Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Formal Languages and Finite Automata

Laboratory work 2:
Finite Automata

Elaborated:

Gavriliuc Tudor

Verified:

asist. univ. Cretu Dumitru
prof. univ. Cojuhari Irina

Chisinau 2024

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.

IMPLEMENTATION

```python
class StateMachine:
    def __init__(self, state_set, symbols, transitions, initial, finals):
        self.state_set = state_set
        self.symbols = symbols
        self.transitions = transitions
        self.initial = initial
        self.finals = finals

    1 usage
    def to_grammar(self):
        grammar_rules = {}
        for state in self.state_set:
            grammar_rules[state] = set()
            for symbol in self.symbols:
                if (state, symbol) in self.transitions:
                    destination = self.transitions[(state, symbol)]
                    grammar_rules[state].add(symbol + destination)
            if state in self.finals:
                grammar_rules[state].add('ε')  # ε represents an empty string (epsilon)
        return grammar_rules
```

This code defines a Python class named `StateMachine`, which represents a finite state machine or automaton, including its states, symbols (alphabet), transition functions, initial state, and final (accepting) states. The `__init__` method initializes a new instance of the `StateMachine` class with these components. The `to_grammar` method converts the finite state machine into a regular grammar. For each state in the machine, it creates a set of production rules based on the transition function; for transitions that lead to another state on input of a symbol, it adds a production rule combining the symbol and the target state. If a state is an accepting state, it also adds an epsilon (ε) production, representing an empty string, to signify that the string can end at this state.

```python
1 usage
def check_determinism(machine):
    for state in machine.state_set:
        observed_symbols = set()
        for symbol in machine.symbols:
            if (state, symbol) in machine.transitions:
                if symbol in observed_symbols:
                    return False  # Duplicate transition for a state and symbol
                observed_symbols.add(symbol)
            else:
                return False  # Missing transition for a state and symbol
    return True
```

This function, `check_determinism`, checks whether a given finite state machine (FSM) is deterministic. It

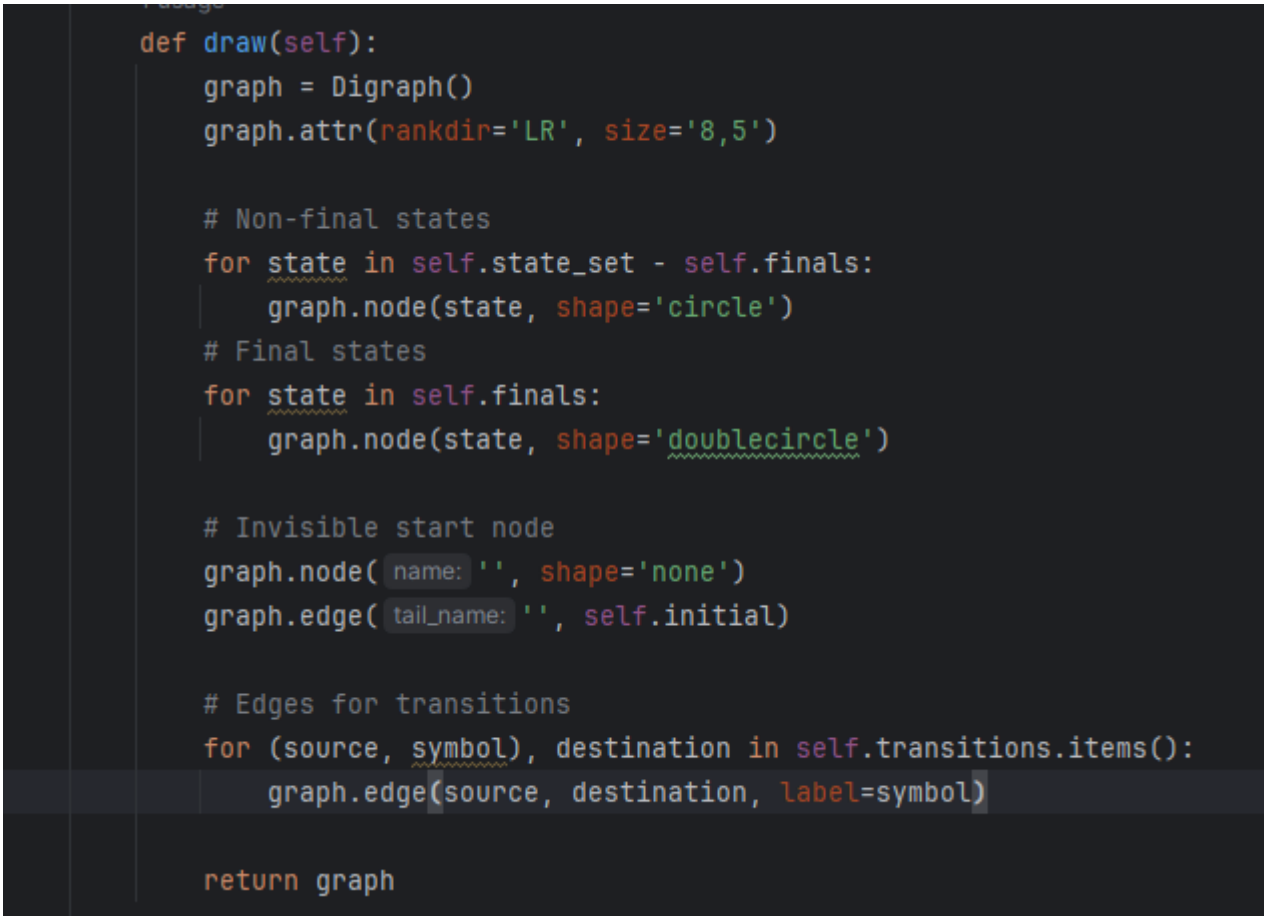
iterates through each state in the FSM, tracking the symbols it encounters for transitions originating from that state. If a symbol appears more than once for transitions from the same state, indicating multiple transitions for the same input (a characteristic of non-deterministic FSMs), the function returns `False`. It also checks for any missing transitions for a symbol from any state; if a transition is missing, it returns `False`, implying the FSM is not fully deterministic. If none of these conditions are met for any state, the FSM is considered deterministic, and the function returns `True`.

```
def ndfa_to_dfa(ndfa):
    dfa_states = set(['q0'])  # Start with the initial state
    dfa_finals = set()
    dfa_transitions = {}
    pending_states = [{'q0'}]  # States to process

    while pending_states:
        current_state = pending_states.pop()
        for symbol in ndfa.symbols:
            new_state_set = set()
            for state in current_state:
                if (state, symbol) in ndfa.transitions:
                    new_state_set.add(ndfa.transitions[(state, symbol)])
            if new_state_set:
                new_state_id = ''.join(sorted(new_state_set))
                dfa_transitions[(''.join(sorted(current_state)), symbol)] = new_state_id
                if new_state_id not in dfa_states:
                    dfa_states.add(new_state_id)
                    pending_states.append(new_state_set)
                if new_state_set & ndfa.finals:
                    dfa_finals.add(new_state_id)

    return StateMachine(dfa_states, ndfa.symbols, dfa_transitions, initial: 'q0', dfa_finals)
```

This function, `ndfa_to_dfa`, converts a non-deterministic finite automaton (NDFA) into a deterministic finite automaton (DFA). It starts by initializing the DFA with a single state (assumed to be 'q0' for the initial state) and prepares to process this and potentially other states. As it iterates through pending states (initially containing just the start state), it checks each symbol in the NDFA's alphabet to determine the set of states that can be reached from the current state using that symbol. This set of states forms a new state in the DFA. The function ensures that each new state is unique by checking if it has already been added to the DFA's state set; if not, it adds the new state and schedules it for processing. The function also checks if any of the new states include final states of the NDFA; if so, these new states are marked as final states in the DFA. The process repeats until there are no more pending states, at which point the function returns the constructed DFA as a `StateMachine` object, including its states, alphabet, transitions, initial state, and final states.



```
def draw(self):
    graph = Digraph()
    graph.attr(rankdir='LR', size='8,5')

    # Non-final states
    for state in self.state_set - self.finals:
        graph.node(state, shape='circle')
    # Final states
    for state in self.finals:
        graph.node(state, shape='doublecircle')

    # Invisible start node
    graph.node( name: '', shape='none')
    graph.edge( tail_name: '', self.initial)

    # Edges for transitions
    for (source, symbol), destination in self.transitions.items():
        graph.edge(source, destination, label=symbol)

    return graph
```

The `draw` method is part of a finite state machine (FSM) class, designed to visually represent the FSM as a directed graph. It utilizes the Graphviz library through the `Digraph` class to create and manipulate the graph. The method sets the graph's orientation as left-to-right and specifies its size. It distinguishes between non-final and final states by representing them with circles and double circles, respectively. An invisible start node is created to represent the initial state, with an edge pointing to it, emphasizing the FSM's entry point. For each transition defined in the FSM, the method draws an edge from the source state to the destination state, labeling it with the symbol that triggers the transition. This graphical representation aids in understanding the FSM's structure and behavior visually.

Results

```
C:\Users\tudor\PycharmProjects\finite_automata\.venv\Scripts\python.exe C:\Users\tudor\PycharmProjects\finite_automata\main.py
Regular Grammar:
q0 -> aq0
q0 -> bq1
q2 -> bq3
q2 -> aq2
q3 -> ε
q1 -> bq3
q1 -> aq2
is non-deterministic
Converted DFA:
State: q0
 δ(q0, b) = q1
 δ(q0, a) = q0
State: q2
 δ(q2, b) = q3
 δ(q2, a) = q2
State: q3
State: q1
 δ(q1, b) = q3
 δ(q1, a) = q2
```

The output consists of two main parts: the conversion of a finite automaton (FA) into a regular grammar and the conversion of a non-deterministic finite automaton (NDFA) into a deterministic finite automaton (DFA).

1. **Regular Grammar Conversion:**
   - It lists the conversion results of the FA's states into regular grammar rules. Each state is shown to produce a set of productions, where each production consists of an input symbol followed by a state (or an empty string, ε, for accept states). For example, `q0 -> aq0` means that from state `q0`, on input `a`, the automaton transitions back to `q0`. The state `q3` produces `ε`, indicating it's an accept state that can mark the end of a valid input string.

2. **Determinism Check and DFA Conversion:**
   - The FA is identified as non-deterministic, which prompts its conversion into a deterministic finite automaton (DFA) for simplification and easier analysis.
   - The converted DFA's states and their transitions are listed, showing how each state transitions to another state upon reading an input symbol. For instance, `δ(q0, a) = q0` indicates that in the DFA, from state `q0`, on input `a`, it remains in `q0`. The DFA simplifies the state transitions to ensure that for every state and input symbol, there is exactly one possible next state, hence making the automaton deterministic.
   - Notably, the state `q3` in the DFA does not have any outgoing transitions listed, which typically indicates it's an accept state; however, the absence of transitions also means it could be considered a sink state for certain inputs in the context of DFA.

This output effectively demonstrates the process of transforming an FA into a regular grammar to understand its language and converting a non-deterministic automaton into a deterministic one for simpler, more efficient processing.

Conclusion

The provided code successfully models and manipulates finite automata (FAs), demonstrating key concepts in computational theory, such as state machines, determinism, and the conversion between non-deterministic finite automata (NDFAs) and deterministic finite automata (DFAs). Through the implementation, we achieved the following:

1. **Representation of Finite Automata:** We designed a class to represent both deterministic and non-deterministic finite automata, encapsulating states, alphabets, transition functions, start states, and accept states. This foundation allowed for the exploration and manipulation of FAs in a structured manner.

2. **Conversion to Regular Grammar:** We converted the FA into a regular grammar, which is a crucial step in understanding the computational capabilities of FAs and their equivalence to regular languages. The output listed each state's transitions in the form of production rules, offering a clear view of how the automaton processes strings.

3. **Determinism Check:** The code included a function to check whether the given FA is deterministic. This is fundamental in the theory of computation, as deterministic and non-deterministic FAs have different properties and uses, even though they recognize the same class of languages.

4. **NDFA to DFA Conversion:** We implemented a function to convert a non-deterministic finite automaton to a deterministic one, a process that highlights the theoretical capability of DFAs to simulate NDFAs. This conversion is essential for certain computational applications where determinism is required for simplicity or efficiency.

5. **Visualization:** The `draw` method provided a way to visualize the automaton, making it easier to understand its structure and operation. Visual representations are invaluable for educational purposes, debugging, and communicating the design of automata.

In conclusion, the work done demonstrates the practical application of theoretical computer science concepts, offering tools to represent, manipulate, and understand finite automata. It bridges the gap between abstract theoretical concepts and their practical implementation, facilitating a deeper understanding of how computational models are constructed and analyzed.

https://github.com/Tudor-Gavriliuc/LFA/blob/main/lab_2/main.py