Компьютерные инструменты в образовании, 2019
№ 2: 87–107
УДК: 37.01:007+ 004.82+004.85+519.713
http://cte.eltech.ru
doi:10.32603/2071-2340-2019-2-87-107

ТАБЛИЦА СОСТОЯНИЙ НЕДЕТЕРМИНИРОВАННОГО КОНЕЧНОГОГ АВТОМАТА: НАУЧНЫЙ ПРОЕКТ ДЛЯ СТАРШЕКЛАССНИКОВ

Абрамян М. Э.[1], кандидат физико-математических наук, доцент, m-abramyan@yandex.ru
Мельников Б. Ф.[2], доктор физико-математических наук, профессор, bf-melnikov@yandex.ru
Мельникова Е. А.[3], кандидат физико-математических наук, доцент, ya.e.melnikova@yandex.ru

[1]Южный федеральный университет, ул. Большая Садовая, д. 105/42, 344006, Ростов-на-Дону, Россия
[2]Совместный университет МГУ – ППИ, район Лунган, Даюньсиньчэн, улица Гоцзидасюеюань, д. 1, 517182, Провинция Гуандун, Шэнчжэнь, Китай
[3]Российский государственный социальный университет, ул. Вильгельма Пика, д. 4, стр.1, 129226, Москва, Россия

Аннотация

С конца 1960-х годов изучается задача минимизации недетерминированных конечных автоматов. В практических программах для больших размерностей получение точного ответа обычно занимает неприемлемо большое время. В связи с этим нас интересуют, среди прочих, эвристические алгоритмы решения задачи — алгоритмы, «ничего не обещающие», однако на практике в большинстве случаев дающие за приемлемое время работы решение, близкое к оптимальному.

Предлагаемый школьникам проект направлен на частичное решение одной из вспомогательных задач, возникающих в упомянутой оптимизационной задаче. Для этого мы специальным образом определяем отношение эквивалентности на множестве таблиц заданного размера $M \times N$, заполненных элементами 0 и 1. Получение количества неэквивалентных таблиц размерности 8×10 будет являться серьёзным шагом на пути к доказательству того факта, что описанный ещё в 1970 г. пример «плохого» автомата (так называемого автомата Ватерлоо) — минимально возможный пример, не имеющий «меньших» аналогов.

Для решения задачи мы сначала предлагаем плохой алгоритм, заключающийся в простом переборе матриц. Этот алгоритм хорошо работает на матрицах малых размерностей, но, как обычно в подобных ситуациях, при переходе к большим размерностям он работает неприемлемо долго. Для уменьшения времени работы алгоритма мы предлагаем несколько эвристик и приводим результаты работы разных версий программы. Цель проекта — создание новых эвристик, ещё большее убыстрение времени работы программы и, по возможности, получение ответа (количества таблиц) для размерности 8×10.

Для большинства описываемых в статье вариантов алгоритма мы приводим реализацию на языке C#, использующую принципы объектно-ориентированного программирования. Мы предполагаем, что дальнейшая работа над проектом будет заключаться в дальнейшей модификации приведённых нами программ.

Ключевые слова: *оптимизационная задача, конечный автомат, эвристический алгоритм, первый шаг в науке.*

Цитирование: Абрамян М. Э., Мельников Б. Ф., Мельникова Е. А. Таблица состояний конечного автомата: научный проект для старшеклассников // Компьютерные инструменты в образовании. 2019. № 2. С. 87–107. doi: 10.32603/2071-2340-2019-2-87-107

1. ВВЕДЕНИЕ. ПРЕДМЕТНАЯ ОБЛАСТЬ

Во введении мы попытаемся кратко рассмотреть предметную область, то есть приведём самые элементарные сведения о недетерминированных конечных автоматах. А именно, кратко рассмотрим так называемые автоматы Рабина-Скотта (Медведева)[1], причём приведём только *неформальные определения*.

Будем считать автоматом ориентированный граф, у которого каждая дуга помечена буквой некоторого конечного алфавита, некоторые вершины (входы) дополнительно помечены маленькими входящими стре́лками, а некоторые другие (выходы) — маленькими выходящими, при этом вход может являться и выходом. Пример — автомат, приведённый на рис. 1:

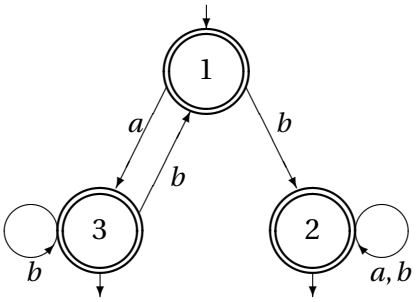

Рис. 1

Его вершины — $\{1,2,3\}$, вход один — $\{1\}$, выходов два — $\{2,3\}$. В автомате возможны дуги-петли, что и видно на приведённом примере: таких петель три, поскольку из вершины 2 в неё же ведут две дуги-петли, с пометками a и b.

Такие автоматы применяются для определения того, принадлежит ли слово заданному автоматом *регулярному* языку; иными словами, язык — множество таких слов — задаётся автоматом. Идя по дугам от какого-нибудь входа автомата к какому-нибудь его выходу, мы выписываем буквы (пометки дуг) подряд, слева направо; то, что при этом получается, является словом автомата (его языка), ничто другое словом автомата не является. В нашем примере словом автомата, приведённого на рис. 1, является, например,

[1] «Как обычно в подобных ситуациях», авторство приписывается *не* российскому (советскому) учёному Ю. Медведеву, а М. Рабину и Д. Скотту. Статья Ю. Медведева [1] вышла в 1956 г. (причём это — не первая статья на данную тему, она вышла уже в обратном переводе на русский язык), в то время как первая из статей Рабина и Скотта вышла в 1959 г.

Однако именно Рабин и Скотт впоследствии (в 1976 г.) получили за конечные автоматы Тьюринговскую премию. При этом даже они не слишком категорично говорят о своём приоритете. Вот, например, цитата из «тьюринговской» лекции Скотта ([2, с. 69]): «… нашу совместную работу [мы с Рабином] сделали в 1957 г. во время летней практики в IBM. Конечно, *мы не были единственными в этой области*, однако нам удалось разработать несколько основных идей».

abbbba. Это слово можно получить, перебирая вершины следующим образом: 1333122 либо 1331222. И несложно доказать, что если некоторый язык может быть задан (хоть одним) конечным автоматом, то он может быть задан бесконечным числом автоматов.

Для нас очень важно отметить следующее. Во-первых, для каждого автомата можно легко построить так называемый зеркальный (изменив на противоположные направления всех его стрелок, «больших» и «маленьких», дуг, входов и выходов), который задаёт зеркальный язык (где каждое слово исходного языка читается справа налево).

Пример зеркального автомата (для автомата, ранее нами рассмотренного) приведён на рис. 2.

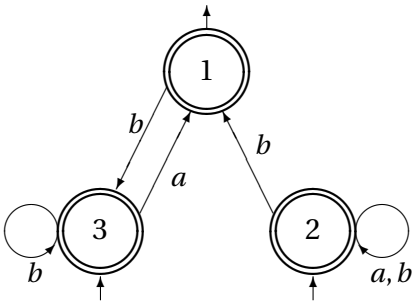

Рис. 2

Во-вторых, если у автомата:

— либо имеется более одного входа;
— либо имеется (хоть одна) вершина, из которой выходят 2 (или более) дуги с одинаковыми пометками,

то такой автомат называется (собственно) недетерминированным. В нашем примере оба автомата (и заданный и зеркальный) — недетерминированные: например, у автомата на рис. 1 из вершины 3 выходят 2 дуги, помеченные одной и той же буквой *b*.

Для каждого автомата существует эквивалентный ему детерминированный, в котором нет ни одного из двух сформулированных «недостатков». Процесс преобразования недетерминированного автомата в детерминированный (так называемая детерминизация) весьма важен, однако здесь мы не будем его рассматривать; скажем только, что в этом процессе мы рассматриваем множество вершин старого автомата как единую конструкцию, единое целое, одну вершину автомата нового.

Рассмотрим другой пример, причём более простой. Пусть исходный автомат изображён на рис. 3:

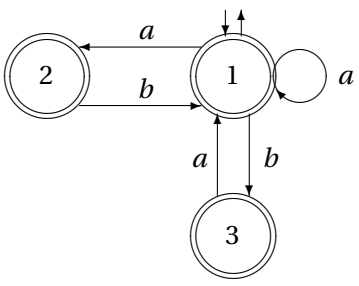

Рис. 3

Эквивалентный ему детерминированный (причём так называемый *канонический*, то есть *минимальный детерминированный*) приведён на рис. 4[2]:

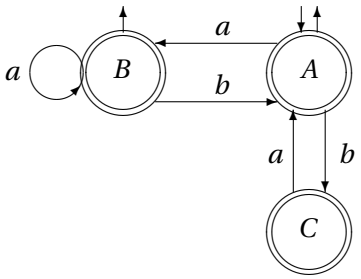

Рис. 4

Отметим, что при этом легко определяется так называемое *соответствие вершин*, фактически уже получавшееся в процессе детерминизации:

— вершине 1 соответствуют две вершины — A и B,
— вершине 2 — только B,
— а вершине 3 — только C.

Теперь рассмотрим зеркальный (для исходного) автомат — он приведён на рис. 5:

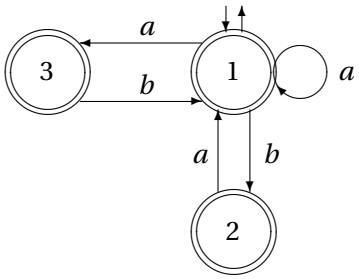

Рис. 5

Сто́ит отметить, что в нашем примере зеркальный автомат фактически совпадает с исходным (то есть он может быть получен из него переименованием вершин, ср. рис. 3), но в общем случае такое совпадение, конечно, не обязательно. Детерминированный (канонический) автомат для зеркального приведён на рис. 6:
Он совпадает с автоматом, приведённым на рис. 4, но в общем случае такой факт тоже неверен. А соответствие вершин в случае зеркальных автоматов и языков таково:

— вершине 1 соответствуют две вершины — X и Y,
— вершине 2 — вершина Z,
— а вершине 3 — вершина Y.

В процессе такого построения мы получили очень важную для дальнейшего изложения информацию:

— вершина B (на рис. 4) «через» вершину 2 (на рис. 3 и 5) соответствует вершине Z (на рис. 6);

[2] Ещё раз отметим, что мы *не будем* приводить алгоритмы построения:
— детерминированного автомата на основе произвольного автомата для заданного регулярного языка;
— канонического автомата на основе детерминированного.

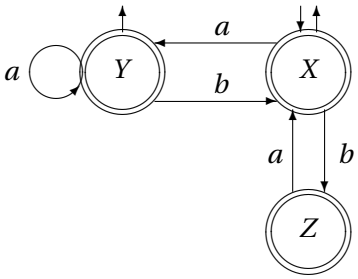

Рис. 6

— аналогично вершина C — вершине Y,
— а каждая из вершин A и B («через» вершину 1) — каждой из вершин X и Y.

Таким образом, получается *таблица соответствия вершин автоматов*, изображённых на рис. 4 и 6 (канонических для заданного и зеркального языков) — в нашем примере такая таблица приведена на рис. 7:

	X	Y	Z
A	#	#	
B	#	#	#
C		#	

	X	Y	Z
A	1	1	0
B	1	1	1
C	0	1	0

Рис. 7

(здесь два варианта: слева — более наглядный, а справа — более формальный). Именно с подобными таблицами мы и будем работать в дальнейшем.

Для программирования старшими школьниками интересны [3] задачи *моделирования работы* конечных автоматов (то есть получения ответа на вопрос, принадлежит ли некоторое заданное слово языку некоторого заданного автомата), однако мы не будем их рассматривать. Также отметим, что, как ни странно, для описания проекта можно обойтись и без вышеприведённых элементарных сведений о недетерминированных конечных автоматах…

2. МОТИВАЦИЯ — ДЛЯ ЧЕГО БУДУТ ПОЛЕЗНЫ РЕЗУЛЬТАТЫ ПРОЕКТА

Очень кратко расскажем о возможном применении результатов проекта «на высшем уровне» (в частности — почему нам интересна именно размерность 8×10).

С конца 1960-х годов изучается *задача минимизации* недетерминированных конечных автоматов, то есть задача построения *недетерминированного* конечного автомата, определяющего заданный регулярный язык и имеющего минимально возможное число состояний. Вряд ли легко удастся найти статью, в которой эта задача была поставлена впервые, а в монографической литературе она впервые была упомянута, по-видимому, в [3] (исходное английское издание книги вышло в 1972 г.). В начале 1990-х в несколь-

[3] Причём не в качестве сложных заданий, а «как лабораторные работы».

ких статьях было независимо показано, что эта задача является NP-полной [4]; чаще всего среди этих статей цитируется [5]. Таким образом, для больших размерностей получение точного ответа обычно занимает неприемлемо большое время. В связи с этим нас всегда (то есть во всех конкретных постановках проблемы) интересуют, среди прочих, и *эвристические* алгоритмы решения задачи — алгоритмы, «ничего не обещающие», однако на практике в большинстве случаев дающие за приемлемое время работы так называемое псевдооптимальное решение, близкое к оптимальному.

Один из возможных подходов к решению задачи (в частности, к созданию эвристических алгоритмов для её решения) — это так называемый подход Полака, заключающийся в выборе блоков (гридов) так называемого универсального конечного автомата (см. некоторые подробности в [6, 8]). Такие блоки выбираются именно на основе таблиц, рассмотренных в предыдущем разделе. Рассматривать более подробно мы не будем (это выходит за рамки предлагаемого проекта), однако предполагаем вернуться к этому вопросу в статье-продолжении.

Ещё в 1970 году в статье [7] был опубликован пример [5], показывающий неприменимость «самого жадного» алгоритма минимизации; однако, по-видимому, «полное осмысление» этого примера было сделано только в упомянутых статьях [6, 8]. Несмотря на то, что этому примеру почти полвека, пока нельзя сказать, насколько он «вреден» для *практических* алгоритмов решения задачи минимизации… Для *частичного* ответа на этот вопрос и предназначен предлагаемый проект.

Отдельным вопросом можно считать описание связи между сформулированными нами вариантами проекта и подсчётом числа *двудольных графов* специального вида (про последние см. [9, 10], современный двухтомник [11, 12] и др.). Как и для рассмотрения множества блоков, приводить более подробное изложение мы не будем, однако также предполагаем вернуться к этому вопросу в статье-продолжении.

На связанные темы упомянем также некоторые публикации авторов настоящей статьи:

— «теоретические» публикации — см. [13–16] и др.;
— описание практических алгоритмов, связанных с *автоматическим исследованием* автомата из [7], — см. статью [17];
— описание подобных алгоритмов, применяемых в других (смежных) задачах дискретной оптимизации — см. [18, 19].

Итак, мы можем сказать, что получение *количества неэквивалентных таблиц* размерности 8 × 10 будет являться серьёзным шагом на пути к доказательству того факта, что пример из [7] — минимально возможный пример «плохого» автомата.

3. ПОСТАНОВКА ЗАДАЧИ — ПРОСТОЙ ВАРИАНТ

Итак, будем работать с матрицами (таблицами), аналогичными приведённой выше на рис. 7; матрица заполнена элементами 0 и 1. Будем считать, не ограничивая общности, что матрица «лежит», то есть пусть число её строк (m) не превышает числа её столбцов (n). Считаем, что матрица обладает специальными «хорошими» свойствами:

— в ней нет одинаковых срок;

[4] Книг про NP-полноту «полностью на школьном уровне», по-видимому, нет. Однако, например, соответствующий материал из [4] старшеклассники освоить могут.

[5] Можно сказать, что в примере рассматривается таблица размерности 8 × 10. То есть *исходный детерминированный* автомат имеет 8 состояний, а *канонический для зеркального* языка — 10 состояний.

— никакая строка не состоит только из 0;
— оба подобных условия выполняются и для столбцов.

Можно доказать ([13]), что любая матрица, заполненная 0 и 1 и имеющая эти свойства, может являться матрицей соответствия вершин некоторого автомата (точнее, некоторого языка, определяемого автоматом). Причём из этих условий следует неравенство $n \leqslant 2m–1$. Таким образом, для чисел m и n выполняется двойное неравенство: $m \leqslant n \leqslant 2m–1$.

Вводный проект. *Сколько для заданных натуральных m и n существует таких «хороших» матриц размерности $m \times n$?*

Как обычно в подобных случаях, эту задачу можно рассматривать:

— и как математическую (комбинаторную),
— и как программистскую (переборную).

То же самое относится и к основной теме работы, которая будет сформулирована далее.

4. ПОСТАНОВКА ЗАДАЧИ — ПОЛНЫЙ ВАРИАНТ

После того как мы научились выбирать все матрицы, которые могут быть матрицами соответствия состояний вершин, возникает такой вопрос. Ведь очевидно, что сами обозначения вершин можно вводить более-менее произвольно. И поскольку мы записали вершины обоих канонических автоматов как строки (столбцы) матрицы, то при других именах строк и/или столбцов мы получим изменение матрицы путём перестановок её строк и/или столбцов. Матрицы, получаемые одна из другой таким образом, будем считать эквивалентными. Простой пример приведён далее на рис. 8 (обозначения строк и столбцов, как мы фактически уже отметили, непринципиальны) — вторая матрица получается из первой перестановкой 2-й и 3-й строк и 2-го и 3-го столбцов:

1	1	1	0
1	0	0	1
1	1	0	0

1	1	1	0
1	0	1	0
1	0	0	1

Рис. 8

Основной проект. *Сколько для заданных натуральных m и n существует «хороших», причём попарно неэквивалентных матриц размерности $m \times n$?*

Рассмотрим простые примеры. Для размерности 2×3 возможны только 3 различных столбца (столбец из одних 0 по условию невозможен) и, вследствие этого, *все* матрицы такого размера эквивалентны между собой. А все возможные попарно неэквивалентные матрицы размерности 3×4 (их 10 штук) приведены на рис. 9.

Далее нами приведена простая реализация простого переборного подхода, однако так задачу решать *не надо* (поэтому программа приведена практически без комментариев). Предметом проекта является выбор и реализация эвристик, позволяющих сильно уменьшить полный перебор. По-видимому, при хорошей организации вычислений удастся примерно за сутки работы компьютера получить ответ для размерности 8×10 — в которой только один вспомогательный алгоритм проверки неэквивалентности потребовал бы перебора 2^{18} различных матриц. Отметим, что хорошо надо организовывать не

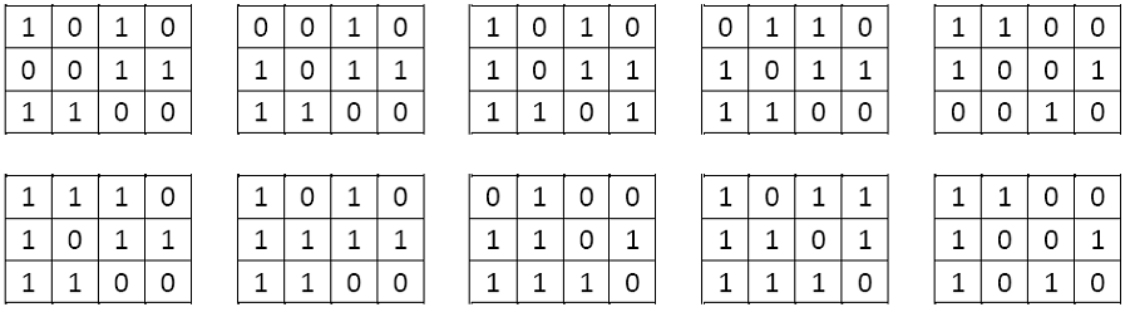

Рис. 9

только алгоритмы, но и структуры данных: например, при больших размерностях хранить все неэквивалентные матрицы малоэффективно . . .

5. КАК МОЖНО РЕШАТЬ ЗАДАЧУ ДЛЯ МАЛЫХ РАЗМЕРНОСТЕЙ (И КАК ЕЁ НЕ НАДО РЕШАТЬ ДЛЯ РАЗМЕРНОСТЕЙ БОЛЬШИХ)

При формулировке проектов мы специально не указывали необходимую размерность — «чем больше, тем лучше!». Проведём, однако, несложные подсчёты: для размерности 3×4 возможны $2^3 = 8$ перестановок строк, $2^4 = 16$ перестановок столбцов, и, следовательно, $2^7 = 128$ «общих перестановок». Таким образом, для проверки двух матриц размерности 3×4 на неэквивалентность необходимо проверить только 128 вариантов. Количество матриц оценивается сверху величиной $2^{12} = 4096$ (фактически матриц будет меньше, так как не все они обладают «хорошими» свойствами). Перебор такого количества пар матриц даже на домашнем компьютере потребует долей секунды. Правда, с ростом размерности анализируемых матриц число как перестановок, так и самих матриц стремительно растёт. Например, уже для размерности 5×5 число «общих перестановок» уже будет примерно в 10 раз больше ($2^{10} = 1024$), а «хорошие» матрицы придется выбирать из 2^{25} возможных (то есть из более чем 33 миллионов матриц).

Тем не менее, как мы уже отметили, мы приводим реализацию простого переборного алгоритма, в которой предусмотрим возможность задавать размерность обрабатываемых матриц. Это, в частности, позволяет исследовать, насколько быстро растёт время работы алгоритма при увеличении размерности матриц.

Реализуем алгоритм на языке C# версии 7.0 и оформим его в виде класса `BMatrix`.

```
class BMatrix {
...
}
```

Класс будет состоять из большого количества членов (здесь мы пока заменили их многоточиями); опишем их по отдельности. Прежде всего, включим в класс `BMatrix` вспомогательный вложенный класс `Permutations`, который «будет отвечать» за генерацию и предоставление наборов перестановок:

```
protected static class Permutations {
static List<int[]>[] available = new List<int[]>[10];
```

```
public static List<int[]> Get(int n) {
if (n < 1 || n > 10) throw new ArgumentOutOfRangeException();
if (available[n - 1] != null) return available[n - 1];
var result = new List<int[]>();
var num = new int[n];
var current = new int[n];

void step(int pos, int val) {
current[pos] = val;
num[val] = -1;
if (pos == n - 1) result.Add(current.ToArray());
else
for (int i = 0; i < n; i++)
if (num[i] != -1) step(pos + 1, num[i]);
num[val] = val;
}

for (int i = 0; i < n; i++) num[i] = i;
for (int i = 0; i < n; i++) step(0, i);
available[n - 1] = result;
return result;
}
}
```

Этот класс состоит из закрытого (`private`) массива `available`, элемент которого с индексом `n` содержит наборы всевозможных перестановок чисел от 0 до n (таким образом, элемент с индексом n содержит все перестановки порядка n), и открытого (`public`) метода `Get(n)`, который позволяет получить полный набор перестановок порядка n.

Набор перестановок нужного порядка вычисляется единственный раз — в тот момент, когда этот набор впервые потребуется в классе `BMatrix`. Последующие запросы `Get` с тем же параметром просто возвращают ссылку на уже вычисленный набор.

Набор перестановок хранится в виде «списка на базе массива» `List`, элементами которого, в свою очередь, являются массивы конкретных перестановок (поэтому набор перестановок описывается как `List<int[]>`, а массив available таких наборов имеет описатель `List<int[]>[]`). Использование списка `List` позволяет *не определять заранее его размер*, поскольку для добавления к списку нового элемента достаточно использовать его метод `Add`. В дальнейшем мы будем часто использовать структуры типа `List` в случае, когда размер набора данных нельзя заранее определить.

Следует обратить внимание на алгоритм, вычисляющий различные перестановки требуемого порядка. Он основан на так называемом «переборе с возвратом» — стандартном методе, используемом в ситуациях, когда требуется перебрать различные комбинации исходных данных (или, иными словами, когда требуется пройти по различным маршрутам в дереве вариантов). «Ядром» перебора с возвратом является рекурсивная функция `step`, добавляющая к текущей комбинации очередной элемент данных (иными словами, выбирающая очередной шаг маршрута в дереве вариантов). После этого данная функция анализирует, построена ли нужная комбинация (пройден ли весь маршрут), и при положительном ответе возвращает полученную комбинацию (маршрут) и завер-

шает работу. При отрицательном ответе она *рекурсивно вызывает себя* для добавления в комбинацию новых данных (выполнения следующего шага в дереве вариантов).

В данной реализации перебора с возвратом все перестановки последовательно строятся в массиве `current`, причём добавленное в этот массив значение `val` помечается во вспомогательном массиве `num` как использованное (соответствующему элементу массива `num` присваивается особое значение –1), и поэтому это значение *уже не может быть выбрано* при построении завершающей части перестановки. Заметим, что при «возврате» из очередного шага алгоритма необходимо «снять» с соответствующего значения метку (присвоенное ранее значение –1), чтобы его можно было добавлять в другие перестановки.

При добавлении в список `result` очередной перестановки, построенной в массиве `current`, для данного массива дополнительно вызывается метод `ToArray()`, создающий копию полученного массива (если этого не сделать, то в списке `result` хранились бы одинаковые ссылки на исходный массив `current`, что, конечно, было бы ошибкой).

Функция `step` описана как вложенная (локальная) функция метода `Get`. Возможность использования вложенных функций, характерная для таких языков, как Паскаль и Питон, особенно удобна при реализации рекурсивных алгоритмов. В Си-подобных языках она, как правило, отсутствует, однако реализована в последней версии 7.0 языка C#.

Мы уделили особое внимание классу Permutations, чтобы продемонстрировать различные особенности языка C#, используемые в дальнейшем, и чтобы описать *пример реализации метода перебора с возвратом*, который также пригодится нам впоследствии.

Следующая группа полей, свойств и методов класса `BMatrix` является традиционной. Мы определяем свойства `M` и `N`, отвечающие за доступ к размерности матрицы, свойство-индексатор для доступа к её элементам, а также описываем два конструктора для создания матриц и метод `ToString`, возвращающий строковое представление матрицы:

```
public readonly int M;
public readonly int N;
protected bool[,] arr;

public bool this[int i, int j] {
get => arr[i, j];
protected set => arr[i, j] = value;
}

public BMatrix(int m, int n, int init) {
M = m;
N = n;
arr = new bool[m, n];
for (int i = 0; i < M; i++)
for (int j = 0; j < N; j++) {
arr[i, j] = (init % 2) == 1;
init = init / 2;
}
}
public BMatrix(BMatrix other) {
M = other.M;
```

```
N = other.N;
arr = new bool[M, N];
for (int i = 0; i < M; i++)
for (int j = 0; j < N; j++)
arr[i, j] = other[i, j];
}
public override string ToString() {
string result = "";
for (int i = 0; i < M; i++) {
for (int j = 0; j < N; j++)
result += arr[i, j] ? "1 " : "0 ";
result += "\r\n";
}
return result;
}
```

Поскольку после создания матрицы её размерность не изменяется, соответствующие поля M и N помечены как `readonly` (это означает, что их значения можно задавать только в конструкторах класса). Элементы матрицы хранятся в (закрытом) булевском двумерном массиве `arr`, однако для доступа к ним удобнее использовать свойство-индексатор с двумя параметрами, которые указываются в квадратных скобках и определяют индекс строки и столбца элемента (индексация, как обычно в C#, выполняется от 0). Для доступа к элементам матрицы `matr` достаточно применить индексирование непосредственно к имени матрицы: `matr[i, j]`. Доступ к элементам является открытым на чтение и защищённым (`protected`) на запись; таким образом, изменять их значения можно только в классах-потомках класса `BMatrix`.

Первый конструктор с параметрами `(m, n, init)` предназначен для быстрой генерации всех возможных матриц требуемого порядка; для этого в качестве инициализирующего значения `init` достаточно указать все целые числа в диапазоне от 0 до 2^{mn}–1. Второй вариант конструктора является конструктором копии и позволяет создавать копию имеющейся матрицы.

В методе `ToString` формируется текстовое представление матрицы, в котором значения `true` обозначаются числами 1, а значения `false` – числами 0. Кроме того, в полученную текстовую строку добавляются символы разрыва текста, чтобы обеспечить наглядное отображение матрицы на экране.

Следующая большая группа методов связана с проверкой матрицы на наличие в ней «хороших» свойств. Основной метод `IsCorrect` является открытым, прочие (вспомогательные) методы являются защищёнными:

```
protected bool RowIsEmpty(int i) {
for (int j = 0; j < N; j++) if (arr[i, j]) return false;
return true;
}

protected bool ColumnIsEmpty(int j) {
for (int i = 0; i < M; i++) if (arr[i, j]) return false;
return true;
}
```

```
protected bool RowsAreTheSame(int i1, int i2) {
for (int j = 0; j < N; j++)
if (arr[i1, j] != arr[i2, j]) return false;
return true;
}

protected bool ColumnsAreTheSame(int j1, int j2) {
for (int i = 0; i < M; i++)
if (arr[i, j1] != arr[i, j2]) return false;
return true;
}

public bool IsCorrect() {
for (int i = 0; i < M; i++)
if (RowIsEmpty(i)) return false;
for (int j = 0; j < N; j++)
if (ColumnIsEmpty(j)) return false;
for (int i1 = 0; i1 < M - 1; i1++)
for (int i2 = i1 + 1; i2 < M; i2++)
if (RowsAreTheSame(i1, i2)) return false;
for (int j1 = 0; j1 < N - 1; j1++)
for (int j2 = j1 + 1; j2 < N; j2++)
if (ColumnsAreTheSame(j1, j2)) return false;
return true;
}
```

С методом `IsCorrect` связан статический метод `GetCorrectMatr(m,n)`, возвращающий список всех матриц размера $m \times n$, обладающих «хорошими» свойствами. В отличие от ранее рассмотренных экземплярных (или динамических) методов, он должен вызываться не для конкретного экземпляра матрицы, а для класса `BMatrix` в целом, например `BMatrix.GetCorrectMatr(3,4)`.

```
public static List<BMatrix> GetCorrectMatr(int m, int n) {
var result = new List<BMatrix>();
int maxInit = 1 << (m * n);
for (int i = 0; i < maxInit; i++) {
BMatrix matr = new BMatrix(m, n, i);
if (matr.IsCorrect()) result.Add(matr);
}
return result;
}
```

Следует обратить внимание на применение операции левого сдвига `<<` для быстрого вычисления степени 2^{mn}.

Очередная группа методов связана с проверкой матриц на эквивалентность. Основным здесь является экземплярный метод `Equiv(other)`, который возвращает `true`, если матрица, для которой вызван данный метод, эквивалентна матрице `other`. В классе

BMatrix реализован *наиболее простой алгоритм*, который перебирает все комбинации перестановок строк и столбцов матрицы other, пока не будет получена матрица, совпадающая с исходной матрицей (в этом случае возвращается true), или пока не будут перебраны все комбинации перестановок (в этом случае возвращается false):

```
public bool EqualDims(BMatrix other) {
return M == other.M && N == other.N;
}

public virtual bool Equiv(BMatrix other) {
if (!EqualDims(other)) return false;
var rowPermutations = Permutations.Get(M);
var columnPermutations = Permutations.Get(N);

bool Congr(int rowPermutationIndex, int columnPermutationIndex) {
for (int i = 0; i < M; i++)
for (int j = 0; j < N; j++)
if (this[i, j] !=
other[rowPermutations[rowPermutationIndex][i],
columnPermutations[columnPermutationIndex][j]])
return false;
return true;
}

for (int i = 0; i < rowPermutations.Count; i++)
for (int j = 0; j < columnPermutations.Count; j++)
if (Congr(i, j)) return true;
return false;
}
```

Именно в этом методе используются наборы перестановок, «предоставляемые» классом Permutations. В целях большей наглядности двойной цикл, в котором матрица other обрабатывается очередной парой перестановок, оформлен в виде локальной функции Congr. Метод Equiv описан как виртуальный, поскольку в классах-потомках BMatrix он может быть заменён на более эффективный. Наконец, последний метод класса BMatrix — статический метод FindEquivMatr(matrices), который анализирует список матриц matrices, выбирает из него попарно неэквивалентные матрицы и возвращает полученный набор в виде списка:

```
public static List<BMatrix> FindEquivMatr(List<BMatrix> matrices) {
var result = new List<BMatrix>();
foreach (var matr in matrices) {
bool addToResult = true;
for (int i = result.Count - 1; i >= 0; i--)
if (matr.Equiv(result[i])) {
addToResult = false;
break;
}
```

```
if (addToResult) result.Add(matr);
}
return result;
}
```

В описанном методе мы использовали два варианта цикла для перебора элементов коллекции (в данном случае — списка матриц). Один из них — традиционный цикл `for`, перебирающий индексы элементов, а другой — цикл `foreach`, параметрами которого являются сами элементы. Поскольку в нашем случае индексы в алгоритме не требуются, можно было бы дважды применить цикл `foreach`, однако мы использовали цикл `for`, для того чтобы более просто организовать перебор элементов коллекции `result` в обратном порядке (заметим, что аналогичный перебор можно организовать и в цикле `foreach`, но он потребует применения дополнительных средств языка C#, которые здесь не рассматриваются). Забегая немного вперед, отметим, что в последующих модификациях алгоритма перебор в обратном порядке позволяет несколько ускорить их работу.

Имея класс `BMatrix`, мы теперь можем легко «собрать» его компоненты для получения первого (как мы покажем далее, наименее эффективного) алгоритма решаемой задачи. Соответствующий метод `BasicAlg`, содержащий, наряду с собственно вычислениями, функции замера времени и операторы вывода, оформим в базовом классе `Program` консольного приложения, а вызовы этого метода для матриц разной размерности выполним в функции `Main` из того же класса:

```
class Program {
static void BasicAlg(int m, int n) {
DateTime t = DateTime.Now;
List<BMatrix> correctMatr = BMatrix.GetCorrectMatr(m, n);
var equivMatr = BMatrix.FindEquivMatr(correctMatr);
Console.WriteLine
($"{m}x{n}: time(sec) = {(DateTime.Now - t).TotalSeconds:f3}");
Console.WriteLine
($"CorrectMatr: {correctMatr.Count} EquivMatr: {equivMatr.Count}");
}

static void Main(string[] args) {
BasicAlg(3, 3);
BasicAlg(3, 4);
BasicAlg(4, 4);
BasicAlg(4, 5);
Console.ReadLine();
}
}
```

Ниже приведены результаты работы данной программы.

```
3 × 3:  time(sec) = 0.042    CorrectMatr: 174    EquivMatr: 8
3 × 4:  time(sec) = 0.070    CorrectMatr: 840    EquivMatr: 10
4 × 4:  time(sec) = 48.144    CorrectMatr: 24360    EquivMatr: 66
4 × 5:  time(sec) = 8415.497    CorrectMatr: 335160    EquivMatr: 168
```

Хотя количество попарно неэквивалентных матриц порядка 4×5 менее чем в 3 раза превосходит соответствующее количество для матриц порядка 4×4, число матриц с «хорошими» свойствами, которые требуется сравнить между собой, выросло примерно в 13 раз, а время расчёта, соответственно, в 175 раз. Понятно, что на этом пути получить результаты для матриц большего размера за приемлемое время не получится, и *необходимо искать пути повышения эффективности базового алгоритма*.

6. НЕКОТОРЫЕ ДОПОЛНИТЕЛЬНЫЕ ЭВРИСТИКИ, УСКОРЯЮЩИЕ БАЗОВЫЙ АЛГОРИТМ

Основное время работы базового алгоритма тратится на попарную проверку эквивалентности матриц из набора матриц с «хорошими» свойствами. В случае неэквивалентных матриц базовый метод `Equiv(other)` переберёт все возможные перестановки строк и столбцов матрицы `other`, и только после этого вернёт значение `false`. Однако в ряде ситуаций проверку можно ускорить, если предварительно проанализировать такие характеристики матриц, как суммарное количество значений `true` в каждой строке и каждом столбце. Две матрицы могут оказаться эквивалентными только в случае, если у них совпадают наборы указанных характеристик для строк и для столбцов (поскольку эти наборы не изменяются при перестановках строк и столбцов матрицы).

Для ускорения сравнения удобно упорядочить набор характеристик для строк данной матрицы и набор характеристик для её столбцов, после чего объединить эти два набора в целочисленный массив `info` размера $M + N$. В начале метода `Equiv(other)` следует сравнить массивы `info`:

— матрицы, для которой вызывается этот метод,
— и матрицы `other`;

если эти массивы не совпадают, нужно немедленно вернуть `false`.

Реализовать указанную эвристику можно непосредственно в классе `BMatrix`, добавив к нему поле `info`, метод `makeInfo`, в котором заполняется данное поле, и откорректировав конструктор класса и метод `Equiv`.

Однако можно поступить и по-другому: не изменяя класс `BMatrix`, определить его класс-потомок `BIMatrix` с новыми полями и методами:

```
class BIMatrix : BMatrix {
int[] info;
void makeInfo() {
// заполнение поля info
}

public BIMatrix(BMatrix m) : base(m) {
makeInfo();
}

bool EqualInfo(BMatrix other) {
if (!EqualDims(other)) return false;
BIMatrix m = other as BIMatrix;
if (m == null) return true;
for (int i = 0; i < info.Length; i++)
```

```
if (info[i] != m.info[i]) return false;
return true;
}

public override bool Equiv(BMatrix other) {
if (!EqualInfo(other)) return false;
return base.Equiv(other);
}

public static List<BMatrix> ToBIMatrixList(List<BMatrix> matrices) {
var result = new List<BMatrix>();
foreach (var e in matrices) result.Add(new BIMatrix(e));
return result;
}
}
```

Для класса `BIMatrix` реализован конструктор, позволяющий создавать объект на базе матрицы типа `BMatrix`. Предусмотрен также статический метод `ToBIMatrix`, преобразующий набор матриц `BMatrix` в набор матриц `BIMatrix`, снабжённых дополнительной информацией (следует обратить внимание на то, что метод `ToBIMatrix` возвращает список типа `List<BMatrix>`, чтобы его можно было передать в метод `FindEquivMatr`). Новый вариант метода `Equiv` является переопределенным вариантом одноимённого виртуального метода. Это означает, в частности, что при вызове метода `matr.Equiv` в методе `FindEquivMatr` будет выбираться тот вариант метода `Equiv`, который соответствует фактическому типу объекта `matr`.

После реализации класса `BIMatrix` его можно использовать для нахождения всех попарно неэквивалентных матриц следующим образом (сравните с методом `BasicAlg`):

```
List<BMatrix> correctMatr =
BIMatrix.ToBIMatrixList(BMatrix.GetCorrectMatr(m, n));
var equivMatr = BMatrix.FindEquivMatr(correctMatr);
```

Приведем время работы модифицированного алгоритма для матриц размерности 3×4, 4×4 и 4×5:

3×4: time(sec) = 0,012
4×4: time(sec) = 1,87
4×5: time(sec) = 147,6

Таким образом, время обработки матриц 4×4 ускорилось в 26, а матриц 4×5 — в 57 раз.

Однако для матриц бóльших размерностей указанного ускорения уже недостаточно, прежде всего, из-за стремительного роста числа матриц с «хорошими» свойствами (то есть из-за так называемого «комбинаторного взрыва»). Например, имеется 3 553 200 «хороших» матриц размера 4×6 и 15 198 120 «хороших» матриц размера 5×5.

Рассмотрим ещё одну эвристику, позволяющую ускорить алгоритм за счёт уменьшения числа анализируемых матриц. Поскольку принадлежность матрицы к одному и тому же набору эквивалентных матриц не меняется при перемене местами её строк или столбцов, выберем среди всех эквивалентных матриц такие, в которых строки и столбцы упорядочены *в лексикографическом порядке их элементов* (например по убыванию). Примерами таких *упорядоченных* матриц размера 3×4 являются следующие:

1	0	0	0
0	1	1	0
0	1	0	1

1	1	0	0
1	0	1	0
0	0	0	1

Рис. 10

Эти матрицы обладают необходимыми «хорошими» свойствами и, кроме того, каждая следующая их строка «меньше» предыдущей, если считать, что строки сравниваются поэлементно и что значение 0 (`false`) меньше значения 1 (`true`). Кроме того, такое же свойство упорядоченности имеет место и для их столбцов.

Понятно, что алгоритм поиска попарно неэквивалентных матриц достаточно реализовать только для матриц, обладающих указанным свойством упорядоченности. Можно даже предположить, что все различные упорядоченные матрицы попарно неэквивалентны и что достаточно просто получить множество всех упорядоченных матриц, чтобы решить основную задачу. К сожалению, это последнее предположение является неверным: например, нетрудно проверить, что эквивалентны приведённые выше упорядоченные матрицы.

Тем не менее, совершенно очевидно, что число различных упорядоченных матриц, обладающих «хорошими» свойствами, гораздо меньше общего числа «хороших» матриц той же размерности.

Метод `GetCorrectOrderedMatrix(m, n)`, генерирующий все упорядоченные матрицы, имеющие «хорошие» свойства, можно включить в класс `BMatrix`, в этом случае он будет возвращать коллекцию `List<BMatrix>`.

В качестве алгоритма генерации всех упорядоченных матриц можно использовать вариант метода перебора с возвратом (*уже использованного нами ранее* для генерации всех перестановок). Функция `step` в данном случае будет иметь три параметра: индексы `i` и `j` добавляемого элемента матрицы и его значение `val`. Первый вызов данной функции будет иметь вид `step(0,0,true)`, поскольку начальный элемент упорядоченной матрицы не может иметь значение `false`. Последующие рекурсивные вызовы функции `step` можно организовать таким образом, чтобы матрица заполнялась либо по строкам, либо по столбцам. Вызов `step(i, j, false)` можно выполнять в любом случае, вызов `step(i, j, true)` — только если добавляемый элемент не нарушит упорядоченности уже построенных частей предыдущей строки и столбца. После заполнения очередной строки или столбца необходимо также проверять, что эта строка (столбец) удовлетворяет дополнительным свойствам: не состоит полностью из значений `false` и не совпадает с предыдущей строкой (столбцом). Все успешно построенные упорядоченные матрицы сохраняются в списке `List`, который является возвращаемым значением метода `GetCorrectOrderedMatrix`.

Для проверки эффективности данной эвристики достаточно передать в метод `FindEquivMatr` набор матриц типа `BMatrix`, возвращённый методом `GetCorrectOrdered Matrix`.

Эвристику 2, основанную на применении упорядоченных матриц, можно комбинировать с ранее рассмотренной эвристикой 1, ускоряющей проверку эквивалентности матриц за счёт анализа их дополнительной характеристики `info`. Для этого достаточно преобразовать список `List<BMatrix>` методом `ToBIMatrixList` и использовать преобразованный список в качестве параметра метода `FindEquivMatr`.

Ниже в таблице 1 приведены результаты тестирования новых вариантов алгоритма для матриц размера 3 × 4, 4 × 4, 4 × 5, 4 × 6 и 5 × 5. Указывается время работы алгоритма (`t`) и количество проанализированных матриц с «хорошими» свойствами (`corr`). Найденное количество попарно неэквивалентных матриц (`eq`) указано в столбце с данными о размерности. Для сравнения в таблицу включены и ранее описанные результаты для предыдущих вариантов алгоритма.

Таблица 1

Dim	BasicAlg	Эвристика 1	Эвристика 2	Эвристики 1+2
3×4 eq=10	t=0,070 corr=840	t=0,012 corr=840	t=0,001 corr=19	t=0,031 corr=19
4×4 eq=66	t=48,144 corr=24360	t=1,867 corr=24360	t=0,187 corr=185	t=0,016 corr=185
4×5 eq=168	t=8415,497 corr=335160	t=147,561 corr=335160	t=6,926 corr=706	t=0,281 corr=706
4×6 eq=282	–	–	t=142,600 corr=1639	t=5,117 corr=1639
5×5 eq=1394	–	–	t=3950,916 corr=9109	t=52,603 corr=9109

Численные эксперименты показали, что при применении эвристики 2 метод `FindEquivMatr` работает быстрее, если перебор матриц во вложенном цикле выполнять в обратном порядке (то есть в порядке убывания индексов `j` для элементов списка `result`).

7. ЗАКЛЮЧЕНИЕ. ВОЗМОЖНЫЕ НАПРАВЛЕНИЯ ДАЛЬНЕЙШЕЙ РАБОТЫ

Итак, применение двух реализованных эвристик позволило обработать матрицы большей размерности за существенно меньшее время.

Однако при дальнейшем увеличении размерности матриц возникают новые проблемы. Например, для матриц размера 6 × 6 и 7 × 7 количество упорядоченных матриц равно соответственно 1 271 091 и 505 051 770. Понятно, что в такой ситуации не следует вначале формировать весь набор анализируемых матриц, а затем приступать к их попарному сравнению: необходимо сразу обрабатывать очередную построенную матрицу, а сохранять только набор уже найденных попарно неэквивалентных матриц. Учитывая экспоненциально растущее количество различных возможных перестановок, следует реализовать более «интеллектуальный» вариант их перебора, при котором анализируются только перестановки строк и столбцов с одинаковыми характеристиками (то есть с одинаковым количеством элементов, равных `true`). Разумеется некоторого ускорения можно также добиться за счёт распараллеливания алгоритма, при котором набор найденных матриц распределяется для анализа по нескольким потокам (`threads`), связанным с различными ядрами многоядерного процессора.

Повторим, что предметом проекта является *выбор и реализация эвристик, позволяющих сильно уменьшить полный перебор*. В частности, было бы очень полезно получить ответ для размерности 8 × 10. Ещё раз отметим, что хорошо надо организовывать не только алгоритмы, но и структуры данных.

Список литературы

1. *Медведев Ю.* О классе событий, допускающих представление в конечном автомате / Автоматы. Сборник статей. М.: Иностранная литература, 1956. С. 385–401.
2. Лекции лауреатов премии Тьюринга за первые двадцать лет. М.: Мир, 1993.
3. *Ахо А., Ульман Дж.* Теория синтаксического анализа, перевода и компиляции. Т. 1. М.: Мир, 1978. 615 с.
4. *Хопкрофт Дж., Мотвани Р., Ульман Дж.* Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2002. 528 с.
5. *Jiang T., Ravikumar B.* Minimal NFA problems are hard // SIAM Journal on Computing. 1993. Vol. 22 № 6. P. 1117–1141.
6. *Polák L.* Minimalizations of NFA using the universal automaton // International Journal of Foundation of Computer Science. 2005. Vol. 16. № 5. P. 999–1010.
7. *Kameda T., Weiner P.* On the state minimization of nondeterministic finite automata // IEEE Transactions on Computers. 1970. Vol. C-19. № 7. P. 617–627.
8. *Lombardy S., Sakarovitch J.* The universal automaton. Logic and Automata // Amsterdam University Press, 2008. P. 457–504.
9. *Харари Ф.* Теория графов. М.: Едиториал, 2003. 296 с.
10. *Харари Ф., Палмер Э.* Перечисление графов. М.: Мир, 1977. 326 с.
11. *Gera R., Hedetniemi S., Larson C. (Eds.)* Graph Theory. Favorite Conjectures and Open Problems – 1. Basel: Springer, 2016. 291 p.
12. *Gera R., Haynes T., Hedetniemi S. (Eds.)* Graph Theory. Favorite Conjectures and Open Problems – 2. Basel: Springer, 2018. 281 p.
13. *Долгов В. Н., Мельников Б. Ф.* Построение универсального конечного автомата. I. От теории к практическим алгоритмам // Вестник Воронежского государственного университета. Серия: Физика. Математика. 2013. № 2. С. 173–181.
14. *Долгов В. Н., Мельников Б. Ф.* Построение универсального конечного автомата. II. Примеры работы алгоритмов // Вестник Воронежского государственного университета. Серия: Физика. Математика. 2014. № 1. С. 78–85.
15. *Melnikov B.* The complete finite automaton // International Journal of Open Information Technologies. 2017. Vol. 5. № 10. С. 9–17.
16. *Melnikov B., Melnikova E.* Waterloo-like finite automata and algorithms for their automatic construction // International Journal of Open Information Technologies. 2017. Vol. 5. № 12. P. 8–15.
17. *Криволапова А.В., Мельникова Е.А., Софонова Н.В.* Некоторые вспомогательные алгоритмы для построения Ватерлоо-подобных автоматов // Вестник Воронежского государственного университета. Серия: Системный анализ и информационные технологии. 2016. № 4. С. 20–28.
18. *Абрамян М. Э., Мельников Б. Ф., Тренина М. А.* Реализация метода ветвей и границ для задачи восстановления матрицы расстояний между последовательностями ДНК // Современные информационные технологии и ИТ-образование. 2019. Т. 15. № 1. С. 81-91.
19. *Melnikov B., Nichiporchuk A., Trenina M., Abramyan M.* Clustering of situations in solving applied optimization problems (on the examples of traveling salesman problem and distance matrix recovery) // International Journal of Open Information Technologies. 2019. Vol. 7. № 5. P. 1–8.

Поступила в редакцию 19.04.2019, окончательный вариант — 06.06.2019.

Computer tools in education, 2019
№ 2: 87–107
http://cte.eltech.ru
doi:10.32603/2071-2340-2019-2-87-107

Finite Automata Table: a Science Project for High School Students

Abramyan M. E.[1], associated professor, m-abramyan@yandex.ru
Melnikov B. F.[2], professor, bf-melnikov@yandex.ru
Melnikova E. A.[3], associated professor, ya.e.melnikova@yandex.ru

[1]Southern Federal University, 105/42, Bolshaya Sadovaya str., 344006, Rostov-on-Don, Russia
[2]Shenzhen MSU – BIT University, No. 1, International University Park Road, Dayun New Town, Longgang District, Shenzhen, Guangdong Province, PRC, 517182 Shenzhen, Chin
[3]Russian State Social University, 4, build. 1, Wilhelm Pieck street, 129226 Moscow, Russia

Abstract

Since the late 1960s, the problem of minimizing non-deterministic finite automata has been studied. In practical programs for large dimensions, obtaining an exact answer usually takes an unacceptably long time. In this regard, we are interested in, among others, heuristic algorithms for solving the problem, i.e. in algorithms that "do not promise anything", which, however, in practice in most cases, they give a solution that is close to optimal for an acceptable working time.

The project proposed for schoolchildren is aimed at a partial solution of one of the auxiliary tasks arising in the mentioned optimization problem. To do this, we define in a special way the equivalence relation on the set of tables of a given size $M \times N$ filled with elements 0 and 1. Obtaining the number of nonequivalent tables of dimension 8×10 will be a serious step on the way to proving the fact that the example of the "bad" automaton described in 1970 (the so-called Waterloo automaton) is the minimal possible example, not having "lesser" analogues.

To solve the problem, we first propose a bad algorithm, which consists in a simple enumeration of matrices. This algorithm works well on matrices of small dimensions, but, as usual in such situations, it works unacceptably long when moving to large dimensions. To reduce the operating time of the algorithm, we offer several heuristics, and present the results of the work of different versions of the program. The goal of the project is the creation of new heuristics, an even greater increase in the operating time of the program and, if possible, obtaining an answer (the number of tables) for the dimension 8×10.

For the majority of variants of the algorithm described in the paper, we present the implementation in C# using the principles of the object-oriented programming. We assume that further work on the project will consist in further modification of the programs we have provided.

Keywords: *optimization problem, finite automaton, heuristic algorithm, the first step in science.*

Citation: M. E. Abramyan, B. F. Melnikov, and E. A. Melnikova, "Finite Automata Table: a Science Project for High School Students," *Computer tools in education*, no. 2, pp. 87–107, 2019 (in Russian); doi:10.32603/2071-2340-2019-2-87-107

References

1. Yu. Medvedev, “A class of events that can be represented in a finite state machine”, In *Automata*, Inostrannaya literatura, Moscow, pp. 385–401, 1956.
2. Lectures by *Turing Prize winners in their first twenty years*, Mir, Moscow, 1993.
3. A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling*, vol. 1, Mir, Moscow, 1978.
4. J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Vilýams, Moscow, 2002.
5. T. Jiang and B. Ravikumar “Minimal NFA problems are hard,” *SIAM Journal on Computing*, vol. 22, no. 6, pp. 1117–1141, 1993.
6. L. Polák, “Minimalizations of NFA using the universal automaton,” *International Journal of Foundation of Computer Science*, vol.16, no. 5, pp. 999–1010, 2005.
7. T. Kameda and P. Weiner, “On the state minimization of nondeterministic finite automata,” *IEEE Transactions on Computers*, vol. C-19, no. 7, pp. 617–627, 1970.
8. S. Lombardy and J. Sakarovitch, “The universal automaton. Logic and Automata,” *Amsterdam University Press*, pp. 457–504, 2008.
9. F. Harary, *Graph Theory*, Editorial URSS, Moscow, 2003.
10. F. Harary and E. M. Palmer, *Graphical Enumeration*, Mir, Moscow, 1977.
11. R. Gera , S. Hedetniemi , C. Larson, Eds. *Graph Theory. Favorite Conjectures and Open Problems — 1*, Springer, Basel, 2016.
12. R. Gera, T. Haynes, S. Hedetniemi, Eds. *Graph Theory. Favorite Conjectures and Open Problems — 2*. Springer, Basel, 2018.
13. V. N. Dolgov and B. F. Melnikov, “Construction of universal finite automaton. I. From theory to the practical algorithm,” *Proceedings of Voronezh State University. Series: Physics. Mathematics*, no. 2, pp. 173–181, 2013 (in Russian).
14. V. N. Dolgov and B. F. Melnikov, “Construction of universal finite automaton. II. Examples of algorithms functioning,” *Proceedings of Voronezh State University. Series: Physics. Mathematics*. no. 1, pp. 78–85, 2014. (in Russian).
15. B. Melnikov, “The complete finite automaton,” *International Journal of Open Information Technologies*, vol. 5, no. 10, pp. 9–17, 2017.
16. B. Melnikov and E. Melnikova, “Waterloo-like finite automata and algorithms for their automatic construction,” *International Journal of Open Information Technologies*, vol. 5, no. 12, pp. 8–15, 2017.
17. A. V. Krivolapova, E. A. Melnikova, and N. V. Sofonova, “Some supporting algorithms of construction of waterloo-like automata,” *Proceedings of Voronezh State University. Series: Systems analysis and information technologies*, no. 4, pp. 20–28, 2016 (in Russian).
18. M. E. Abramyan, B. F. Melnikov, and M. A. Trenina, “Implementation of the Branch and Bound Method for the Problem of Recovering a Distances Matrix Between DNA Strings,” *Modern Information Technologies and IT-Education*, vol. 15, no. 1, 2019 (in Russian).
19. B. Melnikov, A. Nichiporchuk, M. Trenina, and M. Abramyan, “Clustering of situations in solving applied optimization problems (on the examples of traveling salesman problem and distance matrix recovery),” *International Journal of Open Information Technologies*, vol. 7, no. 5, pp. 1–8, 2019.

Received 19.04.2019, the final version — 06.06.2019.