EXPLORING THE TREE OF NUMERICAL SEMIGROUPS

JEAN FROMENTIN AND FLORENT HIVERT

Abstract. In this paper we describe an algorithm visiting all the numerical semigroups up to a given genus using a new representation of numerical semigroups. The interest of this algorithm is that it fits particularly well the architecture of modern computer 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, close 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* [10] we find the values of n_g for $g \leqslant 52$. These values have been obtain by M. Bras-Amorós (view [1] for more details for $g \leqslant 50$). On his home page [3], M. Delgado gives the value of n_{55} without specifying the values of n_{53} and n_{54}.

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 [9] and it is the subject of the 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` [4] of `GAP` [5], 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 semigroup $\mathcal{T}$ and achieve the computation of n_g for $g \leqslant 67$. The cornerstone of our method is a new combinatorial representation of numerical semigroup that is well suited for exploration of the tree $\mathcal{T}$ and allow large code optimization essentially based on use of vectorial instructions and parallelization. The goal of the paper is twofold: first to present our new encoding of the semigroup and the associated algorithms, and second to present the optimization techniques which allows for those kinds of algorithms to get speedups by factors of hundreds and even thousands. We

claim that these technique 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 [2] 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 decription of our new representation of numerical semigroup is done in the third 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 optimisation of the algorithm introduced in Section 3. Il the last section we emphase 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 consults the book *Numerical Semigroups* of J.C. Rosales and P.A. García-Sánchez [6] or the book *The Diophantine Frobenius Problem* of J.L. Ramírez Alfonsín [8].

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) $c(S) = 1 + \max(\mathbb{N} \setminus S)$, the *conductor* of S for S different from $\mathbb{N}$. By convention, the conductor of $\mathbb{N}$ is 0.

By definition a numerical semigroup is infinite object and we need a finite description of such a object. That is the role of 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-zeros elements of S. We denote by $\mathrm{Irr}(S)$ the set of all irreducible elements of S.

Lemma 1.3 (Lemma 2.3, Chap. I, [6])**.** 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. *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) $c(S) \leqslant 2g(S)$*.*

Proof. *i)* Let x ba a positive integer satisfying $x \geqslant c(S) + m(S)$. As the integers x and $x - m(S)$ are greater than $c(S)$, they belong to S. Since x equals $(x - m(S)) + m(S)$ with $x - m(S) \neq 0$ and $m(S) \neq 0$, it could not be irreducible.

ii) The set $\mathbb{N} \setminus S$ of cardinality $g(S)$ contains $\{1, .., m(S) - 1\}$ and so the relation $m(S) \leqslant g(S) + 1$ holds.

iii) Let x be an element of $S \cap \{0, ..., c(S)-1\}$. The integer $y = c(S)-1-x$ lies in $\{0, ..., c(S) - 1\}$. Moreover y is not in S, for otherwise we write $c(S) - 1 = y + x$ where x, y are elements of S implying $c(S) - 1 \in S$. In contradiction with the definition of conductor. Thus we define an involution

$$\begin{array}{rcl} \psi : \{0, ..., c(S) - 1\} & \to & \{0, ..., c(S) - 1\} \\ x & \mapsto & c(S) - 1 - x \end{array}$$

that send $S\{0, ..., c(S) - 1\}$ into $\{0, ..., c(S) - 1\} \setminus S$ which is of cardinality $g(S)$. Let us denote by k the cardinality of $S \cap \{0, ..., c(S) - 1\}$. Since ψ is injective we must have $k \leqslant g(S)$. From the relation

$$\{0, ..., c(S) - 1\} = S \cap \{0, ..., c(S) - 1\} \sqcup \{0, ..., c(S) - 1\} \setminus S$$

we obtain $c(S) = k + g(S)$ and so $c(S) \leqslant 2g(S)$. □

A consequence of Proposition 1.4 *i)* is that $\mathrm{Irr}(S)$ is finite and its cardinality is at most $c(S) + m(S) - 1$. More precisely, using Apéry set, we can prove that the cardinality of $\mathrm{Irr}(S)$ is at most $m(S)$. See Section 2 of Chapter I of [6] for more details.

We now explain the construction of the tree of numerical semigroups. Let S be a numerical semigroup. The set $S' = S \cup \{c(S) - 1\}$ is also a numerical semigroup and its genus is $g(S) - 1$. As each integer greater than $c(S) - 1$ is included in S' we have $c(S') \leqslant c(S) - 1$. 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. *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.*

Proof. If x is not irreducible in S, then there exist a and b in $S \setminus 0$ such that $x = a+b$. Since $a \neq 0$ and $b \neq 0$ hold, the integers a and b belong to $S \setminus \{x\}$. Since $x = a + b$ and $x \notin S^x$, it follows that S^x is not stable under addition.

Conversely, assume that x is irreducible in S. As 0 is never irreducible, the set S^x contains 0. Let a and b be two integers belonging to S^x. The set S is stable under addition, hence $a+b$ lies in S. As S is equal to $S^x \cup \{x\}$, the integer $a+b$ also lies in S^x except if it is equal to x. The latter is impossible since a and b are different from x and x is irreducible. □

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 $c(S)$. 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 semigroup 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 $\mathcal{T}$ resticted to all semigroup of genus $\leqslant g$.

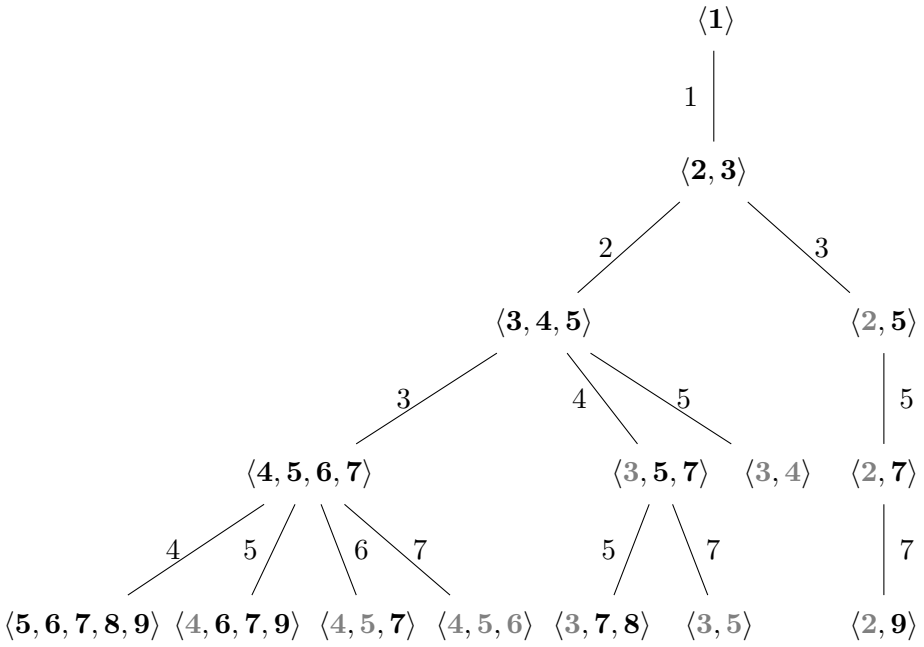

FIGURE 1. The first four layers of the tree $\mathcal{T}$ of numerical semigroups, corresponding to $\mathcal{T}_4$. A generator of a semigroup is in gray if is not greater that $c(S)$. An edge between a semigroup S and its son S' is labelled by x is S' is obtained from S by removing x, that is $S' =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

As the reader can check, the main difficulty to characterize the son of a semigroup is to determine its irreducible elements. In [4], the semigroup are given by their Apéry set and then the difficulty is to describe the Apéry set of S^x from this of S. This approach is elegant but not sufficiently *basic* for our optimisations.

2. Decomposition number

The aim of this section is to describe a new 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 $d_S(x) = \operatorname{card} D_S(x)$. 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 very 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 is why we decide to call $d_S(x)$ the S-decomposition number of x.

Example 2.2. Reconsider the semigroup S_E given at (1). The integer 14 admits three decompositions as sums of two elements of S, namely $14 = 0+14$, $14 = 3+11$ and $14 = 7+7$. Thus the set $D_{S_E}(14)$ is equal to $\{0, 3, 7\}$ and d_{S_E} equals 3.

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 $\{0, ..., \lfloor \frac{x}{2} \rfloor\}$, the relation $d_S(x) \leqslant 1 + \lfloor \frac{x}{2} \rfloor$ holds. For $S = \mathbb{N}$ we have the equality for $D_S(x)$ and so for $d_S(x)$. □

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 $d_S(x) = 1$.

Proof. We start with *i)*. If x is an element of S then x equals $0 + x$. The relation $2 \times 0 \leqslant x$ and $0 \in S$ imply that $D_S(x)$ contains 0, and so $d_S(x) > 0$ holds. Conversely, the relation $d_S(x) > 0$ implies that $D_S(x)$ is non-empty. Let y be an element of $D_S(x)$. As y and $x - y$ belong to S, by definition, the integer $x = (x - y) + y$ is in S (since S is stable by addition).

Let us show *ii)*. Assume x is irreducible in S. There cannot exist y and z in $(S \setminus \{0\})^2$ such that $x = y + z$. The only possible decomposition of x as a sum of two elements of S is $x = 0 + x$. Hence, the set $D_S(x)$ is equal to $\{0\}$ and we have $d_S(x) = 1$. Conversely, let x such that $d_S(x)$ is 1. By *i)* the integer x must be in S. As $x = 0 + x$ is always a decomposition of x as a sum of two elements in S, we obtain $D_S(x) = \{0\}$. If there exist y and z

in S such that $y \leqslant z$ and $x = y + z$ hold then y lies in $D_S(x)$. This implies $y = 0$ and $z = x$. Hence x is irreducible in S. □

We note that 0 is never irreducible despite the fact $d_S(0)$ is 1 for all numerical semigroup S.

We now explain how to compute the S-decomposition numbers function of a numerical semigroup from this 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. Let y be in $\mathbb{N} \setminus \{0\}$. We have

$$D_{S^x}(y) = \{z \in S^x \mid y - z \in S^x \text{ and } 2z \leqslant y\}.$$

We construct two sets as follow:

$$E = \{z \in S^x \mid y - z = x \text{ and } 2z \leqslant y\}$$

$$F = \begin{cases} \{x\} & \text{if } y - x \in S \text{ and } 2x \leqslant y, \\ \emptyset & \text{otherwise.} \end{cases}$$

We have the relation $D_S(y) = D_{S^x}(y) \sqcup E \sqcup F$ and so $d_S(y)$ is equal to $d_{S^x}(y) + \mathrm{card}(E) + \mathrm{card}(F)$ and $D_{S^x}(y)$ is a subset of $D_S(y)$. We start by determining E. The relation $y - z = x$ implies $z = y - x$ and $2z \leqslant y$ becomes $y \leqslant 2x$. Thus we obtain

$$E = \begin{cases} \{y - x\} & \text{if } y - x \in S^x \text{ and } y \leqslant 2x \\ \emptyset & \text{otherwise} \end{cases}$$

The relation $y - x \in S^x$ together with $y \leqslant 2x$ is equivalent to $y - x \in S$ together with $y < 2x$. Therefore we obtain

$$E = \begin{cases} \{y - x\} & \text{if } y - x \in S \text{ and } y < 2x \\ \emptyset & \text{otherwise} \end{cases}$$

Assume $y - x$ lies in S. If $y < 2x$ holds, we have $E = \{y - x\}$ and $F = \emptyset$. Otherwise, we obtain $E = \emptyset$ and $F = \{x\}$. Then if $y - x \in S$ holds, we have $\mathrm{card}(E) + \mathrm{card}(F) = 1$ and so $d_S(y)$ is equal to $d_{S^x}(y) + 1$. If $y - x$ not belong to S, then E and F are empty implying $d_S(y) = d_{S^x}(y)$. As the condition $y - x \in S$ is equivalent to $y \geqslant x$ together with $d_S(y - x) > 0$, by Proposition 2.4 the result holds. □

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. The cost to pay is that, if we want to explore the tree deeper, we must restart from the root.

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

```
1: procedure ExploreRec(S, G)
2:     if g(S) < G then
3:         for x from c(S) to c(S) + m(S) do
4:             if x ∈ Irr(S) then
5:                 ExploreRec(S^x, G)
6:             end if
7:         end for
8:     end if
9: end procedure
```

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

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

```
1: procedure Explore(G)
2:     Stack stack                                  ▷ the empty stack
3:     S ← ⟨1⟩
4:     while stack is not empty do
5:         S ← 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 ∈ Irr(S) then
10:                    S.push(S^x)
11:                end if
12:            end for
13:        end if
14:    end while
15: end procedure
```

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 miss the characterisation 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 characterise the whole semigroup.

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

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. Since $c(S)$ is equal to

$\max(\mathbb{N} \setminus S)$, 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

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

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 :

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

Since, by Proposition 1.4 *i*), all irreducible elements are smaller than $c(S) + m(S) - 1$, which is itself smaller than $3G$ from the relations *ii*) and *iii*) of Proposition 1.4. So Proposition 2.4 gives

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

□

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 semigroup 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 $\mathcal{T}_G$, *i.e*, the tree of numerical semigroup upto genus G.

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

```
function ROOT(G)
    R.c ← 1                                   ▷ R stands for ℕ
    R.g ← 0
    R.m ← 1
    for x from 1 to 3 G do
        R.d[x] ← 1 + ⌊x/2⌋
    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 $c(S)$.

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 ← x + 1
3:     S^x.g ← S.g + 1
4:     if x > S.m then
5:         S^x.m ← S.m
6:     else
7:         S^x.m ← S.m + 1
8:     end if
9:     S^x.d ← S.d
10:    for y from x to 3G do
11:        if S.d[y − x] > 0 then
12:            S^x.d[y] ← S^x.d[y] − 1
13:        end if
14:    end for
15:    return S^x
16: end function
```

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

Proof. Let us check the correctness of the algorithm. By construction S^x is the semigroup $S \setminus \{x\}$. Thus the genus S^x is $g(S)+1$, see Line 1. 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, 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 characterisation of $m(S)$ implies $x = m(S) = c(S)$. Thus S^x contains $m(S)+1$ which is $m(S^x)$. The initialisation of $m(S^x)$ is done by Lines 4 to 8. The correctness of the computation of δ_{S^x} (see Proposition 3.1) 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 $O(\log(G) \times G)$. □

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 $O(\log(G) \times G^3)$.

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

```
1: function Count(G)
2:     n ← [0, ..., 0]                    ▷ n[g] stands for n_g and is initialised to 0
3:     Stack stack                                              ▷ the empty stack
4:     S ← Root(G)
5:     while stack is not empty do
6:         S ← stack.top()
7:         stack.pop()
8:         n[S.g] ← 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:                    S.push(Son(S, x, G))
13:                end if
14:            end for
15:        end if
16:    end while
17:    return n
18: end function
```

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 of semigroup 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 $[c(S), c(S)+m(S)-1]$, a semigroup S has at most $m(S)$ sons. By Proposition 1.4 *ii*), this implies that a semigroup of genus g as at most $g+1$ sons. Therefore the stack contains at most $g+1$ semigroup of genus $g+1$ for $g \leqslant G$. So the size of the stack is bounded by

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

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 $\log(G)$ 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

$$O(\log(G) \times G \times S) = O(\log(G) \times G^3).$$

□

4. Technical optimizations and results

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 genus 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 scale (parallel branch exploration, vectorization)...

To get the greatest speed from modern processor, 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.
- Carefull choice of data type (`uint_fast8_t` for decomposition number, vs `uint_fast64_t` for all indexes).

4.1. **Vectorization.** Assume for example that we want to construct the tree $\mathcal{T}_{100}$ of all numerical semigroup of genus smaller than 100. In this case, the representation of numerical semigroup 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 SON 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` [16] stands for Single Operation Multiple Data. We used `SSE4.1` [17, 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 access.

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 copy 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

[1] A much more speedup can be certainly obtained using `AVX2` technology [14]. However, at this time, we can't access a performent computer with this set of instructions.

entries $16i, \dots 16i+15$ of the children monoid. 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)` therfore returns a block which contains 0 in the bytes corresponding the 0 entries of block and 255 in the non zero one. This results is then bitwise anded with one so that the instruction actually performs a 16 bytes parallel version of

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

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

As we previously said, to gain more speed this core loop is actually unrolled using some kind of Duff device [15].

4.2. **Parallel tree exploration using `Cilk++`.** Our second optimization is to use parallelism on exploration of the tree. Today, CPU of personal computer 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 each cores are busy, giving a new branch when a core is done with a former one. Fortunately there is a technology called `Cilk++` [12, 13] which is particularly well suited for those kind of problems. For our computation, we used the free version which is integrated in the latest version of the GNU C compiler [7].

`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 processors.

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

- `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 obligated 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, you only need 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**
        CilkSpawn ExploreRec($\mathtt{S}^{\mathtt{x}}, \mathtt{G}$)
      end if
    end for
  end if
end procedure

We just tells `Cilk++` that the subtrees rooted at various children can be explored in parallel. The thing 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 performs the same modification resulting in incrementing the variable only once. This is called a data race and leads to nondeterministic wrong result. To cope with those kinds of synchronization problems, `Cilk++` provide 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 `STACK_BOUND` optimal value was around 10-12 for genus in the range $45 \dots 67$ so that `explore_stack` is used more than 99% of the time. The `Cilk++` recursive function does actually very little work but ensure that the work is balanced between the different cores.

4.3. **Various technical optimizations.** Using vectorization and loop enrolling as described previously leads to an extremely fast Son algorithm. Indeed, its cost is comparable to the cost of copying a monoid. 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, then main loop perform 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 monoid, we can construct $\mathtt{S}^x$ directly into the stack memory but we have to copy the top of the stack to `S`. Since the average number of children is the golden ratio, this single copy is far from being negligible. The trick is to use a level of indirection, replacing the stack by an array of monoid and an array of indexes pointing to the array of monoid. We keep the invariant that the array of indexes is a permutation of the array of monoid. 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. 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 cost 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 monoid. No further allocation are done.

Pointer arithmetic and indexes. Due to the way, the `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 element. 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 semigroup upto genus 67. The computation were done one a shared 64 core AMD Opteron$^{\text{TM}}$ Processor 6276. As other heavy computation were running on the machine, we only used 32 cores. The computation 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	170963	46	14463633648
1	1	24	282828	47	23527845502
2	2	25	467224	48	38260496374
3	4	26	770832	49	62200036752
4	7	27	1270267	50	101090300128
5	12	28	2091030	51	164253200784
6	23	29	3437839	52	266815155103
7	39	30	5646773	53	433317458741
8	67	31	9266788	54	703569992121
9	118	32	15195070	55	1142140736859
10	204	33	24896206	56	1853737832107
11	343	34	40761087	57	3008140981820
12	592	35	66687201	58	4880606790010
13	1001	36	109032500	59	7917344087695
14	1693	37	178158289	60	12841603251351
15	2857	38	290939807	61	20825558002053
16	4806	39	474851445	62	33768763536686
17	8045	40	774614284	63	54749244915730
18	13467	41	1262992840	64	88754191073328
19	22464	42	2058356522	65	143863484925550
20	37396	43	3353191846	66	233166577125714
21	62194	44	5460401576	67	377866907506273
22	103246	45	8888486816		

In [19], A.Zhai establishes that the limit of the quotient $\frac{n_g}{n_{g-1}}$, when g go to infinity, is the golden ratio $\phi \approx 1.618$. As the reader can check the convergence is very slow : the quotient n_{67}/n_{66} is ≈ 1.62.

5.1. **Wilf's conjecture.** In the paper [18] 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 [1], we yet know that all numerical semigroup satisfy of genus $g \leqslant 50$ satisfy the Wilf's conjecture. With our exploration algorithm we have proved that there is no counterexample to the Wilf's conjecture of genus $g \leqslant 60$.

5.2. **Timings.** In this section we summarize the timing improvement through the different optimization of our algorithm. All the computations are done on a computer with an `i5-3570K` CPU equipped with `8GB` of memory.

The following table show the time needed by the algorithm for computing the values of n_g for $g \leqslant G$ with $30 \leqslant G \leqslant 40$. All the algorithm are executed on only one thread. Algorithm `depth` is based on a baser breadth exploration of the tree while Algorithm `depth` use a depth exploration. Algorithm `depth+`δ is a refinement of `depth` based on the the S-decomposition function. Algorithm δ`+sse` in an optimization of `depth+`δ using the `SIMD` extension `SSE`. Times are in second.

`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 $g \leqslant 33$ 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.

On the following table we illustrate the impact of parallelization. The test are also be performed on the `i5-3570K` CPU. This CPU has two physics cores that are viewed as four virtual cores by the system. Therefore we have decided to test the parallelization of our algorithm using `Cilk` upto 4 threads. The time of the one threaded algorithm must be compared with the *delta*`+sse` version of the previous table : it illustrate the additional cost induced by the use of `Cilk` technology.

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

On the following table we illustrate the impact of parallelization. The test were performed on a machine holding two `Intel`$^{\text{TM}}$ `Xeon`$^{\text{TM}}$ `X5650` CPU

running at 2.67 GHz. Each of those CPU have 6 physical cores that are able to run 12 threads. However, due to Hyper threading, when more than 12 thread are running, the computation engine are shared between two threads. Therefore, the speedup should by much less when adding more core. We can see that this is indeed happening under the horizontal bar in the following table.

Threads	30	35	40	45	50
1	0.33	2.50	26.1	307	3709
2	0.17	1.25	13.05	153.5	1865
3	0.11	0.71	8.57	102.6	1243
4	0.10	0.55	6.45	77.21	932.4
8	0.10	0.36	3.55	40.22	486.8
12	0.10	0.36	2.61	27.34	325.7
16	0.10	0.37	2.50	25.96	311.2
20	0.10	0.37	2.69	26.04	302.3
24	0.12	0.51	3.18	25.27	290.2

Acknowledgment. We would like to thank Shalom Eliahou for his presentation of the tree of numerical semigroups and for the interesting discussions on the subject. We also would like to thanks Nathan Cohen who suggested the two level of indirection stack optimization.

References

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

[2] N. Borie, *Generation modulo the action of a permutation group*, Proceeding of The 25th International Conference on Formal Power Series and Algebraic Combinatorics (FPSAC '13), DMTCS Proceedings **0**(01) (2013), 767–778

[3] M. Delgado, homepage, `http://cmup.fc.up.pt/cmup/mdelgado/numbers/`

[4] M. Delgado, P.A. Garcia-Sanchez, J. Maris, *NumericalSgps* – GAP package, ver.0.971, 2001 `http://cmup.fc.up.pt/cmup/mdelgado/numericalsgps/`

[5] GAP – Groups, Algorithms, and Programming, Version 4.6.3, `http://www.gap-system.org`

[6] P.A. Garsia-Sanchez, J.C. Rosales, *Numerical semigroups*, Springer, New York, 2009, Developments in Mathematics, vol. 20, x+181

[7] B.V. Iyer, R. Geva, P. Halpern, *Cilk^{TM}+ in GCC* ,GNU Tools Cauldron, 2012, `http://gcc.gnu.org/wiki/cauldron2012?action=AttachFile&do=get&target=Cilkplus_GCC.pdf`

[8] J.L. Ramírez Alfonsín, *The Diophantine Frobenius Problem*, Oxford University press, 2005, Oxford Lecture Series in Mathematics and its Applications, vol. 30, xvi+243

[9] J.C. Rosales, *Fundamental gaps of numerical semigroups generated by two elements*, Linear Algebre Appl. **405** (2005), 200–208

[10] N.J.A. Sloane, *The On-Line Encyclopedia of Integer Sequences*, `http://oeis.org`

[11] Software.intel.com, *Intel Instruction Set Architechture Extensions*, Intel® Developer Zone, 2013, `https://www.cilkplus.org/`

[12] Software.intel.com, *Intel® Cilk^{TM} Homepage*, `ttps://www.cilkplus.org/`

[13] Software.intel.com, *Intel® Cilk^{TM} Plus Reference Guide*,`https://software.intel.com/en-us/node/522579`

[14] Wikipedia, *Advanced Vector Extension*, `http://en.wikipedia.org/wiki/Advanced_Vector_Extensions`
[15] Wikipedia, *Duff's device*, `http://en.wikipedia.org/wiki/Duff's_device`
[16] Wikipedia, *SIMD*, `http://en.wikipedia.org/wiki/SIMD`
[17] Wikipedia, *Streaming SIMD Extensions*, `http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions`
[18] H. S. Wilf, *A circle-of-lights algorithm for the money-changing problem*, Amer. Math. Monthly **85** (1978), 562–565
[19] A. Zhai, *Fibonacci-like growth of numerical semigroups of a given genus*, Semigroup Forum**86**, no.3 (2013), 634–662