TUTORIAL ASSIGNMENT 13

RA2211003010977
A.NIRMAL SAI KANNA
O1

1.Write a Python program to create an NFA that accepts strings containing only the letter 'a'.

CODE

```
from pythomata import SimpleDFA, SimpleNFA

# Create a simple NFA that accepts strings containing only 'a'
nfa = SimpleNFA(
    states={'q0', 'q1'}, # Set of states
    input_symbols={'a'},  # Input alphabet
    initial_state='q0', # Initial state
    final_states={'q1'},   # Accepting state
    # Transition function
    transitions={
        'q0': {'a': {'q0', 'q1'}}, # From q0 to q0 or q1 on 'a'
        'q1': {},                  # No outgoing transitions from q1
    }
)

# Function to check if a string is accepted by the NFA
def is_accepted_by_nfa(nfa, input_string):
    current_states = {nfa.initial_state}
    for symbol in input_string:
        next_states = set()
        for state in current_states:
            next_states.update(nfa.transitions[state].get(symbol, set()))
        current_states = next_states
    return bool(current_states & nfa.final_states)

# Test the NFA
test_strings = ["a", "aa", "aaa", "ab", "ba"]
for string in test_strings:
    if is_accepted_by_nfa(nfa, string):
        print(f"'{string}' is accepted by the NFA.")
    else:
        print(f"'{string}' is not accepted by the NFA.")
```

SAMPLE OUTPUT:

'a' is accepted by the NFA.
'aa' is accepted by the NFA.
'aaa' is accepted by the NFA.
'ab' is not accepted by the NFA.
'ba' is not accepted by the NFA.

1. Create a Python function to check if a given string is accepted by an NFA that recognizes the pattern "ab|ba" (either "ab" or "ba").

CODE

```
def is_accepted_by_nfa(input_string):
    # Define the NFA transitions
    nfa = {
        0: {'a': [1, 3], 'b': [2, 3]},
        1: {'a': [1], 'b': [2]},
        2: {'a': [3], 'b': [3]},
        3: {'a': [3], 'b': [3]}
    }
    current_states = [0]

    for symbol in input_string:
        next_states = []
        for state in current_states:
            if symbol in nfa[state]:
                next_states.extend(nfa[state][symbol])
        current_states = next_states

    # Check if any of the final states (3) are in the current_states
    return 3 in current_states

# Test the function
input_string = "ab"
if is_accepted_by_nfa(input_string):
    print(f"'{input_string}' is accepted by the NFA")
else:
    print(f"'{input_string}' is not accepted by the NFA")
```

```
input_string = "ba"
if is_accepted_by_nfa(input_string):
    print(f"'{input_string}' is accepted by the NFA")
else:
    print(f"'{input_string}' is not accepted by the NFA")

input_string = "abc"
if is_accepted_by_nfa(input_string):
    print(f"'{input_string}' is accepted by the NFA")
else:
    print(f"'{input_string}' is not accepted by the NFA")
```

SAMPLE OUPUT

```
'ab' is accepted by the NFA
'ba' is accepted by the NFA
'abc' is not accepted by the NFA
```

2. Implement a Python script that converts a simple NFA into a DFA with two states.

CODE

```
def nfa_to_dfa(nfa):

    dfa = {}

    queue = []

    alphabet = set()


    for state, transitions in nfa.items():

        alphabet.update(transitions.keys())


    initial_state = frozenset(['q0'])

    queue.append(initial_state)


    while queue:

        current_state_set = queue.pop(0)
```

```
        dfa_state = ",".join(sorted(current_state_set))

        if dfa_state not in dfa:
            dfa[dfa_state] = {}

        for symbol in alphabet:
            next_state_set = set()
            for nfa_state in current_state_set:
                if symbol in nfa.get(nfa_state, {}):
                    next_state_set.update(nfa[nfa_state][symbol])

            next_state_set = frozenset(next_state_set)
            dfa[dfa_state][symbol] = ",".join(sorted(next_state_set))

            if next_state_set not in dfa:
                queue.append(next_state_set)

    return dfa

# Example NFA
nfa = {
    'q0': {
        '0': ['q0'],
        '1': ['q0', 'q1']
    }
```

```
}

# Convert NFA to DFA
dfa = nfa_to_dfa(nfa)

# Print the resulting DFA
for state, transitions in dfa.items():
    print(f'State: {state}, Transitions: {transitions}')
```

SAMPLE OUTPUT

State: q0, Transitions: {'0': 'q0', '1': 'q0,q1'}

State: q0,q1, Transitions: {'0': 'q0', '1': 'q0,q1'}

3. Write a Python program to construct a DFA that accepts binary strings ending in '01'.

CODE

```
# DFA to accept binary strings ending in '01'

# Define the transition function for the DFA
def transition(current_state, input_symbol):
    if current_state == 0 and input_symbol == '0':
        return 0
    elif current_state == 0 and input_symbol == '1':
        return 1
    elif current_state == 1 and input_symbol == '0':
        return 0
    elif current_state == 1 and input_symbol == '1':
```

```python
        return 2
    else:
        return -1  # Invalid input

# Define the set of accepting states
accepting_states = {2}

# Define the main function to check if a string is accepted by the DFA
def is_accepted(input_string):
    current_state = 0 # Start state
    for symbol in input_string:
        current_state = transition(current_state, symbol)
        if current_state == -1:
            return False  # Invalid input
    return current_state in accepting_states

# Input string to be checked
input_string = input("Enter a binary string: ")

if is_accepted(input_string):
    print("Accepted: The binary string ends in '01'.")
else:
    print("Rejected: The binary string does not end in '01'.")
```

SAMPLE OUTPUT

Enter a binary string: 1001

Accepted: The binary string ends in '01'.

Enter a binary string: 1100

Rejected: The binary string does not end in '01'.

Enter a binary string: 01201

Rejected: The binary string does not end in '01'.

4. Develop a Python function that takes an NFA and returns the set of states that can be reached from a given state on a specific input symbol.

CODE

```
def epsilon_closure(nfa, states):
    epsilon_closure_set = set(states)
    stack = list(states)

    while stack:
        current_state = stack.pop()
        if current_state in nfa and '' in nfa[current_state]:
            for epsilon_transition in nfa[current_state]['']:
                if epsilon_transition not in epsilon_closure_set:
                    epsilon_closure_set.add(epsilon_transition)
                    stack.append(epsilon_transition)

    return epsilon_closure_set
```

```
def move(nfa, states, symbol):
    next_states = set()

    for state in states:
        if state in nfa and symbol in nfa[state]:
            next_states.update(nfa[state][symbol])

    return next_states

def get_states_on_symbol(nfa, start_state, symbol):
    epsilon_closure_start = epsilon_closure(nfa, {start_state})
    next_states = move(nfa, epsilon_closure_start, symbol)
    return epsilon_closure(nfa, next_states)

# Example usage:
nfa = {
    'q0': {'a': {'q0', 'q1'}, '': {'q2'}},
    'q1': {'b': {'q1'}},
    'q2': {'c': {'q3'}},
    'q3': {'d': {'q0', 'q1'}},
}

start_state = 'q0'
input_symbol = 'a'
```

```
result = get_states_on_symbol(nfa, start_state, input_symbol)
print("States reachable from 'q0' on input 'a':", result)
```

SAMPLE OUTPUT

States reachable from 'q0' on input 'a': {'q0', 'q1', 'q2'}

5. Create a Python script to minimize a simple DFA with three states by merging equivalent states.

CODE

```
def minimize_dfa(transition_table, accepting_states):
    def are_equivalent(state1, state2, partition):
        for symbol in alphabet:
            next_state1 = transition_table[state1][symbol]
            next_state2 = transition_table[state2][symbol]
            if partition[next_state1] != partition[next_state2]:
                return False
        return True

    def merge_states(partition, state1, state2):
        for i in range(len(partition)):
            if partition[i] == state2:
                partition[i] = state1

    alphabet = set()
    for state, transitions in transition_table.items():
```

```
    alphabet.update(transitions.keys())

states = list(transition_table.keys())
num_states = len(states)

# Initialize the partition, separating accepting and non-accepting states.
partition = [0 if states[i] in accepting_states else 1 for i in range(num_states)]

# Iterate until the partition no longer changes.
while True:
    new_partition = partition.copy()
    for i in range(num_states):
        for j in range(i + 1, num_states):
            if not are_equivalent(states[i], states[j], partition):
                merge_states(new_partition, i, j)
    if new_partition == partition:
        break
    partition = new_partition

# Build the minimized DFA
minimized_transition_table = {}
minimized_accepting_states = set()

for i, state in enumerate(states):
    if partition[i] not in minimized_transition_table:
```

```
            minimized_transition_table[partition[i]] = {}
        for symbol in alphabet:
            next_state = transition_table[state][symbol]
            minimized_transition_table[partition[i]][symbol] = partition[states.index(next_state)]

        if state in accepting_states:
            minimized_accepting_states.add(partition[i])

    return minimized_transition_table, minimized_accepting_states

# Example usage
transition_table = {
    'A': {'0': 'B', '1': 'C'},
    'B': {'0': 'A', '1': 'B'},
    'C': {'0': 'B', '1': 'A'}
}
accepting_states = {'C'}

minimized_transition_table, minimized_accepting_states = minimize_dfa(transition_table,
accepting_states)

print("Minimized Transition Table:")
for state, transitions in minimized_transition_table.items():
    print(state, transitions)

print("Minimized Accepting States:", minimized_accepting_states)
```

SAMLE OUTPUT

```
Minimized Transition Table:
0 {'0': 0, '1': 1}
1 {'0': 0, '1': 0}

Minimized Accepting States: {0}
```

6. Implement a Python function that checks if a given string is accepted by a DFA that recognizes the pattern "ab*c".

CODE

```python
def is_accepted_by_dfa(input_string):
    # Define the DFA transitions as a dictionary of dictionaries
    # The keys are the current states, and the values are dictionaries of transitions to next states.
    transitions = {
        0: {'a': 1, 'b': 0},
        1: {'b': 2},
        2: {'b': 2},
    }

    # Define the set of accepting states
    accepting_states = {2}

    # Initialize the current state as the start state (0)
    current_state = 0
```

```
    # Process the input string character by character
    for char in input_string:
        if char in transitions[current_state]:
            current_state = transitions[current_state][char]
        else:
            return False  # Invalid input character

    # Check if the final state is an accepting state
    return current_state in accepting_states

# Test cases
print(is_accepted_by_dfa("abbbbc")) # True
print(is_accepted_by_dfa("ababab")) # False
print(is_accepted_by_dfa("ac"))      # False
print(is_accepted_by_dfa("abc"))     # False
```

SAMPLE OUTPUT

```
print(is_accepted_by_dfa("abbbbc")) # True
print(is_accepted_by_dfa("ababab")) # False
print(is_accepted_by_dfa("ac"))      # False
print(is_accepted_by_dfa("abc"))     # False
```

7. Write a Python program to create an NFA that accepts strings with an odd number of '1's.

CODE

```
from pythomata import SimpleDFA
from pythomata.alphabets import SimpleAlphabet

# Define the alphabet (0 and 1)
alphabet = SimpleAlphabet({'0', '1'})

# Create a simple NFA
nfa = SimpleDFA(alphabet)

# Define the states
q0 = nfa.add_state(initial=True) # Initial state
q1 = nfa.add_state()
q2 = nfa.add_state()
q3 = nfa.add_state()

# Add transitions for '0' and '1'
nfa.add_transition(q0, '0', q0)
nfa.add_transition(q0, '1', q1)
nfa.add_transition(q1, '0', q1)
nfa.add_transition(q1, '1', q2)
nfa.add_transition(q2, '0', q2)
nfa.add_transition(q2, '1', q3)
nfa.add_transition(q3, '0', q3)
nfa.add_transition(q3, '1', q0)  # Back to the initial state
```

```
# Set accepting states
nfa.set_accepting_states({q0})


# Define a function to check if a string is accepted
def is_odd_ones(string):
    result = nfa.accepts_input(string)
    return result


# Test the NFA
input_string = input("Enter a binary string: ")
if is_odd_ones(input_string):
    print("Accepted: The string has an odd number of '1's.")
else:
    print("Rejected: The string does not have an odd number of '1's.")
```

SAMPLE OUTPUT

8. Develop a Python function that converts a simple regular expression like "a(b|c)*" into an equivalent NFA.

CODE

```
class NFAState:
    def __init_(self):
        self.transitions = {}
        self.is_accepting = False
```

```
def regex_to_nfa(regex):
    def construct_nfa(regex):
        stack = []
        for char in regex:
            if char == 'a':
                state_a = NFAState()
                state_b = NFAState()
                state_a.transitions['a'] = [state_b]
                stack.extend([state_a, state_b])
            elif char == 'b' or char == 'c':
                state = NFAState()
                state.is_accepting = True
                stack.append(state)
            elif char == '|':
                state1 = stack.pop()
                state2 = stack.pop()
                new_state = NFAState()
                new_state.transitions['ε'] = [state1, state2]
                stack.extend([new_state, state1])
            elif char == '*':
                state1 = stack.pop()
                new_state = NFAState()
                new_state.transitions['ε'] = [state1]
                stack.extend([new_state, state1])
```

```
        return stack[0]

    start_state = construct_nfa(regex)
    return start_state

def print_nfa(nfa, prefix=""):
    print(prefix + ("(Accepting)" if nfa.is_accepting else ""))
    for symbol, states in nfa.transitions.items():
        for state in states:
            print(prefix + f"--{symbol}-->", end="")
            print_nfa(state, prefix + "  ")

regex = "a(b|c)*"
nfa_start_state = regex_to_nfa(regex)
print("NFA for the regular expression:", regex)
print_nfa(nfa_start_state)
```

SAMPLE OUTPUT

```
NFA for the regular expression: a(b|c)*
(Accepting)
--a-->(Accepting)
--ε-->(Accepting)
--ε-->(Accepting)
```