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

Course: Formal Languages & Finite Automata

Author: Gîncu Olivia

Theory:

Determinism in Finite Automata:

Determinism is a crucial concept in the realm of finite automata. In a deterministic finite automaton (DFA), the next state of the machine is uniquely determined by the current state and input symbol. Unlike non-deterministic finite automata (NDFA), where multiple transitions might be possible for a given state and input symbol, DFAs simplify this process by ensuring only one transition is possible, making analysis and implementation much more straightforward.

Conversion from NDFA to DFA:

The conversion from a non-deterministic finite automaton (NFA) to a deterministic finite automaton (DFA) is a fundamental process in automata theory. During this conversion, each state of the DFA corresponds to a set of states from the NFA. By carefully determining transition functions based on these sets of states, the resulting DFA ensures that for any given input symbol and state, there exists a single transition. This conversion greatly aids in understanding and processing the language recognized by the automaton.

Chomsky Hierarchy:

Noam Chomsky's hierarchy offers a systematic classification of formal grammars and the languages they generate. At its core are four types of grammars, each with its own expressive power:

Type 0: Unrestricted grammars, capable of generating any recursively enumerable language.

Type 1: Context-sensitive grammars, which generate context-sensitive languages and allow rewriting based on context.

Type 2: Context-free grammars, responsible for generating context-free languages and featuring production rules with single non-terminal symbols on the left-hand side.

Type 3: Regular grammars, the simplest of all, generating regular languages and characterized by production rules equivalent to those of finite automata.

Objectives:

Understand what an automaton is and what it can be used for.

Continuing the work in the same repository and the same project, the following need to be added:

a. Provide a function in grammar type/class that could classify the grammar based on the Chomsky hierarchy.

b. For this I can use the variant from the previous lab.

According to the 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.

Implementation Description:

1. **Implement conversion of a finite automaton to a regular grammar.**
   The function `fa_to_rg` converts a finite automaton (FA) into a regular grammar (RG). It operates by iterating through each state of the FA and for each state, examining transitions for every symbol in the input alphabet. If a transition exists for a state-symbol pair, it adds corresponding productions to the RG. If a state is final, it also adds an ε-transition. The function returns the resulting regular grammar. The example provided utilizes this function with specific FA components and prints out the resulting regular grammar productions.

```
def fa_to_rg(Q, Sigma, delta, F):
    grammar = {}
    for q in Q:
        grammar[q] = []
        for symbol in Sigma:
            transitions = [state for state, s in delta.items()
if s == (q, symbol)]
            if transitions:
                if len(transitions) == 1:
                    grammar[q].append(symbol + transitions[0])
                else:
                    for t in transitions:
                        grammar[q].append(symbol + t)
        if q in F:
            grammar[q].append('')
    return grammar

Q = {'q0', 'q1', 'q2', 'q3'}
Sigma = {'a', 'b', 'c'}
F = {'q3'}
```

```
delta = {
    ('q0', 'a'): 'q0',
    ('q1', 'b'): 'q2',
    ('q0', 'a'): 'q1',
    ('q2', 'a'): 'q2',
    ('q2', 'b'): 'q3',
    ('q2', 'c'): 'q0'
}

regular_grammar = fa_to_rg(Q, Sigma, delta, F)
for state, productions in regular_grammar.items():
    print(state + " -> " + " | ".join(productions))
```

2. **Determine whether your FA is deterministic or non-deterministic.**
   The function `is_deterministic` quickly determines if a finite automaton (FA) is deterministic. It iterates through each state and symbol, checking if there's more than one transition for any state-symbol pair. If multiple transitions exist, it concludes that the FA is non-deterministic and returns `False`; otherwise, it returns `True`. The provided example employs this function to print whether the FA is deterministic or not.

```
def is_deterministic(Q, Sigma, delta):
    for q in Q:
        for symbol in Sigma:
            transitions = [state for state, s in
delta.items() if s == (q, symbol)]
            if len(transitions) > 1:
                return False
    return True

is_det = is_deterministic(Q, Sigma, delta)
print("The FA is deterministic." if is_det else "The FA
is non-deterministic.")
```

3. **Implementing functionality to convert an NDFA to a DFA (using powerset construction):**
   The provided set of functions, `powerset`, `epsilon_closure`, `move`, and `ndfa_to_dfa`, collaboratively facilitate the conversion process of a non-deterministic finite automaton (NFA) into a deterministic finite automaton (DFA). Firstly, `powerset(states)` generates all possible subsets (powerset) efficiently using bitwise operations. Next, `epsilon_closure(state, delta)` computes the epsilon closure for a given state in the NFA, representing the set of states reachable via epsilon transitions. Subsequently, `move(states, symbol, delta)` calculates the set of states reachable from a given set of states via a specific input symbol in the NFA. Finally, `ndfa_to_dfa(Q, Sigma, delta, q0, F)` orchestrates the conversion process. It initializes the DFA states, delta, and final states, iterating over NFA states to compute their epsilon closures and transitions. The resulting DFA states, transitions, and final states are then determined accordingly. The example provided employs these functions with specific NFA components, showcasing the resulting DFA states, delta, and final states after the conversion process.

```
def powerset(states):

    result = []

    for i in range(1 << len(states)):

        subset = [states[j] for j in range(len(states)) if (i
& (1 << j)) > 0]

        result.append(subset)

    return result


def epsilon_closure(state, delta):

    closure = set()

    stack = list(state)  # Convert the set to a list to
iterate over its elements

    while stack:

        current_state = stack.pop()

        closure.add(current_state)  # Add the current state to
the closure

        for (s, t) in delta:

            if s == current_state and t == '':

                next_state = delta[(s, t)]

                if next_state not in closure:

                    stack.append(next_state)  # Add the next
state to the stack for processing

    return frozenset(closure)  # Convert the closure set to a
frozenset for immutability and hashability



def move(states, symbol, delta):

    result = set()

    for state in states:

        if (state, symbol) in delta:

            result.add(delta[(state, symbol)])

    return frozenset(result)
```

```
def ndfa_to_dfa(Q, Sigma, delta, q0, F):
    dfa_states = []
    dfa_delta = {}
    q0_closure = epsilon_closure({q0}, delta)  # Ensure
q0_closure is initialized properly
    worklist = [q0_closure]
    dfa_states.append(q0_closure)

    while worklist:
        current_state = worklist.pop(0)
        for symbol in Sigma:
            next_state = epsilon_closure(move(current_state,
symbol, delta), delta)
            if next_state not in dfa_states:
                dfa_states.append(next_state)
                worklist.append(next_state)
            dfa_delta[(current_state, symbol)] = next_state

    dfa_final_states = [s for s in dfa_states if any(q in F
for q in s)]
    return dfa_states, dfa_delta, dfa_final_states

Q = {'q0', 'q1', 'q2', 'q3'}
Sigma = {'a', 'b', 'c'}
F = {'q3'}
delta = {
    ('q0', 'a'): 'q0',
    ('q1', 'b'): 'q2',
    ('q0', 'a'): 'q1',
    ('q2', 'a'): 'q2',
    ('q2', 'b'): 'q3',
    ('q2', 'c'): 'q0'
}
q0 = 'q0'


dfa_states, dfa_delta, dfa_final_states = ndfa_to_dfa(Q,
Sigma, delta, q0, F)
print("DFA States:", dfa_states)
print("DFA Delta:", dfa_delta)
print("DFA Final States:", dfa_final_states)
```

Conclusions/Screenshots/Results:

In summary, the provided program efficiently converts a non-deterministic finite automaton (NFA) to a deterministic finite automaton (DFA) through a series of well-designed functions. This conversion enables deterministic handling of language recognition tasks, streamlining analysis, and manipulation of automata.

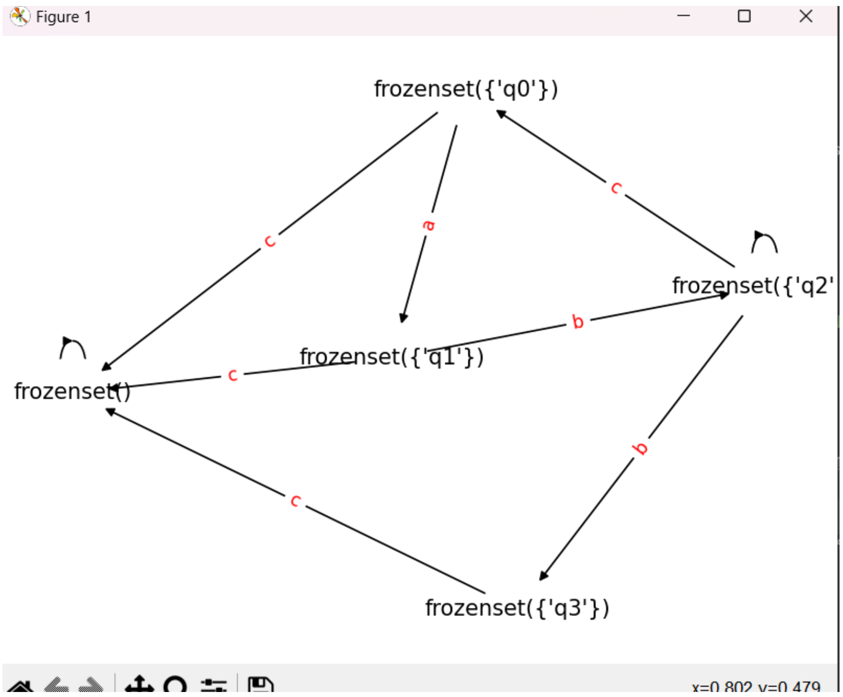
Figure 1
frozenset({'q0'})
frozenset({'q1'})
frozenset({'q2'
frozenset({'q3'})
frozenset()
a
b
b
c
c
c
c
x=0.802 y=0.479