Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

Лабораторная работа № 2

по курсу «Теория формальных языков»

Студент группы ИУ9-51Б Винокурова Е. С.

Преподаватель Непейвода А.Н.

Москва 2025

Содержание

1 Задача

По имеющемуся академическому регулярному выражению построить:

- Минимальный ДКА, распознающий его язык (минимальность обосновать таблицей классов эквивалентности)
- Возможно малый НКА, распознающий его язык. Возможно малый переключающийся (с конъюнкцией) КА, распознающий его язык. Частично обосновать таблицами множеств классов эквивалентности.
- Расширенное регулярное выражение, распознающее тот же язык. В расширенном выражении можно использовать:
  - wildcard-операцию . для замены произвольного символа алфавита;
  - положительную итерацию τ^+ и опцию $\tau?$. $\tau^+ = \tau\tau^*$, $\tau? = (\tau|\varepsilon)$;
  - операции предпросмотра $\tau_0(? = \tau_1)\tau_2 \equiv \tau_0((\tau_1.*) \cap \tau_2)$ и ретроспективной проверки $\tau_0(? <= \tau_1)\tau_2 \equiv (\tau_0 \cap (\tau_1.*))\tau_2$, а также их отрицательные версии $\tau_0(?!\tau_1)\tau_2 \equiv \tau_0((\tau_1.*) \cap \tau_2)$ и $\tau_0(? <!\tau_1)\tau_2 \equiv (\tau_0 \cap (\tau_1.*))\tau_2$
  - классы букв $[c_1 \dots c_k] \equiv (c_1|c_2| \dots |c_k)$ и их дополнения $[\hat{}c_1 \dots c_k]$.
  - (обязательно) маркеры начала и конца выражения ^ и \$.

Провести автоматическое тестирование предполагаемой эквивалентности построенных распознавателей. Тем самым необходимо построить алгоритмы, определяющие принадлежность слова языку академического регулярного выражения, ДКА, НКА и ПКА.

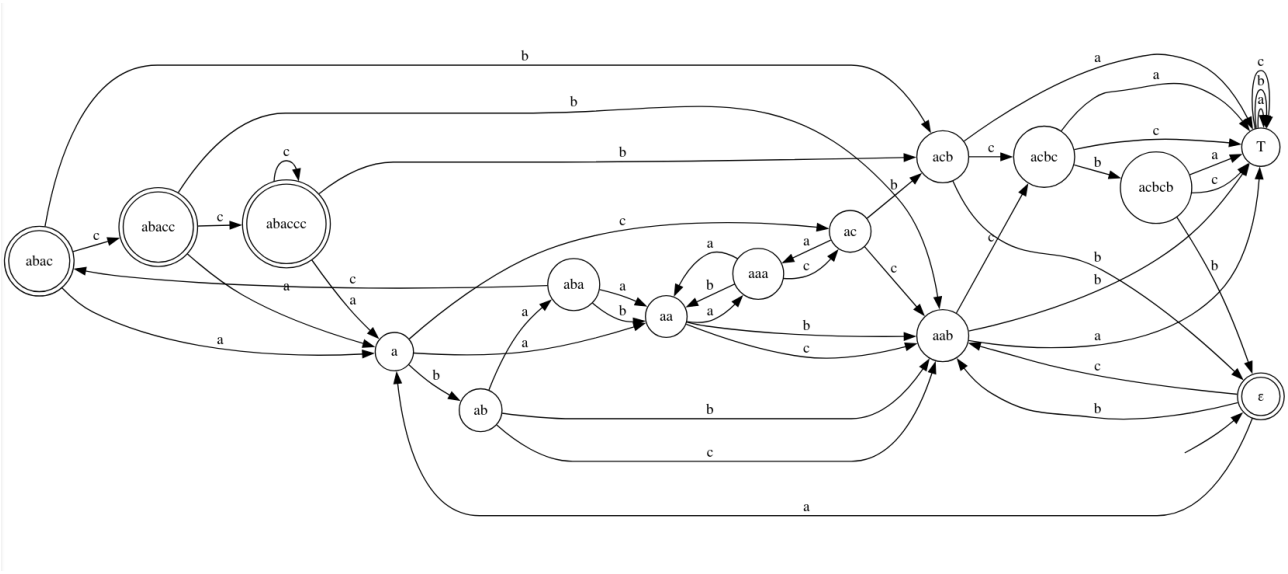
Требуется только фаза-тестирование эквивалентности: строится случайное слово ω и проверяется, принадлежит ли он языкам регулярного выражения, ДКА, НКА и ПКА согласованно.

Вариант 4

$$((aa|ab|ac)^*(bc|cc|ac)bb|abacc^*)^*$$

2 Минимальный ДКА

На основе академического регулярного выражения был построен недетерминированный конечный автомат, который затем детерминизирован и минимизирован; в результате получен минимальный детерминированный конечный автомат, распознающий данный язык. Минимальность ДКА доказана таблицей эквивалентности состояний.



Изображение этого ДКА находится в файле DKA.png

2.1 Обоснование минимальности ДКА

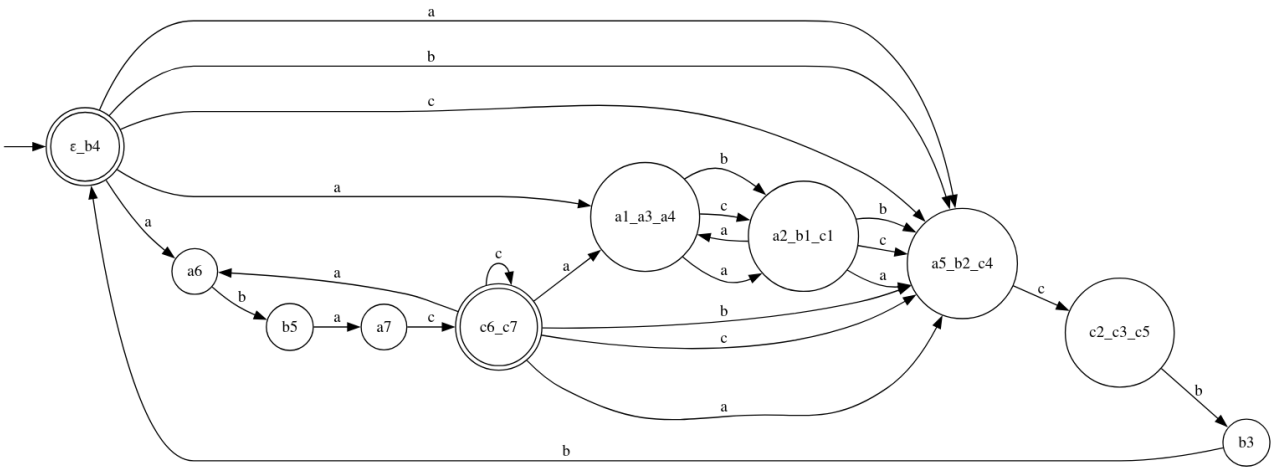
Таблица префиксов и суффиксов для ДКА

-	ε	b	bb	c	ac	$bcbb$	cbb	bac	$accbb$
$abac$	1	0	1	1	0	1	0	0	0
$abaccc$	1	0	1	1	0	1	1	0	0
$abacc$	1	0	0	1	0	1	1	0	0
ε	1	0	0	0	0	1	0	0	0
acb	0	1	0	0	0	0	1	0	0
$acbcb$	0	1	0	0	0	0	0	0	0
aba	0	0	0	1	0	0	1	0	1
ac	0	0	1	0	0	1	0	0	0
$acbc$	0	0	1	0	0	0	0	0	0
ab	0	0	0	0	1	1	0	0	0
a	0	0	0	0	0	0	1	1	1
aaa	0	0	0	0	0	0	1	0	1
aab	0	0	0	0	0	0	1	0	0
T	0	0	0	0	0	0	0	0	0
aa	0	0	0	0	0	1	0	0	0

Построенный ДКА является минимальным, поскольку все его состояния различимы. Это подтверждается таблицей префиксов и суффиксов, так как все строки в ней различны.

3 Возможно малый НКА

На основе академического регулярного выражения с использованием линеаризации был построен, а затем частично минимизирован недетерминированный конечный автомат распознающий его язык.



Изображение этого НКА находится в файле NKA.png

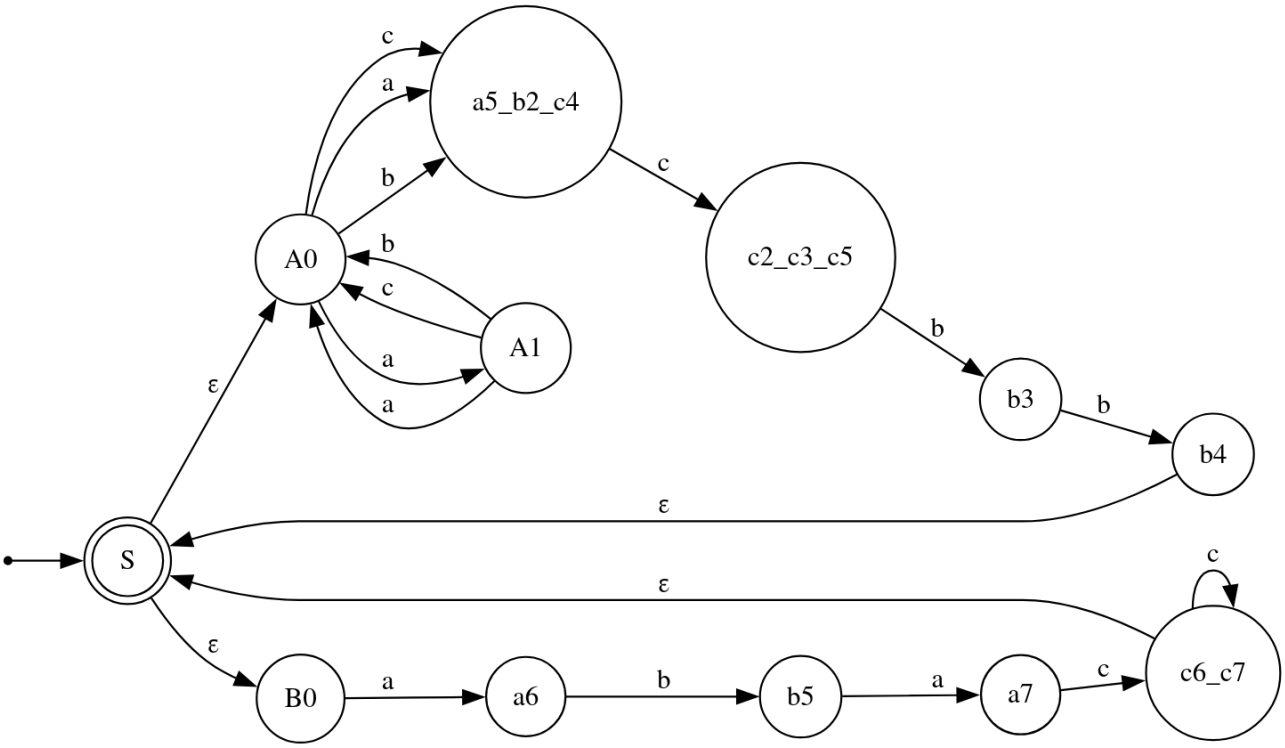
3.1 Частичное обоснование минимальности НКА

Таблица префиксов и суффиксов для НКА

-	bac	ac	b	c	ε	bb	cbb
a	1	0	0	0	0	0	1
ab	0	1	0	0	0	0	0
bcb	0	0	1	0	0	0	0
aba	0	0	0	1	0	0	1
ε	0	0	0	0	1	0	0
bc	0	0	0	0	0	1	0
b	0	0	0	0	0	0	1

4 Возможно малый ПКА

На основе академического регулярного выражения был построен переключающийся конечный автомат распознающий его язык.



Изображение этого ПКА находится в файле PKA.png

4.1 частичное обоснование минимальности ПКА

Таблица префиксов и суффиксов для ПКА

-	b	bb	$bcbb$	c	cbb
$abaccc$	0	1	1	1	1
$abacc$	0	0	1	1	1
aba	0	0	0	1	1
aaa	0	0	0	0	1
aa	0	0	1	0	0

5 Расширенное регулярное выражение

Исходное регулярное выражение

$$((aa|ab|ac)^*(bc|cc|ac)bb|abacc^*)^*$$

описывает произвольную (возможно пустую) последовательность блоков двух форм.

В первой форме блока подвыражение $(aa|ab|ac)$ представляет собой все пары символов, начинающиеся с a и продолжающиеся любой буквой из множества $\{a, b, c\}$. Это эквивалентно записи $(aa|ab|ac) \equiv a[abc]$

Повторение данной группы даёт $(aa|ab|ac)^* \equiv (a[abc])^*$

Аналогично, подвыражение $(bc|cc|ac)$ всегда имеет вид xc, где $x \in \{a, b, c\}$, то есть $(bc|cc|ac) \equiv [abc]c$

Таким образом, первая форма блока становится $(a[abc])^*[abc]cbb$

Вторая форма блока имеет вид $abacc^*$. Поскольку после обязательного символа c следует c^*, общее число букв c в конце не менее одного, следовательно, $abacc^* \equiv abac+$.

Объединяя обе формы, а также сохраняя внешнюю звезду, получаем эквивалентное расширенное регулярное выражение:

$$^\wedge((a[abc])^*[abc]cbb|abac^+)^*\$.$$

6 Фазз-тестирование

Для проверки эквивалентности построенных распознавателей было написано автоматическое тестирование.

Программа генерирует случайные слова над алфавитом a,b,c как полностью случайные, так и случайные, которые соответствуют регулярному выражению. Для каждого слова выполняется проверка принадлежности языку с использованием четырёх методов: регулярного выражения, минимального ДКА, построенного НКА, переключающегося конечного автомата.

Код программы представлен в Листинге 1.

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <cstdlib>
5 #include <ctime>
6 #include <regex>
7 #include <set>
8 #include <map>
```

```cpp
#include <queue>
#include <fstream>

using namespace std;

string generateWord(int maxBlocks = 5) {
    string result = "";
    string xParts[] = {"aa", "ab", "ac"};
    string yParts[] = {"bc", "cc", "ac"};
    int blocks = rand() % (maxBlocks + 1);
    for (int b = 0; b < blocks; ++b) {
        if (rand() % 2 == 0) {
            // (aa|ab|ac)*(ac|bc|cc)bb
            int xCount = rand() % 5;
            for (int i = 0; i < xCount; ++i) {
                result += xParts[rand() % 3];
            }
            result += yParts[rand() % 3];
            result += "bb";
        } else {
            // abacc*
            result += "aba";
            result += "c";
            int cCount = rand() % 6;
            for (int i = 0; i < cCount; ++i)
                result += 'c';
        }
    }
    return result;
}

string generateRandomWord(int n) {
    string word = "";
    string alphabet = "abc";
    for (int i = 0; i < n; ++i) {
        word += alphabet[rand() % 3];
    }
    return word;
}

bool checkRegex(const string& word) {
    regex re("((aa|ab|ac)*(bc|cc|ac)bb|abacc*)*");
    return regex_match(word, re);
}

int transitionDFA(int state, char symbol) {
```

```
55      int table[15][3] = {
56          {10, 4, 2},    // abac
57          {10, 4, 1},    // abaccc
58          {10, 12, 1},   // abacc
59          {10, 12, 12}, // epsilon
60          {13, 3, 8},    // acb
61          {13, 3, 13},   // acbcb
62          {14, 14, 0},   // aba
63          {11, 4, 12},   // ac
64          {13, 5, 13},   // acbc
65          {6, 12, 12},   // ab
66          {14, 9, 7},    // a
67          {14, 14, 7},   // aaa
68          {13, 13, 8},   // aab
69          {13, 13, 13}, // T
70          {11, 12, 12}   // aa
71      };
72      int col = (symbol == 'a' ? 0 : symbol == 'b' ? 1 : 2);
73      return table[state][col];
74  }
75
76  bool checkDFA(const string& word) {
77      int state = 3; //
78      vector<int> acceptingStates = {0, 1, 2, 3};
79      for (char c : word) {
80          state = transitionDFA(state, c);
81      }
82      for (int acc : acceptingStates) {
83          if (state == acc) return true;
84      }
85      return false;
86  }
87
88  set<int> moveNFA(const set<int>& states, char symbol) {
89      set<int> result;
90      for (int state : states) {
91          switch (state) {
92              case 0: // eps_b4
93                  if (symbol == 'a') { result.insert(1); result.insert(2); result.
    insert(3); }
94                  if (symbol == 'b') result.insert(2);
95                  if (symbol == 'c') result.insert(2);
96                  break;
97              case 1: // a1_a3_a4
98                  result.insert(4); // a2_b1_c1
99                  break;
```

```
100             case 2: // a5_b2_c4
101                 if (symbol == 'c') result.insert(5); // c2_c3_c5
102                 break;
103             case 3: // a6
104                 if (symbol == 'b') result.insert(6); // b5
105                 break;
106             case 4: // a2_b1_c1
107                 if (symbol == 'a') { result.insert(1); result.insert(2); }
108                 if (symbol == 'b') result.insert(2);
109                 if (symbol == 'c') result.insert(2);
110                 break;
111             case 5: // c2_c3_c5
112                 if (symbol == 'b') result.insert(7); // b3
113                 break;
114             case 6: // b5
115                 if (symbol == 'a') result.insert(8); // a7
116                 break;
117             case 7: // b3
118                 if (symbol == 'b') result.insert(0); // eps_b4_c7
119                 break;
120             case 8: // a7
121                 if (symbol == 'c') result.insert(9); // c6
122                 break;
123             case 9: // c6_c7
124                 if (symbol == 'a') { result.insert(1); result.insert(2); result.
    insert(3); }
125                 if (symbol == 'b') result.insert(2);
126                 if (symbol == 'c') { result.insert(2); result.insert(9); }
127                 break;
128         }
129     }
130     return result;
131 }
132
133 bool checkNFA(const string& word) {
134     set<int> currentStates = {0}; //
    eps_b4
135     set<int> acceptingStates = {0, 9}; // eps_b4    c6_c7 -
136     for (char c : word) {
137         currentStates = moveNFA(currentStates, c);
138     }
139     for (int st : currentStates) {
140         if (acceptingStates.count(st)) return true;
141     }
142     return false;
143 }
```

```
144
145 set<int> epsilonClosure(const set<int>& states, const map<int, set<int>>&
        epsilonTransitions) {
146     set<int> closure = states;
147     queue<int> q;
148     for (int st : states) q.push(st);
149     while (!q.empty()) {
150         int cur = q.front();
151         q.pop();
152         if (epsilonTransitions.count(cur)) {
153             for (int next : epsilonTransitions.at(cur)) {
154                 if (!closure.count(next)) {
155                     closure.insert(next);
156                     q.push(next);
157                 }
158             }
159         }
160     }
161     return closure;
162 }
163
164 set<int> moveSwitchingFA(const set<int>& states, char symbol,
165                          const map<pair<int,char>, set<int>>& transitions,
166                          const map<int, set<int>>& epsilonTransitions) {
167     set<int> nextStates;
168     for (int st : states) {
169         auto it = transitions.find({st, symbol});
170         if (it != transitions.end()) {
171             nextStates.insert(it->second.begin(), it->second.end());
172         }
173     }
174     return epsilonClosure(nextStates, epsilonTransitions);
175 }
176
177 bool checkSwitchingFA(const string& word) {
178     map<pair<int,char>, set<int>> transitions;
179     map<int,set<int>> epsilonTrans;
180
181     epsilonTrans[0] = {1,7}; // S -> A0, B0
182
183     // (aa|ab|ac)*
184     transitions[{1,'a'}] = {2};
185     transitions[{2,'a'}] = {1};
186     transitions[{2,'b'}] = {1};
187     transitions[{2,'c'}] = {1};
188
```

```
189     transitions[{1,'a'}].insert(3);
190     transitions[{1,'b'}].insert(3);
191     transitions[{1,'c'}].insert(3);
192     transitions[{3,'c'}] = {4};
193     transitions[{4,'b'}] = {5};
194     transitions[{5,'b'}] = {6};
195     epsilonTrans[6] = {0};
196 
197     // abacc*
198     transitions[{7,'a'}] = {8};
199     transitions[{8,'b'}] = {9};
200     transitions[{9,'a'}] = {10};
201     transitions[{10,'c'}] = {11};
202     transitions[{11,'c'}] = {11};
203     epsilonTrans[11] = {0};
204 
205     set<int> start = epsilonClosure({0}, epsilonTrans);
206     set<int> accepting = {0}; // S
207     set<int> currentStates = start;
208     for (char c : word) {
209         currentStates = moveSwitchingFA(currentStates, c, transitions,
    epsilonTrans);
210     }
211     for (int st : currentStates) if (accepting.count(st)) return true;
212     return false;
213 }
214 
215 int main() {
216     srand(time(0));
217 
218     int numWords = 10000;
219     int maxLength = 100;
220 
221     int mismatchesDFA=0;
222     int mismatchesNFA=0;
223     int mismatchesSwitch=0;
224 
225     ofstream logFile("mismatches.txt");
226 
227     for (int i=0;i<numWords;i++) {
228         string word;
229         if (rand() % 2 == 0) {
230             int length = rand()%maxLength + 10;
231             word = generateRandomWord(length);
232         } else {
233             word = generateWord();
```

```
234         }
235
236         bool regexResult = checkRegex(word);
237         if (regexResult) cout << word << endl;
238         bool dfaResult = checkDFA(word);
239         bool nfaResult = checkNFA(word);
240         bool switchResult = checkSwitchingFA(word);
241
242         if (regexResult != dfaResult) mismatchesDFA++;
243         if (regexResult != nfaResult) mismatchesNFA++;
244         if (regexResult != switchResult) mismatchesSwitch++;
245
246         if (regexResult != dfaResult || regexResult != nfaResult || regexResult
    != switchResult) {
247             logFile << "Word: " << word
248                     << " | Regex: " << regexResult
249                     << " | DFA: " << dfaResult
250                     << " | NFA: " << nfaResult
251                     << " | SwitchingFA: " << switchResult << endl;
252         }
253     }
254     logFile.close();
255     cout <<  "DFA mismatches: " << mismatchesDFA << endl;
256     cout <<  "NFA mismatches: " << mismatchesNFA << endl;
257     cout << "Switching FA mismatches: " << mismatchesSwitch << endl;
258
259     return 0;
260 }
```