Determinism in Finite Automata. Conversion from NDFA 2 DFA. Chomsky Hierarchy.

Course: Formal Languages & Finite Automata

Author: Popov Nichita (Variant 22)

Theory

In the fascinating universe of computer science, determinism within finite automata is akin to navigating a complex maze with a clear, predefined path. Imagine a world where every decision point in the maze has a signpost, directing you exactly where to go next. This is the essence of deterministic finite automata (DFA), where every input leads to a singular, unambiguous outcome. However, life and computation are not always so straightforward, introducing us to nondeterministic finite automata (NDFA), where at certain crossroads, you're presented with multiple paths but no sign telling you which one to take. Converting from NDFA to DFA is like drawing a new map of the maze, one where every possible journey is considered and planned for, ensuring that no matter where you start, there's always a clear path to the end. This concept nestles within the Chomsky Hierarchy, a grand classification of grammatical complexity in languages, ranging from simple to complex. Finite automata sit at the lower levels of this hierarchy, handling languages of simpler structures, yet they are fundamental in understanding the building blocks of computational linguistics and the theoretical underpinnings of how machines interpret and process languages.

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 project focuses on the manipulation and understanding of automata, grammars, and the conversion between finite automata (FA) and grammars. It includes functionality for classifying grammars based on the Chomsky hierarchy, converting finite automata to regular grammars, determining the determinism of a finite

automaton, and converting a non-deterministic finite automaton (NDFA) to a deterministic finite automaton (DFA).

- `Grammar` Class: The core difference between class from this laboratory work and previous in method called `chomsky_check` that can identify the type of grammar passed based on the Chomsky hierarchy. Also now user can pass the grammar that he wants due to the `set_grammar` method.

`set_grammar`:

```
    def set_grammar(self, VN, VT, P):
        self.VN = VN
        self.VT = VT
        self.P = P
```

`chomsky_check`:

```
    def chomsky_check(self):
        is_type_3 = True
        is_type_2 = True

        for left, rights in self.P.items():
            # Type 2 checks: Left side must be a single non-terminal.
            if len(left) != 1 or left not in self.VN:
                is_type_2 = False

            for right in rights:
                if len(right) > 2 or (len(right) == 2 and not(right[0] in self.VT and
     right[1] in self.VN)):
                    is_type_3 = False
                if len(right) == 1 and right not in self.VT:
                    is_type_3 = False
                if right == '':
                    is_type_3 = False

        if is_type_3:
            return "Type 3 (Regular)"
        elif is_type_2:
            return "Type 2 (Context-Free)"
        else:
            return "Type 1 (Context-Sensitive)"
```

- `Automata` Class: This class manages the definition and manipulation of finite automata, including conversion to regular grammars (`from_automaton_to_grammar`), setting your own automaton (`set_automaton`), determinism checks(`is_dfa`) and NDFA to DFA conversion(`conversion_ndfa_to_dfa`). Also this class contains method that can visualize the obtained DFA (`draw_dfa`) from the last method mentioned.

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

class FiniteAutomaton:
    def __init__(self):
        self.Q = ['q0','q1','q2']
        self.E = ['a','b']
        self.F = ['q2']
        self.sigma = {
            'q0': [['a','q0'],['b','q1']],
            'q1': [['b','q1'],['b','q2'],['a','q0']],
            'q2': [['b','q1']]
```

```
        }
```

from_automaton_to_grammar:

```
    def from_automaton_to_grammar(self):
            self.VN = ['A'+str(i) for i in range(len(self.Q))]
            self.VT = self.E.copy()
            productions = []
            for state, transitions in self.sigma.items():
                index = self.Q.index(state)
                for transition in transitions:
                    if len(transition) == 2:
                        letter, final_state = transition
                        productions.append(f"A{index} ->
        {letter}A{self.Q.index(final_state)}")

            for accept_state in self.F:
                productions.append(f"A{accept_state[1]} -> ε")

            return f"Vn = {self.VN}\nVt = {self.VT} \n" + "\n".join(productions)
```

set_automaton:

```
    def set_automaton(self, Q, E, F, sigma):
        self.Q = Q
        self.E = E
        self.F = F
        self.sigma = sigma
```

is_dfa:

```
    def is_dfa(self):
        for state in self.Q:
            transitions_for_state = self.sigma.get(state, [])
            symbols_seen = set()

            for transition in transitions_for_state:
                symbol = transition[0]

                if symbol == "ε" or symbol in symbols_seen:
                    return "NDFA"
                symbols_seen.add(symbol)
        return "DFA"
```

conversion_ndfa_to_dfa:

```
    def conversion_ndfa_to_dfa(self):
            if self.is_dfa() == "DFA":
                return "No need to convert. FA is already a DFA"

            dfa_states = [frozenset([self.Q[0]])]
            dfa_transitions = {}
            dfa_final_states = set()

            while len(dfa_states) > len(dfa_transitions):
                for state_set in dfa_states:
                    if state_set not in dfa_transitions:
                        dfa_transitions[state_set] = {}
                        for symbol in self.E:
                            next_state = frozenset(sum([self._get_next_states(s, symbol)
        for s in state_set], []))
```

```
                        if next_state not in dfa_states:
                            dfa_states.append(next_state)
                        dfa_transitions[state_set][symbol] = next_state

                        if not dfa_final_states.intersection(state_set) and any(s in
    self.F for s in state_set):
                            dfa_final_states.add(state_set)

            # Convert frozensets to more readable state names
            state_names = {state: f"D{index}" for index, state in enumerate(dfa_states)}
            dfa_final_states_names = [state_names[state] for state in dfa_final_states]

            # Convert transitions to use the new state names
            dfa_transitions_named = {state_names[state]: {symbol: state_names[next_state]
        for symbol, next_state in transitions.items()} for state, transitions in
        dfa_transitions.items()}
            self.draw_dfa(dfa_transitions_named, dfa_final_states_names)
            return f"Transitions in the form [State: [transition : state]]:
        {dfa_transitions_named}\nFinal state : {dfa_final_states_names}"

    def _get_next_states(self, current_state, symbol):
        return [transition[1] for transition in self.sigma.get(current_state, []) if
        transition[0] == symbol]

draw_dfa:


    def draw_dfa(self, dfa_transitions_named, dfa_final_states_names):
        G = nx.DiGraph()
        # Add nodes with their labels
        for state in dfa_transitions_named:
            G.add_node(state)

        # Add edges with their labels
        for state, transitions in dfa_transitions_named.items():
            for symbol, next_state in transitions.items():
                G.add_edge(state, next_state, label=symbol)

        pos = nx.circular_layout(G)  # Using circular layout for clearer structure

        # Draw the graph
        nx.draw_networkx_nodes(G, pos, node_size=700, node_color='skyblue',
        edgecolors='black')
        nx.draw_networkx_labels(G, pos)
        edges = nx.draw_networkx_edges(G, pos, arrowstyle='->', arrowsize=20,
        connectionstyle='arc3,rad=0.2')

        # Handle self-loops after drawing other edges to avoid overlap issues
        for state, transitions in dfa_transitions_named.items():
            for symbol, next_state in transitions.items():
                if state == next_state:  # Identify self-loops
                    loop_pos = np.array(pos[state])
                    # Drawing self-loop with an arc away from the node
                    plt.annotate("", xy=loop_pos, xycoords='data',
                                 xytext=loop_pos + np.array([0, 0.4]), textcoords='data',
                                 arrowprops=dict(arrowstyle="->", color="red",
                                                 shrinkA=15, shrinkB=15,
                                                 patchA=None, patchB=None,
                                                 connectionstyle="arc3,rad=0.3"))
                    # Placing text for self-loop
                    plt.text(loop_pos[0], loop_pos[1] + 0.5, symbol, ha='center',
        color="red")

        # Highlight final states
```

```
        nx.draw_networkx_nodes(G, pos, nodelist=dfa_final_states_names,
        node_color='lightgreen', edgecolors='black')

        # Edge labels, avoiding self-loop edges since they're handled separately
        edge_labels = {(u, v): d['label'] for u, v, d in G.edges(data=True) if u != v}
        nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, label_pos=0.3)

        plt.title('DFA Visualization')
        plt.axis('off')
        plt.show()
```

- `Menu` Class: This class provides comprehensive solution to navigation in functionality of the program

```
from classes.Automata import Automata
from classes.Grammar import Grammar
from classes.FiniteAutomaton import FiniteAutomaton

aut = Automata()
gr = Grammar()
fa = FiniteAutomaton()

class Menu:
    def __init__(self):
        pass

    def main_menu(self):
        self.user_choice = int(input("""Please choose the option:
            1) Generate strings with progression
            2) Check if string can be obtained
            3) Check the type of Grammar
            4) FA choice menu
            5) Exit\n>>>"""))

        match self.user_choice:
            case 1:
                aut.generating_strings()
            case 2:
                user_input = input("Insert the string: ")
                aut.check_string(user_input)
            case 3:
                self.grammar_choice()
            case 4:
                self.fa_determination_menu()

            case 5:
                exit()

    def grammar_choice(self):
        gm_choice = int(input("""
            1) Check current grammar
            2) Check grammar inserted by user
            >>>"""))
        match gm_choice:
            case 1:
                print(gr.chomsky_check())
            case 2:
                VN = input("Insert Non-terminal symbols separated by
        commas:\n").split(',')
                VT = input("Insert Terminal symbols separated by commas:\n").split(',')
                P = {}
                for _ in VN:
                    transitions = input(f"Please insert all the transitions for {_}
        separated by commas:\n").split(',')
```

```
                P[_] = transitions
            gr.set_grammar(VN, VT, P)
            print(gr.chomsky_check())

    def fa_determination_menu(self):
        self.fa_choice = int(input("""
        What FA use:
        1) Set the new FA
        2) Use already existing
        """))

        match self.fa_choice:
            case 1:
                Q = input("Insert states separated by commas:\n").split(",")
                E = input("Insert alphabet separated by commas:\n").split(",")
                F = input("Insert accept states separated by commas:\n").split(",")
                sigma={}
                for state in Q:
                    transitions_input = input(f"Insert all the transitions and states to
        which they lead for {state} in the format '[a,q0],[b,q2]':\n")
                    if transitions_input.lower() == 'none':
                        sigma[state] = []
                    else:
                        transitions_list = transitions_input.strip("[]").split("],[")
                        transitions = [transition.split(",") for transition in
        transitions_list]
                        sigma[state] = transitions
                sigma = {key: value for key, value in sigma.items() if len(value) > 0}
                fa.set_automaton(Q, E, F, sigma)
                self.fa_menu_choice()
            case 2:
                self.fa_menu_choice()

    def fa_menu_choice(self):
        user_choice = int(input("""Please choose the option:
1) Convert the FA to Grammar
2) Determine the type of FA
3) Convert NDFA to DFA\n>>>"""))

        match user_choice:
            case 1:
                print(fa.from_automaton_to_grammar())
            case 2:
                print(fa.is_dfa())
            case 3:
                print(fa.conversion_ndfa_to_dfa())
```

Results

As a result we get the following: * Conversion from NDFA to DFA (Example of output):

```
    Transitions in the form [State: [transition : state]]:{'D0': {'a': 'D0', 'b': 'D1'},
'D1': {'a': 'D0', 'b': 'D2'}, 'D2': {'a': 'D0', 'b': 'D2'}}
    Final state : ['D2']
```

- Conversion from Automaton to grammar: `Vn = ['A0', 'A1', 'A2']  Vt = ['a', 'b']  A0 -> aA0  A0 -> bA1  A1 -> bA1  A1 -> bA2  A1 -> aA0  A2 -> bA1  A2 -> ε` ## Conclusion

This laboratory work effectively combines theoretical principles with practical application, exploring finite automata, grammar classification, and the Chomsky hierarchy. Through hands-on tasks, it transitions from the concepts of deterministic and nondeterministic automata to practical demonstrations, including grammar

classification and DFA visualization. The project not only fulfills its educational objectives but also serves as an insightful tool for understanding automata theory and its applications in computational linguistics. It showcases the integral relationship between computer science theory and practical implementation, encouraging further exploration and learning in the field.

References

- https://www.youtube.com/watch?v=pnyXgIXpKnc
- https://www.youtube.com/watch?v=–CSVsFIDng
- https://www.youtube.com/watch?v=ehy0jGIYRtE
- https://www.youtube.com/watch?v=dQw4w9WgXcQ