О программной реализации построения группоида автомата PRI и связанных вычислительных экспериментах

Мэн Линцянь, Б. Ф. Мельников

***Аннотация*—Настоящая статья является продолжением предыдущей статьи тех же авторов, в которой рассматривались варианты определения группоида и полугруппы первичного автомата PRI. Для изучения разных возникающих примеров группоида, который в важных частных случаях является полугруппой, мы реализовали на языке Си++ соответствующие компьютерные программы и выполнили вычислительные эксперименты – которые и описываем в настоящей статье.**

Основной целью создания программы и проведения вычислительных экспериментов является попытка понимания внутренней структуры получаемых группоидов (в частных случаях – полугрупп), а также попытка «объяснения природы» рассматривавшихся во многих предыдущих статьях специальных бинарных отношений (так называемых отношений покрываемости и эквивалентности в бесконечности), заданных на множествах конечных формальных языков.

Описываемая в статье программа показала свою эффективность при исследовании сложных примеров, в которых число элементов получаемого группоида измерялось несколькими десятками. Программа выдавала интересные для дальнейшей интерпретации результаты, и при этом время её выполнения примерно совпадало с теоретически вычисленными значениями.

Основными выводами, которые получены, в частности, в качестве анализа проведённых вычислительных экспериментов, можно считать следующие.

Во-первых очевидно, что языки над одним и тем же однобуквенным алфавитом эквивалентны в бесконечности. Между условием получения полугруппы и эквивалентностью в бесконечности существует связь, которая показана именно на этих языках. Мы показываем, что для таких языков (над однобуквенном алфавитом) группоид всегда является полугруппой, причём последняя всегда коммутативна и является моноидом.

Во-вторых показано, что, поскольку при вычислении интересных нам характеров и фактор-представлений элементов полугруппы ряды степеней содержат информацию об идемпотентах как множество пар, состоящих из индекса и периода, то можно не находить идемподенты получаемых полугрупп отдельно, а расчитать их число после вычислений рядов степеней.

Таким образом, на основе рассматриваемых полученных программой примеров мы получаем описания свойств группоида, которые впоследствии могут быть сформулированы в виде утверждений и теорем.

***Ключевые слова*—формальные языки, морфизмы, конечные автоматы, группоиды и полугруппы, алгоритмы и программы, Си++.**

Статья получена 24 февраля 2024 г.

Мэн Линцянь, Московский государственный университет имени М. В. Ломоносова (kostya45e@mail.ru).

Борис Феликсович Мельников, Университет МГУ – ППИ в Шэньчжэне (bormel@smbu.edu.cn).

I. ВВЕДЕНИЕ И МОТИВАЦИЯ

Настоящая статья является продолжением предыдущей статьи [1], в которой рассматривались варианты определения группоида и полугруппы первичного автомата PRI. Для изучения разных примеров, некоторые из которых были рассмотрены в предыдущей статье, мы реализовали соответствующие компьютерные программы и выполнили вычислительные эксперименты – которые и описываем в настоящей статье. Описываемые в статье компьютерные программы выполнены на языке Си++.

Основная цель проведённых вычислительных эксрпериментов заключалась в том, чтобы понять внутреннюю структуру соответствующих группоидов (в частных случаях – полугрупп), а также попытаться «объяснить природу» рассматривавшегося во многих предыдущих статьях бинарных отношений $\trianglelefteq$ и $\trianglelefteq\!\!\trianglelefteq$ (так называемых отношений «покрываемости» и «эквивалентности в бесконечности»), заданных на множествах конечных формальных языков.

К сказанному в предыдущем абзаце следует добавить два важных замечания, связанных, в первую очередь, с алгеброй полугрупп и теорией конечных автоматов. Во-первых, мы при исследовании свойств полугрупп первичного автомата PRI фактически рассматриваем подмоноиды глобального надмоноида («супермоноида») свободного моноида – причём только такие, что каждый из них может быть определён некоторым конечным языком; это можно увидеть на основе названия статьи [2], и, более того, на основе материала вышедшей существенно раньше статьи [3]. Во-вторых, при описании нескольких возможных вариантов конечных автоматов, соответствующих бесконечным итерационным деревьям морфизмов ([4], [5] и др.), мы – при необходимости рассмотрения всех возможных конечных автоматов $\mathrm{PRI}(A, B)$ для заданного (зафиксированного) языка B и произвольного языка A – должны рассмотреть достаточно большое число различных автоматов [1]; вместо всех этих автоматов можно рассмотреть *один* объект – группоид, который можно считать описанием всех возможных *автоматов* PRI.

Приведём содержание статьи по разделам.

В разделе II приводятся необходимые для настоящей статьи основные понятия, ранее определённые в наших предыдущих работах. Эти понятия мы по возможности

[1] Это число достаточно велико, хотя и не бесконечно: как показано в наших предыдущих публикациях, нам достаточно рассмотреть все такие языки A, у каждого из которых все слова образуют некоторое подмножество множества собственных префиксов языка B.

согласовывали с терминологией классических монографий.

В разделе III дано краткое представление общей структуры описываемой компьютерной программы – её «дайджест».

В разделе IV мы приводим описание основной части программы – то есть то, как мы будем представлять алфавит, языки, автоматы и получаемый группоид (полугруппу).

В разделе V представлены алгоритмы создания алфавита, языка и подмножеств собственных префиксов. Также показана реализация функций, необходимых для операций покрытия языков и создания первичного автомата PRI.

Ранее в [1] авторами были описаны некоторые алгебраические свойства и характерные элементы для группоида автомата PRI и, в частных случаях, соответствующей полугруппы. В разделе VI приведены описания функций, реализованных для исследования таких свойств. На основе этих описаний легко оценить временну́ю сложность основных алгоритмов, необходимых для построения полугруппы автомата PRI.

В разделе VII рассматриваются конкретные примеры построения и исследования получаемого группоида – либо соответствующей полугруппы.

В заключении (раздел VIII) мы проводим анализ результатов вычислений и делаем некоторые выводы.

II. ОСНОВНЫЕ ОБОЗНАЧЕНИЯ

Приведём некоторые основные понятия, ранее определённые в наших предыдущих работах [4], [5], [6], [7], [8], [9], [10]. Эти понятия мы по возможности согласовывали с классическими монографиями [11], [12], [13], [14], [15], [16].

Сначала рассмотрим основное необходимое нам понятие – т. н. отношение покрытия для множества конечных языков. Если для конечных языков A и B выполнено условие

$$(\forall u \in A^*)(\exists v \in B^*)(u \in \mathrm{opref}(v)),$$

то будем писать $A \trianglelefteq B$ (либо $B \trianglerighteq A$). Если одновременно выполнены условия $A \trianglelefteq B$ и $A \trianglerighteq B$, мы будем писать $A \trianglelefteq\!\!\trianglerighteq B$.

Сначала мы определим специальную функцию для операции покрытия, которую будем обозначать $\underset{A-B}{\Phi}(u)$; отметим, что аргументов у функции три (два языка $A, B \subseteq \Sigma^*$ и слово $u \in \Sigma^*$), а результатом также является язык из $A, B \subseteq \Sigma^*$. Итак, определяемый язык является следующим:

$$\underset{A-B}{\Phi}(u) = \big\{ z \in \mathrm{opref}(B) \,\big|\, (\exists \tilde{b} \in B^+,\ w \in A)\ (wu = w\bar{u}z,\ \tilde{b} = w\bar{u}) \big\}.$$

Далее расширим приведённое определение для трёх аргументов-языков $A, B, C \subseteq \Sigma^*$:

$$\underset{A-B}{\Phi}(C) = \bigcup_{u \in C} \underset{A-B}{\Phi}(u).$$

В случае использования этой функции для группоидной операции более удобна другая запись:

$$\underset{A-B}{\Phi}(C) = \Phi_B(C, A).$$

Среди нескольких различных возможных вариантов конечных автоматов, выполняющих проверку отношения покрытия (в работах авторов [9], [10] было описано 8 таких вариантов) нам для дальнейших построений наиболее удобен следующий. Детерминированный конечный автомат $\mathrm{PRI}(A, B)$ для заданных конечных языков $A, B \subseteq \Sigma^*$ определяется следующим образом:

$$\overline{\mathrm{PRI}}(A, B) = (\overline{Q}_B, \Delta_A, \delta_{A-B}, \{\{e\}\}, \{\varnothing\}),$$

где:

- $\Delta_A = \{0, 1, ..., n-1\}$, здесь n – число слов языка A;
- $\overline{Q}_B = \mathcal{P}(\mathrm{opref}(B))$;
- для некоторых $q_1, q_2 \in Q_B$ и $a \in \Delta_A$ полагаем
$$q_1 \xrightarrow[\delta_{A-B}]{a} q_2$$
тогда и только тогда, когда
$$\underset{h_A(a)-B}{\Phi}(q_1) = q_2\,.$$

Автомат, получающийся после удаления недостижимых состояний из автомата $\overline{\mathrm{PRI}}(A, B)$, будем обозначать

$$\mathrm{PRI}(A, B) = (Q_B, \Delta_A, \delta_{A-B}, \{\{e\}\}, \{\varnothing\}),$$

где $Q_B \subseteq \mathcal{P}(\mathrm{opref}(B))$.

Определим также связанный с ним группоид:

$$\mathrm{Pri}(B) = \big(\mathcal{P}(\mathrm{opref}(B)), \Phi_B\big).$$

Одновременно можно ввести полугруппу соответствующих ему преобразований

$$\mathrm{Pri}(A, B) = (\mathcal{I}_r(Q_B), \circ),$$

где:

$$\big(\forall\, q \in Q_B\big)\ \big(\forall\, i \in \overline{0, |A|-1}\big)\ \big(\exists\, \phi_{h_A(i)} \in \mathcal{I}_r(Q_B)\big)$$
$$\big(i \xrightarrow{h_A} a_i \xrightarrow{\tau_{\mathfrak{A}}} \phi_{h_A(i)},\ q\phi_{h_A(i)} = \delta_{A-B}(q, a_i) \in Q_B\big).$$

Некоторые другие алгебраические свойства получаемого группоида, а также некоторые его специальные элементы мы более подробно описывли в [1].

Также в примерах, рассмотренных в [1], [9], [10], мы видели, что группоид и полугруппа, соответстующие даже «не очень сложному» языку, нередко имеют сложную для последующего анализа структуру. Интуитивной оценкой мощности множества группоида является экспонента числа слов рассматриваемого языка B, то есть

$$\big|\mathcal{P}(\mathrm{opref}(B))\big| \leqslant 1 + 2^{\sum_{u \in B} |u| - 1},$$

где $|u|$ – длина слова u. В таких ситуациях исследовать структуру группоида приходится с помощью специальных компьютерных программ. Однако при этом надо учитывать, что оценка сложности всех рассматриваемых алгоритмов работы с группоидом зависит не только от числа слов входного языка и их суммарной длины, но и от самого представления входных данных – и такое возможное представление мы рассмотрим далее.

В заключение раздела отметим, что рассматриваемая далее программа, среди прочего, описывает представленный выше алгоритм формирования функции Φ, фактически являющийся определением этой функции[2]. Более

[2] То есть приведённое выше определение функции Φ мы считаем *конструктивным*. При этом, поскольку определение является алгоритмом, мы не исключаем существования алгоритмов более эффективных.

того, мы определяли эту функцию именно для ответа на вопрос о покрытии одного заданного конечного языка другим. На основе этой функции мы в [1] построили функцию переходов автомата PRI и операцию группоида Pri – что и является одной из целей рассматриваемой далее компьютерной программы. Вариант полного описания алгоритма для ответа на вопрос о покрытияи языков выбран на основе материала статей [7], [8]; при дальнейшем описании программы мы покажем, как работает этот алгоритм.

III. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ: ОБЩАЯ СТРУКТУРА

Сначала кратко представим *общую структуру* описываемой компьютерной программы. Выбор в качестве языка реализации Си++ был обусловлен необходимостью получения алгоритмов, работающих для как можно бóльших размерностей задачи, – более конкретно тем, что при линейном увеличении размерности задачи (иными словами – при линейном увеличении суммарной длины слов входного языка) может происходить экспоненциальное увеличение числа слов получаемого группоида.

В первую очередь приведём наш вариант реализации алфавита и языка. Можно было бы хранить как алфавит, так и язык в массиве или в последовательном контейнере – но мы решили инкапсулировать их в классы и использовать ассоциативный контейнер, [17] и др.; наш вариант представления этих данных показан на рис. 1.

Рис. 1. Классы для алфавита и языка.

Для алгоритма создания группоида для некоторого рассматриваемого языка B при использовании определениий, приведённых в [1], нужен алгоритм получения подмножеств: для исходного множества, состоящего из n элементов, каждое его подмножество может соответствовать одному двоичному набору длины n. Алгоритм генерации подмножеств (например, алгоритмом Грэя[3]) заключается в генерации всех двоичные наборов и поэтому не может иметь временну́ю сложность менее $O(2^n)$.

Таблицу для представления автомата мы используем аналогично [1]. Класс для автомата мы не инкапсулировали – потому что используем его только как базовое представление группоида (полугруппы). Отметим по этому поводу, что в случае реализации инкапсуляции:

- во-первых, мы, построив автомат, не можем в дальнейшем изменять его внутреннюю структуру в процессе дальнейшей работы программы (например, для его исследования, его возможного эквивалентного преобразования и пр.) – что нужно в рассматриваемых нами задачах;
- во-вторых – «обратный аргумент»: в рассматриваемой задаче нам не надо программировать переходы из одного состояния в другое – что, конечно, может являться предметом других связанных с конечными автоматами практических задач[4].

Поэтому ни одной возможной причины для реализации языка и/или алфавита с применением инкапсуляции мы здесь не видим.

Отметим также, что на основе теоретической части [1] мы сразу можем видеть, что получаемый нами алгебраический объект является по крайней мере группоидом – то есть мы не обязаны проверять замкнутость. Однако функцию проверки замкнутости для операции мы реализовали, её можно применять, например, для отладки программ – как описываемой, так и возможных в будущем.

Для проверки других свойств группоида и вычисления нужных элементов мы описали и реализовали функции как для группоида (т. е. для одного языка), так и для полугруппы преобразований (т. е. для двух языков). В следующих разделах мы будем в первую очередь рассматривать версии функций для группоида. На наш взгляд, именно вследствие достаточно удачного представление рассматриваемых нами объектов функция проверки свойств и вычисления нужных элементов группоида описываются чётко и лаконично.

Однако при создании таблицы для бинарного отношения, в точности повторяющей приведённое в [1] описание, мы получаем весьма громоздкий объект, такая таблица занимает очень большую память, она трудно анализируема. Поэтому мы используем сокращённый вариант для такой таблицы, фактически являющийся перечислением всех пар элементов и получаемыми для каждой пары результатами группоидной операции; подробности см. далее.

Итак, в следующих разделах показываются различные детали компьютерной программы – связанные с построением рассматриваемого группоида и его исследованием. Также будут рассмотрены реализованные функции,

[3] Принятое в русском языке название «код Грея» – а именно под таким названием в настоящее время имеется статья в Википедии, таким же образом названы соответствующие разделы в различных книгах «про алгоритмы» – конечно же, «вдвойне неудачно»: во-первых, это, конечно, не код, а алгоритм; во-вторых, фамилии вроде “Gray” всегда было принято писать по-русски как «Грэй».

[4] По-видимому, все задачи, относящиеся к конечным автоматам (причём не обязательно детерминированным), можно разделить на два класса:

– задачи, в которых автомат описывает алгоритм для принятия некоторого слова заданного языка;
– задачи, в которых автомат является «более абстрактным» объектом, и мы просто знаем некоторые возможные операции работы с ними (например, алгоритмы эквивалентного преобразования, алгоритмы проверки эквивалентности и др.), причём эти алгоритмы «не опускаются до уровня» исследования входных слов языка автомата.

Такое деление на два класса относится как к чисто теоретическим, так и к «программистским» задачам.

основные и вспомогательные, удобные для проведения различных таких исследований: функция для записи результатов в файл и функции, функция создания группоида, функции проверки свойств и вычисления нужных элементов для заданного языка, в частности – нахождения всех идемпотентов получаемой полугруппы, и др. Отметим использование `vector`, `pair` и `map`, но никаких других специальных библиотек (связанных именно с алгебраическими объектами и автоматами) мы не используем.

IV. РЕАЛИЗАНИЯ ОСНОВНЫХ ОБЪЕКТОВ (ЯЗЫКОВ И МЕТОДОВ РАБОТЫ С НИМИ) ДЛЯ ПОСТРОЕНИЯ ГРУППОИДА

В этом разделе мы приводим описание основной части программы – то есть то, как мы будем представлять алфавит, языки, автоматы и получаемый группоид (полугруппу). При этом сложные алгоритмы использоваться не будут, и, кроме того, мы будем приводить некоторые пояснения, связанные с эффективностью применённых алгоритмов.

Как мы уже отмечали в разделе III, мы описываем классы для алфавита и языка, причём в класс для языка входит указатель на соответствующщий ему алфавит (рис. 2 и 3).

```
class alfavit
{
protected:
    std::map<char, bool> sigma;
    int kol;
public:
    alfavit() ···
    alfavit(string s) ···
    int addBukva(char c) ···
    int addBukvy(string s) ···
    int getKol() ···
    std::map<char, bool> getSigma() ···
    string to_string_sigma() ···
    bool ask_wordInSigmaStar(string s) ···
    bool compareAlf(alfavit* abc) ···
};
```

Рис. 2. Описание класса для алфавита.

```
class jazyk
{
    int count;
    alfavit* azbuka;
    std::map<string, int> slova;
public:
    jazyk() ···
    jazyk(alfavit* a) ···
    jazyk(std::map<string, int> words) ···
    int getCount() ···
    alfavit* getAzbuka() ···
    map<string, int> getSlova() ···
    vector<string> getVecStr_withClean() ···
    int cleanBadWords(bool countCheck) ···
    int addWord_(string s) ···
    int addWord(string s, bool alfavitCheck) ···
    int deleteWord(int index) ···
    string getWord_afterCleaning(int index) ···
    string to_string_jazyk() ···
    bool ask_wordInLang(string s) ···
    bool compareJazyk(jazyk* q) ···
};
```

Рис. 3. Описание класса для языка.

К приведённому тексту программы дадим следующие замечания.

- Для хранения букв и слов используется контейнер `map`, к нему добавлены счёчики `kol` и `count`. Также, как мы уже отмечали, в классе для языка содержится ещё и указатель на алфавит `slova`.
- Контейнер `map` вида «ключ–значение», т. е. «`int, string`», соотносит каждое слово с числом. Фактически это – простейшая версия морфизма.
- Методы добавления, удаления и получения слов тесно связаны с выполняемыми исследованиями группоида: например, сразу понятна роль трёх этих методов для описания алгоритма проверки выполнения отношения покрытия.
- Очень важны методы для сравнения алфавитов и языков – по причине наличия сравнения для проверки ассоциативности.
- Реализованы также и другие методы, выполненные, в первую очередь, для отладками программы. Смысл некоторых из них понятен на основе приведённых в тексте программы названий, и бóльшие подробности мы приводить не будем.

Причина использования контейнера `map` из стандартной шаблонной библиотеки проста: при реализации этого контейнера используется сбалансированное двоичное дерево поиска. Мы учитываем то, что в процессе исследования группоида очень часто происходит поиск букв и слов, и поэтому, несмотря на небольшие размеры алфавитов и языков, мы должны выбрать вариант контейнера с возможностью быстрого поиска. А при большом числе элементов быстро выполнять их вставку, удаление и поиск позволяет контейнер `map` – поэтому мы выбираем именно его.

Также отметим, что для получаемого автомата требуется, вообще говоря, большая память[5] – что можно объяснить следующим образом. При необходимости рассмотрения всех подмножеств множества собственных префиксов языка число состояний имеет экспонентно большое значение – относительно мощности множества префиксов языка, т. е. $2^{|\mathrm{opref}(B)|}$. Более того, отметим, немного забегая вперёд, что в конкретных практических задачах, связанных с построением именно автомата PRI, удаление недостижимых состояний обычно не очень сильно уменьшает их общее число.

Табличное предствление автомата храняется в массиве. Каждая его строка соответствует состоянию, каждый столбец – букве-входу. В каждой клетке таблицы написано слово, являющееся пометкой состояния конца дуги – для конкретного варианта морфизма, для которого строится автомат. Поэтому такая таблица автомата даёт результаты операции между любыми двумя элементами группоида.

В заключение раздела упомянем алгоритм, приведённый на рис. 4, который даёт генерацию подмножеств множества. Этот алгоритм используется при построении автомата PRI и группоида Pri для конкретного варианта генерации подмножеств множества префиксов языка $\mathrm{opref}(B)$ – см. рис. 5 и 6.

[5] Говоря неформально, существенно бóльшая, чем можно было бы ожидать.

```
vector<int*> myCmbntnN_noRe(unsigned int all, unsigned int choose) {
    vector<int*> res;
    if (all <= 0 || choose <= 0 || choose > all) return res;
    int* hold = new int[choose];
    for (int i = 0; i < choose; i++) hold[i] = i;
    while (hold[0] != all - choose) {
        if (hold[choose - 1] >= all) {
            int tmpflag = choose - 1;
            while (hold[tmpflag] - hold[tmpflag - 1] == 1) tmpflag--;
            hold[tmpflag - 1]++;
            for (int i = tmpflag; i < choose; i++)
                hold[i] = hold[i - 1] + 1;
            continue;
        }
        int* tmp = new int[choose];
        for (int i = 0; i < choose; i++) tmp[i] = hold[i];
        res.push_back(tmp); hold[choose - 1]++; }
    res.push_back(hold); return res; }
vector<pair<int, vector<int*>>> mySubsetNum(unsigned int all)
{
    //every element=<how many, combinations are>
    vector<pair<int, vector<int*>>> res;
    for (int i = 0; i <= all; i++) {
        res.push_back(make_pair(i, myCmbntnN_noRe(all, i))); }
    return res;
}
```

Рис. 4. Генерация подмножеств.

```
jazyk* getOPrefixLang(jazyk* lang)
{
    jazyk* pref = new jazyk(lang->getAzbuka());
    pref->addWord_("");
    lang->cleanBadWords(false);
    for (int i = 1; i <= lang->getCount(); i++)
    {
        string t = lang->getWord_afterCleaning(i);
        for (int j = 1; j < t.length(); j++)
            pref->addWord_(t.substr(0, j));
    }
    pref->cleanBadWords(false);
    return pref;
}
```

Рис. 5. Построение языка собственных префиксов.

```
pair< vector<pair<int, vector<int*>>>,
    vector<jazyk*>> getAllSublang(jazyk* lan) {
    /*<vector of all model_subset_numbers,
        vector of sublanguages>*/
    vector<jazyk*> res; auto abc = lan->getAzbuka();
    jazyk* nullLang = new jazyk(abc);
    res.push_back(nullLang);
    auto subsetModel = mySubsetNum(lan->getCount());
    if (lan->getCount() < 1)
        return make_pair(subsetModel, res);
    lan->cleanBadWords(false);
    auto vecIntStr = lan->getVecStr_withClean();
    for (auto& e : subsetModel) {
        if (e.first == lan->getCount())
            res.push_back(lan);
        else for (auto& ee : e.second) {
            jazyk* tmplan = new jazyk(abc);
            for (int i = 0; i < e.first; i++)
                tmplan->addWord_(vecIntStr[ee[i]]);
            tmplan->cleanBadWords(false);
            res.push_back(tmplan);
        }
    }
    return make_pair(subsetModel, res);
}
```

Рис. 6. Построение подмножеств множества собственных префиксов.

Получаемые подмножества мы упорядочиваем следующим образом: пустое множество $\varnothing$, далее множества с 1 элементом, потом с 2, с 3, и так далее. Множества с одинаковым количеством элементов упорядочиваем согласно лексикографическому порядку. Алгоритм генерации самих подмножеств строится на основе генерации сочетаний: для множества, состоящего из n элементов, сочетания вычисляется $n+1$ раз. Таким образом, всего получается, как и ожидалось,

$$\sum_{k=0}^{n} C_n^k = 2^n$$

шагов.

Цикл для получения всех слов множества $\mathrm{opref}(u)$, где $u \in B$, начинается со второго символа ($j = 1$); при этом, согласно определениям из [1], пустое слово ε также включается в рассматриваемое множенство. Результатом функции, приведённой на рис. 6, является язык $\mathcal{P}(\mathrm{opref}(B))$.

На основе рассмотренных выше объектов и алгоритмов в следующем разделе приводятся функции обработки этих объектов.

V. РЕАЛИЗАНИЯ АЛГОРИТМОВ ОПИСАНИЯ ГРУППОИДА

В этом разделе представлены алгоритмы создания алфавита, языка и подмножеств собственных префиксов. Также показана реализация функций, необходимых для операций покрытия языков и создания первичного автомата PRI.

В [1], [4] была определена операция покрытия. Здесь мы кратко опишем соответствующий алгоритм – в виде, удобном для реализации на компьютере.

Алгоритм 1 (вспомогательный алгоритм для проверки выполнения отношения покрытия).

1) Для каждого слова заданного языка u создаются все его конкатенации со словами языка A; они включаются в язык D_{help}.
2) Все слова D_{help}, являющиеся собственным префиксом слова языка B, включаются в результат-язык D_{res}.
3) Для всех слов D_{help}, содержащих слово языка B (т. е. если начало покрыто словом языка B), их «непокрытые» части остаются в D_{help}; «покрытые» части и другие слова удаляются.
4) Повторяются шаги 2 и 3 – до тех пор, пока язык D_{help} не становится пустым. □

Программная реализация алгоритма покрытия Φ показана на рис. 7. Построение функций выполнено согласно описанному алгоритму. Функция не изменяет исходные языки, которые передаются по ссылке, при этом язык-результат создаётся заново, а вспомагательные языки удаляются.

Оценим сложность функции $\underset{A-B}{\Phi}(u)$ для создания группоида. Сразу заметим, что слова языка A и слово u являются собственными префиксами слова языка B. Пусть

$$n = |\mathrm{opref}(B)|, \quad m = \max_{w \in \mathrm{opref}(B)} |w|;$$

```
jazyk* myPhi_AminusB_(jazyk* la, jazyk* lb, string u)
{
    jazyk* bpab_Dres = new jazyk(lb->getAzbuka());
    jazyk* bpab_Dhelp = new jazyk(lb->getAzbuka());
    la->cleanBadWords(false); lb->cleanBadWords(false);
    jazyk* oprefB = getOPrefixLang(lb);
    for (int i = 1; i <= la->getCount(); i++)
        bpab_Dhelp->addWord_(
            u + la->getWord_afterCleaning(i));
    while (1) {
        bpab_Dhelp->cleanBadWords(false);
        if (bpab_Dhelp->getCount() == 0) break;
        string v = bpab_Dhelp->getWord_afterCleaning(1);
        bpab_Dhelp->deleteWord(1);
        for (int i = 1; i <= lb->getCount(); i++) {
            string tmp_u = lb->getWord_afterCleaning(i);
            if (v.substr(0, tmp_u.length()) == tmp_u)
                bpab_Dhelp->addWord(v.substr(
                    tmp_u.length(), v.length()), true); }
        if (oprefB->ask_wordInLang(v))
            bpab_Dres->addWord(v, true);
    }
    delete oprefB; delete bpab_Dhelp;
    bpab_Dres->cleanBadWords(false);
    return bpab_Dres;
}
jazyk* myPhi_AminusB_(jazyk* la, jazyk* lb, jazyk* lc)
{
    jazyk* res = new jazyk(lb->getAzbuka());
    lc->cleanBadWords(false);
    for (int i = 1; i <= lc->getCount(); i++) {
        auto tmp_q = myPhi_AminusB_(
            la, lb, lc->getWord_afterCleaning(i));
        tmp_q->cleanBadWords(false);
        for (int j = 1; j <= tmp_q->getCount(); j++)
            res->addWord_(
                tmp_q->getWord_afterCleaning(j));
        delete tmp_q;
    }
    return res;
}
```

Рис. 7. Функции построения языков $\underset{A-B}{\Phi}(u)$ и $\underset{A-B}{\Phi}(C)$.

тогда

$$\max_{w \in A} |w| \leqslant m, \quad |u| \leqslant m, \quad |B| \simeq O(n), \quad |A| \leqslant n.$$

Поэтому в общем случае число вычислений имеет оценку $O(mn^2)$, т. е. алгоритм полиномиален относительно размера входных данных.

Легко видеть связь между $\Phi_B(X, Y)$ и $\underset{Y-B}{\Phi}(X)$ – см. функцию, приведённую на рис. 8; можно считать, что это – функция для операции группоида.

```
jazyk* operationSemigroup_F_(jazyk* base, jazyk* x,
    jazyk* y) { return myPhi_AminusB_(y, base, x); }
```

Рис. 8. Построение функции $\Phi_B(X, Y)$.

Теперь перейдём к построению группоида Pri (который мы иногда рассматриваем как множество автоматов) – соответствующая программа приведена на рис. 9. Функция построения автомата PRI (от двух языков) принимает язык и возвращает таблицу автомата и множество состояний. А при построении группоида Pri (от одного языка) в цикл также включается вспомогательная функция очистки памяти – понятно, что при построении автомата PRI подобные проблемы с динамической памятью не проявлялись.

```
pair<pair<jazyk*, vector<jazyk*>>, pair<int, int*>>
    tableSemiGroup_allSubsetOprefBaseLanguage_opsg
    (jazyk* bl) {
    jazyk* oprfx_bl = getOPrefixLang(bl);
    auto sublanOprfxBl_pair = getAllSublang(oprfx_bl);
    vector<jazyk*> sublanOprfxBl =
        sublanOprfxBl_pair.second;
    vector<pair<int, vector<int*>>> subsetmodel =
        sublanOprfxBl_pair.first;
    int razmer = sublanOprfxBl.size();
    int* tableSG = new int[razmer * razmer]();
    cout << razmer << endl;//loading bar
    for (int i = 0; i < razmer; i++) {
        for (int j = 0; j < razmer; j++) {
            tableSG[i * razmer + j] = -1;
            jazyk* afterOperation = operationSemigroup_F_
                (bl, sublanOprfxBl[i], sublanOprfxBl[j]);
            afterOperation->cleanBadWords(false);
            int tmprank = 0, tmpcnt = 0,
                aftopsize = afterOperation->getCount();
            for (tmpcnt = 0; tmpcnt < subsetmodel.size();
                tmpcnt++) {
                if (subsetmodel[tmpcnt].first ==
                    aftopsize) break;
                tmprank +=
                    subsetmodel[tmpcnt].second.size(); }
            if (tmpcnt == subsetmodel.size()) continue;
            else for (int k = tmprank;
                k <= subsetmodel[tmpcnt].second.size()
                    + tmprank;
                k++) {
                if (sublanOprfxBl[k]->compareJazyk(
                    afterOperation)) {
                    tableSG[i * razmer + j] = k; break; }
                } delete afterOperation; }
        if (i % 10 == 0) cout << i << " ";//loading bar
    } cout << endl;//loading bar
    return make_pair(make_pair(bl, sublanOprfxBl),
        make_pair(razmer, tableSG));
}
```

Рис. 9. Автомат Pri(B).

На рис. 10 представлена интуитивная структура табличного представления автомата PRI.

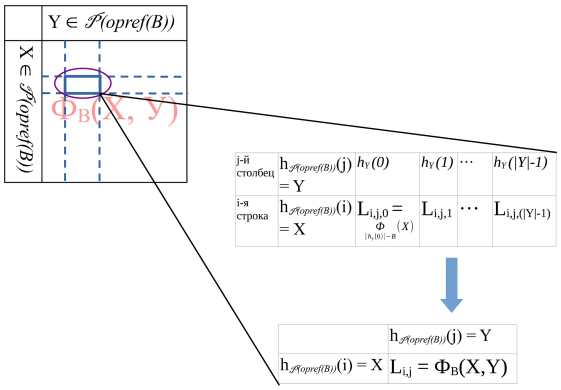

Рис. 10. Табличное представление автомата Pri(B).

Таким образом, при построении автомата Pri каждый раз (т. е. при обработке каждого элемента формируемой группоидной операции) мы получаем некоторый язык; но мы не храним в таблице ни строкового представления, ни

указателя на этот язык – а храним номер двоичного набора по описанному алгоритму генерации подмножеств, т. е., иными словами, просто одну букву для исходного алфавита рассматриваемого морфизма.

Для представления соответствующего языка мы сначала выполняем поиск по количеству слов в языке, а затем среди найденных слов выполняем «точный» поиск. Порядок подмножеств-языков определяется согласно описанному ранее алгоритму их генерации и содержит априорную информацию для определения «места» (номера) такого языка – по нашему мнению, такой подход значительно ускоряет работу программы. Конкретно, пусть $\left|\mathrm{opref}(B)\right| = n$, тогда:

- обычный поиск выпоняется за $O\left(C_n^{[n/2]}\right)$ временны́х единиц,
- а если мы применяем дихотомию (с учётом порядка двоичных наборов и, следовательно, подмножеств-языков), то получается существенное временно́е улучшение, а именно, получаем $O\left(\log C_n^{[n/2]}\right)$ единиц.

Итак, таблица группоида получена. В следующем разделе мы приводим описание функций проверки его свойств и вычисления нужных элементов по полученной таблице.

VI. ОПИСАНИЙ ФУНКЦИЙ, РЕАЛИЗОВАННЫХ ДЛЯ ИССЛЕДОВАНИЯ СВОЙСТВ ГРУППОИДА

Авторами ранее, в статье [1], были описаны некоторые алгебраические свойства и характерные элементы группоида автомата PRI и, в частных случаях, соответствующей полугруппы. В данном разделе приведены описания функций языка Си++, специально реализованных для исследования таких свойств. На основе этих описаний легко оценить временну́ю сложность основных алгоритмов, необходимых для построения полугруппы автомата PRI.

Заметим, что все определения мало различаются для группоида и полугруппы преобразований – поэтому мы приводим только все варианты функций для группоида и самые основные варианты для полугруппы преобразований; по-видимому, для полного понимания всей программы этого достаточно.

Сначала рассмотрим самые основные функции для группоида – функции проверки замкнутости (рис. 11) и ассоциативности (рис. 12)[6]. Как уже было отмечено, эти функции использовались, в первую очередь, в процессе отладки программ – но они могут быть также применены, например, в случае ввода из файла некоторого объекта – предполагаемого группоида.

Функция на рис. 11 проверяет, правильно ли был построен конечный автомат PRI, т. е. действительно ли образовался группоид. Согласно алгоритму построения автомата PRI (раздел V и др.), особое значение -1, возвращаемое этой функцией проверки, говорит о том,

```
vector<int*> testZamknutost(pair<int, int*> ppp)
{
    vector<int*> er;
    int rzmr = ppp.first;
    int* table = ppp.second;
    for (int i = 0; i < rzmr; i++)
        for (int j = 0; j < rzmr; j++)
            if (
                table[i * rzmr + j] == -1
                )
                er.push_back(new int[2] {i, j});
    return er;
}
```

Рис. 11. Функция проверки замкнутости.

что полученный результат не входит в ожидаемое множество, т. е., заданная структура не является замкнутой для рассматриваемой группоидной операции.

После проверки замкнутости можно проверить, обладает ли построенный группоид свойством ассоциативности – см. рис. 12. Если действительно получается ассоциативный группоид, то кроме полугруппы преобразований (которая имеется всегда) мы получаем ещё и полугруппу Pri одного языка, относящуюся к соответствующему автомату PRI[7].

```
vector<int*> testAssociativ(pair<int, int*> ppp)
{
    vector<int*> er;
    int rzmr = ppp.first;
    int* table = ppp.second;
    for (int i = 0; i < rzmr; i++)
        for (int j = 0; j < rzmr; j++)
            for (int k = 0; k < rzmr; k++)
                if (
                    table[i * rzmr + table[j * rzmr + k]]
                    != table[table[i * rzmr + j] * rzmr + k]
                    )
                    er.push_back(new int[3] {i, j, k});
    return er;
}
```

Рис. 12. Функция проверки ассоциативности.

Далее введём функции вычисления нужных элементов. Эти функции, очевидно, должны использоваться только на полугруппах, т.е. ассоциативность должна быть проверена заранее[8].

```
vector<int> idempotent(pair<int, int*> table)
{
    vector<int> res;
    for (int i = 0; i < table.first; i++)
        if (table.second[i * table.first + i] == i)
            res.push_back(i);
    return res;
}
```

Рис. 13. Функция вычисления идемпотентов.

Простая функция вычисления множества идемпотентов (рис. 13) ищет на диагонали таблицы те элементы,

[6] Заметим ещё раз, что в результате рассмотрения любого автомата PRI полугруппа *преобразований* получается *всегда* – поэтому необязательно проверять корректность соответствующих построений. Бо́льшая сложность работы с полугруппой преобразований «в теории» показывается также и «для программы»: соответствующая функция проверки длиннее, она работает намного медленнее и требует существенно больше динамической памяти.

[7] Повторим по этому поводу, что именно на основе этих двух функций наиболее ярко проявляется удобство табличного представления автомата PRI и группоида Pri: в этом случае указанные две функции работают очень быстро даже в случае больших входных данных.

[8] При этом и для группоидов эти функции работают и выдают какие-то результаты, но такие результаты, по-видимому, бессмысленны.

которые при применении полугрупповой операции дают сами себя. Понятно, что эта функция работает достаточно быстро.

Исходя из конечности полугруппы, в любом ряду степеней элементов обязательно встречается элемент повторяющий, т. е. при рассмотрении степенного ряда невозможны ситуации ни с получением бесконечного индекса, ни с получением периода; см. рис. 14.

```
pair<pair<int, int>, vector<int>>
    rqdProizv(pair<int, int*> t, int elem) {
    vector<int> rqd; rqd.push_back(elem);
    while (1) {
        int tmp = t.second[
            rqd[rqd.size() - 1] * t.first + elem];
        for (int i = 0; i < rqd.size(); i++)
            if (rqd[i] == tmp) {
                rqd.push_back(tmp);
                return make_pair(make_pair(
                    i + 1, rqd.size() - i - 1), rqd);
            }
        rqd.push_back(tmp);
    }
    //return < <deg,per>, series >
    return make_pair(make_pair(-1, -1), rqd);
}
vector<pair<pair<int, int>, vector<int>>>
    rqdProizv_full(pair<int, int*> t) {
    vector<pair<pair<int, int>, vector<int>>> res;
    for (int i = 0; i < t.first; i++)
        res.push_back(rqdProizv(t, i));
    return res;
}
```

Рис. 14. Функция вычисления рядов степеней элементов.

Логически более сложная функция, приведённая на рим. 15, вычисляет наименьшую полугруппу для любого подмножества множества элементов рассматриваемой полугруппы. В этой функции для ускроения поиска элемента используется контейнер `set`.

```
vector<int> naimenPodPolugruppa(
    pair<int, int*> t, vector<int> birth) {
    bool flag = true; set<int> rem;
    for (auto e : birth) rem.insert(e);
    while (flag) {
        vector<int> newElem; flag = false;
        for (int i = 0; i < birth.size(); i++)
            for (int j = 0; j < birth.size(); j++) {
                int tmp = t.second[
                    birth[i] * t.first + birth[j]];
                if (rem.find(tmp) == rem.end()) {
                    flag = true; rem.insert(tmp);
                    newElem.push_back(tmp);     }  }
        for (int e : newElem) birth.push_back(e);
        vector<int>().swap(newElem);
    }
    vector<int> res; for (auto e : rem) res.push_back(e);
    return res;
}
```

Рис. 15. Функция вычисления наименьшей подполугруппы.

Функция, приведённая на рис. 16, проверяет, существует ли для группоида бинарное отношение определяемое согласно, например, [1] для пар состояний.

Мы применяем эту функцию, в частности, к полному набору всех возможных пар – получая тем самым таблицу бинарного отношения, см. рис. 17. Мы видим, что такая таблица состоит из

$$C^2_{|Q_B|} = C^2_{\mathcal{P}(\mathrm{opref}(B))}$$

```
bool  binarOtnow_ourDef(pair<int, int*> p,
    int q1, int q2, int q3, int q4, int a)
//must check size boundary in advance
{
    return (
        (p.second[q1 * p.first + a] == q3)
        &&
        (p.second[q2 * p.first + a] == q4)
        );
}
bool  binarOtnow_ourDef(pair<int, int*> p,
    int q1, int q2, int q3, int q4)
{
    for (int i = 0; i < p.first; i++)
        if (binarOtnow_ourDef(p, q1, q2, q3, q4, i))
            return true;
    return false;
}
```

Рис. 16. Функция определения существования бинарного отношения.

```
pair<vector<int*>, bool*>
    binarOtnow_ourDef_table(pair<int, int*> p) {
    auto cmbnNModel = myCmbntnN_noRe(p.first, 2);
    bool* table = new bool[
        cmbnNModel.size() * cmbnNModel.size()];
    for (int i = 0; i < cmbnNModel.size(); i++) {
        for (int j = 0; j < cmbnNModel.size(); j++)
            table[i * cmbnNModel.size() + j] =
                binarOtnow_ourDef(p,
                    cmbnNModel[i][0], cmbnNModel[i][1],
                    cmbnNModel[j][0], cmbnNModel[j][1]);
        if (i % 100 == 0) cout << i << " ";     }
    return make_pair(cmbnNModel, table);
}
string  binarOtnow_ourDef_table(
    pair<int, int*> p, string f)···
```

Рис. 17. Функция построения таблицы бинарного отношения.

строк (столбцов) – т. е. её размер очень большой; поэтому для некоторого облегчения всего исследования мы для класса – таблицы бинарного отношения создали также методы записи в текстовый файл и чтения из файла (в статье мы эти методы не приводим).

```
set<int> findAttainable_priBfull_set(pair<int, int*> t)
{
    set<int> s, att; s.insert(1); att.insert(1);
    int cnt = 1;
    while (cnt) {
        if (att.size() == t.first) break;
        set<int> tmp; cnt = 0;
        for (auto e : s)
            for (int i = 0; i < t.first; i++) {
                if (att.find(t.second[e * t.first + i])
                    == att.end())   {
                    att.insert(t.second[e * t.first + i]);
                    cnt++;          }
                tmp.insert(t.second[e * t.first + i]);
            }
        tmp.swap(s); set<int>().swap(tmp);
    }
    return att;
}
```

Рис. 18. Функция построения множества достижимых состояний.

При работе с автоматом PRI (подробное определение в [1] и др. [9]) мы должны были определять стартовое (и финальное) состояния. А теперь, т. е. для рассматриваемых

[9] Там же см. и информацию о различных версиях *недетерминированных* автоматов, обозначенных NSPRI, обычно с верхними и нижними индексами.

в настоящей статье задач, нас в первую очередь интересует изучение вспомогательных алгоритмов определения множества достижимых состояний; это несложно делается на основе упомянутой информации о стартовом состояния, см. программу на рис. 18.

```
vector<int> idempotent(pair<int, int*> table, set<int> att)
{
    vector<int> res;
    for (int i : att)
        if (table.second[i * table.first + i] == i)
            res.push_back(i);
    return res;
}
```

Рис. 19. Функция построения множества идемпотентов на множестве достижимых состояний.

Отметим, что наименьшая подполугруппа, порождаемая множеством достижимых состояний, всегда совпадает с исходной полугруппой. В случае любого входа, начиная со стартового состояния, можно достичь только этих состояний, поэтому при оформлении переходов как полугрупповой операции не может появиться новый элемент [10]. Согласно изложенному в двух предыдущих абзацах, функция построения множества идемпотентов на множестве достижимых состояний очень проста: см. рис. 19.

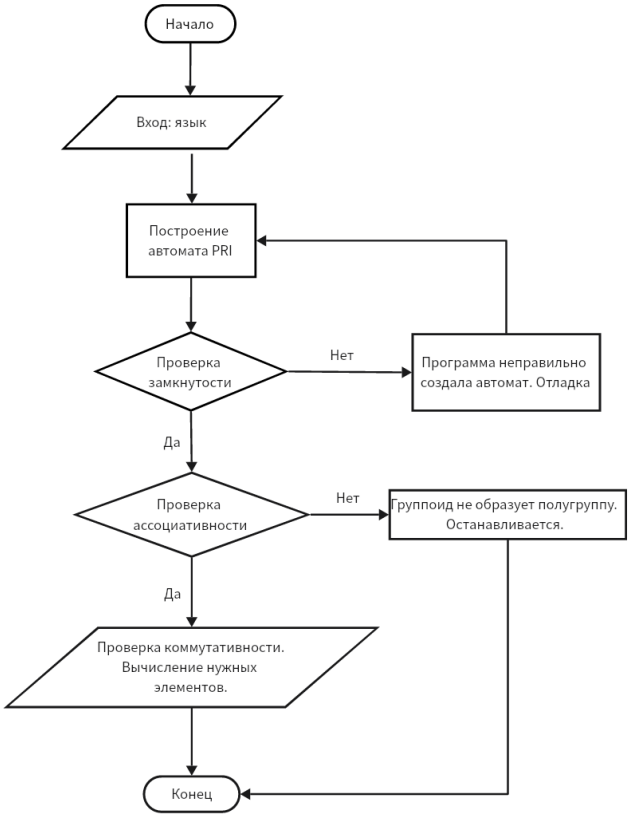

Рис. 20. Общая схема проводимого исследования.

В заключение раздела мы описываем весь процесс проводимого исследования, см. рис. 20 и описание алгоритма ниже [11]. При этом мы не используем промежуточные переменные – то есть одна функция записывает данные в текстовый файл, а другая читает их из этого файла.

Алгоритм 2 (общая схема исследования для одного примера).

1) Создаются алфавит и язык.
2) С помощью генератора чисел получается множество собственных префиксов исходного языка и его подмножеств.
3) Строится таблица автомата PRI.
4) По этой таблице проверяются замкнутость и ассоциативность.
5) В случае получения полугруппы – вычисляются:
   - идемпотенты,
   - ряды степеней элементов,
   - наименьшая подполугруппа заданного подмножества элементов
   - и таблица соотваетствующего бинарного отношения. □

В следующем разделе мы рассмотрим конкретные примеры, полученные на основе такого процесса исследования.

VII. О ВЫЧИСЛИТЕЛЬНЫХ ЭКСПЕРИМЕНТАХ И НЕКОТОРЫХ ПОЛУЧЕННЫХ РЕЗУЛЬТАТАХ

В этом разделе рассматриваются конкретные примеры построения и исследования получаемого группоида – либо соответствующей полугруппы. Мы приводим информацию, полученную как результат работы программы. Первый пример – наглядный, но несложный. Второй – более сложный и более содержательный, в [1] он также был назван примером 2.

Item	Object
Language	[2]{aa,aaab}
Oprefix	[4]{ε,a,aa,aaa}
All sublanguage	Runtime: 1 ms
Time for making table	Runtime: 83 ms
Bad pairs	0
Nonassociative triples	0
Identity elements	0
Commitativity test	Inverse semigroup
Idempotents	0,1,6,15
Series	Runtime: 4 ms
Minimum semigroup, Including {1,2}	{1,2,6,9}
Binary relation table	Runtime: 48 ms

Рис. 21. Результаты выдачи примера 1.

Пример 1. Процесс исследования проводится для языка

$$B = \{aa, aaab\}.$$

[10] Приведём по этому поводу одно из возможных определений *эргодичности* – без конкретных ссылок на само определение. Эргодичность – это специальное свойство некоторых динамических систем, состоящее в том, что в процессе эволюции «почти каждое» состояние с определённой вероятностью проходит вблизи любого другого состояния системы.

[11] Отметим по этому поводу, что в приведённых выше фрагментах программы мы используем промежуточные переменные «очень сложных» классов. Например, `pair<pair<jazyk*>>`, `pair<int, int*>`, `vector<jazyk*>` и др. Никаких дополнительных структур данных при этом не создаётся. Таким образом удобно передавать данные между функциями, а также сохранять возможность проверки элементов в процессе работы всей программы.

Во время работы программа кратко выводит следующую информацию, см. рис. 21.

При этом созданная программой таблица группоида сохраняется в отдельном файле, см. рис. 22.

			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			∅	{c}	{a}	{aa}	{aaa}	{c,a}	{c,aa}	{c,aaa}	{a,aa}	{a,aaa}	{aa,aaa}	{c,a,aa}	{c,a,aaa}	{c,aa,aaa}	{a,aa,aaa}	{c,a,aa,aaa}
←	0	∅	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
→	1	{c}	0	1	2	6	9	5	6	12	11	9	15	11	12	15	15	15
	2	{a}	0	2	6	9	6	11	9	11	15	6	15	15	11	15	15	15
	3	{aa}	0	6	9	6	9	15	6	15	15	9	15	15	15	15	15	15
	4	{aaa}	0	9	6	9	6	15	9	15	15	6	15	15	15	15	15	15
	5	{c,a}	0	5	11	15	15	11	15	15	15	15	15	15	15	15	15	15
	6	{c,aa}	0	6	9	6	9	15	6	15	15	9	15	15	15	15	15	15
	7	{c,aaa}	0	12	11	15	15	15	15	15	15	15	15	15	15	15	15	15
	8	{a,aa}	0	11	15	15	15	15	15	15	15	15	15	15	15	15	15	15
	9	{a,aaa}	0	9	6	9	6	15	9	15	15	6	15	15	15	15	15	15
	10	{aa,aaa}	0	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
	11	{c,a,aa}	0	11	15	15	15	15	15	15	15	15	15	15	15	15	15	15
	12	{c,a,aaa}	0	12	11	15	15	15	15	15	15	15	15	15	15	15	15	15
	13	{c,aa,aaa}	0	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
	14	{a,aa,aaa}	0	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
	15	{c,a,aa,aaa}	0	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15

Рис. 22. Таблица группоида $\mathrm{Pri}(\{aa, aaab\})$.

Мы видим, что автомат и группоид достаточно сложны, также очень сложен и соответствующий им граф – то есть граф в виде рисунка мы не приводим. В рассматриваемом примере полугруппа получается коммутативной. Она не имеет единицы, т. е. моноидом не является. Она содержит четыре идемпотента: 1, 6, 15 и тривиальный 0.

Далее вычисляем множество достижимых состояний; достижимы 9 из 16 состояний:

$$0,\ 1,\ 2,\ 5,\ 6,\ 9,\ 11,\ 12,\ 15.$$

Определяем наименьшую подполугруппу, порождённую множеством $\{1, 2\}$; получаем $\{1, 2, 6, 9\}$.

Далее на рис. 23 приводим результаты вычислений рядов степеней всех элементов полугруппы.

Element	Index	Period	Series
0	1	1	0,0
1	1	1	1,1
2	2	2	2,6,9,6
3	2	1	3,6,6
4	2	2	4,6,9,6
5	3	1	5,11,15,15
6	1	1	6,6
7	2	1	7,15,15
8	2	1	8,15,15
9	1	2	9,6,9
10	2	1	10,15,15
11	2	1	11,15,15
12	2	1	12,15,15
13	2	1	13,15,15
14	2	1	14,15,15
15	1	1	15,15

Рис. 23. Ряды степеней элементов.

Можно заметить, что большинство элементов дают одно и то же свойство относительно операции возведения в степень – а именно, у трёх элементов (2, 4 и 9) период равен 2, а у остальных равен 1. Соответственно, индекс элементов 2 и 4 равен 2, а элемента 9 равен 1.

В таблице бинарного отношения по столбцам и по строкам расположены все пары элементов в лексикографическом порядке [12]. Если пара элементов находится в бинарном отношении, то помечаем соответствующую ячейку, см. рис. 24.

[12] То есть $(0, 0)$, $(0, 1)$, …, $(0, 15)$, $(1, 1)$, $(1, 2)$, …, $(15, 15)$.

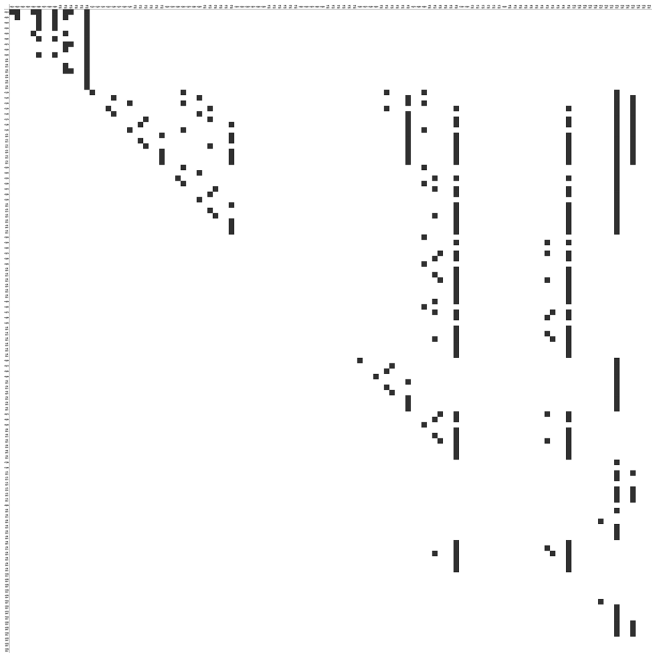

Рис. 24. Таблица полученного бинарного отношения.

Пример 2. Входной язык

$$B = \{aa, aaaaaaab\},$$

см. результаты выдачи основной части программы на рис. 25.

Item	Object
Language	[2]{aa,aaaaaaab}
Oprefix	[8]{ε,a,aa,aaa,aaaa,aaaaa, Aaaaaa,aaaaaaa}
All sublanguage	Runtime: 4 ms
Time for making table	Runtime: 96617 ms
Bad pairs	0
Nonassociative triples	0
Identity elements	1
Commitativity test	Inverse semigroup
Idempotents	0,1,113,255
Series	Runtime: 5 ms
Minimum subsemigroup, Including all attainable elements	{0,1,2,9,10,17,37,38,64,93,94, 99,113,142,163,164,168,177, 219,220,223,247,248,255}
Binary relation table	Runtime: 403 ms

Рис. 25. Результаты выдачи примера 2.

В предыдущих разделах мы оценивали временну́ю сложность некоторых функций представляемой программы. Для примера 2 мы видим, что время выполнения вычислений пропорционально

$$C_{256}^{2} = 32640,$$

и при этом вышеописанная программа на «обычном современном» компьютере работала около получаса.

Соответственно, очень большой размер имеет автомат $\mathrm{PRI}(\{aa, aaaaaaab\})$. Его достижимыми состояниями являются

$$0,\ 1,\ 2,\ 9,\ 17,\ 37,\ 38,\ 44,\ 64\ 93,\ 94,\ 99,\ 113,\ 142,$$
$$163,\ 164,\ 168,\ 177,\ 219,\ 220,\ 223,\ 247,\ 248,\ 255.$$

Здесь все элементы вычислялись на множестве достижимых состояний автомата.

Element	Index	Period	Series
0	1	1	0,0
1	1	1	1,1
2	6	2	2,10,17,44,64,113,142,113
9	7	1	9,37,93,163,219,247,255,255
10	3	1	10,44,113,113
17	2	2	17,113,142,113
37	4	1	37,163,247,255,255
38	3	1	38,220,255,255
44	2	1	44,113,113
64	2	2	64,113,142,113
93	3	1	93,247,255,255
94	3	1	94,247,255,255
99	3	1	99,247,255,255
113	1	1	113,113
142	1	2	142,113,142
163	2	1	163,255,255
164	2	1	164,255,255
168	2	1	168,255,255
177	2	1	177,255,255
219	2	1	219,255,255
220	2	1	220,255,255
223	2	1	223,255,255
247	2	1	247,255,255
248	2	1	248,255,255
255	1	1	255,255

Рис. 26. Рады степеней для достижимых элементов.

Конечно, даже для множества достижимых состояний мы не приводим таблицу (и тем более граф) получаемого автомата – кратко рассмотрим только соответствующий группоид. Как мы видим, получается коммутативный моноид с единицей 1, он содержит четыре идемпотента: 0, 1, 113, 255. Ряды степеней похожи между собой, причём мы видели подобный факт и для первого примера: индекс и период в обоих случаях небольшие. Особыми элементами в этом примере являются элементы 2 и 9, имеющие относительно большие индексы.

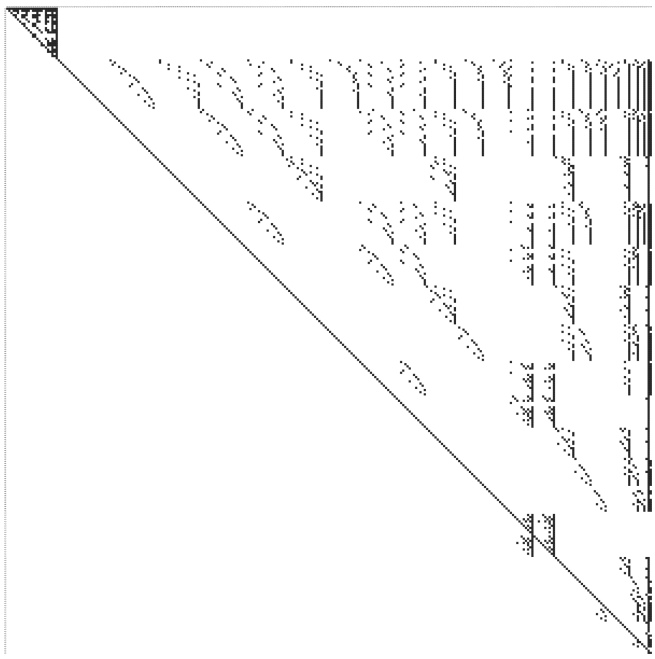

Рис. 27. Таблица бинарного отношения для достижимых состояний.

В приведённой таблице бинарного отношения (рис. 27) мы видим на множестве достижимых элементов «более кучный» график. Также на основе этой таблицы видно, что для пар пар вида $\big((a,b),\,(c,d)\big)$ выполняется следующий факт. Если мы

- либо фиксируем элементы a и c и изменяем b и d;
- либо «пропорционально» изменяем соответствующие элементы «в небольшой области», например, получаем

$$\big((a,b+1),\,(c,d+1)\big),\ \text{или}\ \big((a,b+2),\,(c,d+2)\big),$$

и т. п. [13] –

то бинарное отношение, как правило, не нарушается [14].

VIII. ЗАКЛЮЧЕНИЕ

Проведём анализ результатов вычислений и сделаем некоторые выводы.

В статье мы описали созданную программу, предназначенную для исследования группоида и полугруппы преобразований автомата PRI. Программа показала свою эффективность и при исследовании сложных примеров, в которых число элементов группоида измеряется несколькими десятками. Для изучения разных возникающих примеров группоида, который в важных частных случаях является полугруппой, мы реализовали соответствующие компьютерные программы и выполнили вычислительные эксперименты.

При этом основной целью создания программы и проведения вычислительных экспериментов является *попытка понимания внутренней структуры* получаемых группоидов (в частных случаях – полугрупп), а также попытка «объяснения природы» рассматривавшихся во многих предыдущих статьях специальных бинарных отношений (так называемых отношений покрываемости и эквивалентности в бесконечности), заданных на множествах конечных формальных языков.

Судя по полученным предварительным результатам, разработанная нами программа хорошо справляется с рассматриваемой исследовательской задачей. Она выполняет как самые элементарные (но громоздкие), так и достаточно сложные вычисления и представляет удобочитаемые результаты, предназначенные для анализа группоида и полугруппы автомата PRI. При этом мы не приводим тесты, относящиеся к соответствующим полугруппам преобразований: они также представляют интерес для изучения, но подобное изучение выходит за рамки настоящей статьи.

Итак, описываемая в статье программа показала свою эффективность при исследовании, в том числе, сложных примеров – в которых число элементов группоида измеряется несколькими десятками. Программа выдавала интересные для дальнейшей интерпретации результаты,

[13] Здесь записи $+1$ и $+2$ не означают выполнение арифметических операций: они означают переходы (1- и 2-кратные) к элементам-преемникам на множестве достижимых состояний – согласно лексикографическому порядку. Ср. подобные подходы и записи в языках Си и Си++.

[14] В качестве материала для будущих исследований можно рассмотреть *количественную характеристику*, описывающую вероятность таких нарушений для разных исходных языков B. Рассмотренные примеры показывают, что такая вероятность существенно меньше ожидавшейся.

и при этом время её выполнения обычно примерно совпадало с теоретически вычисленными значениями.

Основными выводами, которые получены в результате анализа проведённых вычислительных экспериментов, можно считать следующие.

Во-первых, очевидно, что любые непустые языки над одним и тем же однобуквенным алфавитом эквивалентны в бесконечности. Между условием получения полугруппы и эквивалентностью в бесконечности существует связь, которая показана именно на этих языках. Мы показываем, что для таких языков (над однобуквенном алфавитом) группоид всегда является полугруппой, причём последняя всегда коммутативна и является моноидом.

Во-вторых, при вычислении интересных нам характеров и фактор-представлений элементов полугруппы ряды степеней содержат информацию об идемпотентах – это пары, состоящие из индекса и соответствующего ему периода. Поэтому в процессе работы программы можно не находить отдельно идемподенты получаемых полугрупп, а просто подсчитать их число после вычислений рядов степеней.

В-третьих, заметим, в обоих рассмотренных выше примерах число идемпотентов равно 4. Кроме несложных вариантов $\varnothing$, $\{\varepsilon\}$ и множества всех собственных префиксов (т. н. “univers”), идемпотентом в обоих случаях является ещё какое-то «промежуточное» множество. Размерность примеров сильно различается – 16 и 256, но число идемпотентов совпадает. К моменту написания статьи мы пока не знаем, из-за чего это происходит, и обязательно ли выполняется этот факт для любого исходного языка.

В ближайшем будущем мы предлагаем изучить ещё больше свойств получаемого группоида и рассмотреть ещё более сложные примеры. Также с рассматриваемыми задачами связано возможное определение аналогичных свойств для группоида (в тех ситуациях, когда он не является полугруппой)[15] – прежде всего это нужно для общего понимания, как эти свойства связаны с возможным получением именно полугруппы.

Итак, в будущих работах мы предлагаем рассмотреть более сложные примеры, на которых собираемся изучить более сложные свойства группоида автомата PRI.

БЛАГОДАРНОСТИ

Работа второго автора была частично поддержана грантом научной программы китайских университетов “Higher Education Stability Support Program” (раздел “Shenzhen 2022 – Science, Technology and Innovation Commission of Shenzhen Municipality”) – 深圳市 2022 年高等院校稳定支持计划资助项目.

[15]Понятно, что здесь нужно ввести очень аккуратное определение: при отсутствии в группоиде ассоциативности возникает естественная проблема, заключающаяся в необходимости рассмотрения произведений, соответствующих разным перестановкам элементов-множителей.

Список литературы

[1] Мэн Л., Мельников Б. Ф. *О группоиде первичного автомата PRI, определяемом для одного конечного языка* // International Journal of Open Information Technologies. – 2023. – Vol. 11. No. 9. – P. 1–12.

[2] Мельников Б. Ф. *Описание специальных подмоноидов глобального надмоноида свободного моноида* // Известия высших учебных заведений. Математика. – 2004. – № 3. – С. 46–56.

[3] Melnikov B. *The equality condition for infinite catenations of two sets of finite words* // International Journal of Foundation of Computer Science. – 1993. – Vol. 4. No. 3. – P. 267–274.

[4] Мельников Б. Ф. *Варианты конечных автоматов, соответствующих бесконечным итерационным деревьям морфизмов. Часть I* // International Journal of Open Information Technologies. – 2021. – Vol. 9. No. 7. – P. 5–13.

[5] Мельников Б. Ф. *Варианты конечных автоматов, соответствующих бесконечным итерационным деревьям морфизмов. Часть II* // International Journal of Open Information Technologies. – 2021. – Vol. 9. No. 10. – P. 1–8.

[6] Melnikov B. *2ω-finite automata and sets of obstructions of their language* // The Korean Journal of Computational and Applied Mathematics (Journal of Applied Mathematics and Computing). – 2000. – Vol. 6. No. 3. – P. 565–574.

[7] Мельников Б. Ф., Мельникова А. А. *Бесконечные деревья в алгоритме проверки условия эквивалентности итераций конечных языков. Часть I* // International Journal of Open Information Technologies. – 2021. – Vol. 9. No. 4. – P. 1–11.

[8] Мельников Б. Ф., Мельникова А. А. *Бесконечные деревья в алгоритме проверки условия эквивалентности итераций конечных языков. Часть II* // International Journal of Open Information Technologies. – 2021. – Vol. 9. No. 5. – P. 1–11.

[9] Мэн Л., Мельников Б. Ф. *Восемь вариантов конечных автоматов для проверки выполнения отношения покрытия итераций двух конечных языков. Часть I* // International Journal of Open Information Technologies. – 2023. – Vol. 11. No. 11. – P. 1–9.

[10] Мэн Л., Мельников Б. Ф. *Восемь вариантов конечных автоматов для проверки выполнения отношения покрытия итераций двух конечных языков. Часть II* // International Journal of Open Information Technologies. – 2024. – Vol. 12. No. 2. – P. 1–11.

[11] Ляпин Е. *Полугруппы*. – М., Физматлит. – 1960. – 592 с.

[12] Биркгоф Г., Барти Т. *Современная прикладная алгебра*. – М., Мир. – 1976. – 400 с.

[13] Лаллеман Ж. *Полугруппы и комбинаторные приложения*. – М., Мир. – 1985. – 440 с.

[14] Саломаа А. *Жемчужины теории формальных языков*. – М., Мир. – 1986. – 159 с.

[15] Скорняков Л. (ред.) *Общая алгебра. Том 2*. – М., Наука. – 1991. – 480 с.

[16] Pin J.-E. *Mathematical Foundations of Automata Theory*. – Berlin, Springer-Verlag. – 2012. – 310 p.

[17] Солтер Н., Клепер С. *C++ для профессионалов*. – М., Диалектика. – 2006. – 904 с.

МЭН Линцянь,
магистрант Московского государственного университета им. М. В. Ломоносова (`http://msu.ru/`),
email: `kostya45e@mail.ru`,
ORCID: `orcidID=0009-0009-2933-5684`.

Борис Феликсович МЕЛЬНИКОВ,
профессор Университета МГУ – ППИ в Шэньчжэне (`http://szmsubit.ru/`),
email$_1$: `bormel@smbu.edu.cn`,
email$_2$: `bf-melnikov@yandex.ru`,
mathnet.ru: `personid=27967`,
elibrary.ru: `authorid=15715`,
scopus.com: `authorId=55954040300`,
ORCID: `orcidID=0000-0002-6765-6800`.

On the software implementation of the groupoid of construction of PRI automaton and related computational experiments

Meng Lingqian, Boris Melnikov

***Abstract*—This paper is a continuation of the previous paper by the same authors, which considered variants for defining the groupoid and semigroup of the primary automaton PRI. To study various emerging examples of a groupoid, which in important special cases is a semigroup, we implemented appropriate computer programs in C++ and performed computational experiments, which are described in this paper.**

The main purpose of creating a program and conducting computational experiments. It is an attempt to understand the internal structure of the resulting groupoids (in special cases, semigroups), as well as an attempt to "explain the nature" of the special binary relations considered in many previous papers (the so-called relations of coverability and equivalence at infinity), given on sets of finite formal languages.

The program described in the article has shown its effectiveness in the study of complex examples in which the number of elements of the resulting groupoid was measured in several tens. The program produced interesting results for further interpretation, and at the same time its execution time approximately coincided with the theoretically calculated values.

The main conclusions, which were obtained, in particular, as an analysis of the conducted computational experiments, can be considered as follows.

It is obvious that languages over the same one-letter alphabet are equivalent at infinity. There is a connection between the condition for obtaining a semigroup and equivalence at infinity, which is shown precisely in these languages. We show that for such languages (over a one-letter alphabet), the groupoid is always a semigroup, and the latter is always commutative and is a monoid.

When calculating the characters and factor representations of the elements of a semigroup of interest to us, the series of degrees contain information of idempotents in a pair consisting of an index and a period. Therefore, it is possible not to find the idempodents of the obtained semigroups separately, and calculate their number after calculating the series of degrees.

Thus, based on the examples obtained by the program, we obtain descriptions of the properties of the groupoid, which can subsequently be formulated in the form of statements and theorems.

***Keywords*—formal languages, morphisms, finite automata, groupoids and semigroups, algorithms and programs, C++.**

References

[1] Meng Lingqian, Melnikov B. *On the groupoid of the primary automaton PRI defined for one finite language* // International Journal of Open Information Technologies. – 2023. – Vol. 11. No. 9. – P. 1–12. (In Russian.)

[2] Melnikov B. *Description of special submonoids of the global supermonoid of the free monoid* // News of higher educational institutions. Mathematics. – 2004. – No. 3. – C. 46–56. (In Russian.)

[3] Melnikov B. *The equality condition for infinite catenations of two sets of finite words* // International Journal of Foundation of Computer Science. – 1993. – Vol. 4. No. 3. – P. 267–274.

[4] Melnikov B. *Variants of finite automata corresponding to infinite iterative morphism trees. Part I* // International Journal of Open Information Technologies. – 2021. – Vol. 9. No. 7. – P. 5–13. (In Russian.)

[5] Melnikov B. *Variants of finite automata corresponding to infinite iterative morphism trees. Part II* // International Journal of Open Information Technologies. –2021. – Vol. 9. No. 10. – P. 1–8. (In Russian.)

[6] Melnikov B. *2ω-finite automata and sets of obstructions of their language* // The Korean Journal of Computational and Applied Mathematics (Journal of Applied Mathematics and Computing). – 2000. – Vol. 6. No. 3. – P. 565–574.

[7] Melnikov B., Melnikova A. *Infinite trees in the algorithm for checking the equivalence condition of iterations of finite languages. Part I* // International Journal of Open Information Technologies. – 2021. – Vol. 9. No. 4. – P. 1–11. (In Russian.)

[8 Melnikov B., Melnikova A. *Infinite trees in the algorithm for checking the equivalence condition of iterations of finite languages. Part II* // International Journal of Open Information Technologies. – 2021. – Vol. 9. No. 5. – P. 1–11. (In Russian.)

[9] Meng Lingqian, Melnikov B. *Eight variants of finite automata for checking the fulfillment of the iteration coverage ratio of two finite languages. Part I* // International Journal of Open Information Technologies. – 2023. – Vol. 11. No. 11. – P. 1–9. (In Russian.)

[10] Meng Lingqian, Melnikov B. *Eight variants of finite automata for checking the fulfillment of the iteration coverage ratio of two finite languages. Part II* // International Journal of Open Information Technologies. – 2024. – Vol. 12. No. 2. – P. 1–11. (In Russian.)

[11] Lyapin E. *Semigrouops*. – Moscow, Fizmatlit. – 1960. – 592 p. (In Russian.)

[12] Birkhoff G., Bartee T. *Modern Applied Algebra*. – NY, McGraw-Hill Ed. – 1970. – 431 p.

[13] Lallement G. *Semigroups and Combinatorial Applications*. – NY, John Wiley & Sons. – 1979. – 376 p.

[14] Salomaa A. *Jewels of Formal Language Theory*. – Rockville, Maryland, Computer Science Press. – 1981. – 144 p.

[15] Skornyakov L. (Ed.) *General algebra. Vol. 2*. – Moscow, Nauka. – 1991. – 480 p. (In Russian.)

[16] Pin J.-E. *Mathematical Foundations of Automata Theory*. – Berlin, Springer-Verlag. – 2012. – 310 p.

[17] Solter N., Kleper S . *Professional C++*. – NJ, John Wiley & Son. – 2005. – 864 p.

MENG Lingqian,
Post-graduate student of Moscow State University
(http://szmsubit.ru/),
email: kostya45e@mail.ru,
ORCID: orcidID=0009-0009-2933-5684.

Boris MELNIKOV,
Professor of Shenzhen MSU – BIT University, China
(http://szmsubit.ru/),
email$_1$: bormel@smbu.edu.cn,
email$_2$: bf-melnikov@yandex.ru,
mathnet.ru: personid=27967,
elibrary.ru: authorid=15715,
scopus.com: authorId=55954040300,
ORCID: orcidID=0000-0002-6765-6800.