Санкт-Петербургский государственный университет

Программная инженерия

Соколова Полина Александровна

Использование автоматов в интерпретаторе MACASM

Выпускная квалификационная работа

Научный руководитель:
ст. преп. кафедры системного программирования, к. ф.-м. н.
Луцив Д. В.

Консультант:
ст. преп. кафедры системного программирования
Баклановский М. В.

Рецензент:
зав. кафедрой технологий программирования Университета ИТМО, д. т. н., проф.
Шалыто А. А.

Санкт-Петербург
2020

SAINT-PETERSBURG STATE UNIVERSITY

Software Engineering

Polina Sokolova

Automata use in interpreter MACASM

Graduation Thesis

Scientific supervisor:
Assistant Prof., C. Sc. Luciv D. V.

Consultant:
Assistant Prof. Baklanovsky M. V.

Reviewer:
Head of the Chair, ITMO University, D. Sc., Prof. Shalyto A. A.

Saint-Petersburg
2020

Оглавление

Введение

Современные интерпретаторы отличаются от своих предшественников. Прежде компилятором называлась программа, которая из исходного кода генерирует код процессора, а интерпретатором – которая построчно читает и исполняет код. Кроме того, раньше эти понятия были взаимоисключающими.

Сегодня интерпретаторы могут читать текст целиком и переводить его в некоторое промежуточное представление до начала исполнения [1]. Эти системы одновременно содержат внутри себя и интерпретатор, и компилятор. Таким образом, сейчас граница между терминами "транслятор", "компилятор", "интерпретатор" стерлась, и они являются синонимами.

В современных интерпретаторах много автоматов. При помощи конечных автоматов реализуются лексические анализаторы и происходит сопоставление строк регулярным выражениям. Синтаксический разбор некоторых скобочных языков также возможен с использованием конечных автоматов с несколькими операциями на стеке (Visibly Pushdown Automaton, VPA) [2]. Кроме того, автоматы могут быть использованы при реализации ассоциативных массивов.

Для упрощения работы eval в интерпретаторах взамен аппаратной адресации используется именная адресация переменных. В таких системах адрес значения возвращается по имени переменной. Скорость поиска переменной в списке напрямую влияет на скорость работы программы. Таким образом, проблема быстрого нахождения переменной в списке является одной из главных внутрисистемных задач для ограниченного класса языков.

Для решения этой проблемы используются ассоциативные массивы. Ассоциативный массив состоит из словаря и массива и используется для хранения пар ключ-значение [3]. Словарь – это множество с операциями добавления, удаления и проверки, содержится ли элемент в множестве.

Разработка языка типа Perl, Python, JavaScript начинается с реализации ассоциативного массива. В основном для этого используют хеш-

таблицы или самобаласирующиеся деревья (красно-черное, например), но использование автоматов не опробовано, хотя это тоже возможная конструкция для хранения словаря.

Автоматный массив (автомассив) – это придуманный нами термин для обозначения автоматной реализации ассоциативного массива, как альтернативы хеш-таблице. Словарь хранится в автомате, который строится инкрементально при добавлении нового элемента.

На этапе лексического анализа существует проблема распознавания служебных слов и идентификаторов [4]. Служебные слова используются для опередеделения конструкций и представляют собой фиксированные символьные строки, например, такие как for, do, if. Идентификатор тоже представляет собой символьную строку, которая ссылается на некоторую сущность, например, переменную или процедуру. Служебные слова обычно удовлетворяют правилам, по которым формируются идентификаторы. Таким образом, необходим механизм, определяющий является строка служебным словом или идентификатором.

Основным способом решения этой проблемы является заранее внести служебные слова в список или массив, а во время лексического анализа выполнить поиск в нем входной строки. Недостатком этого подхода является то, что для обработки одного слова необходимо сделать проход по строке и выполнить ее поиск.

Использование хеш-таблицы кажется удачным, но, когда служебных слов много, поиск в ней также может занимать некоторое время. С другой стороны, в основной лексический анализатор можно встроить автомат, распознающий служебные слова. Преимуществом этого подхода является то, что для обработки любого слова не потребуется больше, чем один проход.

Для уменьшения занимаемого места автомат бывает удобно минимизировать. Минимизация ДКА (детерминированного конечного автомата) – это задача преобразования данного автомата в эквивалентный с минимальным количеством состояний [5]. Автоматы называются эквивалентными, если они допускают один язык.

Одной из целей скриптовых языков является быстрая обработка

больших объемов данных, текстов. Эти задачи решаются при помощи регулярных выражений. Регулярное выражение – это последовательность символов, которая определяет шаблон поиска [6]. Трудно представить современный язык без поддержки регулярных выражений.

Синтаксический анализатор является неотъемлемой частью интерпретатора. При синтаксическом разборе часто используются автоматы с магазинной памятью (МП-автоматы) [7]. Кроме того, Раджив Алюр в своей работе [2] предложил Visibly Pushdown Automaton (VPA) и идею блочного разбора. VPA – это конечный автомат, который использует стек только для определения начала и конца блока. Этот автомат также может использоваться при разборе некоторых языков, например, для обработки XML-документов [8].

Многоуровневая архитектура кода (МАК) – программное обеспечение для обфускации, нанесения цифровых водяных знаков на код и многого другого. Для структур МАК и МАК-обфускации возникла необходимость в написании своего ассемблера MACASM с внутренним языком Macer. Сейчас для работы с ассемблерными файлами используются скриптовые языки, например, Perl, Python, потому что в языке ассемблера нет нужных инструментов. Из-за работы с промежуточным текстовым представлением возникают потери времени: уже сейчас есть задачи МАК, которые выполняются десятки минут, и это только начало. Основная причина разработки MACASM заключается в том, чтобы обеспечить скорость работы таких задач, и заодно написать свой ассемблер с языком, на котором будет удобно программировать.

Разработка системы начинается с реализации ассоциативных массивов в виде автоматных массивов. Автоматная техника является достаточно скоростной, и автомассивы – это только часть возможного её применения. Таким образом, можно сделать шаг в сторону других модулей интерпретатора, использующих конечные автоматы.

1. Постановка задачи

Целью данной дипломной работы является разработка модулей интерпретатора MACASM, использующих конечные автоматы. Для достижения этой цели были сформулированы следующие задачи.

1. Проанализировать популярные реализации ассоциативных массивов и алгоритмы минимизации ДКА.
2. Реализовать ассоциативный массив в виде автоматного массива.
3. Разработать утилиту построения минимального ДКА по словарю.
4. Реализовать модуль для построения ДКА по регулярному выражению.
5. Реализовать модуль для синтаксического разбора языка Macer.
6. Провести апробацию созданных средств.

2. Обзор

2.1. Техники реализации ассоциативных массивов

Ассоциативный массив – это структура данных для хранения пар ключ-значение, которая представляет собой словарь + массив. В словаре хранятся ключи, в массиве – значения (см. рис. 1).

Рис. 1: Ассоциативный массив

Существует множество реализаций словарей [3]. Самая простая может быть основана на обычном массиве или списке. В хеш-таблицах для получения индекса, по которому лежит значение, используется хеш-функция [9]. Также популярны реализации с использованием различных деревьев (префиксное [10], красно-черное [11], АВЛ-дерево [12]), где для получения индекса необходимо пройти от корня до листа по соответствующим ребрам (см. рис. 2).

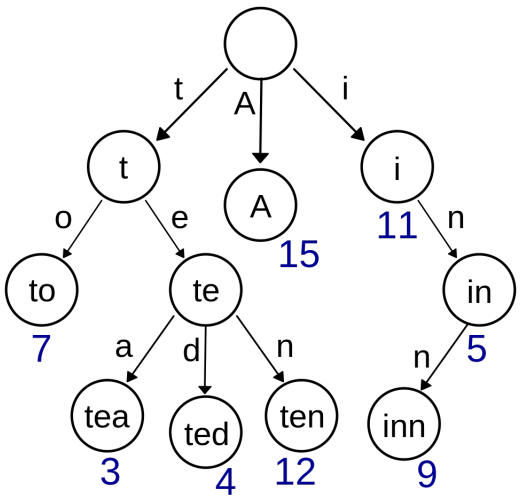

Рис. 2: Префиксное дерево (trie)

2.2. Разрешение коллизий в хеш-таблицах

При использовании хеш-таблиц возможно появление коллизий, т. е. случаев, когда хеш-функция возвращает одно и то же значение для нескольких ключей. Существует два основных способа разрешения коллизий:

- цепочки (separate chaining/open hashing);
- открытая адресация (open addressing/closed hashing).

В первом случае в каждой ячейке массива мы храним указатель на список элементов с одинаковым хеш-значением (см. рис. 3).

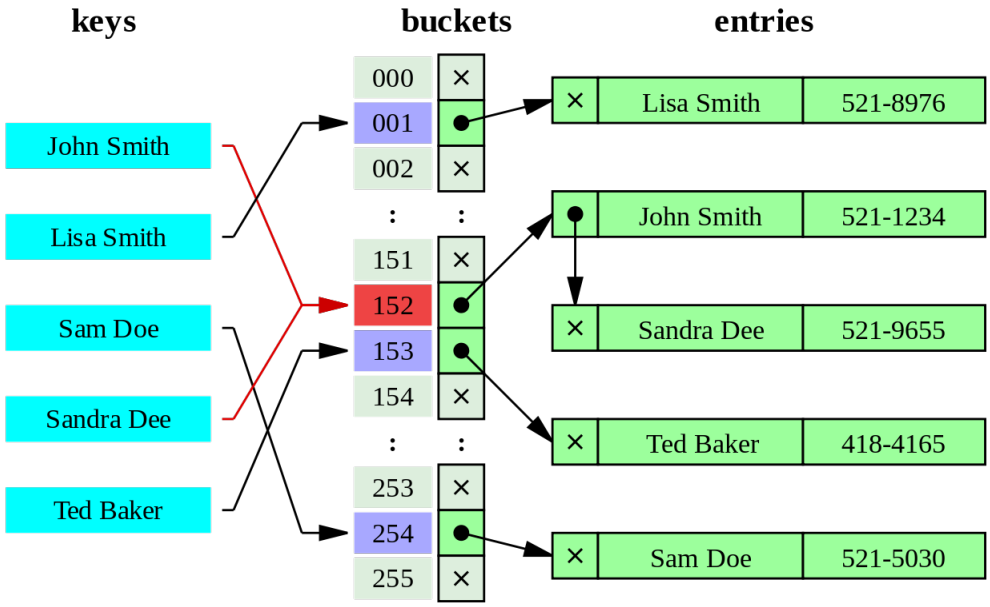

Рис. 3: Разрешение коллизий при помощи цепочек.

Во втором случае при добавлении элемента мы пытаемся найти первую свободную ячейку, начиная с той, на которую указала хеш-функция (см. рис. 4). Поиск такой ячейки также может осуществляться несколькими способами:

- линейное хеширование (ищем следующую свободную ячейку);
- квадратичное хеширование (индекс следующей просматриваемой ячейки вычисляется как значение квадратичной функции от текущей);

- двойное хеширование (интервал между ячейками вычисляется с помощью второй хеш-функции).

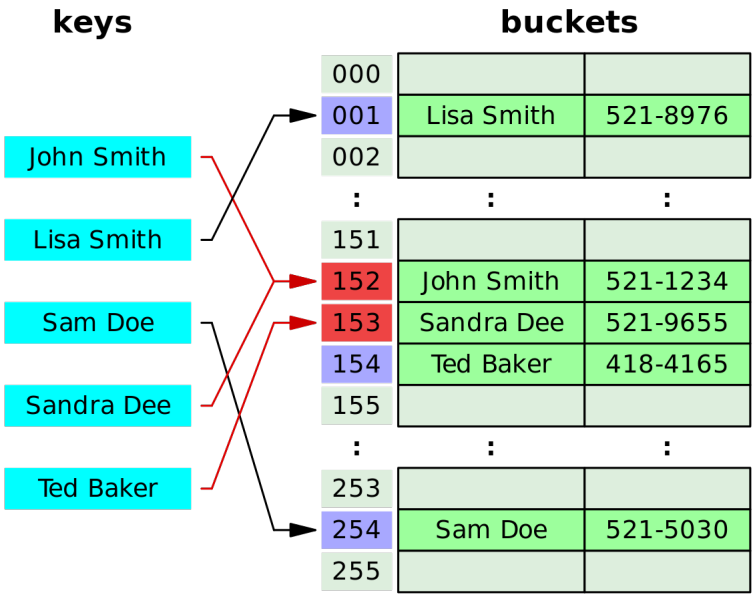

Рис. 4: Разрешение коллизий при помощи открытой адресации.

2.3. Хеш-таблицы в C++ и Go

В табл. 1 представлены основные характеристики наиболее используемых реализаций хеш-таблиц в языках C++ и Go [13, 14, 15, 16]. В таблице указан термин ”бакет” (bucket), который сопровождает цепочечные реализации. Бакет – это место, куда вставляются элементы.

Одним из интересных вариантов разрешения коллизий является хеширование Робина Гуда [17]. Основная его идея заключается в том, что когда мы вставляем новый элемент с помощью линейного хеширования, то считаем, насколько далеко мы находимся от своей идеальной позиции. Если мы находимся дальше от неё, чем текущий элемент от своей идеальной позиции, то мы меняем местами новый элемент с существующим и пытаемся найти новое место для того, который ”выселили”. Автор реализации ska::flat_hash_map в своем блоге экспериментально подтвердил [16], что написал самую быструю хеш-таблицу в C++.

Хеш-таблица	Разрешение коллизий	Пробирование (probing)	Особенности
map (Go)	цепочки	–	8 пар ключ-значение в блоке (bucket)
std::unordered_map (C++)	цепочки	–	–
spp::sparse_hash_map (C++)	открытая адресация	квадратичное	–
google::dense_hash_map (C++)	открытая адресация	квадратичное	–
ska::flat_hash_map (C++)	открытая адресация	линейное	хеширование Робина Гуда, ограничение на количество проб, самая быстрая хеш-таблица в C++

Таблица 1: Характеристики хеш-таблиц в C++ и Go.

2.4. Минимизация конечного автомата

Проблема минимизации ДКА изучалась с конца 50-х годов [18]. Она заключается в поиске минимального конечного автомата, который допускает тот же язык, что и исходный. Алгоритмы, решающие эту задачу, используются в различных приложениях, начиная с построения компиляторов [19]. Автомат является минимальным, если не существует эквивалентного автомата с меньшим количеством состояний.

Существует множество алгоритмов минимизации ДКА. В работах

[19, 20] подробно сравниваются различные подходы к решению этой задачи. Типичный алгоритм минимизации ДКА имеет временную сложность $O(n^2)$ [21]. В 1971 г. Джон Хопкрофт предложил алгоритм [22], сложность которого в худшем случае оценивается как $O(n \log n)$. До сих пор этот результат остается лучшим.

Основная идея алгоритма Хопкрофта заключается в построении последовательности разбиений, в которой каждое разбиение получается из предыдущего в результате выбора класса символов из разбиения и последовательного выполнения шагов расщеплений по сплитерам (C, x) для всех x из входного алфавита. Другими словами, мы последовательно измельчаем разбиение множества состояний на классы эквивалентности до тех пор, пока не получится разбиение, которое уже нельзя измельчить.

В последнее время исследования алгоритмов минимизации автоматов в основном сосредоточены на постепенном построении и динамической минимизации ДКА [20], что является отдельной категорией. В работе [23] был предложен инкрементальный подход к минимизации автомата: с добавлением новых строк одна за другой и сжатием на лету. В этом алгоритме используется альтернативное понятие минимальности (псевдоминимизация). В полученном автомате по-прежнему не существует двух одинаковых путей из двух разных состояний, но он необязательно является минимальным.

Существуют также алгоритмы минимизации автоматов Мура и Мили. Например, алгоритм минимизации полностью определенных автоматов Мили или Мура, предложенный В. М. Глушковым [24]. Процесс минимизации здесь заключается в объединении в одно состояние множеств так называемых совместимых состояний. Состояния являются совместимыми, если все определенные результаты (выходные значения) применения любого слова (входное значение) к ним будут одинаковыми. В автоматах Мура также необходимо, чтобы эти состояния имели одинаковые отметки.

2.5. Многоуровневая архитектура кода

Многоуровневая архитектура кода (МАК) – это программный продукт для обфускации, нанесения цифровых водяных знаков на код, профайлинга и многого другого. На рис. 5 представлена схема трансляции программы в исполняемый файл с использованием МАК.

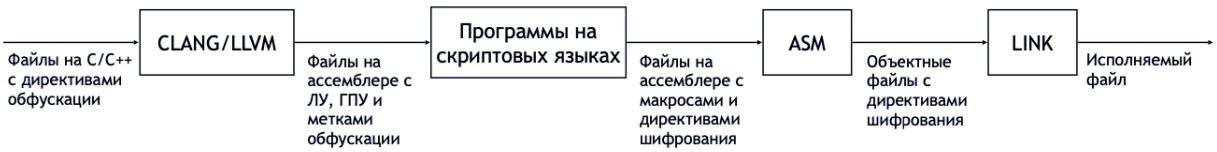

Рис. 5: Схема трансляции в МАК.

На вход поступают программы на C/C++, которые переводятся в ассемблерные файлы. Далее над этими файлами происходит текстовый процессинг при помощи программ на скриптовых языках, поскольку сейчас язык ассемблера не обладает нужными инструментами. Задачи МАК, например, обфускация, делаются на этом этапе.

Уже сейчас возникла проблема в скорости их выполнения: некоторые задачи выполняются десятки минут, хотя на данный момент включены ещё не все возможности. Потери времени возникают как раз из-за промежуточного текстового представления. В связи с этим возникла идея в создании своего ассемблера MACASM с внутренним языком Macer (см. рис. 6).

Рис. 6: Схема трансляции программы с использованием MACASM.

Основная цель MACASM – это специальные объекты, например, линейные участки, которыми хочется манипулировать прямо в тексте ассемблерной программы. Сейчас линейные участки обрабатываются как обычные текстовые строки. Таким образом, управлять объектами

в Macer будет гораздо быстрее, чем обрабатывать строки на скриптовых языках, из-за отсутствия промежуточного представления в виде текста.

Главная задача MACASM заключается в том, чтобы задачи МАК выполнялись быстро. Таким образом, MACASM – это свой современный ассемблер с языком Macer, удобным для программирования и обладающим нужными инструментами для управления спецобъектами.

Основные преимущества использования MACASM:

- скорость;
- гораздо большая универсальность, чем есть сейчас (не нужно, чтобы программа была на C/C++, можем работать с любыми ассемблерными листингами сразу);
- поставляемо заказчику (MACASM можно поставлять с программами на Macer, нет привязки к Clang).

2.6. Идеальное хеширование

Автоматная техника реализации словаря заключается в том, что слова хранятся в автомате, который строится инкрементально при добавлении нового слова. Идеальное хеширование – это хеширование без коллизий. Оно может быть реализовано с использованием автоматной техники реализации словаря.

В обычном автомате, который строится инкрементально и не минимизируется, уже есть идеальное хеширование (номера допускающих состояний для разных слов различны). Если автомат минимизировали, то может возникнуть ситуация, когда у всех слов одно допускающее состояние, и к нему уже нельзя привязать уникальные индексы слов. В работе [25] приведен способ подсчета уникального номера слова.

У каждого состояния есть счетчики количества слов, проходящих через это состояние (см. рис. 7). Есть текущие состояние и буква, по которой нужно сделать переход. Мы просматриваем все переходы из текущего состояния в состояния по буквам, которые расположены по

алфавиту раньше нашей буквы, и складываем значения всех этих состояний в общую сумму. Делаем так для всех букв нашего слова. Если в конце слова мы пришли в допускающее состояние, то прибавляем еще 1 к общей сумме. Она и будет номером слова по алфавиту.

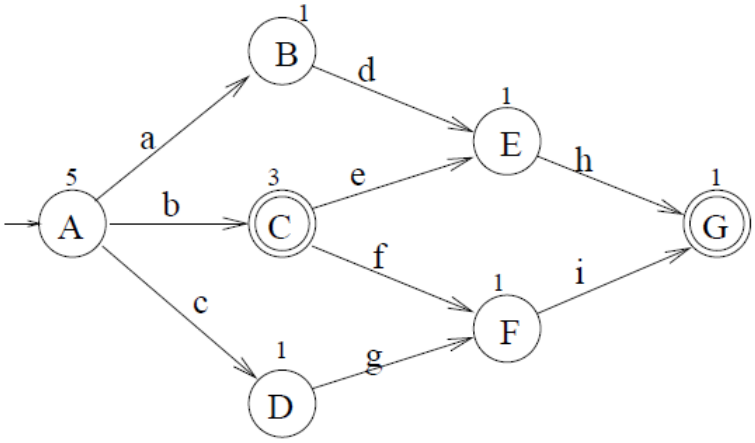

Рис. 7: Пример автомата с подсчетом номера слова.

2.7. Блочный разбор

Основная идея блочного разбора заключается в том, чтобы рассматривать файл не как сплошной поток, а как блоки с сигналами начала и конца. Этот подход был предложен в работе [2]. Если встречается новый блок, то нужно подавать его на вход новому автомату, допускающему другой язык или тот же, но с нуля. Это реализуется при помощи Visibly Pushdown Automaton (VPA), который представляет собой конечный автомат и использует стек только для определения начала и конца блока.

В отличие от МП-автомата, который имеет один алфавит входных символов, VPA имеет три непересекающихся алфавита входных символов: множество вызовов (call), множество возвратов (return) и множество внутренних действий (internal actions). Автомат ограничен тем, что он кладет на стек только тогда, когда читает символ из алфавита вызовов, выталкивает со стека, когда читает символ из алфавита возврата, и не использует стек, когда читает символ из третьего алфавита. В сравнении с МП-автоматом, VPA проще в разработке, отладке и со-

провождении, а также быстрее работает. Класс языков, допускаемых VPA, называется Visibly Pushdown Languages (VPL) и лежит между классами регулярных и КС-языков.

2.8. Преобразование регулярного выражения в конечный автомат

Поиск шаблона регулярного выражения в строке обычно происходит при разборе слова конечным автоматом, допускающим тот же язык. Алгоритм Мак-Нотона–Ямады–Томпсона [4] для каждого подвыражения строит НКА с единственным принимающим состоянием (см. рис. 8).

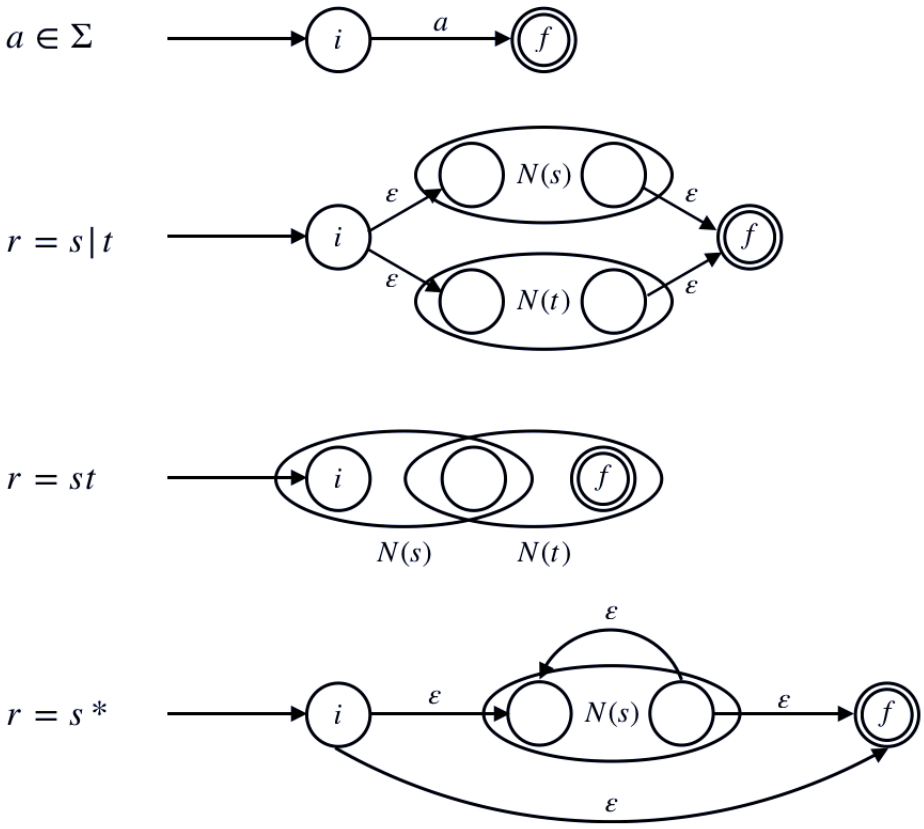

Рис. 8: НКА для регулярных выражений.

Работа с регулярными выражениями требует моделирования ДКА (или НКА) [4]. Моделировать НКА труднее, поскольку при работе с ним часто приходится делать выбор для нескольких входных символов

и ε. Таким образом, оказывается очень важной задача преобразования НКА в ДКА.

Общая идея, лежащая в основе алгоритма построения подмножеств (subset construction) ДКА из НКА, заключается в том, что каждое состояние строящегося ДКА соответствует множеству состояний НКА [4]. После чтения входной строки ДКА находится в состоянии, соответствующем множеству состояний, которых может достичь НКА при чтении этой же строки.

3. Автоматный массив

Основным недостатком хеш-таблиц является перехеширование при достижении коэффициентом заполненности значения, выбранного заранее. Это занимает определенное время, поэтому было решено реализовать ассоциативный массив в виде автоматного массива (автомассива). Автомассив – это придуманный нами термин для обозначения автоматной реализации ассоциативного массива, как альтернативы хеш-таблице. Автомассивы предназначены для работы во время исполнения программы.

3.1. Хранение словаря

Словарь в автомассиве хранится в детерминированном конечном автомате, который строится инкрементально, т. е. не перестраивается, а достраивается при добавлении новых элементов. На рис. 9 изображена таблица переходов автомата, допускающего слова ”мама” и ”папа”.

Символы \ Состояния	0	1	2	3	(4)	5	6	7	(8)
а	-	2	-	4	-	6	-	8	-
м	1	-	3	-	-	-	-	-	-
п	5	-	-	-	-	-	7	-	-
...	-	-	-	-	-	-	-	-	-

Рис. 9: Пример таблицы переходов ДКА.

Индексом для получения значения в автомассиве является номер состояния, в которое мы приходим при разборе ключа автоматом. При условии, что оно является допускающим.

3.2. Управление памятью

В автомассиве есть два типа блоков: таблица переходов (для хранения ключей) и массив значений (для хранения значений) (см. рис. 10). В данной реализации нет такой проблемы как перехеширование, поскольку все данные лежат не в одной области памяти (блоки), а сразу в нескольких.

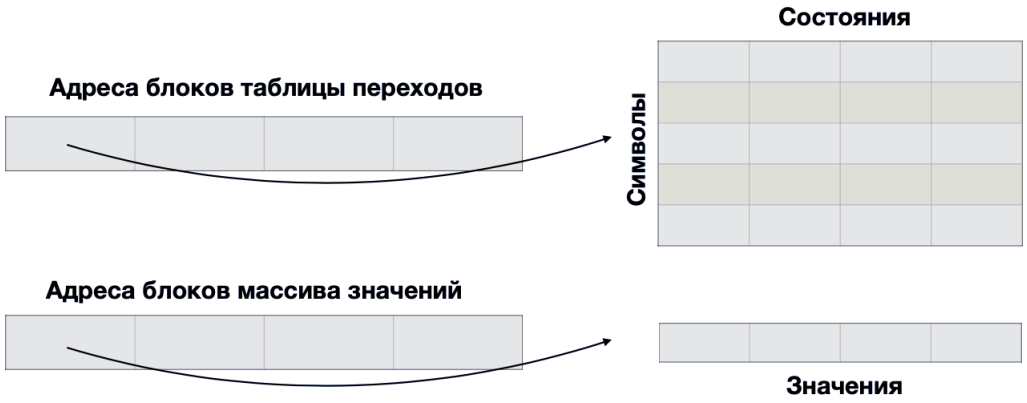

Рис. 10: Блоки в автомассиве.

Изначально выделяется по одному блоку для хранения ключей и значений. Как только они заполнились, выделяется новая пара блоков, куда записываются последующие элементы. Адреса всех таких выделенных блоков хранятся в соответствующих массивах.

На рис. 11 представлен график сравнения скорости вставки строк при разных размерах блока таблицы переходов. На графике видно, что время вставки в разных вариациях блоков примерно одинаково или отличается незначительно.

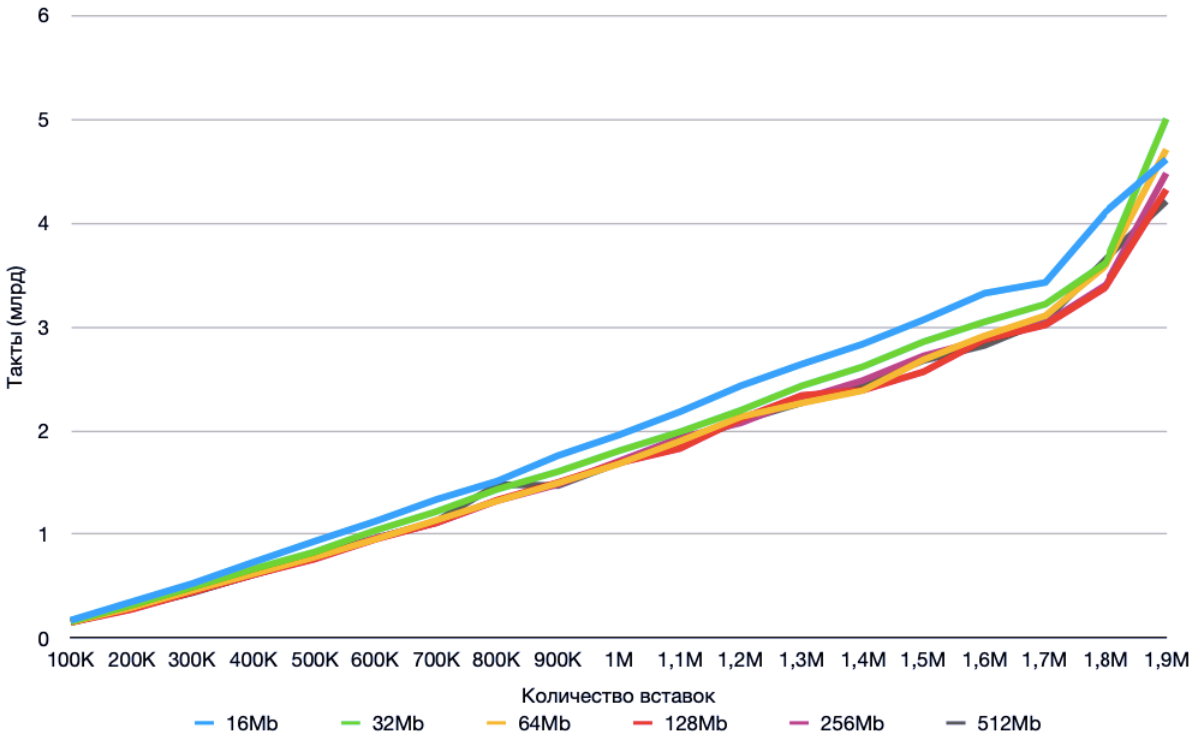

Рис. 11: Вставка строк длины 10 в автомассив при разных размерах блока таблицы переходов.

4. Утилита построения минимального ДКА по словарю

Данная утилита принимает на вход список слов (словарь) с атрибутами (например, токен или номер слова), строит по нему минимальный ДКА и возвращает его пользователю. Утилита предназначена для работы во время разработки интерпретатора.

4.1. Построение

На предварительном этапе утилита расставляет номера слов как атрибуты, если задан соответствующий ключ, и производит сортировку слов по алфавиту. В основе данной утилиты лежит реализация автомассива. На первом этапе происходит добавление слов в автомассив, то есть строится таблица переходов ДКА, допускающая наши слова. При добавлении слова в автомат возле каждого состояния, через которое мы проходим, увеличивается счетчик количества слов, проходящих через него. Это делается для того, чтобы после минимизации уметь вычислять уникальный номер слова. Далее символы объединяются в группы, чтобы не хранить пустые или похожие строки. После этого автомат минимизируется алгоритмом Хопкрофта.

Поскольку данная утилита разрабатывалась для использования на этапе построения лексического анализатора, то необходимо также, чтобы выходной автомат мог говорить не только, допускает он слово или нет, но и возвращать его токен. Таким образом, служебные слова, подаваемые на вход, являются ключом в автомассиве, а их токены (или какие-то другие свойства, например, номера) – значениями. Для этого реализована процедура подсчета номера слова.

Поскольку в данном алгоритме есть перебор переходов по всем символам, лежащим по алфавиту раньше нашего, то по скорости поиска мы падаем до бора (trie). Это недостаток использования минимального автомата для возвращения атрибута по слову. В связи с этим, сделана еще одна версия утилиты без минимизации. В случаях, где проблема

памяти стоит неостро, рекомендуется использовать именно эту версию.

4.2. Пример работы

Поскольку разработка интерпретатора MACASM происходит на языке ассемблера (MASM), то на выходе утилита (autogen) выдает ассемблерный файл, который можно легко подключить и использовать при лексическом анализе. На рис. 12 представлен пример выходного файла с минимальным автоматом.

```
; start state - 1, kill state - 0
;
; BLOCK+1, END+4, ENUM+6, EXIT+3, LEN+10, LOCAL+2, LOOP+5, NUM+9, PARAMS+7, PRINT+8,
;

groups  =   18

        .data
char_n  db  65 dup(0),1,2,3,4,5,3 dup(0),6,0,7,8,9,10,11,12,0,13
        db  14,15,16,2 dup(0),17,39 dup(0)
;             A,B,C,D,E,I,K,L,M,N,O,P,R,S,T,U,X,
state_t dd  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 0
        dd  0,0,2,0,0,7,0,0,12,0,18,0,19,0,0,0,0,0 ; 1
        dd  0,0,0,0,0,0,0,0,3,0,0,0,0,0,0,0,0 ; 2
        dd  0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0 ; 3
        dd  0,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 4
        dd  0,0,0,0,0,0,0,6,0,0,0,0,0,0,0,0,0 ; 5
        dd  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 6
        dd  0,0,0,0,0,0,0,0,0,0,8,0,0,0,0,0,10 ; 7
        dd  0,0,0,0,6,0,0,0,0,0,0,0,0,0,0,9,0 ; 8
        dd  0,0,0,0,0,0,0,0,0,6,0,0,0,0,0,0,0 ; 9
        dd  0,0,0,0,0,0,11,0,0,0,0,0,0,0,0,0,0 ; 10
        dd  0,0,0,0,0,0,0,0,0,0,0,0,0,0,6,0,0 ; 11
        dd  0,0,0,0,0,13,0,0,0,0,0,14,0,0,0,0,0,0 ; 12
        dd  0,0,0,0,0,0,0,0,0,0,6,0,0,0,0,0,0,0 ; 13
        dd  0,0,0,15,0,0,0,0,0,0,0,17,0,0,0,0,0,0 ; 14
        dd  0,16,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 15
        dd  0,0,0,0,0,0,0,0,6,0,0,0,0,0,0,0,0,0 ; 16
        dd  0,0,0,0,0,0,0,0,0,0,0,0,6,0,0,0,0,0 ; 17
        dd  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,9,0 ; 18
        dd  0,20,0,0,0,0,0,0,0,0,0,0,24,0,0,0,0 ; 19
        dd  0,0,0,0,0,0,0,0,0,0,0,0,0,21,0,0,0,0 ; 20
        dd  0,22,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 21
        dd  0,0,0,0,0,0,0,0,0,23,0,0,0,0,0,0,0,0 ; 22
        dd  0,0,0,0,0,0,0,0,0,0,0,0,0,0,6,0,0,0 ; 23
        dd  0,0,0,0,0,0,25,0,0,0,0,0,0,0,0,0,0,0 ; 24
        dd  0,0,0,0,0,0,0,0,0,0,0,11,0,0,0,0,0,0 ; 25
state_f db  6 dup(0),1,19 dup(0)
total_w dd  0,10,1,1,1,1,1,3,2,1,1,1,3,1,2,1,1,1,1,2,1,1,1,1,1,1
value   dq  v0,v1,v2,v3,v4,v5,v6,v7,v8,v9
value_l dd  v0_l,v1_l,v2_l,v3_l,v4_l,v5_l,v6_l,v7_l,v8_l,v9_l
v0      db  '1'
v0_l    =   $-v0
v1      db  '4'
v1_l    =   $-v1
```

Рис. 12: Пример файла, сгенерированного утилитой autogen.

Выходной файл содержит:

- количество групп символов (groups);
- распределение символов по группам (char_n);
- таблицу переходов ДКА (state_t);
- маркеры допускающих состояний (state_f);
- счетчики количества слов, проходящих через каждое из состояний (total_w);
- массивы значений (value) и длин значений (value_l);
- значения и их длины (v0, v0_l, v1, v1_l, ...);
- процедуру разбора входного слова сгенерированным автоматом и выводом его атрибута.

5. Модуль для построения ДКА по регулярному выражению

На вход данному модулю поступает регулярное выражение с операциями конкатенации, перечисления, звезды Клини и скобками. Таким образом, данный модуль рассчитан на работу с нерасширенными регулярными выражениями и предназначен для использования во время исполнения программы.

Поскольку язык регулярных выражений – это скобочный язык, то для его разбора необходимо использование стека. В данном модуле разбор происходит при помощи VPA. Работа со стеком происходит только при чтении скобок: кладем на стек по открывающей скобке, берем со стека по закрывающей скобке.

Автомат разбирает каждый блок как независимый. Другими словами, как только мы прочитали открывающую скобку, текущий "контекст" (начало блока, наличие альтернатив и т. д.) кладется на стек, и мы разбираем новый блок с нуля. Далее при чтении закрывающей скобки мы заканчиваем обработку этого блока, берем со стека прошлый "контекст" и продолжаем разбор старого блока.

Помимо разбора регулярного выражения происходит построение НКА. Для этого в некоторых состояниях автомата выполняются функции (как в автомате Мура). Автомат строится инкрементально: при чтении символов алфавита или операторов происходит модификация текущего автомата в соответствии с алгоритмом преобразования регулярного выражения в НКА.

Далее происходит преобразование НКА в ДКА путем построения подмножеств и минимизация ДКА. Преобразование в ДКА происходит после прочтения всего регулярного выражения, поэтому при необходимости можно оставить автомат недетерминированным и работать с ним. Аналогично можно не выполнять процесс минимизации ДКА.

Это может быть полезным в случаях, когда нам важна скорость преобразования, и мы не хотим хранить автомат после использования. С другой стороны, если регулярное выражение достаточно большое и

преобразование его в автомат занимает определенное время, то бывает нужна функция сохранения автомата (минимального, чтобы уменьшить расходы по памяти).

Автоматы хранятся в автомассиве, который был модифицирован для работы с НКА. Например, в данном модуле в каждой ячейке таблицы переходов автомассива можно хранить не 1 состояние, а список.

При разборе регулярного выражения могут возникать следующие ошибки:

- несоответствие открывающих и закрывающих скобок;
- ошибка при использовании звезды Клини (над пустой подстрокой).

Если регулярное выражение корректно, в результате работы модуля в автомассиве лежит автомат, построенный по регулярному выражению, который можно использовать далее для поиска шаблонов в текстах. На рис. 13 показан пример полученных автоматов. Если регулярное выражение задано неверно, возвращаются значения места и типа ошибки.

```
Regular expression: a|(b*|c)d

Start state: 1, Devil state: 0 (for all automata)
x-->y means epsilon-transition

NFA
1-->5, 1-->4
2-->3
4-a->2
5-->11, 5-->10
6-->8, 6-->7
7-b->6
8-->9
9-d->13
10-->8, 10-->7
11-c->12
12-->9
13-->3
Accept state: 3
Total states: 14

DFA
1-a->2, 1-b->3, 1-c->4, 1-d->5
3-b->3, 3-d->5
4-d->5
Accept state(s): 2 5
Total states: 6

Minimal DFA
1-a->2, 1-b->3, 1-c->4, 1-d->2
3-b->3, 3-d->2
4-d->2
Accept state(s): 2
Total states: 5
```

Рис. 13: Автоматы, построенные по регулярному выражению.

6. Модуль для синтаксического разбора языка Macer

6.1. Описание языка Macer

Интерпретатор MACASM работает с ассемблерными файлами, у которых в фигурных скобках записана программа на языке Macer, а снаружи – на языке ассемблера. Основу синтаксиса языка Macer составляют блоки.

1. {} – пустой блок. Далее фигурные скобки будут опущены.
2. BLOCK

   [LOCAL loc1[, ...]]

   [EXIT]

   END
3. BLOCK bool_expr

   [LOCAL loc1[, ...]]

   [EXIT]

   END
4. BLOCK bool_expr

   [LOCAL loc1[, ...]]

   [EXIT]

   END LOOP
5. BLOCK

   [LOCAL loc1[, ...]]

   [EXIT]

END LOOP bool_expr

6. BLOCK ENUM array_expr

   [LOCAL loc1[, ...]]

   [EXIT]

   END

7. var -> BLOCK BLOCK[PARAMS[prm1[, ...]]]

   [LOCAL loc1[, ...]]

   [EXIT]

   END[locX]

8. arr = (

   BLOCK[PARAMS[prm1[, ...]]]

   [LOCAL loc1[, ...]]

   [EXIT]

   END[locX][

   ,

   ...]

)

9. arr = (

   expr1 -> BLOCK[PARAMS[prm1[, ...]]]

   [LOCAL loc1[, ...]]

   [EXIT]

   END[locX][

   ,

   expr2 -> BLOCK[PARAMS[prm1[, ...]]]

```
        [LOCAL loc1[, ...]]
        [EXIT]
    END[ locX][
    ,
    ...][
    ,
    BLOCK[ PARAMS[ prm1[, ...]]]
        [LOCAL loc1[, ...]]
        [EXIT]
    END[ locX]]]
)
```

6.2. Синтаксический анализатор

Синтаксический анализатор состоит из трех автоматов. Первый автомат игнорирует все, что лежит за пределами фигурных скобок. Как только встречается открывающая фигурная скобка, управление передается второму автомату. С другой стороны, по закрывающей скобке управление передается обратно первому автомату.

Второй автомат - это лексер, который сопоставляет потоку символов токены (служебных слов, идентификатора, служебных символов). Частью этого автомата является ДКА, сгенерированный утилитой построения автомата по словарю служебных слов. Вторая часть лексера распознает служебные символы. Как только встречается служебный символ, лексер подает на вход третьему автомату предыдущий токен.

Третий автомат проверяет соответствие синтаксису языка. Поскольку блоки в языке Macer могут быть вложенными, для синтаксического разбора необходим автомат со стеком. В данном модуле используется VPA, поскольку он строится и работает быстрее, чем МП-автомат.

Работа со стеком происходит только по маркерам начала и конца блока: по слову BLOCK мы кладем на стек состояние, в которое необ-

ходимо вернуться после обработки вложенного блока, по слову END – снимаем со стека. Кроме того, в некоторых состояниях выполняются функции (как в автомате Мура). Например, нам необходимо обновлять состояние, в которое мы хотим вернуться, если встретим вложенный блок. Это состояние не всегда равно текущему.

Если третий автомат не попал в дьявольское состояние, управление снова передается лексеру. В противном случае, произошла синтаксическая ошибка, например, незакрытый BLOCK (без END). Кроме того, могут возникать и другие ошибки:

- несоответствие открывающих и закрывающих фигурных скобок;
- вложенные фигурные скобки;
- неизвестный символ.

На выходе синтаксический анализатор выдает ”OK”, если ошибок нет, в противном случае – место и тип ошибки.

7. Апробация

Тестирование скорости выполнения отдельных модулей проводилось при помощи массового выполнения необходимых действий, например, вставок в автомассив. Время работы модуля для каждого размера входных данных замеряется отдельно. Например, сначала мы вставляем n элементов и замеряем время, далее с нуля вставляем уже n + 10000 элементов и снова замеряем время и так далее. Другими словами, время замеряется не "по пути", а честно, для каждого набора элементов.

Замеры проводились на компьютере с процессором Intel Core i5 1,3 ГГц (i5-4250U). Можно приблизительно считать количество тактов (в млрд) равным количеству секунд.

7.1. Автоматный массив

Проблема памяти в автомассивах стоит очень остро. Это связано с тем, что таблица переходов является сильно разреженной. В районе 2 млн вставок в автомассив наблюдается резкий скачок. Он вызван тем, что оперативная память закончилась, и мы начинаем читать с диска.

В качестве подтверждающего эксперимента были дополнительно проведены замеры времени работы при 1 и 2Gb оперативной памяти на виртуальной машине. На рис. 14 видно, что при увеличении размера памяти граница применимости данного метода так же отодвигается. Это специфика данного метода.

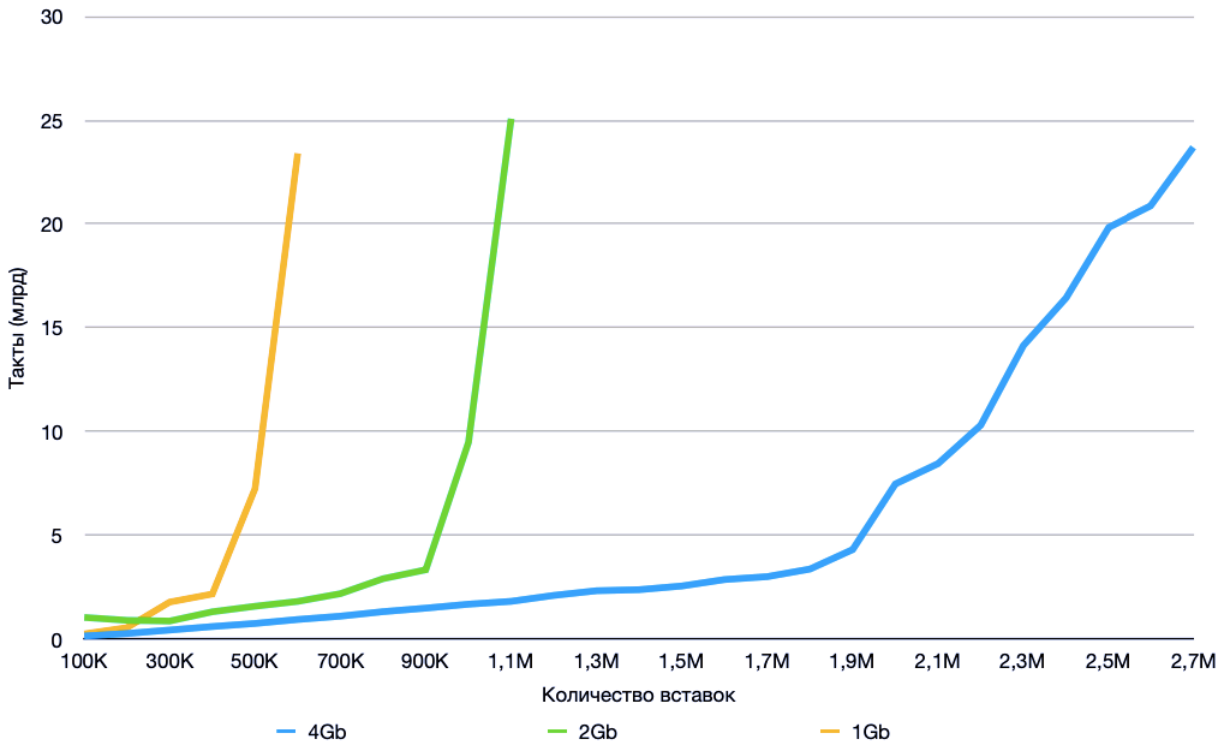

Рис. 14: Вставка строк длины 10 в автомассив при разных объемах RAM.

Зато вот где сейчас находится автомассив (autoarray) в сравнении с промышленными реализациями (см. рис. 15).

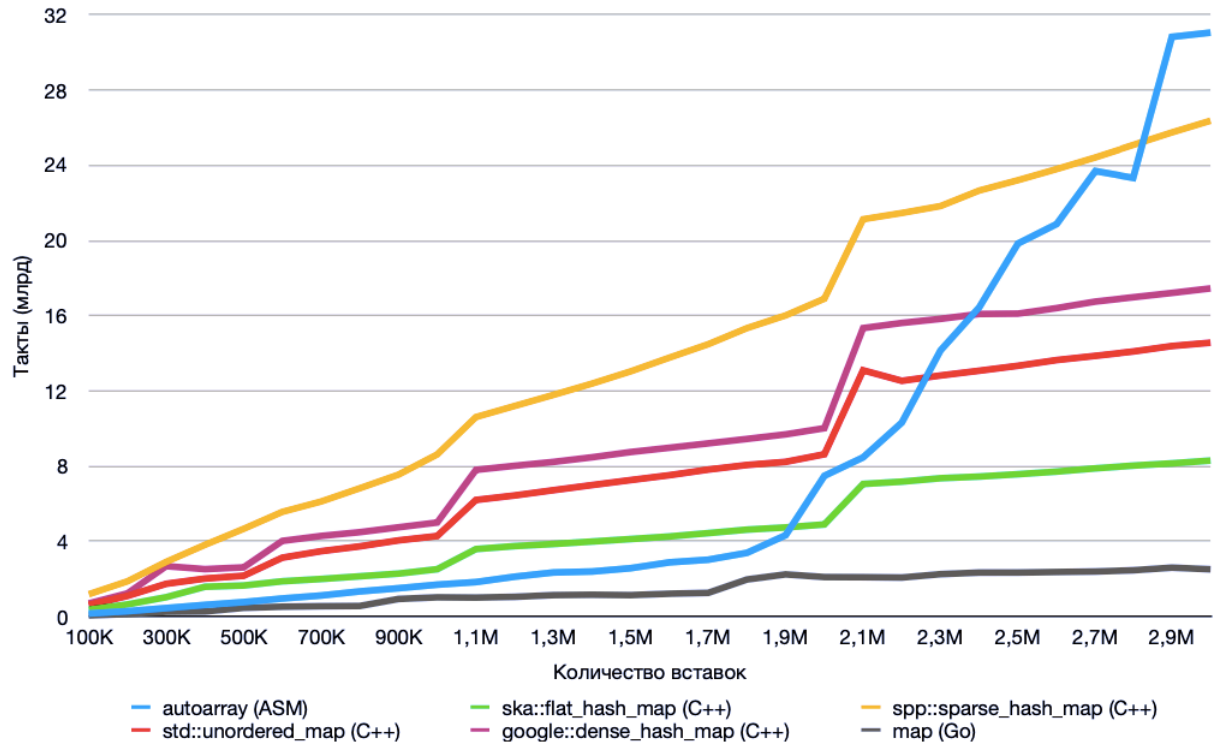

Рис. 15: Вставка строк длины 10 в ассоциативные массивы.

Важно отметить, что мы не затратили никаких усилий на оптимизацию, это наивная исследовательская реализация в лоб с тривиальным управлением памятью. Уже сейчас до нашей границы автомассив в 1,5-2 раза обгоняет самую быструю хеш-таблицу в C++, а если соптимизировать, то и Go мы тоже обгоним. Если на практике мы ограничиваемся 1 млн элементов, то автомассив нам очень подходит.

7.2. Утилита построения минимального ДКА по словарю

При вставке в автомат слова с одинаковыми префиксами не порождают новых веток, в отличие от ситуации с одинаковыми суффиксами. Минимизация по сути ”склеивает” одинаковые хвосты. Таким образом, сжатие будет более существенным, когда в словаре много слов с одинаковыми суффиксами. На рис. 16 показан хороший пример работы утилиты для набора женских имен, заканчивающихся на ”ina” (Polina, Irina и так далее). Здесь автомат без сжатия имеет 62 состояния, а минимальный автомат – 17.

```
; Adelina+8, Alina+7, Arina+5, Evelina+11, Irina+2, Karina+6, Kristina+4, Lina+9,
; Marina+3, Nina+10, Polina+1,

groups  =   20

        .data
char_n  db  65 dup(0),1,3 dup(0),2,3 dup(0),3,0,4,5,6,7,0,8,16 dup(0)
        db  9,2 dup(0),10,11,3 dup(0),12,2 dup(0),13,0,14,15,2 dup(0),16,17,18
        db  0,19,9 dup(0)
; A,E,I,K,L,M,N,P,a,d,e,i,l,n,o,r,s,t,v,
state_t dd  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 0
        dd  0,2,9,10,11,5,15,5,16,0,0,0,0,0,0,0,0,0,0 ; 1
        dd  0,0,0,0,0,0,0,0,0,3,0,0,5,0,0,5,0,0,0 ; 2
        dd  0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0 ; 3
        dd  0,0,0,0,0,0,0,0,0,0,0,0,5,0,0,0,0,0,0 ; 4
        dd  0,0,0,0,0,0,0,0,0,0,0,0,6,0,0,0,0,0,0 ; 5
        dd  0,0,0,0,0,0,0,0,0,0,0,0,0,0,7,0,0,0,0,0 ; 6
        dd  0,0,0,0,0,0,0,0,0,8,0,0,0,0,0,0,0,0,0 ; 7
        dd  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 8
        dd  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3 ; 9
        dd  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,5,0,0,0 ; 10
        dd  0,0,0,0,0,0,0,0,0,10,0,0,0,0,0,0,12,0,0,0 ; 11
        dd  0,0,0,0,0,0,0,0,0,0,0,0,13,0,0,0,0,0,0,0 ; 12
        dd  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,14,0,0 ; 13
        dd  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,5,0 ; 14
        dd  0,0,0,0,0,0,0,0,0,10,0,0,0,0,0,0,0,0,0,0 ; 15
        dd  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0 ; 16
state_f db  8 dup(0),1,8 dup(0)
```

Рис. 16: Пример автомата с минимизацией в 3,5 раза.

В случае, когда слов с одинаковыми суффиксами нет, выполняется только объединение допускающих состояний. Тогда для примера с мужскими именами (Alexander, Vadim, Boris, Ivan, Leonid, Nikita, Oleg, Pavel, Rodion, Fedor, Yaroslav) автомат без сжатия имеет 65 состояний, а минимальный автомат – 53. Таким образом, степень минимизации напрямую зависит от словаря.

На рис. 17 представлен график времени работы утилиты (autogen). Алгоритм Хопкрофта небыстрый: для словаря размером 2000 слов автомат будет строиться около 5 минут. Но на этапе построения интерпретатора время не так важно, в отличие от работы во времени исполнения программы. Если мы хотим использовать сгенерированный автомат не как распознаватель да/нет, а с возвратом атрибутов, то лучше использовать версию без минимизации. Поиск в минимизированном автомате будет равен поиску в боре (из-за подсчета номера слова), и мы потеряем скорость нашей автоматной техники.

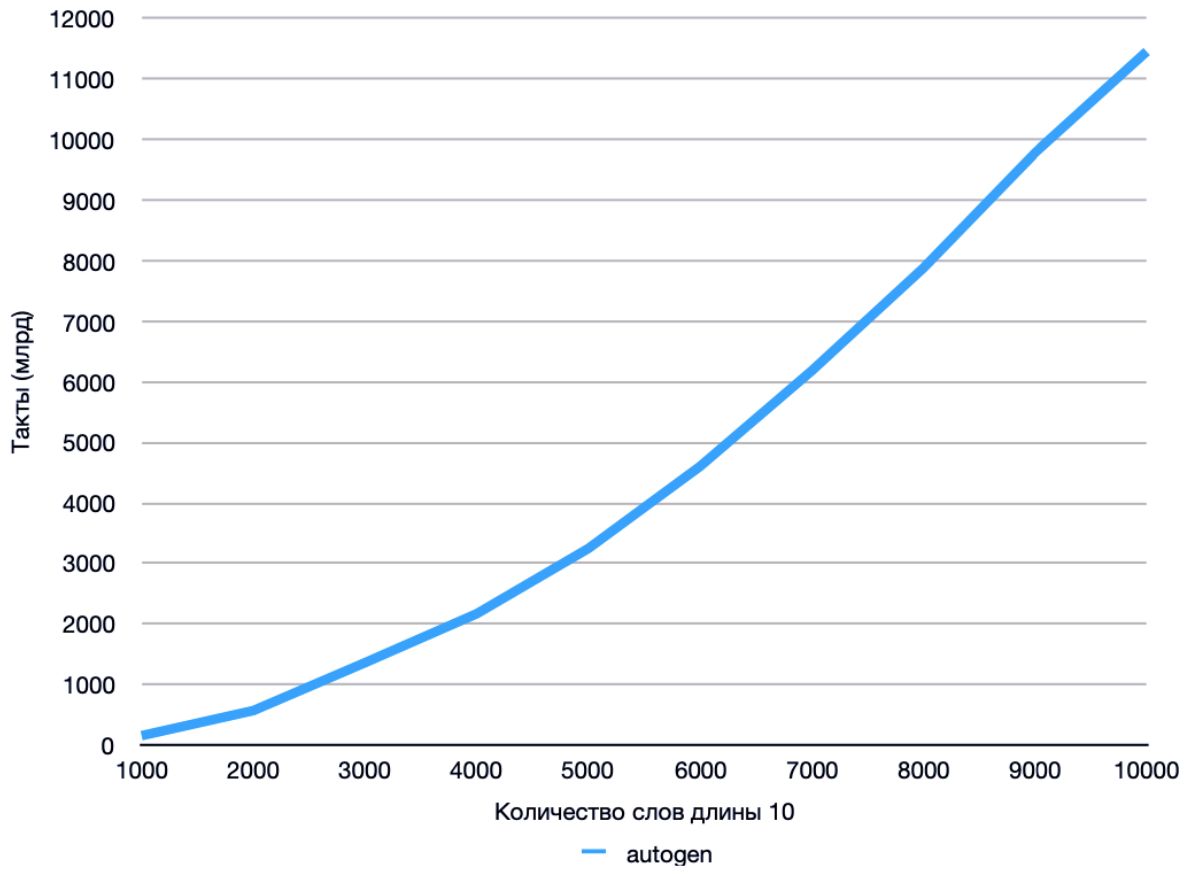

Рис. 17: Построение минимального ДКА по словарю.

7.3. Модуль для построения ДКА по регулярному выражению

На рис. 18 представлен график времени работы модуля построения минимального ДКА по регулярному выражению. Видно, что минимизация автомата занимает значительное время работы, особенно на небольших регулярных выражениях (до 2000 операций). Таким образом, имеет смысл сделать опцию включения/выключения минимизации автомата при работе с регулярными выражениями.

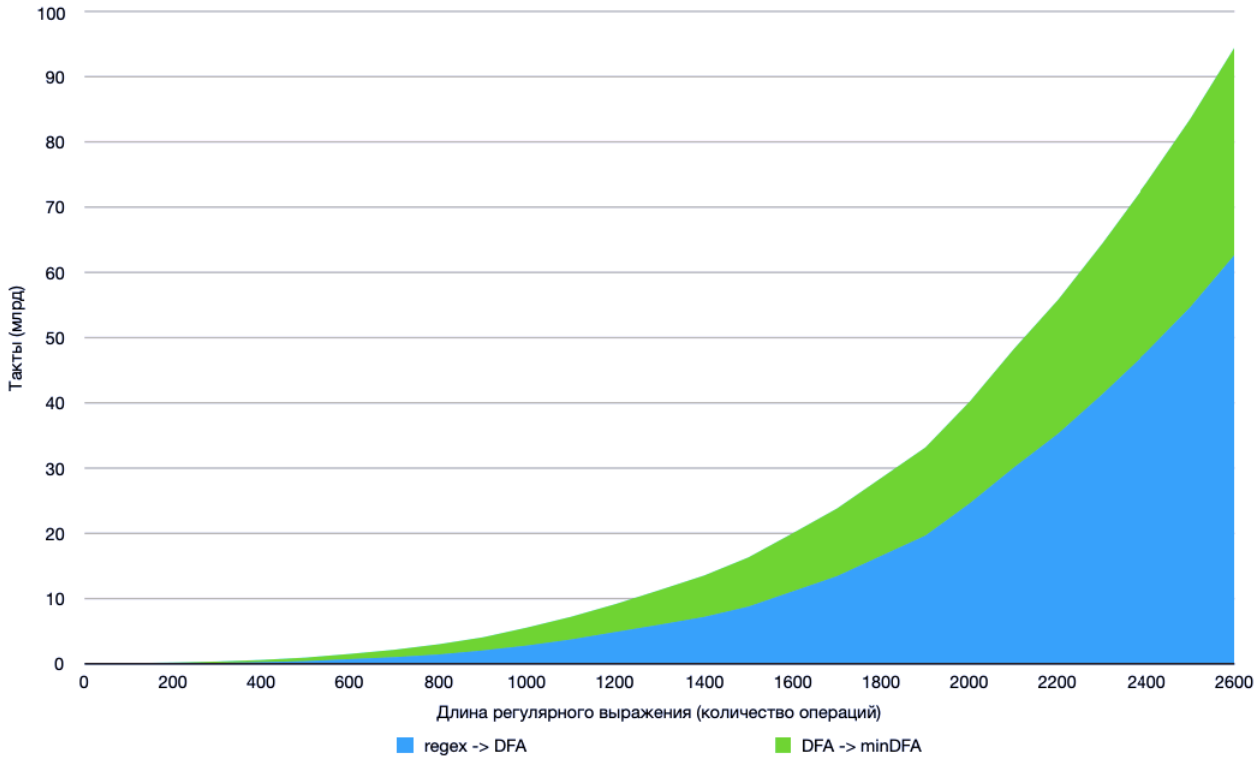

Рис. 18: Преобразование регулярного выражения в минимальный ДКА.

На рис. 19 изображены размеры автоматов, построенных по регулярному выражению. На графике наблюдается линейная зависимость размера минимального ДКА от размера НКА, хотя в теории говорится об экспоненциальной зависимости. Это связано с тем, что в построенном НКА совсем небольшое количество ε-переходов, а в переходах по символам алфавита вообще нет неопределенности.

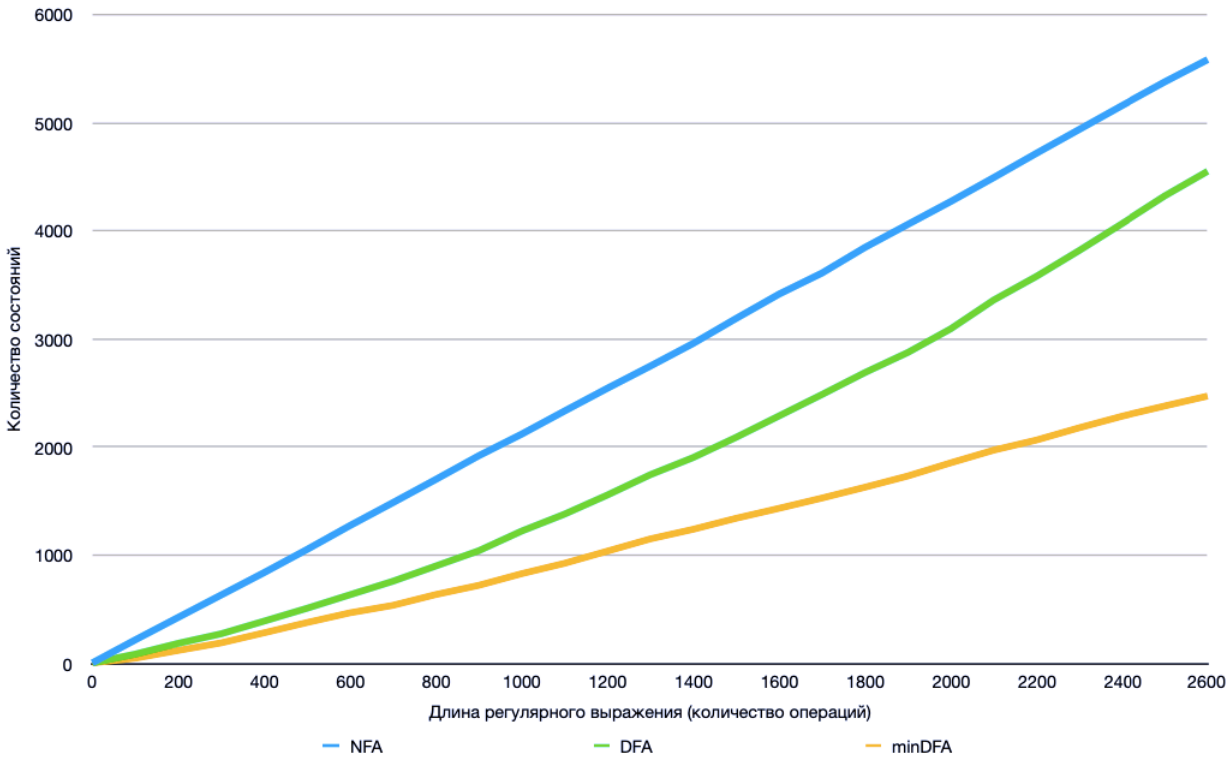

Рис. 19: Размеры автоматов, построенных по регулярному выражению.

Заключение

В ходе данной работы были получены следующие результаты.

1. Проанализированы следующие популярные реализации хеш-таблиц: map (Go), ska::flat_hash_map (C++), std::unordered_map (C++), spp::sparse_hash_map (C++), google::dense_hash_map (C++); проанализированы следующие алгоритмы минимизации ДКА: Хопкрофта, Глушкова, Дацюка–Михова.

2. Реализован ассоциативный массив в виде автоматного массива с хранением словаря в виде ДКА.

3. Разработана утилита построения ДКА по словарю служебных слов в двух вариантах: с минимизацией по алгоритму Хопкрофта (для больших словарей) и без минимизации автомата.

4. Реализован модуль для построения ДКА по регулярному выражению: разбор выражения VPA, построение НКА по алгоритму Мак-Нотона–Ямады–Томпсона, преобразование в ДКА построением подмножеств из НКА, минимизация ДКА по алгоритму Хопкрофта.

5. Реализован модуль для синтаксического анализа языка Macer с разбором VPA.

6. Выполнена апробация созданных средств.

    - Определены границы применимости автомассива: до 1,8 млн вставок строк.
    - Проведено сравнение по скорости работы автомассива с популярными реализациями хеш-таблиц: map (Go) и 4 реализации в C++. Наивная исследовательская реализация автомассива до границ применимости обгоняет самую быструю хеш-таблицу в C++ в 1,5-2 раза, но приблизительно во столько же раз уступает реализации map (Go).

- Определены границы применимости утилиты построения минимального ДКА по словарю: словари размером до нескольких тысяч строк.
- Проведено тестирование модуля построения ДКА по регулярному выражению: минимизация автомата составляет значительное время работы, поэтому она должна быть опциональной; наблюдается линейная зависимость размера минимального ДКА от размера НКА.

Список литературы

[1] Wikipedia. Interpreter (computing). Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Interpreter_(computing).

[2] Alur Rajeev, Madhusudan P. Visibly Pushdown Languages // Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing (STOC '04). June 2004. с. 202–211. URL: https://repository.upenn.edu/cgi/viewcontent.cgi?article=1174&context=cis_papers.

[3] Wikipedia. Associative array. Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Associative_array.

[4] Compilers: Principles, Techniques, and Tools (2nd Edition) / Alfred V. Aho, Monica S. Lam, Ravi Sethi [и др.]. Pearson Education, Inc, 2006.

[5] Wikipedia. DFA minimization. Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/DFA_minimization.

[6] Wikipedia. Regular expression. Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Regular_expression.

[7] Wikipedia. Pushdown automaton. Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Pushdown_automaton.

[8] V. Kumar, P. Madhusudan, M. Viswanathan. Visibly Pushdown Automata for Streaming XML // WWW 2007. May 8–12, 2007. URL: http://www2007.org/papers/paper788.pdf.

[9] Wikipedia. Hash table. Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Hash_table.

[10] Wikipedia. Trie. Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Trie.

[11] Wikipedia. Red-black tree. Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Red-black_tree.

[12] Wikipedia. AVL tree. Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/AVL_tree.

[13] Golang. Github. URL: https://github.com/golang/go/blob/master/src/runtime/map.go.

[14] Shlegeris B. Hash map implementations in practice. Buck Shlegeris. URL: http://shlegeris.com/2017/01/06/hash-maps.

[15] Goetghebuer-Planchon T. Benchmark of major hash maps implementations. Tessil. URL: https://tessil.github.io/2016/08/29/benchmark-hopscotch-map.html.

[16] Scarupke M. I Wrote The Fastest Hashtable. Probably Dance. URL: https://probablydance.com/2017/02/26/i-wrote-the-fastest-hashtable/.

[17] Celis Pedro. Robin Hood hashing // (Technical report). Computer Science Department, University of Waterloo. 1986. URL: https://cs.uwaterloo.ca/research/tr/1986/CS-86-14.pdf.

[18] E.F. Moore. Gedanken-experiments on sequential Machines // Automata Studies,Annals of Mathematical Studies. — Princeton, N.J.: Princeton University Press. 1956. C. 129–153.

[19] Watson B. W. Taxonomies and toolkits of regular language algorithms // Eindhoven: Technische Universiteit Eindhoven. 1995. URL: https://doi.org/10.6100/IR444299.

[20] Efficient Deterministic Finite Automata Minimization Based on Backward Depth Information / Liu D, Huang Z, Zhang Y [и др.] // Gui-Quan Sun, Shanxi University, CHINA. 2016. URL: https://doi.org/10.1371/journal.pone.0165864.

[21] Hopcroft John E., Motwani Rajeev, Ullman Jeffrey D. Introduction to automata theory, languages and computation (2nd ed.). Boston, MA, USA: Addison-Wesley Publishing Corporation, 2001.

[22] J. Hopcroft. An n log n algorithm for minimizing states in a finite automaton // Theory of Machines and Computations. 1971. С. 189–196. URL: https://doi.org/10.1016/B978-0-12-417750-5.50022-1.

[23] Incremental Construction of Minimal Acyclic Finite-State Automata / Jan Daciuk, Bruce W. Watson, Stoyan Mihov [и др.] // Association for Computational Linguistics. 2000. Т. 26, № 1. URL: https://www.aclweb.org/anthology/J00-1002.pdf.

[24] Глушков В. М. Синтез цифровых автоматов. Государственное издательство физико-математической литературы, Москва, 1962.

[25] Daciuk Jan. Incremental Construction of Finite-State Automata and Transducers, and their Use in the Natural Language Processing // PhD thesis, Technical University of Gdańsk, Poland. 1998. URL: http://www.pg.gda.pl/~jandac/thesis/thesis.html.