MINISTERUL EDUCAŢIEI ȘI CERCETĂRII AL REPUBLICII MOLDOVA
UNIVERSITATEA TEHNICĂ A MOLDOVEI
FACULTATEA CALCULATOARE, INFORMATICĂ ȘI MICROELECTRONICĂ

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

Course: Formal Languages and Finite Automata

Author: Cucoș Maria

Chișinău, 2024

TABLE OF CONTENTS

Theory

Formal Languages

A formal language is a set of strings of symbols drawn from some alphabet. These languages are used to describe the syntax of programming languages, regular expressions, and many other areas. An alphabet is a finite set of symbols used to form strings. Strings are sequences of symbols from an alphabet.

In computer science, this formal language is essential for the definition of computer programs and the expression of algorithmic problems. Formal Languages are classified into different levels based on the Chomsky hierarchy. These levels include Recursively enumerable languages (type 0), Context-free languages (type 1), Context-sensitive languages (type 2) and Regular languages (type 3). These languages all have different sets of rules for construction and provide different levels of expressibility. Furthermore, formal language theory provides systematic ways to determine whether a given string adheres to the rules of a language, which is fundamental for the creation of software like compilers or interpreters.

Finite Automata

Finite Automata are abstract computational devices used to recognize patterns within strings. It's characterised by limited memory and the potential to change from one state to another when triggered by external inputs. Types of Finite Automata:

- Deterministic Finite Automata (DFA)
- Nondeterministic Finite Automata (NFA)

Components of Finite Automata:

- States - represent different configurations of the machine.
- Transitions - define the movement from one state to another based on input symbols.
- Accepting states - indicate whether a particular input string is accepted or rejected.

Acceptance Criteria is when a string is accepted if the automaton reaches an accepting state after processing the entire input string. Finite automata can recognize exactly the class of regular languages.

Deterministic Finite Automata

Deterministic Finite Automata (DFA) is a type of Finite Automata where for each state and input symbol, there exists one and only one transition. This essentially means that a DFA cannot have multiple paths for the same input from any given state or an undefined path. DFAs function on a finite set of input symbols and from each state for every input symbol, the automaton deterministically transits to a next state. This brings about deterministic computation, allowing DFAs to process regular languages, which are the simplest form of formal languages in computer science. A DFA accepts an input string if and only if the DFA ends in an accepting (or final) state after processing the entire string.

Non-Deterministic Finite Automata

Non-Deterministic Finite Automata (NFA) is a variation of Finite Automata in which one or more specific condition transitions are not necessarily defined for all states, or there may be several uniquely defined transitions for the same state and input symbol. The ace that a NFA holds over a DFA is its ability to transition to multiple next states from a particular state for the same input symbol. Alternatively, an NFA can choose to completely neglect an input symbol from a state, leading it to a null transition. This allows more flexibility in modelling real-world computational problems. NFAs recognise the same class of languages as DFAs, known as regular languages, though sometimes with a simpler and more intuitive structure.

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, it should be added a function in the grammar class that could classify the grammar based on Chomsky hierarchy.
3. According to my variant number, get the finite automaton definition and do the following tasks:
   a. Implement conversion of a finite automaton to a regular grammar.
   b. Determine whether my FA is deterministic or non-deterministic.
   c. Implement some functionality that would convert an NFA to a DFA.
   d. Represent the finite automaton graphically (Optional, and can be considered as a bonus point).

Implementation Description

In this project, a grammar defined by a set of non-terminals (VN), terminals (VT), and production rules (P) was implemented to generate strings. Additionally, the grammar was converted into a Finite Automaton (FA) to determine the acceptability of generated and pre-defined strings.

Chomsky grammar classification

To classify the grammar based on the Chomsky hierarchy, I had to analyze the production rules to determine the highest type of grammar it represents. The function `classify_grammar()` was added to the Grammar class that determines the Chomsky hierarchy type.

For the `is_regular` function it checks if all production rules are of the form $\mathrm{A} -> \mathrm{a}$ or $\mathrm{A} -> \mathrm{aB}$ where 'a' is a terminal and 'B' is a non-terminal. The `is_context_free` checks if it's not regular but each production rule has only one non-terminal on the left-hand side. In the `is_context_sensitive` function the program checks if it's not context-free but each production rule can rewrite any string of terminals and non-terminals.

```
def classify_grammar(self):
    is_regular = all(len(self.P[non_terminal]) == 1 and
        (len(self.P[non_terminal][0]) == 1 or
        (len(self.P[non_terminal][0]) == 2 and
        self.P[non_terminal][0][1] in self.VN))
        for non_terminal in self.P)

    is_context_free = not is_regular and
        all(len(production) == 1 or
        (len(production) == 2 and
        production[1] in self.VN) for production in self.P)

    is_context_sensitive = not is_context_free and
        all(len(production) <= len(replacement)
        for production, replacement in self.P.items())

    # If none of the above, it's an unrestricted grammar
    return "This grammar is " + \
   ("Unrestricted " if not is_regular and not is_context_free
   and not is_context_sensitive else "") + \
   ("Context-Sensitive " if is_context_sensitive else "") + \
   ("Context-Free " if is_context_free else "") + \
   ("Regular" if is_regular else "")
```

FA to Regular Grammar conversion

Variant 8

$$Q = \{q0, q1, q2, q3, q4\},$$

$$\Sigma = \{a, b\},\ F = \{q3\},$$

$$\delta(q0, a) = q1,$$

$$\delta(q1, b) = q2,$$

$$\delta(q2, b) = q0,$$

$$\delta(q3, a) = q4,$$

$$\delta(q4, a) = q0,$$

$$\delta(q2, a) = q3,$$

$$\delta(q1, b) = q1.$$

The Finite Automaton definition can be written as the following:

$$F_A = (\Sigma, Q, \delta, q0, F)$$

The Grammar definition can be also represented as:

$$G = (V_T, V_N, P, S)$$

By equating both definitions and the extensions of the first one, we get the following:

$$V_T = \Sigma = a, b$$

$$V_N = Q = q0, q1, q2, q3, q4$$

$$P = \{q0- > aq1, q1- > bq1|bq2, q2- > aq3|bq0, q4- > aq0\}$$

In python I created a `Regular Grammar` class, that converts a finite automaton from the `main` class into a regular grammar. The `convert_from_fa` method adds non-terminals for each state in the finite automaton, the terminals from the finite automaton's alphabet and adds production rules for each transition.

```
class RegularGrammar:
def __init__(self):
    self.VN = set()
    self.VT = set()
    self.P = {}

def convert_from_fa(self, finite_automaton):
    self.VN.update(finite_automaton.states)
    self.VT.update(finite_automaton.alphabet)

    for state, transitions in finite_automaton.transitions.items():
        for symbol, next_state in transitions.items():
            production_rule = f"{state} -> {symbol}{next_state}"
            self.P.setdefault(state, []).append(production_rule)
```

In the `main` file I defined the finite automaton, created an instance of `RegularGrammar` and converted from finite automaton using the method `convert_from_fa`.

```
Q = {'q0', 'q1', 'q2', 'q3', 'q4'}
Sigma = {'a', 'b'}
F = {'q3'}
delta = {
    'q0': {'a': {'q1'}},
    'q1': {'b': {'q1', 'q2'}},
    'q2': {'b': {'q0'}, 'a': {'q3'}},
    'q3': {'a': {'q4'}},
    'q4': {'a': {'q0'}},
}
finite_automaton = FiniteAutomaton(Q, Sigma, delta, 'q0', F)
regular_grammar = RegularGrammar()
regular_grammar.convert_from_fa(finite_automaton)
print("Regular Grammar:")
print("Non-terminals:", regular_grammar.VN)
print("Terminals:", regular_grammar.VT)
print("Production Rules:")
for state, production_rules in regular_grammar.P.items():
    for rule in production_rules:
        print(" ", rule)
```

The output is the following:

```
Regular Grammar:
Non-terminals: {'q0', 'q3', 'q4', 'q1', 'q2'}
Terminals: {'a', 'b'}
Production Rules:
  q0 -> a{'q1'}
  q1 -> b{'q1', 'q2'}
  q2 -> b{'q0'}
  q2 -> a{'q3'}
  q3 -> a{'q4'}
  q4 -> a{'q0'}
```

NFA or DFA

In the `FiniteAutomaton` class I have created a new method that checks the finite automaton and states if it is a NFA or DFA. It iterates over each state in the finite automaton, then checks transitions for each symbol in the alphabet. If the symbol was seen before, the finite automaton is a NFA.

```
def is_deterministic(self):
    for state in self.states:
        transitions_from_state = self.transitions.get(state, {})
        seen_symbols = set()

        for symbol in self.alphabet:
            if symbol in transitions_from_state:
                if symbol in seen_symbols or len(transitions_from_state[symbol]) > 1:
                    return False
                seen_symbols.add(symbol)
    return True
```

In the main file the finite automaton is already defined as previously shown. The `if-else` block checks if the finite automaton is a DFA or NFA, the result is printed in the console.

```
    finite_automaton = FiniteAutomaton(Q, Sigma, delta, 'q0', F)
    if finite_automaton.is_deterministic():
        print("The finite automaton is deterministic.")
    else:
        print("The finite automaton is nondeterministic.")
```

The result is the following: `The finite automaton is nondeterministic.`

NFA to DFA

The initial state of a DFA is the epsilon closure of the initial state of the NFA. Since the given NFA doesn't use epsilon closure, I will not be considering epsilon transitions. In this case, the initial state of the DFA will be q0.

From the initial state {q0}:

- When a is input, the NFA transitions to {q1}.
- When b is input, the NFA transitions to {q1, q2}.

From state {q1}:

- When a is input, the NFA transitions to {q3}.
- There is no transition on b from {q1}.

From state {q1, q2}:

- When a is input, the NFA transitions to {q3}.
- When b is input, the NFA transitions to {q0}.

From final state {q3}:

- When a is input, the NFA transitions to {q4}.
- There is no transition on b from {q3}.

From state {q4}:

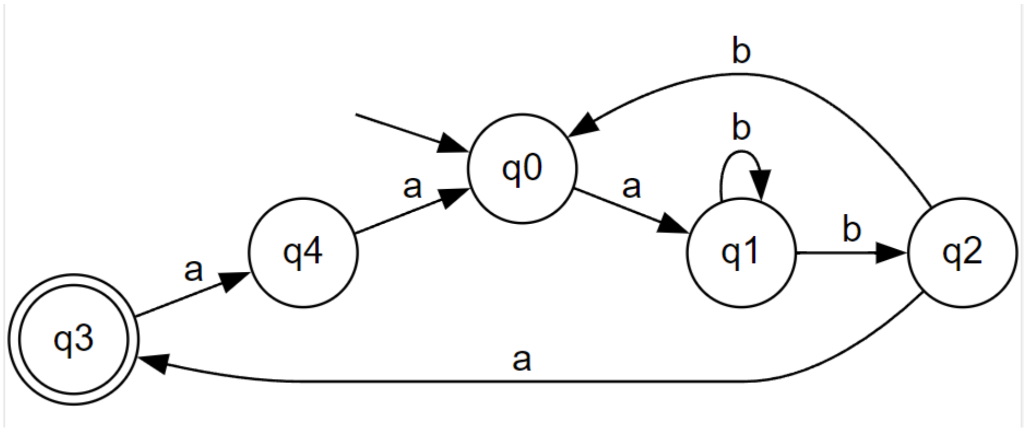
- When a is input, the NFA transitions to {q0}.
- There is no transition on b from {q4}.

In the original NFA, state q3 is a final state. Therefore, any state in the DFA containing q3 will also be a final state. This thing doesn't change, so {q3} remains the only final state.

Graph representation

I used the online tool called Graphviz Visual Editor to create the graph for the given finite automaton.

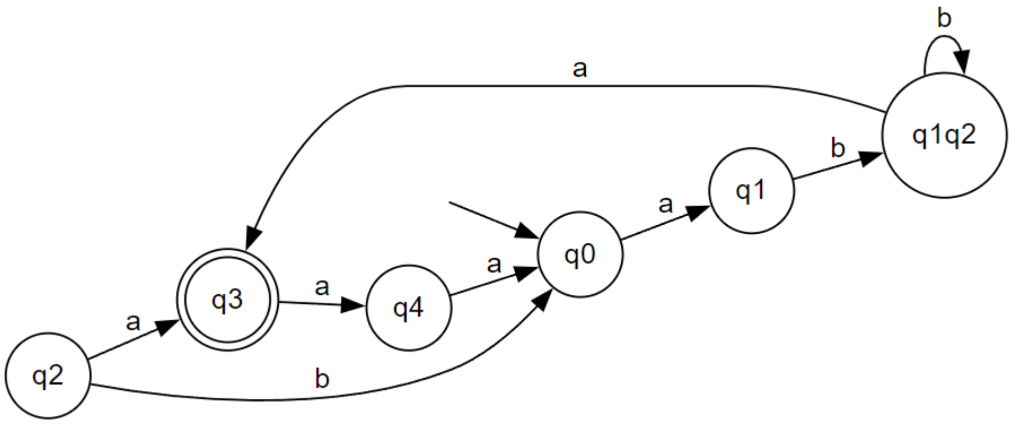
```
digraph finite_state_machine {
    fontname="Helvetica,Arial,sans-serif"
    node [fontname="Helvetica,Arial,sans-serif"]
    edge [fontname="Helvetica,Arial,sans-serif"]
    rankdir=LR;
    node [shape = doublecircle]; q3;
    node [shape = circle];
    q5 [style=invis];
    q5 -> q0;
    q0 -> q1 [label = "a"];
    q1 -> q1 [label = "b"];
    q1 -> q2 [label = "b"];
    q2 -> q3 [label = "a"];
    q2 -> q0 [label = "b"];
    q3 -> q4 [label = "a"];
    q4 -> q0 [label = "a"];
}
```



Graph representation for the given FA

The following figure shows the graphical representation of the given NFA converted to DFA.

```
q0 -> q1 [label = "a"];
q1 -> q1q2 [label = "b"];
q1q2 -> q3 [label = "a"];
q1q2 -> q1q2 [label = "b"];
q2 -> q3 [label = "a"];
q2 -> q0 [label = "b"];
q3 -> q4 [label = "a"];
q4 -> q0 [label = "a"];
```



Graph representation for the converted DFA

Conclusion

In this report, I explored the fundamental concepts of Automata theory and its practical applications. Automata, abstract models of computation, play a crucial role in computer science, linguistics, and various other fields. They are used to model and analyze the behavior of systems and languages, aiding in problem-solving and algorithm design.

I began by understanding the Chomsky hierarchy, a classification system for formal grammars, which categorizes grammars based on their generative power. By implementing a function within my grammar class, I was able to classify the given grammar according to this hierarchy, providing insights into the expressive capabilities of different grammar types.

Moving forward, I applied our understanding of finite automata (FA) to practical tasks. Leveraging the provided variant, I developed algorithms to convert finite automata to regular grammars, determine whether an FA is deterministic or nondeterministic, and implement functionality to convert nondeterministic finite automata (NDFA) to deterministic finite automata (DFA). These tasks involved intricate analyses of state transitions and symbol inputs, leading to the construction of equivalent grammars and automata with different properties.

Furthermore, I explored options to represent finite automata graphically, considering external libraries, tools, or APIs to generate visual representations of automaton structures. While not mandatory, visualizations can enhance understanding and facilitate communication of complex concepts, making them valuable additions to automata-related projects.

In summary, through these tasks, I gained practical insights into automata theory, formal language theory, and computational models, equipping myself with tools to model, analyze, and manipulate language and system behaviors effectively. These foundational concepts form the basis of numerous applications in computer science, linguistics, artificial intelligence, and beyond.