Determinism in Finite Automata.

Course: Formal Languages & Finite Automata

Author: Moraru Gabriel

Theory:

In the case of a **DFA**, there's one transition for every input from every state, thus indicating a single path. On the other hand, an **NFA** allows for multiple transitions and ε-transitions, and both will recognize the same languages. Because DFAs do not allow for these multiple transitions, the method used to convert an NFA to a DFA is known as subset construction

1. The sets of NFA states are treated as a single DFA state.

2. Transitions are defined with respect to all possible moves available from the NFA.

3. DFA final states are those which include an NFA final state.

Chomsky Hierarchy

1. Type 3 - Regular Languages (Finite Automata, simple patterns)

2. Type 2 - Context-Free (Pushdown Automata, programming languages)

3. Type 1 - Context-Sensitive (more complex grammars)

4. Type 0 - Recursively Enumerable (Turing Machines, most powerful)

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.

Implementation description:

This initializes the grammar by defining **non-terminals** (`VN`), **terminals** (`VT`), **production rules** (`P`), and the **start symbol**(`S`). The production rules specify how non-terminals can be replaced by sequences of terminals and/or other non-terminals.

```
def __init__(self):
    self.VN = {"S", "A", "B", "C"}  # Non-terminals
    self.VT = {"a", "b"}  # Terminals
    self.P = {
        "S": ["aA"],
        "A": ["bS", "aB"],
        "B": ["bC"],
        "C": ["aA", "b"]
    }
    self.start_symbol = "S"
```

This function generates a **random string** following the grammar rules. It starts with the `start_symbol` and iteratively replaces non-terminals using random production rules until only terminals remain or the length limit (`max_length`) is reached. If non-terminals still exist in the string, it recursively regenerates to ensure a valid output.

```
def generate_string(self, max_length=10):
    """Generate a random valid string based on the grammar rules."""
    word = self.start_symbol
    while any(symbol in self.VN for symbol in word) and len(word) < max_length:
        for i, symbol in enumerate(word):
            if symbol in self.VN:
                replacement = random.choice(self.P[symbol])  # Choose a random rule
                word = word[:i] + replacement + word[i+1:]  # Replace non-terminal
                break  # Apply one rule per iteration
    return word if all(s not in self.VN for s in word) else self.generate_string(max_length)
```

This function generates a specified number (`count`) of **unique valid strings** using `generate_string()`. It ensures uniqueness by storing generated strings in a `set` and continues generating until the desired number is reached, returning the results as a list.

```
def generate_strings(self, count=5):
    """Generate multiple unique valid strings from the grammar."""
    unique_strings = set()
    while len(unique_strings) < count:
        new_string = self.generate_string()
        unique_strings.add(new_string)
    return list(unique_strings)
```

This function constructs a **Finite Automaton (FA)** from the given grammar. It initializes states, terminals, and transitions, identifying **final states** (where productions contain only terminals). It also builds a transition table mapping non-terminals to their possible next states based on grammar rules.

```
def __init__(self, grammar):
    self.states = grammar.VN
    self.alphabet = grammar.VT
    self.start_state = grammar.start_symbol
    self.transitions = {}
    self.final_states = set()

    # Construct transitions and final states based on the grammar
    for non_terminal, productions in grammar.P.items():
        for production in productions:
            if all(symbol in self.alphabet for symbol in production):
                self.final_states.add(non_terminal)  # If production consists only of terminals,
            else:
                symbol, next_state = production[0], production[1:]  # Extract transition details
                self.transitions.setdefault((non_terminal, symbol), []).append(next_state)
```

This function checks whether an input string is **accepted** by the FA. It starts from the **initial state** and follows valid transitions symbol by symbol. If at least one final state is reached at the end, the string is accepted; otherwise, it is rejected.

```
def accepts(self, input_string):
    """Check if the input string is accepted by the FA."""
    current_states = {self.start_state}  # Start from initial state
    for symbol in input_string:
        next_states = set()
        for state in current_states:
            if (state, symbol) in self.transitions:
                next_states.update(self.transitions[(state, symbol)])  # Move to next states
        if not next_states:
            return False  # If no valid transitions, reject string
        current_states = next_states  # Update current states
    return any(state in self.final_states for state in current_states)  # Check if any final state
```

Conclusion:

This program defines a **formal grammar** and a corresponding **finite automaton (FA)** to generate and validate strings based on specific rules. The Grammar class creates random valid strings using production rules, ensuring they consist only of terminals. The FiniteAutomaton class constructs a state machine from the grammar and checks whether a given string is valid according to its transitions. Together, these components demonstrate the relationship between **context-free grammars** and **finite automata**, illustrating how a structured set of rules can define and recognize a language.

References:

1. Else Course FAF.LFA21.1
2. Finite Automata https://www.geeksforgeeks.org/finite-automata-algorithm-for-pattern-searching/

   https://stackoverflow.com/questions/35272592/how-are-finite-automata-implemented-in-code