MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA

TECHNICAL UNIVERSITY OF MOLDOVA

FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS

Laboratory work 2: Formal Languages and Finite Automata

Elaborated:

st. gr. FAF-223 **Vieru Mihai**

Verified:

asist. univ. **Cretu Dumitru**

Chișinău, 2024

Content

IMPLEMENTATION 3

ProblemA: 3

ProblemB: 4

Problem C: 4

Problem D: 8

CONCLUSION 11

IMPLEMENTATION

ProblemA:

In order to solve problem A, I transformed my given transitions into a grammar by using python .replace function to remake the transition strings.

TransitionsToGrammar:

```python
def get_visual_grammar(transitions, states):
    mapping = generate_mapping(states)
    replaced_transitions = transitions
    for key, value in mapping.items():
        replaced_transitions = replaced_transitions.replace(key, value)

    replaced_transitions = replaced_transitions.replace(",\n","\n").replace(".", "").replace("(", "").replace(")", "").replace(" = ", "").replace(" ","")
    replaced_transitions = replaced_transitions.replace(",", " → ")
    return replaced_transitions
```

Then I utitlize another function to display the final node with a epsilon transition.

DisplayingAndAddingEpsilon:

```python
def displaygrammar(mapping, parsedgrammar, finalstates):
    grammar = unmapgrammar(mapping, parsedgrammar)
    for finalstate in finalstates:
        grammar += f"\n{final_state} → "

    return grammar
```

Result:

$$q0 \to aq0$$

$$q0 \to aq1$$

$$q1 \to aq1$$

$$q1 \to cq2$$

$$q1 \to bq3$$

$$q0 \to bq2$$

$$q2 \to bq3$$

$$q3 \to \varepsilon$$

ProblemB:

In order to detect whether my FA is determininistic or not I just check If I have dublicates in my array of possible moves from a state.

LookForDublicatesInAStateTransition:

```
def isRepeating(parsedgrammar):
    for key, vals in parsedgrammar.items():
        terminals = [y for x in vals for y in x if y == y.lower()]
        terminalsset = set(terminals)

        if len(terminals) != len(terminalsset):
            return True
    return False
```

Problem C:

In order convert an NDFA to a DFA, first I make transition table for all of the possible states.

MakeTrasitionTable:

```
def get_transition_table(self):
        transition_table = {}
        for state in self.Q:
            mapped_state = self.mapped_states[state]
            mapped_transitions = self.parsed_grammar[mapped_state]
```

```
                transition_table.setdefault(mapped_state, [])
                # print(mapped_transitions)
                for terminal in self.alphabet:
                    # print(terminal, [list(x)[-1] for x in mapped_transitions
                     ↪   if terminal in x])
                    transition_table[mapped_state].append(set((list(x)[-1] for
                     ↪   x in mapped_transitions if terminal in x)))

            return transition_table
```

Result:

δ	a	b	c
q0	{'q0', 'q1'}	{'q2'}	{}
q1	{'q1'}	{'q3'}	{'q2'}
q2	{}	{'q3'}	{}
q3	{}	{}	{}

Then I utilize this table to create new states. I do this by creating the prime transition table, and for every possible transition to add the states, If a new state is spoted it enteres a recursive loop, untill there isnt any state left.

MakeStateTrasitionTable:

```
    def get_prime_transition_table(self, transition_table):
            new_Q = set()
            prime_transition_table = {}

            def nfa_to_dfa(state):
                terminals = [set()] * len(self.alphabet)
                for part_of_state in state:
                    for non_terminal_id in range(len(self.alphabet)):
                        terminals[non_terminal_id] = terminals[non_terminal_id]
                         ↪   | transition_table[part_of_state][non_terminal_id]

                prime_transition_table[state] = ["".join(x) for x in terminals]
```

```
            for terminal in terminals:
                hashable_terminal = "".join(terminal)
                if hashable_terminal not in new_Q and len(terminal) >= 1:
                    new_Q.add(hashable_terminal)
                    nfa_to_dfa(hashable_terminal)

        nfa_to_dfa(self.mapped_states[self.initial_state])
        return prime_transition_table
```

Result:

δ'	a	b	c
q0	['q0q1', 'q2']	{}	{}
q0q1	['q0q1', 'q2q3', 'q2']	{}	{}
q2q3	{}	['q3']	{}
q3	{}	{}	{}
q2	{}	['q3']	{}

Then using this table I just generate the DFA:

GenerateDFA:

```
def get_grammar_and_final_states(self, prime_transition_table,
    final_states):
        grammar = ""
        alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ".lower()
        new_final_states = set()
        for key, val in prime_transition_table.items():
            for state_id in range(len(val)):
                if len(val[state_id]) >= 1:
                    for final_state in final_states:
                        if final_state in val[state_id]:
                            new_final_states.add(val[state_id])
                    grammar += f"{key} →
                     {alphabet[state_id]}{val[state_id]} \n"
```

```
        return grammar, list(new_final_states)



    def map_final_states(self):
        final_states = list(self.final_states)
        for state_id in range(len(final_states)):
            final_states[state_id] =
             ↪  self.mapped_states[final_states[state_id]]
        return final_states



    def unmap_grammar_and_final_states(self, dfa_grammar, final_states):
        for state_id in range(len(final_states)):
            final_states[state_id] = unmap_grammar(self.mapped_states,
             ↪  final_states[state_id])

        dfa_grammar = unmap_grammar(self.mapped_states, dfa_grammar)
        return dfa_grammar, final_states



    def nfa_to_dfa(self):
        transition_table = self.get_transition_table()

        self.print_transition_table(transition_table)

        prime_transition_table =
         ↪  self.get_prime_transition_table(transition_table)

        final_states = self.map_final_states()

        dfa_grammar, final_states =
         ↪  self.get_grammar_and_final_states(prime_transition_table,
         ↪  final_states)
```

```
        dfa_grammar, final_states =
         ↪  self.unmap_grammar_and_final_states(dfa_grammar, final_states)
```

Result:

$$
\begin{aligned}
q0 &\rightarrow aq0q1 \\
q0 &\rightarrow bq2 \\
q0q1 &\rightarrow aq0q1 \\
q0q1 &\rightarrow bq2q3 \\
q0q1 &\rightarrow cq2 \\
q2q3 &\rightarrow bq3 \\
q2 &\rightarrow bq3 \\
q3 &\rightarrow \varepsilon
\end{aligned}
$$

Problem D:

In order to represent the finite automaton graphically. After I have created the dfa I just used the code from the last laboratory, since my graphing code was pretty robust.

Implementation:

```
import networkx as nx
import matplotlib.pyplot as plt
from ProblemB import Grammar


class AutomatonVisualizer:
    def __init__(self, grammar):
        cGrammar = Grammar(grammar)
        self.data = cGrammar.parseGrammar(grammar)
        self.start_symbol = "Add a start symbol"
        self.end_symbols = []
```

```
    def addEndSymbols(self, end_symbol):
        if isinstance(end_symbol, list):
            self.end_symbols.extend(end_symbol)
        else:
            self.end_symbols.append(end_symbol)

    def setStartSybol(self, start_symbol):
        self.start_symbol = start_symbol

    def cleanData(self, parsed_grammar):
        for non_terminal, productions in parsed_grammar.items():
            for node in productions:
                if node == node.lower():
                    parsed_grammar[non_terminal].remove(node)
        return parsed_grammar

    def generateGraph(self):
        G = nx.DiGraph()

        edge_labels = {}
        self_loop = {}
        for node, neighbors in self.data.items():
            for neighbor in neighbors:
                transition = neighbor[0]
                target_node = neighbor[1:]
                G.add_edge(node, target_node)
                if node == target_node:
                    if node in self_loop:
                        self_loop[node] += f", {transition}"
                    else:
                        self_loop[node] = transition

                if (target_node, node) in edge_labels:
```

```
                    edge_labels[(target_node, node)] += f", {transition}"
                else:
                    edge_labels[(node, target_node)] = transition

        pos = nx.spring_layout(G, k=0.5)
        nx.draw(G, pos, with_labels=True, node_color='skyblue',
         ↪  node_size=1500, edge_color='gray', arrowsize=20)

        for symbol in self.end_symbols:
            circle_radius = 0.14
            circle_center = pos[symbol]
            circle_patch = plt.Circle(circle_center, circle_radius,
             ↪  color='red', fill=False)
            plt.gca().add_patch(circle_patch)

        for node, transition in self_loop.items():
            pos_labels = pos.copy()
            pos_labels[node][1] += 0.10
            nx.draw_networkx_labels(G, pos_labels, labels={node:
             ↪  transition}, font_color='black')



        nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)

        pos_labels = pos.copy()
        pos_labels[self.start_symbol][0] -= 0.13
        nx.draw_networkx_labels(G, pos_labels, labels={self.start_symbol:
         ↪  '->'}, font_color='black', font_weight='bold')

        plt.show()
```

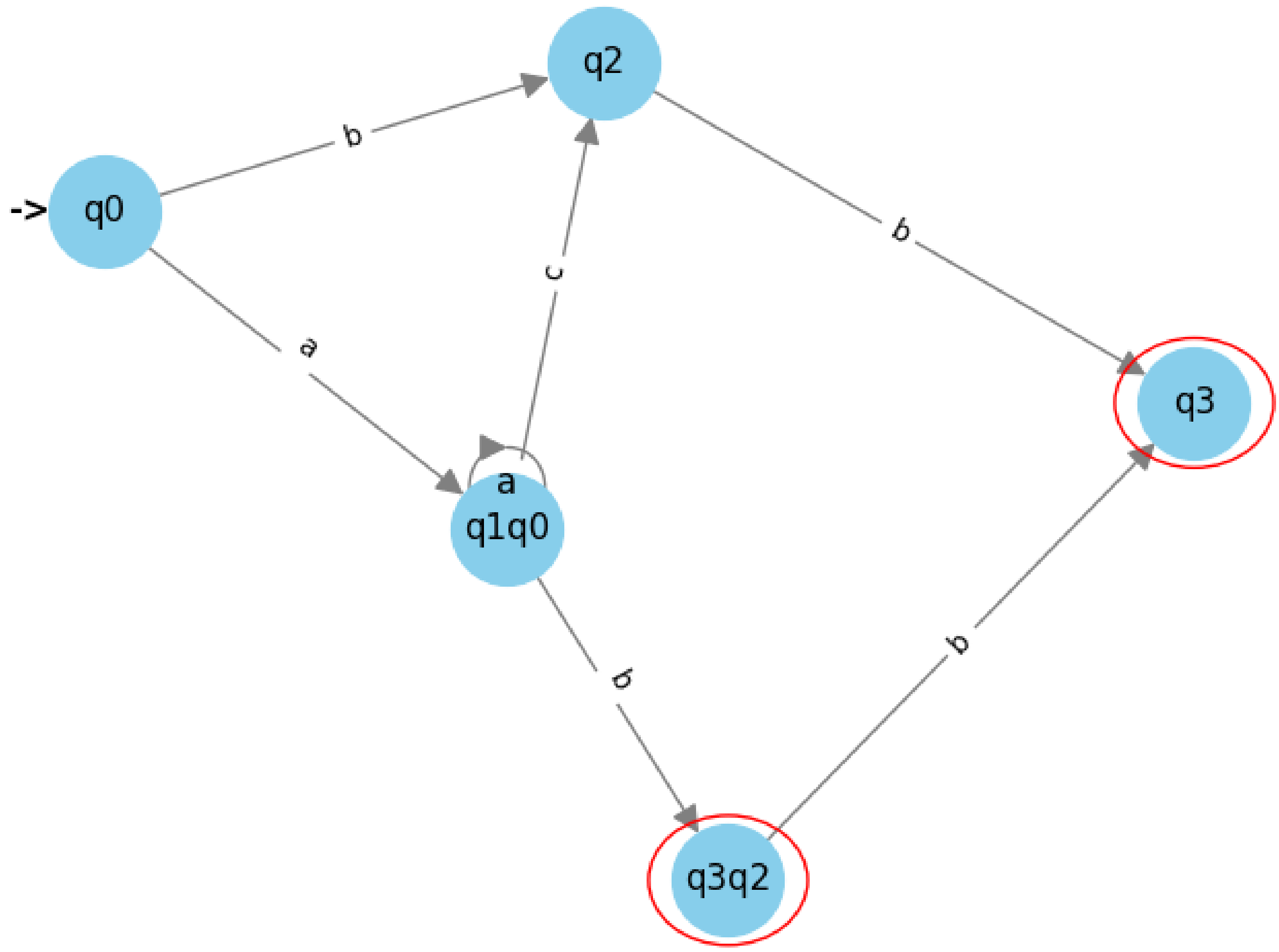

Figure .1 - Graphical DFA

CONCLUSION

In this lab, I explored the fundamental principles of finite automata (FA) and their applications. I achieved the following:

Understanding Grammars: I learned how to transform transition systems into grammars using Python code. This included replacing symbols and adding epsilon transitions to represent final nodes. Analyzing Determinism: I developed code to analyze FA for determinism by identifying duplicate transition states, indicating a non-deterministic FA. NDFA to DFA Conversion: I successfully implemented a multi-step process to convert non-deterministic finite automata (NDFA) to deterministic finite automata (DFA). This involved creating transition tables, identifying new composite states, and recursively generating the final DFA. Visualization of Automata: Finally, I integrated previous work to graphically represent the generated DFA. This visual output aids in understanding the automaton's structure. This lab solidified my comprehension of finite automata. I've gained valuable experience in manipulating FA representations, determining determinism, performing conversions, and creating visualizations –skills crucial for further exploration of computational theory.