MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică

CEBAN VASILE FAF-223

Report

Laboratory work n.2

of Formal Languages and Finite Automata

Checked by:

Dumitru Crețu, *university assistant*

DISA, FCIM, UTM

Chișinău - 2024

Finite Automata Task

Topic: Determinism in Finite Automata. Conversion from NDFA to DFA. Chomsky Hierarchy.

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.
   b. 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*):
      - 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.

The second task involves working with a finite automaton derived from a regular grammar.

- Type 0 (Unrestricted Grammar): There are no restrictions, meaning any grammar is allowed.
- Type 1 (Context-Sensitive Grammar): Rules have the form $\alpha \rightarrow \beta$ where length(α) <= length(β), and there's at least one rule where length(α) < length(β).
- Type 2 (Context-Free Grammar): Rules have the form $A \rightarrow \gamma$, where A is a nonterminal and γ is a string of terminals and/or nonterminals.
- Type 3 (Regular Grammar): Rules have the form $A \rightarrow aB$ or $A \rightarrow a$, where A and B are nonterminals, and a is a terminal.

Source Code:

```
import networkx as nx
import matplotlib.pyplot as plt

class FiniteAutomaton:
   def __init__(self, states, alphabet, transitions, initial_state,
accepting_states):
       self.states = states
       self.alphabet = alphabet
       self.transitions = transitions
       self.initial_state = initial_state
       self.accepting_states = accepting_states

   def is_deterministic(self):
       seen_transitions = set()
       for state, symbol in self.transitions.keys():
           if (state, symbol) in seen_transitions:
               return False
           seen_transitions.add((state, symbol))
       return True

   def convert_to_regular_grammar(self):
       grammar = {}
       for state, symbol in self.transitions.keys():
           destination = self.transitions[(state, symbol)]
           if (state, destination) not in grammar:
               grammar[(state, destination)] = []
           grammar[(state, destination)].append(symbol)

       return grammar

   def convert_to_dfa(self):
       if self.is_deterministic():
           return self  # Already a DFA

       dfa_states = set()
       dfa_transitions = {}
       dfa_initial_state = self.initial_state
       dfa_accepting_states = set()

       queue = [self.epsilon_closure([self.initial_state])]
       visited = set()

       while queue:
           current_states = queue.pop(0)
           current_states = tuple(sorted(current_states))

           if current_states in visited:
               continue
           visited.add(current_states)

           dfa_states.add(current_states)
```

```
            for symbol in self.alphabet:
                next_states = set()
                for state in current_states:
                    if (state, symbol) in self.transitions:

next_states.update(self.epsilon_closure([self.transitions[(state,
symbol)]]))
                next_states = tuple(sorted(next_states))

                if not next_states:
                    continue

                dfa_transitions[(current_states, symbol)] = next_states

                if next_states not in dfa_states:
                    queue.append(next_states)

                if any(state in self.accepting_states for state in
next_states):
                    dfa_accepting_states.add(next_states)

        return FiniteAutomaton(dfa_states, self.alphabet, dfa_transitions,
dfa_initial_state, dfa_accepting_states)

    def epsilon_closure(self, states):
        epsilon_closure_states = set(states)
        queue = list(states)

        while queue:
            current_state = queue.pop(0)
            epsilon_transitions = self.transitions.get((current_state, ''),
[])
            for state in epsilon_transitions:
                if state not in epsilon_closure_states:
                    epsilon_closure_states.add(state)
                    queue.append(state)

        return list(epsilon_closure_states)

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

        for state in self.states:
            G.add_node(state, shape='circle', color='blue')
            if state in self.accepting_states:
                G.nodes[state]['color'] = 'green'  # Accepting state

        for (source, symbol), destination in self.transitions.items():
            G.add_edge(source, destination, label=symbol)

        pos = nx.spring_layout(G)
        node_colors = [G.nodes[state]['color'] for state in G.nodes]
```

```
       edge_labels = {(source, destination): symbol for (source, symbol),
destination in self.transitions.items()}

       nx.draw_networkx_nodes(G, pos, node_color=node_colors)
       nx.draw_networkx_edges(G, pos)
       nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)
       nx.draw_networkx_labels(G, pos)

       plt.show()


# Main client class
if __name__ == "__main__":
   # Finite automaton definition
   states = {'q0', 'q1', 'q2', 'q3', 'q4'}
   alphabet = {'a', 'b', 'c'}
   transitions = {('q0', 'a'): 'q1', ('q1', 'b'): 'q2', ('q1', 'b'): 'q3',
                  ('q2', 'c'): 'q3', ('q3', 'a'): 'q3', ('q3', 'b'): 'q4'}
   initial_state = 'q0'
   accepting_states = {'q4'}

   fa = FiniteAutomaton(states, alphabet, transitions, initial_state,
accepting_states)

   # a. Convert to regular grammar
   print("a. Convert to regular grammar")
   regular_grammar = fa.convert_to_regular_grammar()
   print("Regular Grammar:", regular_grammar)
   print(" ")

   # b. Determine whether the FA is deterministic or non-deterministic
   print("b. Determine whether the FA is deterministic or
non-deterministic")
   if fa.is_deterministic():
       print("The Finite Automaton is deterministic.")
   else:
       print("The Finite Automaton is non-deterministic.")
   print(" ")

   # c. Convert NDFA to DFA
   print("c. Convert NDFA to DFA")
   dfa = fa.convert_to_dfa()
   print("DFA States:", dfa.states)
   print("DFA Transitions:", dfa.transitions)
   print("DFA Initial State:", dfa.initial_state)
   print("DFA Accepting States:", dfa.accepting_states)

   # d. Represent the finite automaton graphically (Optional)
   fa.draw_graph()
```

Result:

```
a. Convert to regular grammar
Regular Grammar: {('q0', 'q1'): ['a'], ('q1', 'q3'): ['b'], ('q2', 'q3'): ['c'], ('q3', 'q3'): ['a'], ('q3', 'q4'): ['b']}

b. Determine whether the FA is deterministic or non-deterministic
The Finite Automaton is deterministic.

c. Convert NDFA to DFA
DFA States: {'q4', 'q2', 'q1', 'q0', 'q3'}
DFA Transitions: {('q0', 'a'): 'q1', ('q1', 'b'): 'q3', ('q2', 'c'): 'q3', ('q3', 'a'): 'q3', ('q3', 'b'): 'q4'}
DFA Initial State: q0
DFA Accepting States: {'q4'}
```

Figure 2. Result at a, b and c points.

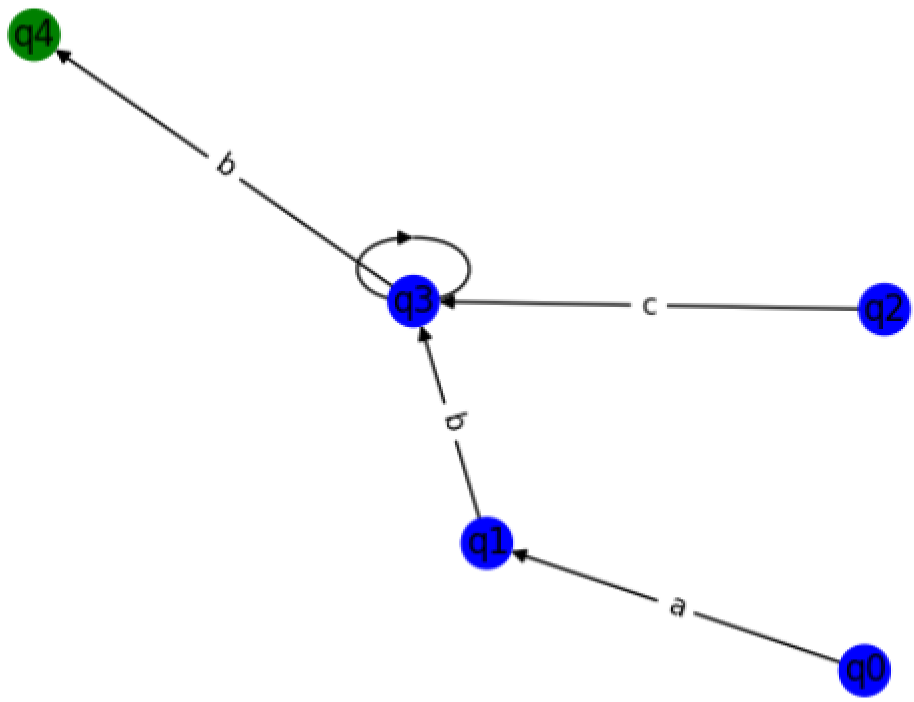

Figure 2. The optional point. Graphic representation of Finite Automata.