereditary graphs. J. Comb. Theory, Ser. B **41**(2), 182–208 (1986). URL https://doi.org/10.1016/0095-8956(86)90043-2
3. Bollig, B., Wegener, I.: Improving the variable ordering of obdds is np-complete. IEEE Trans. Computers **45**(9), 993–1002 (1996). URL https://doi.org/10.1109/12.537122
4. Bouchet, A.: Graphic presentations of isotropic systems. Journal of Combinatorial Theory, Series B **45**(1), 58 – 76 (1988). URL http://www.sciencedirect.com/science/article/pii/009589568890055X
5. Calderbank, A.R., Rains, E.M., Shor, P.W., Sloane, N.J.A.: Quantum error correction via codes over GF(4). IEEE Trans. Information Theory **44**(4), 1369–1387 (1998). URL https://doi.org/10.1109/18.681315
6. Danielsen, L.E., Parker, M.G.: Spectral orbits and peak-to-average power ratio of boolean functions with respect to the $\{I, H, N\}^n$ transform. In: Sequences and Their Applications - SETA 2004, Third International Conference, Seoul, Korea, October 24-28, 2004, Revised Selected Papers, pp. 373–388 (2004)
7. Danielsen, L.E., Parker, M.G.: On the classification of all self-dual additive codes over GF(4) of length up to 12. J. Comb. Theory, Ser. A **113**(7), 1351–1367 (2006). URL https://doi.org/10.1016/j.jcta.2005.12.004
8. Danielsen, L.E., Parker, M.G.: Edge local complementation and equivalence of binary linear codes. CoRR **abs/0710.2243** (2007). URL http://arxiv.org/abs/0710.2243
9. Danielsen, L.E., Parker, M.G.: Edge local complementation and equivalence of binary linear codes. Des. Codes Cryptography **49**(1-3), 161–170 (2008). URL https://doi.org/10.1007/s10623-008-9190-x
10. Dumer, I., Micciancio, D., Sudan, M.: Hardness of approximating the minimum distance of a linear code. IEEE Trans. Information Theory **49**(1), 22–37 (2003). URL https://doi.org/10.1109/TIT.2002.806118
11. F.J.Macwilliams, Sloane, N.: The Theory of Error-Correcting Codes. North-Holland (1977)
12. Gaborit, P., Huffman, W.C., Kim, J., Pless, V.: On additive $GF(4)$ codes. In: Codes and Association Schemes, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, November 9-12, 1999, pp. 135–150 (1999)
13. Glynn, D., Gulliver, T., Maks, J., Gupta, M.: The geometry of additive quantum codes (2004)
14. Gottesman, D.: Stabilizer codes ad quantum error correction. Phd Thesis, Caltech (May 1997). DOI arXiv:quant-ph/9705052
15. Grohe, M., Schweitzer, P.: Isomorphism testing for graphs of bounded rank width. In: IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015, pp. 1010–1029 (2015)
16. Hein, M., Eisert, J., Briegel, H.J.: Multiparty entanglement in graph states. Phys. Rev. A **69**, 062311 (2004). URL https://link.aps.org/doi/10.1103/PhysRevA.69.062311
17. Minato, S.: πdd: A new decision diagram for efficient problem solving in permutation space. In: Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings, pp. 90–104 (2011)
18. Van den Nest, M., Dehaene, J., De Moor, B.: Graphical description of the action of local clifford transformations on graph states. Phys. Rev. A **69**, 022316 (2004). URL https://link.aps.org/doi/10.1103/PhysRevA.69.022316
19. Oum, S.: Rank-width and vertex-minors. J. Comb. Theory, Ser. B **95**(1), 79–100 (2005). URL https://doi.org/10.1016/j.jctb.2005.03.003
20. Raddum, H., Varadharajan, S.: Factorization using binary decision diagrams. Cryptography and Communications **11**(3), 443–460 (2018)
21. Raussendorf, R., Browne, D.E., Briegel, H.J.: Measurement-based quantum computation on cluster states. Phys. Rev. A **68**, 022312 (2003). URL https://link.aps.org/doi/10.1103/PhysRevA.68.022312
22. Rudell, R.: Dynamic variable ordering for ordered binary decision diagrams. In: Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Santa Clara, California, USA, November 7-11, 1993, pp. 42–47 (1993)
23. Schilling, T.E., Raddum, H.: Solving compressed right hand side equation systems with linear absorption. In: Sequences and Their Applications - SETA 2012 - 7th International Conference, Waterloo, ON, Canada, June 4-8, 2012. Proceedings, pp. 291–302 (2012)
24. Uehara, R., Uno, T.: Canonical tree representation of distance hereditary graphs and its applications. Tech. rep. (2006)

25. Van Den Nest, M.: Local equivalence of stabilizer states and codes. Phd thesis, K. U. Leuven, Belgium (May 2005)
26. Vardy, A.: The intractability of computing the minimum distance of a code. IEEE Trans. Information Theory **43**(6), 1757–1766 (1997). URL `https://doi.org/10.1109/18.641542`