Laboratory Work Nr.2 Determinism in Finite Automata. Conversion from NDFA to DFA. Chomsky Hierarchy.

Course: Formal Languages & Finite Automata

Author: Cretu Cristian

Group: FAF-223

Theory

A finite automaton is a mechanism used to represent processes of different kinds. It can be compared to a state machine as they both have similar structures and purpose as well. The word finite signifies the fact that an automaton comes with a starting and a set of final states. In other words, for process modeled by an automaton has a beginning and an ending.

Based on the structure of an automaton, there are cases in which with one transition multiple states can be reached which causes non determinism to appear. In general, when talking about systems theory the word determinism characterizes how predictable a system is. If there are random variables involved, the system becomes stochastic or non deterministic.

That being said, the automata can be classified as non-/deterministic, and there is in fact a possibility to reach determinism by following algorithms which modify the structure of the automaton.

Objectives:

1. Automaton Understanding:

- Define automaton and its applications.

2. Grammar Classification:

- Add function to grammar class for Chomsky hierarchy classification.

3. Finite Automaton Tasks:

- Using variant from previous lab:

- Convert finite automaton to regular grammar.
- Determine determinism of automaton.
- Implement NDFA to DFA conversion.
- Optional: Graphical representation of automaton.

5. Bonus:

- Develop program for automating automaton to regular grammar conversion.
- Optional: Graphical representation using external tools/libraries.

Implementation description

Variant Class:

A simple utility class that extracts my variant from the file.

```
class Variant:
    def __init__(self, _variantpath):
        self.variantpath = _variantpath
```

Grammar Class:

This class is the same as in the previous one, but has been added a method that checks which Chomsky grammar type is this grammar.

```
class Grammar:
    def __init__(self, _VN: list, _VT: list, _P: dict) -> None:
        self.nonterminals = _VN
        self.terminals = _VT
        self.productions = _P

    def check_grammar_type(self) -> str:
        """
        Determines the type of the grammar based on its productions.

        Returns:
            str: The type of the grammar. Possible values are:
                - 'Type 3 - Unrestricted' for grammars with unrestricted productions.
                - 'Type 2 - Context-free' for grammars with context-free productions.
                - 'Type 1 - Context-sensitive' for grammars with context-sensitive produc
                - 'Type 0 - Regular' for grammars with regular productions.
        """
        # Check if all productions are in the form A -> a or A -> BC
```

```
        for nonterminal, production in self.productions.items():
            print(nonterminal, production)
            if production == '':
                return 'Type 3 - Unrestricted'
            elif production[0] in self.nonterminals:
                if len(production) > 1:
                    if production[1] in self.terminals:
                        return 'Type 2 - Context-free'
                    elif production[1] in self.nonterminals:
                        return 'Type 1 - Context-sensitive'
        return 'Type 0 - Regular'
```

Finite Automaton Class

This class has been reworked so that it can be used for more functionality and passed to a Transformer class.

```
class FiniteAutomaton:
    def __init__(self, Q=None, Sigma=None, q0=None, F=None, delta=None,create_trans:bool=
        self.Q = Q if Q is not None else []
        self.Sigma = Sigma if Sigma is not None else []
        self.q0 = q0 if q0 is not None else (self.Q[0] if self.Q else None)
        self.F = F if F is not None else []
        self.delta = delta if delta is not None else {}
        if create_trans is True:
            self.transition_table = self.create_transition_table()
        else:
            self.transition_table = delta
        self.is_deterministic = self.is_deterministic()


    def string_belongs_to_language(self, string:str)->bool:
        state = self.q0
        for char in string:
            if char not in self.Sigma:
                return False
            if char in self.delta[state]:
                state = self.delta[state][char]
            else:
                return False
        return state in self.F

    def create_transition_table(self):
        trans_table = {}
        for state in self.Q:
            trans_table[state]={}
            for char in self.Sigma:
```

```
                trans_table[state][char]=[]
        for element in self.delta:
            for state in element:
                for transition in element[state]:
                    resulting_state = element[state][transition]
                    trans_table[state][transition].append(resulting_state)
        return trans_table

    def print_trans_table(self):
        for state in self.transition_table:
            print(state, self.transition_table[state])


    def is_deterministic(self):
        for state in self.transition_table:
            for transition in self.transition_table[state]:
                if len(self.transition_table[state][transition]) > 1:
                    return False
        return True
```

Transformer Class

This class is responsible for transforming a NDFA to DFA, a FA to Grammar, and a Grammar to FA.

```
from FiniteAutomaton import FiniteAutomaton
from Grammar import Grammar
from copy import deepcopy
class Transformer:
    def __init__(self)->None:
        pass


    '''This part transforms the Grammar into a Finite Automation object,
    which can be used to check if a string belongs to the language of the grammar.
    It follows the algorithm presented by Mrs. Cojuhari at the course.'''
    def grammar_to_finite_automation(self, g:Grammar, q0:str, f:str)->FiniteAutomaton:
        Q = g.nonterminals
        Q.append("X")
        Sigma = g.terminals
        q0 = "S"
        F = ["X"]
        delta = {}
        '''The delta function is a dictionary of dictionaries,
        where the first key is the terminal'''
        for terminal in g.productions:
            for production in g.productions[terminal]:
                if len(production) > 1:
                    transition = production[0]
```

```
                    result_state = production[1]
                    if terminal not in delta:
                        delta[terminal] = {}
                    delta[terminal][transition] = result_state
                else:
                    transition = production
                    result_state = "X"
                    if terminal not in delta:
                        delta[terminal] = {}
                    delta[terminal][transition] = result_state
        return FiniteAutomaton(Q, Sigma, q0, F, delta)

    def finite_automaton_to_grammar(self, fa: FiniteAutomaton) -> Grammar:
        """
        Converts a finite automaton to a grammar.

        Args:
            fa (FiniteAutomaton): The finite automaton to be converted.

        Returns:
            Grammar: The resulting grammar.
        """
        nonterminals = fa.Q
        terminals = fa.Sigma
        productions = deepcopy(fa.transition_table)  # Make a copy to avoid modifying the

        # Iterate over the dictionary and remove entries with empty lists
        for state, transitions in list(productions.items()):
            for symbol, resulting_states in list(transitions.items()):
                if not resulting_states:
                    del productions[state][symbol]

        return Grammar(nonterminals, terminals, productions)

    def NFA_to_DFA(self, nfa: FiniteAutomaton) -> FiniteAutomaton:
        """
        Converts a given NFA (Non-Deterministic Finite Automaton) to a DFA (Deterministic

        Args:
            nfa (FiniteAutomaton): The NFA to be converted.

        Returns:
            FiniteAutomaton: The converted DFA.

        Algorithm:
        1. Initialize the DFA states (Q_dfa) with the same states as the NFA.
        2. Initialize the DFA alphabet (Sigma_dfa) with the same alphabet as the NFA.
        3. Initialize the DFA initial state (q0_dfa) with the same initial state as the N
        4. Initialize the DFA final states (F_dfa) with the same final states as the NFA.
```

```
    5. Initialize an empty dictionary to store the DFA transition table (delta_dfa).
    6. Create a queue to store the sets of states that need to be processed.
    7. For each state in Q_dfa and each symbol in Sigma_dfa, find the resulting state
       - If the resulting state is a set of states, add it to the queue and update th
       - If the resulting state is a single state, update the transition table accord
    8. Add the elements in the queue to Q_dfa.
    9. For each state in Q_dfa, create the delta function for each symbol in Sigma_df
       - If the state is a single state, find the resulting state in the NFA and upda
       - If the state is a set of states, find the resulting state by taking the unio
    10. Convert Q_dfa to a list of strings.
    11. Return the converted DFA.

    Note: The algorithm assumes that the NFA has a transition table and follows the s
    """

    Q_dfa = deepcopy(nfa.Q)
    Sigma_dfa = nfa.Sigma
    q0_dfa = nfa.q0
    F_dfa = nfa.F
    delta_dfa = {}

    # Create the queue
    queue = []
    for state in Q_dfa:
        for symbol in Sigma_dfa:
            resulting_state = nfa.transition_table[state][symbol]
            if len(resulting_state) > 1:
                queue.append(set(resulting_state))
                nfa.transition_table[state][symbol] = set(resulting_state)
            nfa.transition_table[state][symbol] = set(resulting_state)

    # Put the queue in the Q_dfa too
    Q_dfa = [set([element]) for element in Q_dfa]
    for element in queue:
        Q_dfa.append(element)

    # Create the delta functions
    for state in Q_dfa:
        if len(state) <= 1:
            temp_state = list(state)[0]
            delta_dfa[temp_state] = {}
            for symbol in Sigma_dfa:
                delta_dfa[temp_state][symbol] = {}
                resulting_state = nfa.transition_table[temp_state][symbol]
                if len(resulting_state) == 0:
                    resulting_state = ''
                else:
                    resulting_state = ''.join(resulting_state)
                delta_dfa[temp_state][symbol] = resulting_state
```

```
            elif len(state) > 1:
                temp_state = ''.join(state)
                delta_dfa[temp_state] = {}
                for symbol in Sigma_dfa:
                    delta_dfa[temp_state][symbol] = {}
                    resulting_state = set()
                    for element in state:
                        resulting_state = resulting_state.union(nfa.transition_table[elem
                    if len(resulting_state) == 0:
                        resulting_state = ''
                    else:
                        resulting_state = ''.join(resulting_state)
                    delta_dfa[temp_state][symbol] = resulting_state

        # Make Q_dfa a list of strings
        Q_dfa = [''.join(element) for element in Q_dfa]

        return FiniteAutomaton(Q_dfa, Sigma_dfa, q0_dfa, F_dfa, delta_dfa, create_trans=F
```

Main File

```
from Transformer import Transformer,FiniteAutomaton,Grammar
from Variant import Variant

variant = Variant('/home/flexksx/GitHub/LabsLFA/NFA to DFA/variant.json').getVariant()
Q=variant['Q']
Sigma=variant['Sigma']
F=variant['F']
delta=variant['delta']



nfa = FiniteAutomaton(Q=Q,Sigma=Sigma,F=F,delta=delta)
transform = Transformer()

g=transform.finite_automaton_to_grammar(nfa)


# nfa.print_trans_table()
dfa = transform.NFA_to_DFA(nfa)
dfa.print_trans_table()
```

Output

```
q0 {'a': 'q1', 'b': ''}
q1 {'a': 'q1', 'b': 'q2'}
q2 {'a': '', 'b': 'q2q3'}
q3 {'a': 'q1', 'b': ''}
q2q3 {'a': 'q1', 'b': 'q2q3'}
```

Also, for Chomsky Grammar type

```
from Grammar import Grammar
from Variant import Variant

variant = Variant('/home/flexksx/GitHub/LabsLFA/Grammar and Finite Automaton/variant.json
terminals = variant.getVT()
nonterminals = variant.getVN()
productions = variant.getP()

grammar = Grammar(nonterminals, terminals, productions)
print(grammar.check_grammar_type())
```

Output

```
S ['aD']
D ['dE', 'bJ', 'aE']
J ['cS']
E ['e', 'aE']
Type 0 - Regular
```