

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(СПбГУ)

Образовательная программа бакалавриата «Науки о данных»



Отчёт по практике
«Учебная практика (проектно-технологическая практика)»

Выполнил студент 2 курса бакалавриата
(группа 22.Б05-мкн)
Ратьков Андрей Игоревич

Научный руководитель:
д.ф.-м.н. Охотин Александр Сергеевич

Санкт-Петербург
2024

Содержание

| | | |
|----------|--|-----------|
| 1 | Описание алгоритма Хопкрофта | 3 |
| 1.1 | Вступление | 3 |
| 1.2 | Используемые объекты и подготовка к минимизации | 3 |
| 1.3 | Несколько утверждений | 4 |
| 1.4 | Псевдокод | 5 |
| 1.5 | Корректность алгоритма | 6 |
| 1.6 | Асимптотика алгоритма | 6 |
| 2 | Реализация алгоритма | 7 |
| 2.1 | Класс DFA | 8 |
| 2.2 | Класс NFA | 11 |
| 2.3 | Взаимодействие с пользователем | 11 |
| 3 | Анализ работы программы | 13 |
| 3.1 | Сравнение времени работы алгоритмов Хопкрофта и Мура | 13 |
| 3.2 | Расходуемая память | 14 |
| 4 | Заключение | 14 |

1 Описание алгоритма Хопкрофта

1.1 Вступление

На входе подаётся детерминированный конечный автомат (DFA) $A(\Sigma, Q, q_0, \delta, F)$. Σ — входной алфавит — конечный набор символов, Q — конечное множество состояний, $q_0 \in Q$ — начальное состояние, $\delta : Q \times \Sigma \rightarrow Q$ — правила переходов, $F \subset Q$ — множество принимающих состояний. Алгоритм минимизации преобразует автомат до нового минимального автомата A' (автомата с минимальным числом состояний), распознающего тот же язык.

Для начала удаляются состояния, в которые нельзя попасть из начального состояния q_0 . Остальные состояния разбиваются на классы эквивалентности по следующему отношению:

Определение 1. Состояния q_1 и q_2 эквивалентны ($q_1 \sim q_2$), если множество слов, принимаемых из состояния q_1 равно множеству слов, принимаемых из состояния q_2 .

По теореме Мура, получаемый при этом автомат минимальный. Алгоритм Мура минимизации конечных автоматов работает за время $O(n^2)$, где $n = |Q|$ — количество состояний в исходном автомате A . Ниже будет рассмотрен алгоритм Хопкрофта [1], делающий то же самое, но с лучшей асимптотикой $O(n \log n)$.

1.2 Используемые объекты и подготовка к минимизации

Первым делом покажем, что удаление недостижимых из q_0 состояний можно сделать линейно. Будем красить состояния в три цвета: черный, серый, белый. Белый будет означать, что алгоритм не достиг ещё этого состояния, серый — достиг, но не прошёл из него в соседние состояния (т.е. в состояния, в которые можно попасть из текущего по какому-то символу), чёрный — что состояние посещено и достигнуты соседние с ним состояния. В начале алгоритм красит начальное состояние в серый, остальные — в белый, начальное состояние кладёт в очередь (очередь состоит из серых состояний). Далее, пока очередь не пуста, из неё берется серое состояние, перекрашивается в чёрный. Перебираются его соседние состояния, и те из них, которые белые, перекрашиваются в серый и добавляются в очередь. Работает это за время $O(|\Sigma|n)$, так по каждому из рёбер графа, задаваемом A , алгоритм пройдёт не более 1 раза, сделает не более n добавлений и вытаскиваний элемента из очереди (каждая из этих операций — $O(1)$).

В реализации алгоритма понадобится отображение $\delta^{-1} : Q \times \Sigma \rightarrow 2^Q$: каждому состоянию $q \in Q$ и символу $a \in \Sigma$ сопоставляется множество состояний, из которых можно прийти в q по a ($\delta^{-1}(q, a) = \{s \in Q \mid \delta(s, a) = q\}$). Поскольку

$$\sum_{q \in Q} \sum_{a \in A} \delta^{-1}(q, a) = |\Sigma|n$$

(обе части — количество переходов в графе, задаваемом A), для δ^{-1} построение занимает $O(|\Sigma|n)$ времени и $O(|\Sigma|n)$ памяти.

Как и в алгоритме Мура, алгоритм Хопкрофта будет разбивать состояния автомата A на блоки (раскрашивать состояния в цвета), пока в конце не окажется, что разбиение на блоки (цвета) не удовлетворяет следующему условию: два состояния находятся в одном блоке (помечены одним цветом) тогда и только тогда, когда они эквивалентны. Тогда будет получено искомое разбиение на классы эквивалентности.

На каждой итерации алгоритма все состояния покрашены в k цветов (разбиты на k блоков). Блоки будем обозначать $B(0), B(1), \dots, B(k-1)$. Чтобы быстро уметь итерироваться (перебирать все элементы) по какому-нибудь блоку $B(j)$, будем представлять, что состояния находятся в каком-то порядке и для каждого состояния мы храним номер следующего состояния такого же цвета и предыдущего состояния такого же цвета. Также для каждого цвета j удобно хранить первое состояние цвета j (обозначим его за $first(j)$). Тогда проитерироваться по всем состояниям $B(j)$ несложно: для начала обращаемся к состоянию $first(j)$, а затем из состояния идём в следующее состояние этого цвета (пока не достигнем последнего состояния в блоке). При такой конструкции итерирование по всякому блоку $B(j)$ (перебор всех состояний из $B(j)$) занимает $O(|B(j)|)$ времени.

Также для каждого цвета j и символа a будем помнить множество всех состояний цвета j , в которые можно попасть по символу a . Будем обозначать это множество как $\hat{B}(B(j), a)$. Нам понадобится итерироваться по таким множествам $\hat{B}(B(j), a)$. Можно считать, что состояния в каждом множестве $\hat{B}(B(j), a)$ — элементы двусвязного списка. А именно, для каждого состояния $q \in B(j)$, для каждого символа a , если в q можно попасть по a , будем хранить номер следующего состояния из $\hat{B}(B(j), a)$ и предыдущего (если таковые есть). Также для каждого множества $\hat{B}(B(j), a)$ будем хранить первого состояния из этого множества (обозначим его за $first(j, a)$). Тогда итерирование по множеству $\hat{B}(B(j), a)$ выглядит так: сначала обращаемся к $first(j, a)$, а затем из состояния переходим к следующему, которое лежит в $\hat{B}(B(j), a)$ (пока можем перейти, то есть пока следующее есть). При такой конструкции итерирование по состояниям из всякого множества $\hat{B}(B(j), a)$ занимает $O(|\hat{B}(B(j), a)|)$ времени.

На каждом этапе алгоритма мы будем поддерживать, что если у состояний разный цвет (они в разных блоках), то из них принимаются разные языки. То есть для всяких блоков $B(i), B(j)$, ($i \neq j$), для любых двух состояний из них $q_1 \in B(i)$, $q_2 \in B(j)$ существует разделяющая строка $w \in \Sigma^*$ (то есть такая, w — принимаемая

из q_1 , но не принимаемая из q_2 , или наоборот — принимаемая из q_2 , но не принимаемая из q_1). Например, в начале все принимающие состояния покрашены в цвет 0 (блок $B(0)$), отвергающие — в цвет 1 (блок $B(1)$). Разделяющая строка для этих блоков — ϵ .

1.3 Несколько утверждений

Лемма 1. Пусть все состояния разбиты на блоки: из состояний разных блоков принимаются разные языки. Зафиксируем блок $B(i)$ и символ $a \in \Sigma$. Рассмотрим произвольный блок $B(j)$. Определим

$$B'(j) = \{t \in B(j) \mid \delta(t, a) \in B(i)\},$$

$$B''(j) = \{t \in B(j) \mid \delta(t, a) \notin B(i)\}.$$

Тогда $\forall q_1 \in B'(j) \forall q_2 \in B''(j)$ из состояний q_1 и q_2 принимаются разные языки, то есть $q_1 \not\sim q_2$.

Доказательство. Пусть $p_1 = \delta(q_1, a)$. $p_1 \in B(i)$, так как $q_1 \in B'(j)$. Пусть $p_2 = \delta(q_2, a)$. $p_2 \notin B(i)$, так как $q_2 \in B''(j)$.

Состояния p_1 и p_2 лежат в разных блоках, поэтому существует строка w , принимаемая ровно из одного из них. Тогда строка aw принимается ровно из одного из состояний q_1, q_2 (w принимается из $p_1 \iff aw$ принимается из q_1 ; w принимается из $p_2 \iff aw$ принимается из q_2). \square

Отсюда понятно, что если блок $B(j)$ заменить на блоки $B'(j)$ и $B''(j)$ (если они оба непусты), то по-прежнему в новом разбиении на блоки выполнено, что состояния в разных блоках неэквивалентны.

Определение 2. Пусть все состояния разбиты на k блоков $B(0), B(1), \dots, B(k-1)$, зафиксирован блок $B(i)$ и символ $a \in \Sigma$. Для каждого блока $B(j)$ определим множества $B'(j)$ и $B''(j)$ как в теореме выше. Получим новое разбиение: для каждого $j \in \{0, \dots, k-1\}$, если оба множества $B'(j)$ и $B''(j)$ непусты, заменим блок $B(j)$ на блоки $B'(j)$ и $B''(j)$, тем самым получим новое разбиение на блоки — измельчение старого разбиения. Такую операцию измельчения разбиения состояний на блоки будем называть **измельчением с помощью блока $B(i)$ и символа a** . Измельчение будем называть **бесполезным**, если оно не увеличивает число блоков (для всякого блока $B(j)$ один из блоков $B'(j)$ и $B''(j)$ пуст), или **полезным** в противном случае.

Суть алгоритма — сначала разбить состояние на два блока (принимающие и отвергающие состояния). А дальше много раз измельчать разбиение: выбирать какой-то блок $B(i)$ и какой-то символ a и измельчать с помощью них.

Утверждение 1. Каждый блок состоит либо только из принимающих состояний, либо только из отвергающих.

Доказательство. Индукция по количеству блоков. База: 2 блока: $B(0)$ — все принимающие состояния, $B(1)$ — все отвергающие. Переход: количество блоков увеличивается, когда какой-то блок разбивается на два других. Поскольку разбиваемый блок состоял только из принимаемых состояний или только из отвергающих, то и новые два блока тоже. \square

Утверждение 2. Если до какого-то измельчения любые два эквивалентных состояния находились в одном блоке, то и после измельчения тоже.

Доказательство. Пусть эквивалентные состояния q_1 и q_2 находятся в блоке $B(j)$ и происходит измельчение по блоку $B(i)$ и символу a . Возможны два варианта: либо оба $\delta(q_1, a)$ и $\delta(q_2, a)$ лежат в $B(i)$, тогда q_1 и q_2 лежат в $B'(j)$, либо оба $\delta(q_1, a)$ и $\delta(q_2, a)$ не лежат в $B(i)$, тогда q_1 и q_2 лежат в $B''(j)$. Третий вариант, когда только одно из $\delta(q_1, a)$ и $\delta(q_2, a)$ лежит в $B(i)$, не достигается: тогда, по теореме 1, q_1 и q_2 неэквивалентны — противоречие. \square

Лемма 2. Пусть все состояния разбиты на непустые блоки, причём для всякого блока $B(i)$ и символа $a \in \Sigma$ верно, что измельчение по блоку $B(i)$ и символу a бесполезно. Тогда состояния внутри каждого блока эквивалентны.

Доказательство. Предположим, что существуют пары состояний q_1 и q_2 лежащих в одном блоке, которые не эквивалентны. Каждой такой паре сопоставим кратчайшую разделяющую строку. Понятно, что каждая разделяющая строка не является пустой: тогда бы одно из состояний было бы принимающим, а другое нет, что противоречило бы лемме 1.

Рассмотрим пару неэквивалентных состояний q_1 и q_2 из какого-то блока, которым была сопоставлена самая короткая строка (назовём её w). Как мы уже выяснили, w непуста, а значит представима в виде $w = au, a \in \Sigma, u \in \Sigma^*$. Рассмотрим $p_1 = \delta(q_1, a)$ и $p_2 = \delta(q_2, a)$. Состояния p_1 и p_2 лежат в одном блоке (это следует из условия теоремы: если это не так, то измельчение по блоку, содержащему одно из них и по символу a будет полезным — после него q_1 и q_2 окажутся в разных блоках, противоречие).

Если u является разделяющей для состояний p_1 и p_2 , то возникает противоречие с выбором пары q_1, q_2 и строки w — кратчайшей разделяющей строки ($|u| < |w|$). Иначе, оба p_1 и p_2 принимаются/отвергаются по строке u . Тогда оба состояния q_1 и q_2 принимаются/отвергаются по строке $w = au$, а значит она неразделяющая, противоречие. \square

Алгоритм измельчает разбиение, пока может: пытается найти блок $B(i)$ и символ a , измельчение с помощью которых полезно. Понятно, что он завершается, так как количество блоков в конце станет не более n , а значит полезных измельчений случится не более чем $n - 2$. По лемме 2 если какие-то состояния эквивалентны, то они всегда будут оставаться в одном блоке и, следовательно, в конечном разбиении они тоже будут лежать в одном блоке. По теореме 2, если два состояния неэквивалентны, то после завершения алгоритма они будут лежать в разных блоках.

Лемма 3. Пусть состояния разбиты на блоки и происходит измельчение с помощью блока $B(i)$ и символа $a \in \Sigma$. Далее происходит сколько-то каких-то других измельчений, и оказывается, что получилось новое измельчение, в котором блок $B(i)$ заменился на $m \geq 1$ блоков $B(i_1), \dots, B(i_m)$ и произошли измельчения по блокам $B(i_1), \dots, B(i_{m-1})$ и символу a . Тогда, после этого, измельчение по блоку $B(i_m)$ и символу a бесполезно.

Доказательство. Рассмотрим произвольный блок $B(l)$, в котором есть состояние $s \in B(l)$, из которого по a можно попасть в блок $B(i_m)$ ($\delta(s, a) \in B(i_m)$). Покажем, что тогда из всех состояний блока $B(l)$ по a можно попасть в $B(i_m)$. Рассмотрим произвольное состояние: $t \in B(l)$.

Пусть $B(k)$ — блок, в который из t можно попасть по a . Если $k \in \{i_1, \dots, i_{m-1}\}$ (по блоку $B(k)$ и символу a было измельчение), то состояния s и t до, и после измельчения по блоку $B(k)$ и символу a оставались внутри одного блока, хотя ровно из одного из них по a можно попасть в $B(k)$ — противоречие. Если $k \notin \{i_1, \dots, i_m\}$, то состояния s и t до, и после измельчения по блоку $B(i)$ и символу a оставались в одном блоке, хотя ровно из одного из них по a можно было попасть в $B(i)$ по символу a — противоречие. \square

Алгоритм будет хранить для каждого $a \in \Sigma$ множество $L(a)$ номеров блоков, из которых впоследствии будет выполняться измельчение по символу a .

Алгоритм такой. В начале для всякого $a \in \Sigma$ определяем $L(a) = \{\}$. Это множество номеров блоков, с помощью которых и символа a будут происходить измельчения. После разделения состояний на два блока — $B(0)$ (принимающие) и $B(1)$ (отвергающие), для каждого $a \in \Sigma$ в $L(a)$ добавляем 0, если $|\hat{B}(B(0), a)| \leq |\hat{B}(B(1), a)|$, иначе 1. Пока не все $L(a)$ пусты, алгоритм берёт и вытаскивает какой-то номер блока i из какого-то $L(a)$. Для всех $B(j)$ алгоритм, если оба множества $B'(j)$ и $B''(j)$ из теоремы 1 непусты, делит $B(j)$ на два новых блока $B(j_1) = B'(j)$ и $B(j_2) = B''(j)$. Для всякого символа $c \in \Sigma$ если в $L(c)$ был блок $B(j)$, теперь вместо него будут оба блока $B(j_1)$ и $B(j_2)$, иначе туда добавится ровно один из них: если $|\hat{B}(B(j_1), a)| \leq |\hat{B}(B(j_2), a)|$, то $B(j_1)$, иначе $B(j_2)$. Таким образом шаблон алгоритма выглядит так:

1.4 Псевдокод

```

1: while  $\exists a : L(a) \neq \emptyset$  do
2:   select  $a$  and  $i \in L(a)$ , delete  $i$  from  $L(a)$ 
3:   for all blocks  $B(j)$  do
4:     if  $\exists t \in B(j) : \delta(t, a) \in B(i)$  then
5:       divide  $B(j)$  into  $B'(j) = \{t \in B(j) | \delta(t, a) \in \hat{B}(B(i), a)\}$ 
6:       and  $B''(j) = \{t \in B(j) | \delta(t, a) \notin \hat{B}(B(i), a)\}$ 
7:       let  $k$  be a number for a new block
8:        $B(j) = B''(j)$ ,  $B(k) = B''(j)$ 
9:       for  $c \in \Sigma$  do
10:        if  $j \in L(c)$  then
11:           $L(c).add(k)$ 
12:        else if  $|\hat{B}(B(j), c)| \leq |\hat{B}(B(k), c)|$  then
13:           $L(c).add(j)$ 
14:        else
15:           $L(c).add(k)$ 
16:        end if
17:      end for
18:    end if
19:  end for
20: end while

```

▷ We need to divide this block into 2 blocks

Написанный выше псевдокод несложно улучшить по асимптотике, используя уже построенные структуры. Во-первых, на каждой итерации алгоритма (при каждом измельчении), создадим множество R , состоящее из

номеров блоков, которые надо разделить. Для этого в начале нужно его инициализировать $R = \{\}$, а дальше для всякого $t : \delta(t, a) \in B(i)$ добавим в R номер блока, в котором лежит t . То есть, это делается за $O(|\{t | \delta(t, a) \in B(i)\}|)$.

Для каждого номера j из R обозначим за $f(j)$ номер нового блока, в который мы перенесём состояния блока $B(j)$, из которых по a можно попасть в $B(i)$. То есть блок $B(j)$ разделится на блоки $B(j)$ и $B(f(j))$. При этом, возможно, блок $B(j)$ после этого останется пустым (в случае, если до разделения блока $B(j)$ из всех его состояний по a можно было попасть в $B(i)$).

Далее будем снова проходить по состояниям $t : \delta(t, a) \in B(i)$ и менять номер блока каждого из них (если он был j , то станет $f(j)$). Также, поскольку для каждого блока мы храним $|B(j)|$ и $|\hat{B}(B(j), a)|$, будем поддерживать изменения этих параметров для блоков $B(j)$ и $B(f(j))$. И, ещё, поскольку для каждого состояния мы храним указатели на следующее и предыдущее состояния из этого же блока, а также (для каждого $a \in \Sigma$, если в состояние можно попасть по a) указатели на следующее и предыдущее состояние этого же блока, в которые можно попасть по a , эту информацию нужно тоже обновить. В общем, всю $O(|\Sigma|)$ информацию, которую мы храним в каждом состоянии, нужно будет поменять. Это можно сделать за время $O(|\Sigma|)$ (поскольку у нас двусторонние списки, вставка в конец и вытаскивание элемента делаются за $O(1)$).

Итого, одна итерация внешнего цикла while (строка 1 псевдокода) занимает $O(|\Sigma| \cdot |\{q | \delta(q, a) \in B(i)\}|)$ времени — все состояния из множества $\{q | \delta(q, a) \in B(i)\}$ вытаскиваются из своих блоков и кладутся в новые блоки.

1.5 Корректность алгоритма

Лемма 4. *Алгоритм, описанный выше, корректен.*

Доказательство. Достаточно показать, что для любых состояний $q_1, q_2 \in Q$, если $q_1 \sim q_2$, то q_1 и q_2 лежат в одном блоке, иначе в разных.

Первое следует из леммы 2: действительно, в начальном разбиении на $B(0)$ и $B(1)$, если пара состояний эквивалентны, то оба лежат в одном блоке, а далее, в ходе измельчений, они будут продолжать лежать в одном блоке.

Теперь покажем, что если состояния неэквивалентны, то они окажутся в разных блоках. Предположим, что существуют пары состояний q_1 и q_2 такие что $q_1 \not\sim q_2$ и при этом они лежат в одном блоке $B(i)$ в конце работы алгоритма. Каждой паре таких состояний сопоставим кратчайшую разделяющую их строку.

Среди всех таких пар состояний q_1 и q_2 выберем ту, для которой сопоставленная кратчайшая строка является самой короткой. Пусть теперь это пара p_1 и p_2 , а кратчайшая разделяющая их строка — $w \in \Sigma^*$.

Поскольку p_1 и p_2 лежат в одном блоке $B(i)$, они либо оба принимающие, либо оба отвергающие, значит, всякая разделяющая их строка имеет положительную длину. Пусть a — первый символ строки w : $w = au$, где $u \in \Sigma^*$.

Пусть $r_1 = \delta(p_1, a), r_2 = \delta(p_2, a)$. Блок, в котором находится состояние r_1 , назовём $B(j_1)$, а в котором находится состояние r_2 — $B(j_2)$.

Заметим, что $j_1 \neq j_2$: иначе, если $j_1 = j_2$, то получится, что r_1 и r_2 — состояния из одного блока, и при этом они неэквивалентны (u — разделяющая их строка). Тогда у этой пары состояний кратчайшая строка короче ($|u| < |w|$) — противоречие с выбором p_1 и p_2 .

Рассмотрим итерацию алгоритма, после которой состояния r_1 и r_2 оказались в разных блоках, пусть это блоки $B(l_1)$ и $B(l_2)$. После этого не менее, чем одно из чисел l_1, l_2 было помещено в $L(a)$. Не умаляя общности, будем считать, что это l_1 . Начиная с этого момента, состояние r_1 находилось в одном из блоков, упомянутых в списке $L(a)$. Поскольку алгоритм завершился после того, как список $L(a)$ опустел, был хотя бы один момент, когда из $L(a)$ вытащили номер m блока, содержащего состояние r_1 ($r_1 \in B(m)$). Рассмотрим такой момент.

Происходит измельчение по блоку $B(m)$ и символу a . Отметим, что $r_1 \in B(m)$; $r_2 \notin B(m)$, так как $B(m) \subset B(l_1)$ и $r_2 \notin B(l_1)$. Тогда получается, что

$$\delta(p_1, a) = r_1 \in B(m)$$

$$\delta(p_2, a) = r_2 \notin B(m)$$

Состояния p_1 и p_2 находятся в одном блоке. Значит, после измельчения, они должны оказаться в разных блоках. Тогда и по завершении алгоритма они будут в разных блоках. Противоречие. \square

1.6 Асимптотика алгоритма

Утверждение 3. *Пусть $a \in \Sigma, q \in Q$. За всю работу алгоритма, из $L(a)$ извлекался номер блока, содержащего состояние q , не более $\log(n)$ раз.*

Доказательство. Пусть k — количество раз, когда из $L(a)$ извлекался номер блока, содержащего q .

Пусть x_1 — размер этого блока при первом извлечении, x_2 — при втором, и так далее, x_k — размер при последнем извлечении из $L(a)$. Покажем, что $\forall j < k : x_j \geq 2 \cdot x_{j+1}$.

Рассмотрим j -ое извлечение. При нём состояние q перестало находиться в каком-либо блоке, упомянутом в $L(a)$. После этого, в какой-то момент, в $L(a)$ был добавлен блок, содержащий q . Рассмотрим ближайший такой момент.

Он соответствует 10 – 16 строкам псевдокода (происходит $L(a).add(m)$, где m — номер блока, в котором находится состояние q при этом добавлении). Это происходит после разделения блока (строки 5 – 8), в котором лежало состояние q (пусть размер разделившегося блока y_{j+1}). При этом номер этого блока не находился в $L(a)$. Значит, добавлению соответствуют строки 12 – 15 псевдокода (то есть из двух образовавшихся блоков добавили только наименьший). Таким образом, справедливо:

$$x_j \geq y_{j+1} \geq 2 \cdot x_{j+1} \Rightarrow x_j \geq 2 \cdot x_{j+1}$$

Аналогично можно отметить, что $x_1 \leq n/2$, так как $2 \cdot x_1 \leq y_1 \leq n$.

Отсюда следует, что

$$n/2 \geq x_1 \geq 2^{k-1} \cdot x_k$$

Откуда, поскольку $x_k \geq 1$, $n \geq 2^k \Leftrightarrow k \leq \log(n)$. □

Утверждение 4. Рассмотрим $t \in Q$. За всю работу алгоритма, состояние t меняло блок (то есть извлекалось из старого блока в новый) не более $|\Sigma| \cdot \log(n)$ раз.

Доказательство. Пусть $\Sigma = \{a_1, a_2, \dots, a_{|\Sigma|}\}$; $q_i = \delta(t, a_i)$ для всех $i \in \{1, \dots, |\Sigma|\}$. t меняет блок на какой-то итерации алгоритма, если на этой итерации алгоритма из $L(a_i)$ было извлечён блок, содержащий q_i (для некоторого $i \in \{1, \dots, |\Sigma|\}$). Таких итераций, по лемме 3, не более $|\Sigma| \cdot \log(n)$. □

Лемма 5. Алгоритм, описанный выше, имеет временную асимптотику $O(|\Sigma|^2 n \log(n))$.

Доказательство. Удаление недостижимых из q_0 состояний, а также разбиение состояний на блоки $B(0)$ и $B(1)$, работают за линейное время, как было показано выше.

Всего состояний n , значит, по лемме 4, перемещений состояний из одного блока в новый, было всего не более $|\Sigma| n \log(n)$. Одно перемещение занимает $O(|\Sigma|)$ времени (так как нужно поменять номер блока, в котором находится состояние, и, следовательно, информацию о том, какое следующее и предыдущее состояние в том же новом блоке, а также следующее и предыдущее состояния, в которые можно попасть по символу c (для всех $c \in \Sigma$) — это нужно для того, чтобы двусвязные списки оставались корректными).

Итого, все перемещения состояний требуют $O(|\Sigma|^2 n \log(n))$ времени.

Осталось показать, сколько было добавлений и извлечений номеров блоков из $L(c)$ ($c \in \Sigma$). Зафиксируем какое-то $c \in \Sigma$.

Отметим, что можно в $L(c)$ добавлять номера только непустых блоков (действительно, после этого извлекать номера пустых блоков бессмысленно — измельчение с помощью них будет бесполезным). Итак, в $L(c)$ добавляются только блоки положительного размера. Это значит, что в строках 5-6 псевдокода перед этим произошло разбиение $B(j)$ на непустые $B'(j)$ и $B''(j)$. Поскольку блоков положительного размера не более чем n в конце работы алгоритма, то разбиений блока на два непустых, случалось не более $n - 1$ раза за всю работу алгоритма. То есть в $L(c)$ добавляли (и, следовательно, извлекали) номера не более n раз (для фиксированного $c \in \Sigma$). Значит, всего добавлений и извлечений номеров из L было $O(|\Sigma| n)$ раз. Каждое добавление — $O(1)$ по времени (добавление в конец списка: 11, 13, 15 строки псевдокода), каждое извлечение — $O(|\Sigma|)$ времени (строка 2 псевдокода), так как в ней нужно найти первый непустой список $L(a)$ и из него извлечь элемент.

Итого, каждый вид действий, предпринимаемых алгоритмом, занимает не более чем $O(|\Sigma|^2 n \log(n))$ времени. Значит, и алгоритм работает с асимптотикой $O(|\Sigma|^2 n \log(n))$. □

Если считать, что $|\Sigma|$ — некоторая константа, то асимптотика алгоритма Хопкрофта равна $O(n \log(n))$.

2 Реализация алгоритма

Проект реализован в нескольких файлах:

- Директория `src/` — содержит файлы реализаций функций и методов классов:
 - ❖ `dfa_methods.cpp` — реализация методов класса DFA.
 - ❖ `nfa_methods.cpp` — реализация методов класса NFA.
 - ❖ `dfa_build.cpp` — реализация функций обработки команд пользователя и инициализации DFA, исходя из этих команд.
 - ❖ `main.cpp` — работа с пользователем: получение DFA, его минимизация, вывод DFA.

- Директория `include/` — содержит заголовочные файлы.
 - ❖ `dfa_class.h` — объявление класса DFA.
 - ❖ `nfa_class.h` — объявление класса NFA.
- Директория `obj/` — содержит объектные файлы (скомпилированные `.cpp` файлы)
- Файл `minimizer` — скомпилированный проект.
- `Makefile` — файл для сборки.

Как собрать проект: зайти в главную директорию, запустить команду `make`. Как запустить проект: запустить исполняемый файл `minimizer` с несколькими аргументами (подробнее — см. раздел 2.3).

2.1 Класс DFA

Сущность детерминированного конечного автомата как объекта реализована в классе `class DFA`, который объявлен в файле `include/dfa_class.h`. Все состояния хранятся как числа в формате `uint32_t` и могут принимать значения от 0 до $2^{32} - 2 = 4294967294$ (число `UINT32_MAX = 4294967295` зарезервировано как особое "пустое" состояние (`EMPTY_STATE`), которое нужно для упрощения работы некоторых методов).

Основные поля класса `class DFA` (заполняются при инициализации объекта):

- `uint32_t alphabet_length` — длина рабочего алфавита.
- `uint32_t size` — размер автомата.
- `std::vector<std::vector<uint32_t>> delta` — правила переходов автомата. Элемент `[a][q]` — состояние $\delta(q, a)$.
- `uint32_t starting_node` — начальное состояние.
- `std::vector<StateInfo> states_info` — блоки информации для каждого состояния. Один блок информации (объект структуры `struct StateInfo`) содержит следующую информацию о состоянии автомата:
 - ❖ `bool acc` — `true`, если состояние принимающее, `false`, если отвергающее.

Следующие поля нужны только при минимизации автомата (в них храним информацию о разбиении всех состояний на блоки):

- ❖ `uint32_t color` — номер блока, в котором находится состояние (его "цвет").
- ❖ `uint32_t next_state_of_same_color` — номер следующего состояния из того же блока.
- ❖ `uint32_t prev_state_of_same_color` — номер предыдущего состояния из того же блока.

Дополнительные поля класса `class DFA` (используются при минимизации автомата):

- `std::vector<uint32_t> block2first_state_of_this_block` — по номеру блока(цвета) сопоставляет первое состояние этого блока(цвета). То есть i -ый элемент показывает первое состояние в блоке $B(i)$.
- `std::vector<std::vector<uint32_t>> block_and_char2first_node_of_this_color_and_char` — здесь элемент `[a][i]` показывает первое состояние в множестве $\hat{B}(B(i), a)$.
- `std::vector<std::vector<uint32_t>> L` — здесь a -ый элемент — вектор, в котором написаны состояния множества $L(a)$.
- `std::vector<std::vector<bool>> info_L` — здесь элемент `[a][i]` принимает значение `true`, если $i \in L(a)$, иначе принимает значение `false`.

Следующие поля нужны для того, чтобы уметь быстро итерироваться и работать с множествами $\hat{B}(B(i), a)$:

- `std::vector<std::vector<uint32_t>> next_B_cap` — здесь элемент `[a][i]` показывает, какое следующее состояние в том же блоке, что и i -ое состояние, тоже достижимо по символу a (если i -ое состояние не достижимо по символу a ни из какого другого состояния, то нам не важно, что там написано). Если i -ое состояние — последнее в своём блоке, которое достижимо по символу a , то следующим считается "пустое" состояние (`EMPTY_STATE`).
- `std::vector<std::vector<uint32_t>> prev_B_cap` — по аналогии, здесь элемент `[a][i]` показывает, какое предыдущее состояние в том же блоке, что и i -ое состояние, тоже достижимо по символу a .
- `std::vector<std::vector<uint32_t>> B_cap_lengths` — здесь элемент `[a][i]` — это длина $|\hat{B}(B(i), a)|$.

Следующие поля нужны для того, чтобы построить δ^{-1} с небольшой константой по памяти и при этом чтобы итерирование по множеству состояний $\delta^{-1}(q, a)$ было быстрым ($O(|\delta^{-1}(q, a)|)$ времени):

- `std::vector<std::vector<uint32_t>> reversed_delta_lengths` — элемент `[a][i]` показывает $|\delta^{-1}(i, a)|$ — из скольких состояний можно попасть по символу a в состояние i .
- `std::vector<std::vector<uint32_t>> addresses_for_reversed_delta` — элемент `[a][i]` показывает, с какого места в `reversed_delta` начинается последовательность состояний, лежащих в множестве $\delta^{-1}(i, a)$.
- `std::vector<uint32_t> reversed_delta` — вектор состояний длины $n \cdot |\Sigma|$. Он устроен так, что список состояний из $\delta^{-1}(i, a)$ начинается с `addresses_for_reversed_delta[a][i]`-го элемента и занимает длину `reversed_delta_lengths[a][i]` ($i \in Q, a \in \Sigma$).

Следующие поля понадобятся для реализации одной итерации разбиения по блоку $B(i)$ и символу a :

- `std::vector<bool> blocks_need_to_be_separated` — i -ый элемент равен `true`, если блок $B(i)$ существовал и его нужно разделить по символу a , иначе `false`.
- `std::vector<uint32_t> block2index_of_new_block` — если i -ый блок ($B(i)$) разделяется на два блока, то их номера — i и `block2index_of_new_block[i]`.
- `std::vector<uint32_t> sep_blocks` — список разделяемых блоков.
- `std::vector<uint32_t> sep_states` — список состояний, которые окажутся в новых блоках (то есть множество $\delta^{-1}(B(i), a)$), если происходит измельчение с помощью блока $B(i)$ и символа a .
- `std::queue<uint32_t> empty_colors` — неиспользованные цвета (неиспользованный номер для блоков).
- `std::vector<uint32_t> block2index_special` — каждому разбиваемому блоку сопоставляется номер от 0 до количества разбиваемых блоков минус 1 (нужно для упрощения работы в одной итерации минимизации).

Также в классе `class DFA` доступны следующие методы (реализованы в файле `src/dfa_methods.cpp`):

- `void init(uint32_t _alphabet_length, uint32_t _size, uint32_t _starting_node, std::vector<std::vector<uint32_t>> &table, std::vector<bool> &v_acc)` — инициализация автомата новыми значениями длины алфавита, размера, начального состояния, функции δ (`table`) и информацией о принимающих/отвергающих состояниях (`v_acc`).
- `bool check_string(std::vector<uint32_t> &str)` — проверяет, принимается ли строка `str` или нет.
- `void delete_unreachable_states()` — удаление недостижимых состояний (делается в начале минимизации).
- `void construct_reversed_delta()` — строит `reversed_delta_lengths`, `addresses_for_reversed_delta`, `reversed_delta`, используя функцию δ .
- `void color_acc_and_rej_in_2_colors()` — выполняет первую итерацию алгоритма: красит все принимающие состояния в 0 цвет (0 блок), отвергающие — в 1.
- `bool minimize_iteration()` — реализация одной итерации алгоритма Хопкрофта. Это самый важный метод в классе. Работает он так:

1. Поиск $a \in \Sigma$ такого, что $L(a)$ непусто, и выбор какого-нибудь блока $B(i)$ из $L(a)$.
2. Проходимся по всем состояниям `state_i` из i -го блока ($B(i)$), которые достижимы по символу a (то есть, просто перебираем `state_i` из $\hat{B}(B(i), a)$).
Перебираем `sep_state` из $\delta^{-1}(\text{state}_i, a)$. Если состояние одно-единственное в своём блоке, то и разделять нечего.
Иначе добавляем `sep_state` в `sep_states`. Блок ($B(\text{sep_state_color})$), в котором лежит состояние `sep_state`, будет разбит на два новых — $B(\text{sep_state_color})$ (состояние, из которого рёбра по a ведут не в $B(i)$) и $B(\text{block2index_of_new_block}[\text{sep_state_color}])$, из которого рёбра по a ведут в $B(i)$. Номер нового блока `block2index_of_new_block[sep_state_color]` берётся из очереди `empty_colors`.
`sep_state_color` добавляется в `sep_blocks`.
Подсчитывается количество добавляемых блоков `added_blocks`, заполняется `blocks_need_to_be_separated`.
3. Далее будем извлекать состояния из `sep_states` в новые `added_blocks` блоков. Для начала создаются структуры
`std::vector<uint32_t> last_states_of_new_blocks` — тут хранятся последние обработанные состояния для каждого из `added_blocks` новых блоков

и `std::vector<std::vector<uint32_t>> last_states_with_new_color_and_char` — тут для каждого нового блока и символа $a \in \Sigma$ хранится последнее обработанное состояние из этого блока, достижимое по a .

Затем пробегаемся по всем состояниям из `sep_states`, аккуратно отделяем их из старых блоков в новые (чтобы не поломались двусвязные списки, в которых хранятся блоки $B(j)$ и их подмножества $\hat{B}(B(j), c)$ (где, как всегда, $j \in Q, c \in \Sigma$)).

4. Для каждого блока $B(j)$ из `sep_blocks`, разбившегося на блоки $B(j)$ и $B(\text{block2index_of_new_block}[j])$ и для каждого $c \in \Sigma$, нужно также в $L(c)$ добавить j или $\text{block2index_of_new_block}[j]$ в зависимости от того, в какой блок входит меньше ребёр с c и лежит ли j в $L(c)$.
5. В конце нужно очистить `sep_blocks` и `sep_states`, и вернуть в исходное состояние `block2index_of_new_block` (все значения равны `EMPTY_STATE`) и `blocks_need_to_be_separated` (все значения равны `false`).

Функция возвращает значение `true`, если итерация была последней и `false`, если нет.

- `void minimization(bool no_debug)` — в ней происходит минимизация, и затем перестройка автомата: каждый блок становится одним состоянием. Аргумент `bool no_debug` показывает, нужен ли вывод промежуточных результатов минимизации в `stdout` (`false`, если да; `true`, если нет). Схематично, функция работает так:

```

1  void DFA::minimization(bool no_debug) {
2      delete_unreachable_states();
3      construct_reversed_delta();
4      color_acc_and_rej_in_2_colors();
5
6      bool finish = false;
7      while (!finish) {
8          finish = minimize_iteration();
9      }
10
11     /*dfa rebuilding*/
12 }
13
```

В частности, часть методов, нужная практически только для дебага:

- `void print_table()` — вывод таблицы δ в стандартный поток вывода `stdout` (если в DFA не более 50 состояний).
- `void print_current_classes_of_equality(bool finished, bool debug)` — печатает текущее разбиение состояний на блоки в `stdout`.
- `void print_L()` — печатает L в `stdout`.
- `void print_B_caps()` — печатает $\hat{B}(B(j), c)$ для всех $i \in Q, a \in \Sigma$ в `stdout`.
- `uint32_t get_size()` — возвращает размер DFA.

Также есть метод:

- `int save_to_file(char* filename)` — сохранение DFA в бинарный файл, находящемуся по относительному пути, записанному в `filename`. Сначала записываются три параметра `size`, `alphabet_length`, `starting_node` — количество состояний, размер алфавита и номер начального состояния, за тем $|\Sigma| \cdot n$ переходов. Далее записывается информация для каждого состояния, принимающее или отвергающее оно (1 бит) — это информация умещается в $\lceil n/8 \rceil + 1$ байт.

И несколько методов для инициализации (подробнее — в разделе 2.3):

- `DFA(uint32_t _alphabet_length, uint32_t _size, uint32_t _starting_node, std::vector<std::vector<uint32_t>> &_delta, std::vector<bool> &_v_acc)` — инициализация по всем заранее заданным полям.
- `explicit DFA(char* command, char* dfa_str)` — инициализация по двум строкам — командам пользователя — подробнее в разделе 2.3. Функция реализована в файле `src/dfa_build.cpp`. Она по двум аргументам от пользователя генерирует DFA (предварительно проверяя в функции `request_check correctness_of_dfa_input(char* command, char* dfa_str)` корректность вводимых данных (возвращаемое значение `request_check` — пара булевого значения, показывающего, успешна ли проверка или нет и строки, в которой написана, какая ошибка, если она есть).

2.2 Класс NFA

Класс, симулирующий недетерминированный конечный автомат, реализован в `class NFA`, который объявлен в файле `include/nfa_class.h`. Объект этого класса имеет следующие поля:

- `uint32_t alphabet_length` — длина рабочего алфавита.
- `uint32_t size` — размер NFA.
- `std::vector<std::vector<std::vector<uint32_t> > > delta` — функция недетерминированных переходов $\delta: Q \times \Sigma \rightarrow 2^Q$.
- `std::vector<uint32_t> starting_nodes` — начальные состояния NFA.
- `std::vector<bool> v_acc` — информация о принимающих/отвергающих состояниях: элемент `[i]` равен `true` тогда и только тогда когда i -ое состояние принимается.

И методы (реализованы в `src/nfa_methods.cpp`):

- `void init(uint32_t _alphabet_length, uint32_t _size, std::vector<std::vector<std::vector<uint32_t> > > &_delta, std::vector<uint32_t> &_starting_nodes, std::vector<bool> &_v_acc)` — инициализация объекта заданными значениями всех полей.
- `void print()` — печатает NFA в `stdout`.
- `DFA convert2dfa()` — переводит NFA в DFA.
- `uint32_t get_size()` — возвращает размер NFA (количество состояний).
- `bool line_complicated_initial_states()` — `true`, если начальные состояния заданы нетривиально, `false`, если тривиально (то есть начальное состояние одно и оно — 0-ое).
- `std::string longline()` — строковое представление длинных NFA (то есть размера более 61).
- `std::string line()` — строковое представление NFA.
- `state_to_printable_character(int x)` — переводит число от 0 до 61 в цифру или букву (нужно для метода `line()`).

2.3 Взаимодействие с пользователем

Команды пользователя — аргументы функции `main()`. Если проект скомпилирован в файл `minimizer`, то запуск минимизации пользователем в командной строке может выглядеть так:

```
./minimizer from_bin_file binary_files/dfa0.bin -t -np
```

Первые два аргумента (в данном случае — `from_bin_file binary_files/dfa0.bin`) будем называть `char* command` и `char* dfa_str`. Через них однозначно задаётся автомат, который предстоит минимизировать. Первый аргумент — `command` — задаёт тип инициализации DFA. Он может принимать следующие строковые значения:

- * `"from_dfa_string"` — автомат инициализируется из строкового представления DFA, записанного в `dfa_str`. Строковое представление DFA здесь имеет вид `"010110...1_1a6Uy98..."`: сначала идёт n нулей и единиц: i -ая из них означает, принимается ли i -ое состояние или отвергается. Затем идет разделитель `"_"` и $|\Sigma| \cdot n$ состояний (состояние на $(j \cdot a + c)$ -ом месте ($0 \leq c < |\Sigma|$) — состояние в которое идёт ребро из состояния q_j по c -ому символу алфавита Σ). Все состояния кодируются либо цифрой, либо строчной или заглавной латинской буквой (таким образом, в этом случае их не более чем 62). Цифры кодируют первые 10 состояний, строчные буквы — с 11-го по 36-ое, заглавные — последние 26 состояний.
- * `"bamboo"` — в таком случае в `dfa_str` через запятую написан размер и количество символов в алфавите автомата. Сам автомат устроен так: для каждого символа $c \in \Sigma$ из i -го состояния по символу c переход в $(i + 1)$ -ое состояние (если i -ое состояние не последнее), иначе, если состояние последнее, то из него все переходы ведут в себя же. Принимается только последнее состояние. Несложно показать, что такой автомат при минимизации не уменьшается, однако алгоритм Мура на нём работает с асимптотикой $O(n^2)$ по времени, а алгоритм Хопкрофта — за $O(n)$. Этот пример указан в статье Джона Хопкрофта [1].
- * `"circle"` — от автоматов предыдущего типа отличаются лишь тем, что из последнего состояния все переходы ведет не в себя, а в самое первое состояние. Для них верны те же утверждения про алгоритм Мура и Хопкрофта.

- * **"repeated_cycle"** — односимвольный автомат, для которого параметры размер `size` и длина цикла `cycle_size` указываются через запятую в аргументе `dfa_str` (при этом длина цикла — делитель числа состояний!). Переходы тут такие же, как в DFA типа **"circle"** — из i -го состояния переход в состояние $(i + 1) \bmod n$. Но принимающие состояния — все, чьи номера равны $(n - 1)$ по модулю числа **"cycle_size"**. Такой автомат минимизируется до цикла длины `cycle_size`, в котором ровно одно принимающее состояние (последнее), а начальное состояние — первое (то есть до автомата типа **"circle"** размера `cycle_size`).
- * **"from_bin_file"** — чтение DFA из бинарного файла, путь к которому указан во втором аргументе `dfa_str`. Кодировка такая же, как в методе `int save_to_file(char* filename)` класса DFA.
- * **"from_nfa_string"** — сначала с помощью строки `dfa_str` инициализируется NFA, который потом переводится в DFA. В строке `dfa_str` NFA закодирован следующим образом. Для начала для каждого состояния пишется множество состояний, в которые из него можно попасть по каждому из символов Σ . Если множество состоит не из 1 элемента, оно обособляется фигурными скобками. Если состояние начальное, перед тем, как писать его переходы, печатается символ **">"**. Если символов **">"** нет, то начальное состояние по умолчанию только одно — первое. В конце идёт n плюсов и минусов, i -ый плюс означает, что i -ое состояние принимающее, минус — наоборот, отвергающее. Например, у NFA могут быть следующие кодировки: **"{03}1{12}{0}0{02}2{12}--+-"**, **"{01}3>{12}{0}0{02}>2{12}++-"**.

Также пользователь может (но это необязательно) сохранить минимизированный DFA в какой-нибудь бинарный файл. Для этого третьим аргументом необходимо указать слово **"save_to_bin_file"**, а четвёртым — относительный путь к этому файлу.

Также пользователь может указать в конце некоторые из следующих флагов:

- * **-t** — если надо засекать время работы по минимизации автомата (потраченное время будет напечатано в `stdout`).
- * **-nd, --no-debug** — если не надо выводить в `stdout` информацию о успешном начале/конце минимизации и подобных действиях, а также количество состояний в минимизированном автомате.
- * **-np, --no-print** — если не надо в конце выводить в `stdout` новый минимизированный DFA.

Примеры работы программы:

Минимизация автомата, данного в формате NFA, и сохранение его в бинарный файл:

```
1 ./minimizer from_nfa_string {03}1{12}{0}0{02}2{12}--+- save_to_bin_file binary_files/dfa0.bin -t
2 DELETING UNREACHABLE STATES...
3 MINIMIZATION STARTED...
4 MINIMIZATION FINISHED SUCCESSFULLY
5 10 iterations happened
6 DFA UPDATED
7 It has 7 states now
8 Execution time: 0.000243977 seconds.
9 SIZE: 7 LEN_ALPHABET: 2 STARTING_NODE: 2
10 | 0 | 1 | TYPE
11 =====
12 0 | 0 | 0 | ACC
13 =====
14 1 | 0 | 5 | REJ
15 =====
16 2 | 1 | 6 | REJ
17 =====
18 3 | 1 | 0 | ACC
19 =====
20 4 | 4 | 4 | REJ
21 =====
22 5 | 0 | 3 | ACC
23 =====
24 6 | 5 | 4 | REJ
25 =====
26 Saved successfully to binary file
```

Минимизация автомата из файла и сохранение в другой файл:

```
1 ./minimizer from_bin_file binary_files/dfa1.bin save_to_bin_file binary_files/dfa2.bin -t -np -nd
2 Execution time: 8.0964e-05 seconds.
```

Минимизация автомата типа **"bamboo"**, состоящего из 1000000 состояний:

```
1 ./minimizer bamboo 1000000,1 -t -np
2 DELETING UNREACHABLE STATES...
3 MINIMIZATION STARTED...
4 MINIMIZATION FINISHED SUCCESSFULLY
```

```

5 999999 iterations happened
6 DFA UPDATED
7 It has 1000000 states now
8 Execution time: 3.6638 seconds.

```

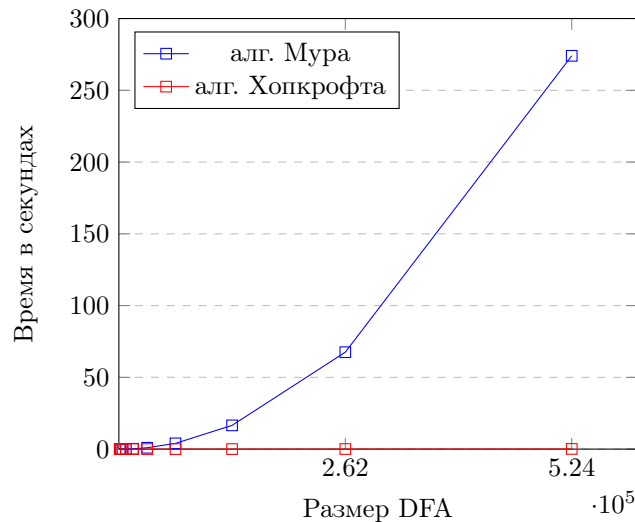
3 Анализ работы программы

3.1 Сравнение времени работы алгоритмов Хопкрофта и Мура

Как уже было отмечено, алгоритм Хопкрофта работает на автоматах типа “бамбук” линейно от размеров автомата, а алгоритм Мура — квадратично. Это было проверено на автоматах размеров 2^n , $n \in \{10, \dots, 24\}$. Результаты измерений времени работы отражены в следующей таблице:

| Размер DFA типа bamboo | Алгоритм Хопкрофта (сек.) | Алгоритм Мура (сек.) |
|------------------------|---------------------------|----------------------|
| 1024 | 0.000269 | 0.0010017 |
| 2048 | 0.00087 | 0.0040186 |
| 4096 | 0.001407 | 0.0155907 |
| 8192 | 0.00277 | 0.0565663 |
| 16384 | 0.004176 | 0.237831 |
| 32768 | 0.008006 | 0.951994 |
| 65536 | 0.014809 | 4.01016 |
| 131072 | 0.029341 | 16.5476 |
| 262144 | 0.058262 | 67.5289 |
| 524288 | 0.120135 | 274.041 |
| 1048576 | 0.235781 | — |
| 2097152 | 0.469479 | — |
| 4194304 | 0.953751 | — |
| 8388607 | 1.97026 | — |

Соотношение времени работы алгоритмов Мура и Хопкрофта на автоматах типа “бамбук”.



Также была исследована работа этих алгоритмов для минимизации DFA для языка, задающего все слова над алфавитом $\{a, b\}$, у которых $(n - 1)$ -ый символ с конца равен a . Строился NFA, задающий этот язык, переводился в DFA, который затем минимизировался (как известно, в минимизированном DFA в таком случае будет не менее чем 2^{n-1} состояний).

| n | Алгоритм Хопкрофта (сек.) | Алгоритм Мура (сек.) |
|----|---------------------------|----------------------|
| 12 | 0.000747 | 0.0010522 |
| 13 | 0.001364 | 0.0017582 |
| 14 | 0.002397 | 0.0029991 |
| 15 | 0.00477 | 0.0049912 |
| 16 | 0.010971 | 0.0135385 |
| 17 | 0.023378 | 0.0270009 |
| 18 | 0.047164 | 0.0590049 |
| 19 | 0.115261 | 0.127855 |
| 20 | 0.302292 | 0.296526 |
| 21 | 0.672999 | 0.65721 |
| 22 | 1.43655 | 1.5208 |
| 23 | 3.25043 | 3.84943 |
| 24 | 6.82125 | 8.92102 |
| 25 | 13.5632 | 18.8834 |
| 26 | 27.5825 | 40.6004 |
| 27 | 59.4406 | — |

Сравнение работы алгоритмов минимизации автоматов, имеющих 2^{n-1} состояний.

При больших n реализация алгоритма Хопкрофта опережает алгоритм Мура. Однако при этом алгоритм расходует намного больше памяти, нежели необходимо алгоритму Мура.

3.2 Расходуемая память

Для каждого состояния хранятся номер блока, в котором оно находится (4 байта), номер следующего и предыдущего состояний из того же самого блока (8 байт). Также для каждого символа $a \in \Sigma$ хранятся номер следующего и предыдущего состояний из этого же блока, которые достижимы по этому символу a (если само состояние достижимо по символу a) — то есть ещё $|\Sigma| \cdot 8$ байт. Для содержания состояния в $L(a)$ нужно $4 + 1 = 5$ байт (4 на само состояние, если оно там лежит, и 1 байт показывающий, лежит ли оно в $L(a)$ или нет). Итого, ещё $|\Sigma| \cdot 5$ байт.

Для каждого блока $B(j)$ (которых может быть от 1 до n) хранится количество состояний в нём, а также количество состояний в нём, достижимых по символу a (для каждого $a \in \Sigma$) — это $(|\Sigma| + 1) \cdot 4$ байт. И ещё хранятся $first(j)$ и $first(j, a)$ (для каждого $a \in \Sigma$) — это ещё $(|\Sigma| + 1) \cdot 4$ байт.

Также каждое из полей класса `class DFA`, отвечающих за одну итерацию минимизации, может достигать размера n (поля `blocks_need_to_be_separated`, `block2index_of_new_block`, `sep_blocks`, `sep_states`, `empty_colors`, `block2index_special`). Они суммарно занимают не более $21 \cdot n$ байт.

Также, для содержания функции δ^{-1} нужно $3 \cdot |\Sigma| \cdot n$ байт. Итого, суммарно используется не более чем

$$(4 + 8 + |\Sigma| \cdot 8 + |\Sigma| \cdot 5 + (|\Sigma| + 1) \cdot 4 \cdot 2 + 21 + 3 \cdot |\Sigma|) \cdot n = (41 + |\Sigma| \cdot 24) \cdot n \text{ байт,}$$

где n — число состояний в исходном DFA.

4 Заключение

Исследована статья Джона Хопкрофта [1], в которой описан этот алгоритм.

Разобрана работа алгоритма Хопкрофта, доказаны утверждения о его корректности и асимптотике, написана программа, минимизирующая входной DFA, пользуясь этим алгоритмом. Поставленная задача выполнена.

Также планируется добавить дополнительную операцию инициализации DFA — `shift` (циклический сдвиг) — получение нового автомата из какого-то путём циклического сдвига. Это преобразование описано в статье [2], с которой я на данный момент разбираюсь.

Список литературы

- [1] John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*, pages 189–196. Elsevier, 1971.
- [2] Galina Jirásková and Alexander Okhotin. State complexity of cyclic shift. *RAIRO-Theoretical Informatics and Applications*, 42(2):335–360, 2008.