Ministerul Educaţiei, Culturii și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Departamentul Ingineria Software și Automatică

REPORT

Laboratory work 2
Course: Formal Languages & Finite Automata
Theme: Intro to formal languages. Regular grammars. Finite Automata

Elaborated: Latcovschi Cătălin, FAF-221

Verified: asist.univ. Dumitru Crețu

Chișinău 2024

Overview

A finite automaton is a mechanism used to represent processes of different kinds. It can be compared to a state machine as they both have similar structures and purpose as well. The word finite signifies the fact that an automaton comes with a starting and a set of final states. In other words, for process modeled by an automaton has a beginning and an ending.

Based on the structure of an automaton, there are cases in which with one transition multiple states can be reached which causes non determinism to appear. In general, when talking about systems theory the word determinism characterizes how predictable a system is. If there are random variables involved, the system becomes stochastic or non deterministic.

That being said, the automata can be classified as non-/deterministic, and there is in fact a possibility to reach determinism by following algorithms which modify the structure of the automaton.

Objectives

Understand what an automaton is and what it can be used for.

Continuing the work in the same repository and the same project, the following need to be added: a. Provide a function in your grammar type/class that could classify the grammar based on Chomsky hierarchy.

b. For this you can use the variant from the previous lab.

According to your variant number (by universal convention it is register ID), get the finite automaton definition and do the following tasks:

a. Implement conversion of a finite automaton to a regular grammar.

b. Determine whether your FA is deterministic or non-deterministic.

c. Implement some functionality that would convert an NDFA to a DFA.

d. Represent the finite automaton graphically (Optional, and can be considered as a bonus point):

You can use external libraries, tools or APIs to generate the figures/diagrams.

Your program needs to gather and send the data about the automaton and the lib/tool/API return the visual representation.

Please consider that all elements of the task 3 can be done manually, writing a detailed report about how you've done the conversion and what changes have you introduced. In case if you'll be able to write a complete program that will take some finite automata and then convert it to the regular grammar - this will be a good bonus point.

Implementation

GrammarConverter

Attributes: non_terminals, terminals, productions.

Function: Serves as a container to store the components of a grammar that can be derived from an automaton (states, alphabet, and transitions of the automaton). However, in the provided code, this class is not fully implemented and only used to store information.

Automaton:

states: The states of the automaton.

alphabet: The set of symbols the automaton operates on.

initial_state: The starting state of the automaton.

final_states: The accepting (or final) states of the automaton.

transitions: The state transition rules, stored as a dictionary.

Methods:

__init__: Constructor that initializes a sample NFA.

is_det: Checks whether the current automaton is deterministic.

to_grammar: Converts the automaton to a grammar format.

nfa_to_dfa: Transforms the NFA into a DFA.

group_states: Groups states in the NFA transition to create compound states for the DFA.

degroup_states: Splits compound states back into individual states.

has_all_states: Checks if all states are included in the transition rules.

draw_automaton: Uses NetworkX and Matplotlib to visualize the automaton.

Conversion from NFA to DFA:

This process is handled by the nfa_to_dfa method and involves the following steps:

Check if the automaton is already deterministic; if so, no conversion is needed.

Initialize a new Automaton instance for the DFA.

Use regular expressions to parse transitions and states.

Create a new set of transitions for the DFA by combining states in the NFA that are reachable through the same input symbol into new compound states.

Update the DFA's states, final states, and transitions based on the newly created compound states.

The conversion continues until all states and transitions are processed and included in the DFA.

Visualization

The draw_automaton method uses NetworkX and Matplotlib to create and display a graphical representation of the automaton. This involves creating a directed graph, adding nodes and edges based on the automaton's transitions, and using a planar layout to position the nodes and labels.

```
class Automaton:


   states = []

   alphabet = []
```

```
initial_state = []

final_states = []

transitions = {}


def __init__(self):

    self.states = ["q0", "q1", "q2", "q3"]

    self.alphabet = ["a", "b", "c"]

    self.initial_state = "q0"

    self.final_states = "q3"

    self.transitions = {

        "q0": ["a q0", "a q1"],

        "q1": ["b q1", "a q2"],

        "q2": ["b q3", "a q0"],

        "q3": []

    }


def is_det(self) -> bool:

    for state, transitions in self.transitions.items():

        symbols = []


        for transition in transitions:

            symbols.append(transition[0])

        if len(symbols) != len(set(symbols)):

            return False


    return True


def to_grammar(self) -> GrammarConverter:

    grammar = GrammarConverter()
```

```
        grammar.non_terminals = self.states
        grammar.terminals = self.alphabet
        grammar.productions = self.transitions

        return grammar

    def nfa_to_dfa(self):
        if self.is_det():
            return
        dfa = Automaton()

        chars_pattern = re.compile(r'^(.*?)\s')
        states_pattern = re.compile(r'\s(.*)$')

        transitions = {self.initial_state:
self.group_states(self.transitions[self.initial_state])}

        finished_states = []
        last_state = ""

        while not (self.has_all_states(transitions)):

            transitions_copy = copy.deepcopy(transitions)
            for value in finished_states:
                del transitions_copy[value]

            for key, values in transitions_copy.items():
                symbols = []
                states = []
```

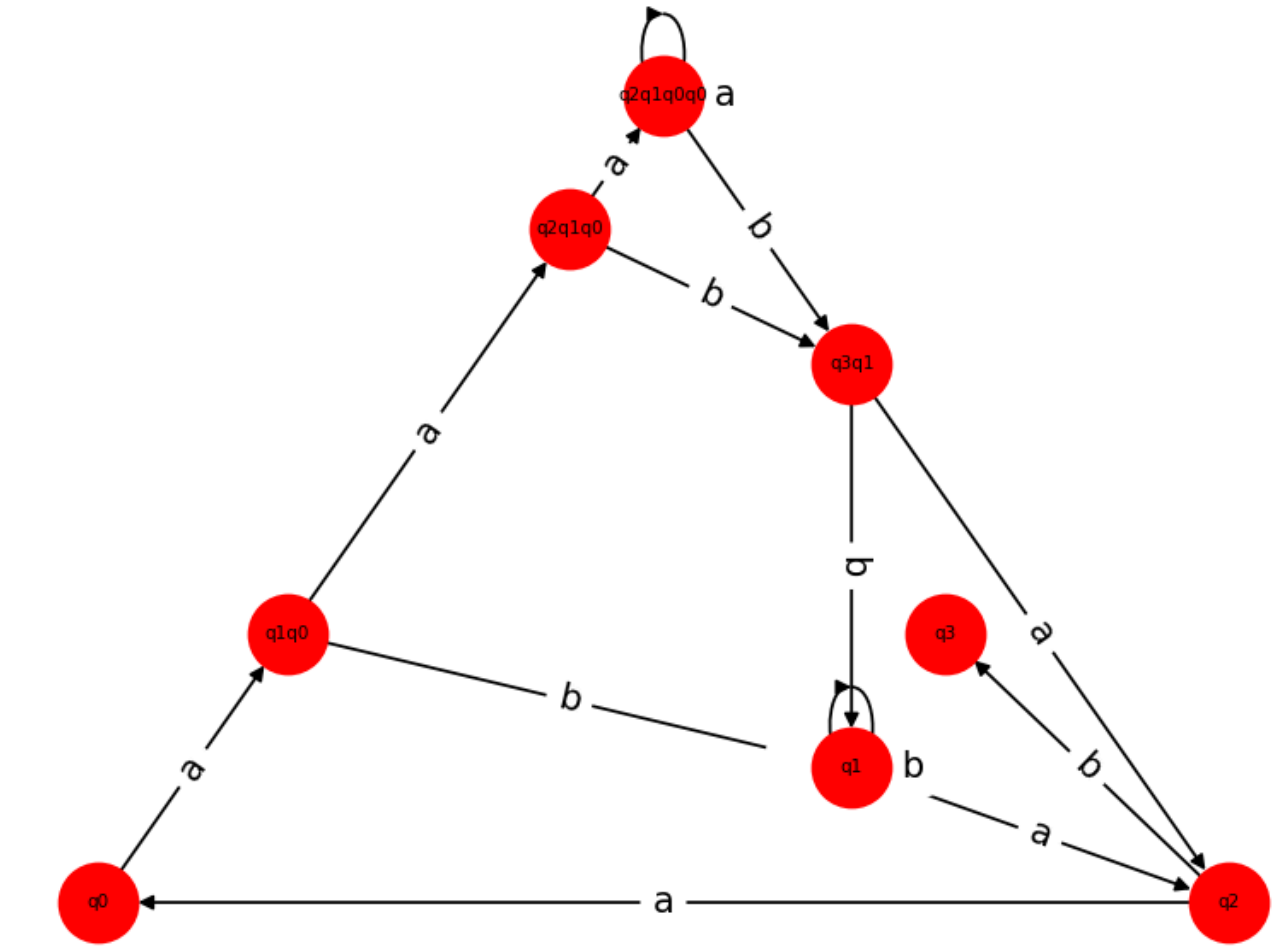
```
                for value in values:
                    symbols.append(chars_pattern.search(value).group(1))
                    states.append(states_pattern.search(value).group(1))

                if not (len(symbols) == len(set(symbols))):
                    unique_symbols = set(symbols)
                    for element in unique_symbols:
                        positions = [i for i, value in enumerate(symbols) if value
== element]
                        if len(positions) > 1:
                            new_state = ""
                            new_state_trans = []
                            for position in positions:
                                if states[position] not in new_state:
                                    new_state += states[position]
                                for state1 in
self.degroup_states(states[position]):
                                    for elem in self.transitions[state1]:
                                        if elem not in new_state_trans:
                                            new_state_trans.append(elem)

                            if new_state in transitions:
                                break

                            transitions[new_state] =
self.group_states(new_state_trans)
                            last_state = new_state

                for state in states:
                    if state not in transitions:
                        new_state_trans = []
```

```
                    new_states = self.degroup_states(state)
                    for new_state in new_states:
                        for trans in self.transitions[new_state]:
                            if trans not in new_state_trans:
                                new_state_trans.append(trans)
                    new_state_trans = self.group_states(new_state_trans)
                    transitions[state] = new_state_trans
                    last_state = state

            finished_states.append(key)

    finished_states.append(last_state)
    dfa.states = finished_states
    dfa.final_states = []
    for state in dfa.states:
        if self.final_states in state:
            dfa.final_states.append(state)
    dfa.transitions = transitions

    return dfa

def group_states(self, states_arr) -> []:
    chars_pattern = re.compile(r'^(.*?)\s')
    states_pattern = re.compile(r'\s(.*)$')
    states_arr.sort()
    for i in range(len(states_arr) - 1, 0, -1):
        if chars_pattern.search(states_arr[i]).group(
                1) == chars_pattern.search(states_arr[i - 1]).group(1):
            temp = states_pattern.search(states_arr[i]).group(1) + \
```

```
                    states_pattern.search(states_arr[i - 1]).group(1)
            states_arr[i - 1] = chars_pattern.search(states_arr[i - 1]) \
                                                .group(1) + " " + temp
            states_arr.pop(i)

    return states_arr

def degroup_states(self, states) -> []:
    degrouped_states = []
    current_state = ""

    for char in states:
        current_state += char
        if current_state in self.states:
            degrouped_states.append(current_state)
            current_state = ""

    return degrouped_states

def has_all_states(self, dictionary):
    states_pattern = re.compile(r'\s(.*)$')

    for key, values in dictionary.items():
        for value in values:
            state = states_pattern.search(value).group(1)

            if state not in dictionary:
                return False
    return True
```

```
   def draw_automaton(self):
       graph = nx.DiGraph()

       for node, edges in self.transitions.items():
           for edge in edges:
               label, target = edge.split(' ', 1)
               graph.add_edge(node, target, label=label)
               if node == target:
                   label = "           " + label
                   graph.add_edge(node, target, label=label)

       pos = nx.planar_layout(graph)
       labels = nx.get_edge_attributes(graph, 'label')
       nx.draw(graph, pos, with_labels=True, node_size=700, node_color="red",
font_size=6)
       nx.draw_networkx_edge_labels(graph, pos, edge_labels=labels,
font_color='black', font_size=12)
       plt.show()



automaton = Automaton()
dfa = automaton.nfa_to_dfa()
print(dfa.transitions)
print(dfa.final_states)
print(dfa.states)
print(dfa.is_det())
dfa.draw_automaton()
```

Results




```
C:\Users\Catlin\AppData\Local\Programs\Python\Python39\python.exe C:\Users\Catlin\Desktop\FLFA\lab2\lab2.py
{'q0': ['a q1q0'], 'q1q0': ['a q2q1q0', 'b q1'], 'q2q1q0': ['a q2q1q0q0', 'b q3q1'], 'q1': ['a q2', 'b q1'], 'q2q1q0q0': ['a q2q1q0q0', 'b q3q1'], 'q3q1': ['a q2', 'b q1'], 'q2
['q3q1', 'q3']
['q0', 'q1q0', 'q2q1q0', 'q1', 'q2q1q0q0', 'q3q1', 'q2', 'q3']
True
```

{'q0': ['a q1q0'], 'q1q0': ['a q2q1q0', 'b q1'], 'q2q1q0': ['a q2q1q0q0', 'b q3q1'], 'q1': ['a q2', 'b q1'], 'q2q1q0q0': ['a q2q1q0q0', 'b q3q1'], 'q3q1': ['a q2', 'b q1'], 'q2': ['a q0', 'b q3'], 'q3': []}

['q3q1', 'q3']

['q0', 'q1q0', 'q2q1q0', 'q1', 'q2q1q0q0', 'q3q1', 'q2', 'q3']

True

Conclusions

In this laboratory work, we embarked on a detailed exploration of finite automata, delving into their structures and operations. By implementing practical exercises, we successfully translated theoretical concepts into tangible outcomes, specifically focusing on the transformation of non-deterministic finite automata (NFAs) into deterministic finite automata (DFAs) and the conversion of automata into regular grammars.

Understanding Automata: We enhanced our comprehension of what an automaton is and its practical applications. The hands-on approach allowed us to directly engage with the mechanisms that drive both deterministic and non-deterministic finite automata.

Implementation of Theoretical Concepts: The laboratory tasks enabled us to apply theoretical knowledge in a practical setting, solidifying our understanding. We managed to implement a function that can convert a given finite automaton into a regular grammar, providing a direct link between different formal language representations.

Conversion Processes: One of the core achievements of this lab was the successful implementation of the conversion from an NFA to a DFA. This process highlighted the intrinsic differences between deterministic and non-deterministic automata and underscored the importance of determinism in simplifying the state management and transition processes of automata.

Programming and Debugging Skills: Throughout the lab, we honed our programming and debugging skills, particularly in manipulating data structures and implementing algorithms in the context of formal languages and automata theory. This not only reinforced our understanding of the subject matter but also improved our coding proficiency.

Visualization and Interpretation: By using external libraries such as NetworkX and Matplotlib, we were able to graphically represent finite automata, which significantly aided in our understanding and analysis. The visual representations provided a clear and immediate understanding of the automata's structure and operation, which is invaluable for both debugging and educational purposes.

Challenges and Solutions: The lab work presented us with several challenges, particularly in the conversion of NFAs to DFAs and the interpretation of complex automata structures. By methodically addressing these challenges, we developed a deeper understanding of the subject matter and enhanced our problem-solving skills.

Collaboration and Knowledge Sharing: Working on this laboratory task provided an opportunity for collaboration and knowledge sharing among peers, which was beneficial for overcoming obstacles and refining our understanding of the concepts.

In conclusion, this laboratory work has been instrumental in providing us with a comprehensive understanding of formal languages, regular grammars, and finite automata. The skills and knowledge acquired here are fundamental to our future studies and professional development in the field of Software Engineering and Automata Theory.