An Algorithm for Checking Normality of Boolean Functions

Magnus Daum Hans Dobbertin Gregor Leander

Ruhr-Universität Bochum, Postfach 102148, 44780 Bochum, Germany
{daum,dobbertin,leander}@itsc.rub.de

Abstract

Normality and weak normality are important properties of Boolean functions, which are defined via the *existence* of a flat fulfilling certain criteria. Since at the moment there are no mathematical methods known to check and even more to disprove such properties without looking at all flats in some way, we usually have to do kind of an exhaustive search and check for every single flat, whether it fulfills the criteria or not.
In this article we present an algorithm which fulfills this task in a much shorter time. The general idea is to do the exhaustive search for smaller flats and then combine them recursively to find all flats on which the function is affine. This tricky way of enumerating all pertained flats is the main part of the algorithm which might be also adaptable to other properties besides normality.
With the help of this algorithm we were able to disprove the (weak) normality of some explicit examples of new bent functions recently found by Dillon and Dobbertin, thus solving an open question about the existence of non normal and non weakly-normal bent functions, which was proposed by Dobbertin. Additionally we present another application of the described algorithm, namely checking whether a given bent function is of Maiorana-McFarland type.

Key words: algorithm, Boolean function, bent function, normal function

1 Introduction

In cryptography Boolean functions are used in many different areas, the probably most important being the design of S-Boxes for symmetric encryption. Besides the algebraic degree and the nonlinearity, the property of normality is one of the most important criteria to understand the structure of a Boolean function. But as it is defined via the *existence* of a flat fulfilling certain criteria, it is very hard to check. Thus it is not surprising that the question about the existence of non normal bent functions, as proposed by Dobbertin in [Dob], has been open for a long time. But with the help of the algorithm presented in this article, we were able to verify that some explicit examples of recently found bent functions (see [DD]) are non normal, thus solving this open question. This is described in [CDDL] in more detail.

Let $n = 2m$ be an even number. Then normality is defined as follows:

Definition 1.1
A Boolean function $f : \mathbb{F}_2^n \to \mathbb{F}_2$ is called normal if there is a coset of an m-dimensional subspace such that f is constant on this coset.

Instead of cosets of m-dimensional subspaces of $\mathbb{F}_2^n$ in the following we will just speak of *flats* of dimension m. As many of the often studied properties of Boolean functions (e.g. bentness) are invariant under addition of affine functions it is natural to consider a generalization of this definition:

Definition 1.2
A Boolean function $f : \mathbb{F}_2^n \to \mathbb{F}_2$ is called weakly-normal if there is a flat of dimension m such that the restriction of f to this flat is affine.

A function f is weakly-normal if and only if there is an $a \in \mathbb{F}_2^n$ such that $f(x) + \langle a, x\rangle$ is normal. Thus every function $f(x)+\langle a, x\rangle$ with $a \in \mathbb{F}_2^n$ is weakly-normal, if f is weakly-normal.

Checking (weak) normality of a function usually needs one to take into account all flats of dimension m to check whether f is constant (affine) on one of them. One possible but rather complex way of doing this would be to do an exhaustive search on all flats of dimension m. In this article we present an algorithm, which, given a Boolean function $f : \mathbb{F}_2^n \to \mathbb{F}_2$, is able to compute a list of all flats of dimension m of $\mathbb{F}_2^n$ on which f is affine in much less time than needed for an exhaustive search.

Additionally besides checking normality this algorithm can also be used to check whether a given bent function is a Maiorana-McFarland bent function, as it is described in Section 6.

2 General Idea

The main idea of the algorithm presented in this article is to make use of the fact that a Boolean function which is affine on a flat A is also affine on all flats contained in A.
Even more the function is either constant on A and hence constant on all flats or we can find two flats $A_0, A_1 \subset A$ with $\dim(A_0) = \dim(A_1) = \dim(A) \Leftrightarrow 1$ and $A = A_0 \cup A_1$ such that the function is 0 on A_0 and 1 on A_1. In the latter case, of course, the function is also constant on all flats of A_0 and A_1 respectively.

Hence, it suffices for a given Boolean function, first to determine the flats of a "small" dimension t_0 on which the function is constant and then to combine these spaces to get those flats of dimension m on which the function is affine.

So the general structure of the algorithm can be described as follows:

Algorithm 2.1

Input: a Boolean function $f : \mathbb{F}_2^n \to \mathbb{F}_2$, a starting dimension t_0
Output: a list of all flats on which f is affine

For all subspaces U of $\mathbb{F}_2^n$ with $\dim(U) = t_0$ do

    Determine all flats $a + U$ with $f|_{a+U} = 0$ and $f|_{a+U} = 1$ resp.

Combine pairs (a_1+U, a_2+U) with $f|_{a_1+U} = f|_{a_2+U} = 0$ and $f|_{a_1+U} = f|_{a_2+U} = 1$ resp.
to get flats $a_1+\tilde{U} = a_1 + < U, a_1+a_2 >$ of dimension t_0+1
with $f|_{a_1+\bar{U}} = 0$ and $f|_{a_1+\bar{U}} = 1$ resp.

Repeat the last step for new flats with equal $\tilde{U}$ up to dimension $m \Leftrightarrow 1$

Combine pairs of flats $(a_1+\hat{U}, a_2+\hat{U})$ with $\dim(\hat{U}) = m \Leftrightarrow 1$
(independent of whether $f|_{a_i+\hat{U}}$ is 0 or 1)
to get those flats of dimension m on which f is affine

Output these flats of dimension m

To implement this algorithm efficiently and prove the correctness of the optimized version, we first have to make some definitions.

3 Definitions and Notations

In this article we represent vectors $u \in \mathbb{F}_2^n$ as n-tuples $u = (u_1, \ldots, u_n)$, $u_i \in \mathbb{F}_2$, we denote the index of the leftmost 1 in this representation by

$$\nu(u) := \max\{i \in \{1, \ldots, n+1\} \mid u_j = 0 \text{ for } 1 \le j < i\}$$

and for a vector space $U \subseteq \mathbb{F}_2^n$ we define $\Upsilon(U) := \{\nu(u) | u \in U \backslash \{0\}\}$.
By using the standard lexicographical ordering $<$ on $\mathbb{F}_2^n$, i.e.

$$u > v \Leftrightarrow (\nu(u) < \nu(v) \quad \text{or} \quad (\nu(u) = \nu(v) \quad \text{and} \quad ((u_{\nu(u)+1}, \ldots, u_n) > (v_{\nu(v)+1}, \ldots, v_n))$$

we can define a unique representation of subspaces $U \subseteq \mathbb{F}_2^n$:

Definition 3.1
An ordered basis $u_1, \ldots, u_k \in \mathbb{F}_2^n$ of U is called a Gauss-Jordan basis (GJB) if

$$u_1 > \ldots > u_k \qquad \textit{and} \qquad (u_j)_{\nu(u_i)} = 0 \quad \forall\ i \neq j.$$

Using this lexicographical ordering is also very efficient for implementations as it corresponds directly to the natural ordering on the integers we get by considering $(u_1, \ldots, u_n)$ as binary representation of $\sum_{i=1}^{n} u_i \cdot 2^{n-i}$.

Lemma 3.2
For each vector space $U \subseteq \mathbb{F}_2^n$ there is one unique GJB.

Proof:
Will be included in the full version of this article. ■

With the notation of $\nu(u)$ we can also define the complement $\bar{U}$ of a vector space U as

$$\bar{U} := \{a \in \mathbb{F}_2^n \mid a_i = 0\ \forall i \in \Upsilon(U)\}$$

and it is obvious that $U \cap \bar{U} = \{0\}$ and thus $U \oplus \bar{U} = \mathbb{F}_2^n$ because of dimensional reasons. So all flats of the form $a+U$ can be uniquely represented as $\bar{a}+U$ with $\bar{a} \in \bar{U}$.

4 Details of the Algorithm

The main data structure of the presented algorithm is a list of all flats of the form $a+U$ (for a given U) on which the given function f is constant:

Definition 4.1
Let $f:\mathbb{F}_2^n \to \mathbb{F}_2$, $u_1,\ldots,u_k \in \mathbb{F}_2^n$ *and* $c \in \{0,1\}$.
If $(u_1,\ldots,u_k)$ *is a GJB of* U *then let*

$$\mathcal{C}_c^{u_1,\ldots,u_k}(f) := \left\{ a \in \bar{U} \mid \quad f|_{a+U} = c \right\}$$

and $\mathcal{C}_c^{u_1,\ldots,u_k}(f) := \emptyset$ *otherwise.*

Using the ideas of Section 2 and the notation of a GJB in order to get each flat only once, we obtain the following relation between lists belonging to different dimensional spaces:

Lemma 4.2
For $f, u_1,\ldots,u_k, c$ *as in Definition 4.1 and for all* $a,b \in \mathbb{F}_2^n$ *the following equivalence holds:*

$$\left.\begin{array}{l} a,b \in \mathcal{C}_c^{u_1,\ldots,u_k}(f) \\ a<b, \qquad a+b<u_k \\ u_{i,\nu(a+b)} = 0 \text{ for } 1 \le i \le k \end{array}\right\} \quad \Leftrightarrow \quad a \in \mathcal{C}_c^{u_1,\ldots,u_k,a+b}(f)$$

Proof:
Will be included in the full version of this article. ■

As for every $a \in \mathcal{C}_c^{u_1,\ldots,u_{k+1}}(f)$ we can write $b = a + u_{k+1}$ such that $a \in \mathcal{C}_c^{u_1,\ldots,u_k,a+b}(f)$ this lemma gives a criterion on how to determine all $\mathcal{C}_c^{u_1,\ldots,u_{k+1}}(f)$ for different u_{k+1} if we know $\mathcal{C}_c^{u_1,\ldots,u_k}(f)$.

This can be done even more efficiently by using the following two ideas:
We can avoid the $a<b$ checks and many $a+b<u_k$ checks by storing the elements of $\mathcal{C}$ in a sorted list. Checking $u_{i,\nu(a+b)}=0$ can be done more efficiently if we once evaluate $\hat{u} := \bigvee_{i=1}^{k} u_i$ (where $\bigvee$ means that componentwise OR of the vectors u_i, i.e. $\hat{u}_j = \max_{i=1}^{k}((u_i)_j)$) and then only check if $\hat{u}_{\nu(a+b)} = 0$.

Another useful criterion to make the computation more efficient is given by the following corollary:

Corollary 4.3
For $f:\mathbb{F}_2^n \to \mathbb{F}_2$, $u_1,\ldots,u_k \in \mathbb{F}_2^n$, $c \in \{0,1\}$ *and* $l \in \{1,\ldots,k \Leftrightarrow 1\}$*:*

$$\left|\mathcal{C}_c^{u_1,\ldots,u_{k-l}}(f)\right| \quad \ge \quad 2^l \cdot \left|\mathcal{C}_c^{u_1,\ldots,u_k}(f)\right|$$

Proof:
Will be included in the full version of this article. ■

Similar to Lemma 4.2 we get the following relations between the flats of dimension m on which f is affine and the lists $\mathcal{C}_c^{u_1,\ldots,u_{m-1}}(f)$ corresponding to dimension $m \Leftrightarrow 1$:

Lemma 4.4
Let $a+U \subset \mathbb{F}_2^n$ be a flat of dimension m. Then the following statements are equivalent:

i) $f|_{a+U}$ *is affine*

ii) $f|_{a+U}$ *is constant*

$$\left.\begin{array}{ll} & \exists \text{subspace } U' \subset U : \dim(U') = m \Leftrightarrow 1 \\ \text{or} & \exists \tilde{u} \in U \backslash U' : U = U' \dot\cup (\tilde{u} + U') \\ & \exists c \in \{0,1\} \end{array}\right\} \text{such that} \left\{\begin{array}{l} f|_{a+U'} = c \\ f|_{a+\tilde{u}+U'} = 1 \Leftrightarrow c \end{array}\right.$$

$$\left.\begin{array}{l} \exists \text{subspace } U' \subset U : \dim(U') = m \Leftrightarrow 1 \\ \quad \text{with GJB } u_1, \ldots, u_{m-1} \\ \exists a' \in a+U', b' \in (a+U) \backslash (a+U') \end{array}\right\} \text{such that} \qquad a', b' \in \bigcup_{c \in \{0,1\}} \mathcal{C}_c^{u_1,\ldots,u_{m-1}}(f)$$

Proof:
Will be included in the full version of this article. ■

This lemma shows that, in order to find all flats on which f is affine, it suffices to compute the lists $\mathcal{C}_c^{u_1,\ldots,u_{m-1}}$ for GJBes of all subspaces of dimension $m \Leftrightarrow 1$.

Together with Corollary 4.3 we can conclude that if we have computed $\mathcal{C}_c^{u_1,\ldots,u_k}(f)$, $c \in \{0,1\}$, we only have to consider pairs of elements of these lists if

$$|\mathcal{C}_c^{u_1,\ldots,u_k}(f)| \geq 2^{m-k} \qquad \text{or} \qquad \left(|\mathcal{C}_c^{u_1,\ldots,u_k}(f)| \geq 2^{m-k-1} \quad \text{and} \quad \left|\mathcal{C}_{1-c}^{u_1,\ldots,u_k}(f)\right| \geq 2^{m-k-1}\right),$$

because otherwise there is no chance to find a flat on which f is affine by considering lists of the form $\mathcal{C}_c^{u_1,\ldots,u_k,\tilde{u}_{k+1},\ldots,\tilde{u}_{m-1}}(f)$.

As described in Section 2 the main idea of the algorithm is to begin with a starting dimension t_0 and to compute the lists $\mathcal{C}_c^{u_1,\ldots,u_{t_0}}(f)$ which we need just by enumerating all corresponding flats and checking directly. Then the lists corresponding to higher dimensions can be generated recursively as described in Lemma 4.2.

So what we need to complete the algorithm is an efficient way to enumerate all initial parts $u_1, \ldots, u_{t_0}$ of GJBes of subspaces of dimension $m \Leftrightarrow 1$.

If we take a look at the definition of a GJB it is obvious that this can easily be done by looping over all increasing sequences $1 \leq \nu_1 < \nu_2 < \ldots < \nu_{t_0} \leq m+1+t_0$ and all integers $z_{i,j} \in \left\{0, \ldots, 2^{\nu_{j+1}-\nu_j-1} \Leftrightarrow 1\right\}$ with $1 \leq i \leq t_0$, $i \leq j \leq t_0$ and defining
$(u_i)_j = \left\{\begin{array}{ll} 0 & \text{if } j < \nu_i \text{ or } j \in \{\nu_{i+1}, \ldots, \nu_{t_0}\} \\ 1 & \text{if } j = \nu_i \end{array}\right.$ and filling in the gaps with the binary representations $\langle z_{i,j} \rangle_2$ of the integers $z_{i,j}$ as shown in the following scheme:

$$\begin{array}{rcccccccccccc} & & 1 & \ldots & \nu_1 & \ldots & \nu_2 & \ldots & \nu_3 & \ldots & \nu_{t_0} & \ldots & n \\ u_1 & = & 0 & \ldots & 1 & \langle z_{1,1} \rangle_2 & 0 & \langle z_{1,2} \rangle_2 & 0 & \ldots & 0 & \langle z_{1,t_0} \rangle_2 & \\ u_2 & = & 0 & \ldots & & \ldots & 1 & \langle z_{2,2} \rangle_2 & 0 & \ldots & 0 & \langle z_{2,t_0} \rangle_2 & \\ u_3 & = & 0 & \ldots & & & & \ldots & 1 & \ldots & 0 & \langle z_{3,t_0} \rangle_2 & \\ & & & & & & & & & \ddots & \vdots & \vdots & \\ u_{t_0} & = & 0 & \ldots & & & & & & \ldots & 1 & \langle z_{t_0,t_0} \rangle_2 & \end{array}$$

Additionally we only have to consider such sets $u_1, \ldots, u_{t_0}$ for which the hamming weight of $\left(\bigvee_{i=1}^{t_0} u_i\right)_{\nu_{t_0}+1,\ldots,n}$ is not greater than $m \Leftrightarrow \nu_{t_0} + 1 + t_0$, as otherwise it cannot be completed to a GJB of dimension $m \Leftrightarrow 1$.

Finally we just have to enumerate all $a \in \bar{U}$ for $U = \langle u_1, \ldots, u_{t_0} \rangle$. This can be done similarly to the enumeration of the u_i themselves just by defining $a_{\nu_i} = 0$ for $i = 1, \ldots, t_0$ and filling in the gaps with all possible binary representations of integers.

So the whole algorithm can be described as follows (some of the ideas described above to make the algorithm even more efficient — e.g. storing the $\mathcal{C}$s in sorted order — are omitted in order to make this description more readable, but they are easily implemented into this algorithm):

Algorithm 4.5

Input: a Boolean function $f : \mathbb{F}_2^n \to \mathbb{F}_2$, a starting dimension t_0

Output: a list of all flats on which f is affine

for all GJBes $u_1, \ldots, u_{t_0}$ with $hammingweight\left(\left(\bigvee_{i=1}^{t_0} u_i\right)_{\nu_{t_0}+1,\ldots,n}\right) \leq m \Leftrightarrow \nu_{t_0} + 1 + t_0$ do

for all $a \in \overline{\langle u_1, \ldots, u_{t_0} \rangle}$ do

if $f(a + \sum \lambda_i \cdot u_i) = c \quad \forall\ \lambda \in \mathbb{F}_2^{t_0}$ then append a to $\mathcal{C}_c^{u_1,\ldots,u_{t_0}}$

Combine$(\mathcal{C}_0^{u_1,\ldots,u_{t_0}}, \mathcal{C}_1^{u_1,\ldots,u_{t_0}}, (u_1, \ldots, u_{t_0}), t_0)$

using the recursive subroutine

Combine$(\mathcal{C}_0, \mathcal{C}_1, (u_1, \ldots, u_k), k)$:

if $(k < m \Leftrightarrow 1)$

then if $\left(|\mathcal{C}_0| < 2^{m-k-1}$ or $\left(|\mathcal{C}_0| < 2^{m-k}$ and $|\mathcal{C}_1| < 2^{m-k-1}\right)\right)$ then $\mathcal{C}_0 := \emptyset$

if $\left(|\mathcal{C}_1| < 2^{m-k-1}$ or $\left(|\mathcal{C}_1| < 2^{m-k}$ and $|\mathcal{C}_0| < 2^{m-k-1}\right)\right)$ then $\mathcal{C}_1 := \emptyset$

$\hat{u} := \bigvee_{i=1}^{k} u_i$

for all $c \in \{0, 1\}, \quad a, b \in \mathcal{C}_c : a < b$ do

if $\left(\hat{u}_{\nu(a+b)} = 0$ and $a + b < u_k\right)$ then append a to $\mathcal{C}_c^{u_1,\ldots,u_k,a+b}$

for all $u_{k+1} \in \mathbb{F}_2^n : u_{k+1} < u_k$ do

Combine$(\mathcal{C}_0^{u_1,\ldots,u_{k+1}}, \mathcal{C}_1^{u_1,\ldots,u_{k+1}}, (u_1, \ldots, u_{k+1}), k + 1)$

else for all $a, b \in \mathcal{C}_0 \cup \mathcal{C}_1 : a < b$ do

output "f is affine on $a + \langle u_1, \ldots, u_k, a + b \rangle$"

In order to choose an optimal starting dimension t_0 we have to take a closer look at some complexity evaluations.

5 Complexity Evaluations

In this section we will evaluate the complexity of the described algorithm, especially in dependence on the chosen starting dimension t_0. This will then lead to a suggestion on how to optimally choose t_0.

In order to be able to make a proper complexity evaluation we have to assume that f is a random Boolean function. We will then evaluate the *expected* complexity of the algorithm.

The time complexity evaluations will be split into two parts, the complexity of the "exhaustive search" part in the main loop and the recursive "combining" part:

Exhaustive search:
The number of subspaces of dimension t_0 in $\mathbb{F}_2^n$ is $\prod_{i=0}^{t_0-1} \frac{2^{n-i}-1}{2^{t_0-i}-1} \approx 2^{(n-t_0)t_0+1}$, and thus the number of flats of this dimension is about

$$2^{(n-t_0)t_0+1} \cdot 2^{n-t_0} = 2^{(n-t_0)(t_0+1)+1}.$$

As checking whether a function is constant on a given subset needs at most two comparisons and three evaluations of f on average, we expect a complexity of about $2^{(n-t_0)(t_0+1)+2}$ steps in the "exhaustive search" part.

E.g. for $n = 14$ and $n = 16$ this estimation gives the following concrete complexities:

$n = 14$:

t_0	1	2	3	4	5	6	7
$\log_2$(compl.)	28	38	46	52	56	58	58

$n = 16$:

t_0	1	2	3	4	5	6	7	8
$\log_2$(compl.)	32	44	54	62	68	72	74	74

¿¿From these tables we can see that it is not feasible to check normality by pure "exhaustive search" for these choices of n as this obviously corresponds to using the above described algorithm with $t_0 = m$ and that has an expected complexity of about 2^{58} and 2^{74} steps respectively.

Combining:
Let $\mathcal{T}_t$ be the combined expected complexity of all calls of **Combine**$(\ldots, t)$ concerning some dimension t. Then for $t < m \Leftrightarrow 1$ this complexity $\mathcal{T}_t$ mainly depends — besides the complexity $\mathcal{T}_{t+1}$ of further recursive calls of **Combine** — on the average size $\mathcal{S}$ of the inputted lists $\mathcal{C}_0$ and $\mathcal{C}_1$. As the main part of **Combine** is a loop over all unordered pairs of $\mathcal{C}_0$ and $\mathcal{C}_1$ respectively, in which mainly two comparisons are performed, the complexity can be estimated as

$$2 \cdot \binom{\mathcal{S}}{2} \cdot 2 \approx 2 \cdot \mathcal{S}^2.$$

As f is supposed to be random, the expected size $\mathcal{S}_t$ of $\mathcal{C}_c^{u_1,\ldots,u_t}(f)$ (i.e. a list corresponding to a subspace of dimension t) is $\mathcal{S}_t = 2^{-2^t} \cdot 2^{n-t}$, since the probability that $f(x) = c$ for all 2^t elements x in one of the corresponding flats is 2^{-2^t} for a random function f and there are 2^{n-t} flats corresponding to the subspace $\langle u_1, \ldots, u_t \rangle$.

As described in the sections above due to the extra conditions **Combine**$(\ldots, (u_1, \ldots, u_t), t)$ is only called once for each subspace $\langle u_1, \ldots, u_t \rangle$ and as we have a number of $\prod_{i=0}^{t-1} \frac{2^{n-i}-1}{2^{t-i}-1}$

subspaces of dimension t the expected total complexity for all calls of **Combine**(... ,t) concerning some dimension $t < m \Leftrightarrow 1$ is about

$$\mathcal{T}_t = \mathcal{T}_{t+1} + 2 \cdot \mathcal{S}_t^2 \cdot \prod_{i=0}^{t-1} \frac{2^{n-i} \Leftrightarrow 1}{2^{t-i} \Leftrightarrow 1} \quad \Rightarrow \quad \mathcal{T}_t \Leftrightarrow \mathcal{T}_{t+1} \approx \mathcal{S}_t^2 2^{(n-t)t+2} = 2^{-2^{t+1}+(n-t)(t+2)+2}.$$

The expected complexity of one call of **Combine**(... ,$m \Leftrightarrow 1$) should also be about $2 \cdot \mathcal{S}^2$, as in this case we loop over all unordered pairs of $\mathcal{C}_0 \cup \mathcal{C}_1$, which is a set of size $2\mathcal{S}$, but we perform only 1 operation per pair. Thus for dimension $m \Leftrightarrow 1$ we get

$$\mathcal{T}_{m-1} \approx 2^{-2^m+(n-m+1)(m+1)+2}.$$

Finally we can say that the expected total complexity $\mathcal{T}_{t_0}$ of all calls of **Combine** in the main loop of the algorithm can be written as

$$\mathcal{T}_{t_0} = \sum_{t=t_0}^{m-2} (\mathcal{T}_t \Leftrightarrow \mathcal{T}_{t+1}) + \mathcal{T}_{m-1} \approx \sum_{t=t_0}^{m-1} 2^{-2^{t+1}+(n-t)(t+2)+2}.$$

As before for the "exhaustive search" part, for the "combining" part we get the following exemplary complexities for $n = 14, n = 16$:

$n = 14$:

t_0	1	2	3	4	5
$\log_2(\mathcal{T}_{t_0})$	43	43	41	30	1

$n = 16$:

t_0	1	2	3	4	5
$\log_2(\mathcal{T}_{t_0})$	52	52	51	42	15

Combined with the table of the complexities for the "exhaustive search" part this table shows that for $n = 14$ and $n = 16$ a proper choice for the starting dimension seems to be $t_0 = 2$ or $t_0 = 3$.

Obviously in the complexity evaluation described so far, we have not taken into account the restrictions on the hammingweight of the vectors in the GJBes in the main loop and the if-statements concerning $|\mathcal{C}_c|$, which are very hard to analyze exactly. But these tweaks on the algorithm should have not much influence on the choice of t_0 and, of course, they only decrease the complexity of the algorithm such that the above described complexities can be seen as estimations of "upper bounds" on the complexity of the algorithm.

An actual implementation of the algorithm which we made on a Pentium IV with 1,5 GHz in C++, needed about 50 hours for $n = 14$ and $t_0 = 3$.

6 Applications

6.1 Checking Normality

The first application is to check whether a given Boolean function is (weakly) normal or not.

In order to check weak normality with the algorithm we can do the following: We just run the algorithm with the given function and in the case that we come to the point where

the algorithm would output "f is affine on ...", we stop the execution and output that f is weakly-normal. In the case that the algorithm does not output anything, we know that f is non weakly-normal.

In order to only check normality we just change the else-part of the **Combine** function from looping over all pairs in $\mathcal{C}_0 \cup \mathcal{C}_1$ to looping over all pairs in $\mathcal{C}_0$ and $\mathcal{C}_1$ separately and we may also change the if-statements concerning $|\mathcal{C}_c|$ accordingly. Then again as in the case of weak normality we just need to check if the algorithm outputs anything (then f is normal and we can stop) or not (then f is non normal).

Checking normality is interesting in particular for bent functions, as it was an open question for several years, if there are non normal or even non weakly-normal bent functions.

With the help of the algorithm presented here, we were able to verify that some explicit examples of bent functions recently found by Dillon and Dobbertin in [DD], are non normal or non weakly-normal respectively. More details on this can be found in [CDDL].

6.2 Maiorana-McFarland Functions

The second application of the algorithm we want to describe here is the problem to decide whether a given bent function is a Maiorana-McFarland bent function.

Definition 6.1 *Let $\pi : \mathbb{F}_2^m \to \mathbb{F}_2^m$ be a permutation and $h : \mathbb{F}_2^m \to \mathbb{F}$ an arbitrary boolean function. Then*

$$f : \mathbb{F}_2^m \times \mathbb{F}_2^m \to \mathbb{F}$$

with

$$f(x, y) = \langle x, \pi(y) \rangle + h(y)$$

is called a Maiorana-McFarland function.

We call MM the class of all functions which are equivalent to a Maiorana-McFarland function under affine transformations.

A result of Dillon which can be found in [Dil] gives a characterization of MM bent functions by stating the following :

Lemma 6.2
Let $f : \mathbb{F}_2^n \to \mathbb{F}_2$ be a bent function. Then the following properties are equivalent:

i) f is a MM bent function.

ii) There exists a subspace U of dimension m such that the derivative of f with respect to every 2-dimensional subspace of U is constant.

Due to the follwing lemma it is possible to use the algorithm described above to determine whether a function is in MM or not.

Lemma 6.3
Let $f : \mathbb{F}_2^n \to \mathbb{F}_2$ be a bent function. The following properties are equivalent:

i) f is a MM bent function.

ii) There exists a subspace U of dimension m such that the function f is affine on every coset of U.

The proof of this Lemma is obvious and as the algorithm described in this paper outputs every coset of dimension m on which f is affine, this property can be checked easily.

In practice this means that for $n = 8$ we can decide whether a bent function is of the MM type in less than a second, for $n = 10$ in less than a minute and even for $n = 14$ in a few days.

The possibility to determine if a given function is of the MM type can be used to compute an experimental bound on the number of bent functions for $n = 8$ as follows:

By generating "random" bent functions and checking whether they are of the MM type as described before, the ratio q of the number of MM-type bent functions to the number of all bent functions can be estimated. Then, if μ_8 is the number of MM-type functions in 8 variables, the number of all bent functions can be estimated as $\frac{1}{q}\mu_8$.

There are two problems that need further research:

First the number μ_8 of MM-functions for $n = 8$ is not known exactly. The MM functions all are affine equivalent to $\langle x, \pi(y)\rangle + h(y)$, where π is a permutation and h an arbitrary Boolean function. The number of functions of this form is $2^{2^m}(2^m!)$. The problem is to determine the length of the orbit under the action of the group $AL(n)$ of all affine transformations. This length is equal to $\#AL(n)$ iff there are no $A \in AL(n)$ such that $f \circ A = f$. We computed the length of the orbit for randomly chosen MM-type functions, but it would be much more satisfying to have a theoretical result.

The second problem is that the generation of bent functions for $n = 8$ usually uses hill-climbing algorithms and this algorithms might find MM-type functions more or less often than they should. A first step to check this can be to determine the above ratio for $n = 6$ and compare it with the proper ratio, which in this case is known (see [Pre]).

References

[CDDL] Canteaut, Anne, Daum, Magnus, Dobbertin, Hans, and Leander, Gregor, "Normal and Non Normal Bent Functions", submitted.

[Dil] Dillon, John F., "Elementary Hadamard Difference-Sets", Ph.D. Thesis, 1974.

[DD] Dillon, John F., and Dobbertin, Hans, "New Cyclic Difference Sets with Singer Parameters", in "Finite Fields and Applications", to appear.

[Dob] Dobbertin, Hans, "Construction of bent functions and balanced Boolean functions with high nonlinearity", Fast Software Encryption (Proceedings of the 1994 Leuven Workshop on Cryptographic Algorithms), LNCS 1008, pp. 61-74, 1995.

[Pre] Preneel, Bart, "Analysis and Design of Cryptographic Hash Functions", Ph.D. Thesis, 1993.