BABA FARID
GROUP OF INSTITUTIONS
Bathinda, Punjab (India)

Formal Language and Automata Theory

Name : Amritpal Singh

Uid : 2272169

Course : Btech Cse

Semester : 5th 'B's

Uni roll no. : 2272169

Submitted to: Er. Sumeet Bharti

Q1. Compare DFA and NFA equivalence. Convert an NFA to an equivalent DFA and discuss the structural and state transition differences.

Ans: Comparison of DFA and NFA

1. Deterministic Finite Automaton (DFA):

- **Deterministic**: For each state and input symbol, the automaton has exactly one possible state to transition to.
- **Transition Function**: Given a state and an input symbol, the transition is a single, deterministic state.
- **State Transitions**: Each state in the DFA has a unique transition for every symbol in the alphabet.
- **Number of States**: Typically larger because every possible combination of input and states is explicitly defined.
- **Execution**: Easier to implement as there's no ambiguity—just one path for a given input sequence.

2. Non-deterministic Finite Automaton (NFA):

- **Non-deterministic**: For each state and input symbol, the automaton can transition to multiple possible states or even none (including ε-transitions).
- **Transition Function**: Given a state and an input symbol, the transition can be to multiple states or can involve ε (empty string) transitions.
- **State Transitions**: The NFA can have multiple transitions for the same state and input symbol.
- **Number of States**: Typically smaller because it allows multiple transitions or paths for a given input sequence.
- **Execution**: Harder to implement directly but more flexible, as multiple paths can be pursued simultaneously.

3. Key Equivalence:

- **Language Acceptance**: DFAs and NFAs are equivalent in terms of the languages they recognize; for every NFA, there exists a DFA that recognizes the same language.

- **Efficiency**: An NFA might be more efficient in representation but less efficient in execution, while the DFA is more efficient in execution but may require exponentially more states.

4. Converting an NFA to an Equivalent DFA

To convert an NFA to an equivalent DFA, we use a process called the **subset construction (or powerset construction)**.

5. Steps for Conversion:

5.1. Start State: The start state of the DFA corresponds to the ε-closure (set of states reachable from the NFA's start state on ε-transitions) of the NFA's start state.
5.2. Transitions: For each new DFA state, compute the set of states the NFA could transition to on each input symbol. Each new set of NFA states becomes a DFA state.
5.3. Final States: A DFA state is considered a final state if it contains any of the NFA's final states.
5.4. New States: Continue the process until all possible sets of NFA states have been considered as DFA states.

Example:

Consider the following simple NFA:

States: {q0, q1, q2}

Alphabet: {a, b}

Start State: q0

Final States: {q2}

Transition Function:

δ(q0, a) = {q0, q1}

δ(q1, b) = {q2}

δ(q2, b) = {q1}

Step 1: Determine the DFA Start State

- The start state of the DFA will be the ε-closure of q0, which is {q0} because there are no ε-transitions in this NFA.

Step 2: Compute the Transitions for Each Input

- From {q0} on input a, the NFA can move to {q0, q1}.
- From {q0} on input b, there is no transition, so it moves to the empty set ∅.
- From {q0, q1} on input a, the NFA can move to {q0, q1} (because of the self-loop on q0 and transition from q0 to q1).

- From {q0, q1} on input b, the NFA can move to {q1, q2} (because of the transitions from q1 to q2).

Step 3: Repeat for All Sets

- Continue computing new DFA states as subsets of the NFA states for all inputs.

DFA Transitions (Example results):

DFA States: { {q0}, {q0, q1}, {q1, q2}, Ø }

Start State: {q0}

Final States: Any state containing `q2`, i.e., {q1, q2}

Transition Function:

δ({q0}, a) = {q0, q1}

δ({q0}, b) = Ø

δ({q0, q1}, a) = {q0, q1}

δ({q0, q1}, b) = {q1, q2}

δ({q1, q2}, a) = Ø

δ({q1, q2}, b) = {q1, q2}

6. **Structural and State Transition Differences:**

- **State Explosion**: The DFA might have more states than the original NFA. In the worst case, the number of DFA states can be up to 2n2^n2n, where n is the number of NFA states.
- **No Ambiguity in DFA**: DFA is deterministic and doesn't have ε-transitions or multiple transitions for a single symbol. The NFA can take multiple paths and may use ε-transitions.
- **Efficiency**: The DFA trades off compactness (from the NFA) for a clear, deterministic execution. The NFA's smaller size may represent a complex behavior, while the DFA expands this into a deterministic state transition system.

Q.2. Illustrate the relationship between DFA and regular expressions. Convert a regular expression into an equivalent DFA with a step-by-step example.

Ans: Relationship Between DFA and Regular Expressions

Both **Deterministic Finite Automata (DFA)** and **regular expressions** represent regular languages, making them **equivalent in expressive power**. Here's a brief summary of their relationship:

- **Regular expressions** describe regular languages in a declarative way by specifying patterns that strings must follow.
- **DFA** is a computational model that can recognize whether a string belongs to a regular language by transitioning between states based on the input symbols.
- Any language that can be defined by a regular expression can also be accepted by a DFA, and for any DFA, a corresponding regular expression can be constructed.

Conversion Process: Regular Expression to DFA

To convert a regular expression into a DFA, the typical process involves the following steps:

1. Convert the regular expression into a **Non-deterministic Finite Automaton (NFA)**, often using Thompson's construction.
2. Convert the NFA into a **Deterministic Finite Automaton (DFA)** using the **subset construction** method.

Step-by-Step Example: Converting Regular Expression to DFA

Let's convert the regular expression (a|b)∗ab(a|b)^*ab(a|b)∗ab into an equivalent DFA. This regular expression describes the language of strings that consist of any combination of 'a' and 'b', followed by the substring "ab".

1. Create the NFA from the Regular Expression

We first construct an NFA for the given regular expression (a|b)∗ab(a|b)^*ab(a|b)∗ab.

Step-by-step NFA construction using Thompson's construction:

- **(a|b)**: This part represents either 'a' or 'b'. We create an NFA with two branches—one for 'a' and one for 'b'.
- **(a|b)^***: The Kleene star means zero or more repetitions of 'a' or 'b'. We add ε-transitions (empty string transitions) to allow for repetition.
- **ab**: After the repetitions, we expect the substring "ab" at the end. This is represented by two sequential transitions, one for 'a' and one for 'b'.

NFA Construction Diagram:

The NFA will have the following structure:

1. An initial state that can transition to either 'a' or 'b' via ε-transitions (representing (a|b)).
2. After consuming 'a' or 'b', it can transition back to the initial state via ε-transitions, representing the Kleene star (a|b)^*.

3. After repeating (or not repeating) 'a' or 'b', it transitions to a state that consumes the literal 'a', followed by a state that consumes the literal 'b' (representing "ab").
4. The state after 'b' is the final accepting state.

2. Convert the NFA into a DFA

We now use the subset construction method to convert the NFA to a DFA. The DFA needs to account for all possible subsets of NFA states at any given time.

Step-by-step Subset Construction:

1. **Start State**: The DFA starts in the ε-closure of the NFA's start state. This means all states reachable from the initial NFA state via ε-transitions are considered.
2. **Transitions**: For each state (or set of states in the DFA) and each input symbol ('a' or 'b'), compute where the NFA could go. Create new DFA states based on these computed sets.
3. **Accepting States**: Any DFA state that contains an NFA accepting state is an accepting state of the DFA.

Let's walk through the key transitions:

- **Initial State**: The ε-closure of the start state (let's call it {q0}) is just {q0} because there are no ε-transitions from the start state.
- From {q0}, on input 'a', the NFA could go to {q0, q1} (since the Kleene star allows repetition). Similarly, on 'b', it could go to {q0, q2}.
- Continuing this process, we enumerate all reachable DFA states.

3. Final DFA Structure

After completing the subset construction, we get the following DFA:

- States: Represent all possible sets of NFA states.
- Transitions: Deterministic transitions for each input symbol.
- Start state: Corresponds to the initial state of the NFA.
- Accepting state: Any state that includes the final accepting state of the NFA.

The DFA will systematically track combinations of the NFA's states and only have one transition for each input symbol at any point, as per the definition of a DFA.

Example DFA (Simplified View):

- **States**: S0, S1, S2, S3, ...
  - S0: Start state
  - S3: Accepting state
- **Alphabet**: {a, b}
- **Transitions**:
  - S0 → S1 on input 'a'

  - S0 → S2 on input 'b'
  - S1 → S3 on input 'b'
  - S2 → S3 on input 'a'
  - (and so on for every possible transition)

Conclusion

1. **DFA and Regular Expressions** both define regular languages, and they can be converted into one another.
2. **Conversion from Regular Expression to DFA** involves:
   - First converting the regular expression to an NFA (using Thompson's construction).
   - Then converting the NFA to a DFA using the subset construction method.

This systematic conversion ensures that any regular expression can be recognized by a DFA.

Q.3. Design DFA for strings that every a is immediately preceded and followed by b.

To design a Deterministic Finite Automaton (DFA) for strings where every occurrence of 'a' is immediately preceded and followed by 'b', let's break down the requirements:

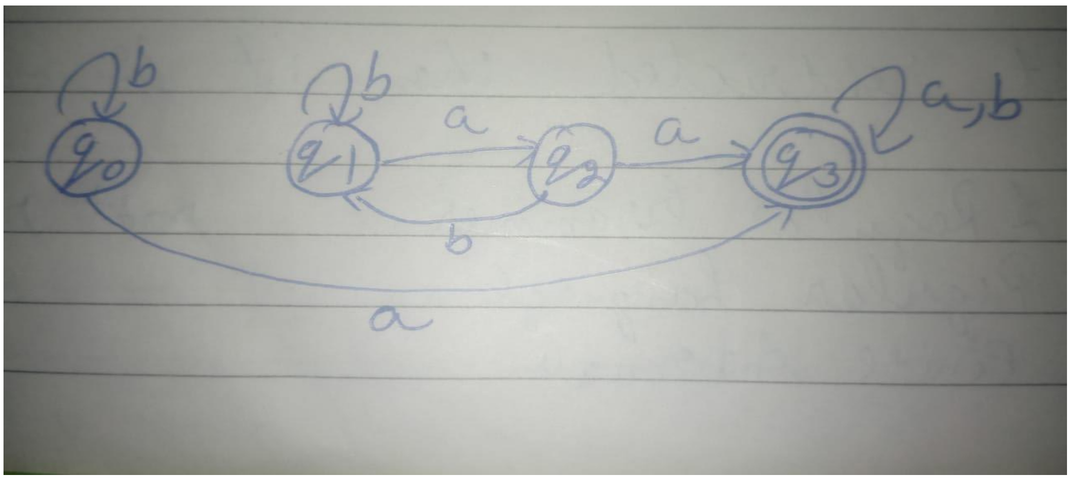
Problem Analysis:

- The character 'a' can only appear if it is surrounded by 'b' on both sides.
  - That is, for every 'a', the pattern should look like bab.
- Any number of 'b's can appear without any restrictions.
- The string can also have no 'a's, but if 'a' appears, it must follow the rule.

DFA Design:

We will define four states:

1. **q0 (Start state):** Represents the state where we haven't yet encountered a violation of the pattern.
2. **q1 (Expecting 'a' state):** After encountering 'b', we are in a state expecting an 'a' next.
3. **q2 (Expecting 'b' after 'a' state):** After encountering 'a', we are expecting a 'b' to follow it.
4. **q3 (Dead state):** Represents an invalid state when the pattern is violated.



State Transitions:

- **q0 (Start state)**:
  - On input b: Stay in q0 (because 'b' alone is fine).
  - On input a: Go to q3 (because 'a' can't start the string without being preceded by 'b').
- **q1 (Expecting 'a' state)**:
  - On input b: Stay in q1 (we are still okay, waiting for 'a').
  - On input a: Go to q2 (this 'a' is preceded by 'b', so we move to expect the next 'b').
- **q2 (Expecting 'b' after 'a' state)**:
  - On input b: Go to q1 (because the 'b' satisfies the requirement of following the 'a', now we're ready to look for another 'a').
  - On input a: Go to q3 (because 'a' must be preceded by 'b' and followed by 'b', but we got 'a' again).
- **q3 (Dead state)**:
  - On input a or b: Stay in q3 (because once we violate the rule, we remain in an invalid state).

Final State:

- The final states where the string is valid are **q0** and **q1**.

Transition Table:

Current State	**Input 'a'**	**Input 'b'**
q0	q3	q0
q1	q2	q1
q2	q3	q1

Current State	Input 'a'	Input 'b'
q3 (Dead)	q3	q3

DFA Diagram Explanation:

1. **q0 (start):** No 'a' has been encountered yet, or 'b's are being processed.
2. **q1:** We are ready to encounter an 'a' (after a 'b').
3. **q2:** We have just encountered an 'a' and are expecting the next character to be 'b'.
4. **q3:** The DFA enters this state if the pattern is violated (i.e., 'a' is not surrounded by 'b').

Thus, the DFA ensures that every occurrence of 'a' is surrounded by 'b' on both sides.