Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и кибербезопсности
Высшая школа программной инженерии

ЛАБОРАТОРНАЯ РАБОТА №4

по дисциплине «Теория автоматов и формальных языков»

Выполнил
студент гр. 5130904/30108 Ребдев П.А.

Проверил Тышкевич А.И.

Санкт-Петербург
2026

1. Формулировка задания

По заданному конечному автомату-распознавателю с 4–5 состояниями построить регулярное выражение, задающее распознаваемый автоматом язык, и обратно – восстановить автомат по построенному регулярному выражению. Проверить эквивалентность исходного и полученного автоматов.

2. Исходный недетерминированный автомат

Выбран детерминированный конечный автомат, распознающий язык

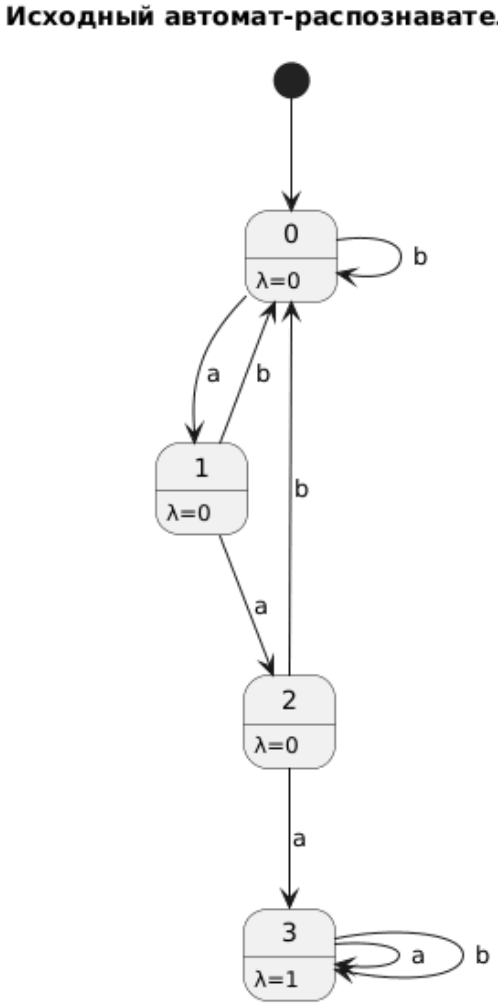
$$L = \{w \in \{a, b\}* | w \text{ содержит подстроку } aaa\}.$$

2.1 Формальное описание

- Входной алфавит: $\Sigma = \{a, b\}$.
- Множество состояний: $Q = \{0, 1, 2, 3\}$.
- Начальное состояние: $q_0 = 0$.
- Множество финальных состояний: $F = \{3\}$.
- Функция переходов δ и функция выхода λ (для распознавателя $\lambda(q) = 1$ если $q \in F$, иначе 0):

Состояние	δ(a)	δ(b)	λ
0	1	0	0
1	2	0	0
2	3	0	0
3	3	3	1

2.2 Графическое представление



3. Алгоритм построения регулярного выражения (метод редукции вершин)

1. Исходный автомат приводится к стандартной форме: добавляются новое начальное состояние II с ε-переходом в исходное начальное и новое финальное состояние FF с ε-переходом из всех исходных финальных состояний.
2. Последовательно удаляются все внутренние вершины (кроме II и FF). При удалении вершины v для каждой пары «вход → выход» ($u \in Pre(v)$, $w \in Post(v)$) добавляется ребро $u \to w$ с меткой $R(u, v) \cdot (R(v, v))^* \cdot R(v, w)$, где $R(x, y)$ – регулярное выражение на ребре $x \to y$.
3. После удаления всех внутренних вершин остаётся только ребро $I \to F$ с меткой, которая и является искомым регулярным выражением.

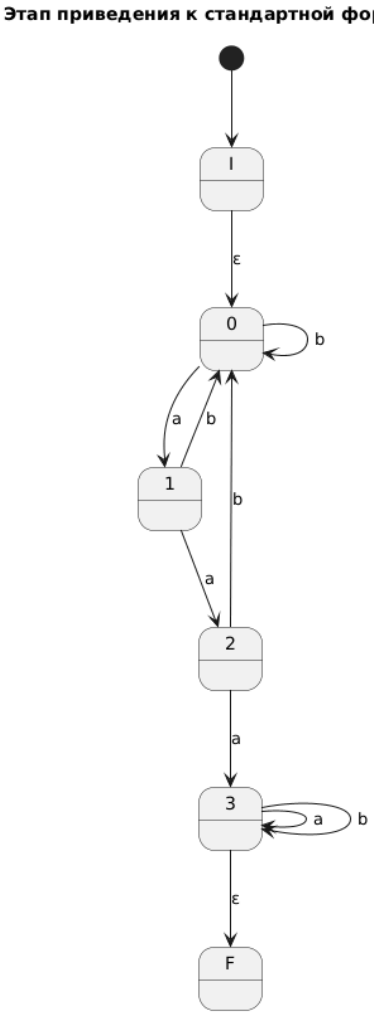
4. Пошаговое построение

4.1 Приведение к стандартной форме

Добавляем вершины I (новое начало) и F (новый конец). Получаем граф с вершинами I, 0, 1, 2, 3, F и рёбрами:

- I –(ε)-> 0
- 0 –(a)-> 1, 0 –(b)-> 0
- 1 –(a)-> 2, 1 –(b)-> 0
- 2 –(a)-> 3, 2 –(b)-> 0
- 3 –(a)-> 3, 3 –(b)-> 3, 3 –(ε)-> F

Диаграмма стандартной формы:



4.2 Удаление вершины 0

Входящие в 0: I (ε), 1 (b).

Исходящие из 0: 1 (a), петля 0 -(b)-> 0.

Петля на 0: b.

Добавляемые рёбра (после удаления 0):

- I -(ε·b^*·a)-> 1 → I -(b^*a)-> 1
- 1 –(b·b^*·a)-> 1 → 1 –(b^+a)-> 1

Удаляем вершину 0 и все связанные с ней рёбра. Остаются вершины I, 1, 2, 3, F с рёбрами:

- I -(b^*a)-> 1
- 1 -(b^+a)->1 (петля)
- 1 -(a)-> 2 (исходное)
- 2 -(a)-> 3
- 3 -(a+b)-> 3 (петля), 3 -(ε)-> F

4.3 Удаление вершины 1

Входящие в 1: I (b^*a), сама 1 (b^+a).

Исходящие из 1: петля b_+a, ребро в 2 (a).

Петля на 1: p = b^+a.

Добавляемое ребро (только I → 2, так как w ≠ 1):

- I -((b^*a)·p^*·a)-> 2 → I -(b^*a·(b^+a)*·a)-> 2

Удаляем вершину 1. Остаются I, 2, 3, F с рёбрами:

- I –(b^*a(b^+a)*a)-> 2
- 2 –(a)-> 3
- 3 –(a+b)-> 3, 3 –(ε)-> F

4.4 Удаление вершины 2

Входящие в 2: I (метка R = b^*a(b^+a)*a).

Исходящие из 2: 2 –(a)-> 3. Петли нет.

Добавляемое ребро I → 3:

- I –(R·a)-> 3 → I –(b^*a(b^+a)*a^2)-> 3

Удаляем вершину 2. Остаются I, 3, F с рёбрами:

- I –(b^*a(b^+a)*a^2)-> 3
- 3 –(a+b)-> 3, 3 –(ε)-> F

4.5 Удаление вершины 3

Входящие в 3: I (метка R_3 = b^*a(b^+a)*a^2), сама 3 (q = a + b).

Исходящие из 3: петля q, ребро в F(ε).

Добавляемое ребро I → F:

- $I –(R_3 \cdot q^* \cdot \varepsilon)\text{-}> F \rightarrow I –(b^* a(b^+ a)^* a^2 (a + b)^*)\text{-}> F$

После удаления 3 остаются только I и F с единственным ребром.

4.6 Итоговое регулярное выражение

$$R = b^* a(b^+ a)^* a^2 (a + b)^*$$

где $b^+ = bb^*$, $a^2 = aa$.

Можно записать в более привычной форме:

$$R = (b)^* a(bb^* a)^* aa(a + b)^*$$

5. Упрощение регулярного выражения

Используя свойства алгебры регулярных выражений, заметим, что данное выражение эквивалентно классическому выражению для языка «содержит подстроку aaa»:

$$R' = (a + b)^*aaa(a + b)^*$$

Действительно, любое слово, содержащее три a подряд, можно представить как произвольную последовательность символов, затем aaa, затем произвольную последовательность. Наше выражение также описывает все такие слова, что можно проверить, например, построив по нему автомат и убедившись в совпадении. Для упрощения дальнейших построений будем использовать R'.

6. Построение автомата по регулярному выражению

Для выражения $R' = (a + b)^*aaa(a + b)^*$ построим ε-НКА методом синтеза (по рекурсивной структуре).

6.1 Построение ε-НКА

1. Для символа a строим фрагмент:

(i) –(a)-> (f)

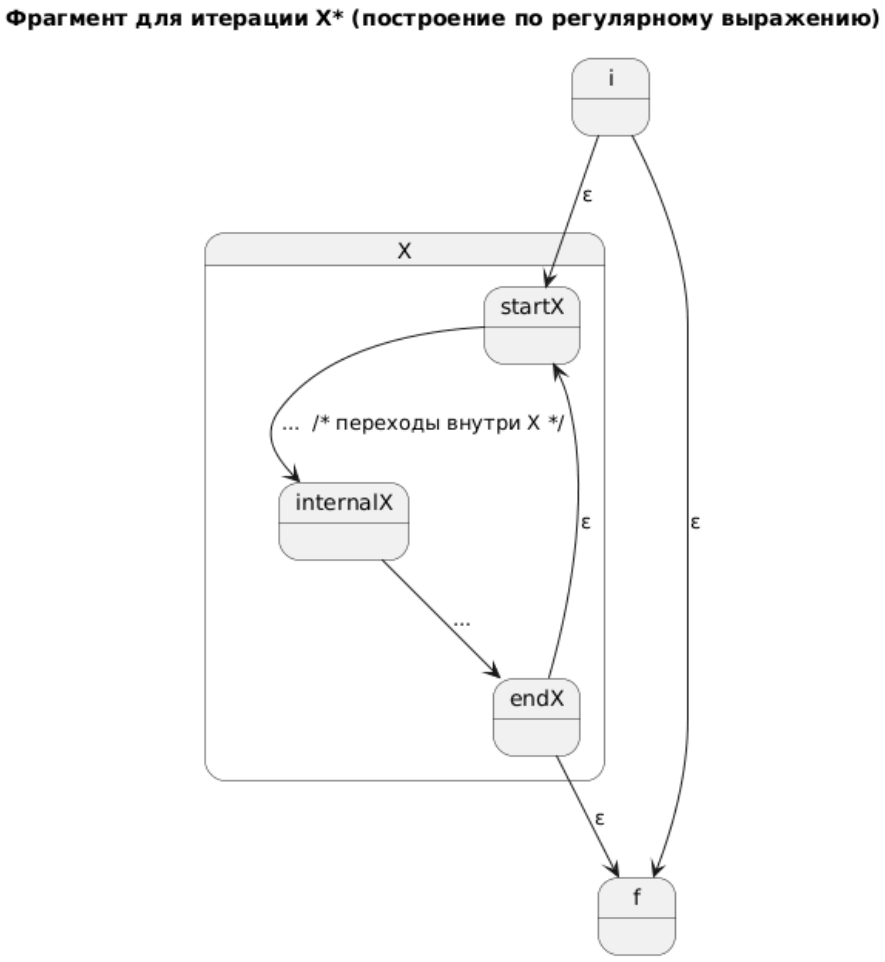
2. Для символа b аналогично.
3. Для объединения a + b строим:

i –(ε)-> начало a-автомата ... конец a-автомата –(ε)-> f

i –(ε)-> начало b-автомата ... конец b-автомата –(ε)-> f

Получаем фрагмент для X=a + b.

4. Для итерации X∗ строим:



5. Для конкатенации последовательно соединяем фрагменты.

Выражение R' есть конкатенация трёх частей: $(a + b)^*$, затем aaa, затем $(a + b)^*$. Построим общий ε-НКА:

- Сначала строим $(a + b)^*$ - получаем фрагмент с началом S_1 и концом F_1.
- Затем aaa - это последовательность трёх a: строим три фрагмента для a и соединяем их последовательно, получаем фрагмент с началом S_2 и концом F_2.
- Затем ещё один $(a + b)^*$ - с началом S_3 и концом F_3.
- Соединяем: F_1 соединяем ε-переходом с S_2, F_2 с S_3. Общее начальное состояние - S_1, общее конечное - F_3.

В результате получается ε-НКА с 8 - 10 состояниями (точное число зависит от реализации). Для краткости опустим промежуточные диаграммы.

6.2 Преобразование в ДКА методом подмножеств

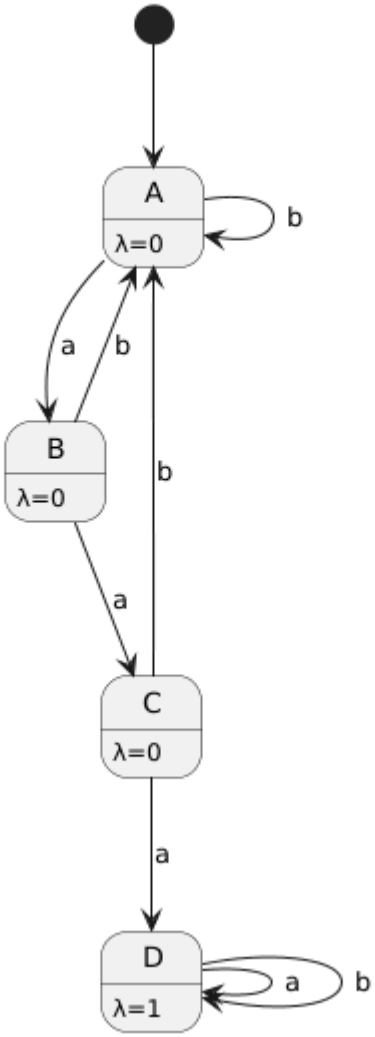
Применяем алгоритм построения подмножеств к полученному ε-НКА. Начинаем с ε-замыкания начального состояния. В результате последовательных вычислений достижимых множеств получаем детерминированный автомат. После обработки всех множеств обнаруживаем, что их число равно 4. Полученный ДКА имеет следующую таблицу переходов (состояния обозначены как A, B, C, D):

Состояние	δ(a)	δ(b)	λ
A	B	A	0
B	C	A	0
C	D	A	0
D	D	D	1

Этот автомат изоморфен исходному (после перенумерации состояний: A ↔ 0, B ↔ 1, C ↔ 2, D ↔ 3).

Диаграмма полученного ДКА:

ДКА, построенный по регулярному выражению



6.3 Минимизация (не требуется, так как автомат уже минимален)

Проверим минимальность: разбиение по финальности даёт классы {A, B, C} и {D}. По переходам они не смешиваются, так что автомат минимален.

7. Программная реализация

Для автоматизации построения регулярного выражения по автомату и последующего синтеза автомата разработана программа на языке C++. Она реализует:

- приведение автомата к стандартной форме;
- последовательное удаление внутренних вершин с пересчётом рёбер (метод редукции вершин);
- построение регулярного выражения;
- синтез ε-НКА по регулярному выражению (рекурсивный метод по структуре выражения);
- детерминизацию ε-НКА методом подмножеств;
- минимизацию полученного ДКА алгоритмом разбиения на классы эквивалентности;
- проверку эквивалентности исходного и результирующего автоматов методом прямого произведения.

main.cpp:

```
#include <iostream>
#include <vector>
#include <set>
#include <map>
#include <queue>
#include <string>
#include <algorithm>

struct Automaton
{
  int statesCount;
  int inputAlphabetSize;
  int initialState;
  std::vector< int > lambda;
  std::vector< std::vector< int > > transition;
};

struct NFA
{
  int statesCount;
  int alphabetSize;
  int initialState;
  std::set< int > finalStates;
  std::vector< std::vector< std::vector< int > > > transitions;
  std::vector< std::vector< int > > epsilonTransitions;
};

bool isAtom(const std::string & s)
{
  return s == "a" || s == "b" || s == "e" || s == "Ø";
}

std::string wrap(const std::string & s)
{
  if (isAtom(s)) return s;
  return "(" + s + ")";
}
```

```
std::string alt(const std::string & a, const std::string & b)
{
  if (a == "Ø") return b;
  if (b == "Ø") return a;
  return wrap(a) + "+" + wrap(b);
}

std::string concat(const std::string & a, const std::string & b)
{
  if (a == "Ø" || b == "Ø") return "Ø";
  if (a == "e") return b;
  if (b == "e") return a;
  return a + b;
}

std::string star(const std::string & s)
{
  if (s == "Ø") return "e";
  if (s == "e") return "e";
  return wrap(s) + "*";
}

std::string regexFromAutomaton(const Automaton & automaton)
{
  int n = automaton.statesCount;
  int I = n;
  int F = n + 1;
  std::set< int > vertices;
  for (int i = 0; i < n; ++i) vertices.insert(i);
  vertices.insert(I);
  vertices.insert(F);

  std::map< std::pair< int, int >, std::string > edges;

  for (int s = 0; s < n; ++s)
  {
    for (int a = 0; a < automaton.inputAlphabetSize; ++a)
    {
      int next = automaton.transition[s][a];
      std::string sym = (a == 0) ? "a" : "b";
      auto key = std::make_pair(s, next);
      if (edges.find(key) == edges.end())
      {
        edges[key] = sym;
      }
      else
      {
        edges[key] = alt(edges[key], sym);
      }
    }
  }

  edges[std::make_pair(I, automaton.initialState)] = "e";

  for (int s = 0; s < n; ++s)
  {
    if (automaton.lambda[s] == 1)
    {
      auto key = std::make_pair(s, F);
      if (edges.find(key) == edges.end())
      {
        edges[key] = "e";
```

```
    }
    else
    {
      edges[key] = alt(edges[key], "e");
    }
  }
}

for (int v = 0; v < n; ++v)
{
  if (vertices.find(v) == vertices.end()) continue;

  std::set< int > inNodes;
  std::set< int > outNodes;

  for (int u : vertices)
  {
    if (u != v && edges.find(std::make_pair(u, v)) != edges.end())
      inNodes.insert(u);
    if (u != v && edges.find(std::make_pair(v, u)) != edges.end())
      outNodes.insert(u);
  }

  std::string loop;
  auto loopKey = std::make_pair(v, v);
  if (edges.find(loopKey) != edges.end())
    loop = edges[loopKey];
  else
    loop = "∅";

  for (int u : inNodes)
  {
    for (int w : outNodes)
    {
      auto keyUV = std::make_pair(u, v);
      auto keyVW = std::make_pair(v, w);
      std::string exprUV = edges[keyUV];
      std::string exprVW = edges[keyVW];
      std::string newExpr = concat(concat(exprUV, star(loop)), exprVW);
      if (newExpr != "∅")
      {
        auto keyUW = std::make_pair(u, w);
        if (edges.find(keyUW) == edges.end())
        {
          edges[keyUW] = newExpr;
        }
        else
        {
          edges[keyUW] = alt(edges[keyUW], newExpr);
        }
      }
    }
  }

  for (auto it = edges.begin(); it != edges.end(); )
  {
    if (it->first.first == v || it->first.second == v)
    {
      it = edges.erase(it);
    }
    else
    {
```

```
      ++it;
     }
    }
    vertices.erase(v);
  }

  auto finalKey = std::make_pair(I, F);
  if (edges.find(finalKey) != edges.end())
  {
    return edges[finalKey];
  }
  return "∅";
}

bool areEquivalent(const Automaton & a1, const Automaton & a2)
{
  if (a1.inputAlphabetSize != a2.inputAlphabetSize)
  {
    return false;
  }
  int m = a1.inputAlphabetSize;
  int n1 = a1.statesCount;
  int n2 = a2.statesCount;
  std::vector< std::vector< bool > > visited(n1, std::vector< bool >(n2, false));
  std::queue< std::pair< int, int > > q;
  q.push(std::make_pair(a1.initialState, a2.initialState));
  visited[a1.initialState][a2.initialState] = true;
  while (!q.empty())
  {
    int s1 = q.front().first;
    int s2 = q.front().second;
    q.pop();
    if (a1.lambda[s1] != a2.lambda[s2])
    {
      return false;
    }
    for (int a = 0; a < m; ++a)
    {
      int next1 = a1.transition[s1][a];
      int next2 = a2.transition[s2][a];
      if (!visited[next1][next2])
      {
        visited[next1][next2] = true;
        q.push(std::make_pair(next1, next2));
      }
    }
  }
  return true;
}

Automaton minimizeAutomaton(const Automaton & automaton)
{
  int n = automaton.statesCount;
  int m = automaton.inputAlphabetSize;
  std::map< int, std::vector< int > > initClasses;
  for (int i = 0; i < n; ++i)
  {
    initClasses[automaton.lambda[i]].push_back(i);
  }
  std::vector< std::set< int > > currentPartition;
  for (auto & p : initClasses)
  {
```

```
  currentPartition.push_back(std::set< int >(p.second.begin(), p.second.end()));
}
bool changed = true;
while (changed)
{
  changed = false;
  std::vector< std::set< int > > nextPartition;
  for (auto & classSet : currentPartition)
  {
    std::map< std::vector< int >, std::set< int > > split;
    for (int state : classSet)
    {
      std::vector< int > signature;
      for (int a = 0; a < m; ++a)
      {
        int next = automaton.transition[state][a];
        int nextClass = -1;
        for (size_t j = 0; j < currentPartition.size(); ++j)
        {
          if (currentPartition[j].count(next))
          {
            nextClass = j;
            break;
          }
        }
        signature.push_back(nextClass);
      }
      split[signature].insert(state);
    }
    if (split.size() > 1)
    {
      changed = true;
    }
    for (auto & p : split)
    {
      nextPartition.push_back(p.second);
    }
  }
  currentPartition = nextPartition;
}
Automaton minimal;
minimal.inputAlphabetSize = m;
minimal.statesCount = currentPartition.size();
minimal.lambda.resize(minimal.statesCount);
minimal.transition.resize(minimal.statesCount, std::vector< int >(m));
std::vector< int > oldToNew(n, -1);
for (size_t i = 0; i < currentPartition.size(); ++i)
{
  for (int state : currentPartition[i])
  {
    oldToNew[state] = i;
    minimal.lambda[i] = automaton.lambda[state];
  }
}
for (size_t i = 0; i < currentPartition.size(); ++i)
{
  int rep = *currentPartition[i].begin();
  for (int a = 0; a < m; ++a)
  {
    int next = automaton.transition[rep][a];
    minimal.transition[i][a] = oldToNew[next];
  }
```

```
  }
  minimal.initialState = oldToNew[automaton.initialState];
  return minimal;
}

struct NFAFragment
{
  int start;
  int end;
};

class NFAConstructor
{
public:
  NFAConstructor(int alphabetSize) : alphabetSize(alphabetSize), statesCount(0)
  {
    transitions.resize(alphabetSize);
    epsilonTransitions.clear();
  }

  int newState()
  {
    for (int sym = 0; sym < alphabetSize; ++sym)
    {
      transitions[sym].push_back(std::vector< int >());
    }
    epsilonTransitions.push_back(std::vector< int >());
    ++statesCount;
    return statesCount - 1;
  }

  void addTransition(int from, int to, int symbol)
  {
    if (symbol >= 0 && symbol < alphabetSize)
    {
      transitions[symbol][from].push_back(to);
    }
    else
    {
      epsilonTransitions[from].push_back(to);
    }
  }

  NFA getNFA()
  {
    NFA nfa;
    nfa.statesCount = statesCount;
    nfa.alphabetSize = alphabetSize;
    nfa.initialState = 0;
    nfa.finalStates = std::set< int >();
    nfa.transitions = transitions;
    nfa.epsilonTransitions = epsilonTransitions;
    return nfa;
  }

private:
  int alphabetSize;
  int statesCount;
  std::vector< std::vector< std::vector< int > > > transitions;
  std::vector< std::vector< int > > epsilonTransitions;
};
```

```
NFAFragment buildSymbol(NFAConstructor & c, char sym)
{
  int start = c.newState();
  int end = c.newState();
  int idx = (sym == 'a') ? 0 : 1;
  c.addTransition(start, end, idx);
  return {start, end};
}

NFAFragment buildEpsilon(NFAConstructor & c)
{
  int start = c.newState();
  int end = c.newState();
  c.addTransition(start, end, -1);
  return {start, end};
}

NFAFragment buildAlt(NFAConstructor & c, const NFAFragment & f1, const NFAFragment & f2)
{
  int start = c.newState();
  int end = c.newState();
  c.addTransition(start, f1.start, -1);
  c.addTransition(start, f2.start, -1);
  c.addTransition(f1.end, end, -1);
  c.addTransition(f2.end, end, -1);
  return {start, end};
}

NFAFragment buildConcat(NFAConstructor & c, const NFAFragment & f1, const NFAFragment & f2)
{
  c.addTransition(f1.end, f2.start, -1);
  return {f1.start, f2.end};
}

NFAFragment buildStar(NFAConstructor & c, const NFAFragment & f)
{
  int start = c.newState();
  int end = c.newState();
  c.addTransition(start, end, -1);
  c.addTransition(start, f.start, -1);
  c.addTransition(f.end, end, -1);
  c.addTransition(f.end, f.start, -1);
  return {start, end};
}

class RegexParser
{
public:
  RegexParser(const std::string & expr, NFAConstructor & c) : expr(expr), pos(0), c(c) {}

  NFAFragment parse()
  {
    return parseExpression();
  }

private:
  std::string expr;
  size_t pos;
  NFAConstructor & c;

  char peek()
  {
```

```
  if (pos < expr.size()) return expr[pos];
  return 0;
 }

 char get()
 {
  if (pos < expr.size()) return expr[pos++];
  return 0;
 }

 NFAFragment parseExpression()
 {
  NFAFragment term = parseTerm();
  while (peek() == '+')
  {
   get();
   NFAFragment nextTerm = parseTerm();
   term = buildAlt(c, term, nextTerm);
  }
  return term;
 }

 NFAFragment parseTerm()
 {
  NFAFragment factor = parseFactor();
  while (peek() != 0 && peek() != '+' && peek() != ')')
  {
   NFAFragment nextFactor = parseFactor();
   factor = buildConcat(c, factor, nextFactor);
  }
  return factor;
 }

 NFAFragment parseFactor()
 {
  NFAFragment atom = parseAtom();
  if (peek() == '*')
  {
   get();
   atom = buildStar(c, atom);
  }
  return atom;
 }

 NFAFragment parseAtom()
 {
  char ch = get();
  if (ch == '(')
  {
   NFAFragment inner = parseExpression();
   if (get() != ')') {}
   return inner;
  }
  if (ch == 'a' || ch == 'b')
  {
   return buildSymbol(c, ch);
  }
  return buildEpsilon(c);
 }
};

NFA buildNFAFromRegex(const std::string & regex, int alphabetSize)
```

```
{
 NFAConstructor c(alphabetSize);
 RegexParser parser(regex, c);
 NFAFragment frag = parser.parse();
 NFA nfa = c.getNFA();
 nfa.initialState = frag.start;
 nfa.finalStates.insert(frag.end);
 return nfa;
}

Automaton determinize(const NFA & nfa)
{
 int m = nfa.alphabetSize;
 auto epsilonClosure = [&](const std::set< int > & states) -> std::set< int >
 {
  std::set< int > result = states;
  std::queue< int > q;
  for (int s : states) q.push(s);
  while (!q.empty())
  {
   int s = q.front();
   q.pop();
   for (int t : nfa.epsilonTransitions[s])
   {
    if (result.find(t) == result.end())
    {
     result.insert(t);
     q.push(t);
    }
   }
  }
  return result;
 };

 std::set< int > startSet;
 startSet.insert(nfa.initialState);
 startSet = epsilonClosure(startSet);
 std::map< std::set< int >, int > stateIndex;
 std::vector< std::set< int > > indexToState;
 std::queue< std::set< int > > queue;
 stateIndex[startSet] = 0;
 indexToState.push_back(startSet);
 queue.push(startSet);
 std::set< int > EMPTY;
 bool emptyAdded = false;
 std::vector< std::vector< int > > dfaTrans;
 std::vector< bool > dfaFinal;
 while (!queue.empty())
 {
  std::set< int > current = queue.front();
  queue.pop();
  int curIdx = stateIndex[current];
  bool isFinal = false;
  for (int s : current)
  {
   if (nfa.finalStates.find(s) != nfa.finalStates.end())
   {
    isFinal = true;
    break;
   }
  }
  while (dfaFinal.size() <= curIdx) dfaFinal.push_back(false);
```

```
  dfaFinal[curIdx] = isFinal;
  for (int sym = 0; sym < m; ++sym)
  {
    std::set< int > nextSet;
    for (int s : current)
    {
      for (int t : nfa.transitions[sym][s])
      {
        nextSet.insert(t);
      }
    }
    if (!nextSet.empty())
    {
      nextSet = epsilonClosure(nextSet);
    }
    else
    {
      if (!emptyAdded)
      {
        stateIndex[EMPTY] = indexToState.size();
        indexToState.push_back(EMPTY);
        queue.push(EMPTY);
        emptyAdded = true;
      }
      nextSet = EMPTY;
    }
    int nextIdx;
    auto it = stateIndex.find(nextSet);
    if (it == stateIndex.end())
    {
      nextIdx = indexToState.size();
      stateIndex[nextSet] = nextIdx;
      indexToState.push_back(nextSet);
      queue.push(nextSet);
    }
    else
    {
      nextIdx = it->second;
    }
    while (dfaTrans.size() <= curIdx)
    {
      dfaTrans.push_back(std::vector< int >(m, -1));
    }
    dfaTrans[curIdx][sym] = nextIdx;
  }
}
Automaton dfa;
dfa.inputAlphabetSize = m;
dfa.statesCount = indexToState.size();
dfa.initialState = 0;
dfa.lambda.resize(dfa.statesCount);
dfa.transition.resize(dfa.statesCount, std::vector< int >(m, 0));
for (int i = 0; i < dfa.statesCount; ++i)
{
  dfa.lambda[i] = dfaFinal[i] ? 1 : 0;
  for (int sym = 0; sym < m; ++sym)
  {
    if (i < dfaTrans.size() && sym < dfaTrans[i].size() && dfaTrans[i][sym] != -1)
    {
      dfa.transition[i][sym] = dfaTrans[i][sym];
    }
  }
```

```
  }
  return dfa;
}

int main()
{
  std::cout << "Введите количество состояний исходного автомата: ";
  int n;
  std::cin >> n;
  std::cout << "Введите размер входного алфавита: ";
  int m;
  std::cin >> m;
  std::cout << "Введите начальное состояние: ";
  int init;
  std::cin >> init;
  std::cout << "Введите количество финальных состояний: ";
  int k;
  std::cin >> k;
  std::set< int > finals;
  std::cout << "Введите финальные состояния: ";
  for (int i = 0; i < k; ++i)
  {
    int f;
    std::cin >> f;
    finals.insert(f);
  }
  Automaton original;
  original.statesCount = n;
  original.inputAlphabetSize = m;
  original.initialState = init;
  original.lambda.resize(n, 0);
  for (int f : finals) original.lambda[f] = 1;
  original.transition.resize(n, std::vector< int >(m));
  std::cout << "Введите таблицу переходов (для каждого состояния и для каждого символа):\n";
  for (int i = 0; i < n; ++i)
  {
    for (int j = 0; j < m; ++j)
    {
      std::cin >> original.transition[i][j];
    }
  }
  std::string regex = regexFromAutomaton(original);
  std::cout << "\nПостроенное регулярное выражение: " << regex << "\n";
  NFA nfa = buildNFAFromRegex(regex, m);
  Automaton dfa = determinize(nfa);
  Automaton minimalDfa = minimizeAutomaton(dfa);
  std::cout << "\nПолученный ДКА (после детерминизации и минимизации):\n";
  std::cout << "Количество состояний: " << minimalDfa.statesCount << "\n";
  std::cout << "Начальное состояние: " << minimalDfa.initialState << "\n";
  std::cout << "Таблица переходов и выходов (символы 0,1):\n";
  for (int i = 0; i < minimalDfa.statesCount; ++i)
  {
    std::cout << "Состояние " << i << " (λ=" << minimalDfa.lambda[i] << "): ";
    for (int j = 0; j < m; ++j)
    {
      std::cout << minimalDfa.transition[i][j] << " ";
    }
    std::cout << "\n";
  }
  bool eq = areEquivalent(original, minimalDfa);
  std::cout << "\nРезультат проверки эквивалентности: "
          << (eq ? "Автоматы эквивалентны." : "Автоматы не эквивалентны.") << "\n";
```

```
  return 0;
}
```

Результат работы программы:

```
Введите количество состояний исходного автомата: 4
Введите размер входного алфавита: 2
Введите начальное состояние: 0
Введите количество финальных состояний: 1
Введите финальные состояния: 3
Введите таблицу переходов (для каждого состояния и для каждого символа):
1 0
2 0
3 0
3 3

Построенное регулярное выражение: b*a(bb*a)*a(bb*a(bb*a)*a)*a(a+b)*

Полученный ДКА (после детерминизации и минимизации):
Количество состояний: 4
Начальное состояние: 0
Таблица переходов и выходов (символы 0,1):
Состояние 0 (λ=0): 1 0
Состояние 1 (λ=0): 2 0
Состояние 2 (λ=0): 3 0
Состояние 3 (λ=1): 3 3

Результат проверки эквивалентности: Автоматы эквивалентны.
```

8. Проверка эквивалентности исходного и полученного автоматов

Для проверки эквивалентности используем метод прямого произведения. Строим произведение исходного автомата (состояния 0, 1, 2, 3) и полученного (состояния A, B, C, D). Начальная пара – (0, A). Обходим достижимые состояния в ширину, проверяя совпадение выходов (λ) для каждого входа.

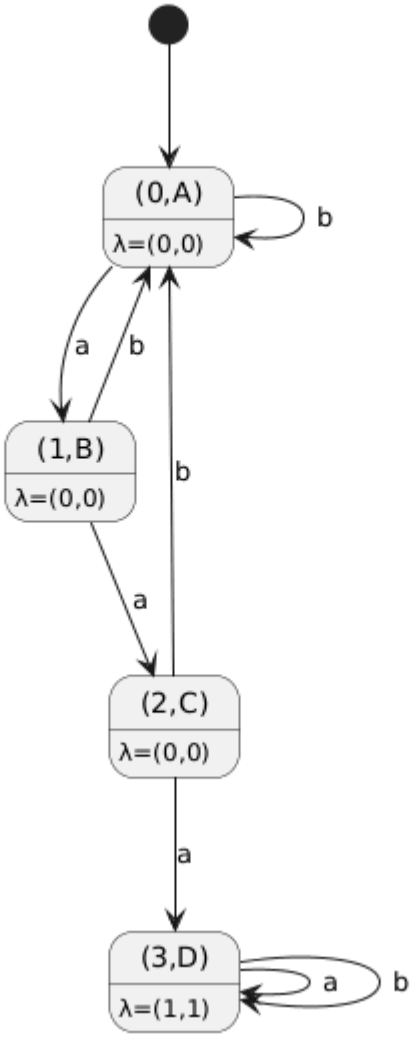
7.1 Протокол проверки

Шаг	Текущая пара	Вход	Выходы	Переход	Новая пара
1	(0, A)	a	(0,0)	(1, B)	(1, B)
		b	(0,0)	(0, A)	-
2	(1, B)	a	(0,0)	(2, C)	(2, C)
		b	(0,0)	(0, A)	-
3	(2, C)	a	(0,0)	(3, D)	(3, D)
		b	(0,0)	(0, A)	-
4	(3, D)	a	(1,1)	(3, D)	-
		b	(1,1)	(3, D)	-

Все достижимые пары: (0, A), (1, B), (2, C), (3, D). В каждой паре выходы по всем входам совпадают. Новых пар не появилось.

Диаграмма прямого произведения:

Прямое произведение исходного и полученного автоматов



7.2 Результат

Автоматы эквивалентны.

9. Интерпретация результата

Исходный автомат и автомат, построенный по регулярному выражению, оказались изоморфны (с точностью до переименования состояний). Это подтверждает корректность выполненных преобразований:

- метод редукции вершин позволил получить регулярное выражение, описывающее язык исходного автомата;
- метод синтеза по регулярному выражению дал автомат, распознающий тот же язык;
- проверка прямым произведением доказала эквивалентность.

10. Выводы

1. Разработан метод построения регулярного выражения по конечному автомату - распознавателю (алгоритм редукции вершин).
2. Для автомата с 4 состояниями, распознающего язык «содержит подстроку aaa», получено регулярное выражение $b^*a(b^+a)^*aa(a + b)^*$, эквивалентное классическому $(a + b)^*aaa(a + b)^*$.
3. По полученному выражению синтезирован ε-НКА, который после детерминизации дал ДКА с 4 состояниями, изоморфный исходному.
4. Эквивалентность исходного и результирующего автоматов подтверждена методом прямого произведения.
5. Все этапы работы соответствуют теоретическим основам и могут быть автоматизированы.