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

Author: Plămădeală Maxim

Group: FAF-222

Theory:

Overview

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. 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.

Please consider that all elements of the task 3 can be done manually, writing a detailed report about how you've done the conversion and what changes have you introduced. In case if you'll be able to write a complete program that will take some finite automata and then convert it to the regular grammar - this will be **a good bonus point**.

Implementation

Part 1: Grammar Classification

In the code, the `Grammar` class plays the role of a grammar detective, trying to figure out the "personality" of different grammars. Let's break down how it does that:

1.1 Chomsky Hierarchy:

1. **Type 0 - Unrestricted Grammars:**
   - These are like the rebels of grammars with no rules to follow.
   - The code checks for productions that break the norm, like longer left-hand sides or multiple nonterminals on the left.
2. **Type 1 - Context-Sensitive Grammars:**
   - Grammars that depend on the context; the left side can be replaced based on what comes before and after.
   - The code ensures that all productions adhere to a context-sensitive vibe with `len(lhs) <= len(rhs)`.
3. **Type 2 - Context-Free Grammars:**
   - The cool kids of grammars, where production rules don't care about the context.
   - Each production is checked to have a single nonterminal on the left side.
4. **Type 3 - Regular Grammars:**
   - These are the linear thinkers, either leaning to the right or to the left.
   - The code checks for right and left linearity in productions.

In the following code I check every use case for each grammar and classify them:

```
def classify_grammar(self):
        for lhs, rhs_list in self.productions.items():
            for rhs in rhs_list:
                if len(lhs) > len(rhs) or (len(lhs) > 1 and any(char in se
                    return "UNRESTRICTED"

                if len(lhs) != 1 or lhs not in self.nonterminals:
                    return "CONTEXT_SENSITIVE"
```

```
        is_right_linear = any(all(symbol in self.terminals for symbol in rl
        is_left_linear = any(rhs[0] in self.nonterminals and all(symbol in

        if is_right_linear and not is_left_linear:
            return "REGULAR_RIGHT_LINEAR"
        elif is_left_linear and not is_right_linear:
            return "REGULAR_LEFT_LINEAR"

        return "CONTEXT_FREE"
```

Part 2: Finite Automaton Operations

2.1 Conversion to Regular Grammar

2.1.1 The Scoop:

- The `convert_to_regular_grammar` function takes an NFA and transforms its transitions into a more grammatical style.
- Epsilon ('ε') is like the wildcard, representing those moments when there's no specific symbol.

```
def convert_to_regular_grammar(fa):
    """
    Converts a finite automaton to a regular grammar.

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

    Returns:
        dict: The regular grammar representation of the finite automaton.
    """
    grammar = {}
    for (state, symbol), next_states in fa.transitions.items():
        if state not in grammar:
            grammar[state] = []
        for next_state in next_states:
            grammar[state].append(f"{symbol}{next_state}")
    for final_state in fa.final_states:
        if final_state in grammar:
            grammar[final_state].append("ε")  # ε represents the empty str
        else:
            grammar[final_state] = ["ε"]
    return grammar
```

2.2 Deterministic Finite Automaton (DFA) Check

2.2.1 In Simple Terms:

- The `is_deterministic` function is like a DFA detective, checking if it's straightforward or a bit of a wildcard.
- If any state has more than one transition for the same symbol, it's a sign that things might get a bit wild.

```
def is_deterministic(fa):
    """
    Check if a finite automaton is deterministic.

    Args:
        fa (FiniteAutomaton): The finite automaton to check.

    Returns:
        bool: True if the finite automaton is deterministic, False otherwi
    """
    for next_states in fa.transitions.values():
        if len(next_states) > 1:
            return False
    return True
```

2.3 Conversion of NDFA to DFA

2.3.1 The Unveiling:

- The `convert_ndfa_to_dfa` function uses some power (set construction) to turn a less decisive NDFA into a more decisive DFA.
- Each DFA state represents a set of states from the NDFA, making decisions as a team.

```
def convert_ndfa_to_dfa(ndfa):
    """
    Converts a non-deterministic finite automaton (NDFA) to a deterministi

    Args:
        ndfa (FiniteAutomaton): The NDFA to be converted.

    Returns:
        FiniteAutomaton: The DFA resulting from the conversion.
    """
    # rest of the code
```

But mainly the algorithm works in three main steps:

1. **Initialization:** Start with a single state representing the NDFA's starting point and mark it for exploration.

```
dfa_states = {frozenset([ndfa.start_state]): '0'}
dfa_transitions = {}
dfa_final_states = []
unmarked_states = [frozenset([ndfa.start_state])]
```

2. **Explore Transitions:**
   - For each unprocessed state and symbol:
     - Find all possible next states in the NDFA based on transitions.
     - If no transitions exist, move on to the next symbol.
   - If new states are found:
     - Create a new state in the DFA with a unique name.
     - Mark the new state for exploration.
     - Record the transition from the current state and symbol to the new state.

```
 while unmarked_states:
# Pop a state from unmarked_states to explore its transitions
current_dfa_state = unmarked_states.pop()
for symbol in ndfa.alphabet:
    # For each symbol in the alphabet, find the next states in the NDF
    next_states = set()
    for ndfa_state in current_dfa_state:
        # Check if there is a transition for the current symbol from e
        if (ndfa_state, symbol) in ndfa.transitions:
            # Update the set of next states with the transitions
            next_states.update(ndfa.transitions[(ndfa_state, symbol)])
    # Skip if there are no next states for the current symbol
    if not next_states:
        continue
    # Create a frozenset of the next states for set comparison
    next_states_frozenset = frozenset(next_states)
    # If the set of next states is new, mark it as unmarked and assign
    if next_states_frozenset not in dfa_states:
        new_state_name = str(len(dfa_states))
        dfa_states[next_states_frozenset] = new_state_name
        unmarked_states.append(next_states_frozenset)
    # Add the transition from the current DFA state to the new DFA sta
    dfa_transitions[(dfa_states[current_dfa_state], symbol)] = dfa_sta
```

3. **Identify Final States:**
   - Check each DFA state:
     - If it contains any of the NDFA's final states, mark the DFA state as final.

```
for dfa_state, label in dfa_states.items():
        if any(state in ndfa.final_states for state in dfa_state):
            dfa_final_states.append(label)

    return FiniteAutomaton(list(dfa_states.values()), ndfa.alphabet, dfa_t
```

This process ensures a complete exploration of all possible paths in the NDFA and creates an equivalent DFA with deterministic transitions.

Part 3: Console results

Checking grammar types:

```
CONTEXT_SENSITIVE
UNRESTRICTED
REGULAR_LEFT_LINEAR
CONTEXT_FREE
REGULAR_RIGHT_LINEAR
REGULAR_RIGHT_LINEAR
```

Last three tasks:

```
Is NFA deterministic?
False

DFA from NFA:

States: ['0', '1', '2', '3']
Alphabet: ['a', 'b', 'c']

Transitions:
It's read as: δ(state, symbol) = next_state
    δ(0, 'a') = ['1']
    δ(1, 'a') = ['1']
    δ(1, 'b') = ['2']
    δ(2, 'a') = ['2']
    δ(2, 'c') = ['3']
    δ(3, 'c') = ['3']
Start State: 0
Final States: ['3']


Regular Grammar from NFA:
```

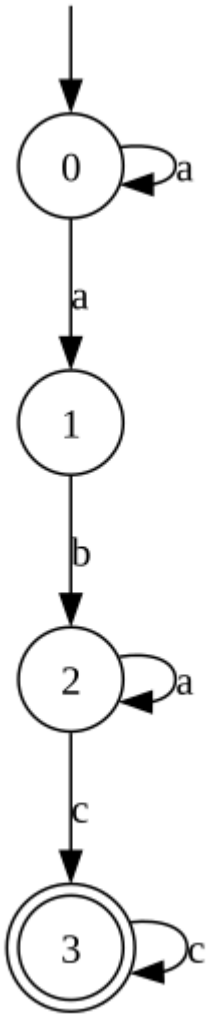
```
0 -> a0
0 -> a1
2 -> a2
2 -> c3
1 -> b2
3 -> c3
3 -> ε
```

Part 4: Visualization

Visualization is done the same way as in the previous LAB:

```python
from graphviz import Digraph

def visualize_finite_automaton(fa, file_name):
    dot = Digraph(comment='Finite Automaton')

    # Add states to the graph
    for state in fa.states:
        if state in fa.final_states:
            dot.attr('node', shape='doublecircle')
        else:
            dot.attr('node', shape='circle')
        dot.node(state)

    # Add transitions to the graph
    for (state, symbol), next_states in fa.transitions.items():
        for next_state in next_states:
            dot.edge(state, next_state, label=symbol)

    # Mark the start state with an edge
    dot.attr('node', shape='none')
    start = 'start'
    dot.node(start, label='')
    dot.edge(start, fa.start_state)

    # Render the graph to a file (e.g., 'finite_automaton.gv')
    dot.render(file_name, view=True)
```

So when it comes to visualizing FA:



Conclusion

In this coding adventure, we explored how grammars get classified and how finite automata undergo transformations. The code provides a practical peek into the language of grammars and automata, laying the groundwork for understanding more complex language structures and recognition algorithms. It's like unraveling the hidden stories behind the symbols and transitions.

References:

- Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2006). Introduction to Automata Theory, Languages, and Computation (3rd ed.). Addison-Wesley.
- Chomsky, N. (1956). Three models for the description of language. IRE Transactions on Information Theory, 2(3), 113–124.