Intro to formal languages. Regular grammars. Finite Automata.

Course: Formal Languages & Finite Automata

Author: Polisciuc Vlad (FAF-223)

Theory

In the realm of formal language theory, regular grammars and finite automata serve as fundamental constructs for modeling and analyzing the properties of languages. Regular grammars, are characterized by a set of production rules typically in the form of productions of the type A -> aB or A -> a, where A and B are non-terminals, and a is a string of terminals. These grammars generate languages that can be recognized by deterministic finite automata (DFAs) or nondeterministic finite automata (NFAs). Finite automata, in turn, are abstract machines consisting of a finite set of states, transitions between states governed by input symbols, an initial state, and one or more accepting states. A transition between the 2 can be easily made and reverted if needed, these changes do not affect the grammar of the language

Objectives:

For the Grammar:

```
 Variant 20:
VN={S, A, B, C},
VT={a, b, c, d},
P={
    S → dA
    A → d
    A → aB
    B → bC
    C → cA
    C → aS
}
```

a. Implement a type/class for this grammar;

b. Add one function that would generate 5 valid strings from the language expressed by this given grammar;

c. Implement some functionality that would convert an object of type Grammar to one of type Finite Automaton;

d. For the Finite Automaton, add a method that checks if an input string can be obtained via the state transition from it;

Implementation description

I have created the following class to denote the language formed from the given grammar. it contains terminal, and nonterminal symbols as well as the rules through which words can be formed.

```

class Language():
    rules = {}
    words = []
    VN=[]
    VT=[]
    def __init__(self,rules,VN,VT):
        self.rules=rules
        self.VN=VN
        self.VT=VT
```

In order to keep a clean class structure I have created a function that given a grammar input(Ex below), would extract all of the symbols and the necessary rules.

```
 import collections

def grammar_to_language(grammar):
    lines=grammar.split("\n")
    VN = lines[1].split("=")[1].translate({ord(c): None for c in "{} "}).split(",")[:-1]
    VT = lines[2].split("=")[1].translate({ord(c): None for c in "{} "}).split(",")[:-1]
    grammar_rules = lines[3:]
    rules = collections.defaultdict(list)

    for rule in grammar_rules:
        if rule and "{" not in rule and "}" not in rule:
            lhs, rhs = rule.split("→")
            rules[lhs.strip()].append(rhs.strip())
    print(rules)
    return rules,VN,VT
```

Yes the use of the collections library is not necessary, I just like the defaultdict data structure

Following the instantiation of the language class I have implemented a method for word generation according to the given rules with S as the starting symbol:

```
    def generate_word(self, number):
        start = self.rules["S"][0]
        steps = [f"\nS -> {start}"]
        unique=0
        while any(c.isupper() for c in start):
            non_terminals = [i for i, c in enumerate(start) if c.isupper()]
            for i in non_terminals[::-1]:
                options = self.rules[start[i]]
                replacement = np.random.choice(options)
                start = start[:i] + replacement + start[i + 1 :]
                steps.append(f" -> {start}")

        if start not in self.words:
            self.words.append(start)
            unique=1
            print("\n".join(steps))

        if number > 0:
            self.generate_word(number - unique)
```

This makes use of the np.random.choice function that chooses a random element inside a list, this makes sure that we use all the rules without special handling of the recursive situations, as the probability of getting an infinite recursion with this approach is literally 0. if a word has been formed for the first time using this language it is appended to the words attribute of the language object, untill a certain number of unique words has been created, according to the parameter 'number'

This is an example use case:

```
 grammar="""Variant 20:
VN={S, A, B, C},
VT={a, b, c, d},
P={
    S → dA
    A → d
    A → aB
    B → bC
    C → cA
    C → aS
}

"""
rules, VN, VT = grammarHelper.grammar_to_language(grammar)



lang=Language(rules,VN,VT)
lang.generate_word(5)
print(lang.words)
print(lang.VN)
print(lang.VT)
```

and this is the output:

```
S -> dA
 -> daB
 -> dabC
 -> dabcA
 -> dabcaB
 -> dabcabC
 -> dabcabcA
 -> dabcabcaB
 -> dabcabcabC
 -> dabcabcabcA
 -> dabcabcabcd

S -> dA
 -> dd

S -> dA
 -> daB
 -> dabC
 -> dabcA
 -> dabcaB
 -> dabcabC
 -> dabcabaS
 -> dabcabadA
 -> dabcabadd

S -> dA
 -> daB
 -> dabC
 -> dabaS
 -> dabadA
 -> dabadaB
 -> dabadabC
 -> dabadabaS
 -> dabadabadA
 -> dabadabadd

S -> dA
 -> daB
 -> dabC
 -> dabcA
 -> dabcaB
 -> dabcabC
 -> dabcabcA
 -> dabcabcd

S -> dA
 -> daB
 -> dabC
 -> dabaS
 -> dabadA
 -> dabadaB
 -> dabadabC
 -> dabadabcA
 -> dabadabcaB
 -> dabadabcabC
 -> dabadabcabaS
 -> dabadabcabadA
 -> dabadabcabadd
['dabcabcabcd', 'dd', 'dabcabadd', 'dabadabadd', 'dabcabcd', 'dabadabcabadd']
['S', 'A', 'B', 'C']
['a', 'b', 'c', 'd']
```

At the end we can see the full list of the generated words, while above it we can take a look at the way the respective words have been formed.

Following that I have created a finite automata class:

```
import networkx as nx
import matplotlib.pyplot as plt

from collections import defaultdict

class FiniteAutomaton:
    def __init__(self, grammar):
        self.grammar = grammar
        self.transitions = defaultdict(list)
        self.start_symbol = 'S'
        self._build_transitions()

    def _build_transitions(self):
        for non_terminal, productions in self.grammar.items():
            for production in productions:
                if len(production) > 1:
                    self.transitions[non_terminal].append((production[0], production[1:]))
                else:
                    self.transitions[non_terminal].append((production,))
```

Here, based on the previous grammar helper method that parses the given grammar string into a dictionary of rules a finite automata class has been defined. Following that we have the function can_generate that using the can_reach helper method, checks whether or not a given input word can be generated via the finite automatam

```
    def _can_reach(self, state, input_string):
        if not input_string:
            return True
        for transition in self.transitions[state]:
            if transition[0] == input_string[0]:
                if self._can_reach(transition[-1], input_string[1:]):
                    return True
        return False

    def can_generate(self, input_string):
        return self._can_reach(self.start_symbol, input_string)
```

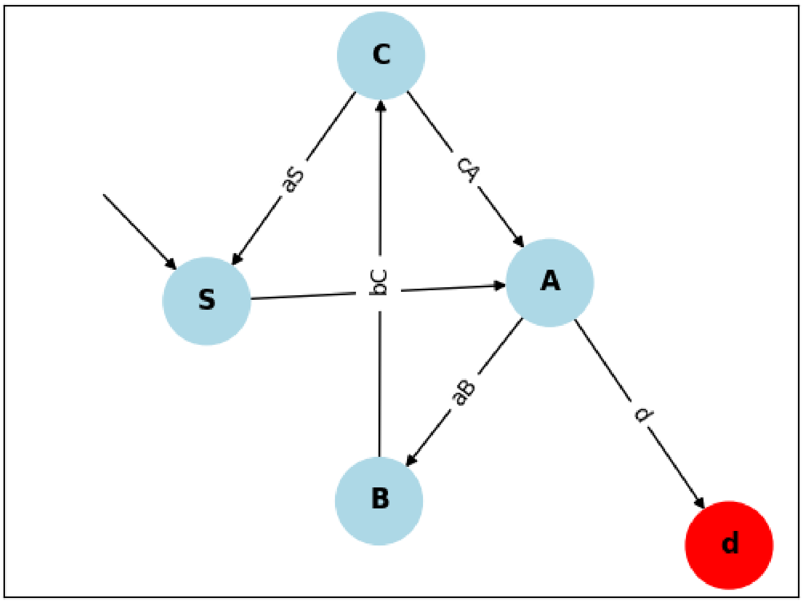
Next we have the get_automata function that generates a graph using networkx for the given finiteAutomata. the red vertexes represent terminal states, and the purple ones are Non terminal. The respective states are connected to each other via transitions

```
def getAutomata(self):

        rules=self.grammar
        G = nx.DiGraph()
        n=0
        used_keys={}
        node_colors=[]



        for node, edges in rules.items():
            if node not in used_keys:
                used_keys[node] = n
                n +=  1
                node_colors.append("lightblue")

            for edge in edges:
                uppercase = [x for x in edge if x.isupper()]

                if uppercase:
                    for char in uppercase:
                        if char not in used_keys:
                            used_keys[char] = n
                            n +=  1
                            node_colors.append("lightblue")


                        if G.has_edge(f'{char}',f'{node}'):
                            existing = G.get_edge_data(f'{char}',f'{node}')
                            G.add_weighted_edges_from([(f'{node}', f'{char}',f"{existing['weight']} \n{edge}")])
                        else:
                            G.add_edge(f'{node}', f'{char}',weight=edge)


                else:
                    if edge not in used_keys:
                        used_keys[edge] = n
                        n +=  1
                    G.add_edge(f'{node}', f'{edge}',weight=edge)
                    node_colors.append("red")


        G.add_edge(" ","S")

        node_colors.append("white")

        pos = {" ": (-10, 4), "S": (-1, 0)}

        spring_pos = nx.spring_layout(G, pos=pos, k=10,fixed=[" ","S"],weight="weights")

        pos.update(spring_pos)

        for node in pos:
            pos[node] = [pos[node][0] * 2, pos[node][1] * 2]

        weights = nx.get_edge_attributes(G, 'weight')
        nx.draw_networkx_nodes(G, pos, node_color=node_colors, node_size=1500)
        nx.draw_networkx_edges(G, pos, node_size=1500)
        nx.draw_networkx_labels(G, pos, font_weight='bold', font_size=12)
        nx.draw_networkx_edge_labels(G, pos, edge_labels=weights)
        plt.show()
```

This is an example usage:

```
 grammar="""Variant 20:
VN={S, A, B, C},
VT={a, b, c, d},
P={
    S → dA
    A → d
    A → aB
    B → bC
    C → cA
    C → aS
}

"""
rules, VN, VT = grammarHelper.grammar_to_language(grammar)

fa = FiniteAutomaton(rules)



input_string = "ab"
if fa.can_generate(input_string):
    print(f"The string '{input_string}' can be obtained from the grammar.")
else:
    print(f"The string '{input_string}' cannot be obtained from the grammar.")

fa.getAutomata()
```

Running this code allows us to check all of the functionalities of the class: checking if a given string can be reached via the automata and the graphic representation of the automata:



Conclusions

In the end we have managed to generate a language and a finite automata based on a given grammar, the language can create different words based on the grammar, which has been confirmed via the finita automata checking. Furthermore we have created a visual representation of he automaton to help illustrate the result.

References

Documentation of networkx