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

Course: Formal Languages & Finite Automata

Author: Daniela Vornic

Theory

Finite automata are abstract machines used to model computational processes, akin to state machines, with a finite number of states. Each automaton starts in a designated initial state and transitions between states based on input symbols, leading to a set of final or accepting states, signaling the completion of a process.

The key distinction between **deterministic finite automata (DFA)** and **nondeterministic finite automata (NDFA)** lies in their transition functions. In a DFA, each state has exactly one transition for each input symbol, ensuring a single predictable path for any given input. Conversely, an NDFA allows for multiple possible transitions for a given symbol, introducing non-determinism, where multiple paths could be taken, potentially leading to multiple possible outcomes.

Determinism in finite automata refers to the predictability of the system's behavior. DFAs, by their nature, are deterministic, as the automaton's state at any point is precisely determined by the input processed up to that point. NDFAs, with their multiple possible transitions for a single input, introduce a level of unpredictability or non-determinism.

The conversion from NDFA to DFA, also known as the **subset construction method**, involves creating DFA states that represent sets of NDFA states. This process ensures that the resulting DFA has no ambiguity in its transitions, thus converting a nondeterministic system into a deterministic one. This conversion can lead to an exponential increase in the number of states, potentially making the DFA much larger than the original NDFA.

The Chomsky hierarchy classifies formal grammars into four levels:

- **Type 0 (Recursively Enumerable)**: The least restricted grammars that can generate any language recognized by a Turing machine.
- **Type 1 (Context-Sensitive)**: Grammars where productions can expand a non-terminal only in a context.
- **Type 2 (Context-Free)**: Grammars that allow productions to replace a non-terminal with a string of terminals and non-terminals.

- **Type 3 (Regular)**: The most restricted grammars, which can be represented by finite automata and generate regular languages.

Regular languages, corresponding to Type 3 grammars in the Chomsky hierarchy, are of particular interest in the context of finite automata. Both DFAs and NDFAs can recognize these languages, making the study of automata integral to understanding the computational capabilities and limitations of regular 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

Chomsky hierarchy grammar classification

This section of the report discusses the implementation of a method named `get_type` within a class that represents a grammar. The purpose of this method is to classify the grammar according to the Chomsky Hierarchy, determining whether it is Unrestricted, Context-Sensitive, Context-Free, Regular Right-Linear, or Regular Left-Linear. The function begins by assuming the grammar is both context-free and regular, which are the least restrictive types within the Chomsky hierarchy.

```
def get_type(self):
    is_context_free = True
    is_regular = True
    is_right_linear = False
    is_left_linear = False
```

The function iterates over each production rule in the grammar. Here, `lhs` represents the left-hand side of a production, and `rhs_list` is a list of possible right-hand sides.

```
for lhs, rhs_list in self.rules.items():
    for rhs in rhs_list:
```

The function first checks if the grammar violates the constraints of a context-sensitive grammar, which would make it unrestricted:

```
if len(lhs) > len(rhs) or (len(lhs) > 1 and any(char in self.non_terminals for char in lh
    return GrammarType.UNRESTRICTED
```

An unrestricted grammar is identified if the length of the left-hand side is greater than the right-hand side or if the left-hand side contains more than one non-terminal symbol.

The function then checks for violations of context-free grammar rules. A context-free grammar must have a single non-terminal on the left-hand side of each production rule.

```
if len(lhs) != 1 or lhs not in self.non_terminals:
    is_context_free = False
    is_regular = False
```

To check if the grammar is regular, the function analyzes the structure of the right-hand side of each production. A grammar is considered regular if it is either right-linear (non-terminal at the end) or left-linear (non-terminal at the beginning).

```
if all(symbol in self.terminals for symbol in rhs):
    continue
elif rhs[-1] in self.non_terminals and all(symbol in self.terminals for symbol in rhs[:-1
    is_right_linear = True
elif rhs[0] in self.non_terminals and all(symbol in self.terminals for symbol in rhs[1:])
    is_left_linear = True
```

```
else:
    is_regular = False
```

Based on the checks above, the function concludes the grammar type:

- If the grammar is regular, it further checks if it is right-linear or left-linear.
- If no violations for a context-free grammar are found, it is classified as context-free.
- If none of the above conditions are met, but it's not unrestricted, it defaults to context-sensitive.

```
if is_regular:
    if is_right_linear and not is_left_linear:
        return GrammarType.REGULAR_RIGHT_LINEAR
    elif is_left_linear and not is_right_linear:
        return GrammarType.REGULAR_LEFT_LINEAR
    else:
        raise ValueError('Grammar is not regular')

if is_context_free:
    return GrammarType.CONTEXT_FREE

return GrammarType.CONTEXT_SENSITIVE
```

Conversion of a finite automaton to a regular grammar

The `to_reg_gr` function is designed to convert a given finite automaton (FA) into an equivalent regular grammar (RG). This conversion is based on the states and transitions of the FA, mapping them to non-terminals and production rules of the RG, respectively.

The function starts by defining the components of the regular grammar:

```
non_terminals = self.states
terminals = self.alphabet
start_symbol = self.initial_state
rules = {nt: [] for nt in non_terminals}
```

- `non_terminals` : Each state in the FA becomes a non-terminal in the RG.
- `terminals` : The alphabet of the FA directly translates to the terminals in the RG.
- `start_symbol` : The initial state of the FA is used as the start symbol in the RG.
- `rules` : A dictionary is initialized to hold the production rules for each non-terminal, initially empty.

The function iterates over each transition in the FA, converting them into production rules for the RG:

```
for state, transitions in self.transitions.items():
    for symbol, next_states in transitions.items():
        for next_state in next_states:
            rules[state].append(f"{symbol}{next_state}")
            # If the next state is an accept state, add a production rule ending in the t
            if next_state in self.accept_states and not self.transitions.get(next_state):
                rules[state].append(symbol)
```

For each transition from a state on a given symbol to one or more next states, a production rule is created in the following manner:

- The current state (acting as a non-terminal) produces a string consisting of the transition symbol followed by the next state (also a non-terminal).
- If the next state is an accepting state and has no outgoing transitions (indicating it's an end state), an additional production rule is added where the current state produces just the transition symbol. This accounts for the acceptance of the input string by terminating the derivation sequence.

It's important to note that this implementation does not handle epsilon (ε) transitions, which are transitions that consume no input. Handling epsilon transitions would require additional steps to ensure that any sequence of epsilon transitions leading to an accepting state is also represented in the regular grammar.

Finally, the function constructs and returns a new Grammar object with the defined non-terminals, terminals, production rules, and start symbol:

```
return Grammar(non_terminals, terminals, rules, start_symbol)
```

Determinism of the finite automaton

The `is_dfa` function is designed to determine whether a given automaton is a DFA or an NFA based on its transition function. This distinction is crucial as it affects the computational behavior and analysis of the automaton.

A DFA is characterized by a single, unique next state for every state-symbol pair in its transition function. Conversely, an NFA may have multiple next states for a given state-symbol pair, introducing nondeterminism.

The function iterates over all the transitions defined in the automaton:

```
for state, transitions in self.transitions.items():
    for symbol, next_states in transitions.items():
```

For each state, it examines the transition rules defined for each input symbol. The `transitions` dictionary contains mappings from symbols to lists of next states (`next_states`). The core logic to determine nondeterminism (and hence, identify an NFA) is:

```
if len(next_states) > 1:
    return False
```

If, for any state-symbol pair, the list of next states (`next_states`) contains more than one state, it implies that the automaton has a nondeterministic transition. Therefore, the function returns `False` , indicating that the automaton is not a DFA.

If the function completes the iteration over all transitions without finding any state-symbol pair leading to multiple next states, it concludes that the automaton is deterministic.

```
return True
```

Conversion of an NDFA to a DFA

The `to_dfa` function is designed to convert a given nondeterministic finite automaton (NDFA) into an equivalent deterministic finite automaton (DFA). This process, known as subset construction, involves creating states in the DFA that correspond to sets of states in the NDFA.

If the automaton is already deterministic, the function returns itself without modification:

```
if self.is_dfa:
    return self
```

Otherwise, the conversion process starts by initializing the DFA's components:

```
init_s = self.initial_state
dfa_transitions = {}
dfa_accept_states = []
queue = [frozenset([init_s])]
visited = set()
dfa_states_map = {frozenset([init_s]): init_s}
```

- `init_s` is the initial state of the NDFA.

- `dfa_transitions` will hold the DFA's transitions.
- `dfa_accept_states` is a list to keep track of the DFA's accept states.
- `queue` is used to manage the sets of NDFA states that need to be processed.
- `visited` keeps track of the sets of NDFA states that have already been processed.
- `dfa_states_map` maps sets of NDFA states to DFA state names.

The function processes each set of NDFA states (representing a single DFA state) in the queue. For each set of states, it computes the DFA transitions for each symbol in the alphabet.

```
while queue:
    current_states = queue.pop(0)
    if current_states in visited:
        continue

    visited.add(current_states)
    dfa_state_name = self._state_set_to_name(current_states)
    dfa_transitions[dfa_state_name] = {}
```

For each symbol, the function calculates the next set of states and updates the DFA transitions. If the next set of states is new, it is added to the queue for processing and mapped to a DFA state name.

```
for symbol in self.alphabet:
    next_states_set = frozenset(
        [next_state for state in current_states for next_state in self.transitions.get(
            state, {}).get(symbol, [])]
    )

    if not next_states_set:
        continue

    next_state_name = self._state_set_to_name(next_states_set)
    dfa_transitions[dfa_state_name][symbol] = [next_state_name]
```

The function also identifies accept states in the DFA. If the set of next states contains any of the NDFA's accept states, the corresponding DFA state is marked as an accept state.

```
if next_states_set.intersection(self.accept_states):
    dfa_accept_states.append(next_state_name)
```

After processing all sets of states, the function constructs and returns a new `FiniteAutomaton` object representing the DFA.

The `_state_set_to_name` helper function seen earlier generates a name for a DFA state based on a set of NDFA states. For single states, it returns the state name directly. For composite states, it generates a name by joining the sorted individual state names.

```
def _state_set_to_name(self, states_set):
    if len(states_set) == 1:
        return next(iter(states_set))
    return ','.join(sorted(states_set))
```

Graphical representation of the finite automaton

The `create_diagram` function generates a graphical representation of a given deterministic finite automaton (DFA) using the Graphviz library. This visual representation helps in understanding the structure and transitions of the DFA.

The function begins by creating a new directed graph using Graphviz's `Digraph` class:

```
dot = Digraph(comment='Finite Automaton')
dot.attr(rankdir='LR')
```

Each state in the DFA is added to the graph as a node. The shape of the node indicates whether it is an accepting state. States are represented by circles, with accepting states distinguished by a double circle (doublecircle).

```
for state in self.states:
    dot.node(state, state, shape='doublecircle' if state in self.accept_states else 'circ
```

A special invisible node is added to the graph to mark the initial state with an incoming arrow:

```
dot.node('', '', shape='none')
dot.edge('', self.initial_state)
```

The function iterates over the transitions defined in the DFA, adding an edge for each transition between states. Each transition is represented by an edge labeled with the input symbol that triggers the transition from one state to another.

```
for state, transitions in self.transitions.items():
    for symbol, next_states in transitions.items():
        for next_state in next_states:
            dot.edge(state, next_state, label=symbol)
```

Finally, the function returns the `dot` object, which encapsulates the entire graph.

Conclusions / Results

This report has explored the fundamental concepts of finite automata, including the distinctions between deterministic finite automata (DFA) and nondeterministic finite automata (NDFA), their conversion, and their classification within the Chomsky hierarchy. Through the implementation of various functions, I've demonstrated how to classify grammars based on the Chomsky hierarchy, convert finite automata to regular grammars, determine the determinism of finite automata, and transform NDFAs into DFAs.

The detailed results of the implementations are comprehensively documented in the Jupyter Notebook titled `lab2.ipynb` in this folder. This interactive notebook includes code snippets, execution outputs, and visual diagrams that illustrate the theoretical concepts discussed in this report.