The Theory of Rep-Rate Pattern Generation in the SNS Timing System

E. Bjorklund

Abstract

In this Tech Report, we discuss the theoretical basis behind the algorithms that compute the rep-rate patterns used in the SNS timing system. Particular attention is given to the computational complexity of the algorithms and the quality of the patterns that they produce. A simple and efficient algorithm is presented that solves the general problem of distributing *n* pulses over *m* "timing slots" in the most even way possible, even though *n* may not necessarily be an even divisor of *m*. We next consider the problem of maintaining pattern evenness when constraints are introduced into the system that limit which timing slots (or "cycles") are available for assignment. Algorithms are presented for both the "ideally constrained" case (in which an ideal pulse distribution can be mapped into the constrained list), and the non-ideally constrained case (in which it can not).

1 Introduction

Some components of the SNS accelerator, such as the high-voltage power supplies, prefer to run at repetition rates that are as evenly distributed over time as possible. Other components, such as the diagnostic "NADs", are not as sensitive to the evenness of their timing pulses, but do require a minimum separation between pulses that could be violated if the pattern were too uneven. The strategy of the SNS timing system is to distribute the timing patterns as evenly as possible over the 10-second (600 pulse) super-cycle.

The problem is trivial, of course, when the repetition rate (n) evenly divides 600. It is easy to see, for example, what the pattern should be for n=100 (10.0 Hz.). The optimal pattern is not so obvious, however, when n=87 (8.7 Hz.). The problem is further complicated by system dependencies and constraints (e.g. Gate A may only occur on the same cycle as Gate B, but must not be coincident with Gate C). These constraints will limit the number of timing slots on which a gate may legally occur. Algorithms for three cases are discussed in this tech-note:

1) The simple, or "unconstrained" case, in which a gate may legally occur in any slot. An efficient (linear time) algorithm is presented which can distribute any n pulses over m slots ($n<m$) as evenly as possible.
2) The "ideally constrained" case, in which not all of the available slots are "legal", however a simple rotation of the ideal pattern produced in case 1 will fit within the slots that are legal. An $\mathcal{O}(m^2)$ algorithm is presented for this case where m is the number of slots in the pattern.
3) The "non-ideally constrained" case, in which the ideal pattern must be permuted or "deformed" in order to fit the legal slots. Several heuristic algorithms are presented which range in complexity from roughly $\mathcal{O}(m)$ to $\mathcal{O}(m^5)$.

The result of each of these algorithms will be a "gate bitmap", m bits long, indicating for which machine cycles the gate is enabled, and for which cycles it is not enabled. The bitmap represents an interval of time (in the SNS case, 10 seconds) and is repeated

continuously. We are not concerned in this report with the details of scheduling a gate within the timeline of a given cycle (i.e. when does it come on, for how long, etc.).

2 The Unconstrained Case

Assume that the bitmap is initially filled with 0's (gate off), and that we wish to evenly distribute n 1's (gate on) throughout it. Let us further assume, without loss of generality, that $n \le m/2$ where m is the number of slots in the bitmap (super-cycle size)[1]. For illustration, let us take the example of distributing five pulses over thirteen cycles as evenly as possible. Notice that at the start, we have two optimally distributed strings – one string of all 0's and another string of all 1's.

00000000 11111

We start by simply dividing the five 1's into the eight 0's. This will put a 1 after every 0, with three 0's left over.

01 01 01 01 01 000

Once again, we can think of this as two optimally distributed strings. One strings is ten bits long and contains five "01" pairs (an optimal distribution of five pulses over ten slots). The second string contains three 0's. If we now distribute the three 0's over the five "01" pairs, we get three "010" triples with two "01" pairs left over.

010 010 010 01 01

We still have two optimally distributed strings. "010010010" is an optimal distribution of three pulses over nine slots and "0101" is an optimal distribution of two pulses over four slots. We now repeat the process, dividing the two "01" strings into the three "010" strings. This gives us two five-bit strings ("01001") with one three-bit string ("010") as a remainder.

01001 01001 010

We can stop the process when the remainder reaches one or zero. Our final pattern, therefore, is:

0100101001010

which is as evenly as we can distribute five pulses over thirteen slots.

It might be argued that the string would be more optimally distributed if we put the final remainder pattern ("010") between the two "01001" patterns (giving "0100101001001") instead of just tacking it on at the end. The net result, however, is just a rotation of the first pattern. If the pattern is to be repeated *ad nauseam* by the timing system, then there is no real advantage (other than aesthetic) to be gained by the rotation.

Now that we know the "optimal pattern", we would like a method for efficiently computing it. By examining the final string in the above example, we discover that it has

[1] Note that if $n > m/2$, the problem can be turned around by starting with all 1's and distributing a pattern of $m-n$ 0's.

a recursive structure. The final, or top-level, string is composed of two strings from the previous level, plus a remainder from the level before that. This structure is shown graphically below:

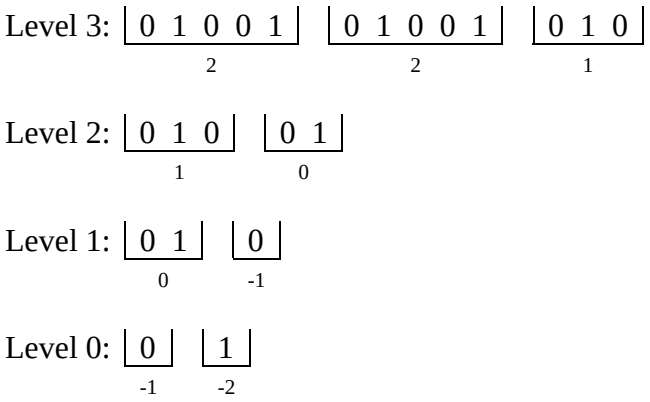

Figure 1

In general, a string at level ℓ will be composed of one or more strings at level $\ell - 1$, and at most one string at level $\ell - 2$. This implies that the string could be computed by a recursive procedure such as the following:

```
void function build_string (int level)
{
    if (level == -1)
        append a "0" to the end of the bitmap;

    else if (level == -2)
        append a "1" to the end of the bitmap;

    else {
        for (i=0; i < count[level]; i++)
            build_string (level-1);
        if (remainder[level] != 0)
            build_string (level-2);
    }/*end else*/
}
```

Figure 2

In the above algorithm (and in figure 1), the levels were chosen so that recursion ends when the level number gets below zero. Level "–1" implies that a 0 should be inserted in the string. Level "–2" implies that a 1 should be inserted in the string. The "`count`" array tells us how many level $\ell - 1$ strings make up a level ℓ string. The "`remainder`" array is used to tell us if the level ℓ string contains a level $\ell - 2$ string. To see how we obtain the values for the "`count`" and "`remainder`" arrays, let us go back and look again at how the strings were constructed.

The level-zero string ("01") was constructed by dividing eight 0's by five 1's. 8/5 = 1 r 3, so we construct a level-zero string by appending one level "–2" string ("1") to one level "–1" string ("0"). Therefore, `count[1]` should be one (since there is only one

level-zero string in the level-one string). We use the `remainder` array both to keep track of the remainder of the previous division (it will become the denominator of the next division), and to determine how many levels deep we need to go. The process stops either when the remainder is zero (we have achieved a completely even distribution) or when the remainder is one (we have reached the end of the remainder series). We should therefore make `remainder[0]=5` (the original dividend) and `remainder[1]=3` (the remainder
of 8/5).

The level-one string ("010") is constructed by dividing the three remaining level "–1" strings (0's) into the five level-zero strings. 5/3 = 1 r 2, so we have three level-one strings with two level-zero strings left over. Therefore `count[1]=1`, `remainder[1]=3` (from the previous step), and `remainder[2]=2`. In similar fashion, the level-two string ("01001") is constructed by dividing the two level-zero strings into the three level-one strings. 3/2 = 1 r 1, so `count[2]=1` and `remainder[3]=1`. Since there is now only one level-one string to distribute between two level-two strings, we can stop with `count[3]=2` and `remainder[3]=1`. This process is shown graphically in figure 3.

				Remainder	Count
Level -1:	00000000	11111			
Level 0:	01 01 01 01 01	0 0 0	8/5 = 1r3	5	1
Level 1:	010 010 010	01 01	5/3 = 1r2	3	1
Level 2:	01001 01001	010	3/2 = 1r1	2	1
Level 3:	0100101001010			1	2

Figure 3

The complete bitmap calculation algorithm is show below in figure 4.

```
void function compute_bitmap (int num_slots, int num_pulses)
{
   /*---------------------
    * First, compute the count and remainder arrays
    */
    divisor = num_slots - num_pulses;
    remainder[0] = num_pulses;
    level = 0;

    do {
        count[level] = divisor / remainder[level];
        remainder[level+1] = mod(divisor, remainder[level]);
        divisor = remainder[level];
        level = level + 1;
    while (remainder[level] > 1);

    count[level] = divisor;

   /*---------------------
    * Now build the bitmap string
    */
    build_string (level);
}
```

Figure 4

Note that because of our assumption that the number of pulses is not greater than half the number of slots, the values in the count array will always be greater than or equal to one.

Although it may not be obvious on casual inspection, the algorithm in figure 4 can be computed in linear time, as shown by the following theorem:

Theorem:
The `compute_bitmap` function shown in figure 4 is computable in $\mathcal{O}(m)$ time, where m is the number of slots in the pattern (the super-cycle size).

Proof of Theorem:
Examining the function in figure 4 reveals two questions we need to answer in order to determine the computational complexity. These are: 1) How many times do we iterate through the loop that populates the `count` and `remainder` arrays? and 2) How many times do we call the recursive `build_string` function? To answer both these questions, we must first determine what the maximum depth of recursion will be.

Let:

m = The number of slots in the bitmap (the super-cycle size)
p = The number of 1's in the bitmap
q = The number of 0's in the bitmap
ℓ = The value of `level` at the end of the `count` and `remainder` computation.

Note that $m = p + q$. As before, we assume that $p \le q$.

It is easy to show that the best-case time for the algorithm is $\mathcal{O}(m)$. At the very least, we must go through the initial loop (the one that computes the `count` and `remainder` array values) one time. This implies that the minimum value for ℓ is 1. We must also call the `build_string` function at least once for each bit in the bitmap. Consequently, there must be at least m calls to the `build_string` function. Since the minimum value for ℓ is 1, however, this implies that there must be at least two additional "overhead" calls to `build_string` before we can reach the –1 and –2 values of ℓ that actually write the bits to the bitmap. This means that `build_string` must be called at least m+2 times.

The maximum value for ℓ will determine the maximum depth of recursion for the `build_string` function. It also determines the maximum number of times you execute the `count` and `remainder` construction loop – which determines the amount of space you need to allocate for the `count` and `remainder` arrays. Clearly, this would be a useful value to have.

From figures 1 and 3, we see that the maximum value of ℓ is determined by the length of the remainder series generated by the values of p and q. Since the longest remainder series are generated by starting with two consecutive Fibonacci numbers, it follows that the maximum value of ℓ will come when m, q, and p are three consecutive Fibonacci numbers such that $m = q + p$.

We have already seen in the case where m=13, q=8, and p=5, that ℓ=3. If we move up the Fibonacci sequence and make m=21, q=13, and p=8, we observe that ℓ will be 4. In

fact, it is fairly easy to show that for any $m \geq 5$ such that $m=F(k)$, where $F(k)$ is the kth Fibonacci number:

$$\begin{aligned} \ell &= F^{-1}(m) - 4 \\ &= k - 4 \end{aligned} \tag{1}$$

or

$$m = F(\ell + 4) \tag{2}$$

If you prefer not to count Fibonacci numbers in order to determine how big your arrays should be, Knuth [1] has given the following analytical solution for $F(i)$:

$$F(i) = \left[\frac{\tau^i}{\sqrt{5}} \right] \quad \text{(Rounded to the nearest integer)} \tag{3}$$

where

$$\tau = \frac{1+\sqrt{5}}{2} \quad \text{(the Golden Ratio)}$$

The maximum ℓ for any m can therefore be determined by:

$$\ell = \log_\tau (m\sqrt{5}) - 4 \tag{4}$$

and the size of the count and remainder arrays (since the indexing starts at 0) is given by:

$$\ell + 1 = \log_\tau (m\sqrt{5}) - 3 \tag{5}$$

We observe from (4) that $\ell < m$, so we conclude that the algorithm's running time is dominated by calls to the `build_string` function and not by the computation of the `count` and `remainder` arrays.

We know that there are exactly m "useful" calls to `build_string` (calls that actually produce a 0 or a 1). We now need to know how many "overhead" calls there are (calls where `level` is non-negative).

We know from figure 4, that there is exactly one call to `build_string` at level ℓ. We also know that the greatest number of "overhead" calls will occur at the maximum value for ℓ, which occurs when m, q, and p are consecutive Fibonacci numbers. We therefore know that for $m \geq 5$, the `count` and `remainder` loop will terminate after a division of 3 by 2 produces a remainder of 1 in `remainder[`ℓ`]`. From this we know that `count[`ℓ`]` is 2, which means there are exactly two calls to `build_string` at level $\ell - 1$. We also know that at levels below $\ell - 1$, the result of the division will always be 1, and that there will always be a remainder > 1. This means that for $0 \leq k < \ell$, each call to `build_string` at level k will call `build_string`(k–1) exactly once and `build_string`(k–2) exactly once. It follows then that for $k < \ell - 1$, the total number of calls to `build_string` at level k is equal to the total number of calls to `build_string` at level k+1 plus the total number of calls to `build_string` at level k+2.

If we define the function $N(k)$ to be the number of calls to `build_string` at level k, we get the following series:

$$N(\ell) = 1$$
$$N(\ell - 1) = 2$$
$$N(k) = N(k+1) + N(k+2) \quad \text{for } k < \ell - 1 \tag{6}$$

What this is, in fact, is another Fibonacci sequence which increases as k decreases. Thus, $N(k)$ can be written as:

$$N(k) = F(\ell - k + 2) \quad \text{for} \quad 0 \le k \le \ell \tag{7}$$

The total number of <u>overhead</u> calls to `build_string` then, is just the sum of all the $N(k)$'s from 0 to ℓ (the "overhead levels").

Let t be the total number of calls ("overhead" plus "useful") to `build_string`. We can now write:

$$\begin{aligned} t &= m + \sum_{k=0}^{\ell} N(k) \\ &= m + \sum_{i=2}^{\ell+2} F(i) \\ &= m + \left(\sum_{i=1}^{\ell+2} F(i) \right) - F(1) \\ &= m + \left(\sum_{i=1}^{\ell+2} F(i) \right) - 1 \end{aligned} \tag{8}$$

It can be shown by induction that:

$$\sum_{i=1}^{n} F(i) = F(n+2) - 1 \tag{9}$$

Which means that we can re-write (8) as:

$$t = m + F(\ell + 4) - 2 \tag{10}$$

We know from equation (2), however, that $F(\ell + 4) = m$. Therefore, the worst-case total number of calls to `build_string` is given by:

$$t = 2(m - 1) \tag{11}$$

We see from (5) that the size of the `count` and `remainder` arrays (and therefore the amount of work required to compute them) is bounded by $\log_{\tau}(m\sqrt{5}) - 3$. We also see from (11) that the number of calls to `build_string` is bounded by $2(m-1)$. We therefore conclude that the `compute_bitmap` function can be computed in $\mathcal{O}(m)$ time. Q.E.D.

3 The Ideally Constrained Case

Sometimes it is not possible to put a pulse exactly where the ideal distribution algorithm says that it should go. There will be various system constraints and dependencies that will dictate which pulses a given timing gate may or may not occur on. We can still preserve our ideal pattern, however, if we can discover a way to rotate the pattern such that it will conform to all the system constraints.

Let us assume that we can produce a list of all the slots on which our gate may legally occur. Let us further assume that if *s* is the number of slots in the legal list, and *n* is the number of pulses we wish to set, then $s > n$. If $s \le n$, then the system is "over-constrained". We can deal with an over-constrained system in a number of ways. Typically we just reduce the number of pulses from *n* to *s* and use the list of legal slots to set the bitmap.

As an example of a system that is not over-constrained, suppose we have three pulses to distribute over eight slots, and that there are only four legal slots (0, 1, 3, and 6) on which these pulses may occur. The "ideal pattern" algorithm from the previous section would want to put the three pulses in slots 1, 4, and 7. The situation is illustrated below in figure 5.

0	1	2	3	4	5	6	7	
0	1	0	0	1	0	0	1	"ideal" pattern
1	1	0	1	0	0	1	0	"legal" pattern

Figure 5

We define the "forward distance", $d(i,j)$, between two slots, *i* and *j*, to be the distance you must travel in a <u>forward</u> direction (possibly wrapping around to the first slot) to get from slot *i* to slot *j*. For example, if there are eight slots in the bitmap, then:

$d(5,5) = 0$
$d(5,6) = 1$
$d(6,5) = 7$

The forward distance function is computed by:

$$d(i, j) = (j - i) \bmod m \tag{12}$$

Note that this requires a definition of the "mod" function that always produces positive results – even when the argument is negative, i.e:

$$(-x) \bmod m = (m - x) \bmod m \tag{13}$$

Let I be the array of slot numbers that contain pulses in the <u>ideal</u> pattern. Let L be the array of <u>legal</u> slot numbers for this gate. In our current example:

$I = (1, 4, 7)$
$L = (0, 1, 3, 6)$

We can now define the "forward distance matrix", *D*, to be an $s \times n$ matrix such that:

$$D_{j,\,k} = d\left(I_k, L_j\right) \tag{14}$$

Figure 6 shows the forward distance matrix for our current example.

	1	4	7	
D =	7	4	1	0
	0	5	2	1
	2	7	4	3
	5	2	7	6

Figure 6

The forward distance matrix gives the forward distance between every pulse in the ideal array (columns) and every pulse in the legal array (rows). Note that no row or column in *D* may contain the same number more than once. This is convenient, since it allows us to treat the rows and columns as sets.

Let C_k be the set of values in the *k*th column of *D*. The elements of C_k represent the forward distances between the *k*th bit in the ideal pattern (*I*) and every bit in the pattern of legal slots (*L*). Let *R* be the set intersection of all the C_k sets.

$$R = \bigcap_{k=0}^{n-1} C_k \tag{15}$$

It follows that there exists a rotation of the ideal pattern which maps directly into the legal pattern if and only if *R* is not empty. Furthermore, if *R* is not empty, then the elements of *R* are the amounts by which the ideal pattern can be rotated to fit the legal pattern. In our current example, *R* = (2, 7). So we can rotate the ideal pattern by either 2 slots or 7 slots and it will fit the legal pattern.

Since set manipulation is not a feature of most programming languages, another useful way to solve the problem is to count how many times each number from 0 to *m*–1 appears in *D*, and then find the number which occurs the most times. If any number occurs *n* times in *D*, then that number will produce a rotation which will fit the legal pattern. Note that since no number may appear more than once in any column of *D*, and since $n < s$, it follows that *n* is the maximum number of times that any number may appear in *D*. Figure 7 shows the "forward distance frequency count" array for our current example. Note that the values 2 and 7 both occur with frequency 3, meaning that a rotation of the ideal pattern by either 2 or 7 slots will completely map it into the legal pattern.

Value =	0	1	2	3	4	5	6	7
Frequency =	1	1	3	0	2	5	0	3

Figure 7

This frequency count array, and its maximum, will prove useful when we consider the non-ideally constrained case in the next section.

The complexity analysis of the ideally-constrained case is fairly straight forward. Computing the ideal pattern requires $\mathcal{O}(m)$ steps. Computing the *D* matrix requires $\mathcal{O}(ns)$ steps. The set intersection operation is at most $\mathcal{O}(ns)$, and determining the maximum number of times any number appears in *D* is also $\mathcal{O}(ns)$. In the worst case the product, *ns*, is bounded by m^2. So the worst case computation time for the ideally-constrained case is $\mathcal{O}(m^2)$.

It might be noted that the above analysis does not include the time to compute the constraint pattern (*L*). The short answer is that the constraint pattern could have come from anywhere and therefore its generation is outside the scope of this paper. Typically, the constraint pattern either is, or is derived from, the rep-rate pattern of another gate on which this gate depends. If this is the case, the cost of the constraint pattern can be ascribed to the cost of computing the antecedent gate and can again be ignored for the purposes of this paper.

4 The Non-Ideally Constrained Case

Suppose that there is no rotation which will completely map the ideal pattern into the legal pattern. Figure 8 is an example of one such situation:

0	1	2	3	4	5	6	7	
0	1	0	0	1	0	0	1	"ideal" pattern
0	1	0	1	1	1	0	0	"legal" pattern

Figure 8

In this case we would like to find a "deformed ideal" pattern, *P*, such that $P \subset L$, $|P|=n$, and the pulse distribution of *P* is as even as possible. There are several ways to accomplish this, with the tradeoff typically being between computational complexity and pattern quality.

We will now explore several techniques for handling the non-ideally constrained case. The system designer can "tune" the performance of the timing system by either choosing one of these techniques or inventing another one.

4.1 Brute Force

"Brute Force" is the only currently known method that will guarantee the most evenly distributed pattern. The technique is simple:

1) Select all possible combinations of *n* slots from the *s* elements of *L*.
2) Choose the one with the most evenly distributed pattern.

Before we start coding, however, it would be instructive to look at the computation time required for this method.

We will start with the second step – choosing the most even pattern. In [2] we developed the general theory of rep-rate pattern "Ugliness". If we define $\delta_j(i)$ to be the forward distance between pulse *i* and the *j*th pulse after *i* in the pattern *P*, i.e.:

$$\delta_j(i) = d\left(P_i,\ P_{(i+j) \bmod n}\right) \tag{16}$$

then the "Ugliness" of pattern *P*, as given by [2] is:

$$Ugliness(P) = \frac{2}{n}\sum_{j=1}^{n/2}\left[\sum_{i=0}^{n-1}\left(\delta_j(i) - \frac{jm}{n}\right)^2\right] \tag{17}$$

This gives us a metric for comparing the patterns that is computable in $\mathcal{O}(n^2)$ time, where *n* is the number of pulses in pattern *P* and *m* is the total size of the super-cycle.

Now we need to address the first step – extracting all the possible *n*-element subsets from *L*. The number of unique *n*-element subsets that can be extracted from a set of size *s* is given by the binomial coefficient:

$$\binom{s}{n} \tag{18}$$

where *s* is the number of pulses in the legal pattern, *L*, and *n* is the rep-rate of the desired pattern. So the total complexity of the "Brute Force" method is given by:

$$\binom{s}{n} n^2 \tag{19}$$

To get a feeling for the worst case time, we first recall that the binomial coefficient can be expressed as:

$$\binom{s}{n} = \frac{s!}{(s-n)!\,n!} \tag{20}$$

We also note that the largest value of the binomial coefficient occurs when *n* is exactly one half the size of *s*, e.g:

$$\binom{s}{s/2} \tag{21}$$

Referring back to (20), we have:

$$\binom{s}{s/2} = \frac{s!}{(s-s/2)!\,(s/2)!} = \frac{s!}{(s/2)!^2} \tag{22}$$

To see how big that is in "real" numbers, we can use Stirling's approximation [3], which states:

$$s! \approx \sqrt{2\pi s}\left(\frac{s}{e}\right)^{s} \tag{23}$$

Substituting (23) back into (22) gives us:

$$\frac{s!}{(s/2)!^{2}} \approx \frac{\sqrt{2\pi s}\left(\frac{s}{e}\right)^{s}}{\left(\sqrt{2\pi\, s/2}\left(\frac{s/2}{e}\right)^{s/2}\right)^{2}}$$

$$\approx \frac{\sqrt{2\pi s}\left(\frac{s}{e}\right)^{s}}{\frac{2\pi s}{2}\left(\frac{s}{2e}\right)^{s}}$$

$$\approx \frac{\left(s^{s}/e^{s}\right)}{\frac{1}{2}\sqrt{2\pi s}\left(s^{s}/2^{s}e^{s}\right)}$$

$$\approx \frac{2^{s}}{\frac{1}{2}\sqrt{2\pi s}}$$

$$\approx \frac{2^{s+1}}{\sqrt{2\pi s}} \tag{24}$$

From (24), we see that the worst case time for the "Brute Force" method is $\mathcal{O}(2^{s+1})$, where s is the number of pulses in the legal pattern (L). Since s could be as large as 600 in the SNS timing system, we conclude that the "Brute Force" method may not be a practical choice.[2]

4.2 Redistribution

We move now from the most expensive algorithm with the best pulse distributions to the least expensive algorithm with (potentially) the worst pulse distributions.

[2] The situation is not quite as bad as it appears. When s=600, we have the "unconstrained" case, which can be solved in linear time. At s=599, s=598, etc. it is unlikely that you will encounter a problem that can't be solved by a simple pattern rotation (the "ideally constrained" case). However, even s=300 yields a sufficiently large value to discourage us from considering the Brute Force method any further.

Recall that the "ideal" pattern (I) was produced by using the `compute_bitmap` algorithm from section 2 to distribute the n desired pulses over the m slots in the super-cycle. The idea behind Redistribution is to use `compute_bitmap` to compute a new "ideal" pattern ($\hat{I}$) by redistributing the n desired pulses over s slots, and then projecting the new pattern onto the set of legal pulses (L).

In the example shown above in figure 8, I = (1, 4, 7) and L = (1, 3, 4, 5). I is the ideal distribution of n=3 pulses over m=8 slots. If, instead, we were to ideally distribute 3 pulses over 4 slots (the size of L), we would get:

$\hat{I}$ = (0, 1, 2), or the string "1110".

Projecting $\hat{I}$ onto L selects the first three elements of L, giving us:

P = (1, 3, 4), or the string "01011000".

This is not the best distribution possible, but neither is it the worst. And it is cheap. $\hat{I}$ can be computed in $\mathcal{O}(s)$ time, and the projection onto L can also be accomplished in $\mathcal{O}(s)$ time. Therefore we say that the redistribution method can be computed worst case in $\mathcal{O}(m)$ time.[3]

A few additional points about the redistribution method are worth making.

1) Redistribution will always produce the most even pattern possible as long as the constraint pattern is perfectly even (i.e. *Ugliness(L)* = 0). As the ugliness of the constraint pattern increases, the ugliness of the projection increases proportionally.
2) Rotations of the redistribution pattern can produce either better or worse projection patterns. One could consider modifying the redistribution method such that we take the projection of all possible rotations and choose the one with the best pattern. This increases the computational complexity to $\mathcal{O}(n^2 s)$ as there are $s-1$ rotations and the ugliness metric is $\mathcal{O}(n^2)$. Since n is bounded by s and s is bounded by m, this gives us a worst-case time of $\mathcal{O}(m^3)$. If we're going to work that hard however, there are other methods that will give us better patterns.

[3] For the rest of this report, we will ignore the fact that it took us $\mathcal{O}(m^2)$ time (worst case) to discover that the problem was "non-ideally constrained" in the first place – and in need of another solution like "redistribution".

4.3 Constraint Pattern Pruning

The next cheapest heuristic is called "Constraint Pattern Pruning". This is the first of a series of "pruning" heuristics that use the individual ugliness metric from [2] to search for better "deformed ideal" patterns.[4] To summarize from [2], the individual ugliness, $u(i)$ of a pulse, i, in pattern, P, is given by:

$$u(i) = \frac{1}{n-1}\sum_{j=1}^{n-1}\left(\delta\ (i) - \overline{\delta(i)}\right)^2 \tag{25}$$

where:

$$\overline{\delta(i)} = \frac{1}{n-1}\sum_{j=1}^{n-1}\delta_j(i) \tag{26}$$

also where n is the rep-rate of the pattern and $\delta_j(i)$ is the forward distance to the jth neighbor of pulse i (as defined above in equation (16)).

The idea behind constraint pattern pruning is that you have a "legal" pulse set, L, of size s and an "ideal" pulse set, I, of size $n < s$. We will ignore I because we know it is unobtainable and concentrate instead on pruning L down to size n. The basic algorithm is:

1) Use equation (25) to identify the ugliest pulse in L. Remove that pulse.
2) Repeat step 1 $s - n$ times.

Returning to our example in figure 8, in which L = (1, 3, 4, 5), if we apply equation (25) to each of the elements of L we get:

$u(0) = {}^2/_3 = 0.\overline{6666}$
$u(1) = 4^2/_3 = 4.\overline{6666}$
$u(2) = 6^2/_9 = 6.\overline{2222}$
$u(3) = 1^5/_9 = 1.\overline{5555}$

The ugliest pulse in L is therefore $L_2 = 4$. Eliminating the pulse in slot 4 gives us the pattern "01010100", which is as even as we can distribute three pulses over the constraint pattern L.

The individual ugliness of a single pulse in L can be computed in $\mathcal{O}(s)$ time. Therefore, the ugliest pulse in L can be found in $\mathcal{O}(s^2)$ time. Trimming the legal pattern down to n pulses requires $s - n$ iterations, so our worst-case compute time for constraint pattern pruning is bounded by $\mathcal{O}(m^3)$.

Constraint pattern pruning produces its best patterns (and works most efficiently) when $s - n$ is small. When s is significantly larger than n, the algorithm is susceptible to local minima. In the following example, m = 60, n = 8, s = 15, and L = (3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59). Figure 9 shows how the constraint pattern pruning

[4] These algorithms could also be called "Survivor" methods, since they work by voting the ugliest pulse out of the pattern.

method reduces *L* from 15 pulses to 8 pulses. The x characters show which pulse was eliminated to produce the pattern at each step.

```
s = 15   ooo1ooo1ooo1ooo1ooo1ooo1ooo1ooo1ooo1ooo1ooo1ooo1ooo1ooo1ooo1
s = 14   oooXooo1ooo1ooo1ooo1ooo1ooo1ooo1ooo1ooo1ooo1ooo1ooo1ooo1ooo1
s = 13   oooooooo1ooo1ooo1ooo1ooo1ooo1oooXooo1ooo1ooo1ooo1ooo1ooo1ooo1
s = 12   oooooooo1ooo1ooo1ooo1ooo1ooo1oooooooo1ooo1ooo1oooXooo1ooo1ooo1
s = 11   oooooooo1ooo1oooXooo1ooo1ooo1oooooooo1ooo1ooo1oooooooo1ooo1ooo1
s = 10   oooooooo1ooo1oooooooo1ooo1ooo1oooooooo1oooXooo1oooooooo1ooo1ooo1
s = 9    oooooooo1ooo1oooooooXooo1ooo1oooooooo1oooooooo1oooooooo1ooo1ooo1
s = 8    oooooooo1ooo1oooooooooooooo1ooo1oooooooo1oooooooo1oooooooo1oooXooo1
final:   oooooooo1ooo1oooooooooooooo1ooo1oooooooo1oooooooo1oooooooo1ooooooo1
```

Figure 9

Because *L* starts out as a perfectly even pattern in this example, we can use redistribution to determine that the most even distribution of 8 pulses over *L* would be something like:

```
oooooooo1oooooooo1oooooooo1oooooooo1oooooooo1oooooooo1oooooooo1ooooo1
```

which is not the final pattern arrived at in figure 9. The algorithm starts out well by eliminating, in order, the pulses in slots 3, 31, 47, 15, and 39. At $s = 10$, however, it starts to go wrong by choosing to eliminate the pulse in slot 19. We would have preferred it to have choosen either 27 or 55.

The pattern at $s = 10$ is actually quite interesting in that, from an ugliness point of view, the pulses in slots 7, 11, 19, 23, 51, and 59 are all equally good candidates. Not knowing *a priori* how we wanted the final pattern to look, the algorithm chose the pulse in slot 19 more or less at random (or more probably because the roundoff error in the calculations for slot 19 was slightly higher than the roundoff error for the other pulses).

This illustrates the need to give our heuristics more of a clue about where we want them to go.

4.4 Add Least Ugly Method

The "add least ugly" method is almost the opposite of "constraint pattern pruning". Instead of deleting the most offensive pulses from a larger pattern, it seeks to add the least egregious pulses to a smaller pattern. It also attempts to incorporate a better picture of what the desired result should look like. The algorithm is:

1) Compute the forward distance frequency array as described above in section 3 (recall that this array counts the frequency of the forward distances between pulses in the ideal set (I) and pulses in the legal set (L)).
2) Pick the value, r, with the highest frequency in the forward distance frequency array. Rotate the ideal pattern by r slots giving the "rotated ideal" pattern, I^r.
3) Let $P_0^r = I^r \bigcap L$ be the intersection of I^r and L. Note that P_0^r has size, $p = |P_0^r| = frequency(r)$, and satisfies $p < n < s$. The elements of P_0^r represent the maximum number of elements from L that match the "rotated ideal" pattern, I^r. This forms the starting point from which we will build our final pattern.

4) Let $C_0^r = L - P_0^r$ be the set of pulses in *L* which are not in P_0^r. C_0^r will be the initial set of pulses (the "candidate" set) from which we can choose to build up the final pattern.
5) Using the individual ugliness metric given in equation (25), select the pulse in C_0^r that would have the lowest individual ugliness value if it were added to P_0^r. Form P_{i+1}^r by adding this pulse to P_i^r and form C_{i+1}^r by removing this pulse from C_i^r.
6) Repeat step 5 above $n - p$ times.

The initial pattern (P_0^r) can be constructed in $\mathcal{O}(s)$ time. The complexity of the individual ugliness metric is $\mathcal{O}(p)$. For each iteration of step 5, we need to compute the individual ugliness of $s - p$ pulses (once for each pulse in C_i^r)[5]. Step 5 is repeated $n - p$ times, so the total number of computations is $\mathcal{O}(s+p(s-p)(n-p))$. Since *p*, *n*, and *s* are all bounded by *m*, the worst-case compute time is $\mathcal{O}(m^3)$.

Almost…

It turns out that the final pattern (P_{n-p}^r) is highly dependent on the rotation pattern (I^r) from which we constructed the initial pattern (P_0^r). For best results, therefore, we need to repeat the algorithm for each possible rotation of *I*. This brings the worst-case time up to $\mathcal{O}(m^4)$. We will talk more about rotations in the next section.

One problem with the "Add Least Ugly" method is that it is sensitive to aliasing. Consider the example where *m* = 45, *n* = 8, *s* = 15, *L* = (2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44) and *I* = (2, 8, 13, 19, 24, 30, 35, 41). In this case, *L* is a perfectly distributed pattern of 15 pulses over 45 slots. *I* is an "ideal", but not perfectly even, distribution of 8 pulses over 45 slots. For purposes of illustration, we will only consider the initial "un-rotated" ideal pattern[6]. Figure 10 shows the bitmaps for *I*, *L*, and the starting pattern, P_0^0.

L = oo1oo1oo1oo1oo1oo1oo1oo1oo1oo1oo1oo1oo1oo1oo1
I = oo1ooooo1oooo1ooooo1oooo1ooooo1oooo1ooooo1ooo
P_0^0 = oo1ooooo1oooooooooooooooooooooooooo1ooooo1ooo

Figure 10

[5] This is a bit of an oversimplification, since the value of *p* increases with each iteration of step 5.
[6] It turns out, that for this example, the initial "un-rotated" pattern is the only one we need to consider. The reason will become clearer in the next section.

The pulses in P_0^0 are "clustered" at the beginning and the end of the pattern. This kind of clustering will cause the algorithm to either create another cluster in the center, or append on to existing clusters. In the example above, the final pattern produced by the "Add Least Ugly" method is:

```
oo1ooooo1oo1oo1oo1oo1oooooooooooooo1ooooo1ooo
```

This is hardly a "shining example" of pattern evenness.

If we modify our algorithm slightly such that step 5 uses the general rather than the individual ugliness metric (equation (17)), then we get a much better final pattern:

```
oo1ooooo1ooooo1ooooo1ooooo1oo1ooooo1ooooo1ooo
```

The general ugliness metric, however, is $\mathcal{O}(p^2)$ whereas the individual ugliness metric is only $\mathcal{O}(p)$. This brings our total worst-case compute time for the "Add Least Ugly" method up to $\mathcal{O}(m^5)$, which may be a bit expensive.

One final observation that we can make from this section is that the individual ugliness metric is generally much better at removing pulses from a pattern than it is at adding them. In fact, it has been our observation that the individual ugliness metric is almost as effective as the general ugliness metric at removing pulses.

4.5 Rotations and Periodicity

There are some steps we can take to reduce the number of rotations we need to consider. First, we claim (without any kind of rigorous proof) that our results will be just as good if we only consider those rotations that correspond to the maximum value found in the forward distance frequency array. Second, we can take advantage of any known "periodicity" of the *I* and *L* patterns.

A good ideal pattern will, by its nature, have a cyclic structure if at all possible. Furthermore, the "periodicity" of an ideal pattern can be determined from its construction. To see this, consider how the ideal patterns are made. Strings at level ℓ are constructed by taking one or more strings from level $\ell - 1$ and appending zero or one string from level $\ell - 2$. In section 2, figure 3, we see an illustration of this process. First we produce a string with five cycles of length two. Then we produce a string with three cycles of length three, then a string with two cycles of length five, and finally a string with a single cycle of length 13. The final string produced in figure 3 is <u>not</u> cyclic, of course, but this is not surprising since 13 is a prime number. The exercise does show, however, that by keeping track of the lengths of the strings at each level, we can determine the periodicity of the final pattern. Figure 11 shows how we might modify the `compute_bitmap` function from figure 4 to keep track of the pattern's periodicity.

```
void function compute_bitmap (int num_slots, int num_pulses)
{
   /*---------------------
    * First, compute the count and remainder arrays
    */
    divisor = num_slots - num_pulses;
    remainder[0] = num_pulses;
    level = 0;
    cycleLength = 1;
    remLength = 1;

    do {
        count[level] = divisor / remainder[level];
        remainder[level+1] = mod(divisor, remainder[level]);
        divisor = remainder[level];
        newLength = (cycleLength * count[level]) + remLength;
        remLength = cycleLength;
        cycleLength = newLength;
        level = level + 1;
    while (remainder[level] > 1);

    count[level] = divisor;
    if (remainder[level] > 0)
        cycleLength = (cycleLength * count[level]) + remLength;

   /*---------------------
    * Now build the bitmap string
    */
    build_string (level);
}
```

Figure 11

If the pattern is cyclic (`cycleLength` < *m*), then we can further reduce our workload by only considering those "maximum frequency" rotations whose rotation amount is less than `cycleLength`. Furthermore, if the constraint pattern is also cyclic, then we only need to consider rotation amounts less than the minimum of the ideal and legal pattern periods.

4.6 Deformed Ideal Pattern Methods

As we have seen in the previous sections, using the ugliness metrics to add or delete pulses one at a time can be expensive and can lead you into local minima and aliasing traps that might be avoided with a more global perspective. One way to increase this global perspective – along with reducing the number of ugliness evaluations – is to start the heuristic with a "deformed" ideal pattern.

A deformed ideal pattern method has the form:

1) Compute the forward distance frequency array as described above in section 3.
2) Pick the value, *r*, that has the highest frequency (*p*) and rotate the ideal pattern by *r* slots giving the "rotated ideal" pattern, I^r.
3) Construct the initial pattern, P_0^r, by matching up each pulse in I^r with the nearest corresponding legal pulse in *L*. If an ideal pulse does not have an obvious closest legal pulse, leave it unassigned.

4) If, as a result of step 3, not all the ideal pulses were assigned to legal pulses, use a method such as "constraint pattern pruning" or "add least ugly" to fill in the missing pulses.
5) Repeat steps 2 through 4 for each rotation, *r*, in the forward distance frequency array such that *frequency*(*r*) = *p*, or until the rotation amount exceeds the periodicity of either the ideal pattern or the legal pattern.

It is fairly obvious how the "add least ugly" method can be applied to a "deformed ideal" pattern. "Constraint pattern pruning" can also be used if you restrict the candidates for pruning to those pulses in *L* which are not also in P_i^r. As before, "add least ugly" works better when you use the general rather than the individual ugliness metric.

The worst-case complexity of the "deformed ideal" methods is basically the same as the worst-case complexity of the "constraint pattern pruning" or "add least ugly" methods. However, by starting with a deformed pattern (which can be computed in $\mathcal{O}(s)$ time), we can achieve a considerable reduction in the average-case complexity. The starting deformed pattern, P_0^r, is highly dependent on the rotation, so it is important not to skip step 5.

4.7 Anti-Pattern Pruning

We have seen in our analysis (and also in our tests) that the "constraint pattern pruning" method usually produces better patterns than the "add least ugly with individual ugliness" method, and that the "add least ugly with global ugliness" method produces better patterns than "constraint pattern pruning", but at a higher cost. This is because the individual ugliness metric is much better at removing ugly pulses from a pattern than it is at adding "desirable" pulses. One way around this problem is to "add" pulses to a deformed ideal pattern by using the individual ugliness metric to "remove" those pulses from the "anti-pattern".

Let P_0^r be the initial deformed ideal pattern for rotation I^r. Define the "Anti-Pattern", $\overline{P}_0^r$ to be:

$$\overline{P}_0^r = \left\{ i \in 0 \ldots m-1 \mid i \notin P_0^r \right\} \tag{27}$$

Because "ugliness" is basically the measurement of the asymmetry of a pattern, it follows that any action that reduces the ugliness of a pattern (makes it more symmetrical) will also reduce the ugliness of the anti-pattern, and vice versa.

The algorithm for the "anti-pattern pruning" method is:

1) Construct an initial deformed ideal pattern, P_0^r, for rotation, *r*, by the method described in the previous section.

2) Construct the anti-pattern, $\overline{P}_0^r$, from P_0^r using equation (27).
3) Let $C_0^r = L - P_0^r$ be the set of candidate pulses to consider for addition to our deformed ideal pattern (note that $C_0^r \subset \overline{P}_0^r$).
4) Using $\overline{P}_0^r$ as the pattern, compute the individual ugliness of each pulse in C_0^r. Select the pulse with the lowest individual ugliness and construct P_{i+1}^r by adding this pulse to P_i^r. Construct $\overline{P}_{i+1}^r$ by removing this pulse from $\overline{P}_i^r$, and construct C_{i+1}^r by removing this pulse from C_i^r.
5) Repeat step 4 $n - p$ times.
6) Repeat steps 1 through 5 above for each rotation, *r*, in the forward distance frequency array such that *frequency*(*r*) = *p*, or until the rotation amount exceeds the periodicity of either the ideal pattern or the legal pattern.

The initial deformed ideal pattern (P_0^r) can be constructed in $\mathcal{O}(s)$ time, and its anti-pattern ($\overline{P}_0^r$) can be constructed in $\mathcal{O}(m)$ time. The "candidate set" (C_0^r) is also computed in $\mathcal{O}(s)$ time, and in fact, can be computed at the same time as P_0^r. The individual ugliness metric requires $\mathcal{O}(m-p)$ time and must be performed for each of the $s - p$ elements of C_i^r in order to select the next pulse to add to P_{i+1}^r. This process must be repeated $n - p$ times to build up the final pattern. The total number of rotations that need to be considered is determined by the forward distance frequency array and the periodicity of *I* and *L*, but is bounded by *n*. Each rotation will also require an $\mathcal{O}(n^2)$ ugliness computation in order to select the best final pattern. The total time for the "anti-pattern pruning" method is therefore $\mathcal{O}(n_r(s+m+(n-p)(s-p)(m-p)+n^2))$ where n_r is the number of rotations considered. This reduces to a worst-case time bounded by $\mathcal{O}(m^4)$.

This is comparable to the running time for the "constraint pattern pruning" method using deformed ideal patterns. "Anti-pattern pruning" usually requires slightly more time than "constraint pattern pruning", mainly because the individual ugliness metric requires $\mathcal{O}(m-p)$ time for "anti-pattern pruning" and $\mathcal{O}(s)$ time for "constraint pattern pruning". This would not be true, of course, if *s* were larger than $m - p$. However, it should be pointed out that the larger *s* is, the less likely we are to be inside the non-ideally constrained domain in the first place.

4.8 Summary of Methods for the Non-Ideally Constrained Case

In this section, we briefly summarize the non-ideally constrained methods we have discussed so far and compare their pattern quality and compute time characteristics. The conclusions we express here are the result of our initial experiments using several "test-case" problems, and cannot be construed to be an exhaustive analysis.[7]

[7] In other words, "Your mileage may vary."

Since the "brute force" method requires more time than the known age of the universe, we rule out any further consideration of this method in favor of the more computationally tractable heuristic methods.

Our experience so far has shown that the best quality patterns are generally produced by the "add least ugly" method operating on a deformed ideal pattern and using the general rather than the individual ugliness metric. This is also our most expensive heuristic with a worst-case time of $\mathcal{O}(m^5)$.

In most of our test cases, "anti-pattern pruning" has produced patterns just as good as "add least ugly" and at a worst-case cost of only $\mathcal{O}(m^4)$.

"Constraint pattern pruning" on deformed ideal patterns is slightly more efficient than "anti-pattern pruning", although it is still $\mathcal{O}(m^4)$ worst-case. The pattern quality has been noticeably inferior, however.

In general, none of the "non-deformed ideal pattern" methods produce particularly good patterns, although "add least ugly" using general ugliness produces quite acceptable results.

Of the $\mathcal{O}(m^3)$ algorithms, "un-deformed add least ugly" produces some of the worst patterns, being highly susceptible to aliasing and clustering. "Un-deformed constraint pattern pruning" produced better patterns, but was still susceptible to local minima traps.

"Redistribution" has the interesting property of provably producing the best possible patterns – and in $\mathcal{O}(m)$ time – as long as the constraint pattern is perfectly even ($Ugliness(L) = 0$). As the evenness of the constraint pattern deteriorates, however, so does the quality of the "redistribution" patterns – like a beautiful image reflected in a fun-house mirror.

5 Acknowledgements

Most of the work on the unconstrained and ideally-constrained cases reported in sections 2 and 3 was performed at the Los Alamos Neutron Science Center (LANSCE). These two sections are quoted almost verbatim from [4]. The work on the non-ideally constrained section (section 4) is unique to the Spallation Neutron Source.

6 References

[1] D.E.Knuth, “Fundamental Algorithms”, Section 1.2.8, Addison-Wesley (1969).
[2] E.Bjorklund, “A Metric for Measuring the Evenness of Timing System Rep-Rate Patterns”, SNS ASD Tech Note, SNS-NOTE-CNTRL-100 (2003).
[3] J.Stirling, “Methodus Differentialis”, p.137 (1730).
[4] E.Bjorklund, “Algorithms for Optimally Distributed Timing Pulses”, LANL Report, LA-UR-89-3558 (1989).