Laboratory Work No. 1

Course: Formal Languages & Finite Automata

Student: Iordan Liviu, Group: FAF-223

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, a 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:
  - Implement conversion of a finite automaton to a regular grammar.
  - Determine whether your FA is deterministic or non-deterministic.
  - Implement some functionality that would convert an NDFA to a DFA.

Description Implementation:

This file (**main.py**) serves as the entry point of the program and the place where all the tests are located.

- Verification of the first requirement: classify the grammar based on Chomsky hierarchy.

```
# Define the grammar variant
grammar = Grammar()
grammar.VN = {'S', 'B', 'D'}
grammar.VT = {'a', 'b', 'c', 'd'}
grammar.P = {
   'S': ['aS', 'bB'],
   'B': ['cB', 'd', 'aD'],
   'D': ['aB', 'b']
}

# Check the type of each grammar
print("Grammar Classification:", grammar.check_grammar_type())
```

The output:

Grammar Classification: Type-3 (Regular)

- Verification of the second requirement: Implement conversion of a finite automaton to a regular grammar.

```
finite_automaton = FiniteAutomaton()
finite_automaton.Q = {'q0', 'q1', 'q2'}
finite_automaton.Sigma = {'a', 'b', 'c'}
finite_automaton.delta = {('q0', 'a', 'q0'), ('q0', 'b', 'q1'), ('q1', 'c',
'q1'),
                          ('q1', 'c', 'q2'), ('q2', 'a', 'q0'), ('q1', 'a',
'q1')}
finite_automaton.q0 = 'q0'
```

```
finite_automaton.F = {'q2'}


# Convert finite automaton to regular grammar
regular_grammar = finite_automaton.to_regular_grammar()


# Print the regular grammar productions
print("Regular Grammar Productions for the NDFA:")
for non_terminal, productions in regular_grammar.P.items():
   for production in productions:
       print(non_terminal, "->", production)
```

The output:
Regular Grammar Productions for the NDFA:

q1 -> aq1

q1 -> cq2

q1 -> cq1

q2 -> aq0

q0 -> bq1

q0 -> aq0

- Verification of the third requirement: Determine whether your FA is deterministic or non-deterministic.

```
is_deterministic = finite_automaton.is_deterministic()
if is_deterministic:
   print("The NDFA is deterministic.")
else:
   print("The NDFA is non-deterministic.")
```

The ouput: The NDFA is non-deterministic.

- Verification of the fourth requirement: Implement some functionality that would convert an NDFA to a DFA.

The output: The converted DFA is deterministic.

The file (**grammar.py**) was modified from the last laboratory work. The following function was added:

```
def check_grammar_type(self):
    start_symbol = None
    has_epsilon = False
    for non_terminal, productions in self.P.items():
        if not start_symbol:
            start_symbol = non_terminal
        for production in productions:
            if 'ε' in production:
                has_epsilon = True
            if len(production) > 2:
                return "Type-0 (Unrestricted)"
            if len(production) == 2:
                if production[0] in self.VN and production[1] in self.VT:
                    return "Type-1 (Context-Sensitive)"
            if len(production) == 1:
                if production[0] in self.VT:
                    return "Type-3 (Regular)"
    if start_symbol and not has_epsilon:
        return "Type-2 (Context-Free)"
    return "Type-0 (Unrestricted)"
```

This function **check_grammar_type** examines the production of a grammar to determine its type according to the Chomsky hierarchy.

1. It starts by initializing variables start_symbol and has_epsilon.
2. It iterates through the productions of the grammar.
3. If a production contains the empty string symbol ('ε'), it sets has_epsilon to True.
4. It checks the length of each production:
    - If a production has more than two symbols, the grammar is classified as Type-0 (Unrestricted).
    - If a production has exactly two symbols, and the first symbol is a non-terminal (VN) and the second symbol is a terminal (VT), the grammar is classified as Type-1 (Context-Sensitive).
    - If a production has only one symbol and it is a terminal (VT), the grammar is classified as Type-3 (Regular).
5. If the grammar does not fall into the Type-0, Type-1, or Type-3 categories based on the productions, it further checks:

- If there is a start symbol and no productions contain the empty string symbol ('ε'), the grammar is classified as Type-2 (Context-Free).
- Otherwise, it is classified as Type-0 (Unrestricted).

6. The function returns the determined type of the grammar.

The file (**finite_automaton.py**) was also modified from the last laboratory work. The following functions were added:

- This **to_regular_grammar** function converts a finite automaton (FA) into a regular grammar.

```
def to_regular_grammar(self):
    regular_grammar = Grammar()
    regular_grammar.VN = self.Q
    regular_grammar.VT = self.Sigma
    regular_grammar.P = {}

    for state in self.Q:
        regular_grammar.P[state] = []

    for transition in self.delta:
        if transition[2] != 'X':
            next_state_str = ''.join(transition[2])  # Convert tuple to string
            regular_grammar.P[transition[0]].append(transition[1] +
next_state_str)  # Concatenate strings

    return regular_grammar
```

The function works in the following way:

Initialization:

a. It creates an empty regular grammar object.
b. It assigns the set of states (self.Q) of the finite automaton to the non-terminal symbols (VN) of the regular grammar.
c. It assigns the set of input symbols (self.Sigma) of the finite automaton to the terminal symbols (VT) of the regular grammar.
d. It initializes an empty dictionary for the productions (P) of the regular grammar.

Populating Productions:

e. It iterates through each state in the finite automaton.
f. For each state, it initializes an empty list of productions in the regular grammar.
g. It then iterates through each transition in the finite automaton (self.delta).
h. If the next state of the transition is not 'X' (indicating a non-final state), it converts the next state tuple into a string.

i. It concatenates the input symbol and the next state string and appends it to the list of productions for the current state in the regular grammar.

Return:

j. It returns the regular grammar object containing the converted productions.

- Is DFA:This is_deterministic function checks whether the finite automaton (FA) is deterministic.

```
def is_deterministic(self):
   for state in self.Q:
       for symbol in self.Sigma:
           next_states = {next_state for (_, input_symbol, next_state) in
self.delta
                          if _ == state and input_symbol == symbol}
           if len(next_states) > 1:
               return False
   return True
```

Here is what the function does:

Nested Loop:

a. It iterates through each state in the set of states (self.Q) of the finite automaton.
b. Inside the outer loop, it iterates through each symbol in the set of input symbols (self.Sigma) of the finite automaton.

Next States:

a. For each state and symbol combination, it collects the set of next states that can be reached from the current state with the given symbol. This set of next states is obtained from the transitions (self.delta) of the finite automaton.

Determinism Check:

a. If the number of next states for any state-symbol combination is greater than 1, it means that the finite automaton is non-deterministic because there are multiple possible transitions for a single state and input symbol.
b. In this case, it immediately returns False to indicate that the finite automaton is non-deterministic.
c. If no state-symbol combination results in multiple next states, it means that the finite automaton is deterministic, and it returns True.

- This to_deterministic_finite_automaton function converts a non-deterministic finite automaton (NFA) to a deterministic finite automaton (DFA)

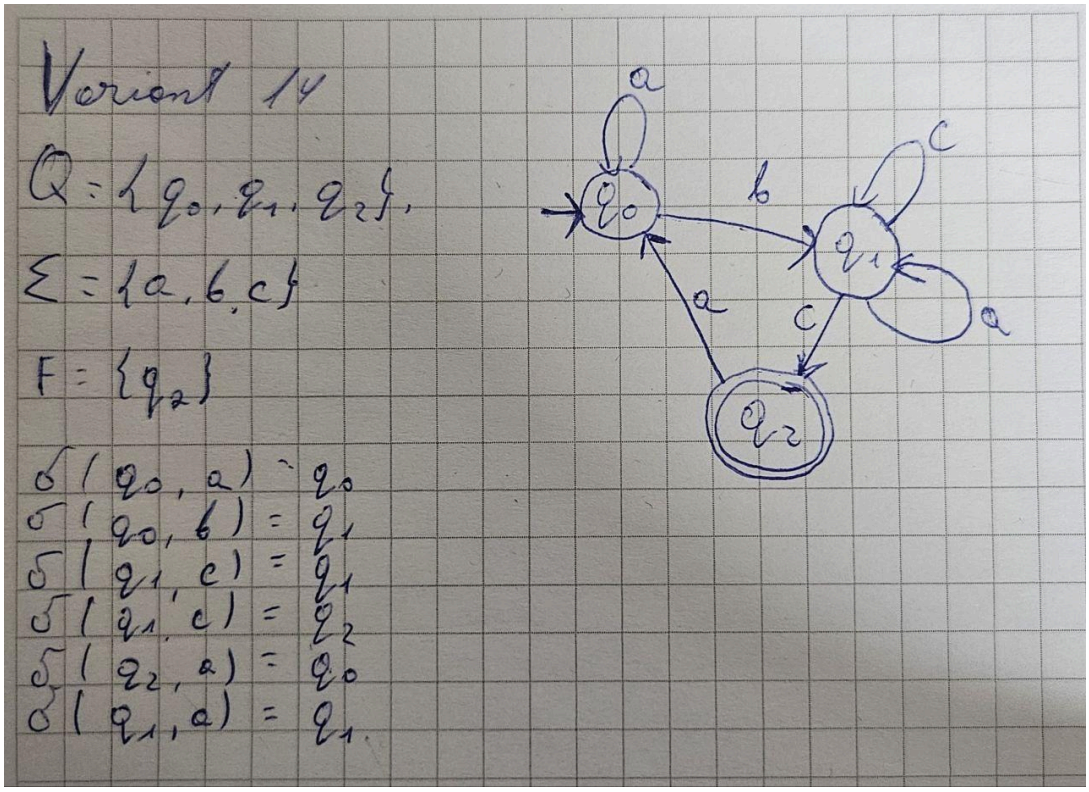
```
def to_deterministic_finite_automaton(self):
   dfa = FiniteAutomaton()
   dfa.Sigma = self.Sigma
```

```
    dfa.q0 = frozenset([self.q0])  # Initial state is the epsilon closure of the
original initial state
    dfa.F = set()
    dfa.Q = set()  # Initialize set of states
    dfa.delta = set()

    # Compute epsilon closure of a state in the NFA
    def epsilon_closure(state):
        closure = set(state)
        stack = list(state)
        while stack:
            currentState = stack.pop()
            for (_, input_symbol, nextState) in self.delta:
                if currentState == nextState and input_symbol == 'ε' and
nextState not in closure:
                    closure.add(nextState)
                    stack.append(nextState)
        return frozenset(closure)

    # Initialize unprocessed states with the epsilon closure of the initial
state
    unprocessed_states = [dfa.q0]
    dfa.Q.add(dfa.q0)

    # Explore states of the DFA using subset construction algorithm
    while unprocessed_states:
        current_state = unprocessed_states.pop(0)
        for symbol in dfa.Sigma:
            next_state = set()
            for state in current_state:
                # Find next states according to the NFA transitions and epsilon
closures
                next_state |= {next_state for (_, input_symbol, next_state) in
self.delta
                               if state in current_state and input_symbol ==
symbol}
            next_state_closure = epsilon_closure(next_state)
            if next_state_closure:
                dfa.delta.add((current_state, symbol, next_state_closure))
                if next_state_closure not in dfa.Q:
                    dfa.Q.add(next_state_closure)
                    unprocessed_states.append(next_state_closure)
                if any(state in self.F for state in next_state_closure):
                    dfa.F.add(next_state_closure)

    return dfa
```

- Initialization:
    - It initializes a new instance of the FiniteAutomaton class to represent the DFA.

    - It copies the alphabet (input symbols) of the original NFA to the DFA.
    - It sets the initial state (q0) of the DFA as the epsilon closure of the original initial state of the NFA.
    - It initializes empty sets to represent the set of states (Q), the set of final states (F), and the set of transitions (delta) of the DFA.
- Epsilon Closure Function:
    - It defines a nested function epsilon_closure to compute the epsilon closure of a state in the NFA.
    - The epsilon closure of a state includes the state itself and all states reachable from it via epsilon transitions.
    - It uses a depth-first search algorithm to find all states reachable via epsilon transitions.
- Subset Construction:
    - It initializes a list unprocessed_states with the epsilon closure of the initial state of the DFA.
    - It initializes the set of states of the DFA (Q) with the epsilon closure of the initial state.
    - It explores the states of the DFA using the subset construction algorithm.
    - For each current state in the DFA:
        - For each input symbol in the alphabet:
            - It computes the set of next states by considering all possible transitions from the current state with the given input symbol in the NFA.
            - It computes the epsilon closure of the set of next states.
            - It adds a transition to the DFA from the current state to the next state (with the input symbol) if the next state is not empty.
            - If the next state is not already in the set of states of the DFA, it adds it to the set and to the list of unprocessed states.
            - If any state in the next state closure is a final state in the NFA, it adds the next state closure to the set of final states of the DFA.
- Return:
    - It returns the DFA representing the deterministic version of the original NFA.

- Graphical representation of the NDFA:



Conclusion:
In conclusion, this laboratory work has provided a comprehensive exploration of finite automata, focusing on determinism, conversion between different types of grammars, and the relationship between automata and formal language theory. Throughout this laboratory work, several key objectives were achieved.

Firstly, an in-depth understanding of finite automata and their significance in modeling processes was attained. This included grasping the fundamental concepts of states, transitions, and acceptance criteria within automata, as well as recognizing the distinction between deterministic and non-deterministic automata.

Secondly, practical implementation skills were developed through the creation of Python classes to represent grammars and finite automata. This involved defining methods for generating strings from grammars, converting between different types of grammars, and determining the type of a grammar based on the Chomsky hierarchy. Additionally, methods for checking determinism and converting non-deterministic finite automata (NFA) to deterministic finite automata (DFA) were implemented.

Furthermore, the process of converting between grammars and finite automata provided valuable insights into the relationship between formal languages and automata theory. This process

required careful consideration of the structure and rules of both grammars and automata, highlighting the interconnected nature of these theoretical concepts.

Overall, this laboratory work has been instrumental in deepening understanding and enhancing practical skills related to formal languages and finite automata. By completing the assigned tasks, valuable knowledge was gained regarding determinism in automata, conversion algorithms between different types of grammars and automata, and the broader implications of these concepts in the field of computer science and linguistics. Through hands-on implementation and experimentation, a solid foundation has been established for further exploration and study in this fascinating area of theoretical computer science.