

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 3](#_Toc153126584)

[1.1 Условия задачи 3](#_Toc153126585)

[1.2 Пример преобразования НКА в ДКА 3](#_Toc153126586)

[1.3 Язык программирования 3](#_Toc153126587)

[2 РЕАЛИЗАЦИЯ ПРОГРАММЫ 4](#_Toc153126588)

[3 РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ 7](#_Toc153126589)

[4 ВЫВОД 9](#_Toc153126590)

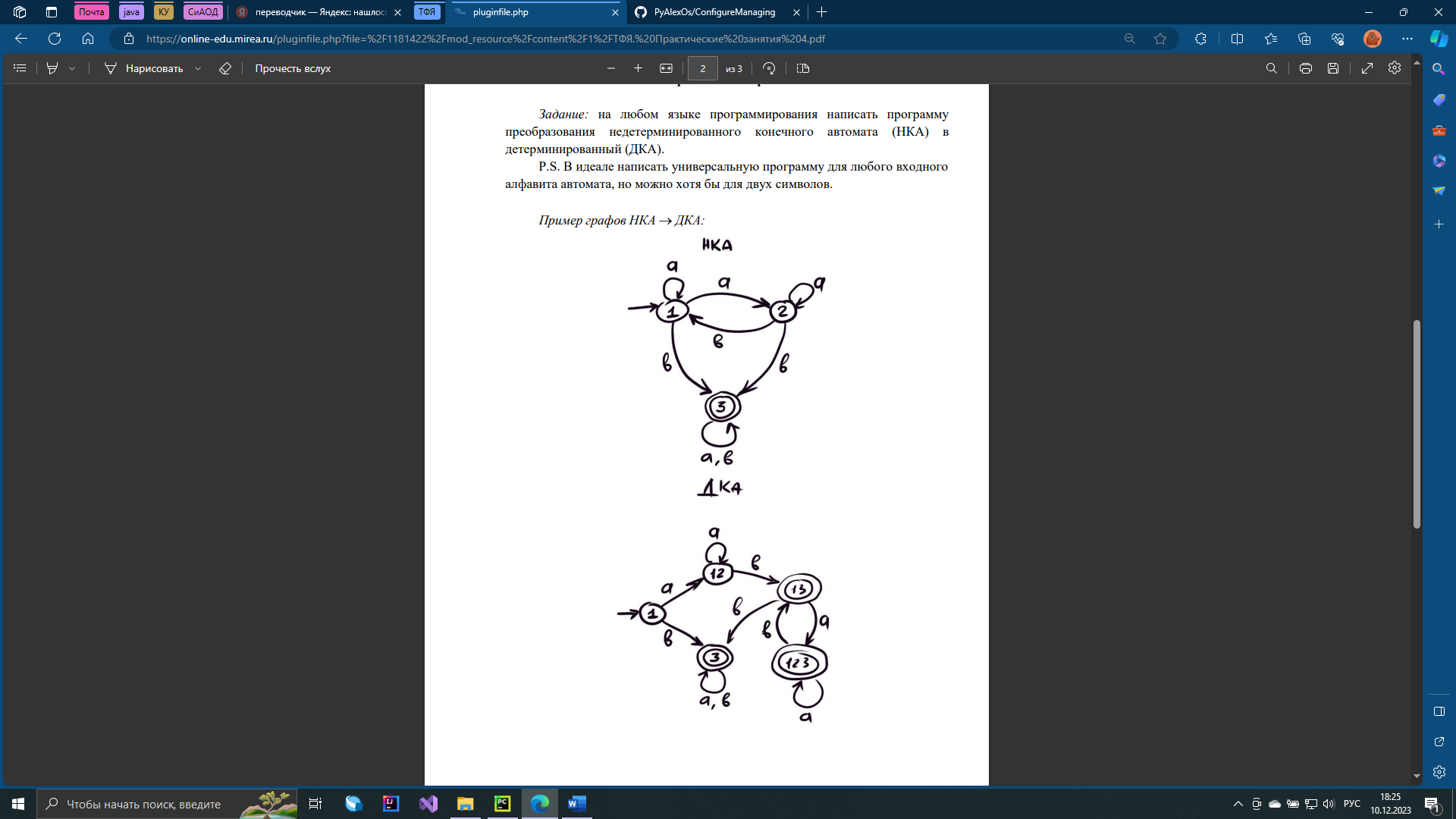
# ПОСТАНОВКА ЗАДАЧИ

## Условия задачи

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

## Пример преобразования НКА в ДКА

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



**Рисунок 1 – Преобразование НКА в ДКА**

## Язык программирования

Для реализации поставленных задач был выбран язык программирования Python.

# 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()*

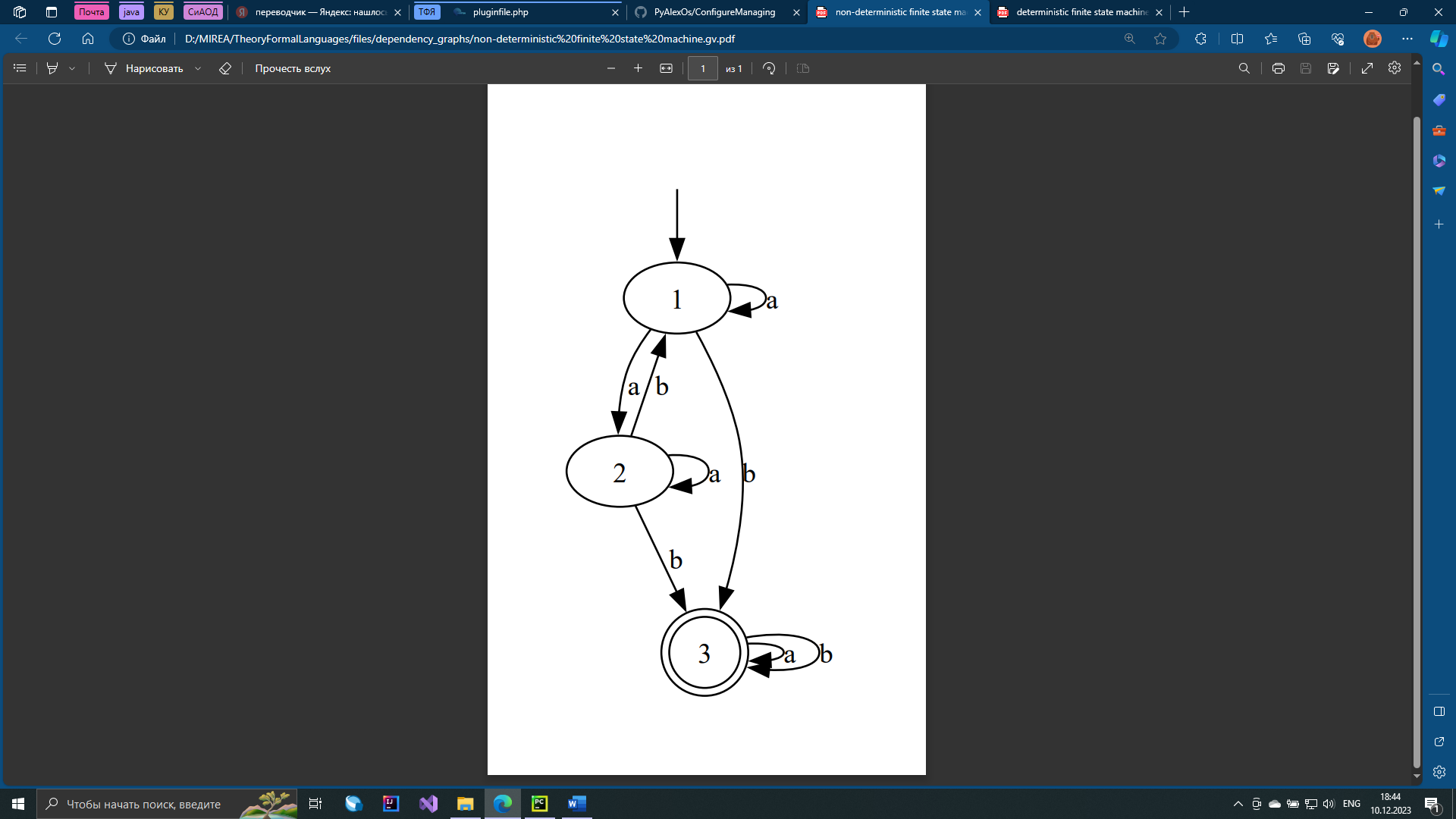
# РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ

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

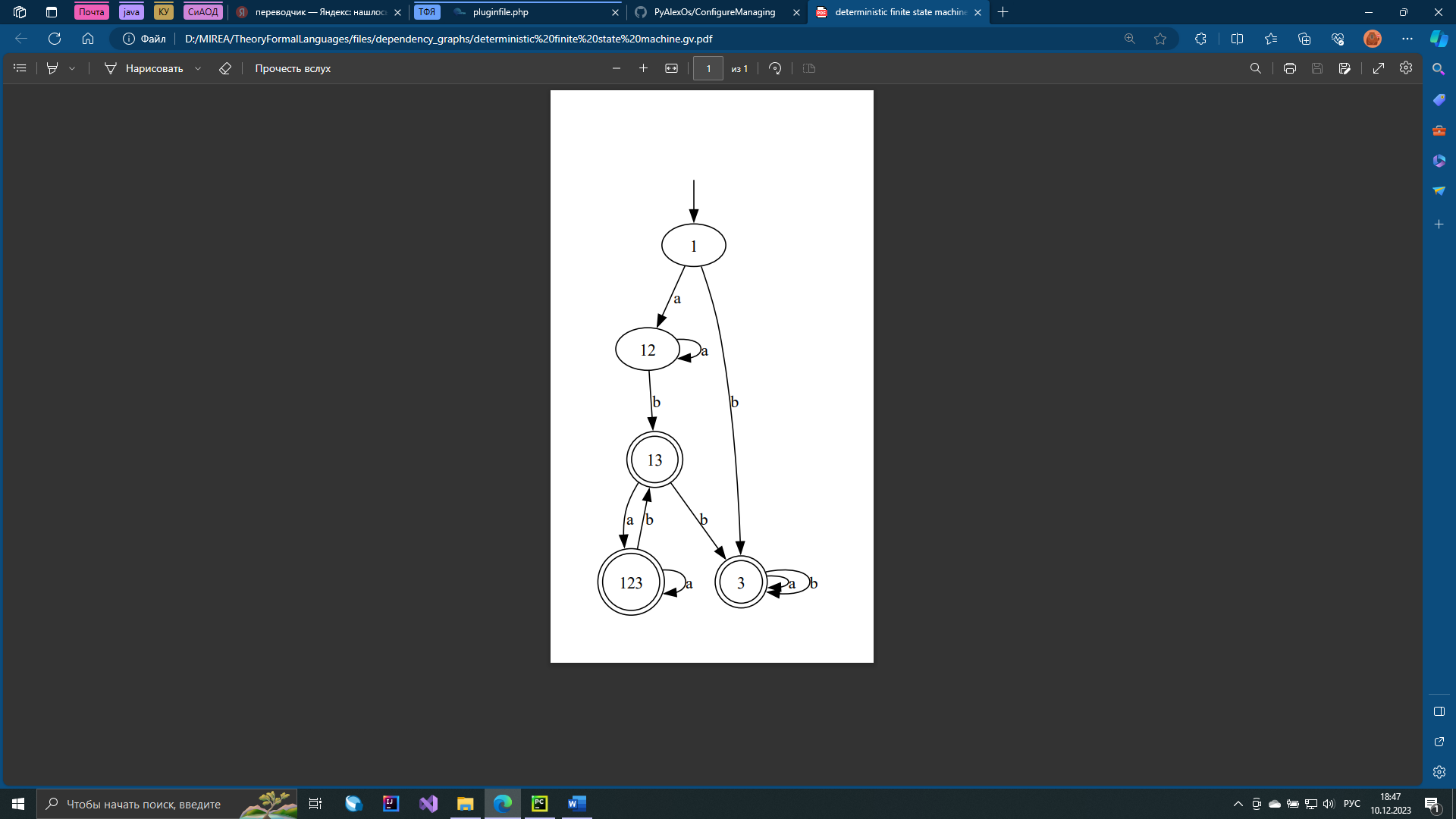
*Таблица 1 – Тестирование программы, преобразующей НКА в ДКА*

|  |  |  |  |
| --- | --- | --- | --- |
|  | 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**

# 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.