Grammar to Finite Automaton Conversion and String Generation

Parfene Daniel

FAF-222

1 Theory

In formal language theory, there is a close relationship between grammars and automata. A grammar defines a language by generating strings, while an automaton recognizes or accepts strings from a language. The conversion from a grammar to a finite automaton enables us to verify whether a given string belongs to the language defined by the grammar.

2 Objectives

- To implement a Python program that converts a context-free grammar to a finite automaton.
- To generate valid strings based on the grammar.
- To check if input strings are accepted by the generated finite automaton.

3 Implementation Description

3.1 Grammar Class

Defines a context-free grammar with non-terminal symbols (VN), terminal symbols (VT), and production rules (P).

```
class Grammar:
    def __init__(self):
        self.VN = {'S', 'A', 'B', 'C'}
        self.VT = {'a', 'b', 'c', 'd'}
        self.P = {
            'S': ['dA'],
            'A': ['d', 'aB'],
            'B': ['bC'],
            'C': ['cA', 'aS']
        }
```

3.2 FiniteAutomaton Class

Represents a finite automaton with states, alphabet, transitions, initial state, and final states. It provides methods to convert from a grammar and check strings. The FiniteAutomaton class now accepts a grammar as an optional argument in its constructor. If a grammar is provided, it automatically converts it to an automaton.

```
class FiniteAutomaton:
    def __init__(self, grammar=None):

        self.states = set()
        self.alphabet = set()
        self.transitions = {}
        self.initial_state = None
```

```
        self.final_states = set()

        # If a grammar is provided, convert it to an automaton
        if grammar:
            self.convert_from_grammar(grammar)

    def convert_from_grammar(self, grammar):

        self.states = grammar.VN
        self.alphabet = grammar.VT

        # Iterate through grammar productions
        for symbol in grammar.P:
            for production in grammar.P[symbol]:
                # If production length is 1, it is a final state transition
                if len(production) == 1:
                    self.transitions[(symbol, production)] = 'final'
                else:
                    # Otherwise, store the transition symbol
                    self.transitions[(symbol, production[0])] = production[1]

        self.initial_state = 'S'
        self.final_states = {symbol for symbol in grammar.P if symbol.isupper()}

    def check_string(self, input_string):

        current_state = self.initial_state

        # Iterate through characters in the input string
        for char in input_string:
            # Check if the current state and input character combination exists in transitions
            if (current_state, char) in self.transitions:
                # Update current state to the next state based on the transition
                current_state = self.transitions[(current_state, char)]
            else:

                return False

        return True
```

3.3 Main Class

Contains methods to generate valid strings based on the grammar and to run the program.

```
import random
from Grammar import Grammar
from FiniteAutomaton import FiniteAutomaton

class Main:
    @staticmethod
    def generate_valid_strings(grammar, num_strings):

        valid_strings_with_transitions = []

        for _ in range(num_strings):
            string = ''
            transitions = [('S', 'S')]
            stack = ['S']

            # Depth-first traversal to generate strings based on grammar productions
            while stack:
                current_symbol = stack.pop()

                # If the current symbol is a terminal, add it to the string
                if current_symbol in grammar.VT:
                    string += current_symbol
```

```
                else:
                    # If the current symbol is non-terminal, select a random production and expand the stack
                    production = random.choice(grammar.P[current_symbol])
                    stack.extend(reversed(production))  % Push the production onto the stack
                    transitions.append((current_symbol, production))  % Record the transition

            # Append generated string and transitions to the list
            valid_strings_with_transitions.append((string, transitions))
        return valid_strings_with_transitions

    @staticmethod
    def run():

        grammar = Grammar()
        finite_automaton = FiniteAutomaton(grammar)

        print("Generated strings:")
        valid_strings_with_transitions = Main.generate_valid_strings(grammar, 5)
        for i, (string, transitions) in enumerate(valid_strings_with_transitions, start=1):
            print(f"{i}.", end=' ')
            for j, transition in enumerate(transitions):
                if j == 0:
                    print(f"{transition[0]} -> {transition[1]}", end=' ')
                else:
                    print(f"-> {transition[1]}", end=' ')
            print(f"-> {string}")

        input_strings = ["ddc", "dabadd", "dd", "dcab", "dcad"]
        print("\nChecking if input strings are accepted by the Finite Automaton:")
        for string in input_strings:

            if finite_automaton.check_string(string):
                print(f"'{string}' is accepted by the Finite Automaton.")
            else:
                print(f"'{string}' is not accepted by the Finite Automaton.")

if __name__ == "__main__":
    Main.run()
```

4 Program Execution

The `run()` method in the `Main` class initializes a grammar, converts it to a finite automaton, generates valid strings, and checks input strings against the automaton.

5 Conclusions / Screenshots / Results

The program successfully converts the grammar to a finite automaton and demonstrates the recognition of valid strings by the automaton. Here are three examples of generated strings and the validation of the strings by the automaton:

```
Generated strings:
1. S -> dA -> d -> dd
2. S -> dA -> aB -> bC -> aS -> dA -> aB -> bC -> cA -> d -> dabcabcd
3. S -> dA -> d -> dd
4. S -> dA -> aB -> bC -> cA -> d -> dabcd
5. S -> dA -> aB -> bC -> cA -> d -> dabcd

Checking if input strings are accepted by the Finite Automaton:
'ddc' is not accepted by the Finite Automaton.
'dabadd' is accepted by the Finite Automaton.
'dd' is accepted by the Finite Automaton.
'dcab' is not accepted by the Finite Automaton.
```

```
'dcad' is not accepted by the Finite Automaton.
```

```
Generated strings:
1. S -> dA -> d -> dd
2. S -> dA -> aB -> bC -> cA -> aB -> bC -> aS -> dA -> aB -> bC -> aS -> dA -> d ->
    dabcabadabadd
3. S -> dA -> d -> dd
4. S -> dA -> d -> dd
5. S -> dA -> aB -> bC -> cA -> aB -> bC -> aS -> dA -> aB -> bC -> aS -> dA -> aB -> bC ->
    cA -> d -> dabcabcabcd

Checking if input strings are accepted by the Finite Automaton:
'ddc' is not accepted by the Finite Automaton.
'dabadds' is not accepted by the Finite Automaton.
'dabcabcabcd' is accepted by the Finite Automaton.
'dabadabcabadabadabcd' is accepted by the Finite Automaton.
'dcadccc' is not accepted by the Finite Automaton.
```

```
Generated strings:
1. S -> dA -> aB -> bC -> cA -> aB -> bC -> aS -> dA -> d -> dabcabadd
2. S -> dA -> d -> dd
3. S -> dA -> aB -> bC -> cA -> d -> dabcd
4. S -> dA -> aB -> bC -> aS -> dA -> d -> dabadd
5. S -> dA -> aB -> bC -> aS -> dA -> aB -> bC -> cA -> aB -> bC -> cA -> d -> dabadabcabcd

Checking if input strings are accepted by the Finite Automaton:
'dabcd' is accepted by the Finite Automaton.
'ddaaa' is not accepted by the Finite Automaton.
'dabcabcabcd' is accepted by the Finite Automaton.
'dabadabcabadd' is accepted by the Finite Automaton.
'dcadca' is not accepted by the Finite Automaton.
```

6 References

1. Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2006). Introduction to Automata Theory, Languages, and Computation (3rd ed.). Pearson Education.