МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

"МИРЭА - Российский технологический университет"

РТУ МИРЭА

Институт информационных технологий (ИТ)

Кафедра Вычислительной техники (ВТ)

ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ №4

по дисциплине

«Теория формальных языков»

Тема. Преобразование недетерминированного конечного автомата в детерминированный.

Выполнил студент группы ИКБО-04-22 Основин А.И.

Принял преподаватель Боронников А.С.

Москва 2023

СОДЕРЖАНИЕ

1 ПОСТАНОВКА ЗАДАЧИ

1.1 Условия задачи

На любом языке программирования написать программу преобразования недетерминированного конечного автомата (НКА) в детерминированный (ДКА).

1.2 Пример преобразования НКА в ДКА

На Рисунке 1 представлен пример преобразования недетерминированного конечного автомата в детерминированный.

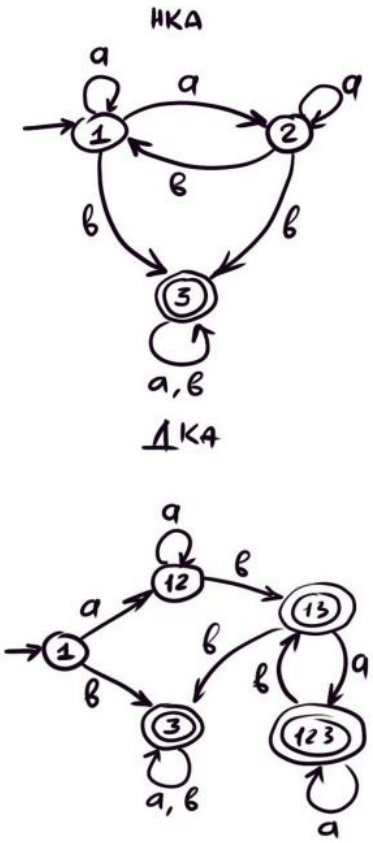

Рисунок 1 – Преобразование НКА в ДКА

1.3 Язык программирования

Для реализации поставленных задач был выбран язык программирования Python.

2 РЕАЛИЗАЦИЯ ПРОГРАММЫ

В Листинге 1 представлена реализация программы преобразования недетерминированного конечного автомата в детерминированный.

Листинг 1 – Код программы на языке программирования Python

```
import os
import graphviz


def transform(states: tuple, alphabet: tuple, in_states: tuple, out_states:
tuple, state_transitions: tuple) \
        -> dict[str, tuple]:
    """ Converts a non-deterministic finite automaton into a deterministic
finite automaton. """

    def get_next_state_transitions(current_state: tuple) -> set[tuple]:
        """ Searches for the following possible states from the current one.
"""

        next_state_transitions = set()
        for symbol in alphabet:
            by_symbol = set()
            for element in current_state:
                for (from_state, by, to_state) in state_transitions:
                    if from_state == element and by == symbol:
                        by_symbol.add(to_state)

            next_state_transitions.add(tuple([symbol, tuple(by_symbol)]))
        return next_state_transitions

    new_state_transitions = set()
    new_out_states = set()
    executing_states = list(tuple([in_states]))
    for state in executing_states:
        new_state_transition = get_next_state_transitions(state)
        for st in new_state_transition:
            new_state_transitions.add(tuple([''.join(state), st[0],
''.join(sorted(st[1]))]))
            if tuple(sorted(st[1])) not in executing_states:
                executing_states.append(tuple(sorted(st[1])))
            for out_state in out_states:
                if out_state in st[1]:
                    new_out_states.add(''.join(sorted(st[1])))

    return {
        "states": tuple(''.join(state) for state in executing_states),
        "alphabet": alphabet,
        "in_states": in_states,
        "out_states": new_out_states,
        "state_transitions": new_state_transitions
    }


def input_finite_state_machine() -> list:
    """ Obtains the grammar of a non-deterministic finite automaton. """

    states = tuple(input('Enter set of states: ').split(' '))
    alphabet = tuple(input('Enter the input alphabet: ').split(' '))
```

```
    print('Enter state-transitions function(current state, input character,
next state).'
          'Every three is on a new line. Enter "\\" to complete entry.')

    state_transitions = set()
    state_transition = input()
    while state_transition != '\\':
        state_transitions.add(tuple(state_transition.split(' ')[:3:]))
        state_transition = input()

    state_transitions = tuple(state_transitions)
    in_states = tuple(input('Enter a set of initial states: ').split(' '))
    out_states = tuple(input('Enter a set of final states: ').split(' '))

    return [states, alphabet, in_states, out_states, state_transitions]


def print_finite_state_machine(new_state_machine: dict):
    """ Outputs the grammar of a deterministic finite automaton. """

    print("\nSet of states: ", end='')
    print(*new_state_machine["states"], sep=", ")

    print("\nInput alphabet: ", end='')
    print(*new_state_machine["alphabet"], sep=", ")

    print("\nState-transitions function:")
    print(*[f'D({state_transition[0]}, {state_transition[1]}) =
{state_transition[2]}'
            for state_transition in new_state_machine["state_transitions"]],
sep="\n")

    print("\nInitial states: ", end='')
    print(*new_state_machine["in_states"], sep=", ")

    print("\nFinal states: ", end='')
    print(*new_state_machine["out_states"], sep=", ")


def make_graph(new_state_machine_transitions: tuple[tuple[str, str, str],
...],
               in_states: tuple, out_states: tuple, name: str):
    """ Generates a graph in graphviz language based on the passed
dictionary. """

    graph = graphviz.Digraph(name=name)
    handled_init_states = set()
    for (from_state, by, to_state) in new_state_machine_transitions:
        if from_state in in_states and from_state not in handled_init_states:
            graph.node(' ', shape='plaintext')
            graph.edge(' ', from_state)
            handled_init_states.add(from_state)

        if to_state in out_states:
            graph.node(to_state, shape='doublecircle')

        graph.edge(from_state, to_state, label=by)

    graph.render(f'files/dependency_graphs/{name}.gv', view=True)


def main():
```

```
    # states, alphabet, in_states, out_states, state_transitions =
input_finite_state_machine()

    # test example
    states = ('1', '2', '3')
    alphabet = ('a', 'b')
    in_states = ('1',)
    out_states = ('3',)
    state_transitions = (('1', 'a', '1'), ('1', 'a', '2'), ('1', 'b', '3'),
('2', 'a', '2'),
                         ('2', 'b', '1'), ('2', 'b', '3'), ('3', 'a', '3'),
('3', 'b', '3'))

    new_state_machine = transform(states, alphabet, in_states, out_states,
state_transitions)
    print_finite_state_machine(new_state_machine)

    os.environ["PATH"] += os.pathsep + 'C:/Program Files/Graphviz/bin/'
    make_graph(state_transitions, in_states, out_states, "non-deterministic
finite state machine")
    make_graph(tuple(new_state_machine["state_transitions"]),
               new_state_machine["in_states"],
               new_state_machine["out_states"],
               "deterministic finite state machine")


if __name__ == '__main__':
    main()
```

3 РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ

В Таблице 1 представлены результаты тестирования программы, преобразующей НКА в ДКА.

Таблица 1 – Тестирование программы, преобразующей НКА в ДКА

	Ввод (характеристики НКА)	Вывод программы (характеристики ДКА)	Ожидаемый вывод
Множество состояний	1, 2, 3	1, 3, 12, 13, 123	1, 3, 12, 13, 123
Алфавит	a, b	a, b	a, b
Правила переходов	(1, a, 1), (1, a, 2), (1, b, 3), (2, a, 2), (2, b, 1), (2, b, 3), (3, a, 3), (3, b, 3)	D(12, b) = 13 D(3, a) = 3 D(13, a) = 123 D(123, b) = 13 D(3, b) = 3 D(13, b) = 3 D(123, a) = 123 D(1, b) = 3 D(1, a) = 12 D(12, a) = 12	D(12, b) = 13 D(3, a) = 3 D(13, a) = 123 D(123, b) = 13 D(3, b) = 3 D(13, b) = 3 D(123, a) = 123 D(1, b) = 3 D(1, a) = 12 D(12, a) = 12
Входные состояния	1	1	1
Выходные состояния	3	123, 13, 3	123, 13, 3

Для удобства тестирования разработанной программы и более лёгкого восприятия результата работы программы была добавлена функции создания графа на основе НКА и ДКА. На Рисунках 2 и 3 представлен граф данного НКА и граф построенного по нему ДКА соответственно.

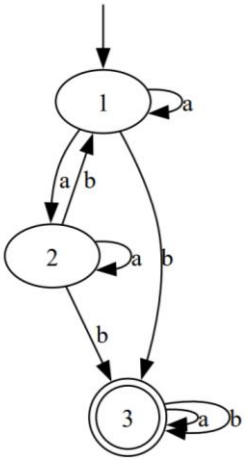

Рисунок 2 – Данный для преобразования НКА

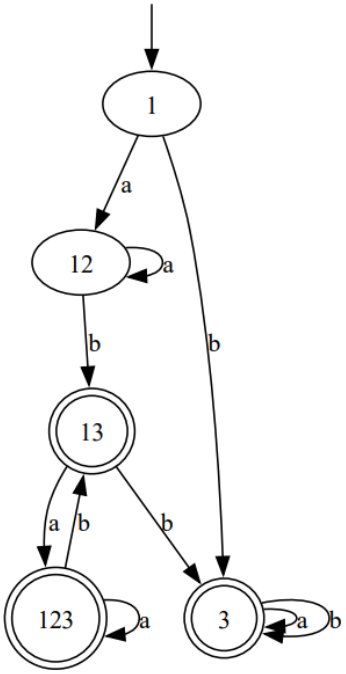

Рисунок 3 – ДКА, построенный по входному НКА

4 ВЫВОД

В ходе выполнения данной практической работы был изучен процесс преобразования недетерминированного конечного автомата (НКА) в детерминированный конечный автомат (ДКА); была разработана программа, реализующая преобразование НКА в ДКА. Программа успешного прошла тестирование, следовательно, реализация корректна.