MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Formal Languages and Finite Automata

Laboratory work 2 : Determinism in Finite Automata. Conversion from NDFA 2 DFA. Chomsky Hierarchy.

Elaborated:
st.gr. FAF-211 Nistor Stefan
Verified:
asist.univ. Vasile Drumea

Chișinău, 2023

Content

Introduction

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

1. Understand what an automaton is and what it can be used for.
2. 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.
   (a) For this you can use the variant from the previous lab.
3. 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).

1 Implementation

1. Not much, just to mention that it would be enough for you to implement the project just to work with your specific variant. Of course, it would be gr8 if you could make it as generic as possible.
2. In order to show the execution you can implement a client class/type, which is just a ”Main” class/type in which you can instantiate the types/classes. Another approach would be to write unit tests if you are familiar with them.

1.1 Code:

```
    class Main:
# Initialize the Main class by setting up a grammar,
# converting it to a finite automaton, and setting up a FiniteAutomaton object
def __init__(self):
    self.productions = {
        'S': ['aA', 'aB'],
        'A': ['bS'],
        'B': ['aC'],
        'C': ['a', 'bS'],
    }
    self.start_symbol = 'S'
    self.grammar = Grammars(self.productions, self.start_symbol)
    self.finite_automaton = self.grammar.to_finite_automaton()
    self.automaton = FiniteAutomaton

# Generates strings from the grammar
def generate_strings(self, num_strings):
    for i in range(num_strings):
        string = self.grammar.generate_string()
        print(string)

if __name__ == '__main__':
    # Create a Main object
    main = Main()
```

```
    # Generate and print 5 strings from the grammar
    main.generate_strings(5)

    # Convert the grammar to a finite automaton
    automatons = main.grammar.to_finite_automaton()

    # Define a finite automaton manually
    automaton = {
        'states': {'q0', 'q1', 'q2', 'q3', 'q4', 'q5'},
        'alphabet': {'a', 'b'},
        'transition': {
            'q0': {'a': 'q1'},
            'q1': {'b': 'q2', 'a': 'q3'},
            'q2': {'a': 'q4'},
            'q3': {'a': 'q1', 'b': 'q5'},
            'q4': {'a': 'q5', 'b': 'q5'},
            'q5': {'a': 'q5', 'b': 'q5'}
        },
        'start_state': 'q0',
        'final_states': {'q5'}
    }

    # Create a FiniteAutomaton object for the manually-defined automaton
    checker = FiniteAutomaton(automaton)

    # Check whether a list of strings are accepted by the finite automaton
    checker.check_strings(['aaa', 'abaaa', 'ababaa', 'aa', 'abababa'])

    # Print the finite automaton
    print(automatons)

automation = Automaton()

automation.states = ['q0', 'q1', 'q2', 'q3']
```

```
automation.alphabet = ['a', 'b', 'c']
automation.transitions = {('q0', 'a'): ['q0', 'q1'],
                  ('q1', 'b'): ['q2'],
                  ('q2', 'a'): ['q2'],
                  ('q3', 'a'): ['q3'],
                  ('q2', 'b'): ['q3']}
automation.start_state = 'q0'
automation.accept_states = ['q3']
print('')
print('')
print('')
# Check if automaton is deterministic
is_deterministic = automation.is_deterministic()
print(f"Is automaton deterministic? {is_deterministic}")

# Convert NDFA to DFA
dfa = automation.to_dfa()
print(f"DFA states: {dfa.states}")
print(f"DFA transition function: {dfa.transitions}")
print(f"DFA initial state: {dfa.start_state}")
print(f"DFA final states: {dfa.accept_states}")

# Convert automaton to regular grammar
grammar = automation.to_grammar()
print(f"Regular grammar productions: {grammar}")
```

Here is the 'Automaton' class

```
   import matplotlib.pyplot as plt
import numpy as np
import networkx as nx
class Automaton:
    def __init__(self):
        # Initializes the attributes of the automaton object
```

```
        self.states = {'q0', 'q1', 'q2', 'q3'}  # Set of states
        self.alphabet = {'a', 'b', 'c'}  # Set of input symbols
        self.start_state = 'q0'  # The start state
        self.accept_states = {'q3'}  # Set of accept states
        self.transitions = {
            ('q0', 'a'): {'q0', 'q1'},
            ('q1', 'b'): {'q2'},
            ('q2', 'a'): {'q2'},
            ('q3', 'a'): {'q3'},
            ('q2', 'b'): {'q3'}
        }  # Dictionary of transition function mappings

    def is_deterministic(self):
        # Checks if the automaton is deterministic
        for state in self.states:
            for symbol in self.alphabet:
                next_states = self.transitions.get((state, symbol), set())
                if len(next_states) != 1:
                    return False
        return True

    def to_dfa(self):
        # Converts the automaton to a deterministic finite automaton (DFA)
        if self.is_deterministic():
            return self

        dfa_states = set()
        dfa_accept_states = set()
        dfa_transitions = dict()
        state_queue = [frozenset([self.start_state])]
        while state_queue:
            current_states = state_queue.pop(0)
            dfa_states.add(current_states)
            if any(state in self.accept_states for state in current_states):
```

```
                dfa_accept_states.add(current_states)
            for symbol in self.alphabet:
                next_states = set()
                for state in current_states:
                    next_states |= set(self.transitions.get((state, symbol), set()))
                if next_states:
                    next_states = frozenset(next_states)
                    dfa_transitions[(current_states, symbol)] = next_states
                    if next_states not in dfa_states:
                        state_queue.append(next_states)

    dfa = Automaton()
    dfa.states = dfa_states
    dfa.accept_states = dfa_accept_states
    dfa.transitions = dfa_transitions
    return dfa

def to_grammar(self):
    # Initialize an empty dictionary to hold the productions
    productions = dict()

    # For each state and symbol in the automaton's transitions,
    # create a production for each possible next state.
    for state in self.states:
        for symbol in self.alphabet:
            next_states = self.transitions.get((state, symbol), set())
            for next_state in next_states:
                if next_state in self.accept_states:
                    # If the next state is an accept state, add a production
                    # that generates the symbol for the current state.
                    if state not in productions:
                        productions[state] = set()
                    productions[state].add(symbol)
                else:
```

```
                    # If the next state is not an accept state, add a production
                    # that generates the symbol concatenated with the next state.
                    if next_state not in productions:
                        productions[next_state] = set()
                    productions[next_state].add(symbol + state)

    # Set the start symbol to be the start state of the automaton.
    start_symbol = self.start_state
    if start_symbol in productions:
        # If there is already a production for the start symbol, rename it to "S"
        # and add new productions that use "S" instead of the start symbol.
        productions['S'] = productions[start_symbol]
        del productions[start_symbol]
        for state in self.states:
            for symbol in self.alphabet:
                next_states = self.transitions.get((state, symbol), set())
                for next_state in next_states:
                    if state in productions and symbol + state in productions[state]
                        if next_state not in productions:
                            productions[next_state] = set()
                        productions[next_state].add(symbol + 'S')
    else:
        # If there is no production for the start symbol, create one that generates
        # the empty string and concatenates it with each accept state.
        start_symbol = 'S'
        productions[start_symbol] = set()
        for accept_state in self.accept_states:
            productions[start_symbol].add('eps' + accept_state)

    # Return the start symbol and the productions.
    return start_symbol, productions

def render(self):
    # Create a directed graph using networkx
```

```
G = nx.DiGraph()

# Add nodes to the graph
for state in self.states:
    G.add_node(state, shape='circle')
G.nodes[self.start_state]['shape'] = 'doublecircle'
for state in self.accept_states:
    G.nodes[state]['peripheries'] = 2

# Add edges to the graph
for (from_state, symbol), to_states in self.transitions.items():
    for to_state in to_states:
        G.add_edge(from_state, to_state, label=symbol)

# Set up positions for the nodes using networkx spring_layout
pos = nx.spring_layout(G, seed=42)

# Draw the graph using matplotlib
nx.draw_networkx_nodes(G, pos, node_size=1000, alpha=0.8)
nx.draw_networkx_edges(G, pos, width=2, alpha=0.8)
nx.draw_networkx_labels(G, pos, font_size=18, font_family='sans-serif')
edge_labels = {(u, v): d['label'] for u, v, d in G.edges(data=True)}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=18, font
plt.axis('off')
plt.show()
```

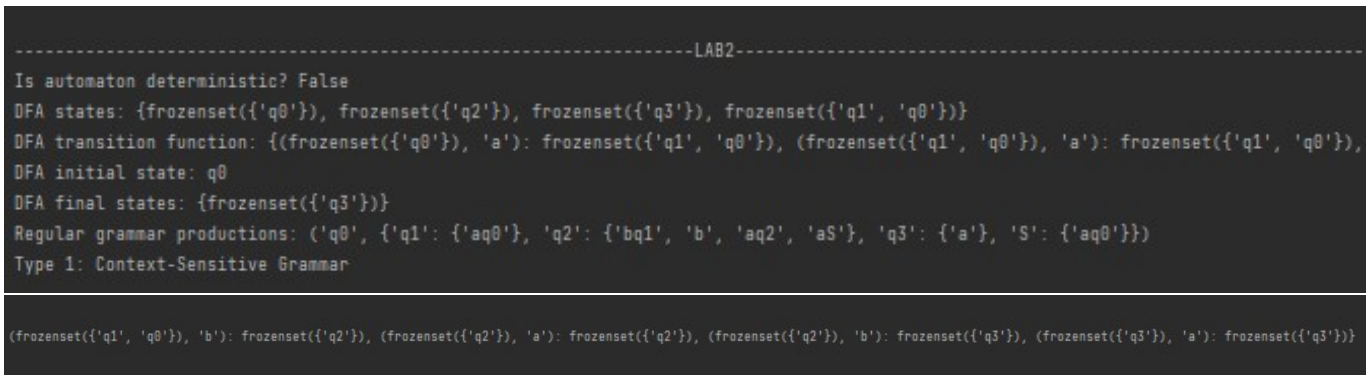
Here is the 'Chomsky-classification' function in the Grammar class

```
def chomsky_classification(self):
# Check if each production in the grammar is in Chomsky normal form
for symbol, productions in self.productions.items():
    for production in productions:
        # If the production is a single terminal symbol, it's in CNF
        if len(production) == 1 and production.islower():
            continue
        # If the production is a pair of nonterminal symbols, it's in CNF
```

```
            elif len(production) == 2 and production.isupper():
                continue
            # If the production is a single nonterminal symbol, it's not in CNF
            elif len(production) == 1 and production.isupper():
                return "Type 0: Unrestricted Grammar"
            # If the production is not a pair of nonterminal symbols or a single ter
            elif len(production) != 2 or not production.isupper():
                return "Type 1: Context-Sensitive Grammar"

    # Check if the start symbol has a production that only consists of the empty str
    if self.start_symbol in self.productions and '' in self.productions[self.start_s
        # If there are any other productions for the start symbol, the grammar is no
        if len(self.productions[self.start_symbol]) > 1:
            return "Type 2: Context-Free Grammar"
        # If there are no other productions for the start symbol, the grammar is in
        else:
            return "Type 3: Regular Grammar"
    # If the start symbol doesn't have a production that only consists of the empty
    else:
        return "Type 2: Context-Free Grammar"
```

1.2 Screenshot:



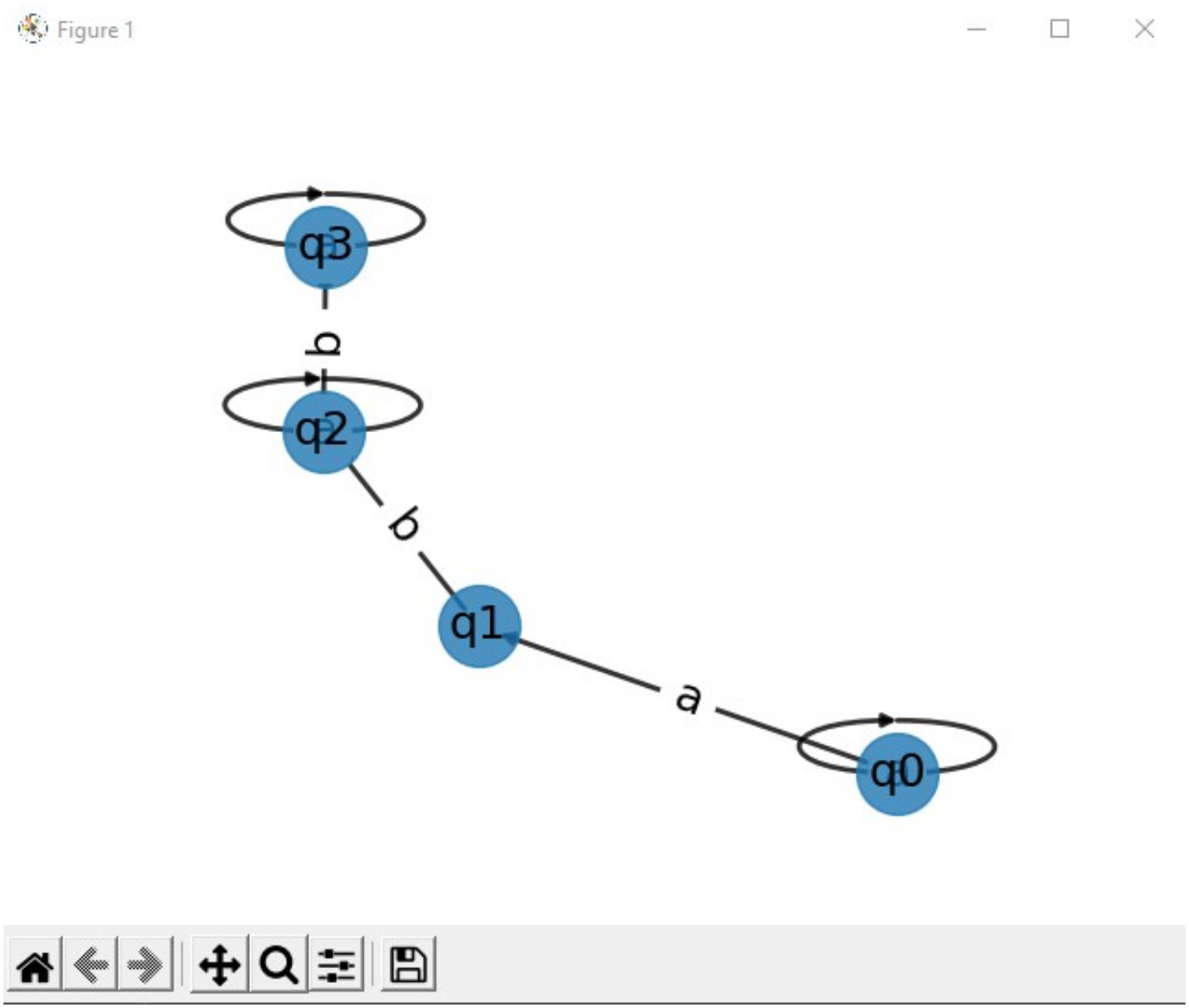


Figure 1
q3
q2
q1
q0
a
b
a

Conclusions

In the field of computer science, automata theory is a crucial topic that deals with the study of abstract computing devices that are capable of recognizing and generating languages. Finite automata are a type of automaton that is widely used to represent different kinds of processes.

A finite automaton is essentially a mathematical model of a machine that can move through a finite number of states based on its current state and input symbols. The finite nature of the automaton means that it has a starting state and a set of final states, which together determine the beginning and end of the process modeled by the automaton.

The structure of a finite automaton is quite similar to that of a state machine, and the two are often used interchangeably. However, while a state machine can be used to represent any kind of process, a finite automaton is specifically designed for processes that can be modeled using a finite number of states.

One of the key challenges when working with finite automata is dealing with non-determinism. This occurs when a single input symbol can lead to multiple states, making the behavior of the system unpredictable. This issue can be addressed by modifying the structure of the automaton using algorithms to create a deterministic version.

Determinism is an essential concept in systems theory, which characterizes how predictable a system is. If a system involves random variables, it becomes stochastic or non-deterministic. Finite automata can be classified as either deterministic or non-deterministic, depending on their structure and behavior.

In conclusion, finite automata are a powerful tool in computer science, used to model a variety of processes. While non-determinism can make the behavior of the system unpredictable, algorithmic modifications can be made to create a deterministic version. Understanding automata theory is essential in formal language theory, parsing, compilers, and other areas of computer science. Finite automata are a fundamental concept that can help us understand the nature of computation and the behavior of complex systems.