**Лабораторная работа №1. Распознавание цепочек   
регулярного языка**

Студент ИУ7-22М Лахов А.Е.

**Задание**

1. Преобразует регулярное выражение непосредственно в ДКА.
2. По ДКА строит эквивалентный ему КА, имеющий наименьшее возможное количество состояний. Указание. Воспользоваться алгоритмом, приведенным по адресу: [https://neerc.ifmo.ru/wiki/index.php?title=Минимизация\_ДКА,\_алгоритм\_за\_O(n^2)\_с\_построением\_пар\_различимых\_состояний](https://neerc.ifmo.ru/wiki/index.php?title=Минимизация_ДКА,_алгоритм_за_O(n%5e2)_с_построением_пар_различимых_состояний)
3. Моделирует минимальный КА для входной цепочки из терминалов исходной грамматики.

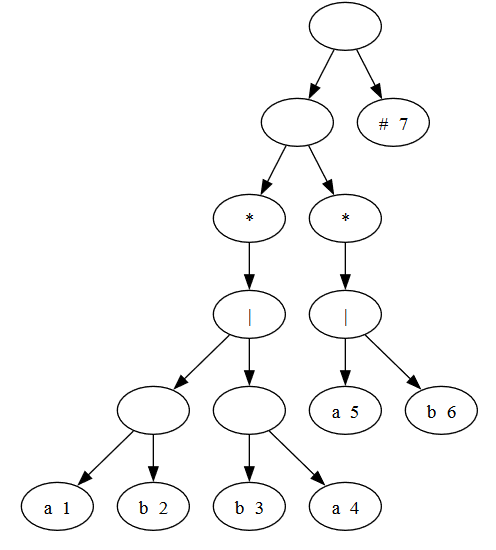
**Тестирование**

|  |  |  |  |
| --- | --- | --- | --- |
| **Регулярное  выражение** | **Проверяемое  выражение** | **Ожидаемый  результат** | **Результат** |
| ((ab)|(ba))\*(a|b)\* | abbaa | OK | OK |
| ab | OK | OK |
| ba | OK | OK |
| (a|b)\*abb | aabb | OK | OK |
| babb | OK | OK |
| abb | OK | OK |
| aaabb | OK | OK |
| bbabb | OK | OK |
| bbaaab | INVALID STRING | INVALID STRING |
| ab | INVALID STRING | INVALID STRING |
| a+b+c | aaabbc | OK | OK |
| abbccc | INVALID STRING | INVALID STRING |
| abca | INVALID STRING | INVALID STRING |
| ab | INVALID STRING | INVALID STRING |
| a\*b\*c\* | abc | OK | OK |
| bbbcccc | OK | OK |
| cccccccc | OK | OK |
| aaabbc | OK | OK |
| aaccbb | INVALID STRING | INVALID STRING |

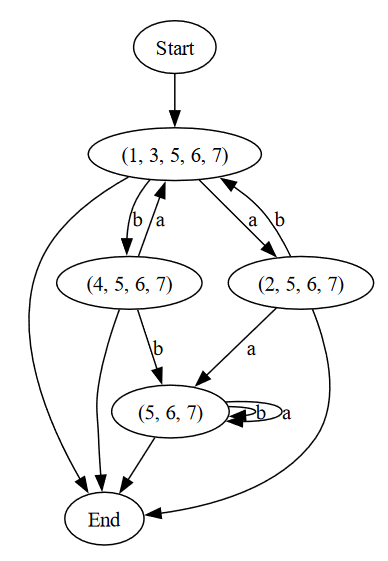
**Результаты работы**

результаты работы для регулярного выражения ((ab)|(ba))\*(a|b)\*

1. Дерево разбора (‘#’ – символ окончания выражения)

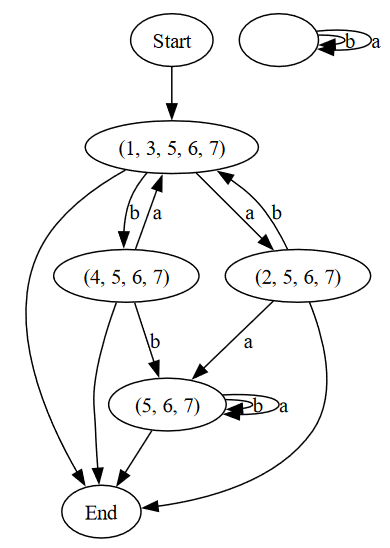


2. Детерминированный конечный автомат

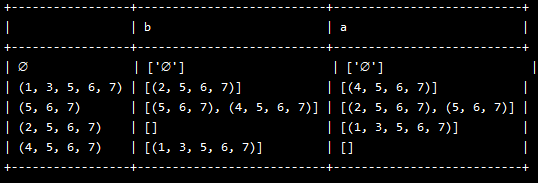


3. Минимизация детерминированного конечного автомата:

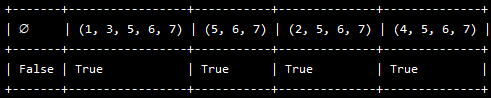
**шаг 0**: ДКА с добавленным состоянием ∅, в которое ведут все переходы по всем символам, которых не было в исходном ДКА (т.к. таких переходов в данном автомате нет – добавленное состояние является недостижимым).



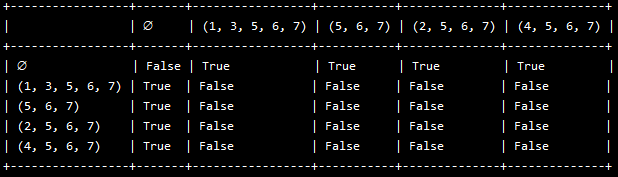
**шаг 1**: таблица обратных ребер



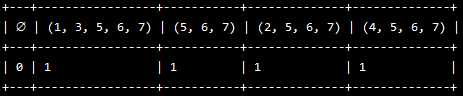
**шаг 2**: массив достижимости состояний из стартового



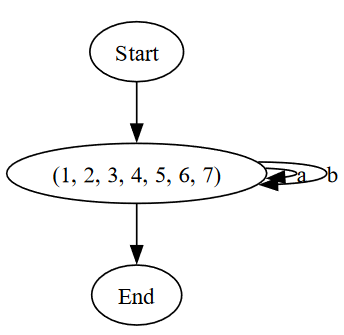
**шаг 3-4**: таблица пар эквивалентных состояний



**шаг 5**: классы эквивалентности



**шаг 6**: минимизированный детерминированный конечный автомат



**Контрольные вопросы**

1. Какие из следующих множеств регулярны? Для тех, которые регулярны, напишите регулярные выражения.
2. Множество цепочек с равным числом нулей и единиц.
3. Множество цепочек из {0, 1}\* с четным числом нулей и нечетным числом единиц.
4. Множество цепочек из {0, 1}\*, длины которых делятся на 3.
5. Множество цепочек из {0, 1}\*, не содержащих подцепочки 101.
6. Найдите праволинейные грамматики для тех множеств из вопроса 1, которые регулярны.
7. Найдите детерминированные и недетерминированные конечные автоматы для тех множеств из вопроса 1, которые регулярны.
8. Найдите конечный автомат с минимальным числом состояний для языка, определяемого автоматом M = ({A, B, C, D, E}, {0, 1}, d, A, {E, F}), где функция задается таблицей

|  |  |  |
| --- | --- | --- |
| Состояние | Вход | |
| 0 | 1 |
| A | B | C |
| B | E | F |
| C | A | A |
| D | F | E |
| E | D | F |
| F | D | E |

Вопросы №1-3:

1) **Множество цепочек с равным числом нулей и единиц.**

Данное множество не является регулярным.

2) **Множество цепочек из {0, 1}\* с четным числом нулей и нечетным числом единиц.**

Праволинейная грамматика:

S -> 0A

S -> 1B

A -> 0S

A -> 1C

B -> 0C

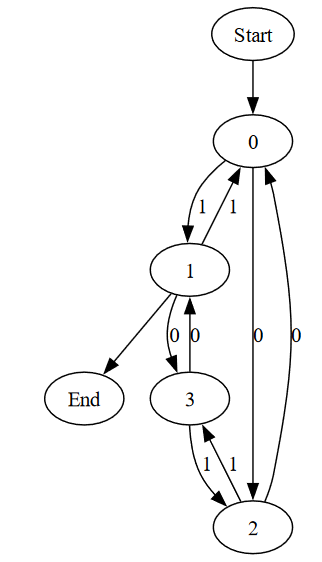
B -> 1S

B -> ε

C -> 1A

C -> 0B

ДКА



3) **Множество цепочек из {0, 1}\*, длины которых делятся на 3.**

регулярное выражение:

((0|1)(0|1)(0|1))\*

Праволинейная грамматика:

S -> 0A

S -> 1A

S -> ε

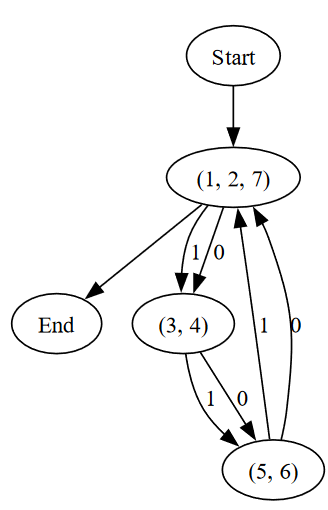
A -> 0B

A -> 1B

B -> 0S

B -> 1S

ДКА:



4) **Множество цепочек из {0, 1}\*, не содержащих подцепочки 101.**

Регулярное выражение: 0\*(1|(00)|(000))\*0\*

Праволинейная грамматика:

S -> 0A

S -> 1A

S -> ε

A -> 0B

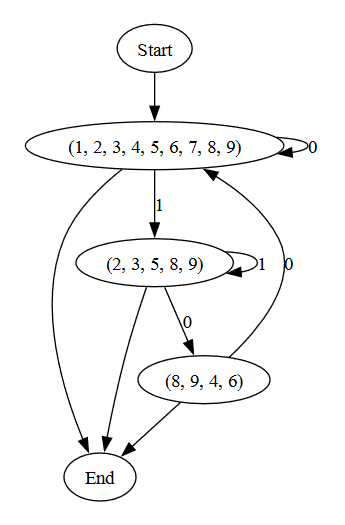
A -> 1A

A -> ε

B -> 0S

B -> ε

ДКА:



**Текст программы**

Модуль main:

**from** builder **import** \*  
  
  
**def** main():  
 regexp = input(**"Введите регулярное выражение: "**)  
 dfa = create\_dfa(regexp)  
 print(**"Детерминированный конечный автомат (см. файл \"ДКА.pdf\")"**)  
 dfa.show\_automaton(**"ДКА"**)  
  
 mdfa = dfa.minimization()  
 print(**"Минимизированный детерминированный конечный автомат (см. файл \"МДКА.pdf\")"**)  
 mdfa.show\_automaton(**"МДКА"**)  
  
 **while** (**True**):  
 check = input(**"Введите строку для моделирования МКА (для выхода введите '\_end\_'): "**)  
 **if** check == **'\_end\_'**:  
 **return  
 else**:  
 **if** mdfa.model\_check(check):  
 print(**"OK"**)  
 **else**:  
 print(**"INVALID STRING"**)  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 main()

Модуль builder

**from** automaton **import** DFA  
**from** typing **import** List, Tuple  
**from** graphviz **import** Digraph  
  
isDebug = **False  
  
  
class** TreeNode:  
 **def** \_\_init\_\_(self, v=**None**, l=**None**, r=**None**):  
 self.left = l  
 self.right = r  
 self.value = v  
 self.followpos = []  
 self.label\_i = **''  
  
 def** copy(cur):  
 **return** TreeNode(cur.value, cur.left, cur.right)  
 left = **None** right = **None** value = **None  
  
 def** show\_tree(self, dot=**None**, show\_params=**False**, show\_end=**False**):  
 **if not** dot:  
 dot = Digraph()  
 label = **"END" if** show\_end **and** self.value == **'#' else** str(self.value)  
 label += (**" "** + str(self.label\_i) **if** self.label\_i **else ""**)  
  
 **def** nodes\_to\_idx(nodes):  
 **return** [node.idx **for** node **in** nodes]  
 **if** show\_params:  
 label = label + **"("** + str(self.nullable) + **", "** \  
 + str(nodes\_to\_idx(self.firstpos)) + **", "**\  
 + str(nodes\_to\_idx(self.lastpos)) + **", "** + str(nodes\_to\_idx(self.followpos)) + **")"** dot.node(str(self.idx), label)  
  
 **if** self.left:  
 dot.edge(str(self.idx), str(self.left.idx))  
 self.left.show\_tree(dot, show\_params=show\_params)  
 **if** self.right:  
 dot.edge(str(self.idx), str(self.right.idx))  
 self.right.show\_tree(dot, show\_params=show\_params)  
  
 **return** dot  
  
 **def** update\_idx(self, i=0):  
 self.idx = i  
 **if** self.left:  
 self.left.update\_idx(i \* 2 + 1)  
 **if** self.right:  
 self.right.update\_idx(i \* 2 + 2)  
  
 **def** update\_nullable(self):  
 **if** self.left:  
 self.left.update\_nullable()  
 **if** self.right:  
 self.right.update\_nullable()  
  
 **if** self.value == **'\*'**:  
 self.nullable = **True  
 elif** self.value == **'|'**:  
 self.nullable = self.left.nullable **or** self.right.nullable  
 **elif** self.value == **''**:  
 self.nullable = self.left.nullable **and** self.right.nullable  
 **else**:  
 self.nullable = **False  
 return** self.nullable  
  
 **def** calc\_firstpos(self):  
 **if** self.left:  
 self.left.calc\_firstpos()  
 **if** self.right:  
 self.right.calc\_firstpos()  
  
 **if** self.value **in** [**'\*'**, **'+'**]:  
 self.firstpos = self.left.firstpos.copy()  
 **elif** self.value == **''**:  
 self.firstpos = self.left.firstpos.copy()  
 **if** self.left.nullable:  
 self.firstpos += self.right.firstpos  
 **elif** self.value == **'|'**:  
 self.firstpos = self.left.firstpos + self.right.firstpos  
 **else**:  
 self.firstpos = [self]  
 **return** self.firstpos  
  
 **def** calc\_lastpos(self):  
 **if** self.left:  
 self.left.calc\_lastpos()  
 **if** self.right:  
 self.right.calc\_lastpos()  
  
 **if** self.value **in** [**'\*'**, **'+'**]:  
 self.lastpos = self.left.lastpos.copy()  
 **elif** self.value == **''**:  
 self.lastpos = self.right.lastpos.copy()  
 **if** self.right.nullable:  
 self.lastpos += self.left.lastpos  
 **elif** self.value == **'|'**:  
 self.lastpos = self.left.lastpos + self.right.lastpos  
 **else**:  
 self.lastpos = [self]  
 **return** self.lastpos  
  
 **def** calc\_followpos(self):  
 **if** self.left:  
 self.left.calc\_followpos()  
 **if** self.right:  
 self.right.calc\_followpos()  
 **if** self.value == **''**:  
 **for** i **in** self.left.lastpos:  
 i.followpos += self.right.firstpos  
 **elif** self.value == **'\*' or** self.value == **'+'**:  
 **for** i **in** self.lastpos:  
 i.followpos += self.firstpos  
  
 **def** dfs\_tree(self, func\_in=**None**, func\_out=**None**):  
 **if** func\_in:  
 func\_in(self)  
 **if** self.left:  
 self.left.dfs\_tree(func\_in, func\_out)  
 **if** self.right:  
 self.right.dfs\_tree(func\_in, func\_out)  
 **if** func\_out:  
 func\_out(self)  
  
  
**def** print\_debug(\*msgs):  
 **if** isDebug:  
 print(**'DBG: EXPR: '**, \*msgs)  
  
  
**def** \_make\_tree(regexp) -> Tuple[TreeNode, List[TreeNode], List[str]]:  
 **def** get\_expr(exp) -> Tuple[TreeNode, int]:  
 **def** cat\_exprs(left, right):  
 **return** TreeNode(**''**, left, right)  
  
 print\_debug(**""**.join(exp))  
 cur\_pos = 0  
 tree = **None  
  
 if** exp[cur\_pos] == **'('**:  
 cur\_pos += 1  
 e, n = get\_expr(exp[cur\_pos:])  
 cur\_pos += n  
 tree = e  
 **while** exp[cur\_pos] != **')'**:  
 e, n = get\_expr(exp[cur\_pos:])  
 cur\_pos += n  
 tree = cat\_exprs(tree, e)  
 **elif** exp[cur\_pos] **not in** [**'+'**, **'\*'**, **'|'**, **')'**]:  
 tree = TreeNode(exp[cur\_pos])  
 **else**:  
 **raise** Exception(**'Waiting for "(", term: '** + **""**.join(exp) + **' '** + str(cur\_pos))  
  
 cur\_pos += 1  
 **if** cur\_pos < len(exp) **and** exp[cur\_pos] **in** [**'+'**, **'\*'**]:  
 tree = TreeNode(exp[cur\_pos], tree)  
 cur\_pos += 1  
  
 **if** cur\_pos < len(exp) **and** exp[cur\_pos] == **'|'**:  
 next\_expr, l = get\_expr(exp[cur\_pos+1:])  
 **return** TreeNode(**'|'**, tree, next\_expr), cur\_pos + 1 + l  
  
 **return** tree, cur\_pos  
 tree, \_ = get\_expr([**'('**] + list(regexp) + [**')'**])  
 tree.update\_idx()  
 tree.update\_nullable()  
 tree.calc\_firstpos()  
 tree.calc\_lastpos()  
 tree.calc\_followpos()  
  
 leafs = []  
  
 **def** add\_leaf(node: TreeNode):  
 **if** node.value **not in** [**'|'**, **'\*'**, **'+'**, **''**]:  
 leafs.append(node)  
  
 tree.dfs\_tree(func\_out=add\_leaf)  
 **for** idx, leaf **in** enumerate(leafs, 1):  
 leaf.label\_i = idx  
 alphabet = list(set([leaf.value **for** leaf **in** leafs]))  
  
 **return** tree, leafs, alphabet  
  
  
**def** build\_dfa(root: TreeNode, alphabet, end\_symb=**'#'**):  
 **def** make\_state(nodes):  
 **return** tuple(sorted([node.label\_i **for** node **in** nodes]))  
  
 **def** is\_end(nodes):  
 **for** node **in** nodes:  
 **if** node.value == end\_symb:  
 **return True  
 return False** Dstates = []  
  
 table = {}  
 queue = [root.firstpos]  
  
 last = []  
 **while** len(queue):  
 state = queue.pop()  
 Dstates += [make\_state(state)]  
  
 **for** c **in** alphabet:  
 U = []  
 **for** p **in** state:  
 **if** p.value == c:  
 U += p.followpos  
 U = list(set(U))  
 **if not** len(U):  
 **continue  
  
 if** make\_state(U) **not in** Dstates:  
 queue.append(U)  
  
 table[make\_state(state), c] = [make\_state(U)]  
  
 print\_debug(make\_state(state), c, [make\_state(U)])  
  
 **if** is\_end(state):  
 last += [make\_state(state)]  
  
 alphabet2 = []  
 **for** c **in** alphabet:  
 **if** c != end\_symb:  
 alphabet2.append(c)  
 **return** DFA(table=table, init\_state=[Dstates[0]], final\_states=last, alphabet=alphabet2)  
  
  
**def** create\_dfa(regexp: str):  
 tree, leafs, alphabet = \_make\_tree(regexp + **'#'**)  
 print(**"Синтаксическое дерево (см. файл \"Синтаксическое дерево.pdf\")"**)  
 tree.show\_tree().render(**"Синтаксическое дерево"**, view=**True**)  
  
 **return** build\_dfa(tree, alphabet)

Модуль automaton

**import** os  
  
os.environ[**"PATH"**] += os.pathsep + **'C:/Program Files/Graphviz/bin/'  
  
from** graphviz **import** Digraph  
**from** prettytable **import** PrettyTable  
  
  
**class** DFA:  
 **def** \_\_init\_\_(self, table, init\_state, final\_states, alphabet: list):  
 self.table = table  
 self.init\_state = list(set(init\_state))  
 self.final\_states = list(set(final\_states))  
  
 self.states = list(set([k[0] **for** k **in** self.table.keys()]).union(set(self.final\_states))) *# self.\_init\_incidence\_list()* self.alphabet = {}  
 **for** i **in** range(len(alphabet)):  
 self.alphabet[alphabet[i]] = i  
  
 @staticmethod  
 **def** \_get\_incidence\_list(states: list, table: dict) -> list:  
 incidence\_list = [[] **for** \_ **in** states]  
 **for** k, v **in** table.items():  
 **for** val **in** v:  
 incidence\_list[states.index(k[0])].append(states.index(val))  
 **return** incidence\_list  
  
 **def** show\_automaton(self, title: str, show\_end=**True**, show\_isolated\_empty=**True**):  
 dot = Digraph()  
  
 dot.node(**'S'**, **'Start'**)  
 **if** show\_end:  
 dot.node(**'E'**, **'End'**)  
  
 **for** i **in** self.init\_state:  
 dot.edge(**'S'**, str(i))  
  
 **for** st **in** self.final\_states:  
 **if** show\_end:  
 dot.edge(str(st), **'E'**)  
 **else**:  
 dot.node(str(st), shape=**'cds'**)  
  
 is\_empty\_isolated = **True  
 for** st\_from, v **in** self.table:  
 **for** st\_to **in** self.table[(st\_from, v)]:  
 **if** (str(st\_from) == **'∅' or** str(st\_to) == **'∅'**) **and** st\_from != st\_to:  
 is\_empty\_isolated = **False  
  
 for** st\_from, v **in** self.table:  
 **for** st\_to **in** self.table[(st\_from, v)]:  
 st\_from = str(st\_from)  
 st\_to = str(st\_to)  
 **if not** show\_isolated\_empty **and** is\_empty\_isolated **and** st\_from == **'∅'**:  
 **continue  
 if** st\_from == **'∅'**:  
 st\_from = **""  
 if** st\_to == **'∅'**:  
 st\_to = **""** dot.node(st\_from)  
 dot.node(st\_to)  
 dot.edge(st\_from, st\_to, v)  
  
 dot.render(title, view=**True**)  
  
 @staticmethod  
 **def** \_get\_inverse\_states(alphabet: dict, states: list, table: dict):  
 inverse\_states = [[[] **for** \_ **in** alphabet] **for** \_ **in** states]  
 **for** st\_from, c **in** table.keys():  
 **for** st\_to **in** table[(st\_from, c)]:  
 inverse\_states[states.index(st\_to)][alphabet[c]].append(states.index(st\_from))  
  
 content = [[[states[state] **for** state **in** st\_list] **for** st\_list **in** row] **for** row **in** inverse\_states]  
 DFA.\_print\_table(title=**"таблица обратных ребер (шаг 1)"**, col\_names=list(alphabet.keys()), content=content,  
 row\_names=states)  
 **return** inverse\_states  
  
 @staticmethod  
 **def** \_buildMarkedTable(alphabet: dict, states: list, final\_states: list, inverse\_edges):  
 marked = [[**False for** \_ **in** range(len(states))] **for** \_ **in** range(len(states))]  
 q = []  
  
 **for** i **in** range(len(states)):  
 **for** j **in** range(len(states)):  
 **if** marked[i][j] **is False and** ((states[i] **in** final\_states) != (states[j] **in** final\_states)):  
 marked[i][j] = marked[j][i] = **True** q.append((i, j))  
 **"""DFA.\_print\_table(title="Пары вершин, соединяющие терминальные и нетерминальные состояния (шаг 3)", col\_names=states,  
 content=marked, row\_names=states)"""  
  
 while** len(q) > 0:  
 i, j = q[0]  
 q.remove((i, j))  
 **for** c **in** alphabet.values():  
 **for** r **in** inverse\_edges[i][c]:  
 **for** s **in** inverse\_edges[j][c]:  
 **if not** marked[r][s]:  
 marked[r][s] = marked[s][r] = **True** q.append((r, s))  
 DFA.\_print\_table(title=**"Таблица пар эквивалентных состояний (шаги 3 и 4)"**, col\_names=states, content=marked, row\_names=states)  
 **return** marked  
  
 @staticmethod  
 **def** \_dfs(states: list, table: dict, init\_state: list) -> list:  
 incidence\_list = DFA.\_get\_incidence\_list(states=states, table=table)  
 visited = [**False for** \_ **in** states]  
 q = [states.index(init\_state[0])]  
 **while** len(q) > 0:  
 curr = q[-1]  
 q.pop(len(q) - 1)  
 visited[curr] = **True  
 for** to **in** incidence\_list[curr]:  
 **if not** visited[to]:  
 q.append(to)  
  
 DFA.\_print\_table(title=**"массив достижимости состояний из стартового (шаг 2)"**, content=[visited], col\_names=states)  
 **return** visited  
  
 **def** minimization(self):  
 prepared\_states = [**'∅'**] + self.states.copy()  
 idx = prepared\_states.index(self.init\_state[0])  
 **if** idx != 1:  
 prepared\_states[1], prepared\_states[idx] = prepared\_states[idx], prepared\_states[1]  
  
 prepared\_states\_table = self.table.copy()  
 **for** c **in** self.alphabet.keys():  
 **for** st **in** prepared\_states:  
 **if** (st, c) **not in** prepared\_states\_table:  
 prepared\_states\_table[(st, c)] = [prepared\_states[0]]  
  
 prepared\_dfa = DFA(table=prepared\_states\_table, init\_state=self.init\_state.copy(),  
 final\_states=self.final\_states.copy(), alphabet=list(self.alphabet.keys()))  
 print(**"ДКА с добавленным состоянием ∅, в которое ведут все переходы по всем символам, которых не было в исходном ДКА (см. файл \"Prepared DFA.pdf\")"**)  
 prepared\_dfa.show\_automaton(**"Prepared DFA"**)  
  
 inverse\_edges = DFA.\_get\_inverse\_states(alphabet=self.alphabet, states=prepared\_states, table=prepared\_states\_table)  
 reachable = self.\_dfs(states=prepared\_states, table=prepared\_states\_table, init\_state=self.init\_state)  
 marked = self.\_buildMarkedTable(alphabet=self.alphabet, states=prepared\_states,  
 final\_states=self.final\_states, inverse\_edges=inverse\_edges)  
  
 component = [-1 **for** \_ **in** prepared\_states]  
 **for** i **in** range(len(prepared\_states)):  
 **if not** marked[0][i]:  
 component[i] = 0  
  
 components\_count = 0  
 **for** i **in** range(1, len(prepared\_states)):  
 **if not** reachable[i]:  
 **continue  
 if** component[i] == -1:  
 components\_count += 1  
 component[i] = components\_count  
 **for** j **in** range(i+1, len(prepared\_states)):  
 **if not** marked[i][j]:  
 component[j] = components\_count  
  
 self.\_print\_table(title=**"Классы эквивалентности (шаг 5)"**, content=[component], col\_names=prepared\_states)  
  
 new\_table = {}  
 new\_states = {}  
 **for** comp **in** range(1, len(component)):  
 new\_states[component[comp]] = set()  
 **for** idx\_comp **in** range(1, len(component)):  
 new\_states[component[idx\_comp]].add(prepared\_states[idx\_comp])  
 **for** k, v **in** new\_states.items():  
 new\_states[k] = self.\_concat\_states(v)  
  
 **for** k, v **in** self.table.items():  
 new\_from = tuple(new\_states[component[prepared\_states.index(k[0])]])  
 new\_to = tuple(new\_states[component[prepared\_states.index(v[0])]])  
 new\_table[(new\_from, k[1])] = [new\_to]  
  
 new\_init\_state = []  
 new\_final\_states = []  
 **for** st **in** self.init\_state:  
 new\_init\_state.append(tuple(new\_states[component[prepared\_states.index(st)]]))  
 **for** st **in** self.final\_states:  
 new\_final\_states.append(tuple(new\_states[component[prepared\_states.index(st)]]))  
  
 new\_init\_state = list(set(new\_init\_state))  
 new\_final\_states = list(set(new\_final\_states))  
 **return** DFA(table=new\_table, init\_state=new\_init\_state, final\_states=new\_final\_states, alphabet=list(self.alphabet.keys()))  
  
 **def** model\_check(self, check\_str):  
 curr\_st = self.init\_state[0]  
 **for** c **in** check\_str:  
 print(curr\_st, c)  
 **if** (curr\_st, c) **not in** self.table:  
 **return False** curr\_st = self.table[(curr\_st, c)][0]  
  
 **return** curr\_st **in** self.final\_states  
  
 @staticmethod  
 **def** \_concat\_states(states\_list: set) -> set:  
 res = set()  
 **for** st **in** states\_list:  
 res = res.union(set(st))  
 **return** res  
  
 @staticmethod  
 **def** \_print\_table(title: str, content: list, col\_names: list, row\_names: list = **None**):  
 **if** row\_names **is not None**:  
 col\_names = [**""**] + col\_names  
 content = [[row\_names[i]] + content[i] **for** i **in** range(len(row\_names))]  
 table = PrettyTable(align=**'l'**)  
 table.title = title  
 table.field\_names = col\_names  
 table.add\_rows(content)  
  
 print(table)  
 print()