Linear Feedback Shit Registers - LFSRs

Linear Feeback shift registers are one of the easiest way to generate seemingly random bits from known bits. The word linear suggests that the algorithm is linear in nature, as in, the next output bit depends linearly on previous bit(s).

A simple way to write LFSR is: $$S_{n} = \sum_{j=1}^{j=n}a_j*S_{n-j}$$ where
$n$ is total numebr of input bits
$S_i$ is the input bits being used
$a_j$ is the value of the cofficient, which in our case is 0 or 1 as all the calculation is modulo 2

This type of approach does generate seeming random bits very fast, but due to its **linear** nature, its cryptanalysis becomes realtively easy.

LFSR were used for stream ciphers and are still used today in algorithms like A5/1, A5/2 for providing over-the-air communication in GSM phones. They are also used for pattern generation for exhaustive testing.

LFSR algorithm:

Linear Feedback Shift Register is a shift register whose input bit is a function of it's previous state.
Let's say we want to generate random bits using n-bit LFSR with seed $ S = S_1, S_2, ..., S_n$. Some fixed values lets say $a_1, a_2,... , a_n$ are used to compute the next input bit (these $a_i$'s are also called feedback polynomial or combination coefficients). Next input bit is given by the formula: $$S_{n} = \sum_{j=1}^{j=n}a_j*S_{n-j}$$ After computing the next bit ($S_N$), the rightmost bit is of the input is considered as *output bit* and the newly computed bit is attached to the left and the process is repated **k** number of times (called steps or cycles). This gives the *$k$* bits of output which is than used in cryptographic systems or for some other purpose.

During every such step, the $a_i$'s remain the same, as the next input bit is linearly dependent on the previous bits thorough $a_i$'s. Given sufficient number of output bits, we can successfully guess a minimal feedback polynomial and minimal size of LFSR such that we can successfully generate the same random bits.

So we need to find the $a_i$'s and the minimal value $n$. This is where Berlekamp - Massey algorithm comes into play. It's helps in finding $a_i$'s in $O(n^2)$.

Berlekamp – Massey Algorithm

The Berlekamp–Massey algorithm is an algorithm that will find the shortest linear feedback shift register (LFSR) for a given binary output sequence.
This algorithm starts with the assumption that the length of the LSFR is $l = 1$, and then *iteratively* tries to generate the known sequence and if it succeeds, everything is well, if not, $l$ must be *increased.*

To solve a set of linear equations of the form $S_i+v+Λ_1S_i+ v−1 + ... + Λ_{v−1}S_{i + 1} + Λ_vS_i=0$, a potential instance of $Λ$ is constructed step by step. Let $C$ denote such a potential candidate, it is sometimes also called the "connection polynomial" or the "error locator polynomial" for L errors, and defined as $C = c_LX_L + c_{L−1}X_{L−1} + ...+ c_2X_2 + c_1X + 1$. The goal of the Berlekemp - Massey algorithm is to now determine the minimal degree $L$ and to construct $C$ such, that $S_n+c_1S_{n−1} + ...

+ c_LS_{n−L}= 0, \forall L≤n≤N−1$, where N is the total number of syndromes, and n is the index variable used to loop through the syndromes from 0 to N−1.

With each iteration, the algorithm calculates the **discrepancy** d between the candidate and the actual feedback polynomial: $$ d = S_k+c_1S_{k−1}+ ... + c_LS_{k−L} $$ If the discrepancy is zero, the algorithm assumes that C is a valid candidate and continues. Else, if $d≠0$, the candidate C must be adjusted such, that a recalculation of the discrepancy would lead to $d = 0$. This re-adjustments is done as follows: $$ C= C− (d/b)X^mB $$ where,
B is a copy of the *last candidate* C since L was updated,
b a copy of the *last discrepancy* since L was updated,
and the multiplication by X^m is but an index shift.
The new discrepancy can now easily be computed as $d = d−(d/b)b = d−d = 0$. This above algorithm can further be simplified for modulo 2 case. See Wiki.

Background

LFSRs were used in stream ciphers in early years of internet. Later on, Berlekamp published a paper that talked about an algorithm to decode BCH codes. Later on, James Massey recognized its application to LFSRs and simplified the algorithm.

Application of LFSRs

LFSRs have been extensively used in stream cipher in low processing devices where speed is the need. They have also been used in combiner generator such as Geffe generator, Shrinking generator or any non-linear combination of multiple LFSR generated bits.

Geffe Generator

The Geffe generator consists of three LFSRs: LFSR-1, LFSR-2 and LFSR-3. If we denote the outputs of these registers by x_1, x_2 and x_3, respectively, then the Boolean function that combines the three registers to provide the generator output is given by $$ F(x_1, x_2, x_3) = (x_1 \land x_2) \oplus (\lnot x_1 \land x_2) $$ There are $2^3 = 8$ possible values for the outputs of the three registers, and the value of this combining function for each of them is shown in the table below:

x_1	x_2	x_3	$F(x_1, x_2, x_3)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Consider the output of the third register, x_3. The table above makes it clear that of the 8 possible outputs of x 3 {\displaystyle x_{3}} x_{3}. 6 of them are equal to the corresponding value of the generator output, F (x 1 , x 2 , x 3) {\displaystyle F(x_{1},x_{2},x_{3})} F(x_{1},x_{2},x_{3}), i.e. x 3 = F (x 1 , x 2 , x 3) {\displaystyle x_{3}=F(x_{1},x_{2},x_{3})} x_{3}=F(x_{1},x_{2},x_{3}) in 75% of all possible cases. Thus we say that LFSR-3 is correlated with the generator. This is a weakness we may exploit as follows:

Our work.

We have encoded the **seed** finding problem into Z3Prover and tried and recovered the seed for given feedback polynomial. We began by coding 16-bit Fibonacci LFSR and than general n-bit LFSR. This was not much hard as it is a fairly simple algorithm.

Later on we tried and understood the Berlekamp-Massey algorithm. The original algo of Berlekamp is very notation intensive. And a bit non trivial to be understood on first try. Massey's extenstion was also a bit notation intensive but by that time we had become familiar with many of the symbols. Reading from multiple blogs and some slides of professors, we successfully understood the algorithm probably in 3-4 days. (Yeah it is a bit much but it was worth it.)

Modelling

Than we tried coding it in python and with proper understanding of the algorithm it was not much effort to test and debug. Encoding it in Z3 was a challenge.

After couple of trial and erros and after much brainstorming we modeled lfsr. Berlekamp Massey algorithm successfully pridicted 2048-bit seed within 1 second.

Geffe's generator can predict three 20 bit LFSRs in 197.15s using only 256 output bits. 20 bit 128 known output in 1638s

References:

- Wikipedia - Berlekamp Massey Algorithm
- This Blog Post
- BMA - IIT Kharagpur
- Wikipedia - Correlation Attack