Bachelor of Science Thesis

Spring 2008

Efficient Computation of the Intersection of the Preimages of Subsets of Finite Commutative Groups under a Family of Homomorphisms

Radoslav R. Zlatev
Jacobs University Bremen
r.zlatev@jacobs-university.de

ABSTRACT

Let us have a family of surjective homomorphisms from $\mathbb{Z}^r$ to finite commutative groups and be given subsets of those groups. We want to compute the intersection of the preimages of these subsets and show that it is empty. One way to do so efficiently is to consider the factor groups modulo some integer $B > 1$ and show this for the induced homomorphisms. In this paper we present one such algorithm. It first finds efficiently B for which we might expect that the intersection in the factored (and thus in the original) case is empty. Then it actually constructs the factored intersection and checks whether it is empty or not.

1 Introduction

This paper is the final report for the course Guided Research in Mathematics. The latter is a two-semester course in the Jacobs University Bremen curriculum aiming to help students conduct original mathematical work under the supervision of a faculty member. Supervisor of my project was Prof. Michael Stoll.

Together with a C++ implementation of the algorithm described here, this document constitutes my Bachelor Thesis. (The implementation is available at `http://dl.dropbox.com/u/3721315/grm_Radoslav_Zlatev.tar.gz`.)

In the sequel of this section we formulate the problem and recall a few mathematical facts which we need later. In Section 2 we discuss the motivation for this work. In Section 3 we present the algorithm. We describe its main idea and give pseudo code listings of the more interesting procedures it uses. In Section 4 we refer to our C++ implementation and clarify some specific points, which need to be taken care of in the real computational settings. In addition, we discuss the bottleneck of the performance and optimize it to gain speed. Finally, in Section 5 we present our results and give a discussion on the performance.

1.1 Problem formulation

Consider the following situation. Let $\{\phi_i\}$ be a finite family of surjective group homomorphisms

$$\phi_i : \mathbb{Z}^r \longrightarrow G_i$$

where each G_i is a finite commutative group. Suppose we are given a subset $S_i \subset G_i$ for each i. Now we want to decide whether there exists some element of $\mathbb{Z}^r$ mapped into S_i for all i. Since for the application of this (see Section 2) we want to show that there is no such element, we want to decide whether

$$\bigcap_i \phi_i^{-1}(S_i) = \emptyset$$

holds in any given example.

Note that the problem is computational, so what we are interested in is constructing an efficient algorithm, which tells us whether the intersection is empty or else computes it explicitly.

1.2 Mathematical Background

We recall a few mathematical facts here, which we use in Section 2: Motivation and in Section 3: The Algorithm. We also note that the proofs of the facts are

either too easy or too difficult to include here, so are omitted. In case of interest, we provide references.

Theorem 1. *Let C_n denote the cyclic group of order n. For two relatively prime numbers p and q, we have that $C_{pq} \cong C_p \oplus C_q$.*

Theorem 2. *Every finite commutative group G is isomorphic to a direct product of cyclic groups, each of order p^k, where p is a prime and k is a positive integer.*

The proofs of these facts are easy.

Definition 1. *Let C be an algebraic curve defined over $\mathbb{Q}$. Denote the set of rational points of C by $C(\mathbb{Q})$.*

Theorem 3 (Faltings)**.** *Let C be a non-singular algebraic curve over $\mathbb{Q}$ of genus $g > 1$. Then the number of rational points on C is finite.*

Theorem 4 (Mordell-Weil)**.** *For an abelian variety A over $\mathbb{Q}$, the set of rational points $A(\mathbb{Q})$ is a finitely generated commutative group.*

The proofs of Theorem 3 and 4 are by far non-trivial. In fact, Faltings was awarded his Fields medal for the proof of Theorem 3. In case of interest, the reader may refer to [4] and [5], respectively.

Throughout the rest of the text, we use the above results without further reference.

2 Motivation

Let C be an algebraic curve defined over the rationals of genus $g > 1$, e.g. $y^2 = f(x)$, where $f(x) \in \mathbb{Z}[x]$ is square-free and $\deg f \geq 5$. We want to show that the set of rational points on C, $C(\mathbb{Q})$, is empty. Note that by Fact 3 (Faltings) we know that $C(\mathbb{Q})$ is finite. Also we note that if $\deg f$ is odd or $\deg f$ is even and the leading coefficient is square, then there is a rational point at infinity.

Associated to C is its Jacobian variety J—an abelian variety of dimension g. We can make the following two assumptions.

(1) In many cases we can find an embedding $\iota : C \hookrightarrow J$ defined over $\mathbb{Q}$. If there is no such embedding, then $C(\mathbb{Q})$ is empty.

By Theorem 4 the set of rational points $J(\mathbb{Q})$ on J is a finitely generated commutative group.

(2) Assume we can determine generators of $J(\mathbb{Q})$. In many cases this can be done [1],[2]. Programs performing the necessary computations have been implemented by Michael Stoll [3].

Now, for a prime p of "good reduction" for C, we have the following commutative diagram

$$\begin{array}{ccc} C(\mathbb{Q}) & \xhookrightarrow{\iota} & J(\mathbb{Q}) \\ \downarrow & & \downarrow \phi \\ C(\mathbb{F}_p) & \xhookrightarrow{\iota} & J(\mathbb{F}_p) \end{array}$$

Note that $J(\mathbb{F}_p)$ is a finite commutative group. Put $G := J(\mathbb{F}_p)$, $S := \iota(C(\mathbb{F}_p)) \subset G$. Then $\#G \approx p^g$, $\#S \approx p$. Since the above diagram is commutative, we must have $\iota(C(\mathbb{Q})) \subset \phi^{-1}(S)$. Now let P be a set of primes. Then we can combine the diagrams for the various $p \in P$ into

$$\begin{array}{ccc} C(\mathbb{Q}) & \xhookrightarrow{\iota} & J(\mathbb{Q}) \\ \downarrow & & \downarrow \phi \\ \prod_{p\in P} C(\mathbb{F}_p) & \xhookrightarrow{\iota} & \prod_{p\in P} J(\mathbb{F}_p) \end{array}$$

We hope to find $\phi(J(\mathbb{Q})) \cap \prod_{p\in P} \iota(C(\mathbb{F}_p)) = \emptyset$. Then, again by the commutativity of the diagram, we must have that $C(\mathbb{Q}) = \emptyset$.

Furthermore,

$$J(\mathbb{Q}) \cong \mathbb{Z}^r \oplus T$$

with a finite commutative group T. Then for primes $p_1, \ldots, p_n$, take $G_j = J(\mathbb{F}_{p_j})$. If we fix generators of $J(\mathbb{Q})$, we then have

$$\phi_j : \mathbb{Z}^r \longrightarrow J(\mathbb{Q}) \longrightarrow J(\mathbb{F}_{p_j}) = G_j$$

$$S_j = \iota(C(\mathbb{F}_{p_j})) \subset G_j$$

To make the computation more efficient, we can work with the following maps instead

$$\begin{array}{ccccc} C(\mathbb{Q}) & \xhookrightarrow{\iota} & J(\mathbb{Q}) & \longrightarrow & J(\mathbb{Q})/BJ(\mathbb{Q}) \\ \downarrow & & \downarrow \phi & & \downarrow \alpha_B \\ \prod_{p\in P} C(\mathbb{F}_p) & \xhookrightarrow{\iota} & \prod_{p\in P} J(\mathbb{F}_p) & \xrightarrow{\beta_B} & \prod_{p\in P} J(\mathbb{F}_p)/BJ(\mathbb{F}_p) \end{array}$$

If $\mathrm{im}(\alpha_B) \cap \mathrm{im}(\beta_B) = \emptyset$ ($\Leftrightarrow \alpha_B^{-1}(\mathrm{im}(\beta_B)) = \emptyset$), then $C(\mathbb{Q}) = \emptyset$.

For $B_0 = 1, B_1 = q_1, \ldots, B = B_k = q_1 q_2 \ldots q_k$, we compute inductively

$$\Sigma(B_j) := \alpha_{B_j}^{-1}(\mathrm{im}(\beta_{B_j})) \subset J(\mathbb{Q})/B_j J(\mathbb{Q}) \cong (\mathbb{Z}/B_j\mathbb{Z})^r$$

and hope to find that $\Sigma(B) = \emptyset$.

3 The Algorithm

We now describe the algorithm in detail. We first explain the main idea and then clarify some of its specific aspects. In the last subsection we give pseudo code listings of the more interesting procedures.

3.1 The main idea

One possible approach is to use brute force—find the preimage of S_i in $\mathbb{Z}^r$ for each i (as a finite union of cosets of $\ker \phi_i$) and then compute their intersection. Note that we can actually find the preimage explicitly: since G_i is finite and $\mathbb{Z}^r$ is a direct product of r cyclic groups, then there exists an integer B', such that for any $h \in \bigcap_i \phi^{-1}(S_i)$ and $z \in \mathbb{Z}^r$ also $h + B' \cdot z \in \bigcap_i \phi^{-1}(S_i)$ and if $(h_1, \ldots, h_r) \in \bigcap_i \phi^{-1}(S_i)$, then $h_j = h'_j + k_j B'$ with $h'_j < B'$ and $k_j \geq 0$ for all $1 \leq j \leq r$ and $(h'_1, \ldots, h'_r) \in \bigcap_i \phi^{-1}(S_i)$. This follows from the fact that the intersection of all the kernels of the homomorphisms is a finite-index subgroup of $\mathbb{Z}^r$, hence contains $B'\mathbb{Z}^r$ for some B' (which can be taken to be the least common multiple of the orders of the groups G_i). Thus, we can only consider the restrictions of the homomorphisms $\phi_i : (\mathbb{Z}/B'\mathbb{Z})^r \to G_i$ and since they are of finite domain, we can find the preimages. However, one should expect the computational cost for such a procedure to be too high, since B' will usually be very large.

In this paper we present a different approach; still, we use the observation just made. Let B be an integer, such that $B > 1$ and $B|B'$, B' as above. Now we can consider the family of induced homomorphisms

$$\phi_{i,B} : (\mathbb{Z}/B\mathbb{Z})^r \longrightarrow G_i/BG_i.$$

Let $S_{i,B}$ denote the image of the set S_i in the factored group G_i/BG_i. Then we note that if $\bigcap_i \phi_{i,B}^{-1}(S_{i,B})$ is empty, then so is $\bigcap_i \phi^{-1}(S_i)$. In the general case, however, the map sending the second set to the first as a map from $\mathbb{Z}^r$ to $(\mathbb{Z}/B\mathbb{Z})^r$ need not be surjective, so we might find the size of the factored intersection much larger, if we count the cosets. Also, one should expect the size of the first set to be much larger for smaller B and next to equal to the second for B close to B', counting as before. On the other hand, computation of the intersection is faster for smaller B's. As we are interested in the case when the intersection is empty, we can proceed in the following way. Let us denote

$$\Sigma(B) := \{g \in (\mathbb{Z}/B\mathbb{Z})^r : \phi_{i,B}(g) \in S_{i,B} \text{ for all } i\}$$

and its expected size by

$$s(B) := \mathbb{E}(\#\Sigma(B)) = B^r \cdot \prod_i \frac{\#S_{i,B}}{\#G_i/BG_i}.$$

Now we can fix a threshold $\varepsilon \ll 1$ and observe that it is very likely that $\Sigma(B)$ is empty if $s(B) < \varepsilon$.

Note that if such B exist, then $B|B'$ and we can use some iterative procedure to check all possible B's. Thus, we note that the procedure can be viewed as a tree search. Indeed, let $B' = p_1^{r_1} p_2^{r_2} \ldots p_l^{r_l}$, where $p_1, \ldots, p_l$ are distinct primes and $r_i > 0$ for all $1 \leq i \leq l$. Then we can construct a *directed* search tree having 1 as a root, all B's such that $B|B'$ as the other nodes, and p_1 through p_l as edges; an edge p then leaves a node B if $pB|B'$. The nodes thus denote the product of the path from the root to that node. We associate to a path, the number labeling the node it reaches, that is the product of its edges.

The problem now asks to find a path (a sequence of primes), whose associated number (product of the edges) will allow us to compute the factored intersection, for when we want to show that it is empty, relatively fast.

3.2 Cost of a path

We search the tree in a best–first manner. Let $B = q_1 q_2 \ldots q_m$ with q_i prime for $i = 1, \ldots, m$. We define the cost of the path $cost(q_1, q_2, \ldots, q_m)$ to be the time we expect is needed to compute $\Sigma(B)$. Of course, the expected time should depend on the expected size of $\Sigma(B)$. The computation can be done in an inductive manner.

We compute $\Sigma(q_1)$ directly. This amounts to check the images of the elements of $(\mathbb{Z}/q_1\mathbb{Z})^r$. If we have computed $\Sigma(B)$ (violating notation), for $\Sigma(qB)$ we have to check only the lifts of each element of $\Sigma(B)$ to any of the r components of $(\mathbb{Z}/qB\mathbb{Z})^r$. The time needed for doing this is roughly proportional to $\#\Sigma(B) \cdot q^r$, so we define

$$cost(q_1, q_2, \ldots, q_m) := \sum_{k=0}^{m-1} s(q_1 q_2 \ldots q_k) \cdot q_{k+1}^r$$

as an estimate to the total time required to compute $\Sigma(B)$. Note that this value depends on the order of the primes in the path and not only on their product B. It is fairly called the cost of the path, since we actually want to find a small number, divisor of B', for which we have that the expected size of the intersection is less than the threshold ε and can compute the intersection relatively fast. Thus, the cost of the path is the value by which we expand in best-first manner the search

tree. Note that the search could also be done over the s values of the paths. However, the total time for computing the intersection need not be proportional to its expected size.

3.3 Bound on path "length"

We already discussed the existence of an upper bound. Here we show explicitly how to compute such a bound B'. Note that for the algorithm, we need a value, which can be easily computed and not necessarily the theoretical bound from 3.1.

Rewrite G_i as a product of finite cyclic groups $(\mathbb{Z}/p^k\mathbb{Z})$ and think of each homomorphism as a map into the product of all G_i's.

$$\begin{aligned} G_j &\cong \prod_{\alpha} \mathbb{Z}/p_{j,\alpha}^{e_{j,\alpha}}\mathbb{Z} \\ \prod_j G_j &\cong \prod_{j,\alpha} \mathbb{Z}/p_{j,\alpha}^{e_{j,\alpha}}\mathbb{Z} \\ &\cong \prod_{k=1}^{l}\prod_{i=1}^{m_k} \mathbb{Z}/p_k^{e_{k,i}}\mathbb{Z} \end{aligned}$$

where $p_1, \ldots, p_l$ are the distinct primes in $\#\prod_j G_j$ and $e_{k,1} \geq e_{k,2} \geq \ldots \geq e_{k,m_k}$. Take the product of the r-th largest powers for each of the primes

$$B' = \prod_{k=1}^{l} p_k^{e_{k,r}},$$

where $e_{k,r} := 0$ if $r > m_k$. The choice of $e_{k,r}$ is motivated by the observation that this value of B' is the largest one such that we can have injective induced homomorphisms $\phi_i' : (\mathbb{Z}/B'\mathbb{Z})^r \to \prod_i G_i/B'G_i$. Thus if we use strict multiples of this value of B', we lose information.

3.4 Computation of the intersection

After we have found B as above, we must check if indeed the factored intersection is empty. Let q_1 be the first prime in the sequence. As explained earlier, we compute $\Sigma(q_1)$ directly in time $O(q_1^r)$. If we have computed $\Sigma(B)$ (violating notation), then $\Sigma(qB)$ can be computed from it in the following way. For each $g \in \Sigma(B)$ and each $h \in (\mathbb{Z}/q\mathbb{Z})^r$ check if $B \cdot h + g$ is mapped to $S_{i,qB}$ via each homomorphism $\phi_{i,qB}$. In that case, add $B \cdot h + g \in (\mathbb{Z}/qB\mathbb{Z})^r$ to $\Sigma(qB)$.

3.5 Pseudo code

Finally, we present the algorithm in pseudocode. It consists of five main procedures and three structure declarations. We also assume the usage of a stable sort algorithm, a priority queue and a set.

MAIN-LOOP, as the name suggests, is our main tree search procedure. It supports a priority queue Q, which contains all paths explored so far. At every iteration the path with smallest cost is popped from the queue, expanded by each of the allowed edges, and the resulting paths are put back in the queue, ordered by their costs. The loop terminates whenever a path of s value less than ε is to be inserted in the queue. This guarantees that, if found, such a path will be of smallest cost. The procedure returns "Failure" in case that no such path is found before the queue gets empty (i.e. all the paths with assigned numbers divisors of B' have already been checked).

Procedure 1 MAIN-LOOP

Input: void
Output: Sequence of primes as described in Sec 3.1

```
1:  B', P ← COMPUTE-BOUND-ON-LENGTH()
2:  firstPath.nodes ← [1]
3:  firstPath.cost ← 0
4:  firstPath.s ← 1
5:  Q ← [firstPath]
6:  B ← ∅
7:  bestPath ← firstPath
8:  while bestPath.s > ε do
9:    for each p ∈ P do
10:     declare newPath
11:     newPath.edges ← bestPath.edges :: [p]
12:     newPath.s ← COMPUTE-S(newPath)
13:     newPath.cost
14:           ← COMPUTE-COST(bestPath, p)
15:     B ← newPath.number
16:     if B|B' and B ∉ 𝓑 then
17:       Q.insert(newPath)
18:       𝓑 ← 𝓑 ∪ {B}
19:     end if
20:   end for
21:   if not Q.empty then
22:     bestPath ← Q.first
23:     Q.pop-first
24:   else
25:     return FAILURE
26:   end if
27: end while
28: return bestPath
```

The procedures COMPUTE-BOUND-ON-LENGTH, COMPUTE-S, and COMPUTE-COST compute the respective values. All three of them assume knowledge of the groups G_i. The last two are straight-forward computations using the formulas from Section 3.1

and 3.2. COMPUTE-BOUND-ON-LENGTH is a bit more interesting, so we present it here. It first loops over all the groups and inserts into the list P all the prime divisors of the order of the current group, if not yet present in P. The procedure also supports an additional list of lists P'. Positions of P' are labeled by the primes from P and each of its elements is a list containing the powers to which the respective prime occurs in the product of the cyclic groups which build $\prod_i G_i$ (see Section 3.3). Then the lists are sorted in decreasing order and the r-th largest powers are used for the computation of B'.

The sort algorithm can be any stable sort, e.g. `MERGE-SORT` ($O(n \log n)$) or `BUBBLE-SORT` ($O(n^2)$).

Procedure 2 COMPUTE-BOUND-ON-LENGTH

Input: void
Output: B', P

1: $P \leftarrow \emptyset$
2: **for each** i **do**
3: **for each** prime $p | G_i.\text{order}$ **do**
4: **if** $p \notin P$ **then**
5: $P \leftarrow P :: [p]$
6: **end if**
7: **end for**
8: **end for**
9: **for each** $p \in P$ **do**
10: **for each** cyclic group C_{p^e} **of each** G_i **do**
11: $P'[\text{`}p\text{'}] \leftarrow (P'[\text{`}p\text{'}]) :: [e]$
12: **end for**
13: **end for**
14: **for each** $p \in P$ **do**
15: STABLE-SORT-REV($P'[\text{`}p\text{'}]$)
16: **end for**
17: $B' \leftarrow 1$
18: **for each** $p \in P$ **do**
19: $B' \leftarrow B' \cdot \max(1, p^{P'[\text{`}p\text{'}][r]})$
20: **end for**
21: **return** B', P

The last procedure we present is CHECK-INTERSECTION. It computes the actual intersection of the preimages of the factored subsets and decides whether it is indeed empty. This is done in the following way. Let the solution path be $[q_1, \ldots, q_m]$. First compute $\Sigma(q_1)$ directly—for each of the elements of $(\mathbb{Z}/q_1\mathbb{Z})^r$ check if it maps to S_{i,q_1} via all ϕ_{i,q_1}. From there on, we can compute the rest iteratively each time computing $\Sigma(qB)$ considering all the lifts of the elements of $\Sigma(B)$ to $(\mathbb{Z}/qB\mathbb{Z})^r$.

We also define three main object structures: `group`, `path` and `searchTree`.

The `group` structure needs to store information of the cyclic groups which build G_i and the set S_i explicitly. This information can be stored as follows. Each group is a list of pairs $\langle p, e \rangle$, which represent the respective cyclic groups. The group order and the number of the cyclic groups are also useful to store to gain speed. The sets are represented as list of vectors, where every position of the vector corresponds to the respective cyclic group in the decomposition of G_i. Finally, it is also useful to store the size of the set. Then the following methods can be derived: `factor(`B`: int)` which sets the factor group G_i/BG_i, `factor-size` which computes the order of the factor group, and `S-factor-size` which counts the images of the elements of S_i in the factor group.

Procedure 3 COMPUTE-S

Input: $oldPath$: path
Output: $s(B)$

1: $B \leftarrow oldPath.\text{number}$
2: $prod \leftarrow 1$
3: **for each** G_i **do**
4: $G_i.\text{factor}(B)$
5: $prod \leftarrow prod \cdot \frac{G_i.\text{S-factor-size}}{G_i.\text{factor-size}}$
6: **end for**
7: **return** $B^r \cdot prod$

The `path` structure is even simpler. It needs knowledge of the nodes only (resp., the edges). The the following methods can be derived: `number` is the reached node (resp., the product of the edges), `valid` is 1 if `number`$|B'$ and 0 otherwise, `s` is the expected size of the intersection when `factor` for all groups G_i is set to `number`, and finally `cost` is the expected time to compute the intersection.

Procedure 4 COMPUTE-COST

Input: p: integer, $oldPath$: path
Output: $cost(B)$

1: **return** $oldPath.\text{cost} + (oldPath.\text{s}) \cdot p^r$

The `searchTree` structure has a single instance which represents the search tree. It supports a priority queue Q and a set $\mathcal{B}$: Q stores the expanded paths that are to be checked in ascending order with respect to their costs; since we are interested in considering every B just once, with factors ordered in the way in which we first encounter their product, we maintain the set $\mathcal{B}$ to keep all the B's already considered. Then an expanded path is put on the queue only if its associated number is not already in $\mathcal{B}$.

Procedure 5 CHECK-INTERSECTION

Input: $[q_1, q_2, \ldots, q_m]$: path

Output: $\#\left(\cap_i \phi_{i,B}^{-1}(S_{i,B})\right)$, where $B = q_1 q_2 \ldots q_m$

```
1: Σ(q_1) ← ∅
2: for each h ∈ (Z/q_1Z)^r do
3:    bool ← true
4:    for each i do
5:       if φ_{i,q_1}(h) ∉ S_{i,q_1} then
6:          bool ← false
7:       end if
8:    end for
9:    if bool = true then
10:      Σ(q_1) ← Σ(q_1) ∪ {h}
11:   end if
12: end for
13: B ← q_1
14: for q ← q_2 to q_m do
15:   Σ(qB) ← ∅
16:   for each h ∈ (Z/qZ)^r do
17:      for each g ∈ Σ(B) do
18:         bool ← true
19:         for each i do
20:            if φ_{i,qB}(B · h + g) ∉ S_{i,qB} then
21:               bool ← false
22:            end if
23:         end for
24:         if bool = true then
25:            Σ(q_1) ← Σ(q_1) ∪ {B · h + g}
26:         end if
27:      end for
28:   end for
29:   B ← B · q
30: end for
31: return Σ(B)
```

4 Implementation

In the previous section we described the main idea and some specific points of our algorithm. In the last subsection we presented its main procedures explicitly in pseudo code. The purpose of this section is to present an implementation of the algorithm in the C++ programming language. We begin with a relative comparison between several higher order programming languages and explain why we preferred C++ over the other candidates. We then present the declarations and methods on which our program builds. For this part of the paper, the reader needs to refer to the code examples: `declr.h`, `declr.cpp`, `funcs.h`, and `funcs.cpp`.

4.1 Programming languages and computational speed

Since we are interested in a clear and short implementation we can concentrate on some of the well-known free programming languages like C/C++, C#, Java, etc. Note that those may not be as convenient as some of the mathematical software products like Magma, SAGE, GAS in the representation and support of mathematical objects. However, they are approachable for a larger audience and since are on a more basic level, are faster with respect to computational speed. Among the object-oriented programming languages C++ is the fastest and the most well-known (most widely used). We want to exclude the imperative languages, since they do not support objects and implementing every method as independent function may become lengthy and messy. Since those were our main concerns for the implementation itself, we stick to C++.

4.2 Declarations

We begin with a description of the data structures used. The first thing we observe is the order of the numbers we will be dealing with. Values for B' easily exceed 30-digit numbers, so even the `unsigned long long` type cannot handle them properly. Additionally, we need to raise these numbers to powers and deal with floating point numbers (e.g., when computing s). On the other hand, we want to keep the code simple and clear. That is why, we use a class definition for big integers and adopt a common convention for the floats.

Practice shows that a number of the order of 10^{-3} is admissible value for ε. However, s can also get arbitrarily large according to B and keeping such a large float is not feasible. That is, why we fix a certain precision, e.g. 10^{-6}, and use the `BigInt` class definition [6] for arbitrarily large integers.

Since all arithmetic operations (including integer division with remainder) are defined for numbers of the class and their size is practically unlimited (depends only on the system capabilities!), we use it as the default datatype of the (expected) large numbers in the application: group orders, path-associated numbers, and *cost* values. We also use `BigInt` type for the s value but with different interpretation. We fix a decimal point precision 10^{-6}. Thus, $10^6_{\texttt{BigInt}} = 1$. Indeed, since we target paths of s value less than 10^{-3}, the fact that numbers smaller that 10^{-6} become 0 does not bother us.

Keeping the code clear, we define the following datatypes[1]:

[1] grm prefix stands for "Guided Research in Mathematics"

```
typedef vector<unsigned int> grmList;
typedef vector<unsigned int> grmElement;
typedef vector<vector<unsigned int> >
        grmDList;
typedef RossiBigInt grmLong;
typedef vector<RossiBigInt> grmListLongs;
```

We next discuss the `grmGroup` class. It defines the `group` structure and methods. Note that for any group we have to store the orders of the cyclic groups whose direct product it is isomorphic to. However, having seen how the computation of B' is carried out and what the main search loop looks like, we only store the primes and their powers (see Section 3.5) as two separate arrays of the same length. Those values are relatively small and can be stored as `unsigned int` or `unsigned long` numbers.

As already discussed, we store the s value and the cost as `BigInt` (note that for consistency, we renamed the class `BigInt` to `grmLong` in the files `declr.h`, `declr.cpp`, `funcs.h`, and `funcs.cpp`). We also store the factor as parameter, since we need it for short computations only and do not want to allocate space for every new factor group. For convenience, we additionally store the number of cyclic subgroups (`_numCycGrs`), the size of S_i (`_sizeOfS`), and r (`_r`).

We can also store the set S_i as a two-dimensional array, since we consider any element of G_i as an array itself: each entry comes from the respective cyclic group. We store the images of the r generators of $\mathbb{Z}^r$ in the standart order. Since every group is associated to precisely one homomorphism, this (+ a variable `_n` to hold the number of these homomorphisms) is enough to define the family of homomorphisms completely. Note that when we refer to arrays, we will sometimes mean `vector` objects from the `STL` library.

```
class grmGroup {
  private:
        int _r;
        int _numCycGrs;
        int _sizeOfS;

        grmLong _factor;
        grmLong _size;

        grmList _factorSequence;

        grmListLongs _orders;
        grmList _primes;
        grmList _powers;

        grmDList _generatorsImgs;
        grmDList _S;
        ...
```

Our next class declaration is the class `grmPath`. It is closely related to and used along with the `grmSearchTree` class. `grmPath` supports the paths as presented in Section 3.1 and 3.4.

For each path we need to keep track of its edges (which we wrongly call nodes in the source code), s value, and cost. We store these entries in the variables `_sequenceOfNodes`, `_s`, `_cost`, respectively. However, for reasons explained in Section 3.4, we keep track of the number of times which we can expand certain edges (that is, we cannot expand a path to a node that does not divide B'). We take care of this by having an array `_allowedNodes` with `unsigned int` elements. The counter at the i-th position determines the number of p_i edges by which the path can be expanded. The indexing corresponds to the indexing of the primes used during the main iterations when searching through the tree.

```
class grmPath {
  private:
        grmList _sequenceOfNodes;
        grmList _allowedNodes;
        grmLong _s;
        grmLong _cost;
        grmLong B;
        ...
```

The last definition used in the search for B is the class `grmSearchTree`. There will be one object of this class in our application but it combines all the ideas conveyed so far. It stores the following values: `_paths`, `_epsilon`, `_primes`, `_bounds`; and keeps a pointer `_G`. The pointer is essentially important, since the group and path classes are otherwise unrelated. Here `_paths` is a `set` (in the `STL` sense) of `grmPath` elements. It plays the role of the priority queue from Procedure 1. The value of `_epsilon` is 1000 initially, which refers to 0.001 by the convention we adopted earlier. `_primes` and `_bounds` are both arrays of type `unsigned int` or `unsigned long` and represent (clearly) the primes and their powers in the decomposition of B'. `_primes`, in this order, defines the order of the counters in `_allowedNodes`.

```
class grmSearchTree {
  private:
        set<grmPath, setOfPathsCompare> _paths;
        set<grmLong> _expandedBs;
        grmLong _epselon;

        grmList _primes;
        grmList _bounds;

        grmGroup *_G;
        int _n;

        grmList _solution;
        ...
```

The last declaration is of the class `grmFamilyOfMaps`. As in `grmSearchTree` it keeps a pointer to the groups (`_G`) and their number `_n`. Here we assume, we are given a number B to check. It is passed from the main loop as a list of primes. For optimization purposes, we allocate space for factored generators and sets of each of the groups G_i. We also allocate space needed for the computation of $\Sigma(B)$, declared as `SQ`.

```
class grmFamilyOfMaps {
        private:
                grmGroup *_G;
```

```
    int _n;
    unsigned int _r;

    grmList _path;
    grmLong B;

    grmDList* factoredGens;

    set<grmList>* factoredS;
    set<grmListLongs> SQ;
...
```

4.3 Methods

Now we move to the implementation of the methods.

A side note is that the insertion in the queue is still efficient ($O(n \log n)$), if we use the `STL set` container rather than the `queue` container ($O(\log n)$). Since it has some advantages compared to the `queue` and `stack` containers, from now on by `queue` we would mean (in the `STL` sense) `set`.

We first discuss the format of the file for the homomorphisms specification. The specification can be any plain text file. The file starts with the constants r and n, the number of homomorphisms. Then every group homomorphism is given as a separate code block. Notation is the following: on the first line we read the the number of the cyclic groups (`_numCycGrs`), described above; on the second line we have a pair of numbers, a prime and its power, for every cyclic group; numbers are separated by a space or a new line; the following r lines give the images of the generators: each image is composed of `_numCycGrs` integers separated by space. Finally, we read the cardinal number of the set S_i and then `_sizeOfS` lines representing the subset elements in same fashion as the images of the generators. A template example is given below.

```
<_r> <_n>

<_numCycGrs>
<_p[0]> <_e[0]> <_p[1]> <_e[1]> ...
<image of generator #1>
<image of generator #2>
...
<image of generator #_r>
<_sizeOfS>
<element of S #1>
<element of S #2>
...
<element of S #_sizeOfS>
```

The specification file is read and the groups with their subsets and orders are initialized via the `loader()` function, defined in `funcs.h`. We then compute B' but unlike in the pseudo code example, we rather extract all the distinct primes, put them into an array (thus fixing order on them), and then build every prime's powers list (and put it into an array, according to the specified order on the primes). Thus, we do not keep B' as a number, but rather keep it as list of integers, linked implicitly. Then for any path, instead of computing its associated number and checking B''s divisibility by that number, we just keep track on how many times the particular prime has been used. A path cannot be expanded by p_i if the i-th counter in `_allowedNodes` is 0. We check this by the method `bool canExpand(int)`. Note that to facilitate the calculations, the latter decreases the value of the respective counter when ***true*** is returned. That is why, when positive, it should always be used with `void expand(unsigned int node, grmGroup*, int)`. The last method does expand the path by `node` and computes the new s value and cost, so justifies the decrement by `canExpand()`.

Another important aspect is the computation of the s value. Since it involves division $\frac{\#S_{i,B}}{\#G_i/BG_i}$, one gets 0 if this operation has priority over the multiplication by *prod*. We recall that *prod* should be initialized to $1000000 = 10^6$ as a `BigInt` number, since this relates to 1 as `int` number. Then the computation becomes the following

```
Prod=(Prod*G[i].getFns())/G[i].getFr();
```

Furthermore, note that for the implementation of the set $\mathcal{B}$ in the main loop, we indeed use the `STL` container `set`. It is implemented as a Red-Black tree, so has $O(\log n)$ worst-case time complexity for search, insertion, and deletion. Also an insertion of already present element would return a pair of NULL and *false* and an insertion of not present element—a reference to the element and *true*. This is essential for the expansion of paths on every iteration, as otherwise we have too many paths to explore and a exhausting search is simply not possible.

The other intersting aspect was the computation of the sizes of the factored groups and sets. For the size of the groups we need to compute the greatest common divisors of the cyclic groups and the factor. However, we get into trouble if we do that from scratch every time. Note that we will even have to do that twice. That is why we maintain the factor as a list of primes. Then we can easily get the number of occurrences of a prime in the list and take the prime raised to the minimum between this number and the power of the cyclic group. For the S_i's, we again employ the `STL set` container: factor all the elements, put them in the set and then simply return the size of the set.

To conclude this subsection and clarify the previous lines, in the remainder we present the methods in each class definition. The declarations names are self-explanatory.

```
class grmGroup {
```

```
  ...
    grmGroup();
    ~grmGroup();

    void set(grmListLongs,grmList,grmList,int,int);
    void setHom(grmDList);
    void setFactor(grmLong);
    void setS(grmDList,int);

    void setFactorSequence(grmList);
    const grmList getFactorSequence();

    const grmLong getOrder(int);
    const int getNumCycGrs();
    const int getSizeOfS();

    grmList computeFactors();
    const grmLong getSizeOfFactoredS(grmList);
    const grmLong getSizeOfFactoredGr(grmList);
    const grmLong getFactor();
    const grmLong getSize();

    unsigned int getPrimes(int);
    grmList getPrimes();
    unsigned int getPowers(int);
    grmList getPowers();

    grmDList getGenImages();
    grmDList getS();

    void print();
    void printS();
};

class grmPath {
  ...
    grmPath();
    ~grmPath();
    grmPath(unsigned int);
    grmPath(grmList,grmList,grmLong,grmLong,
grmLong);
    const grmLong s();
    const grmLong cost();
    const grmLong getB();
    const bool canExpand(int);
    void expand(unsigned int,grmGroup*,int);
    void print();
    const bool operator <(grmPath);
    const grmList getNodes();
};

class grmSearchTree {
  ...
    grmSearchTree();
    ~grmSearchTree();
    grmSearchTree(grmGroup*,int);

    const bool expand();
    grmList getSolution();
    void print();
};

class grmFamilyOfMaps {
  ...
    grmFamilyOfMaps();
    ~grmFamilyOfMaps();
    grmFamilyOfMaps(grmGroup*,int,grmList);
    grmList map(int,grmListLongs);

    void computeFactoredSets();
    set<grmListLongs> computePreimage();
};
```

4.4 Running time

Although it is hard to analyze the program precisely with respect to its assymtotic behavior (because of the parameter dependent complexities of a number of the subprocedures), we can get a very positive picture by looking at the sample running times in Section 5: Results. In 28 out of 30 examples a solution was found and for the other 2 it took 4.5 minutes for each to make the first 90 iterations. We can not speak of average running time, as this is heavily dependent on the value of B' and the number of the groups. However, apart from two tough cases, which took a bit more than 6 minutes each, the majority of the test cases were computed in less than 2 minutes each. In particular, the 10 examples with non-empty intersections were computed for 3 minutes and 26 seconds in total. Since we expect a fewer number of tough cases on average (the 20 files were selected as such), we may conclude that the program is effective and efficient on a fair amount of the data.

4.5 User's Guide

The implementation of the algorithm is provided in the file `grm_Radoslav_Zlatev.tar.gz`. You can unzip and untar it at once by typing the command below (note that you may want to create a directory for the files first)

```
$ tar xvfz grm_Radoslav_Zlatev.tar.gz
```

The main part of the code is in the `./final` directory. The four files already discussed `declr.h`, `declr.cpp`, `funcs.h`, `funcs.cpp`, and the main-function file `main.cpp` are copied there. Also the `BigInt` class files and a `Makefile` for the whole program are included. In addition there are two directories `./final/tests` and `./final/outs`. The first directory contains the 30 example specification files from Section 5; the second contains the output of some of the tests.

To compile the program, simply type (you must go to the directory first)

```
$ make
```

To run it on a sample file use the following format

```
$ ./grmrz test/g_data_{01,...,20} [steps]
```

or

```
$ ./grmrz test/h_data_{01,...,10} [steps]
```

Note the difference between the numbers! The files `g_data_*` are non-trivial examples of families of homomorphisms with empty intersection! The files `h_data_*` are families with NON-empty intersection. [steps] is an optional argument that specifies for how many iterations the main loop should run. By default, [steps] is set to 30.

After a solution is found, the user is asked whether to continue with the actual computation of the intersection. You must press 'c' to continue or any other key to quit.

Note that the program is compiled with the `-pg` flag, so if you want to get precise running time measurements you can use

```
$ time ./grmrz [file] [steps]
```

If you want to check the program on another file, just type the URL of the file instead.

The second directory is `./parser`. There you can find a parser to convert from Magma (or Michael Stoll's) specification files to the format used here. To do that simply go there and type

```
$ ./raw magma_formated_file
$ ./fine parsed.txt [output_filename]
```

If you skip the output file, the parsed specification file will be saved as `parsed_final.txt` and will overwrite a previously existing one.

Finally, the directory `./data` contains some 40 more unparsed specification files, provided by Michael Stoll.

5 Results

We conclude this paper with the outputs of the 30 example specifications from `./final/test`. As we discussed in Section 4.5, we can get a positive feeling as none of the examples crashed with respect to computational speed.

```
file: g_data_03
```

```
B'=7119426563808370812000
15 primes, largest: 47
43 groups
Goal reached (in 34 steps)!
Path nodes: 2,11,5,13,17,5,7,3,2,5,
Intersection: Empty
real    1m44.035s
user    1m42.686s
sys     0m0.272s
```

```
file: g_data_04
```

```
B'=753353335445911200
15 primes, largest: 47
37 groups
Goal reached (in 26 steps)!
Path nodes: 2,2,11,3,2,23,5,29,13,7,
Intersection: Empty
real    0m39.583s
user    0m38.902s
sys     0m0.104s
```

```
file: g_data_05
```

```
B'=10789875691263094169944680 4800
15 primes, largest: 47
61 groups
Goal reached (in 35 steps)!
Path nodes: 3,3,2,2,7,19,5,11,31,13,
Intersection: Empty
real    2m52.670s
user    2m50.063s
sys     0m0.596s
```

```
file: g_data_06
```

```
B'=2872677515986278000
15 primes, largest: 47
40 groups
Goal reached (in 17 steps)!
Path nodes: 3,11,19,2,17,2,13,5,43,
Intersection: Empty
real    0m36.395s
user    0m35.718s
sys     0m0.104s
```

```
file: g_data_07
```

```
B'=1326726079655583454848000
15 primes, largest: 47
49 groups
Goal not reached in 90 steps.
Maximal s = 2233
real    4m31.570s
user    4m28.333s
sys     0m0.816s
```

```
file: g_data_08
```

```
B'=824283857239350203968272 00
15 primes, largest: 47
61 groups
Goal reached (in 43 steps)!
Path nodes: 3,2,7,11,5,31,23,5,3,29,
Intersection: Empty
real    6m12.677s
user    6m10.195s
sys     0m1.068s
```

```
file: g_data_09
```

```
B'=1854778135791229028780 4000
15 primes, largest: 47
54 groups
Goal reached (in 42 steps)!
Path nodes: 2,2,3,7,11,5,3,37,7,3,29,
Intersection: Empty
real    6m20.214s
user    6m15.183s
sys     0m1.180s
```

```
file: g_data_10
```

```
B'=1073818920721237858767600
15 primes, largest: 47
```

```
54 groups
Goal reached (in 21 steps)!
Path nodes: 2,2,2,3,31,5,29,7,13,23,3,
Intersection: Empty
real    1m12.779s
user    1m11.388s
sys     0m0.248s
```

file: g_data_11

```
B'=206670936161775567081600
15 primes, largest: 47
55 groups
Goal reached (in 24 steps)!
Path nodes: 2,3,2,5,3,7,3,19,2,11,13,23,
Intersection: Empty
real    1m14.461s
user    1m13.225s
sys     0m0.256s
```

file: g_data_12

```
B'=1225203641213533581600
15 primes, largest: 47
50 groups
Goal reached (in 38 steps)!
Path nodes: 3,29,7,5,2,3,11,17,2,13,2,47,
Intersection: Empty
real    1m52.212s
user    1m50.015s
sys     0m0.320s
```

file: g_data_13

```
B'=20135731761887939000163360 0
15 primes, largest: 47
56 groups
Goal reached (in 37 steps)!
Path nodes: 3,3,13,17,2,7,5,11,19,
Intersection: Empty
real    2m32.914s
user    2m29.073s
sys     0m0.588s
```

file: g_data_14

```
B'=2515251526013710299816 00
15 primes, largest: 47
55 groups
Goal reached (in 30 steps)!
Path nodes: 2,3,7,5,13,41,11,43,17,
Intersection: Empty
real    2m11.645s
user    2m9.432s
sys     0m0.452s
```

file: g_data_15

```
B'=1274946909046803045012960
15 primes, largest: 47
44 groups
Goal reached (in 14 steps)!
Path nodes: 3,3,2,2,5,2,2,17,7,11,
Intersection: Empty
real    0m25.977s
user    0m25.506s
sys     0m0.112s
```

file: g_data_16

```
B'=252350766774316874664000
15 primes, largest: 47
57 groups
Goal reached (in 34 steps)!
Path nodes: 11,3,3,2,2,5,13,17,7,19,37,
Intersection: Empty
real    3m22.646s
user    3m19.900s
sys     0m0.560s
```

file: g_data_17

```
B'=92723411567040221455488 00
15 primes, largest: 47
55 groups
Goal not reached in 90 steps.

real    4m43.233s
user    4m39.005s
sys     0m0.924s
```

file: g_data_18

```
B'=9297133512737990119200
15 primes, largest: 47
49 groups
Goal reached (in 39 steps)!
Path nodes: 2,2,3,7,23,11,3,5,41,2,
Intersection: Empty
real    1m56.887s
user    1m54.887s
sys     0m0.328s
```

file: g_data_19

```
B'=11363163182235321256800
15 primes, largest: 47
48 groups
Goal reached (in 23 steps)!
Path nodes: 2,19,37,7,13,5,43,2,2,
Intersection: Empty
real    0m48.005s
user    0m47.011s
sys     0m0.180s
```

file: g_data_20

```
B'=19942351384822988805684 0000
15 primes, largest: 47
59 groups
Goal reached (in 26 steps)!
Path nodes: 3,3,5,2,2,5,11,7,13,17,
Intersection: Empty
real    3m33.328s
user    3m28.857s
sys     0m0.576s
```

file: h_data_01

```
B'=66474504616076629352280 00
15 primes, largest: 47
122 groups
Goal reached (in 1 steps)!
Path nodes: 2,
```

```
Intersection: NOT empty
real    0m5.143s
user    0m4.564s
sys     0m0.028s
```

```
file: h_data_02
```

```
B'=13055549188100156337600
14 primes, largest = 41
48 groups
Goal reached (in 8 steps)!
Path nodes: 2,2,2,7,5,11,3,
Intersection: NOT empty
real    0m20.393s
user    0m19.961s
sys     0m0.080s
```

```
file: h_data_03
```

```
B'=599815065337934846400
14 primes, largest = 47
38 groups
Goal reached (in 29 steps)!
Path nodes: 3,3,2,2,2,7,2,5,19,29,37,23,
Intersection: NOT empty
real    0m51.346s
user    0m50.451s
sys     0m0.120s
```

```
file: h_data_04
```

```
B'=525126924981349891200
15 primes, largest = 47
41 groups
Goal reached (in 12 steps)!
Path nodes: 2,2,2,3,5,7,13,11,
Intersection: NOT empty
real    0m19.844s
user    0m19.441s
sys     0m0.080s
```

```
file: h_data_05
```

```
B'=142813110794746392000
14 primes, largest = 47
39 groups
Goal reached (in 23 steps)!
Path nodes: 2,2,3,17,5,5,13,19,7,29,2,23,2,
Intersection: NOT empty
real    0m37.044s
user    0m36.158s
sys     0m0.128s
```

```
file: h_data_06
```

```
B'=312927908477522594256000
15 primes, largest = 47
51 groups
Goal reached (in 5 steps)!
Path nodes: 2,5,13,2,2,
Intersection: NOT empty
real    0m6.375s
user    0m6.248s
sys     0m0.024s
```

```
file: h_data_07
```

```
B'=230589673410096000
15 primes, largest = 47
45 groups
Goal reached (in 5 steps)!
Path nodes: 2,5,2,3,41,
Intersection: NOT empty
real    0m9.156s
user    0m8.881s
sys     0m0.020s
```

```
file: h_data_08
```

```
B'=19060789854072151785600
15 primes, largest = 47
41 groups
Goal reached (in 10 steps)!
Path nodes: 2,5,3,7,2,11,47,
Intersection: NOT empty
real    0m18.498s
user    0m18.261s
sys     0m0.040s
```

```
file: h_data_09
```

```
B'=74377068101903920953600
15 primes, largest = 47
46 groups
Goal reached (in 11 steps)!
Path nodes: 2,3,3,3,2,5,41,7,
Intersection: NOT empty
real    0m20.082s
user    0m19.633s
sys     0m0.084s
```

```
file: h_data_10
```

```
B'=17296992581838121152000
15 primes, largest = 47
51 groups
Goal reached (in 10 steps)!
Path nodes: 2,3,3,3,5,11,13,
Intersection: NOT empty
real    0m21.025s
user    0m20.809s
sys     0m0.084s
```

We leave the discussion whether or not this meets the requirements from the project for the faculty members. However, we are sorry we could not perform some detailed analysis on the behaviour of the program when the cost function was taken to be some combination of s and *cost*. This might be done in recent future by the author.

References

[1] Bruin N. and Stoll M., *Deciding existence of rational points on curves: an experiment*, `arXiv:math.NT/0604524`, 2006

[2] Bruin N. and Stoll M., *The Mordell-Weil sieve: Proving non-existence of rational points on curves*, in preparation

[3] Michael Stoll, *Implementing 2-descent for Jacobians of hyperelliptic curves*, Acta Arith. 98, pp. 245–277 (2001)

[4] Faltings, *Endlichkeitssä tze für abelsche Varietäten über Zahlkörpern. (German)*, Invent. Math. 73 (1983), no. 3, pp. 349–366

[5] André Weil, *L'arithmétique sur les courbes algébriques. (French)*, Acta Math. 52 (1929), no. 1, pp. 281–315.

[6] Vinokur A., Rossi W., BigInt C++ class, 2007
`https://sourceforge.net/projects/cpp-bigint/`

[7] Hungerford T. W., *Algebra (Graduate Texts in Mathematics)*, Springer 1974