MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

Departamentul Inginerie Software și Automatică

MARTINIUC ARTIOM FAF-222

Report

Laboratory work n.2

of Limbaje Formale și Automate

Checked by:

Dumitru Cretu, *university assistant*

DISA, FCIM, UTM

Chișinău – 2024

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

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.

Variant 13:

Q = {q0,q1,q2,q3},

∑ = {a,b},

F = {q3},

δ(q0,a) = q0,

δ(q0,b) = q1,

δ(q1,a) = q1,

δ(q1,a) = q2,

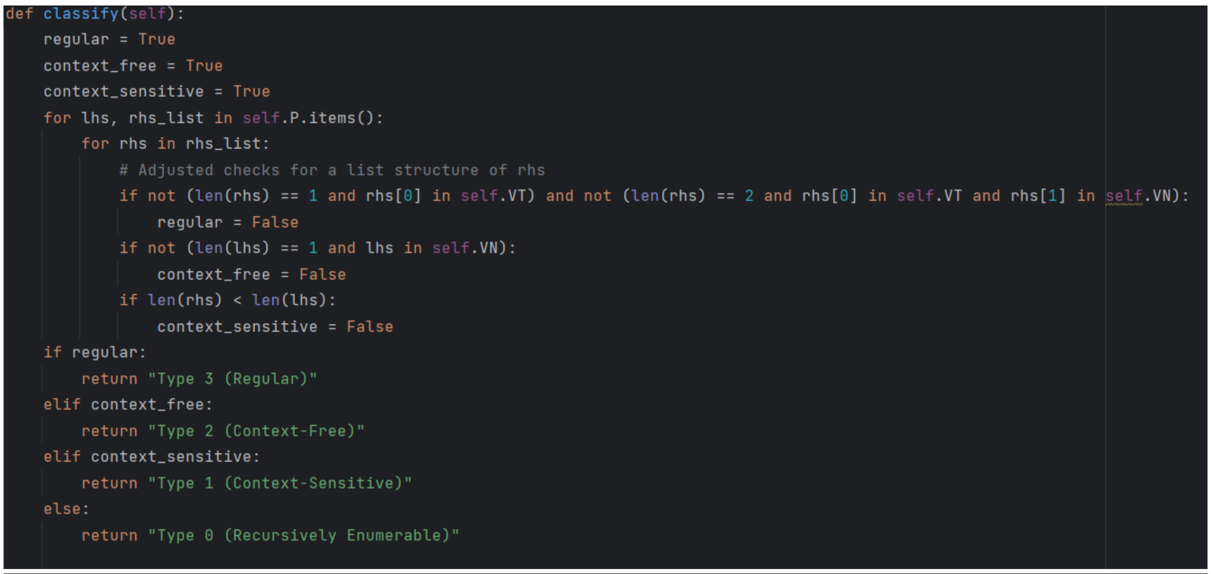
δ(q1,b) = q3,

δ(q2,a) = q2,

δ(q2,b) = q3.

Implementation:

TASK 2.A



```
def classify(self):
    regular = True
    context_free = True
    context_sensitive = True
    for lhs, rhs_list in self.P.items():
        for rhs in rhs_list:
            # Adjusted checks for a list structure of rhs
            if not (len(rhs) == 1 and rhs[0] in self.VT) and not (len(rhs) == 2 and rhs[0] in self.VT and rhs[1] in self.VN):
                regular = False
            if not (len(lhs) == 1 and lhs in self.VN):
                context_free = False
            if len(rhs) < len(lhs):
                context_sensitive = False
    if regular:
        return "Type 3 (Regular)"
    elif context_free:
        return "Type 2 (Context-Free)"
    elif context_sensitive:
        return "Type 1 (Context-Sensitive)"
    else:
        return "Type 0 (Recursively Enumerable)"
```

The method iterates over the productions of the grammar stored in the dictionary self.P. Each production consists of a left-hand side (lhs) symbol and a right-hand side (rhs) list of symbols.

For each production, it checks certain conditions to determine the type of grammar:

A grammar is regular if all its productions are of the form A → a or A → aB, where A is a non-terminal symbol, a is a terminal symbol, and B is a non-terminal symbol. This check ensures that the right-hand side is either a single terminal symbol or a non-terminal followed by a terminal.

A grammar is context-free if all its left-hand sides are single non-terminal symbols.

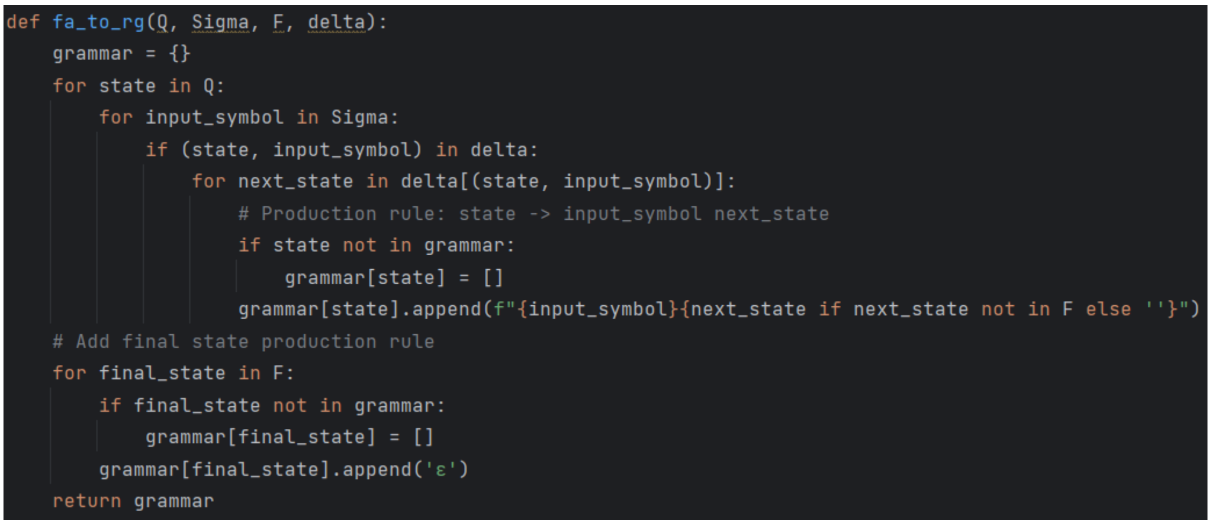
A grammar is context-sensitive if there's at least one production where the length of the right-hand side is greater than or equal to the length of the left-hand side.

If none of the conditions hold it indicates that the grammar is recursively enumerable.

If any production violates these conditions, it sets the corresponding boolean variable to False.

After iterating through all productions, it checks the boolean flags to determine the type of the grammar based on the conditions checked. It returns a string indicating the type of the grammar.

<u>TASK 3.A</u>



```
def fa_to_rg(Q, Sigma, F, delta):
    grammar = {}
    for state in Q:
        for input_symbol in Sigma:
            if (state, input_symbol) in delta:
                for next_state in delta[(state, input_symbol)]:
                    # Production rule: state -> input_symbol next_state
                    if state not in grammar:
                        grammar[state] = []
                    grammar[state].append(f"{input_symbol}{next_state if next_state not in F else ''}")
    # Add final state production rule
    for final_state in F:
        if final_state not in grammar:
            grammar[final_state] = []
        grammar[final_state].append('ε')
    return grammar
```

We have dictionary that will store the productions of the regular grammar.

This method Iterate over each state and input symbol combination.

If there exists a transition for this combination in the FA, we iterate over each possible next state reached from state by input_symbol.

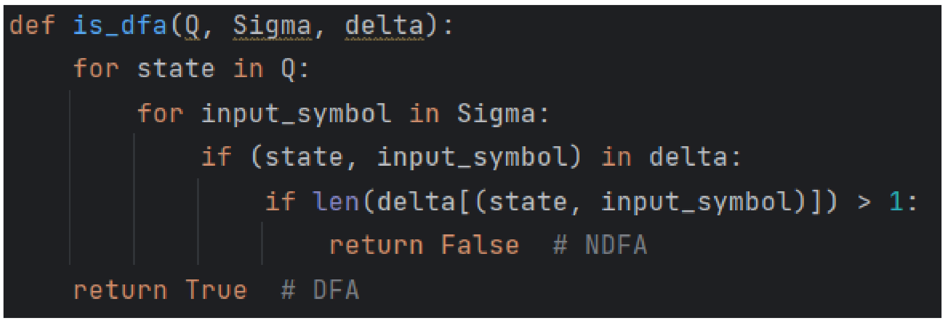
Then, create a production rule and add it to the grammar. If next_state is a final state, the production will not include it.

If state is not already in the grammar dictionary, initialize its value as an empty list to store its production rules.

To add ε-transitions we iterate over each final state in F.

Add an epsilon (ε) production rule from each final state to account for transitions where no input is consumed.

<u>TASK 3.B</u>



```
def is_dfa(Q, Sigma, delta):
    for state in Q:
        for input_symbol in Sigma:
            if (state, input_symbol) in delta:
                if len(delta[(state, input_symbol)]) > 1:
                    return False  # NDFA
    return True  # DFA
```

To check if FA is DFA, method iterates over each state.

For each state, it iterates over each input symbol in the alphabet.

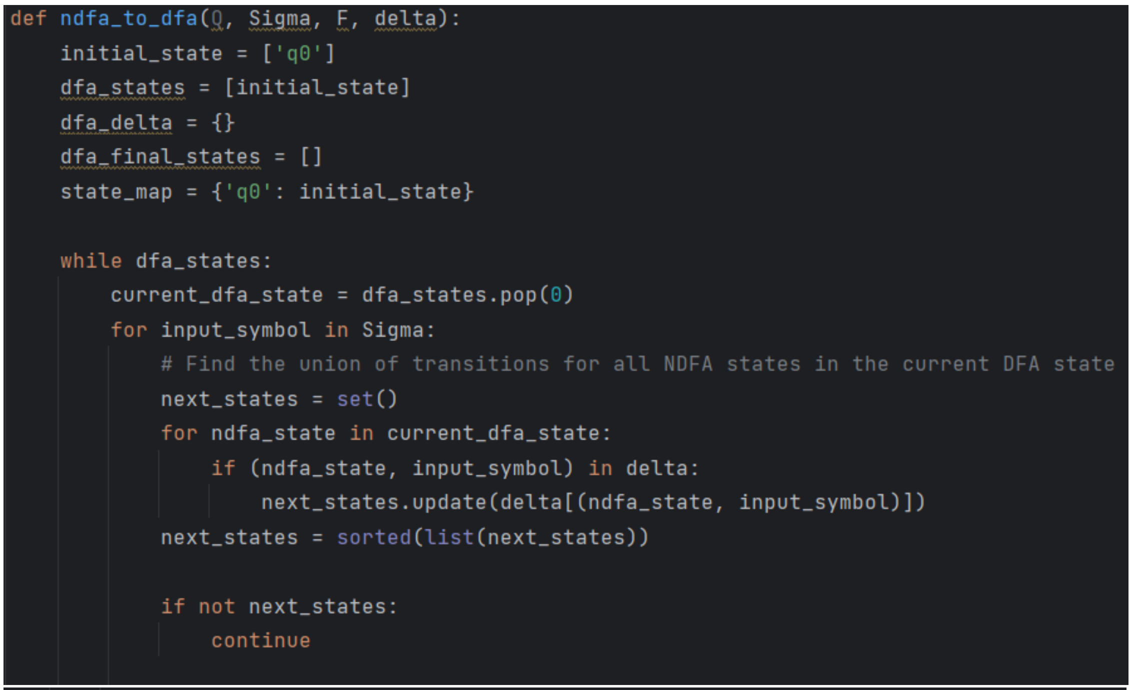
It checks if there exists a transition for the current state and input symbol combination in the transition functions.

If there is a transition, It checks if states resulting from this transition has more than one element.

If it does have more than one element, it means that there are multiple possible next states for a given state and input symbol combination, which violates the determinism property of DFAs. Hence, it returns False, indicating that the automaton is not a DFA.

If the loop completes without finding any non-deterministic transitions, it returns True, indicating that the automaton is a DFA.

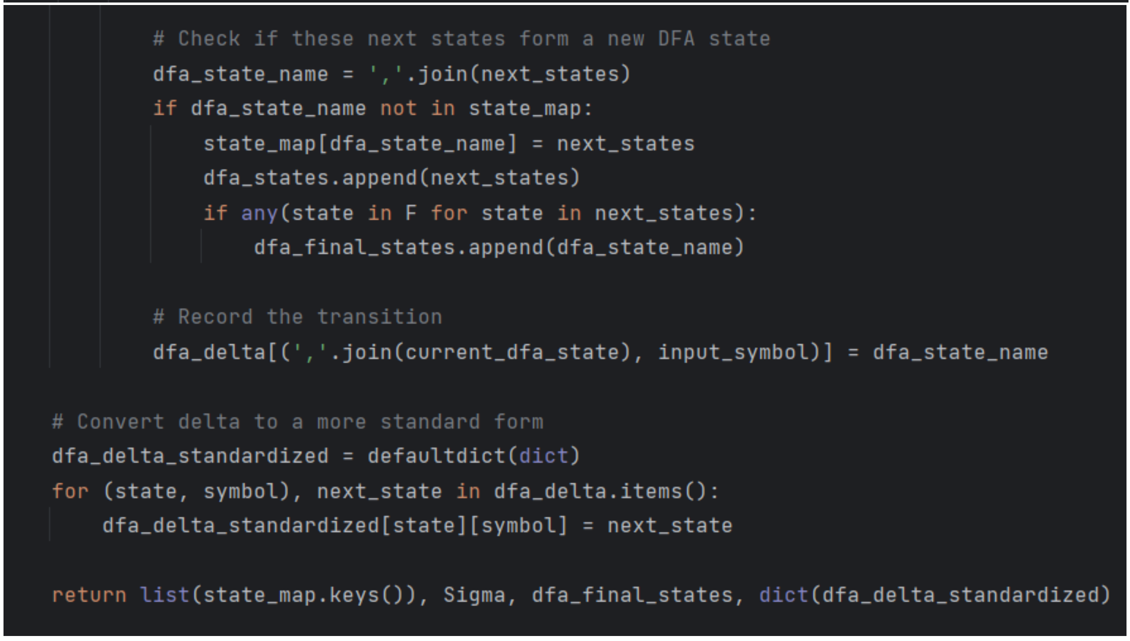
TASK 3.C



```python
def ndfa_to_dfa(Q, Sigma, F, delta):
    initial_state = ['q0']
    dfa_states = [initial_state]
    dfa_delta = {}
    dfa_final_states = []
    state_map = {'q0': initial_state}

    while dfa_states:
        current_dfa_state = dfa_states.pop(0)
        for input_symbol in Sigma:
            # Find the union of transitions for all NDFA states in the current DFA state
            next_states = set()
            for ndfa_state in current_dfa_state:
                if (ndfa_state, input_symbol) in delta:
                    next_states.update(delta[(ndfa_state, input_symbol)])
            next_states = sorted(list(next_states))

            if not next_states:
                continue
```



```python
            # Check if these next states form a new DFA state
            dfa_state_name = ','.join(next_states)
            if dfa_state_name not in state_map:
                state_map[dfa_state_name] = next_states
                dfa_states.append(next_states)
                if any(state in F for state in next_states):
                    dfa_final_states.append(dfa_state_name)

            # Record the transition
            dfa_delta[(','.join(current_dfa_state), input_symbol)] = dfa_state_name

    # Convert delta to a more standard form
    dfa_delta_standardized = defaultdict(dict)
    for (state, symbol), next_state in dfa_delta.items():
        dfa_delta_standardized[state][symbol] = next_state

    return list(state_map.keys()), Sigma, dfa_final_states, dict(dfa_delta_standardized)
```

For each state, this method iterates through each input symbol in the input alphabet.

It computes the set of states that the DFA could transition to on the current input symbol by considering all possible transitions from each state in the current DFA state.

It checks if this set of states forms a new DFA state. If so, it adds it to the DFA states and updates the state map accordingly. If any of the new states contain a final state from the NFA, it adds the new DFA state to the list of final states.

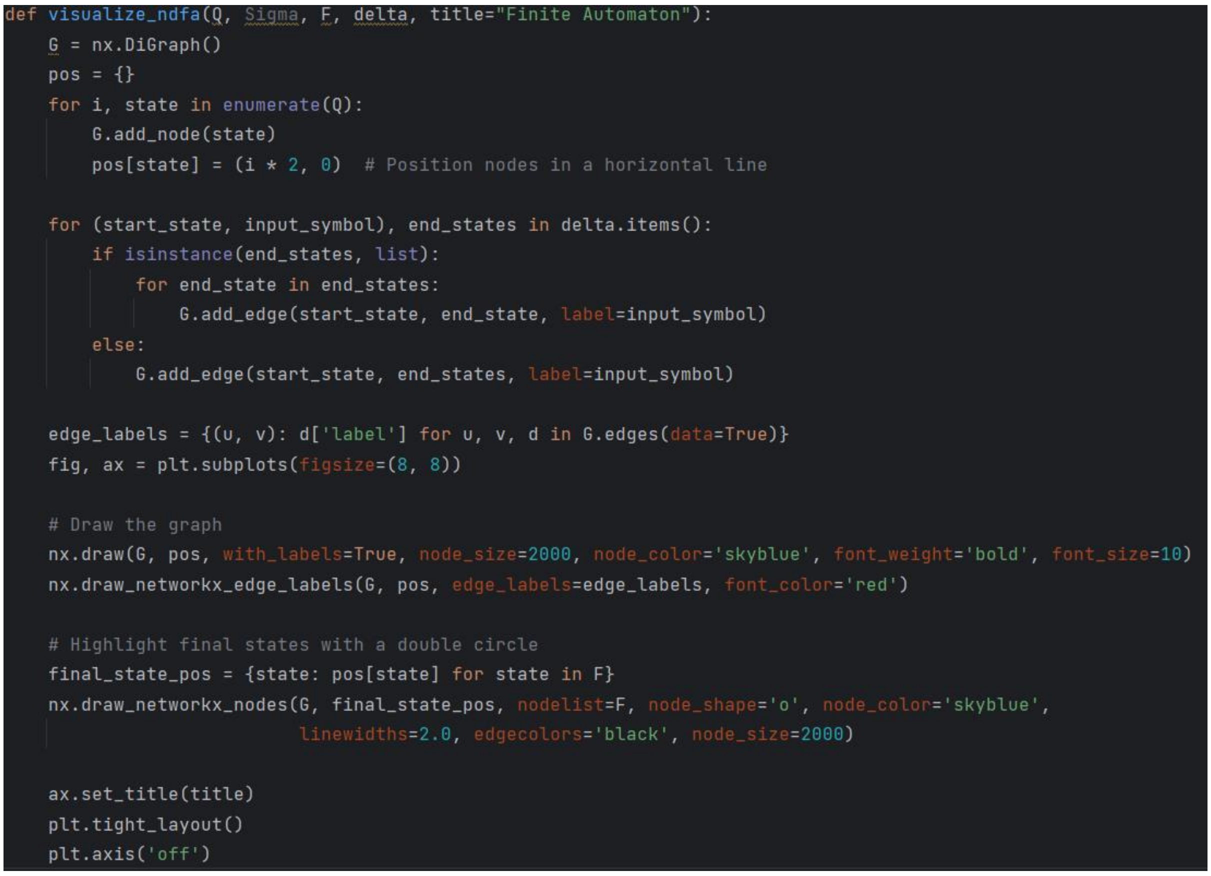
It records the transition from the current DFA state to the new DFA state in the DFA transition function.

Finally, it converts the transition function to a more standard form where each state maps to a dictionary of input symbols and their corresponding next states.

<u>TASK 3.D</u>

I’ve implemented two methods, one to draw an NDFA and one to draw a DFA.

Below we can see the method for NDFA.



```python
def visualize_ndfa(Q, Sigma, F, delta, title="Finite Automaton"):
    G = nx.DiGraph()
    pos = {}
    for i, state in enumerate(Q):
        G.add_node(state)
        pos[state] = (i * 2, 0)  # Position nodes in a horizontal line

    for (start_state, input_symbol), end_states in delta.items():
        if isinstance(end_states, list):
            for end_state in end_states:
                G.add_edge(start_state, end_state, label=input_symbol)
        else:
            G.add_edge(start_state, end_states, label=input_symbol)

    edge_labels = {(u, v): d['label'] for u, v, d in G.edges(data=True)}
    fig, ax = plt.subplots(figsize=(8, 8))

    # Draw the graph
    nx.draw(G, pos, with_labels=True, node_size=2000, node_color='skyblue', font_weight='bold', font_size=10)
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='red')

    # Highlight final states with a double circle
    final_state_pos = {state: pos[state] for state in F}
    nx.draw_networkx_nodes(G, final_state_pos, nodelist=F, node_shape='o', node_color='skyblue',
                           linewidths=2.0, edgecolors='black', node_size=2000)

    ax.set_title(title)
    plt.tight_layout()
    plt.axis('off')
```

It initializes a directed graph using NetworkX to represent the automaton, dictionary to store the positions of each node in the graph.

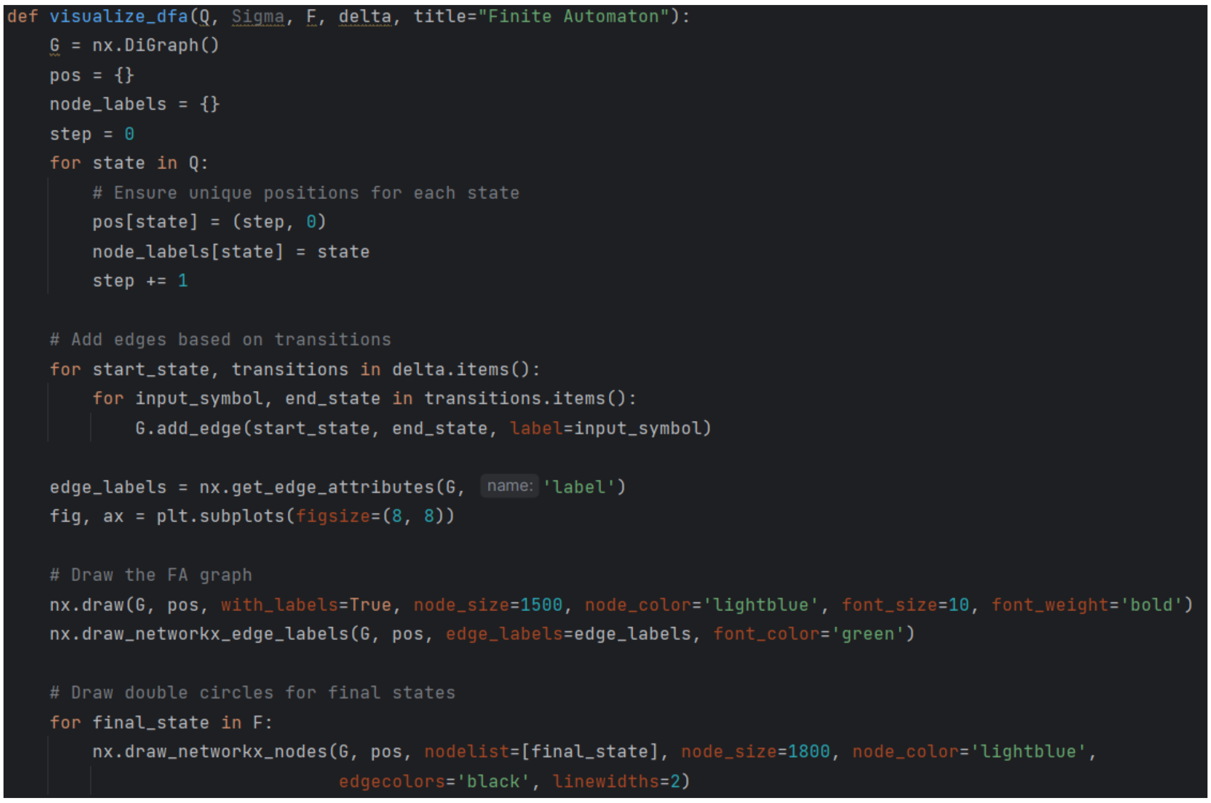
It adds nodes to the graph for each state in states, positioning them in a horizontal line.

It iterates through the transition functions and adds edges to the graph between states, labeled with the corresponding input symbols.

If an end state is a list (indicating non-determinism), it adds multiple edges with the same start and end states but different input symbols.

It draws the graph using NetworkX, highlights final states by drawing them with a double circle.

To visualize DFA, we have to change several blocks. Here is the method.



```python
def visualize_dfa(Q, Sigma, F, delta, title="Finite Automaton"):
    G = nx.DiGraph()
    pos = {}
    node_labels = {}
    step = 0
    for state in Q:
        # Ensure unique positions for each state
        pos[state] = (step, 0)
        node_labels[state] = state
        step += 1

    # Add edges based on transitions
    for start_state, transitions in delta.items():
        for input_symbol, end_state in transitions.items():
            G.add_edge(start_state, end_state, label=input_symbol)

    edge_labels = nx.get_edge_attributes(G, 'label')
    fig, ax = plt.subplots(figsize=(8, 8))

    # Draw the FA graph
    nx.draw(G, pos, with_labels=True, node_size=1500, node_color='lightblue', font_size=10, font_weight='bold')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='green')

    # Draw double circles for final states
    for final_state in F:
        nx.draw_networkx_nodes(G, pos, nodelist=[final_state], node_size=1800, node_color='lightblue',
                               edgecolors='black', linewidths=2)
```

Below will be explained only those parts that are different from the previous method.

For each state in alphabet, it assigns a unique position along the x-axis and stores the positions in pos dictionary. Node labels are set to the corresponding state names.

Edges are added to the graph based on transitions specified in the transition function. Each edge represents a transition from one state to another on a specific input symbol.

The DFA graph is drawn using NetworkX's, nodes are represented as circles, final states are highlighted by drawing double circles around them.