EXPLORING THE TREE OF NUMERICAL SEMIGROUPS

JEAN FROMENTIN AND FLORENT HIVERT

Abstract. In this paper we describe an algorithm visiting all numerical semigroups up to a given genus using a well suited representation. The interest of this algorithm is that it fits particularly well the architecture of modern computers allowing very large optimizations: we obtain the number of numerical semigroups of genus $g \leqslant 67$ and we confirm the Wilf conjecture for $g \leqslant 60$.

Introduction

A *numerical semigroup* S is a subset of $\mathbb{N}$ containing 0, closed under addition and of finite complement in $\mathbb{N}$. For example the set

$$S_E = \{0, 3, 6, 7, 9, 10\} \cup \{x \in \mathbb{N}, x \geqslant 12\} \tag{1}$$

is a numerical semigroup. The *genus* of a numerical semigroup S, denoted by $g(S)$, is the cardinality of $\mathbb{N} \setminus S$. For example the genus of S_E is 6, the cardinality of $\{1, 2, 4, 5, 8, 11\}$.

For a given positive integer g, the number of numerical semigroups of genus g is finite and is denoted by n_g. In J.A. Sloane's *on-line encyclopedia of integer sequences* [1] we find the values of n_g for $g \leqslant 52$. These values have been obtained by M. Bras-Amorós ([2] for more details for $g \leqslant 50$). On his home page [3], M. Delgado gives the value of n_{55}.

M. Bras-Amorós used a depth first search exploration of the tree of numerical semigroups $\mathcal{T}$ up to a given genus. This tree was introduced by J.C. Rosales and al. in [4] and it is the subject of Section 1. Starting with all the numerical semigroups of genus 49 she obtained the number of numerical semigroups of genus 50 in 18 days on a pentium D runing at 3GHz. In the package `NumericalSgs` [5] of `GAP` [6], M. Delgado together with P.A. Garcia-Sanchez and J. Morais used the same method of exploration.

Here we describe a new algorithm for the exploration of the tree of numerical semigroups $\mathcal{T}$ and achieve the computation of n_g for $g \leqslant 67$. The cornerstone of our method is a combinatorial representation of numerical semigroups that is well suited and allows large code optimization essentially based on the use of vectorial instructions and parallelization. The goal of the paper is twofold: first to present our encoding of numerical semigroups and the associated algorithms, and second to present the optimization techniques which allow, for those kinds of algorithms, to get speedups by factors of hundreds and even thousands. We claim that these techniques are fairly general for those kinds of algorithms. As a support for the claim, we applied it to an algorithm of N. Borie enumerating integer vector modulo permutation groups [?] and got a speedup by a factor larger than 2000 using 8 cores.

The paper is divided as follows. In Section 1 we describe the tree of numerical semigroups and give bounds for some parameters attached to a numerical semigroup. The description of our representation of numerical semigroups is done in the second section. In Section 3 we describe an algorithm based on the representation given in Section 2 and give its complexity. Section 4 is more technical, and is devoted to the optimization of the algorithm introduced in Section 3. In the last section we emphasize the results obtained using our algorithm.

1. The tree of numerical semigroups

We start this section with definitions and properties of numerical semigroups that will be used in the sequel. For a more complete introduction, the reader can usefully consult the book *Numerical Semigroups* by J.C. Rosales and P.A. García-Sánchez [7] or the book *The Diophantine Frobenius Problem* by J.L. Ramírez Alfonsín [8].

Date: April 10, 2014.
2010 *Mathematics Subject Classification.* 05A15,68R05,68W10.
Key words and phrases. semigroups, tree, algorithm, `SSE`, `Cilk`, optimization.

Definition 1.1. Let S be a numerical semigroup. We define
i) $m(S) = \min(S \setminus \{0\})$, the *multiplicity* of S;
ii) $g(S) = \mathrm{card}(\mathbb{N} \setminus S)$, the *genus* of S;
iii) $f(S) = \max(\mathbb{Z} \setminus S)$, the *Frobenius* of S;
iv) $c(S) = f(S) + 1$, the *conductor of S*.

By definition a numerical semigroup is an infinite object and we need a finite description of such an object. That is provided by generating sets.

Definition 1.2. A subset $X = \{x_1 < x_2 < ... < x_n\}$ of a semigroup is a *generating set* of S if every element of S can be expressed as a sum of elements in X. In this case we write $S = \langle x_1, ..., x_n\rangle$.

If we reconsider the numerical semigroup of (1), we obtain

$$S_E = \{0, 3, 6, 7, 9, 10\} \cup [12, +\infty[= \langle 3, 7\rangle\,. \tag{2}$$

A non-zero element x of a numerical semigroup S is said to be *irreducible* if it cannot be expressed as a sum of two non-zero elements of S. We denote by $\mathrm{Irr}(S)$ the set of all irreducible elements of S.

Lemma 1.3 (Lemma 2.3 of [7]). For a numerical semigroup S, the set $\mathrm{Irr}(S)$ is the minimal generating set of S relatively to the inclusion ordering.

The different parameters we have defined on a numerical semigroup, satisfy the following relations.

Proposition 1.4 (Proposition 2.12 and Lemma 2.14 of [7]). *For every numerical semigroup S, we have*
i) $x \in \mathrm{Irr}(S)$ *implies* $x \leqslant c(S) + m(S) - 1$;
ii) $m(S) \leqslant g(S) + 1$;
iii) $c(S) \leqslant 2g(S)$.

A consequence of Proposition 1.4 *i)* is that $\mathrm{Irr}(S)$ is finite and its cardinality is at most $c(S)+m(S)-1$. Moreover, the cardinality of $\mathrm{Irr}(S)$ is at most $m(S)$ since any two distinct elements of $\mathrm{Irr}(S)$ cannot be congruent modulo $m(S)$. See Section 2 of Chapter I of [7] for more details.

We now explain the construction of the tree of numerical semigroups. Let S be a numerical semigroup. The set $S' = S \cup \{f(S)\}$ is also a numerical semigroup and its genus is $g(S) - 1$. As each integer greater than $f(S)$ is included in S' we have $c(S') \leqslant f(S)$. Therefore every semigroup S of genus g can be obtained from a semigroup S' of genus $g - 1$ by removing an element of S' greater than or equal to $c(S')$.

Proposition 1.5 (Proposition 7.28 of [7]). *Let S be a numerical semigroup and x an element of S. The set $S^x = S \setminus \{x\}$ is a numerical semigroup if and only if x is irreducible in S.*

Proposition 1.5 implies that every semigroup S of genus g can be obtained from a semigroup S by removing a generator x of S that is greater than or equal to $c(S)$. Hence the relation $S' = S^x$ holds.

We construct the tree of numerical semigroups, denoted by $\mathcal{T}$ as follows. The root of the tree is the unique semigroup of genus 0, *i.e*, $\langle 1\rangle$ that is equal to $\mathbb{N}$. If S is a semigroup in the tree, the sons of S are exactly the semigroups S^x where x belongs to $\mathrm{Irr}(S) \cap [c(S), +\infty[$. By convention, when depicting the tree, the numerical semigroup S^x is in the left of S^y if x is smaller than y. With this construction, a semigroup S has depth g in $\mathcal{T}$ if and only if its genus is g, see Figure 1. We denote by $\mathcal{T}_g$ the subtree of $\mathcal{T}$ restricted to all semigroups of genus $\leqslant g$.

2. Decomposition number

The aim of this section is to describe a representation of numerical semigroups, which is well suited to an efficient exploration of the tree $\mathcal{T}$ of numerical semigroups.

Definition 2.1. Let S be a numerical semigroup. For every x of $\mathbb{N}$ we set

$$D_S(x) = \{y \in S \mid x - y \in S \text{ and } 2y \leqslant x\}$$

and $d_S(x) = \mathrm{card}\, D_S(x)$. We called $d_S(x)$ the *S-decomposition number* of x. The application $d_S : \mathbb{N} \to \mathbb{N}$ is the *S-decomposition numbers function.*

Assume that y is an element of $D_S(x)$. By definition of $D_S(x)$, the integer $z = x - y$ also belongs to S. Then x can be decomposed as $x = y + z$ with y and z in S. Moreover the condition $2y \leqslant x$ implies $y \leqslant z$. In other words if we define $D'_S(x)$ to be the set of all $(y, z) \in S \times S$ with $x = y + z$ and $y \leqslant z$ then $D_S(x)$ is the image of $D'_S(x)$ under the projection on the first coordinate. Hence $D_S(x)$ describes how x can be decomposed as sums of two elements of S. This justifies the name given to the function d_S.

Example 2.2. Reconsider the semigroup S_E given at (1). The integer 14 admits two decompositions as sums of two elements of S, namely $14 = 0 + 14$ and $14 = 7 + 7$. Thus the set $D_{S_E}(14)$ is equal to $\{0, 7\}$ and $d_{S_E} = 2$ holds.

Lemma 2.3. For every numerical semigroup S and every integer $x \in \mathbb{N}$, we have $d_S(x) \leqslant 1 + \left\lfloor \dfrac{x}{2} \right\rfloor$ and the equality holds for $S = \mathbb{N}$.

Proof. As the set $D_S(x)$ is included in $\left\{0, ..., \left\lfloor \frac{x}{2} \right\rfloor\right\}$, the relation $d_S(x) \leqslant 1 + \left\lfloor \frac{x}{2} \right\rfloor$ holds. For $S = \mathbb{N}$ we have the equality for the set $D_S(x)$ and so for the integer $d_S(x)$. □

A straightforward consequence of the definition of S-decomposition numbers is :

Proposition 2.4. *For a numerical semigroup S and $x \in \mathbb{N} \setminus \{0\}$, we have:*
i) x lies in S if and only if $d_S(x) > 0$.
ii) x is in $\operatorname{Irr}(S)$ if and only if $d_S(x) = 1$.

We note that 0 is never irreducible despite the fact $d_S(0)$ is 1 for all numerical semigroups S. We now explain how to compute the S-decomposition numbers function of a numerical semigroup from that of its father.

Proposition 2.5. *Let S be a numerical semigroup and x be an irreducible element of S. Then for all $y \in \mathbb{N} \setminus \{0\}$ we have*

$$d_{S^x}(y) = \begin{cases} d_S(y) - 1 & \text{if } y \geqslant x \text{ and } d_S(y - x) > 0, \\ d_S(y) & \text{otherwise.} \end{cases}$$

Proof. A direct consequence of $D_{S^x}(y) = D_S(y) \setminus \{y - x, x\}$. □

3. A new algorithm

We can easily explore the tree of numerical semigroups up to a genus G using a depth first search algorithm (see Algorithm 1). This approach does not seem to have been used before. In particular, M. Bras-Amorós and M. Delgado use instead a breadth first search exploration. The main advantage in our approach is the small memory needs. Indeed, in the case of breadth first search algorithm one needs to compute and store the list L_g of all numerical semigroups of genus g before visiting the numerical semigroups of genus $g+1$, which is not required for a depth first search exploration. In this paper we are interested in the exploration of the list L_g for $g \in \mathbb{N}$, not in storing it. This is the reason we use a depth first search algorithm for the exploration of the tree $\mathcal{T}_g$. The only limitation is then the duration of the exploration and not the amount of available memory. For example the list L_{54} needs several terabytes to be stored.

Algorithm 1 Recursive Depth first search exploration of the tree $\mathcal{T}_g$.

```
procedure ExploreRec(S, G)
    if $g(S) < G$ then
        for x from $c(S)$ to $c(S) + m(S)$ do
            if $x \in \operatorname{Irr}(S)$ then
                ExploreRec($S^x$, G)
            end if
        end for
    end if
end procedure
```

Equivalently, we can use an iterative version which uses a stack:

In Algorithm 1 we do not specify how to compute $c(S)$, $g(S)$ and $m(S)$ from S neither how to test if an integer is irreducible. It also misses the characterization of S^x from S. These items depend heavily of the representation of S. Our choice is to use the S-decomposition numbers function. The first task is to use a finite set of such numbers to characterize the whole semigroup.

Proposition 3.1. *Let G be an integer and S be a numerical semigroup of genus $0 < g \leqslant G$. Then S is entirely described by the vector $\delta_S = (d_S(0), ..., d_S(3\,G)) \in \mathbb{N}^{3G+1}$. More precisely we can obtain $c(S), g(S), m(S)$ and $\operatorname{Irr}(S)$ from δ_S.*

Algorithm 2 Iterative Depth first search exploration of the tree $\mathcal{T}_g$

```
1: procedure EXPLORE(G)
2:     Stack stack                                         ▷ the empty stack
3:     stack.push($\mathbb{N}$)
4:     while stack is not empty do
5:         S $\leftarrow$ stack.top()
6:         stack.pop()
7:         if $g$(S) $<$ G then
8:             for x from $c$(S) to $c$(S) $+ m$(S) do
9:                 if x $\in$ Irr(S) then
10:                    stack.push(S$^{x}$)
11:                end if
12:            end for
13:        end if
14:    end while
15: end procedure
```

Proof. By Proposition 1.4 *iii)* we have the relation $c(S) \leqslant 2g(S)$ and so the S-decomposition number of $c(S)$ occurs in δ_S. Proposition 3.1 implies

$$c(S) = 1 + \max\{i \in \{0, ..., 3G\},\ d_S(i) = 0\}.$$

As all elements of $\mathbb{N} \setminus S$ are smaller than $c(S)$, their S-decomposition numbers are in δ_S and we obtain

$$g(S) = \operatorname{card}\{i \in \{0, ..., 3G\},\ d_S(i) = 0\}.$$

By Proposition 1.4 *ii)*, the relation $m(S) \leqslant g(S)+1$ holds. This implies that the S-decomposition number of $m(S)$ appears in δ_S :

$$m(S) = \min\{i \in \{0, ..., 3G\},\ d_S(i) > 0\}.$$

By Proposition 1.4, all irreducible elements are smaller than $c(S) + m(S) - 1$, which is itself smaller than $3G$. Hence, Proposition 2.4 gives

$$\operatorname{Irr}(S) = \{i \in \{0, ..., 3G\},\ d_S(i) = 1\}. \qquad \square$$

The previous representation of numerical semigroup (in terms of the vector δ_S) is similar but a little different from this used in [**?**] and by people concerned with coding theory.

Even though it is quite simple, the computation of $c(S), m(S)$ and $g(S)$ from δ_S has a non negligible cost. We represent a numerical semigroup S of genus $g \leqslant G$ by $(c(S), g(S), c(S), \delta_S)$. In an algorithmic context, if the variable `S` stands for a numerical semigroups we use:

– `S.c`, `S.g` and `S.m` for the integers $c(S)$, $g(S)$ and $m(S)$;
– `S.d[i]` for the integer $d_S(i)$.

For example the following Algorithm initializes a representation of the semigroup $\mathbb{N}$ ready for an exploration of the tree $\mathcal{T}_G$ (the tree of numerical semigroup of genus at most G.)

Algorithm 3 Returns the root of $\mathcal{T}_G$

```
function ROOT(G)
    R.c $\leftarrow 0$                                     ▷ R stands for $\mathbb{N}$
    R.g $\leftarrow 0$
    R.m $\leftarrow 1$
    for x from 0 to 3G do
        R.d[x]$\leftarrow 1 + \lfloor \frac{x}{2} \rfloor$
    end for
    return R
end function
```

We can now describe an algorithm that returns the representation of the semigroup S^x from that of the semigroup S where x is an irreducible element of S greater than $c(S)$.

Proposition 3.2. *Running on (S, x, G) with $g(S) \leqslant G$, $x \in \operatorname{Irr}(S)$ and $x \geqslant c(S)$, Algorithm 4 returns the semigroup S^x in time $O(\log(G) \times G)$.*

Algorithm 4 Returns the son S^x of S with $x \in \mathrm{Irr}(S) \cap [c(S), c(S) + m(S)[$.

```
 1: function SON(S,x,G)
 2:     $S^x.c \leftarrow x + 1$
 3:     $S^x.g \leftarrow S.g + 1$
 4:     if $x > S.m$ then
 5:         $S^x.m \leftarrow S.m$
 6:     else
 7:         $S^x.m \leftarrow S.m + 1$
 8:     end if
 9:     $S^x.d \leftarrow S.d$                          ▷ copy all the decomposition numbers
10:     for y from x to 3 G do
11:         if $S.d[y - x] > 0$ then
12:             $S^x.d[y] \leftarrow S.d[y] - 1$         ▷ decrease the decomposition number by 1
13:         end if
14:     end for
15:     return $S^x$
16: end function
```

Proof. Let us check the correctness of the algorithm. By construction S^x is the semigroup $S \setminus \{x\}$. Thus the genus of S^x is $g(S) + 1$, see Line 3. Every integer of $I = [x + 1, +\infty[$ lies in S since x is greater than $c(S)$, so the interval I is included in S^x. As x does not belong to S^x, the conductor of S^x is $x+1$, see Line 2. For the multiplicity of S^x we have two cases. First, if $x > m(S)$ holds then $m(S)$ is also in S^x and so $m(S^x)$ is equal to $m(S)$. Assume now $x = m(S)$. The relation $x(S) \geqslant c(S)$ and the characterization of $m(S)$ implies $x = m(S) = c(S)$. Thus S^x contains $m(S) + 1$ which is $m(S^x)$. The initialization of $m(S^x)$ is done by Lines 4 to 8. The correctness of the computation of δ_{S^x} (see Proposition 3.1 for the definition of δ_{S^x}) done from Line 9 to Line 15 is a direct consequence of Proposition 2.4.

Let us now prove the complexity statement. Since by relations *ii*) and *iii*) of Proposition 1.4 we have $x \leqslant 3G$ together with $m(S) \leqslant G+1$, each line from 2 to 8 is done in time $O(\log(G))$. The **for** loop needs $O(G)$ steps and each step is done in time $O(\log(G))$. Summarizing, these results give that the algorithm runs in time $O(\log(G) \times G)$. ☐

Algorithm 5 Returns an array containing the value of n_g for $g \leqslant G$

```
 1: function COUNT(G)
 2:     $n \leftarrow [0, ..., 0]$                 ▷ n[g] stands for $n_g$ and is initialized to 0
 3:     Stack stack                                      ▷ the empty stack
 4:     stack.push(ROOT(G))
 5:     while stack is not empty do
 6:         $S \leftarrow$ stack.top()
 7:         stack.pop()
 8:         $n[S.g] \leftarrow n[S.g] + 1$
 9:         if $S.g < G$ then
10:             for x from $S.c$ to $S.c + S.m$ do
11:                 if S.d[x] = 1 then
12:                     stack.push(SON(S, x, G))
13:                 end if
14:             end for
15:         end if
16:     end while
17:     return n
18: end function
```

Proposition 3.3. *Running on $G \in \mathbb{N}$, Algorithm 5 returns the values of n_g for $g \leqslant G$ in time*

$$O\left(\log(G) \times G \times \sum_{g=0}^{G} n_g\right)$$

and its space complexity is $O(\log(G) \times G^3)$.

Proof. The correctness of the algorithm is a consequence of Proposition 3.2 and of the description of the tree $\mathcal{T}$ of numerical semigroups.

For the time complexity, let us remark that Algorithm Son is called for every semigroup of the tree $\mathcal{T}_G$ (the tree of semigroups of genus $\leqslant G$). Since there are exactly $N = \sum_{g=0}^{G} n_g$ such semigroups, the time complexity of Son established in Proposition 3.2 guarantees that the running time of Count is in $O(\log(G) \times G \times N)$, as stated.

Let us now prove the space complexity statement. For this we need to describe the stack through the run of the algorithm. Since the stack is filled with a depth first search algorithm, it has two properties. The first one is that reading the stack from the bottom to the top, the genus increases. The second one is that, for all $g \in [0, G]$, every semigroup of genus g in the stack has the same father. As the number of sons of a semigroup S is the number of S-irreducible elements in the set $\{c(S), ..., c(S) + m(S) - 1\}$, a semigroup S has at most $m(S)$ sons. By Proposition 1.4 *ii*), this implies that a semigroup of genus g has at most $g + 1$ sons. Therefore the stack contains at most $g + 1$ semigroups of genus $g + 1$ for $g \leqslant G$. So the size of the stack is bounded by

$$M = \sum_{g=0}^{G} g = \frac{G(G+1)}{2}.$$

A semigroup S is represented by a quadruple $(c(S), g(S), m(S), \delta_S)$. By relations *ii*) and *iii*) of Proposition 1.4, we have $c \leqslant 2g(S)$ and $m \leqslant g(S) + 1$. As $g(S) \leqslant G$ holds, the integers c, g and m of the representation of S require a memory space in $O(\log(G))$. The size of $\delta_S = (d_S(0), ..., d_S(3G))$ is exactly $3G + 1$. Each entry of δ_S is the S-decomposition number of an integer smaller than $3G$ and hence requires $O(\log(G))$ bytes of memory space. Therefore the space complexity of δ_S is in $O(\log(G) \times G)$, which implies that the space complexity of the Count algorithm is

$$O(\log(G) \times G \times M) = O(\log(G) \times G^3). \qquad \square$$

4. Technical optimizations and results

Even though there are asymptotically faster algorithms than the one presented here, thank to careful optimizations, we were able to compute n_g for much larger genuses than before. This is due to the fact that our algorithm is particularly well suited for the current processor architecture. In particular, it allows to use parallelism at various scales (parallel branch exploration, vectorization)...

To get the greatest speed from modern processors, we used several optimization tricks, which we will elaborate in the following section:

- Vectorization (`MMX`, `SSE` instructions sets) and careful memory alignment;
- Shared memory multi-core computing using `Cilk++` for low level enumerating tree branching;
- Partially derecursived algorithm using a stack;
- Avoiding all dynamic allocation during the computation: everything is computed “in place”;
- Avoiding all unnecessary copy (the cost of the Son algorithm is roughly the same as copying);
- Aggressive loop unrolling: the main loop is unrolled by hand using some kind of Duff’s device;
- Careful choice of data type (`uint_fast8_t` for decomposition number, vs `uint_fast64_t` for all indexes).

The source code of our algorithm is available in [**?**].

4.1. Vectorization.

Assume for example that we want to construct the tree $\mathcal{T}_{100}$ of all numerical semigroups of genus smaller than 100. In this case, the representation of numerical semigroups given in Section 2 uses decomposition numbers of integers smaller than 300. By Lemma 2.3, such a decomposition number is smaller than 151 and requires 1 byte of memory. Thus at each **for** step of Algorithm Son, the CPU actually works on 1 byte. However current CPUs usually work on 8 bytes and even on 16 bytes using vector extensions. The first optimization uses this point.

To go further we must specify that the array of decomposition numbers in the representation of a semigroup corresponds to consecutive bytes in memory. In the **for** loop of Algorithm Son we may imagine two cursors: the first one, denoted `src` pointing to the memory byte of `S.d[0]` and the second one, denoted `dst` pointing to the memory byte `T.d[y]`. Using these two cursors, Lines `10` to `14` of Algorithm 4 can be rewritten as follows:

```
src ← address(S.d[0])
dst ← address(T.d[x])
```

```
i ← 0
while i ⩽ 3G − x do
    if content(src) > 0 then
        decrease content(dst) by 1
    end if
    increase src,dst,i by 1
end while
```

In this version we can see that the cursors `src` and `dst` move at the same time and that the modification of the value pointed by `dst` only needs to access the values pointed by `src` and `dst`. We can therefore work in multiple entries at the same time without collision. Current CPUs allow this thanks to the `SIMD` technologies as `MMX`, `SSE`, etc. The acronym `SIMD` [9] stands for Single Operation Multiple Data. We used `SSE4.1` [10, 11] technology as it allows for the largest speedup[1]. This need to respect some constraints in the memory organization of the data, namely “memory alignment”. Recall that an address is 16 bytes aligned if it is a multiple of 16. `SSE` memory access are much faster for aligned memory.

The computation of the children is then performed as follows. First, the parent’s decomposition numbers are copied in the children’s using the following `C++` code

```
void copy_blocks(dec_blocks &dst, dec_blocks const &src)
  for (ind_t i=0; i<NBLOCKS; i++) dst[i] = src[i];
```

Here `dec_blocks` is a type for arrays of 16 bytes blocks whose size `NBLOCKS` are just large enough to store the decomposition numbers (that is $3G$ rounded up to a multiple of 16). The instruction `dst[i] = src[i]` actually copies a full 16 bytes block.

Then the core of the **while** loop in the preceding algorithm is translated as a `for` loop as follows (recall that x denotes the generator of the father which is to be removed in the children):

```
start = x >> 4;      // index of the block containing x
shift = x & 0xF;     // offset of x inside the block
...                  // some specific instructions to handle the first incomplete block.
for (long int i=start+1; i<NBLOCKS; i++)
  block = load_unaligned_epi8(src + ((i-start)<<4) - shift);
  dst[i] -= ((block != zero) & one);
```

The instruction `load_unaligned_epi8` (specific to `SSE` technology) loads 16 consecutive entries of the decomposition number of the parent (called `src` semigroup) in the variable `block`. Those entries will be used to compute the entries $16i, \dots 16i+15$ of the children semigroups. Since the removed generator x is not necessarily a multiple of 16, the data are not aligned in memory, hence the use of a specific instruction. The `zero` (resp. `one`) constants are initialized as 16 bytes equal to 0 (resp. 1). The comparison `(block != zero)` therefore returns a block which contains 0 in the bytes corresponding the 0 entries of block and 255 in the non zero one. This result is then bitwise and-ed with one so that the instruction actually performs a 16 bytes parallel version of

$$\mathtt{dst} \leftarrow \mathtt{dst} - \textbf{ if } \mathtt{block} \neq 0 \textbf{ then } 1 \textbf{ else } 0$$

which is equivalent to Lines `10` to `14` of Algorithm 4.

As we previously said, to gain more speed this core loop is actually unrolled using some kind of Duff device [12].

4.2. Parallel tree exploration using `Cilk++`.

Our second optimization is to use parallelism on exploration of the tree. Today, CPUs of personal computers have several cores (2, 4 or more). The given version of our exploration algorithm uses a single core and so a fraction only of the power of a CPU. The idea here is that different branches of the tree can be explored in parallel by different cores of the computer. The tricky part is to ensure that all cores are busy, giving a new branch when a core is done with a former one. Fortunately there is a technology called `Cilk++` [13, 14] which is particularly well suited for those kinds of problems. For our computation, we used the free version which is integrated in the latest version of the GNU C compiler [15].

`Cilk` is a general-purpose language designed for multithreaded parallel computing. The `C++` incarnation is called `Cilk++`. The biggest principle behind the design of the `Cilk` language is that the programmer should be responsible for *exposing* the parallelism, identifying elements that can safely be executed in parallel; it should then be left to the run-time environment, particularly the scheduler, to decide during execution how to actually divide the work between cores.

The crucial thing is that two keywords are all that are needed to start using the parallel features of `Cilk++`:

[1]A much greater speedup can be certainly obtained using `AVX2` technology [?]. However, at this time, we cannot access a performant computer with this set of instructions.

– `cilk_spawn`: used on a procedure call, indicates that the call can safely operate in parallel with the remaining code of the current function. Note that the scheduler is not obliged to run this procedure in parallel; the keyword merely alerts the scheduler that it can do so.
– `cilk_sync`: indicates that execution of the current procedure cannot proceed until all previously spawned procedures have completed and returned their results to the current frame.

As a consequence, to get a parallel version of the recursive Algorithm 1, one only needs to modify it as

Algorithm 6 `Cilk` version of Algorithm 1

```
procedure ExploreRec(S, G)
    if $g(\mathtt{S}) < \mathtt{G}$ then
        for x from $c(\mathtt{S})$ to $c(\mathtt{S}) + m(\mathtt{S})$ do
            if $\mathtt{x} \in \mathrm{Irr}(\mathtt{S})$ then
                cilk_spawn ExploreRec($\mathtt{S}^{\mathtt{x}}$, G)
            end if
        end for
    end if
end procedure
```

We just tell `Cilk++` that the subtrees rooted at various children can be explored in parallel. Things are actually only a little bit more complicated. First we have to gather the results of the exploration. If we simply write

$$\mathtt{result}[g(\mathtt{S})] \leftarrow \mathtt{result}[g(\mathtt{S})] + 1$$

then we face the problem of two cores incrementing the same variable at the same time. Incrementing a variable is actually done in 3 steps: reading the value from the memory, adding one, storing back the result. Since there is by default non synchronization, the following sequence of actions for two cores is possible: Read1 / Read2 / Add1 / Add2 / Store1 / Store2. Then the two cores perform the same modification resulting in incrementing the variable only once. This is called a data race and leads to nondeterministic wrong results. To cope with those kinds of synchronization problems, `Cilk++` provides the notion of *reducer* which are variables local to each thread which are gathered (here added) when a thread is finishing its job.

A more important problem is that the cost of a recursive call in non negligible. Using `Cilk++` recursive calls instead of `C++` calls makes it even worse. The solution we use is to switch back to the non recursive version using a stack when the genus is close to the target genus. This leads to the following `Cilk++` code:

```
void explore(const Semigroup &S) {
  unsigned long int nbr = 0;
  if (S.g < MAX_GENUS - STACK_BOUND) {
    auto it = generator_iter<CHILDREN>(S);
    while (it.move_next()) { //iterate along the children of S
      auto child = remove_generator(S, it.get_gen()).
      cilk_spawn explore(child);
      nbr++;
    }
    cilk_results[S.g] += nbr;
  }
  else explore_stack(S, cilk_results.get_array());
}
```

Note that in our version, we found that the `STACK_BOUND` optimal value was around 10 to 12 for genus in the range $45 \ldots 67$ so that `explore_stack` is used more than 99% of the time. The `Cilk++` recursive function does actually very little work but ensures that the work is balanced between the different cores.

4.3. **Various technical optimizations.** Using vectorization and loop unrolling as described previously leads to an extremely fast Son algorithm. Indeed, its cost is comparable to the cost of copying a semigroup. It is therefore crucial for performance to avoid any extra cost. We list here various places where unnecessary cost can be avoided.

Avoiding all unnecessary copy. We also used a trick to avoid copying from and to the top of the stack. Indeed, the main loop performs the following sequences of operations:

```
S ← stack.top()
stack.pop()
```

```
for all children S^x of S do
    S.push(S^x)
end for
```

If we use a stack of semigroup, we can construct $\mathtt{S}^{\mathtt{x}}$ directly into the stack memory but we have to copy the top of the stack to `S`. In [16], A. Zhai establishes that the limit of the quotient $\frac{n_g}{n_{g-1}}$, when g go to infinity, is the golden ratio $\phi \approx 1.618$. Therefore this single copy is far from being negligible. The trick is to use a level of indirection, replacing the stack by an array of semigroups A and an array of indexes I pointing to the array of semigroups. The array I can be viewed as a permutation of the array A. Now instead of copying `S` out of the stack, we keep it on the stack, pushing the children in the second position by exchanging the indexes in I. Here is the relevant part of the code:

```
Semigroup data[MAX_GENUS-1], *stack[MAX_GENUS], *current;
Semigroup **stack_pointer = stack + 1;
for (ind_t i=1; i<MAX_GENUS; i++) stack[i] = &(data[i-1]);
[...]
while (stack_pointer != stack) {
    --stack_pointer;
    current = *stack_pointer;
    [...] for each children {
      *stack_pointer = *(stack_pointer + 1);
      [...] construct the children in **stack_pointer
      [...] using the parent in *current
      stack_pointer++;
      [...]
    }
    *stack_pointer = current;
}
```

Avoiding dynamic allocation. Compared to the Son algorithm, dynamic allocation costs orders of magnitude more. Therefore, during the derecursived stack algorithm, we only allocate (on the system stack rather than on the heap) the stack of semigroups. No further allocations are done.

Pointer arithmetic and indexes. Due to the way `C++` does its pointer arithmetic, even if the index in the array are less than $3G$ and therefore fits in 8 bits, we use 64 bits indexes (namely `uint_fast64_t`) to avoid conversion and sign extension when computing addresses of indexed elements. This single standard trick save 10% of speed.

5. Results

Running the `Cilk` version of our optimized algorithm we have explored the tree of numerical semigroups up to genus 67. The computations were done on a shared 64 core AMD Opteron$^{\text{TM}}$ Processor 6276. As other heavy calculations were running on the machine, we only used 32 cores. The computations took 18 days. The values of n_g for $g \leqslant 67$ are:

g	n_g	g	n_g	g	n_g
0	1	23	170 963	46	14 463 633 648
1	1	24	282 828	47	23 527 845 502
2	2	25	467 224	48	38 260 496 374
3	4	26	770 832	49	62 200 036 752
4	7	27	1 270 267	50	101 090 300 128
5	12	28	2 091 030	51	164 253 200 784
6	23	29	3 437 839	52	266 815 155 103
7	39	30	5 646 773	53	433 317 458 741
8	67	31	9 266 788	54	703 569 992 121
9	118	32	15 195 070	55	1 142 140 736 859
10	204	33	24 896 206	56	1 853 737 832 107
11	343	34	40 761 087	57	3 008 140 981 820
12	592	35	66 687 201	58	4 880 606 790 010
13	1 001	36	109 032 500	59	7 917 344 087 695
14	1 693	37	178 158 289	60	12 841 603 251 351
15	2 857	38	290 939 807	61	20 825 558 002 053
16	4 806	39	474 851 445	62	33 768 763 536 686
17	8 045	40	774 614 284	63	54 749 244 915 730
18	13 467	41	1 262 992 840	64	88 754 191 073 328
19	22 464	42	2 058 356 522	65	143 863 484 925 550
20	37 396	43	3 353 191 846	66	233 166 577 125 714
21	62 194	44	5 460 401 576	67	377 866 907 506 273
22	103 246	45	8 888 486 816		

As the reader can check the convergence of sequence $\frac{n_g}{n_{g-1}}$ established by A. Zhai in [16] is very slow: $\frac{n_{67}}{n_{66}} \approx 1.62$.

5.1. **Wilf's conjecture.** In the paper [?] of 1978, H.S. Wilf conjectured that all numerical semigroup S satisfy the relation

$$\operatorname{card}(\operatorname{Irr}(S)) \geqslant \frac{c(S)}{c(S) - g(S)}.$$

Since the work of M. Bras-Amorós, see [2], we yet know that all numerical semigroups of genus $g \leqslant 50$ satisfy Wilf's conjecture. With our exploration algorithm we have proved that there is no counterexample to Wilf's conjecture up to genus 60. We have tested the Wilf's conjecture on a different machine than the one used to determine n_{67}. As its performance is lower we have only tested the conjecture for $g \leqslant 60$.

5.2. **Timings.** In this section we summarize the timing improvements through the different optimizations of our algorithm.

The following table shows the time needed by the algorithm for computing the values of n_g for $g \leqslant G$ with $30 \leqslant G \leqslant 40$ on a machine equipped with an `Intel`™ `i5-3570K` CPU running at `3.4GHz` and `8GB` of memory. All algorithms are executed on only one core. Algorithm `breadth` is based on a breadth exploration of the tree while Algorithm `depth` use a depth exploration. These two algorithms are based on the same naive representation of numerical semigroups. The only difference concerns the tree exploration algorithm used. Algorithm `depth+`δ is a refinement of `depth` based on the S-decomposition function. Algorithm δ`+sse` in an optimization of `depth+`δ using the `SIMD` extension `SSE`. Times are in seconds.

`Algorithm`	30	31	32	33	34	35	36	37	38	39	40
`breadth`	5.0	8.3	14	23	38	1251					
`depth`	3.4	5.8	9.2	16	27	45	75	125	204	346	557
`depth+`δ	0.3	0.6	1.0	1.7	2.7	4.2	7.4	12	20	32	74
δ`+sse`	0.1	0.2	0.3	0.4	0.8	1.2	2.0	3.1	5.1	9.0	14

The computation of n_g for $g \leqslant 35$ with algorithm `breadth` is very long because all the `8GB` of memory are consumed and the system must use swap memory to finish the computation. This algorithm was not launched for genus $g \geqslant 36$.

The following table illustrates the impact of parallelization on the same machine based on the `Intel`™ `i5-3570K` CPU which have 4 physical cores that are able to run four threads.

The time of the one threaded algorithm must be compared with the δ`+sse` version of the previous table : it illustrates the additional cost induced by the use of `Cilk` technology.

It should be noticed that the `TurboBoost` technology [?] is present on the CPU. Therefore the clock of the CPU is a bit higher when the number of threads is smaller. Therefore the gain of using more threads is a bit over than the one suggestested by the table.

Finally we tested our algorithm on a machine holding two `Intel`$^{\text{TM}}$ `Xeon`$^{\text{TM}}$ `X5650` CPU running at `2.67GHz`. Each of those CPU has 6 physical cores that are able to run 12 threads thanks to the `Hyper-Threading` technology [**?**]: the machine has also 12 physical cores and is able to run 24 threads. However, when more than 12 threads are running, the computation engines are shared between two threads and the speedup should be much less when adding more cores. The following table resumes the time needed by the algorithm to explore $\mathcal{T}_{50}$ on different numbers of threads (the double vertical line delimite the use of `Hyper Threading`). For reference, we put in the column called `C++` the computation time of the same program compiled without `Cilk`.

Further improvements on the computation of n_g or on the verification of Wilf's conjecture will be published on our home pages and on [**?**].

Acknowledgment. We would like to thank Shalom Eliahou for introducing us to the tree of numerical semigroups and for interesting discussions on the subject. We are grateful to Laura Grigori who suggested us that Cilk could be a good technology to tackle the tree-level parallelism for this kind of problems. We would finally like to thanks Nathan Cohen who suggested the two levels of indirection stack optimization.

References

[1] N. Sloane, "The on-line encyclopedia of integer sequences."

[2] M. Bras-Amorós, "Fibonacci-like behavior of the number of numerical semigroups of a given genus," *Semigroup Forum*, vol. 76, no. 2, pp. 379–384, 2008.

[3] M. Delgado, "Homepage."

[4] J. C. Rosales, "Fundamental gaps of numerical semigroups generated by two elements," *Linear Algebra Appl.*, vol. 405, pp. 200–208, 2005.

[5] J. M. Manuel Delgado, Pedro A. Garcia-Sanchez, *NumericalSgps – GAP package, Version 0.971*. T, 2011.

[6] The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.6.3*, 2013.

[7] J. C. Rosales and P. A. García-Sánchez, *Numerical semigroups*, vol. 20 of *Developments in Mathematics*. New York: Springer, 2009.

[8] J. L. Ramírez Alfonsín, *The Diophantine Frobenius problem*, vol. 30 of *Oxford Lecture Series in Mathematics and its Applications*. Oxford: Oxford University Press, 2005.

[9] Wikipedia, "Simd," 2014.

[10] Wikipedia, "Streaming simd extensions," 2014.

[11] M. Girkar, *Intel Instruction Set Architecture Extensions — Intel® Developer Zone*. Software.intel.com, 2013.

[12] Wikipedia, "Duff's device," 2014.

[13] "Intel® cilk$^{\text{TM}}$ homepage," 2013.

[14] Software.intel.com, *Intel® Cilk^{TM} Plus Reference Guide*, 2013.

[15] P. H. Balaji V. Iyer, Robert Geva, "Cilk$^{\text{TM}}$+ in gcc," in *GNU Tools Cauldron*, 2012.

[16] A. Zhai, "Fibonacci-like growth of numerical semigroups of a given genus," *Semigroup Forum*, vol. 86, no. 3, pp. 634–662, 2013.

Jean Fromentin
ULCO, LMPA J. Liouville, B.P. 699, F-62228 Calais, France
CNRS, FR 2956, France
`fromentin@lmpa.univ-littoral.fr`

Florent Hivert
Bureau 33, Laboratoire de Recherche en Informatique (UMR CNRS 8623)
Bâtiment 650, Université Paris Sud 11, 91405 Orsay CEDEX, France
`florent.hivert@lri.fr`

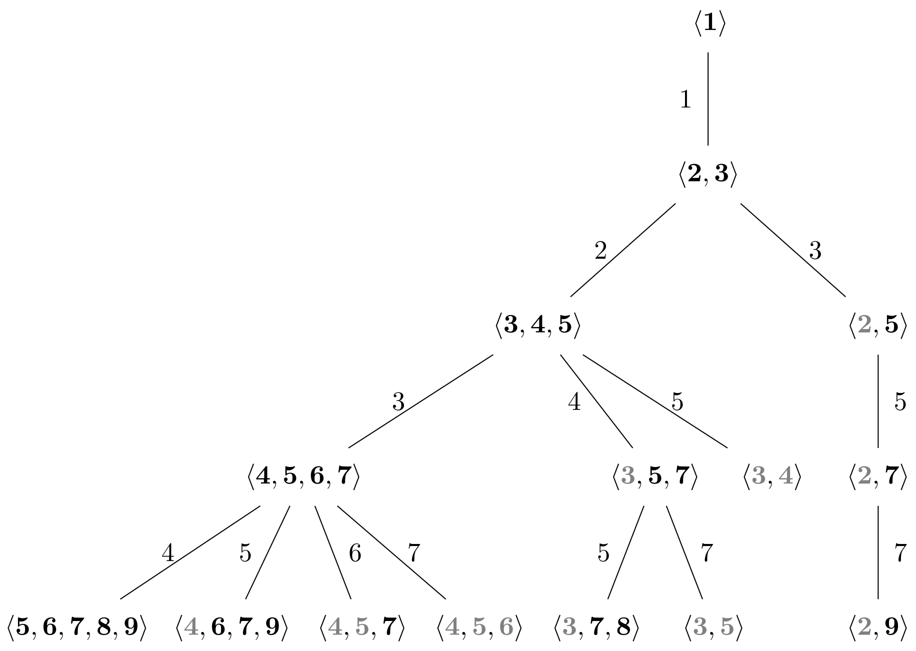

FIGURE 1. The first four layers of the tree $\mathcal{T}$ of numerical semigroups, corresponding to $\mathcal{T}_4$. A generator of a semigroup is it in gray if is not greater than $c(S)$. An edge between a semigroup S and its son S' is labelled by x if S' is obtained from S by removing x, that is if $S' = S^x$ holds.

Threads	30	35	40	45	50
1	0.11	1.26	14.9	182	2201
2	0.06	0.65	7.50	92	1110
3	0.05	0.44	5.14	63	747
4	0.04	0.34	4.02	48	489

Threads	`C++`	1	2	4	8	12	16	20	24
Time (s)	3588	3709	1865	932.4	486.8	325.7	311.2	302.3	290.2