Київський Національний Університет ім. Т.Г. Шевченка
Механіко-математичний факультет
Кафедра алгебри і комп'ютерної математики

**Курсова робота**
**на тему:**

# Складність проблеми слів в Ханойських групах

студента 3-го курсу
механіко-математичного факультету
**Зашкольного Давида Олександровича**

Керівник :
Доцент кафедри, доктор фізико-математичних наук
**Бондаренко Євген Володимирович**

КИЇВ — 2020

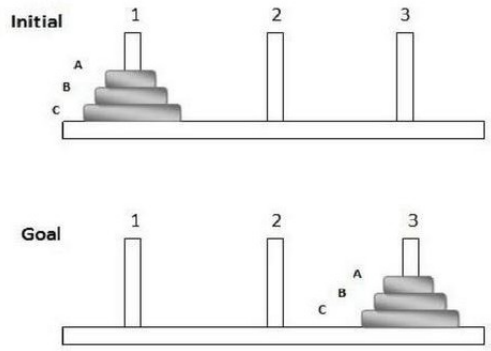
Contents

## 1. Introduction

This article presents investigation of algorithm for solving word problem in the Hanoi Tower Group. Here we show that such algorithm exists and its asymptotic growth depends on one quality of elements called *size*. We also denote the explicit form of elements with the biggest possible size and exact formula to calculate it (spoiler: it will be $O(n \log n)$ where $n$ means a word's length). For making this to be possible we introduce a tree-like representation of elements, investigate its structure and some interesting properties, including distribution of n-th level segments. With such tool we also try to explain an average size of elements which is non-trivial question.

## 2. Hanoi Tower Game and automaton groups

Fix integers $k, n \in \mathbb{N}, k \geq 3$. The Hanoi Tower Game is played on k pegs, labeled by $0, 1, ...k-1$, with n disks labeled by $1, 2, ...n$. All the $n$ disks have different size and their labels mean the relative size of disks from the smallest to the biggest. Any placement of these $n$ disks on $k$ pegs which satisfies condition that there is no a bigger disk upon the smaller one is a correct *configuration*. In a single step a gamer can move the top disk from one peg to another iff the result placement is a valid configuration. Thus, for any two pegs with non-zero number of disks in total there is only one possible move which involves these two pegs (the smaller top disk will be moved). At the beginning all the disks are placed on peg 0 and the goal of the game is to move all of them to peg 1 in the smallest possible numbers of steps (check [5] for more information).




As in the [2] we consider the free monoid $X^*$ of words over the alphabet $X = \{0, ...k-1\}$. The structure of an $X$ can be represented as a $k$-regular rooted tree $T$ (an empty word is the root, the $n$-th level consists of $n$-length words and for each $w \in X^*$ a corresponding vertex in the tree has children $wx$ for $x = 0, ...k-1$). Define recursive functions $a_{(ij)} : X^* \to X^*$ in such a way:

$$a_{(ij)}(iw) = jw, \quad a_{(ij)}(jw) = iw, \quad a_{(ij)}(xw) = xa_{(ij)}(w), \quad for\, x \notin \{i, j\}$$

We can think about these functions either as moves in the Hanoi Tower Game (the word $(w = x_1x_2...x_n) \in X^*$ represents a valid configuration of $n$ disks on $k$ pegs) or as automorphisms of the $T$. Since any automorphism $g$ of $T$ can be (uniquely) decomposed as $\pi_g(g_0, g_1, ...g_{k-1})$ (where $\pi_g \in S_k$ is called *root permutation* of $g$ and $g_x$, $x = 0, ...k-1$, are the tree automorphisms called the (first level) *sections* of $g$), defined functions also have such representation:

$$a_{(ij)} = (ij)(a_0, a_1, ...a_{k-1}),$$

where $a_s$ is the identity function (automorphism) if $s \in \{i, j\}$ and $a_s = a_{(ij)}$ otherwise.

**Definition 1.** *Hanoi Towers group on k pegs, $k \geq 3$, is the group* $H^{(k)} = \left\langle \left\{ a_{(ij)} | 0 \leq i < j \leq k-1 \right\} \right\rangle$.

Note: $H^{(k)}$ is an example of *automaton groups* (see [3]).

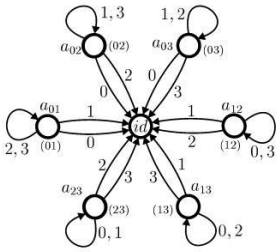



FIGURE 1. The automaton generating $H^{(4)}$.

## 3. Word problem and Trees

Now let's take a look at the $H^{(3)}$ group. Hereafter define generators:

$$a := a_{(12)} = (12)(e, e, a_{(12)})$$
$$b := a_{(13)} = (13)(e, a_{(13)}, e)$$
$$c := a_{(23)} = (23)(a_{(23)}, e, e)$$

Let's also introduce a new symbol $\delta$ which means a recursive call (its $\delta$ because this letter similar to $\circlearrowleft$). Thus, we have such representation of the $H^{(3)}$ atomic elements:

$$e = ()(\delta, \delta, \delta), \quad a = (12)(e, e, \delta), \quad b = (13)(e, \delta, e), \quad c = (23)(\delta, e, e)$$

With these denotations it's easy to write simple multiplication rules for any $g \in H^{(3)}$, $g = \pi(g_1, g_2, g_3)$ :
$ga = \pi * (12)(g_2, g_1, g_3 a), \quad gb = \pi * (13)(g_3, g_2 b, g_1), \quad gc = \pi * (23)(g_1 c, g_3, g_2)$ (henceforth multiplication in $S_3$ defined in the such way: $(\pi * \sigma)(x) = \sigma(\pi(x))$).

Note that $a^2 = b^2 = c^2 = e$ (due to definition of the automorphism's action or it's also implies from the multiplication rules). It allows us to consider at the same time a free group $G$ over an alphabet $\Sigma = \{a, b, c\}$ with relations $aa = bb = cc = \lambda$ where $\lambda$ is an empty word. *The word problem* is to describe an entire set of elements $R$ such that quotient $G / \langle R \rangle$ is isomorphic to $H^{(3)}$ (a.k.a presentation of the group). This problem also can be reinterpret as determining whether element $g \in G$ is trivial or not.

It should be pointed out that in [4] a full presentation for $H^{(3)}$ is obtained. It is:

$$H^{(3)} = \left\langle a, b, c, \mid a^2, b^2, c^2, \tau^n(w_1), \tau^n(w_2), \tau^n(w_3), \tau^n(w_4) \, \forall n \geq 0 \right\rangle$$

where $\tau$ is an endomorphism of $H^{(3)}$ defined by the substitution

$$a \mapsto a, \quad b \mapsto b^c, \quad c \mapsto c^b$$

and where

$$w_1 = [b, a][b, c][c, a][a, c]^b[a, b]^c[c, b]$$
$$w_2 = [b, c]^a[c, b][b, a][c, a][a, b][a, c]^b$$
$$w_3 = [c, b][a, b][b, c]^a[c, b]^2[b, a][b, c]^a[b, c]^a$$
$$w_4 = [b, c]^a[a, b]^c[b, a]^2[a, c][a, b]^c[c, a][c, b]$$

But in this research let's forget about other relations. Since we have the homomorphism between $G$ and $H^{(3)}$ all the multiplication rules for the described forms of elements from $H^{(3)}$ are also fair in the $G$.

**Statement 1.** *Any element of the $G$ can be represented as a finite rooted 3-regular tree with labels.*

*Proof.* Let there is any $w_g \in G$ and relative to it $g \in H^{(3)}$. Since $g = \pi(g_1, g_2, g_3)$ $w_g$ also satisfies corresponding representation $w_g = (w_1, w_2, w_3)$ where $w_i \in G$, $i = 1..3$. Due to structure of the atomic elements and multiplication rules $|w_i| \leq |w_g|$ and $|w_i| = |w_g| \Leftrightarrow w_i = w_g$, $w_j = e$, $j = 1..3, j \neq i$. Moreover, $|w_1| + |w_2| + |w_3| = |w_g|$. Thus, we can augment this representation to the next levels until we get to recursive calls which we substitute with $\delta$. Hence, we have a finite graph and it will have no cycles if we label each vertex as $w_{i_1 i_2 \ldots i_k}$ recursively:

$$w_{i_1 i_2 \ldots i_k} = (w_{i_1 i_2 \ldots i_k 1},\ w_{i_1 i_2 \ldots i_k 2},\ w_{i_1 i_2 \ldots i_k 3})$$

□

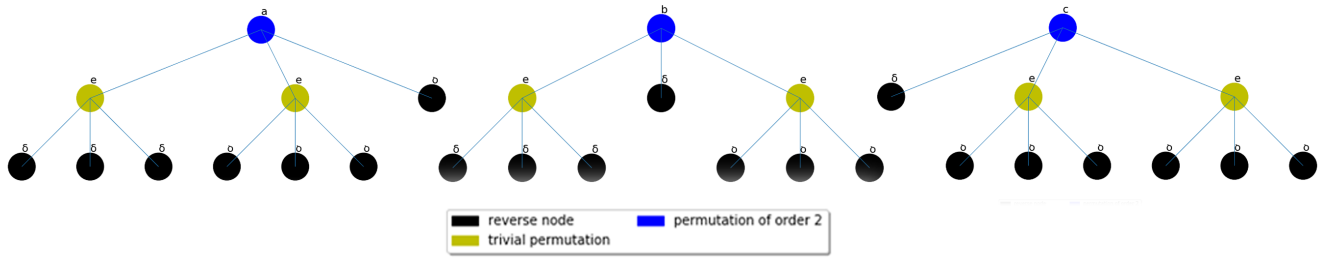



FIGURE 2. Corresponding trees for atomic elements.

Hereupon for $w \in G$ we define its tree as $T_w$.

Now we can define functions $v(w)$ which means amount of vertices in the $T_w$ and $h(w)$ which means height of the $T_w$ (in both cases excluding $\delta$ vertices).

**Definition 2.** *Function* ***size*** $: G \to \mathbf{N} \cup \{0\}$ *is defined recursively:*

$$(1) \qquad size(w = \pi(w_1, w_2, w_3)) = \begin{cases} 0, & w = e \\ 0, & w = \delta \\ |w| + size(w_1) + size(w_2) + size(w_3), & otherwise \end{cases}$$

Thus, $size(a) = size(b) = size(c) = 1$ by definition (Fig 2).

Similarly, $size(aa) = size(bb) = size(cc) = 2$, $\quad size(ab) = size(ac) = ... = size(bc) = 4$

## 4. Algorithm description

Instead of describing explicit form of the trivial elements we consider an algorithm for checking whether given element $w \in G$ is trivial or not in guaranteed time $O(n \log n)$ where $n = |w|$. Also we will try to investigate an average case which is the main goal of this article.

Fix $w = (w_1, w_2, w_3) \in G$ and a homomorphism $\varphi : G \to H^{(3)}$ such that $\varphi(w_a) = a$, $\varphi(w_b) = b$, $\varphi(w_c) = c$, $\varphi(\lambda) = e$.

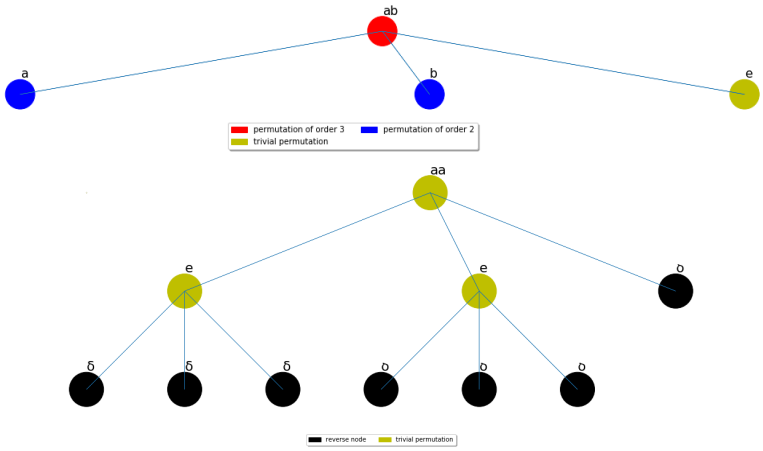



FIGURE 3. The words from $\{a, b, c\}^2$ can be divided into 2 classes.

Recursive conditions of triviality for $\varphi(w)$ are pretty simple:

- The root permutation of $\varphi(w)$ should be an identity one
- $\forall i = 1...3 \quad \varphi(w_i)$ is trivial or $w_i = \delta$

The main problem since we have a tree representation $T_w$ is to calculate the root permutation for each vertex in the tree.

Therefore, now we can write a full algorithm. Here is a Python-like pseudocode (well, actually it's real Python code, interface of used classes you can find in the appendix of this article, full implementation you can find at github.com/davendiy/automata-groups)

```
1
2  def is_trivial(w: AutomataGroupElement):
3
4      if w.permutation != TRIVIAL_PERMUTATION:
5          return False
6
7      res = True
8
9      # traverse all the children to make a recursive call for
10     # each w_i that isn't \lambda
11     for w_i in w.children:
12
13         if w_i.reverse:
14             continue
15         if not is_trivial(w_i):
16             res = False
17             break
18
19     return res
```

## 5. Asymptotic growth

Let $t(w)$ determines amount of elementary operations that requires aforementioned function $is\_trivial(w)$.

**Statement 2.** $t(w) = O(size(w))$.

*Proof.* For each vertex $v_s \in T_w$ that represents $s \in G$ we can compute the root permutation of $\varphi(s)$ and its first-level sections doing $O(|s|)$ operations. Thus,

$$t(w) = \sum_{v_s \in T_w} O(|s|) = O(size(w)), \text{ by definition (1).}$$

□

Therefore we can describe asymptotic growth of the function $size(w)$ and its dependence of $|w|$ will also give us a nice above limitation for asymptotic growth of $is\_trivial(w)$.

## 6. Size. The worst case

**Definition 3.** ***abc-subset*** $X \subset G$ *is such subset:*

$$X = \left\{(a^\pi b^\pi c^\pi)^k,\ (a^\pi b^\pi c^\pi)^k a^\pi,\ (a^\pi b^\pi c^\pi)^k a^\pi b^\pi \middle|\ k \in \mathbf{N} \cup \{0\},\ \pi \in S_3\right\} \tag{2}$$

*where* $a^\pi$ *means* $S_3$ *group action the alphabet* $\{a, b, c\}$*;*
$(w_1 w_2 w_3)^k$ *means repeating* $k$ *times of the word in the parenthesis.*

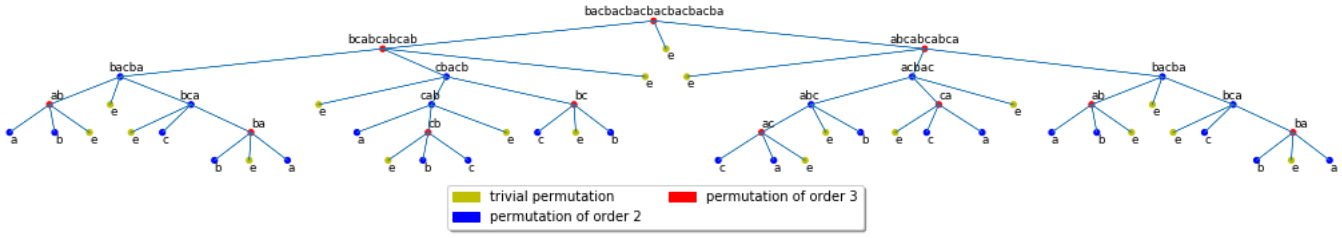
**Lemma 1.** $\forall w \in X$ $w$ *will have one of the next possible structures:*

$$\pi(w_1, w_2, e), \quad \pi(w_1, e, w_2), \quad \pi(e, w_1, w_2),$$

*where* $w_1, w_2 \in X, \quad |w_1|, |w_2| \in \left\{\left\lfloor \frac{n}{2} \right\rfloor, \left\lceil \frac{n}{2} \right\rceil\right\}, \quad |w_1| + |w_2| = n, \quad n = |w|$

*Proof.* Follows from definition of multiplication. □

Example:




Therefore, here we have recursive formula for calculating function $size$ (1) for elements from $X$:

$$size(w \in X) =: a(n) = a\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + a\left(\left\lceil \frac{n}{2} \right\rceil\right) + n, \quad a(1) = 1, \quad a(0) = 0, \quad n = |w|$$

**Theorem 2.** *Exact form of function* $a(n)$*:*

$$a(n) = \begin{cases} n\left(\lfloor \log_2 n + 1 \rfloor\right) + 2n - 2^{\lfloor \log_2 n \rfloor + 1}, & n > 0 \\ a(0) = 0 \end{cases} \tag{3}$$

*Proof.*

$$a(n+1) = n+1+a\left(\left\lfloor\frac{n+1}{2}\right\rfloor\right)+a\left(\left\lceil\frac{n+1}{2}\right\rceil\right) = n+1+a\left(\left\lfloor\frac{n}{2}\right\rfloor+1\right)+a\left(\left\lceil\frac{n}{2}\right\rceil\right)$$

$$\text{Let } b(n) := a(n+1)-a(n) = 1+a\left(\left\lfloor\frac{n}{2}\right\rfloor+1\right)-a\left(\left\lfloor\frac{n}{2}\right\rfloor\right) \text{ - auxiliary recursion.}$$

$$b(n) = b\left(\left\lfloor\frac{n}{2}\right\rfloor\right)+1, \quad b(2)=4 \quad \Rightarrow \quad b(n) = \lfloor\log_2 n\rfloor+3 \tag{4}$$

$$a(n+1) = b(n)+b(n-1)+...+b(1) = \sum_{1\le k\le n}\left(\lfloor\log_2 k+3\rfloor\right) \quad \Rightarrow \quad a(n) = 2(n-1)+\sum_{1\le k\le n}\left(\lfloor\log_2 k+1\rfloor\right) =$$

$$= \left[\begin{matrix}\text{amount of bits in all the}\\ \text{numbers from 1 to } N-1\end{matrix}\right] = 2(n-1)+\sum_{i=1}^{\lfloor\log_2 n\rfloor}(n-2^i) = n\left(\lfloor\log_2 n+1\rfloor\right)+2n-2^{\lfloor\log_2 n\rfloor+1}$$

□

**Theorem 3.** *Elements from abc-subset $X$ have maximum size, or in other words*

$$a(n) = \max(size(w)\,|\,w\in\{a,b,c\}^n)$$

*Proof.* Induction on $n$.

(1) Base: check examples of size calculation.

(2) Induction step:
Let $\forall k < n \quad a(k) = \max(size(w)\,|\,w \in \{a,b,c\}^k)$. Hereafter we consider any $w\in\{a,b,c\}^n$,
$w = \pi(w_1,w_2,w_3)$, $w_1,w_2,w_3 \in G\cup\{e\}$. Now we need to show that $size(w)\le a(n)$

Let $x_1 := |w_1|$, $x_2 := |w_2|$, $x_3 := |w_3|$. If $w$ has maximum size then, due to induction hypothesis

$$size(w) \le a(x_1)+a(x_2)+a(x_3)+n \tag{5}$$

(we don't know whether any triplet $w_1,w_2,w_3$ from $X$ could appear in $w$, so we use $\le$ symbol). Let's investigate how big this functional could be.

Since $b(x) = a(x+1)-a(x)$ is a monotonically increasing function (4) value of functional (5) grows as long as difference between $w_i$. Thus, we are allowed to assume that $\exists i\, w_i = e$, because we know that otherwise

$$size(w) \le a(x_1)+a(x_2)+a(x_3)+n \le a\left(\left\lfloor\frac{n}{2}\right\rfloor\right)+a\left(\left\lceil\frac{n}{2}\right\rceil\right)+n$$

It's easy to check that there are only 2 possible types of $w$ with one or more trivial element below: familiar to us $w\in X$ or such $w = z_1z_2z_3...z_n$ where there are at least one $i$ such that $z_i = z_{i+1}$, $\Rightarrow$ $w$ could be reduced $\Rightarrow$ $size(w) < a(n)$ due to definition.

□

As a result now we can say that the words problem of $H^{(3)}$ can be guaranteed solved in $O(n\log n)$ time. Although in real life it works way faster.

## 7. Schreier graph

There is one thing that makes it harder to investigate the average case. Tree-like structure is pretty bad because it doesn't give a lot of information about distribution of segments on the $n$-th level. Since $H^{(3)}$ is an automaton group there is a finite automaton $A = (S, X, \tau, \rho)$ that generates all the elements. Due to [1] consider a Shreirer graph $\Gamma_n = \Gamma_n(G, P_n, S)$. This graph is connected, have $3^n$, $n = 0, 1, ...$ vertices and is indeed the graph of action of $G$ on level $n$ in the tree $T_w$ for any $w \in G$.

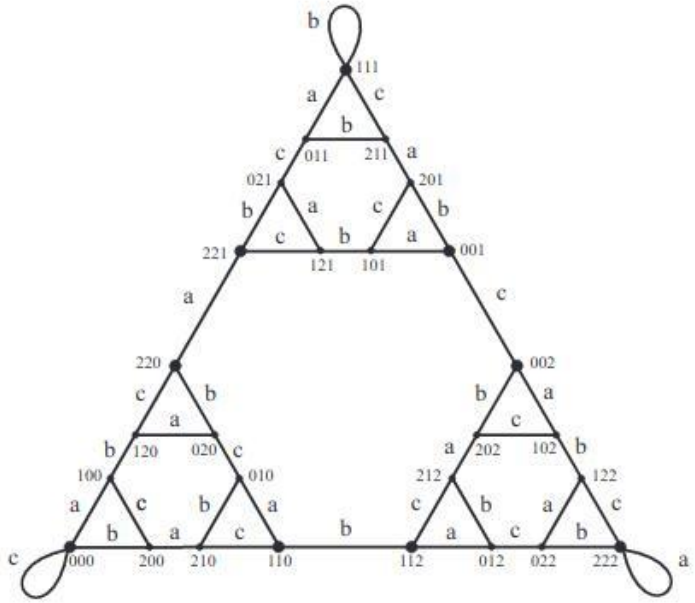



Figure 4. The Schreier graph of $H^{(3)}$ at level 3.

Fix the $n \in \mathbb{N}$ and relative $\Gamma_n$. Consider a finite Mealy automaton $A_n = (S_n, S_0, \Sigma, \Lambda, T, G)$, where

- $S_n = \Sigma^n$ - a finite set of states.
- $S_0 = w_n \in S_n$ - a start state
- $\Sigma = \{a, b, c\}$ - an input alphabet
- $\Lambda = \Sigma$ - an output alphabet
- $T_n : S_n \times \Sigma \to S_n$ - a transition function
- $G_n : S_n \times \Sigma \to \Lambda$ - an output function.

Define the transition $T_n$ function due to graph $\Gamma_n$ understanding vertices' numbers as words from $\Sigma^n$. Also define the output function $G_n$ in such a way:

$$G_n((aaa, a)) = a, \quad G_n((bbb, b)) = b, \quad G_n((ccc, c)) = c$$

and returns empty word otherwise.

**Statement 3.** *Let $w, v \in H^{(3)}$, $A(v)$ stops in state $S_r$ and returns $v_x$. Then,*

$$(wv)|_{S_r} = w|_{S_0} v_x.$$

*In other words, automaton $A$ simulates changes of $w_{S_0}$ after multiplying $w$ on $v$.*

*Proof.* Due to properties of multiplication in $H^{(3)}$ multiplying $w$ by $a$ means also multiplying $w|_{a^n}$. Similarly $b$ implies on $w|_{b^n}$ and $c$ on $w|_{b^n}$. Otherwise $w|_x$ does only change its own position (i.e. $x$). Such properties is the conditions the automaton is constructed on. □

## 8. Average case. Hypotheses

Now let's take a look at the aforementioned average size of trees. The main problem is we don't know which level is the last because we don't know properties of the tree's height.

**Statement 4.** *The $k$-th level isn't the last as long as it has at least one element $w_k$ which consists of $\geq 2$ different symbols.*

As a result for $k$-th level we can investigate how many elements $w \in G$, $|w| = n$ have height $\geq k$ considering multiplication $e$ by $w$ as a random walk on the $\Gamma_n$. The path on the $\Gamma_n$ is acceptable for us iff it contains at least two different loops. If number of such elements is big enough we can make an estimation for asymptotic growth of the average size.

## References

[1] Rostislav Grigorchuk, Zoran Sunik. Asymptotic aspects of Schreier graphs and Hanoi Towers groups. Department of Mathematics, Texas A&M University, MS-3368, College Station, TX, 77843-3368, USA, 2006

[2] Rostislav Grigorchuk, Zoran Sunik. SCHREIER SPECTRUM OF THE HANOI TOWERS GROUP ON THREE PEGS 2007

[3] R.I. Grigorchuk, V.V. Nekrashevich, V.I. Sushchanskiı, Automata, dynamical systems, and groups, Tr. Mat. Inst. Steklova 231 (2000) 134-214 (Din. Sist., Avtom. i Beskon. Gruppy).

[4] Bartholdi, Laurent; Siegenthaler, Olivier; Zalesskii, Pavel. The con-gruence subgroup problem for branch groups,Israel J. Math.187(2012),419450.

[5] Andreas M. Hinz,The Tower of Hanoi, Enseign. Math. (2)35(1989), no. 3-4, 289–321. MR MR1039949 (91k:05015)

## Appendix Appendix A Python interface for elements of automaton groups

A full implementation you can find at github.com/davendiy/automata-groups

```
# File: src/source2.py

class AutomataTreeNode(Tree):
    """ Recursive tree for element of Automata group.
    :param permutation: root permutation of node
    :param value:    name of element (will be printed on picture)
    :param reverse:  True if this node means the recursive callback of
                     the parent (\delta)
    :param simplify: True if it could be drawn as point with name
                     (e.g. for e, a, b, c)
    """
    def __init__(self, permutation=TRIVIAL_PERM, value=None, reverse=False,
                 simplify=False):
        ...
    @property
    def name(self):
        """ Get the name of a node.
        """
        ...

    def height(self) -> int:
        """ Get height of the w's tree.
        Complexity - \Theta(V) (DFS), where V is amount of vertices.
        """
        ...

    def vert_amount(self) -> int:
        """ Get amount of tree's vertices.
        Complexity - \Theta(V) (DFS).
        """
        ...

    def size(self) -> int:
        """ Get size w's tree.
        Complexity - \Theta(V) (DFS).
        """
        ...

    def add_child(self, child, position=None):
        """ Add a child (first-level section) to the root of the tree
        and put it on a (possibly not) given position.

        Complexity - \Theta(W) where W is amount of child's vetices
        (it adds copy of the child in order to escape cyclics).
        """
        ...

    def remove(self, child):
        """ Remove the child (first-level section) of the root.
        Complexity - O(k) where k is amount of root's children.
        """
        ...

```

```
1
2
3  class AutomataGroupElement:
4      """ Implementation of an element of an automaton group.
5
6      This class represents any element that has structure
7      g = \pi (g_1, g_2, g_3). Group defines by initialising an alphabet of
8      its generators (a1, a2, a3 ...) using __init__ method and then for every
9      word w from the alphabet corresponding element can be created using
10     function $$$ from_string $$$.
11
12     a = AutomataGroupElement(name, permutation, children)
13
14     :param name: the name of an atom (for instance a, b, c, e for H3)
15     :param permutation: object sympy.combinatorics.Permutation
16     :param children: a list of AutomataTreeNode or AutomataGroupElement
17                      elements that defines tree-like structure of the
18                      element (first-level state). Those elements
19                      should have $ name $ of the respective atom
20                      or have parameter $ reverse $ with value True
21                      that means recursive call.
22
23     Example of using (Hanoi tower group H^3):
24     >>> e = AutomataGroupElement('e', simplify=True)
25     >>> a = AutomataGroupElement('a', permutation=Permutation([1, 0, 2]), \
26                                  children=(e, e, reverse_node()), simplify=True)
27     >>> b = AutomataGroupElement('b', permutation=Permutation([2, 1, 0]), \
28                                  children=(e, reverse_node(), e), simplify=True)
29     >>> c = AutomataGroupElement('c', permutation=Permutation([0, 2, 1]), \
30                                  children=(reverse_node(), e, e), simplify=True)
31     >>> from_string('abcbabcbabcbabc')
32     abcbabcbabcbabc = (0 2) (acacacac, bbb, bbbb)
33
34
35     As you can see element is completely defined by it's tree structure.
36     The multiplication of two elements is just combining of their trees in
37     some way.
38
39     WARNING: you can't create in this way elements that reffers to each other,
40     for example
41                  a = ()(a, b, e)      and  b = ()(b, e, a)
42     because such elements don't have tree-like structure.
43     Well in fact, you can do it but I don't guarantee it will work properly.
44     """
45
46     def __init__(self, name, permutation=TRIVIAL_PERM,
47                      children=(AutomataTreeNode(reverse=True),
48                                AutomataTreeNode(reverse=True),
49                                AutomataTreeNode(reverse=True)),
50                      simplify=False):
51         ...
52
53     def __len__(self):
54         """ Returns the |w|.
55         E.g. |e| = 0, |ai| = 1.
56         """
57         ...
58
59     @property
60     def permutation(self) -> Permutation:
61         ...
```

```
62
63      def is_trivial(self) -> bool:
64          """ Returns whether element is trivial or not (algorithm will be
65          explained a bit later).
66
67          Complexity - O(V), where V is amount of vertices.
68          """
69          ...
70
71      @classmethod
72      def from_cache(cls, el: str):
73          """ Get already initialised element by name el.
74          """
75          ...
76
77      def __mul__(self, other):
78          """ Multiplication of AutomataGroupElements. Complexity - O(V_1 + V_2),
79          where V_1 and V_2 are amount of vertices of self and other respectively.
80          """
81          ...
82
83      def order(self) -> int:
84          """ Get the order of the element (algorithm will be explained a bit later).
85          """
86          ...
87
88      # and methods for drawing using matplotlib
89      ...
90
91
92  def from_string(w: str) -> AutomataGroupElement:
93      """ Get element that related to the given string.
94
95      For example, you can define atomic elements 'a' 'b' and 'c' and then
96      get an element 'abababababacbcbabc'
97
98      Complexity - O(n * V), where n = |w| and V = v(w) (amount of vertices)
99
100     :param w: a word over an alphabet formed by atomic elements.
101     """
102     ...
103
104 def reverse_node():        # auxiliary
105     return AutomataTreeNode(reverse=True)
106
107
108 def initial_state():
109     """ Initialise atomic elements for Hanoi tower Group.
110     """
111     global e, a, b, c
112     e = AutomataGroupElement('e', simplify=True)
113     a = AutomataGroupElement('a', permutation=Permutation([1, 0, 2]),
114                              children=(e, e, reverse_node()), simplify=True)
115     b = AutomataGroupElement('b', permutation=Permutation([2, 1, 0]),
116                              children=(e, reverse_node(), e), simplify=True)
117     c = AutomataGroupElement('c', permutation=Permutation([0, 2, 1]),
118                              children=(reverse_node(), e, e), simplify=True)
119
120 # initialisation is done during importing of this module
121 e = a = b = c = ...      # type: AutomataGroupElement
122 initial_state()
```