Computing the Kernel of a Non-Linear Code

Mike LeVan and Kevin T. Phelps
Department of Discrete and Statistical Sciences
Auburn University
Auburn, Alabama
USA 36849-5307
levanjm@mail.auburn.edu
phelpkt@mail.auburn.edu

ABSTRACT. Given a binary code C, the set, K, of all vectors which leave C invariant under translation is called the kernel of C. The main concern of this paper is the development of an efficient algorithm for computing the kernel of C. We present such an algorithm with run-time $O(|C| \log |C|)$ which is the best possible.

1 Introduction

A binary code C of length n is simply a subset $C \subseteq \{0,1\}^n$. A code C' is a translation of C if there exists a vector $a \in \{0,1\}^n$ such that $C' = C + a = \{u + a | u \in C\}$. The set K, of all vectors which leave C invariant under translation (i.e. $C = C + a, a \in K$) is called the *kernel* of C. The *kernel* K is clearly a linear code, moreover, if we assume the zero vector is in C, then the *kernel* is a linear subcode of C. The main concern of this paper is the development of an efficient algorithm for computing the *kernel* of a binary code. For convenience, we will always assume a code C contains the zero vector and is binary.

The size of the kernel is in one sense a measure of the linearity of a code. A related concept is that of maximal linear subcodes of a given code C. We define M to be a maximal linear subcode if it is contained in C, linear, and it is not properly contained in any other linear subcode of C. The question of finding a maximal linear subcode, or determining if a code is linear, turns out to be an important building block in our algorithm.

The most obvious method of finding the kernel is the following: Pick a vector $w \in C$. Then add w to every other vector in C and compare to see

JCMCC 20 (1996), pp. 237-241

if the sums are in C as well. By using a binary search, one can determine if a vector is in C in $\log|C|$ steps. So, to determine if the sum of $w+u$ is in C, for each $u \in C$, would take $|C|\log|C|$ steps. To do this process for all $|C|$ vectors in C would take a total of $|C|(|C|\log|C|) = |C|^2\log|C|$ steps. This shall be called the "Brute Force" method. Notice that determining whether a vector w is in the kernel takes $O(|C|\log|C|)$ steps.

2 Preliminary results

As was stated earlier, the kernel is in one sense a measure of the linearity of C. Also, since the kernel is a linear subcode of C, it is contained in a maximal linear subcode, but one can say more. Here are some preliminary results regarding the kernel of the code C.

Lemma 1. *The kernel is equal to the intersection of all maximal linear subcodes of C.*

Proof: Given a kernel K, assume the K is not contained in a maximal linear subcode M. Let $a \in K$, be an element in the kernel which is not in M. Then $M+a \subseteq C$ and $M \cup (M+a)$ is a linear subcode of C containing M, contradicting the fact that M is maximal. Thus $K \subseteq M$, and $K \subseteq \cap M$, M a maximal linear subcode of C.

On the other hand, for each $y \in C$, there exists a maximal linear subcode M_y, which contains y. Thus, for each x which is contained in the intersection of all maximal linear subcodes, and all $y \in C$, we know that $x \in M_y$ and thus $x+y \in M_y \subseteq C$, and $x \in K$. □

Lemma 2 ([1]). *C is equal to the disjoint union of the kernel and cosets of the kernel.*

One consequence of Lemma 2 is that a maximal linear subcode may also be written as a disjoint union of the kernel and cosets of the kernel.

Since the whole idea behind the paper is to find an algorithm which will find the kernel in a reasonable amount of time, an obvious question is what is a reasonable time? Suppose that someone hands you a set of vectors K, claiming it is a *kernel* for a code. If the set contains $|K|$ vectors, how much time will it take to check to see if this set is at least contained in the kernel? Using an argument similar to the brute force argument, one can check; to see if a set, K, is contained in the kernel in $|K|(|C|\log|C|)$ steps. This obviously runs on $O(|C|\log|C|)$ time. So, if one can find an algorithm which runs on $O(|C|\log|C|)$ time, you would expect to have the fastest algorithm available.

Next we consider how to efficiently determine if a code is linear. Certainly if the code is linear, then the kernel is actually the code itself! So finding the kernel can not be any faster than determining if the code is linear. The following is a procedure to tell whether a given code is linear.

Initialize: $\underline{0} \in C$, place $\underline{0}$ in M

Step 1: Pick a word $y \in (C \setminus M)$. Add y to everything in M. If $M + y$ is contained in C, then $M := M \cup M + y$

Step 2: Repeat step 1 until $|C \setminus M| = 0$ *or* any single sum is not an element of C. If $|C \setminus M| = 0$, then C is linear, else C is not.

Again, testing to see if a vector is in C takes $\log |C|$ n-bit comparisons which must be done at most $|M|$ times. But $\Sigma |M| \log |C| = |C| \log |C|$. So to check to see if a code is linear or not requires at most $|C| \log |C|$ comparisons and $|C|$ additions.

3 How to construct a maximal linear subcode

The algorithm to find a maximal linear subcode is just a minor variation of the previous algorithm.

Initialize M_0 to contain $\underline{0}$ and initialize C_0 be to $C \setminus \{0\}$. Then

Repeat

$i := i + 1$

Pick $x \in C_{i-1}$. $C_i := C_{i-1}$

Set $T = \emptyset$.

Repeat

For each $y \in M_{i-1}$. If $x + y \in C_i$, then place $x + y$ in T, and let $C_i := C_i \setminus \{x + y\}$. Until $x + y \notin C$, or $T = M_{i-1} + r$.

$$M_i := \begin{cases} \{M_{i-1}\} \cup T & \text{if } (T = M_{i-1} + x) \subseteq C \text{ and} \\ M_{i-1} & \text{otherwise.} \end{cases}$$

until $|C_i| = 0$.

How many steps does it take to find a maximal linear subcode? For each successive i you perform at most $(|T| + 1)$ additions and $(|T| + 1) \log |C|$ additions and comparisons. Since $|C_{i-1}| - |C_i| = |T| = t_i > 0$, and $\Sigma t_i = |C|$, and thus it will take $O(|C| \log |C|)$ steps to find M.

We know that the kernel is the intersection of all maximal linear subcodes of a code C. We also have a procedure to find a maximal linear subcode, however, it is not efficient to find ALL maximal linear subcodes. However, if M, N are linear subcodes then clearly $M \cap N$ is the kernel for $M \cup N$. This suggests we need only find a covering of C of maximal linear subcodes to find the kernel of C. However, finding a covering could still prove to be highly inefficient. So let us move on to the main algorithm of the paper.

4 Constructing the Kernel of a Code

Suppose that M is a linear subcode containing K, the *kernel* of C. For instance, M must contain K if M is maximal in C. Then for some $x \in C \setminus M$

either $M + x \subseteq C$, and thus $(M \cup M + x)$ is a linear subcode covering that set, *or* $M + x \not\subseteq C$, in which case we can find a linear subcode M' with $K \subseteq M' \subseteq M$ and $|M'| \leq \frac{1}{2}|M|$ such that $M' + x \subseteq C$ and $(M' \cup (M' + x))$ is a linear code covering $M' + x$. Iterating will give us the kernel and a covering. The following is a precise statement of the algorithm:

Construct a maximal linear subcode M_1, and let $C_1 := C \setminus M_1$..

$i := 1$.

Repeat

Pick $x \in C_i$.

Let $M' := M_i$.

For each $y \in M_i$.

if $x + y \notin C$, then $M' := M' \setminus \{y\}$.

$M_{i+1} :=$ a maximal linear subcode of M'.

$C_{i+1} := C_i \setminus (M_{i+1} + x)$.

$i := i + 1$. Until $|C_{i+1}| = 0$.

Then the kernel, K, of the code C is the final maximal linear subcode which was found, namely, M_i. Let us now count the number of steps involved with this algorithm.

First of all, one needs to find a maximal linear subcode of C, and as we noted earlier, this requires $O(|C| \log |C|)$ steps to find. Then, within each loop, one needs to calculate M', and this requires $|M_i| \log |C|$ steps. Also within each loop, one needs to find a maximal linear subcode of M'. This requires at most $c|M'| \log |M'| < c|M_{i+1}| \log |M_{i+1}| < c|M_{i+1}| \log |C|$. We are finished with the algorithm when $|C_i| = \emptyset$, i.e. when the M_i's have covered C, which is obviously a finite number of steps. Again using the fact that $\Sigma |M_i| \leq |C|$, we see that this is an $O(|C| \log |C|)$ algorithm. In fact, careful analysis would suggest at most $8|C| \log |C|$ basic n-bit operations and comparisons.

5 Practical Refinements to the Algorithm

As with any algorithm, there are always certain steps one can take to try to improve the algorithm to make it more practical. For instance, here are just two examples of what could be done during the algorithm. First, when the user is finding $M_i + x$, if the users notices that $M_i + x$ contains many more vectors in C than not in C, the user may want to try the following approach:

Let $y \in M_i + x$, $y \notin C$. Add y to M_i. If $y + w \in C$, for some $w \in C$, then w is obviously not in kernel, and may be removed from M'. This will cut down on the time required to find the next maximal linear subcode.

Secondly, if the user notices that the size of M_i is "relatively small", say

$\log(\log|C|)$, then the user may wish to go ahead and use the brute force method on M_i to tell which vectors in M_i exist in the *kernel*.

Certainly there are many variations one may add to the algorithm. However, as we have already seen, one can't expect to do any better than to run the algorithm faster than on the order of magnitude of $O(|C|\log|C|)$.

References

[1] H. Bauer, B. Ganter, F. Hergert, Algebraic Techniques for Nonlinear Codes, *Combinatorica* **3** (1983), 5–6.

[2] F. Herbert, "Algebraische Methoden fur nichtlineare Codes", Dissertation, Darmstadt, 1985.