Determinism in Finite Automata
Conversion from NDFA to DFA
Chomsky Hierarchy

Parfene Daniel

March 4, 2024

1 Introduction

Determinism in finite automata plays a crucial role in understanding the predictability and complexity of language recognition systems. This report explores the concept of determinism, the process of converting non-deterministic finite automata (NDFA) to deterministic finite automata (DFA), and its relevance in the Chomsky hierarchy of formal grammars.

2 Finite Automata and Determinism

Finite automata are mathematical models used to recognize patterns within strings of symbols. A deterministic finite automaton (DFA) is defined by a finite set of states, a finite alphabet of input symbols, a transition function that maps states and input symbols to the next state, an initial state, and a set of accepting (or final) states. In a DFA, for each state and input symbol, there is exactly one possible transition to a next state.

2.1 Non-Deterministic Finite Automata (NDFA)

Non-deterministic finite automata (NDFA) differ from DFA in that for some state and input symbol combinations, there may be multiple possible next states. This non-determinism introduces ambiguity into the automaton's behavior, making it harder to predict the outcome of string recognition.

2.2 Conversion from NDFA to DFA

The process of converting an NDFA to a DFA aims to eliminate non-determinism by determining the set of states reachable from each state in the NDFA for each input symbol. This conversion results in a DFA that retains the same language recognition capabilities as the original NDFA but with deterministic behavior.

3 Implementation Details

3.1 FiniteAutomaton Class

The provided code implements a FiniteAutomaton class capable of converting an NDFA to a DFA. It consists of methods to construct an NDFA, convert it to a DFA, and visualize both automata using the Graphviz library.

```
class FiniteAutomaton:
    def __init__(self):
        self.Q = {'q0', 'q1', 'q2', 'q3'}
        self.sigma = {'a', 'b', 'c'}
        self.delta = {
```

```
            ('q0', 'a'): ['q0', 'q1'],
            ('q2', 'a'): 'q2',
            ('q1', 'b'): 'q2',
            ('q2', 'c'): 'q3',
            ('q3', 'c'): 'q3',
        }
        self.q0 = 'q0'
        self.F = {'q3'}
```

Listing 1: FiniteAutomaton Class

3.2 Conversion Process

The conversion from an NDFA to a DFA involves several steps, including computing the epsilon closure of each state in the NDFA and determining the transitions for each input symbol. This process ensures that the resulting DFA is deterministic and captures all possible state transitions.

```
    def get_dfa_start_state(self):
        dfa_start_state = set()

        for (state, symbol), next_state in self.delta.items():
            if isinstance(next_state, list):
                dfa_start_state.update(next_state)

        return dfa_start_state
```

Listing 2: Getting the Start State of the DFA

3.2.1 NFA to Transition Table

The first step in converting an NDFA to a DFA is to construct a transition table that represents all possible state transitions for each input symbol. This table provides a systematic way to analyze and manipulate the automaton's behavior during the conversion process.

```
    def nfa_to_transition_table(self):
        transition_table = {}

        for state in self.Q:
            for symbol in self.sigma:
                transition_table[(state, symbol)] = set()

        for key, value in self.delta.items():
            state, symbol = key
            if isinstance(value, list):
                for next_state in value:
                    transition_table[(state, symbol)].add(next_state)
            else:
                transition_table[(state, symbol)].add(value)

        return transition_table
```

Listing 3: NFA to Transition Table

3.2.2 NFA to DFA

Once the transition table is constructed, the next step is to determine the set of states reachable from each state in the NDFA for each input symbol. This information is used to construct the equivalent DFA transition table, ensuring deterministic behavior.

```
    def dfa_transition_table(self):
        nfa_transition_table = self.nfa_to_transition_table()
        dfa_start_state = self.get_dfa_start_state()

        dfa_states = [dfa_start_state]
```

```
        unmarked_states = [dfa_start_state]
        dfa_transition = {}

        while unmarked_states:
            current_state = unmarked_states.pop(0)

            for symbol in self.sigma:
                next_state = set()

                for state in current_state:
                    next_state.update(nfa_transition_table.get((state, symbol), set()))

                dfa_transition[(tuple(current_state), symbol)] = tuple(next_state)

                if tuple(next_state) not in dfa_states:
                    dfa_states.append(tuple(next_state))
                    unmarked_states.append(tuple(next_state))

        return dfa_transition
```

Listing 4: NFA to DFA

3.2.3 Determinism

Determinism is a key concept in finite automata theory, ensuring that for each state and input symbol combination, there is exactly one possible next state. Converting an NDFA to a DFA eliminates non-determinism, resulting in a clearer and more predictable automaton.

```
        def is_deterministic(self):
        for state in self.Q:
            transitions = {symbol for (s, symbol), _ in self.delta.items() if s == state}
            if len(transitions) != len(self.sigma):
                return False
        return True
```

Listing 5: Function to find the Determinism

3.2.4 Conversion to Regular Grammar

In addition to automata, grammars are another way to represent languages. The process of converting an NDFA to a DFA can also yield a regular grammar that generates the same language as the automaton. This grammar can then be analyzed and manipulated using formal language theory techniques.

```
        def to_regular_grammar(self):

        start_symbol = self.q0

        VN = self.Q

        VT = self.sigma

        P = []
        for q in self.Q:
            for a in self.sigma:
                if (q, a) in self.delta:
                    to_state = self.delta[(q, a)]
                    if isinstance(to_state, list):
                        for state in to_state:
                            P.append(f"{q} -> {a}{state}")
                    else:
                        P.append(f"{q} -> {a}{to_state}")
            if q in self.F:
                P.append(f"{q} -> epsilon")

        print(f"Start symbol: {start_symbol}")
        print(f"Non-terminals: {', '.join(VN)}")
```

```
        print(f"Terminals: {', '.join(VT)}")
        print("Production rules:")
        for rule in P:
            print(rule)
```

Listing 6: Conversion to Regular Grammar

4 Chomsky Hierarchy and Formal Grammars

The Chomsky hierarchy classifies formal grammars based on their generative power and expressive capability. It consists of four types of grammars, ranging from Type 0 (unrestricted grammars) to Type 3 (regular grammars). This hierarchy provides insight into the complexity of languages and the computational resources required for their recognition.

```
    class Grammar:
    def __init__(self):
        self.VN = {'S', 'A', 'B', 'C'}
        self.VT = {'a', 'b', 'c', 'd'}
        self.P = {
            'S': ['dA'],
            'A': ['d', 'aB'],
            'B': ['bC'],
            'C': ['cA', 'aS']
        }

    def classify_grammar(self):
        is_regular = self.check_regular_grammar()
        is_context_free = self.check_context_free_grammar()
        is_context_sensitive = self.check_context_sensitive_grammar()

        if is_context_sensitive:
            return print("Context-sensitive Grammar")
        elif is_context_free:
            return print("Context-free Grammar")
        elif is_regular:
            return print("Regular Grammar")

    def check_regular_grammar(self):
        # Regular grammar has only productions of the form A -> aB or A -> a
        for non_terminal, productions in self.P.items():
            for production in productions:
                if len(production) > 2:
                    return False
                if len(production) == 2:
                    if production[0] not in self.VT or production[1] not in self.VN:
                        return False
                if len(production) == 1:
                    if production not in self.VT:
                        return False
        return True

    def check_context_free_grammar(self):
        # Context-free grammar has productions of the form A -> w
        for non_terminal, productions in self.P.items():
            for production in productions:
                for symbol in production:
                    if symbol in self.VN and symbol not in self.VT:
                        return False
        return True

    def check_context_sensitive_grammar(self):
        # Context-sensitive grammar has productions of the form uAv -> uwv
        for non_terminal, productions in self.P.items():
            for production in productions:
                if len(production) < 3:
                    return False
```

```
53          return True
```

Listing 7: Grammar Classification

5 Conclusion

Determinism in finite automata is essential for ensuring predictable and efficient language recognition. The process of converting from non-deterministic to deterministic automata enables clearer understanding and more straightforward implementation of language recognition systems. By considering the Chomsky hierarchy, we can contextualize the complexity of languages and the corresponding grammatical structures.