Ministry of Education and Research of the Republic of Moldova
Technical University of Moldova
Department of Software and Automation Engineering

REPORT

Laboratory work No. 2
Discipline: Formal Languages & Finite Automata
Topic: Determinism in Finite Automata. Conversion from NDFA 2 DFA. Chomsky Hierarchy.

Done by: Barbarov Nadejda st.gr., FAF-221

Checked by:
asist.univ. Crețu Dumitru

Chișinău 2024

Theory

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.

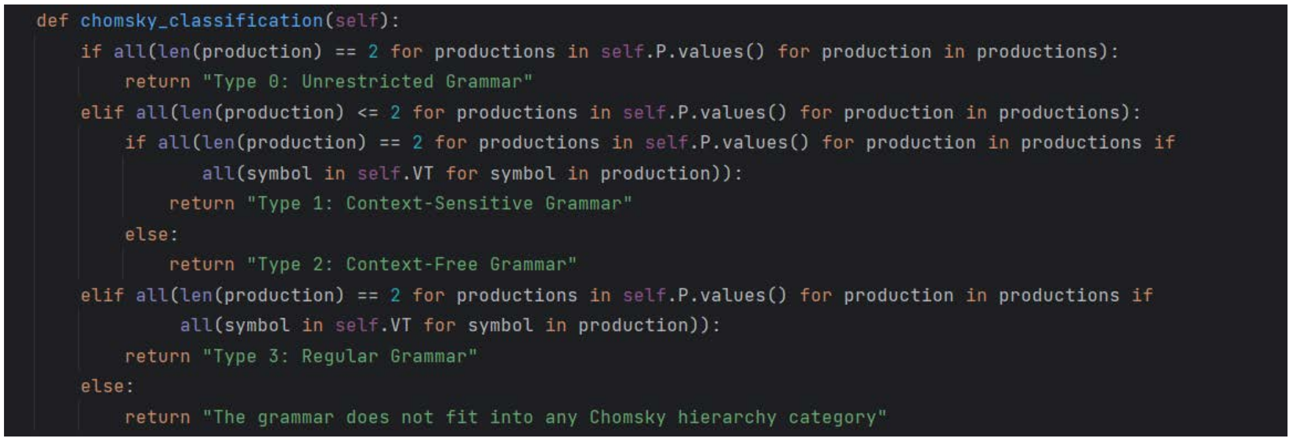
  c. Implement some functionality that would convert an NDFA to a DFA.

  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.

Implementation description

First of all, in the previous program I added a method that checks the type of grammar. This method classifies grammars based on the Chomsky hierarchy, which categorizes grammars into four types according to their generative power. It starts by checking if all productions consist of exactly two symbols, concluding that the grammar is Type 0: Unrestricted Grammar if so. If not, it checks if productions have at most two symbols and, if they do, whether they consist solely of terminal symbols, classifying the grammar as Type 3: Regular Grammar if true or Type 2: Context-Free Grammar otherwise. If some productions have more than two symbols, it classifies the grammar as Type 1: Context-Sensitive Grammar if they consist solely of terminal symbols. If none of these conditions are met, it concludes that the grammar does not fit into any Chomsky hierarchy category.



```
def chomsky_classification(self):
    if all(len(production) == 2 for productions in self.P.values() for production in productions):
        return "Type 0: Unrestricted Grammar"
    elif all(len(production) <= 2 for productions in self.P.values() for production in productions):
        if all(len(production) == 2 for productions in self.P.values() for production in productions if
                all(symbol in self.VT for symbol in production)):
            return "Type 1: Context-Sensitive Grammar"
        else:
            return "Type 2: Context-Free Grammar"
    elif all(len(production) == 2 for productions in self.P.values() for production in productions if
              all(symbol in self.VT for symbol in production)):
        return "Type 3: Regular Grammar"
    else:
        return "The grammar does not fit into any Chomsky hierarchy category"
```

Figure 1. Chomsky classification

The next method converts a finite automaton (FA) into a regular grammar. It iterates over the states and symbols of the FA to create productions, which map each state-symbol pair to its corresponding next states. It also adds epsilon transitions as empty strings to the productions. Finally, it generates grammar productions based on the transitions and adds productions for final state.

```
def fa_to_regular_grammar(Q, sigma, delta, q0, F):
    # Initialize the productions dictionary
    productions = {}

    # Add productions for each state
    for state in Q:
        for symbol in sigma:
            productions[(state, symbol)] = []

    # Add transitions to the productions
    for (state, symbol), next_states in delta.items():
        for next_state in next_states:
            productions[(state, symbol)].append(next_state)

    # Add epsilon transitions
    for state in Q:
        productions[(state, '')] = []
    # Generate the grammar productions
    grammar_productions = []

    for state, transitions in productions.items():
        if transitions:
            for next_state in transitions:
                grammar_productions.append(f"{state[0]} ->
{state[1]}{next_state}")

    # Add final state productions
    for state in F:
        grammar_productions.append(f"{state} -> ε")

    return grammar_productions
```

The next method checks if a finite automaton (FA) is deterministic. It iterates over each transition in the FA's transition function and checks if any input symbol has multiple next states. If any such transition is found, it concludes that the FA is non-deterministic; otherwise, it determines the FA to be deterministic.

```
def is_deterministic(delta):
    # Iterate over each transition
    for (state, symbol), next_states in delta.items():
        # Check if there are multiple next states for the same input symbol
        if len(next_states) > 1:
            return False

    # If no transition has multiple next states for the same input symbol, the
FA is deterministic
    return True
```

Next method computes the epsilon closure of a given state in a non-deterministic finite automaton (NFA). It starts with the given state and explores all epsilon transitions to reach other states reachable from the initial state via epsilon transitions.

```
def epsilon_closure(state, delta):
    closure = set()
    stack = [state]

    while stack:
        current_state = stack.pop()
        closure.add(current_state)

        # Check epsilon transitions
        if (current_state, '') in delta:
            for next_state in delta[(current_state, '')]:
                if next_state not in closure:
                    stack.append(next_state)
    return closure
```

This method converts a non-deterministic finite automaton (NFA) into a deterministic finite automaton (DFA). It initializes the DFA's states, alphabet, transition function, initial state, and final states. Then, it performs epsilon closure on the initial state to obtain the DFA's initial state and iteratively constructs the DFA's states and transitions based on the NFA's transitions and epsilon closures.

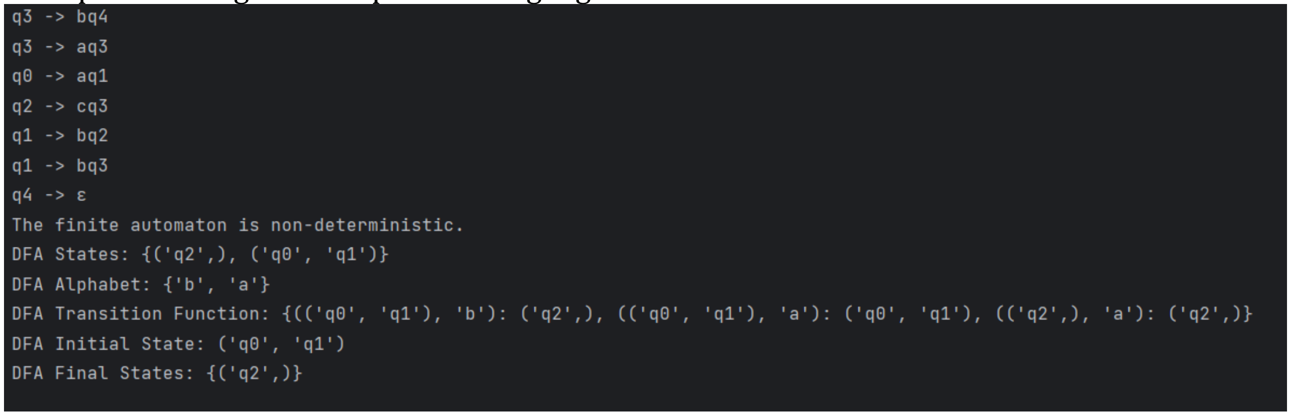
```
def nfa_to_dfa(Q, sigma, delta, q0, F):
    dfa_states = set()
    dfa_sigma = sigma
    dfa_delta = {}
    dfa_initial = epsilon_closure(q0, delta)
    dfa_final = set()
    stack = [dfa_initial]

    while stack:
        current_state_set = stack.pop()
        dfa_states.add(tuple(sorted(current_state_set)))

        # Check if the current state set contains a final state of the NFA
        if any(state in F for state in current_state_set):
            dfa_final.add(tuple(sorted(current_state_set)))

        for symbol in dfa_sigma:
            next_state_set = set()

            for state in current_state_set:
                if (state, symbol) in delta:
                    next_state_set.update(delta[(state, symbol)])

            next_state_set_closure = set()
            for state in next_state_set:
                next_state_set_closure |= epsilon_closure(state, delta)

            if next_state_set_closure:
                dfa_delta[(tuple(sorted(current_state_set)), symbol)] =
tuple(sorted(next_state_set_closure))

                if tuple(sorted(next_state_set_closure)) not in dfa_states:
                    stack.append(next_state_set_closure)

    return dfa_states, dfa_sigma, dfa_delta, tuple(sorted(dfa_initial)),
dfa_final
```

Next method visualizes a finite automaton (FA) graphically using the NetworkX library. It creates a directed graph representing the FA's states and transitions, with nodes representing states and edges representing transitions labeled with symbols. The initial state is highlighted, and final states are distinguished by a different color. The resulting graph is displayed using matplotlib.

```
def draw_fa(Q, delta, q0, F):
    # Create a directed graph
    G = nx.DiGraph()

    # Add states as nodes
    for state in Q:
        if state in F:
            G.add_node(state, color='pink', style='filled',
shape='doublecircle')
        else:
            G.add_node(state, color='turquoise', style='filled',
shape='circle')

    # Add transitions as edges
    for (state, symbol), next_states in delta.items():
        for next_state in next_states:
            G.add_edge(state, next_state, label=symbol)

    # Highlight initial state
    G.nodes[q0]['color'] = 'lightgreen'

    # Define node and edge attributes
    node_colors = [G.nodes[node]['color'] for node in G.nodes()]
    edge_labels = {(u, v): d['label'] for u, v, d in G.edges(data=True)}

    # Draw the graph
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True, node_color=node_colors, node_size=1000,
arrows=True)
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)
    plt.show()
```

Conclusions / Screenshots / Results

In this laboratory work, I focused on an important aspect of finite automata: determinism. I investigated the process of converting non-deterministic finite automata (NDFA) into deterministic ones (DFA), which is crucial for language recognition. This involved creating algorithms to handle the conversion, helping me understand how states and transitions interact. Additionally, I explored the Chomsky hierarchy, a system for categorizing grammars by their complexity. Through these efforts, I gained valuable insights into how computers recognize and process languages.



```
q3 -> bq4
q3 -> aq3
q0 -> aq1
q2 -> cq3
q1 -> bq2
q1 -> bq3
q4 -> ε
The finite automaton is non-deterministic.
DFA States: {('q2',), ('q0', 'q1')}
DFA Alphabet: {'b', 'a'}
DFA Transition Function: {(('q0', 'q1'), 'b'): ('q2',), (('q0', 'q1'), 'a'): ('q0', 'q1'), (('q2',), 'a'): ('q2',)}
DFA Initial State: ('q0', 'q1')
DFA Final States: {('q2',)}
```

Figure 2. End results

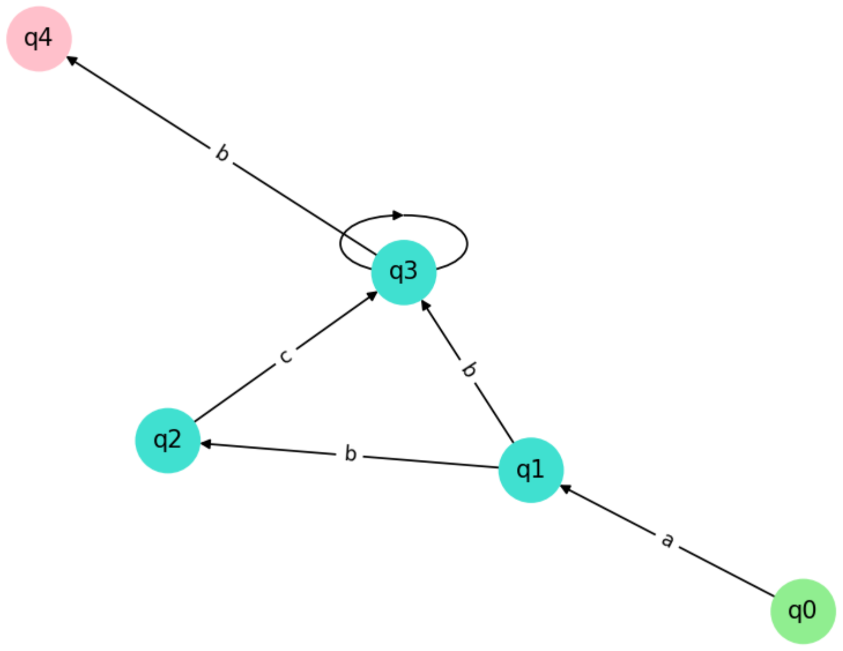


Figure 3. FA Graphical representation

References

1. **Github.**https://github.com/filpatterson/DSL_laboratory_works/blob/master/2_FiniteAutomata/task.md

2. **Github.** https://github.com/nadea-b/LFA