Московский государственный университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра математической кибернетики

Лобанов Алексей Андреевич

Нахождение минимальных клонов трёхзначных и четырёхзначных логик

КУРСОВАЯ РАБОТА

Научный руководитель:
доцент, д.ф.-м.н.
С.Н. Селезнёва

Москва, 2018

Содержание

1. Введение

В данной работе рассматривается задача построения всех минимальных замкнутых классов в трёхзначных и четырёхзначных логиках. Ранее, в работе [1] уже получены все такие классы для случая трёхзначной логики, а в [2] эти функции записаны в форме полиномов над $\mathbb{E}^3$. В [3] было указано сколько таких классов для случая четырёхзначной логики.

Эта задача интересна тем, что сложность проверки выполнимости системы ограничений, например, в коньюктивных запросах к базам данных, зависит только от того, какие функции сохраняют все отношения этой системы.

2. Постановка задачи

2.1. Основные определения

2.1.1. Конечнозначные функции

Пусть $k \geq 2, k \in \mathbb{N}$, множество $\mathbb{E}_k = \{0, 1, \ldots, k-1\}$

Функция f называется *k-значной*, если

$$f : \mathbb{E}_k^n \to \mathbb{E}_k,$$

где $n \in \mathbb{N}$. Обозначим через $\mathbb{P}_k$ множество всех k-значных функций.

K-значная функция называется *идемпотентной*, если $\forall i \in \{0, 1, \ldots, k-1\}$:

$$f(i, \ldots, i) = i$$

Обозначим все идемпотентные функции k-значной логики от n переменных как I_k^n.

Пусть $f(x, y, z) \in \mathbb{P}_k$ и существенно зависит от трёх переменных. Назовём её *функцией большинства*, если из неё можно получить $h(x', y', z')$ какой-либо перестановкой переменных, что выполняется:

$$h(x, x, y) = h(x, y, x) = h(y, x, x) = x$$

Пусть $f(x_1, \ldots, x_k)$, $k \geq 3$, не равна $x_1, ..., x_k$, и для неё верно, что $\exists i \in \mathbb{N}$: $1 \leq i \leq k$, что если среди элементов $x_1, \ldots, x_k$ хотя бы два совпадающих, то

$$f(x_1, \ldots, x_k) = x_i$$

назовём такую функцию *полупроекцией*.

2.1.2. Формула

Пусть $A \subseteq \mathbb{P}_k$. *Формула* над множеством A определяется по индукции:

1. *Базис индукции.* Если $f^n \in A$ – n-местная функция и $u_1, \ldots, u_n$ – набор из n произвольных переменных, то выражение $f(u_1, \ldots, u_n)$ – формула.

2. *Индуктивный переход.* Если $F_1, \ldots, F_n$ – уже построенные формулы или переменные и $f^n \in A$ – n-местная функция, то выражение $f(F_1, \ldots, F_n)$ – формула.

3. Других формул нет, т.е. каждая формула построена либо по базису индукции, либо по индуктивному переходу.

Каждая формула над множеством $A \subseteq \mathbb{P}_k$ задаёт некоторую *k-значную функцию*. Функция f_F, задаваемая формулой F определяется по индукции:

1. *Базис индукции.* Если $F = u$, где u – переменная, то $f_F = u$, т.е. функция f_F тождественно равна переменной u.

2. *Индуктивный переход.* Если $F = f(F_1, \ldots, F_n)$, где $F_1, \ldots, F_n$ – формулы или переменные и $f^n \in A$, то $f_F = f(f_{F_1}, \ldots, f_{F_n})$.

2.1.3. Замкнутый класс

Пусть $A \subseteq \mathbb{P}_k$. *Замыканием множества* A называется множетсво всех функций, задаваемых формулами над множеством A. Обозначим как $[A]$.

Если $[A] = A$, то A называется замкнутым классом.

2.2. Формулировка задач

В рамках данной курсовой работы рассматриваются следующие задачи:

1. Написать программу, которая строит все минимальные клоны, порождаемые какой-то идемпотентной двухместной функцией f трёхзначной логики. Получить экспериментальный результат в виде списка функций от двух переменных, содержащихся в каждом таком клоне.

2. Написать программу, которая строит все клоны, порождаемые какой-то идемпотентной двухместной функцией f четырёхзначной логики, не содержащие функций большинства и полупроекций. Получить экспериментальный результат в виде списка функций двух переменных, содержащихся в каждом таком клоне.

3. Основная часть

Рассматриваются 5 свойств минимальных классов:

1. Содержат константу.
2. Содержат функцию большинства.
3. Содержат $f(x,y)$, не равную x или y, для которой $f(x,x)=x$
4. Содержат $f(x,y,z)=x-y+z$, где $+$ операция коммутативной группы на $\mathbb{E}_k$
5. Содержат полупроекцию.

Если выполняются условия 1, 2 и 4, то задача проверки выполнимости ограничений полиномиальна, например, в случае. коньюктивных запросы к базе данных. Если выполняется только 5, то такая задача NP-полна. Если выполняется только 3, то этот случай ещё не достаточно исследован, поэтому решался именно он.

Введём некоторые обозначения: f_k^x, f_k^y – двухместные функции k-значной логики, тождественно равные своему первому и второму аргументу соответственно. Назовём "плохими" функции $f(x,y)$ из I_k^2, которые точно не смогут породить минимальные классы.

Простейшая реализация алгоритма решения данной задачи может не привести к успеху: для $k=4$ количество функций в $I_4^2 = 4^{4\cdot4-4} = 4^{12} = 16777216$. Построение класса по каждой из них слишком трудоёмко, если знать, что некоторая часть этих классов по размеру сопоставимы с самим I_4^2. Таким образом, уже для $k=4$ необходим более быстрый алгоритм.

Для написания алгоритма решения поставленной задачи, воспользуемся следующим утверждением.

***Утверждение* 1.** *Любая функция из I_k^2, кроме, быть может, f_k^x и f_k^y встречается не более, чем в одном минимальном классе*

Алгоритм представим в виде трёх частей:

1. **Генерация функций для перебора** В этой части мы должны оставить для рассмотрения только те функции $f(x,y)$ из I_k^2, которые удовлетворяют следующим свойствам:

   Проверка каждого из этих свойств тривиальна, в том числе и вычислительно, относительно других частей.

   Полученные функции мы кладём в очередь d.

2. **Расширение множеств функций** В этой части мы берём один элемент из очереди, производим "расширение" класса функций 1 и кладём в конец очереди. Причём расширение происходит таким образом, что на каждом расширении для одного и того же множества, $\frac{\mathsf{max}_{n+1}}{\mathsf{max}_n} = \lambda$, где $1 < \lambda < 2$, а n – номер расширения. Это необходимого для экономного расхода памяти ЭВМ.

3. **Обработка множеств функций после расширения** В этой части мы должны рассмотреть все обработанные множества функций и обработать их, псевдокод ниже 2. Это самая концептуально сложная часть алгоритма.

При такой декомпозиции возможно использовать многоядерность современных ЭВМ для распараллеливания расширения множеств функций, что позволяет почти линейно уменьшить время работы.

Опишем работу некоторых ключевых процедур. Главной такой является "расширение" множества функций 1.

Algorithm 1 Расширение множества функций

```
function ExtendFunctionClass(class, max)
    if size(class) ≥ max then
     return (class, False)
    end if
    is_finished ← False
    last_size ← 0
    while True do
        new_funcs ← {}                                  ▷ Пустое множество
        for all f_1 ∈ class do
            new_funcs.add(reversed(f_1))        ▷ Для всех f_1(x, y) добавим f'_1(y, x)
            for all f_2 ∈ class do
                new_funcs.add(f_1 (f_2, y))
            end for
        end for
        new_funcs.remove(f_k^x)
        new_funcs.remove(f_k^y)  ▷ Могли добавиться тождественные функции, их нужно убрать
        if size(new_funcs) = last_size then
            is_finished ← True
            break                ▷ Мы закончили построение этого класса функций
        end if
        class ← class ∪ new_funcs
        last_size ← size(new_funcs)
        if size(class) > max then
            break                ▷ Класс стал достаточно большим – нужно выходить
        end if
    end while
     return (class, is_finished)
end function
```

Также важной функцией является обработка новых множеств 2.

Для реализации полученных алгоритмов для выполнения на ЭВМ был выбран язык программирования C++14.

Algorithm 2 Обработка расширенных множеств

```
procedure ProcessSets(d, bad_functions)
    while Не все функции закончены do
        local_d ← d
        d.clear()
        local_d сортируем по размеру множества
        cur_size ← λ^n
        for all task ∈ local_d do
            if size(task) > cur_size then
                break        ▷ Текущие множества слишком большие, могут быть незаконченные
            end if
            if task не закончен then
                if в нём есть хотя бы одна "плохая" функция then
                    completed ← completed + 1        ▷ Текущий класс точно не минимальный
                else
                    d.add(task)                      ▷ Класс надо ещё расширить
                end if
            else
                completed ← completed + 1
                if в нём нет "плохих" функций then
                    положим класс в список минимальных классов
                end if
                все функции task сделаем "плохими"   ▷ Ни одна из этих функций уже не сможем породить минимальный класс
            end if
        end for
    end while
end procedure
```

4. Полученные результаты

С помощью написанных программ для ЭВМ на языке программирования C++14 удалось получить решения поставленных задач.

Опишем кодирование решения. Пусть исходная функция имеет значения

$$f\,(x,y) = (0ab\,c1d\,et2)$$

тогда число $\overline{abcdet}$ рассмотрим в десятичной системе и назовём *номером функции*.

Для трехзначной логике список номеров порождающих функций ниже:

0, 8, 10, 11, 16, 17, 20, 26, 33, 35, 36, 37, 38, 40, 41, 42, 43, 47, 53, 68, 71, 80, 116,122, 125, 178, 179, 188, 206, 215, 280, 281, 286, 287, 290, 296, 364, 368, 448, 449, 458, 528, 530, 557, 624, 692, 728. Всего 42 функции.

Для случая четырёхзначной логики таких функций 2279.

Список используемых источников

[1] B.Csakany – All minimal clones on three-element set

[2] Hajime Machida, Michael Pinsker – Polynomials as Generators of Minimal Clones

[3] Karsten Schölzel – The minimal clones generated by semiprojections on a four-element set

Приложение А. Исходный код программы на C++14

main.cpp

```
#include <iostream>
#include <fstream>
#include <vector>
#include <deque>
#include <algorithm>
#include <array>
#include <map>
#include <set>
#include <unordered_set>
#include <utility>
#include <cstring>
#include <functional>
#include <list>
#include <cctype>
#include <thread>
#include <mutex>
#include <atomic>
#include <iterator>

#include "al_utility.hpp"
#include "finite_function.hpp"

using namespace std;

const CellType CUR_BASE = 4;
const int ARGS_COUNT = 2;
const string CLASSES_FILENAME = "classes.txt";

FiniteFunction<CUR_BASE> identical_x;
FiniteFunction<CUR_BASE> identical_y;

template <CellType BASE>
pair<FiniteFunction<BASE>, FiniteFunction<BASE>> get_identicals() {
    if ( BASE == 3 )
        return make_pair(
            FiniteFunction<CUR_BASE>(string("000 111 222")),
            FiniteFunction<CUR_BASE>(string("012 012 012"))
        );
    else if ( BASE == 4 )
        return make_pair(
```

```
            FiniteFunction<CUR_BASE>(string("0000 1111 2222 3333")),
            FiniteFunction<CUR_BASE>(string("0123 0123 0123 0123"))
        );
}

struct FunctionTask {
    bool is_finished;
    vector<FiniteFunction<CUR_BASE>> current;
};

template <size_t BASE>
class FixedIniter {
    public:
        // rules это отображение обязательных значений аргументов, к
        // значениями функции, которые не должны при них изменяться
        explicit FixedIniter(map<pair<size_t, size_t>, int> rules) : _rules(rules), _cu
            for (auto it = rules.begin(); it != rules.end(); ++it) {
                size_t arg1 = it->first.first;
                size_t arg2 = it->first.second;
                int value = it->second;
                _cur_values.at(FiniteFunction<BASE>::get_index(arg1, arg2)) = value;

                _used_indexes.insert(FiniteFunction<BASE>::get_index(arg1, arg2));
            }
        }

        int operator() (int first, int second) const {
            return _cur_values[FiniteFunction<BASE>::get_index(first, second)];
        }

        // Возвращает true, пока может построить следующую функцию
        // если не может это сделать, то возвращает false
        bool set_next() {
            bool is_overflow = true;

            for (size_t i = 0; is_overflow; ++i) {
                if ( i >= _cur_values.size() )
                    return false;
                if ( _used_indexes.count(i) > 0 )
                    continue;
                _cur_values.at(i) += 1;
                bool is_overflow = (_cur_values.at(i) > static_cast<int>(BASE) - 1);
                if ( is_overflow )
                    _cur_values.at(i) = 0;
```

```
                else
                    return true;
            }
            throw "Wrong logic";
        }
    private:
        set<size_t> _used_indexes;
        map<pair<size_t, size_t>, int> _rules;
        vector<int> _cur_values;
};

template <class Iterable>
void write_function_class(ofstream &f_out, Iterable begin, Iterable end) {
    for (;begin != end; ++begin)
        f_out << *begin << "   ";
    f_out << endl;
}

template <class ClassesContainer>
void append_classes(const ClassesContainer& classes) {
    std::ofstream f_out(CLASSES_FILENAME.c_str(), ios_base::app);
    for (const auto& func_class: classes) {
        write_function_class(f_out, func_class.begin(), func_class.end());
    }
    f_out.close();
}

/*
template <int BASE>
void get_permutations(vector<array<int, BASE>> &permutations) {
    permutations.clear();
    array<int, BASE> cur_perm;
    for (int i = 0; i < BASE; ++i)
        cur_perm[i] = i;
    do {
        permutations.push_back(cur_perm);
    } while ( next_permutation(cur_perm.begin(), cur_perm.end()) );
}
*/

template<class TripleArgsFiniteFunction>
bool is_one_arg_func(const TripleArgsFiniteFunction &h) {
    bool is_equaled_x = true;
    bool is_equaled_y = true;
```

```
    bool is_equaled_z = true;
    for (CellType x = 0; x < CUR_BASE; ++x)
        for (CellType y = 0; y < CUR_BASE; ++y) {
            for (CellType z = 0; z < CUR_BASE; ++z) {
                auto h_res = h(x,y,z);
                is_equaled_x = is_equaled_x && (h_res == x);
                is_equaled_y = is_equaled_y && (h_res == y);
                is_equaled_z = is_equaled_z && (h_res == z);
            }
            if ( !is_equaled_x && !is_equaled_y && !is_equaled_z )
                return false;
        }
    return true;
}

template<class FourthArgsFiniteFunction>
bool is_one_arg_func_fourth(const FourthArgsFiniteFunction &h) {
    bool is_equaled_x = true;
    bool is_equaled_y = true;
    bool is_equaled_z = true;
    bool is_equaled_w = true;
    for (CellType x = 0; x < CUR_BASE; ++x)
        for (CellType y = 0; y < CUR_BASE; ++y) {
            for (CellType z = 0; z < CUR_BASE; ++z) {
                for (CellType w = 0; w < CUR_BASE; ++w) {
                    auto h_res = h(x,y,z,w);
                    is_equaled_x = is_equaled_x && (h_res == x);
                    is_equaled_y = is_equaled_y && (h_res == y);
                    is_equaled_z = is_equaled_z && (h_res == z);
                    is_equaled_w = is_equaled_w && (h_res == w);
                }
            }
            if ( !is_equaled_x && !is_equaled_y && !is_equaled_z && !is_equaled_w)
                return false;
        }
    return true;
}

template<class TripleArgsFiniteFunction>
bool is_projection(const TripleArgsFiniteFunction &h) {
    bool is_projection = true;
    for (CellType x = 0; x < CUR_BASE; ++x)
        for (CellType y = 0; y < CUR_BASE; ++y) {
            is_projection = (is_projection
```

```
                && h(x,x,y) == h(x,y,x)
                && h(x,y,x) == h(y,x,x)
                && h(y,x,x) == x);
        }
    return is_projection;
}

template<class FourthArgsFiniteFunction>
bool is_projection_fourth(const FourthArgsFiniteFunction &h) {
    bool is_projection = true;
    for (CellType x = 0; x < CUR_BASE; ++x)
        for (CellType y = 0; y < CUR_BASE; ++y) {
            is_projection = (is_projection
                && h(x,x,x,y) == h(x,x,y,x)
                && h(x,x,y,x) == h(x,y,x,x)
                && h(x,y,x,x) == h(y,x,x,x)
                && h(y,x,x,x) == x);
        }
    return is_projection;
}

template<class TripleArgsFiniteFunction>
bool is_semiprojection(const TripleArgsFiniteFunction &h) {
    bool is_equaled_x = true;
    bool is_equaled_y = true;
    bool is_equaled_z = true;
    for (CellType x = 0; x < CUR_BASE; ++x)
        for (CellType y = 0; y < CUR_BASE; ++y)
            for (CellType z = 0; z < CUR_BASE; ++z) {
                // если все различны, то не рассматриваем
                if ( x != y && x != z && y != z )
                    continue;

                auto h_res = h(x,y,z);
                is_equaled_x = is_equaled_x && (h_res == x);
                is_equaled_y = is_equaled_y && (h_res == y);
                is_equaled_z = is_equaled_z && (h_res == z);
                if (
                        !is_equaled_x
                     && !is_equaled_y
                     && !is_equaled_z
                )
                    return false;
            }
```

```
    return is_equaled_x || is_equaled_y || is_equaled_z;
}

template<class FourthArgsFiniteFunction>
bool is_semiprojection_fourth(const FourthArgsFiniteFunction &h) {
    bool is_equaled_x = true;
    bool is_equaled_y = true;
    bool is_equaled_z = true;
    bool is_equaled_w = true;
    for (CellType x = 0; x < CUR_BASE; ++x)
        for (CellType y = 0; y < CUR_BASE; ++y)
            for (CellType z = 0; z < CUR_BASE; ++z) {
                for (CellType w = 0; w < CUR_BASE; ++w) {
                    set<CellType> s;
                    s.insert(x);
                    s.insert(y);
                    s.insert(z);
                    s.insert(w);
                    if ( s.size() == 4 )
                    // если все различны, то не рассматриваем
                        continue;

                    auto h_res = h(x,y,z,w);
                    is_equaled_x = is_equaled_x && (h_res == x);
                    is_equaled_y = is_equaled_y && (h_res == y);
                    is_equaled_z = is_equaled_z && (h_res == z);
                    is_equaled_w = is_equaled_w && (h_res == w);
                    if (
                            !is_equaled_x
                         && !is_equaled_y
                         && !is_equaled_z
                         && !is_equaled_w
                    )
                        return false;
                }
            }
    return is_equaled_x || is_equaled_y || is_equaled_z || is_equaled_w;
}

bool is_passed_rosenberg(const FiniteFunction<CUR_BASE> &f) {
    auto h_1 = [f](const CellType x, const CellType y, CellType z) -> CellType {
        return f( f(x,y), f(x,z) );
    };
    if ( !is_one_arg_func(h_1) ) {
```

```
    if ( is_projection(h_1) || is_semiprojection(h_1) )
        return false;
}

auto h_2 = [f](const CellType x, const CellType y, CellType z) -> CellType {
    return f( f(x, y), f(z, x) );
};
if ( !is_one_arg_func(h_2) ) {
    if ( is_projection(h_2) || is_semiprojection(h_2) )
        return false;
}

auto h_3 = [f](const CellType x, const CellType y, CellType z) -> CellType {
    return f( f(x,y), f(y,z) );
};
if ( !is_one_arg_func(h_3) ) {
    if ( is_projection(h_3) || is_semiprojection(h_3) )
        return false;
}

auto h_4 = [f](const CellType x, const CellType y, CellType z) -> CellType {
    return f( f(x,y), f(z,y) );
};
if ( !is_one_arg_func(h_4) ) {
    if ( is_projection(h_4) || is_semiprojection(h_4) )
        return false;
}

if ( CUR_BASE == 4 ) {
    auto g_1 = [f, h_1](const CellType x, const CellType y, CellType z, CellType u)
        return f( h_1(x,y,z), u );
    };
    if ( !is_one_arg_func_fourth(g_1) ) {
        if ( is_projection_fourth(g_1) || is_semiprojection_fourth(g_1) )
            return false;
    }

    auto g_2 = [f, h_2](const CellType x, const CellType y, CellType z, CellType u)
        return f( h_2(x,y,z), u );
    };
    if ( !is_one_arg_func_fourth(g_2) ) {
        if ( is_projection_fourth(g_2) || is_semiprojection_fourth(g_2) )
            return false;
    }
```

```
        auto g_3 = [f, h_3](const CellType x, const CellType y, CellType z, CellType u)
            return f( h_3(x,y,z), u );
        };
        if ( !is_one_arg_func_fourth(g_3) ) {
            if ( is_projection_fourth(g_3) || is_semiprojection_fourth(g_3) )
                return false;
        }

        auto g_4 = [f, h_4](const CellType x, const CellType y, CellType z, CellType u)
            return f( h_4(x,y,z), u );
        };
        if ( !is_one_arg_func_fourth(g_4) ) {
            if ( is_projection_fourth(g_4) || is_semiprojection_fourth(g_4) )
                return false;
        }

        auto g_both = [f](const CellType x, const CellType y, CellType z, CellType u) -
            return f( f(x,y), f(z,u) );
        };
        if ( !is_one_arg_func_fourth(g_both) ) {
            if ( is_projection_fourth(g_both) || is_semiprojection_fourth(g_both) )
                return false;
        }
    }

    return true;
}

template <CellType BASE>
vector<FiniteFunction<BASE>> get_funcs() {
    size_t count = 0;

    map<pair<size_t, size_t>, int> rules;
    for (int i = 0; i < BASE; ++i)
        rules[make_pair<size_t, size_t>(i,i)] = i;
    FixedIniter<BASE> initer(rules);

    //vector<array<int, BASE>> permutations;
    //get_permutations<BASE>(permutations);

    vector<FiniteFunction<BASE>> funcs;
    set< FiniteFunction<BASE> > permutated_funcs;
    do {
```

```
        ++count;
        auto cur_func = FiniteFunction<BASE>(initer);
        funcs.push_back(cur_func);
    } while (initer.set_next());
    cout << "Total " << count << " functions" << endl;

    funcs.erase(
        remove_if(
            funcs.begin(),
            funcs.end(),
            [](const FiniteFunction<BASE> & f) {
                return !is_passed_rosenberg(f);
            }
        ),
        funcs.end()
    );

    cout << "After Rosenberg " << funcs.size() << " functions" << endl;
    //if ( permutated_funcs.size() != count )
     //   throw "Permutation's logic error!";
    return funcs;
}


template <CellType BASE>
set<FiniteFunction<BASE>> generate_function_class(FiniteFunction<BASE> base_function) {
    auto FiniteFunctionHasher = [](const FiniteFunction<BASE> &f) -> uint32_t {
        return f.get_hash();
    };
    unordered_set<
        FiniteFunction<CUR_BASE>,
        decltype(FiniteFunctionHasher)
    > func_class(1024, FiniteFunctionHasher);
    func_class.insert(base_function);
    //cout << "start with " << base_function << " ";

    size_t last_size = 0;  // размер в прошлой итерации
    while (true) {
        unordered_set<
            FiniteFunction<CUR_BASE>,
            decltype(FiniteFunctionHasher)
        >  new_funcs(1024, FiniteFunctionHasher);

        for (const auto& f_main: func_class) {
```

```
            new_funcs.insert(f_main.reversed());
            //new_funcs.insert(f_main.equaled());
            for (const auto& f_applied: func_class) {
                //FiniteFunction<BASE> f_left_1, f_left_2;
                //tie(f_left_1, f_left_2) = f_main.apply_to_first(f_applied);
                FiniteFunction<BASE> f_left;
                f_left = f_main.apply_to_first_partial(f_applied);
                new_funcs.insert(f_left);
            }
        }
        if ( new_funcs.size() == last_size ) {
            break;
        }

        func_class.insert(new_funcs.begin(), new_funcs.end());
        last_size = new_funcs.size();
    }
    return set<FiniteFunction<BASE>>(func_class.begin(), func_class.end());
}


// Возвращает новый функциональный класс, размером не сильно больше, чем max_size
// Вторым результатом возвращает true, если вычисления закончились успешно
// и false, если прервались по достижению max_size
template <CellType BASE>
pair<vector<FiniteFunction<BASE>>, bool> extend_function_class(
        const vector<FiniteFunction<BASE>>& base_class,
        size_t max_size
) {
    if ( base_class.size() ≥ max_size ) {
        return make_pair(
            vector<FiniteFunction<BASE>>(base_class),
            false
        );
    }
    auto FiniteFunctionHasher = [](const FiniteFunction<BASE> &f) -> uint32_t {
        return f.get_hash();
    };
    unordered_set<
        FiniteFunction<CUR_BASE>,
        decltype(FiniteFunctionHasher)
    > func_class(1024, FiniteFunctionHasher);
    for (auto&& base_function: base_class)
        func_class.insert(base_function);
```

```
    bool is_finished = false;
    size_t last_size = 0;  // размер в прошлой итерации
    while (true) {
        unordered_set<
            FiniteFunction<CUR_BASE>,
            decltype(FiniteFunctionHasher)
        >  new_funcs(1024, FiniteFunctionHasher);

        for (const auto& f_main: func_class) {
            new_funcs.insert(f_main.reversed());
            for (const auto& f_applied: func_class) {
                FiniteFunction<BASE> f_left;
                f_left = f_main.apply_to_first_partial(f_applied);
                new_funcs.insert(f_left);
            }
        }
        new_funcs.erase(identical_x);
        new_funcs.erase(identical_y);
        if ( new_funcs.size() == last_size ) {
            is_finished = true;
            break;
        }

        func_class.insert(new_funcs.begin(), new_funcs.end());
        last_size = new_funcs.size();

        // слишком много насчитали -- выходим
        if ( func_class.size() > max_size ) {
            break;
        }
    }
    vector<FiniteFunction<BASE>> res(func_class.begin(), func_class.end());
    res.shrink_to_fit();
    return make_pair(
        res,
        is_finished
    );
}


template <class Iterable>
bool is_bad_class(
        const Iterable& func_class,
```

```
        const set<FiniteFunction<CUR_BASE>>& bad_funcs
) {
    for (auto&& func: func_class)
        if ( bad_funcs.find(func) != bad_funcs.end() )
            return true;
    return false;
}


size_t total_possible_functions;
atomic<long> completed_tasks;
atomic<long> tasks_to_extend; // количество тасков, которые не обработаны
list< vector<FiniteFunction<CUR_BASE>> > shared_function_classes;
mutex shared_functions_mutex;


deque<FunctionTask> task_list;
mutex task_mutex;

vector<FunctionTask> processed_task_list;
mutex processed_task_mutex;

set<FiniteFunction<CUR_BASE>> bad_functions;
set<FiniteFunction<CUR_BASE>> good_functions;
atomic<int> current_max_coeff;

void do_work() {
    std::chrono::milliseconds SLEEP_TIME(10);
    while ( true ) {
        FunctionTask task;
        task_mutex.lock();
        if ( task_list.begin() != task_list.end() ) {
            task = task_list.front();
            task_list.pop_front();
            task_mutex.unlock();
        } else {
            task_mutex.unlock();
            if ( completed_tasks < total_possible_functions ) {
                // Не все таски, подождём, пока добавят ещё
                std::this_thread::sleep_for(SLEEP_TIME);
                continue;
            } else {
                cout << "thread " << this_thread::get_id() << ": "
                    << " finished" << endl;
```

```
                break;
            }
        }

        tie(task.current, task.is_finished) = extend_function_class(
            task.current,
            get_math_coeff(current_max_coeff)
        );
        processed_task_mutex.lock();
        processed_task_list.push_back(task);
        processed_task_mutex.unlock();
        --tasks_to_extend;
    }

}


void process_task_lists() {
    cout << "processing starts " << endl;

    std::chrono::milliseconds SLEEP_TIME(10);

    vector<FunctionTask> local_processed_tasks;

    while ( completed_tasks < total_possible_functions ) {
        if ( tasks_to_extend ) {
            // подождём, пока не закончатся таски в очереди
            std::this_thread::sleep_for(SLEEP_TIME);
            continue;
        }
        task_mutex.lock();
        if ( task_list.size() != 0 )
            cout << "IMPOSSIBLE task_list.size!!" << endl;
        task_list.clear();
        task_list.shrink_to_fit();
        task_mutex.unlock();

        // опустошим выполненные таски
        processed_task_mutex.lock();
        for (auto && task: processed_task_list)
            local_processed_tasks.push_back(task);
        processed_task_list.clear();
        processed_task_list.shrink_to_fit();
        processed_task_mutex.unlock();
```

```
cout << "sorting finished of "<<  local_processed_tasks.size() << endl;
size_t total_funcs = 0;
for (auto&& task: local_processed_tasks)
    total_funcs += task.current.size();
cout << "estimated size: "
   << sizeof(FunctionTask) * total_funcs / 1024 / 1024 << " MB" << endl;
// Обеспечим увеличение размеров, чтобы не было проблем со включением одного
// в другое
sort(
    local_processed_tasks.begin(),
    local_processed_tasks.end(),
    [](const FunctionTask& a, const FunctionTask& b) {
        return a.current.size() < b.current.size();
    }
);

// сохраним, чем должен был равняться максимальный размер
auto last_math_coeff = get_math_coeff(current_max_coeff);
// а для всех новых увеличим его
++current_max_coeff;

for ( auto&& task: local_processed_tasks ) {
    // если в каких-то больше, чем положено, то не трогаем их
    if ( task.current.size() >  last_math_coeff)
        break;
    if ( !task.is_finished ) {
        if ( is_bad_class(task.current, bad_functions) ) {
            //cout << "bad class" << endl;
            ++completed_tasks;
            if ( print_progress(completed_tasks, total_possible_functions) ) {
                append_classes(shared_function_classes);
                shared_function_classes.clear();
            }
        } else {
            task_mutex.lock();
            ++tasks_to_extend;
            task_list.push_back(task);
            task_mutex.unlock();
        }

    } else {
        //cout << "task finished, appending" << endl;
        ++completed_tasks;
```

```
            if ( print_progress(completed_tasks, total_possible_functions) ) {
                append_classes(shared_function_classes);
                shared_function_classes.clear();
            }
            sort(task.current.begin(), task.current.end());
            auto func_class = task.current;
            bool is_need_append = true;
            vector<decltype(shared_function_classes)::iterator> functions_to_remove

            is_need_append = !is_bad_class(task.current, bad_functions);

            for (auto&& f: task.current)
                if ( good_functions.find(f) == good_functions.end() ) {
                    is_need_append = false;
                    break;
                }

            // Делаем плохими ВСЕ функции без учёта того, порждают ли они
            // минимальный класс. Очень опасно! Должны гарантировать, что
            // классы меньше быть не могут, потому что мы их всех уже
            // перебрали
            // иначе ДОЛЖНО быть в if ( is_need_append )
            for (auto&& func: func_class)
                bad_functions.insert(func);
            if ( is_need_append ) {
                shared_function_classes.push_back(func_class);
            }
            for (auto&& to_remove: functions_to_remove) {
                shared_function_classes.erase(to_remove);
                cout << "Removing class from shared_function_classes" << endl;
            }
        }
    }
    local_processed_tasks.erase(
        remove_if(
            local_processed_tasks.begin(),
            local_processed_tasks.end(),
            [last_math_coeff](const FunctionTask& task) {
                return task.current.size() ≤ last_math_coeff;
            }),
        local_processed_tasks.end()
    );
    local_processed_tasks.shrink_to_fit();
```

```
        // Поспим, чтобы не работать слишком часто
        std::this_thread::sleep_for(SLEEP_TIME);
    }
}


int main() {
    tie(identical_x, identical_y) = get_identicals<CUR_BASE>();

    cout << "sizeof FiniteFunction<" << (int)CUR_BASE << "> = "
    << sizeof(FiniteFunction<CUR_BASE>) << endl;
    cout << "sizeof FunctionTask = "<< sizeof(FunctionTask) << endl;
    auto FiniteFunctionHasher = [](const FiniteFunction<CUR_BASE> &f) -> uint32_t {
        return f.get_hash();
    };

    auto THREADS_COUNT = max(static_cast<int>(thread::hardware_concurrency()), 2);
    cout << "Using " << THREADS_COUNT << " threads" << endl;

    auto funcs = get_funcs<CUR_BASE>();
    cout << "Removing permutations " << funcs.size() << " functions" << endl;

    list< set<FiniteFunction<CUR_BASE>> > function_classes;
    unordered_set<
        FiniteFunction<CUR_BASE>,
        decltype(FiniteFunctionHasher)
    > allowed_functions(funcs.begin(), funcs.end(), 128, FiniteFunctionHasher);

    allowed_functions.erase(identical_x);
    allowed_functions.erase(identical_y);

    completed_tasks = 0;
    current_max_coeff = 1;
    for (auto&& func: allowed_functions) {
        good_functions.insert(func);
        FunctionTask task;
        task.current = {func};
        task.is_finished = false;
        task_list.push_back(task);
    }

    cout << "Total funcs in list " << task_list.size() << " functions" << endl;

    total_possible_functions = task_list.size();
```

```
    tasks_to_extend = task_list.size();
    vector< thread > thread_pool;
    thread task_processer(process_task_lists);

    // удалим старый контент
    std::ofstream f_out("classes.txt");
    f_out.close();

    for (int i = 0; i < THREADS_COUNT - 1; ++i)
        thread_pool.push_back(thread(do_work));

    for (auto&& t: thread_pool)
        t.join();

    task_processer.join();

    cout << "Shared " << shared_function_classes.size() << " functions!" << endl;
    // перегоняем список с классами в массив с классами
    vector< vector<FiniteFunction<CUR_BASE>> > vector_classes(
        shared_function_classes.begin(),
        shared_function_classes.end()
    );

    append_classes(vector_classes);
    return 0;
}
```

al_utility.hpp

```
#ifndef _AL_UTILITY_
#define _AL_UTILITY_

#include <iostream>
#include <iomanip>
#include <sstream>
#include <cmath>

bool print_progress(long current, long total, double print_every=0.1);
int get_math_coeff(int k);
#endif // _AL_UTILITY_
```

al_utility.cpp

```cpp
#include "al_utility.hpp"

bool print_progress(long current, long total, double print_every) {
    long integer_part = total * print_every;
    if ( current % integer_part != 0 )
        return false;

    // сделаем потокобезопасным
    std::stringstream ss;
    ss << "Progress " << current << " of " << total
       << " " << std::fixed << std::setw(5) << std::setprecision(2)
       << 100. * current / total << "%" << std::endl;
    std::cout << ss.str();
    return true;
}

int get_math_coeff(int k) {
    return static_cast<int>(pow(1.1, k + 2));
}
```

finite_function.hpp

```cpp
#include <array>
#include <string>
#include <utility>
#include <iostream>
typedef uint8_t CellType;

// Пока положим, что только два аргумента, чтобы упростить реализацию
template <CellType BASE>
class FiniteFunction {
    public:
        FiniteFunction() {}

        // Устанавливает результат функции, использую другую функцию как
        // инициализатор
        template <class Callable>
        FiniteFunction(Callable initer) : _num(0) {
            for (auto && el: _results)
                el = 0;
            for (CellType first = 0; first < BASE; ++first)
                for (CellType second = 0; second < BASE; ++second) {
```

```
            auto inited_result = initer(first, second);
            set_result(first, second, inited_result);
        }
    update_num();
}

explicit FiniteFunction(std::string text_repr) {
    for (auto && el: _results)
        el = 0;
    size_t cur_ind = 0;
    for (auto ch: text_repr) {
        if ( !isdigit(ch) )
            continue;
        set_result(
            cur_ind / BASE,
            cur_ind % BASE,
            std::stoi(std::string(1, ch))
        );
        ++cur_ind;
    }
    update_num();
}
~FiniteFunction() {}

//  Устанавливает результат функции по двум аргументам
void set_result(CellType first, CellType second, CellType result) {
    //std::cout << "setting " << (int)first << "," << (int)second << " to " <<

    uint8_t byte_num = get_index(first, second) / 4;
    uint8_t shift = 2*(get_index(first, second) % 4);
    //std::cout << "before: " << (int)_results[byte_num];
    _results[byte_num] = (_results[byte_num] ^ (((_results[byte_num] >> shift)
     & 0x03) << shift)) | (result <<shift);
     //std::cout << " -> " << (int)_results[byte_num] << std::endl;
}

int operator() (CellType first, CellType second) const {
    uint8_t byte_num = get_index(first, second) / 4;
    uint8_t shift = 2*(get_index(first, second) % 4);
    // возьмём последние два бита
    return (_results[byte_num] >> shift) & 0x03;
}

bool operator < (const FiniteFunction &f) const {
```

```
    return _num < f._num;
}

bool operator == (const FiniteFunction &f) const {
    return _num == f._num;
}

FiniteFunction reversed() const {
// f(x,y) -> f(y,x)
    FiniteFunction res;
    for (CellType first = 0; first < BASE; ++first)
        for (CellType second = 0; second < BASE; ++second) {
            res.set_result(first, second, (*this)(second, first));
        }
    res.update_num();
    return res;
}

FiniteFunction equaled() const {
// f(x,y) -> f(x,x)
    FiniteFunction res;
    for (CellType first = 0; first < BASE; ++first)
        for (CellType second = 0; second < BASE; ++second) {
            res.set_result(first, second, (*this)(first, first));
        }
    res.update_num();
    return res;
}

FiniteFunction apply_to_first_partial(const FiniteFunction &g) const {
    // -> this(g(x,y), x)
    FiniteFunction res;
    for (CellType first = 0; first < BASE; ++first)
        for (CellType second = 0; second < BASE; ++second) {
            res.set_result(first, second, (*this)(g(first, second), first));
        }
    res.update_num();
    return res;
}

std::tuple<FiniteFunction, FiniteFunction> apply_to_first(const FiniteFunction
    // -> this(g(x,y), x), this(g(x,y), y)
    FiniteFunction res_1, res_2;
    for (CellType first = 0; first < BASE; ++first)
```

```
            for (CellType second = 0; second < BASE; ++second) {
                res_1.set_result(first, second, (*this)(g(first, second), first));
                res_2.set_result(first, second, (*this)(g(first, second), second));
            }
        res_1.update_num();
        res_2.update_num();
        return std::make_tuple(res_1, res_2);
    }

    FiniteFunction apply_two(const FiniteFunction &g, const FiniteFunction &h) cons
        FiniteFunction res;
        for (CellType first = 0; first < BASE; ++first)
            for (CellType second = 0; second < BASE; ++second) {
                res.set_result(
                    first,
                    second,
                    (*this)(g(first, second), h(first, second))
                );
            }
        res.update_num();
        return res;
    }

    FiniteFunction apply_permutation(std::array<int, BASE> perm) const {
        FiniteFunction res;
        for (CellType first = 0; first < BASE; ++first)
            for (CellType second = 0; second < BASE; ++second) {
                res.set_result(
                    first,
                    second,
                    std::find(
                        perm.begin(),
                        perm.end(),
                        (*this)(perm[first], perm[second])
                    ) - perm.begin()
                );
            }
        res.update_num();
        return res;
    }

    uint32_t get_hash() const {
        return _num;
    }
```

```
        static size_t get_index(CellType first, CellType second) {
            return first * BASE + second;
        }
        template <CellType _BASE>
        friend std::ostream& operator << (std::ostream& os, const FiniteFunction<_BASE>
    private:
        void update_num() {
            _num = 0;
            for (auto&& val: _results) {
                _num *= 256;
                _num += val;
            }
        }
        uint32_t _num;
        std::array<CellType, (BASE*BASE + 3) / 4> _results;
};


template <CellType BASE>
std::ostream& operator << (std::ostream& os, const FiniteFunction<BASE> &f){
    for (int first = 0; first < BASE; ++first) {
        for (int second = 0; second < BASE; ++second) {
            os << f(first, second);
        }
        if ( first != BASE-1 )
            os << " ";
    }
    return os;
}
```