DYNAMIC ALGORITHMS IN NUMERICAL SEMIGROUPS

GILAD MOSKOWITZ

ABSTRACT. In this paper we describe various methods for calculating the factorization set, length set, and delta set for a numerical semigroup and demonstrate the utility of using dynamic algorithms. We then show that the calculations for the delta set can be further improved by using an algebraic methodology.

BACKGROUND

We begin by describing several of the important sets associated with a numerical semigroup. Let S be a numerical semigroup with finite complement. We can write $S = \langle n_1, n_2, ..., n_k \rangle$ to indicate that the elements of S are generated by $n_1, n_2, ..., n_k$. This means that any element of S can be written as a linear combination of the generating elements. Given $n \in S$ such that

$$n = p_1 n_1 + p_2 n_2 + ... + p_k n_k \text{ for some } p_i \in \mathbb{N}$$

we can write this specific factorization of n as $\mathbf{p} = (p_1, p_2, ..., p_k)$ and the set of factorizations of n is

$$Z_S(n) = \{\mathbf{p} \in \mathbb{N}^k : p_1 n_1 + p_2 n_2 + ... + p_k n_k = n\}.$$

When there is no uncertainty regarding the numerical semigroup S, we simply write $Z(n)$. We also say that the factorization set of S is

$$Z(S) = \{Z(n) : n \in S\}.$$

Given a factorization $\mathbf{p}$ of n, we say that the length of that factorization is

$$|\mathbf{p}| = p_1 + p_2 + ... + p_n$$

and the length set for a given n is

$$L(n) = \{|\mathbf{p}| : \mathbf{p} \in Z(n)\}.$$

The a length set of S is defined as

$$L(S) = \{L(n) : n \in S\}.$$

Let $L(n) = \{l_0, l_1, ..., l_k\}$ for some $n \in S$ and finite k, where $l_i < l_j$ for $i < j$. We say that the delta set of n is

$$\Delta(n) = \{|l_i - l_{i-1}| : i = 1, 2, .., k\}.$$

We say that the delta set of S is

$$\Delta(S) = \cup_{n=0}^{\infty} \Delta(n).$$

Now that we have a notion of the sets we are discussing, we can start to describe the various computational methodologies used to calculate these sets.

Coding Definitions

Definition 1. Dictionary: A mapping of a unique set of keys to specific values.

Example 1.

Dict = {“Name1” : “Leeav”, “Age1” : 24, “Name2” : “Gilad”, “Age2” : 24}

As we can see we have all unique keys but some repeated values. If we wanted to call a value from the dictionary, we would use Dict[“Name1”] and we would get an output of Leeav.

Definition 2. List : An array of values.

Example 2.

L = [1, 2, 1, 4, 7]

The elements of a list are indexed starting at zero. For example, in this list if we were to call L[1] we would get an output of 2. Note: the elements of a list can be a variety of things, but for the purposes of this paper we will only look at lists of numbers.

Notation 1. For the purposes of this paper, when presenting code or pseudo-code for a function, we use S to represent the list that corresponds to the minimal generators of the numerical semigroup. This means that, when one would typically write $S = \langle s_1, s_2, ..., s_q\rangle$ we use $S = [s_1, s_2, ..., s_q]$.

1. Brute Force Methodology

One potential type of algorithm for solving problems is computational brute force. This means testing all possible combinations that could potentially work and seeing which ones do. Brute force is often considered the least efficient type of algorithm and even small bounding conditions can vastly improve the runtime. We now present a generic brute force algorithm for finding the $Z(S)$ up to a specific element, and then demonstrate how one small constraint vastly improves runtime. A brief explanation of how the program works follows the code.

```
def BadBruteForceFact(S, nmax):
    F = {}
    F[0] ← [0 vector]
    for n in [1 .. nmax]:
        F[n] ← $\{(a_1, a_2, \ldots, a_q) : a_1s_1 + a_2s_2 + \ldots + a_qs_q = n,$ for $a_i \leq n/s_i\}$
    return F
```

The code takes in two pieces of information, the generators of the semigroup $S = \langle s_1, s_2, ..., s_q \rangle$ in the form of a list, and the number up to which we are finding factorization sets for. The output of the function is a dictionary that maps the numbers 0 to *nmax* to their respective factorization set. The way the code works is by finding the factorization set for every $0 \leq n \leq nmax$. For a given n we generate a list (M) of $floor(n/s_i)$ for all i from 1 to q. This list corresponds to the max possible copies of each generator in any factorization of n. Then the code runs through all possible combinations starting with the all zeros list and counting up to the list M, testing each combination as a linear combination of the generators and seeing if they sum up to n. If a specific combination does work, it gets added to the dictionary corresponding to the key n.

Example 3. Let $S = \langle 6, 9, 20 \rangle$ and $nmax = 500$, suppose we are at the specific iteration that is looking at $n = 180$. We start by generating the list M which will look like $M = [180/6, 180/9, 180/20] = [30, 20, 9]$. Then, we start counting and checking all possible combinations of $6a + 9b + 20c$ for $0 \leq a \leq 30$, $0 \leq b \leq 20$, and $0 \leq c \leq 9$. We do this by generating the list $[0, 0, 0]$ and checking if this works, then $[0, 0, 1]$, ..., then the list $[30, 20, 9]$. Any list we check that works gets added to the dictionary corresponding to the key n, in this case 180.

Using the above code turns out to be extremely inefficient since we end up looking through a lot of unnecessary combinations. One simple way of significantly improving the runtime of the “Bad” brute force method, is to solve for

$$a_2 s_2 + a_3 s_3 + ... + a_q s_q = n \mod s_1$$

which means only looking at linear combinations of some of the generators (all but the first) and seeing if taking n minus that linear combination yields something that is divisible by the smallest generator. The code is presented below followed by an example.

```
def BetterBruteForceFact(S, nmax):
    F = {}
    F[0] ← [0 vector]
    for n in [1 .. nmax]:
        F[n] ← $\{(n/a_1, a_2, \ldots, a_q) : n - (a_2 s_2 + \ldots + a_q s_q) = 0 \bmod s_1,$ for $a_i \leq n/s_i\}$
    return F
```

Example 4. Let $S = \langle 6, 9, 20 \rangle$ and $nmax = 500$, suppose we are at the specific iteration that is looking at $n = 180$. We start by generating the list M which will look like $M = [0, 180/9, 180/20] = [0, 20, 9]$. When then check to see if 180 is divisible by 6 and since it is, we add $(180/6, 0, 0)$ to the set of factorizations. Then, we start counting and checking all possible combinations of $(180 - 9b + 20c)/6$ for $0 \leq b \leq 20$

and $0 \leq c \leq 9$ and see if we get an integer value. If we do then we make the list $[(180-9b+20c)/6, b, c]$ and add it to the dictionary corresponding to 180.

Now we have shown the brute force methodology for finding factorization sets of a numerical semigroup. The methodology for finding the length set uses the same basic code as that for the factorization, except instead of adding a list to the dictionary, we sum together all the values in the list and add that to the dictionary. We use the basis of the code described above to write the brute force methodology for the length set.

```
def BadBruteForceLen(S, nmax):
    F = {}
    F[0] ← [0]
    for n in [1 .. nmax]:
        F[n] ← $\{\sum_{i=1}^{q} a_i : a_1s_1 + a_2s_2 + \ldots + a_qs_q = n,$ for $a_i \leq n/s_i\}$
    return F

def BetterBruteForceLen(S, nmax):
    F = {}
    F[0] ← [0 vector]
    for n in [1 .. nmax]:
        F[n] ← $\{n/a_1 + \sum_{i=2}^{q} a_i : n - (a_2s_2 + \ldots + a_qs_q) = 0 \bmod s_1,$ for $a_i \leq n/s_i\}$
    return F
```

Finding the Factorization Set and Length Set for a specific element. All the aforementioned code returns a dictionary which stores all the factorization sets and length sets of the values 0 to *nmax*. The benefit of having this type of output is that after running the code, an individual has access to the information they desire for a wide range of elements of S. Referencing the information in the dictionary takes no time ($\mathcal{O}(1)$ for those familiar with this notation). However, if an individual only cares about the factorization set or length set of an element, n, in S it would be more efficient to only calculate it for that element. This can be done in the same manner described in the previous examples for finding the factorization set or length set of a specific number while iterating between 1 and *nmax*. The code for doing so is as follows

```
def BadBruteForceLenOne(S, n):
    F = []
    F ← $\{\sum_{i=1}^{q} a_i : a_1s_1 + a_2s_2 + \ldots + a_qs_q = n,$ for $a_i \leq n/s_i\}$
    return sorted(F)

def BetterBruteForceLenOne(S, n):
    F = []
    F ← $\{n/a_1 + \sum_{i=2}^{q} a_i : n - (a_2s_2 + \ldots + a_qs_q) = 0 \bmod s_1,$ for $a_i \leq n/s_i\}$
    return sorted(F)

def BadBruteForceFactOne(S, n):
    F = []
    F ← $\{(a_1, a_2, \ldots, a_q) : a_1 s_1 + a_2 s_2 + \ldots + a_q s_q = n,$ for $a_i \leq n/s_i\}$
    return F

def BetterBruteForceFactOne(S, n):
    F = []
    F ← $\{(n/a_1, a_2, \ldots, a_q) : n - (a_2 s_2 + \ldots + a_q s_q) = 0 \bmod s_1,$ for $a_i \leq n/s_i\}$
    return F
```

2. Dynamic Algorithms Methodology

Background. A dynamic algorithm is a way of solving the problem that breaks up the problem into many smaller parts and solves it recursively. The dynamic algorithm can store the information from the solutions to the smaller problems and use those to solve the larger problem.

Example 5. Suppose you wanted to write a code that will give you the n-th number of the Fibonacci sequence. One way to write this code is to solve the problem recursively, this means that if we wanted a function $Fib(n)$ that returns the n-th number of the Fibonacci sequence using a recursive algorithm we can code it as follows

```
import math
def Fib(n):
    if n <= 1:
        return n
    else:
        return Fib(n - 1) + Fib(n - 2)
```

The function takes an input of n as the number in the sequence we are trying to reach. Suppose that we put $n = 4$. The way the algorithm would run is as follows:
First iteration: Fib(3) + Fib(2)
Second iteration: (Fib(2) + 1) + (1 + 0)
Third iteration: ((1 + 0) + 1) + (1 + 0) = 3
So as we see, calling this function requires us to calculate $Fib(2)$ twice. For higher values of n we would have to make repeated calculations even more times.
A dynamic algorithm for finding the n-th term in the Fibonacci sequence can be written as follows

```
import math
def Fib(n);
    Dict = {}
    Dict[0] = 0
    Dict[1] = 1
    if n <= 1:
        return Dict[n]
    for i in [2..n]:
        Dict[i] = Dict[i - 1] + Dict[i - 2]
    return Dict[n]
```

This algorithm stores the previously calculated Fibonacci numbers and uses them to calculate the next one. There is no repeated calculations needed and this will calculate each Fibonacci number between 2 and n exactly once.
Clearly the dynamic algorithm for calculating the n-th term in the Fibonacci sequence is much faster than the recursive method that repeats calculations. In general, a dynamic algorithm for solving a problem is very quick.

Now that we have introduced the concept of a dynamic algorithm, we can describe the use of dynamic algorithms in solving for the factorization set and the length set of a numerical semigroup.

Dynamic Algorithms for Factorization Set and Length Set. We begin with the following lemma regarding the factorization set of n in a numerical monoid (note that for the purposes of this paper numerical monoid and numerical semigroup are interchangeable).

Lemma ([1, Lemma 3.1])**.** *Fix a reduced, finitely generated monoid M (written additively) with irreducible elements $m_1, m_2, ..., m_k$. For each non-zero $n \in M$, we have*

$$\begin{aligned} Z(n) &= \cup_{i=1}^{k}\{\boldsymbol{a} + \boldsymbol{e}_i : \boldsymbol{a} \in Z(n - m_i)\} \\ &= \sqcup_{i=1}^{k}\{\boldsymbol{a} + \boldsymbol{e}_i : \boldsymbol{a} \in Z(n - m_i), a_j = 0 \text{ for each } j < i\} \end{aligned}$$

Using the information in the lemma we can write a dynamic program for getting the factorization sets up to a specific element. First, we present an example to demonstrate the first claim made by the lemma, then we will present the code that we can use to actually compute the factorization set.

Example 6. Let $S = \langle 6, 9, 20\rangle$. Suppose we wanted to calculate the factorization set for the element 15. We first look to see if $15 - 6 = 9 \in S$, which it is, so we check the factorization set of 9. We see that the only factorization of 9 is $(0, 1, 0)$. To get a factorization for 15 from this factorization of 9 we increment the first spot (i.e. the spot corresponding to the number of 6's in a factorization) by 1. We get the factorization $(1, 1, 0)$, which is indeed a factorization for 15. We now move onto the second generator and check to see if $15 - 9 = 6 \in S$. Since $6 \in S$ with a factorization $(1, 0, 0)$ we now increment the second spot, corresponding to the number of 9's, by 1 to get $(1, 1, 0)$. Finally, we check $15 - 20 = -5 \notin S$, so we get our factorization set for 15 is $\{(1, 1, 0)\} \cup \{(1, 1, 0)\} = \{(1, 1, 0)\}$.

The dynamic code that calculates the factorization set stores all the previous factorization sets in a dictionary and then calls on those dictionary values to generate the new factorization.

```
def FactorizationsUpToElement(S, nmax):
    start = time.time()
    F = {}
    p = [0]*len(S)
    F[0] = []
    F[0].append(p)
    for n in [1 .. nmax]:
        F[n] = set([])
        for i in range(len(S)):
            if n - S[i] < 0:
                continue
            for a in F[n - S[i]]:
                s = list(a)
                s[i] += 1
                s = tuple(s)
                if s not in F[n]:
                    F[n] = F[n].union(set([s]))
    runtime = time.time() - start
    print runtime
    return F
```

NOTE: *The code as written above only looks at the union of the previous factorization sets, and not the disjoint union of portions of the previous factorization sets. Implementing a code that utilizes the second equality in the lemma involves sorting the factorization sets of each element, but would be substantially more optimized.*

The dynamic algorithm for calculating the length set is very similar in nature to the one provided by Lemma 3.1. Using Lemma 3.1 we can see that

Corollary ([1, Lemma 3.5]). *Fix a reduced, finitely generated monoid M (written additively) with irreducible elements $m_1, m_2, ..., m_k$. For each non-zero $n \in M$, we* *have*

$$\begin{aligned}L(n) &= \{|\boldsymbol{p}| : \boldsymbol{p} \in Z(n)\}\\ &= \cup_{i=1}^{k}\{|\boldsymbol{a} + \boldsymbol{e}_i| : \boldsymbol{a} \in Z(n - m_i)\}\\ &= \cup_{i=1}^{k}\{|\boldsymbol{a}| + 1 : \boldsymbol{a} \in Z(n - m_i)\}\\ &= \cup_{i=1}^{k}\{L(n - m_i) + 1\}\\ &= \cup_{i=1}^{k}\{l + 1 : l \in L(n - m_i)\}\end{aligned}$$

From the corollary we can write a dynamic program that calculates the length set of a specific $n \in S$ without the need to look at the factorization set at all.

Example 7. Let $S = \langle 6, 9, 20\rangle$. We see that the factorization length of 0 is 0. If we were to try to calculate the length set of 6 we would look if $6 - 6 = 0 \in S$, $6 - 9 = -3 \notin S$, and $6 - 20 = -14 \notin S$. Since we only have that $0 \in S$ we only care about the length set of 0 which is 0. So we see based on the corollary that $L(6) = \{0 + 1\} = \{1\}$. Let's also examine the length set of 15. For 15 we have $15 - 6 = 9 \in S$, $15 - 9 = 6 \in S$, and $15 - 20 = -5 \notin S$, so we need to look at the length set of 6 and 9. By the corollary we have that

$$\begin{aligned}L(15) &= \{L(6) + 1\} \cup \{L(9) + 1\}\\ &= \{1 + 1\} \cup \{1 + 1\}\\ &= \{2\} \cup \{2\}\\ &= \{2\}\end{aligned}$$

A dynamic algorithm using the corollary as a basis stores the length sets of previously calculated elements in a dictionary and then calls on those values to calculate the next length set. The code for such an algorithm is as follows

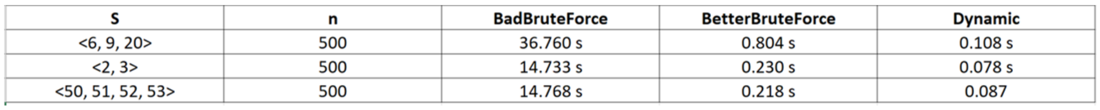
```
def LengthSetsUpToElement(S, nmax):
    start = time.time()
    lengthsets = {}
    lengthsets[0] = [0]
    for n in [1 .. nmax]:
        if n in lengthsets:
            continue
        lengthsets[n] = []
        for i in range(len(S)):
            if n - S[i] < 0:
                continue
            lengthsets[n] += [l + 1 for l in lengthsets[n - S[i]]]
        lengthsets[n] = sorted(list(Set(lengthsets[n])))
    runtime = time.time() - start
    print runtime
    return lengthsets
```

NOTE: *The code as written above could be further optimized by using Booleans to check if a specific factorization length has already been accounted for. Currently the code makes a list of all possible factorization lengths for a specific n then removes the duplicates and sorts it.*

The dynamic algorithms both output a dictionary with values for each element from 0 to *nmax*. Similarly to before, if an individual wanted to be able to access the information for many different elements then using this dictionary output is ideal. However, using a dynamic algorithm, if an individual only wanted to access the factorization set or length set of a specific element, they would need to set *nmax* equal to that element and then call on the dictionary element *nmax*. Typically, this will still run faster then brute forcing the factorization set or length set for the specific element, but there are some extremal cases for which this is not true.

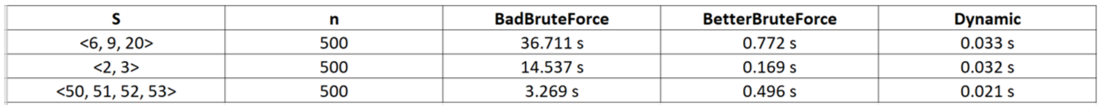
Runtimes. Below we have a table of runtimes for finding the factorization set up to a specific element using all three previously presented algorithms.
The runtimes show that for calculating all the values up to n the dynamic algorithm is significantly faster than the other two. However, for cases with a small number of generators the slightly optimized brute force methodology works slightly faster than the dynamic algorithm. Further tests and runtimes need to be recorded.

Calculating factorization sets up to n, various runtimes.



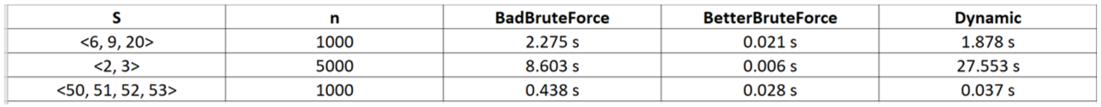
S	n	BadBruteForce	BetterBruteForce	Dynamic
<6, 9, 20>	500	36.760 s	0.804 s	0.108 s
<2, 3>	500	14.733 s	0.230 s	0.078 s
<50, 51, 52, 53>	500	14.768 s	0.218 s	0.087

Calculating length sets up to n, various runtimes.



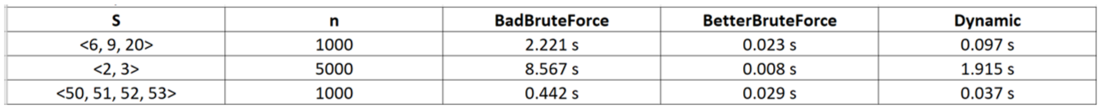
S	n	BadBruteForce	BetterBruteForce	Dynamic
<6, 9, 20>	500	36.711 s	0.772 s	0.033 s
<2, 3>	500	14.537 s	0.169 s	0.032 s
<50, 51, 52, 53>	500	3.269 s	0.496 s	0.021 s

Calculating factorization set of n, various runtimes.



S	n	BadBruteForce	BetterBruteForce	Dynamic
<6, 9, 20>	1000	2.275 s	0.021 s	1.878 s
<2, 3>	5000	8.603 s	0.006 s	27.553 s
<50, 51, 52, 53>	1000	0.438 s	0.028 s	0.037 s

Calculating length set of n, various runtimes.



S	n	BadBruteForce	BetterBruteForce	Dynamic
<6, 9, 20>	1000	2.221 s	0.023 s	0.097 s
<2, 3>	5000	8.567 s	0.008 s	1.915 s
<50, 51, 52, 53>	1000	0.442 s	0.029 s	0.037 s

3. Computing the delta set

Before the dynamic algorithm for computing the length set, there was no way to reasonably (*within a relatively small amount of time*) computationally calculate the delta set of a numerical semigroup. Once the dynamic algorithm was created, using the bound for the number of elements you have to look at, N_S [2], we can get the delta set of every element up to N_S and then union those sets to get the delta set of S. This allows us to find the delta set of S in reasonable computational time. The pseudo-code for calculating the delta set is as follows (note: that here S only represents the list of the generators of the numerical semigroup)

```
def DeltaSet(S):
    Calculate $N_S$
    nmax = $N_S + \text{lcm}(\min(S), \max(S))$
    A = LengthSetsUpToElement(S, nmax)
    $\Delta$ = $\cup_{i=0}^{nmax}\Delta(A[i])$
    return $\Delta$
```

with the code calculating the delta set for all the elements in S between 0 and $N_S + \text{lcm}(\min(S), \max(S))$ and then taking the union of all those sets to get $\Delta(S)$. This computation of the delta set is resonably fast, since if you recall the algorithm for calculating the length sets of all the elements in S is very quick. However, for substantially large N_S, this code can still take a while to run.
Here I would like to call back to the example I used earlier of calculating the n-th element of the Fibonacci Sequence. We saw that using the classic recursive algorithm has us calculate the same values over and over, and we can make a much more efficient dynamic algorithm. The dynamic algorithm was very fast, but still required us to run through all the elements of the Fibonacci Sequence from the first to the n-th. There is another, faster, methodology for calculating the n-th term of the Fibonacci Sequence, and that is to use a formula. The code for that would look as follows

```
import math
def Fib(n):
    return $\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$
```

and has $\mathcal{O}(1)$ meaning that the time it takes to run this is constant and doesn't depend on n. This is significantly more efficient than the dynamic algorithm we presented earlier. The reason I call attention to this is to say that, if an individual has some amount of formulas or algebra they can use to solve a complicated problem then it can simplify the problem significantly more than a purely dynamic algorithm. For calculating the delta set, an algebraic methodology was found that substantially reduced the computational time.

3.1. **Algebraic Methodology for calculating the Delta Set.** We start with the ring $\Bbbk[x_1, x_2, ..., x_n]$ and the numerical semigroup $S = \langle s_1, s_2, ..., s_k\rangle$. We will also have a mapping from $\varphi : \mathbb{Z}^k_{\geq 0} \to S$ such that $\varphi(a) = a_1s_1 + a_2s_2 + ... + a_ks_k$ We make an ideal $I_S = \langle \mathbf{x^a} - \mathbf{x^b} : \varphi(\mathbf{a}) = \varphi(\mathbf{b})\rangle$. This is essentially saying that I_S is the binomial ideal generated by the elements in S with multiple factorizations. We then add a new variable t to I_S with a lex definition $t > x_i$ such that we homogenize all the generators of I_S using t. We find that $\deg(t)$ actually represents the difference in length between two factorizations of an element in S.

Example 8. Let $S = \langle 6, 9, 20\rangle$. Then one of our generators of the ideal I_S is going to be $x_1^3 - x_2^2$, which comes from the two ways of factoring $18 \in S$. To homogenize this element we simply take $x_1^3 - tx_2^2$ and here we see that the degree of t is 1, which is exactly the difference in factorization length between the two.

Once we have homogenized the elements of I_S in the manner described above, we calculate a reduced lex Grobner Basis G for I_S and we find that $\Delta(S) = \{d : t^d\mathbf{x^a} - \mathbf{x^b} \in G\}$. The proof for this uses Hilbert Basis and chains of ideals [3], and this allows us to come up with a much cleaner and more efficient algorithm for calculating the delta set. Instead of using the dynamic algorithm for calculating the length set up to an extremely large element of S, we can simply look at the delta sets of a small subset of elements in S. The pseudo-code for calculating the delta set in the manner described above is as follows (note: $S = \langle s_1, s_2, ..., s_k\rangle$)

```
def DeltaSet(S):
    p = minimal presentation of $S_A = \langle (1,0), (1,s_1), (1,s_2), ...(1,s_k)\rangle$
    G = Grobner Basis of $\langle t^i\mathbf{x^a} - t^j\mathbf{x^b} : ((i,a),(j,b)) \in p\rangle$
    $\Delta$ = $\{d : t^d\mathbf{x^a} - \mathbf{x^b} \in G\}$
    return $\Delta$
```

and using this code we are able to calculate the delta set signigificantly faster than using the previous dynamic algorithm.

4. Conclusion

We can see that there a variety of methodologies for calculating the various properties of a numerical semigroup S. Typically we have found that the brute force methodology for calculating the factorization set and length set is rather slow. By finding a dynamic methodology for making these calculations we are able to do them in reasonable computational time. This also allows us to calculate other properties of a numerical semigroup, such as the delta set, in a reasonable time. However, by finding an algebraic algorithm for calculating the delta set we are able to significantly reduce the runtime needed for it.

5. Appendix

The following is the actual code to all the programs described above. The name of the function is the same as that of the pseudo-code:

```
import math
import time
def BadBruteForceFact(S, nmax):
     start = time.time()
     F = {}
     p = [0]*len(S)
     F[0] = []
     F[0].append(p)
     for n in [1 .. nmax]:
          F[n] = []
          M = []
          for i in range(len(S)):
               M.append(floor(n/S[i]))
          N = [0]*len(M)
          while(N != M):
               for i in range(1, len(M)):
                    if(N[len(M)-i] == M[len(M)-i]):
                         N[len(M)-i] = 0
                         if(N[len(M) - (i+1)] != M[len(M) - (i+1)]):
                              N[len(M) - (i+1)] += 1
                              break
                    else:
                         N[len(M)-i] += 1
                         break
               a = 0
               for l in range(len(N)):
                    a += N[l]*S[l]
               if a == n:
                    F[n].append(N[:])
     runtime = time.time() - start
     print runtime
     return F

import math
import time
```

```
def BetterBruteForceFact(S, nmax):
    start = time.time()
    F = {}
    p = [0]*len(S)
    F[0] = []
    F[0].append(p)
    for n in [1 .. nmax]:
        F[n] = []
        M = []
        for i in range(len(S)):
            if i == 0:
                M.append(0)
            else:
                M.append(floor(n/S[i]))
        N = [0]*len(M)
        if n%S[0] == 0:
            A = N[:]
            A[0] = n/S[0]
            F[n].append(A)
        while(N != M):
            for i in range(1, len(M)):
                if(N[len(M)-i] == M[len(M)-i]):
                    N[len(M)-i] = 0
                    if(N[len(M) - (i+1)] != M[len(M) - (i+1)]):
                        N[len(M) - (i+1)] += 1
                        break
                else:
                    N[len(M)-i] += 1
                    break
            a = 0
            for l in range(len(N)):
                a += N[l]*S[l]
            if ((n - a)%S[0] == 0 and (n - a) >= 0):
                A = N[:]
                A[0] = (n - a)/S[0]
                F[n].append(A)
    runtime = time.time() - start
    print runtime
    return F

import math
import time
def BadBruteForceLen(S, nmax):
    start = time.time()
    F = {}
    p = 0
    F[0] = []
    F[0].append(p)
    for n in [1 .. nmax]:
        F[n] = []
        M = []
        for i in range(len(S)):
            M.append(floor(n/S[i]))
        N = [0]*len(M)
        while(N != M):
            for i in range(1, len(M)):
                if(N[len(M)-i] == M[len(M)-i]):
                    N[len(M)-i] = 0
                    if(N[len(M) - (i+1)] != M[len(M) - (i+1)]):
                        N[len(M) - (i+1)] += 1
                        break
                else:
                    N[len(M)-i] += 1
                    break
            a = 0
            for l in range(len(N)):
                a += N[l]*S[l]
            if a == n:
                b = 0
                for i in N:
                    b += i
                F[n].append(b)
        F[n] = sorted(list(set((F[n]))))
    runtime = time.time() - start
    print runtime
    return F
```

```
def BetterBruteForceLen(S, nmax):
    start = time.time()
    F = {}
    p = 0
    F[0] = []
    F[0].append(p)
    for n in [1 .. nmax]:
        F[n] = []
        M = []
        for i in range(len(S)):
            if i == 0:
                M.append(0)
            else:
                M.append(floor(n/S[i]))
        N = [0]*len(M)
        if n%S[0] == 0:
            F[n].append(n/S[0])
        while(N != M):
            for i in range(1, len(M)):
                if(N[len(M)-i] == M[len(M)-i]):
                    N[len(M)-i] = 0
                    if(N[len(M) - (i+1)] != M[len(M) - (i+1)]):
                        N[len(M) - (i+1)] += 1
                        break
                else:
                    N[len(M)-i] += 1
                    break
            a = 0
            for l in range(len(N)):
                a += N[l]*S[l]
            if ((n - a)%S[0] == 0 and (n - a) >= 0):
                A = N[:]
                A[0] = (n - a)/S[0]
                b = 0
                for i in A:
                    b += i
                F[n].append(b)
        F[n] = sorted(list(set((F[n]))))
    runtime = time.time() - start
    print runtime
    return F

def BadBruteForceLenOne(S, n):
    start = time.time()
    F = []
    M = []
    for i in range(len(S)):
        M.append(floor(n/S[i]))
    N = [0]*len(M)
    while(N != M):
        for i in range(1, len(M)):
            if(N[len(M)-i] == M[len(M)-i]):
                N[len(M)-i] = 0
                if(N[len(M) - (i+1)] != M[len(M) - (i+1)]):
                    N[len(M) - (i+1)] += 1
                    break
            else:
                N[len(M)-i] += 1
                break
        a = 0
        for l in range(len(N)):
            a += N[l]*S[l]
        if a == n:
            b = 0
            for i in N:
                b += i
            F.append(b)
    F = sorted(list(set((F))))
    runtime = time.time() - start
    print runtime
    return F

def BetterBruteForceLenOne(S, n):
    start = time.time()
    F = []
    M = []
    for i in range(len(S)):
        if i == 0:
            M.append(0)
        else:
            M.append(floor(n/S[i]))
    N = [0]*len(M)
    if n%S[0] == 0:
        F.append(n/S[0])
    while(N != M):
        for i in range(1, len(M)):
            if(N[len(M)-i] ==M[len(M)-i]):
                N[len(M)-i] = 0
                if(N[len(M) - (i+1)] != M[len(M) - (i+1)]):
                    N[len(M) - (i+1)] += 1
                    break
            else:
                N[len(M)-i] += 1
                break
        a = 0
        for l in range(len(N)):
            a += N[l]*S[l]
        if ((n - a)%S[0] == 0 and (n - a) >= 0):
            A = N[:]
            A[0] = (n - a)/S[0]
            b = 0
            for i in A:
                b += i
            F.append(b)
    F = sorted(list(set((F))))
    runtime = time.time() - start
    print runtime
    return F

def BadBruteForceLenOpt(S, nmax):
    start = time.time()
    F = {}
    p = 0
    F[0] = []
    F[0].append(p)
    for n in [1 .. nmax]:
        F[n] = []
        M = []
        for i in range(len(S)):
            M.append(floor(n/S[i]))
        N = [0]*len(M)
        r = floor(n/max(S))
        q = floor(n/min(S))
        D = {}
        for j in [r..q]:
            D[j] = False
        while(N != M):
            for i in range(1, len(M)):
                if(N[len(M)-i] == M[len(M)-i]):
                    N[len(M)-i] = 0
                    if(N[len(M) - (i+1)] != M[len(M) - (i+1)]):
                        N[len(M) - (i+1)] += 1
                        break
                else:
                    N[len(M)-i] += 1
                    break
            a = 0
            for l in range(len(N)):
                a += N[l]*S[l]
            if a == n:
                b = 0
                for i in N:
                    b += i
                D[b] = True
        for j in [r..q]:
            if D[j]:
                F[n].append(j)
    runtime = time.time() - start
    print runtime
    return F

def BetterBruteForceLenOpt(S, nmax):
    start = time.time()
    F = {}
    D = {}
```

```
p = 0
F[0] = []
F[0].append(p)
for n in [1 .. nmax]:
    r = floor(n/max(S))
    q = floor(n/min(S))
    D = {}
    for j in [r..q]:
        D[j] = False
    F[n] = []
    M = []
    for i in range(len(S)):
        if i == 0:
            M.append(0)
        else:
            M.append(floor(n/S[i]))
    N = [0]*len(M)
    if n%S[0] == 0:
        D[n/S[0]] = True
    while(N != M):
        for i in range(1, len(M)):
            if(N[len(M)-i] == M[len(M)-i]):
                N[len(M)-i] = 0
                if(N[len(M) - (i+1)] != M[len(M) - (i+1)]):
                    N[len(M) - (i+1)] += 1
                    break
            else:
                N[len(M)-i] += 1
                break
        a = 0
        for l in range(len(N)):
            a += N[l]*S[l]
        if ((n - a)%S[0] == 0 and (n - a) >= 0):
            A = N[:]
            A[0] = (n - a)/S[0]
            b = 0
            for i in A:
                b += i
                D[b] = True
            for j in [r..q]:
                if D[j]:
                    F[n].append(j)
        runtime = time.time() - start
        print runtime
        return F

def BadBruteForceFactOne(S, n):
    start = time.time()
    F = []
    M = []
    for i in range(len(S)):
        M.append(floor(n/S[i]))
    N = [0]*len(M)
    while(N != M):
        for i in range(1, len(M)):
            if(N[len(M)-i] == M[len(M)-i]):
                N[len(M)-i] = 0
                if(N[len(M) - (i+1)] != M[len(M) - (i+1)]):
                    N[len(M) - (i+1)] += 1
                    break
            else:
                N[len(M)-i] += 1
                break
        a = 0
        for l in range(len(N)):
            a += N[l]*S[l]
        if a == n:
            F.append(N[:])
    runtime = time.time() - start
    print runtime
    return F

def BetterBruteForceFactOne(S, n):
    start = time.time()
    F = []
    M = []
    for i in range(len(S)):
        if i == 0:
            M.append(0)
        else:
            M.append(floor(n/S[i]))
    N = [0]*len(M)
    if n%S[0] == 0:
        A = N[:]
        A[0] = n/S[0]
        F.append(A)
    while(N != M):
        for i in range(1, len(M)):
            if(N[len(M)-i] == M[len(M)-i]):
                N[len(M)-i] = 0
                if(N[len(M) - (i+1)] != M[len(M) - (i+1)]):
                    N[len(M) - (i+1)] += 1
                    break
            else:
                N[len(M)-i] += 1
                break
        a = 0
        for l in range(len(N)):
            a += N[l]*S[l]
        if ((n - a)%S[0] == 0 and (n - a) >= 0):
            A = N[:]
            A[0] = (n - a)/S[0]
            F.append(A)
    runtime = time.time() - start
    print runtime
    return F
```

References

[1] T. Barron, C. O'Neill, and R. Pelayo, *On dynamic algorithms for factorization invariants in numerical monoids*, Mathematics of Computation, 86 (2016), p. 2429–2447.

[2] J. I. García-García, M. A. Moreno-Frías, and A. Vigneron-Tenorio, *Computation of delta sets of numerical monoids*, 2014.

[3] P. A. García-Sánchez, C. O'Neill, and G. Webb, *On the computation of factorization invariants for affine semigroups*, 2015.

Mathematics Department, San Diego State University, San Diego, CA 92182
Email address: gilad.moskowitz@gmail.com